

## 10 Matrix Computations

A great fraction of scientific computations is based on matrix mathematics. This is both a necessity and a convenience. It's a necessity in that a great deal of applied math and applied science/engineering reduces to matrix mathematics. It's a convenience in that since we can do matrix operations with reasonable understanding and ease, we make the effort to reduce the problems in science and engineering to matrix operations because then we have some hope of figuring out how to solve them.

As with ordinary operations, we ignore addition and concentrate on multiplication.

If we have two matrices of  $m$  rows and  $n$  columns, then adding them takes  $m \cdot n$  adds.

If we have matrices  $A$  of  $m$  rows and  $l$  columns and  $B$  of  $l$  columns and  $n$  rows, then the product  $A \cdot B$  has  $m$  rows and  $n$  columns and the naive approach takes  $m \cdot l \cdot n$  individual multiplications.

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$

```
for(rowsub = 0; rowsub < m; rowsub++)
{
  for(colsub = 0; colsub < n; colsub++)
  {
    c[rowsub][colsub] = 0.0;
    for(innersub = 0; innersub < l; innersub++)
    {
      c[rowsub][colsub] += a[rowsub][innersub] * b[innersub][colsub];
    }
  }
}
```

If we simplify things to the point of dealing with square matrices only so that we can reduce everything to just one variable  $n$  for  $n \times n$  matrices, then this is order  $n^3$  multiplications (and additions, but of course we ignore these as being of much lower order).

Ignoring communication time, we could parallelize in several different ways some of which are actually cost optimal.

With  $n$  processors, we could do  $n$  inner loops in parallel. This would require  $n$  sets of inner loops to be done, and each takes  $n$  ops so the time complexity is  $n^2$ . The cost is therefore  $n$  times  $n^2$  and is thus still  $n^3$ .

With  $n^2$  processors, we could do all  $n^2$  inner loops in parallel. Each takes  $n$  time steps, so we are still at cost  $n^3$ .

According to Moldovan (1993),  $\log n$  is the lower bound in time. We can get this by using  $n^3$  processors, one for each inner loop subscript. We compute all the products at once, and then combine them with a binary tree for the summation, and the binary tree takes  $\log n$  steps.

However,  $n^3 \cdot \log n$  is larger than  $n^3$  and is therefore not cost optimal.

Now, reality is that communication actually does matter.

## 10.1 Multiprocessor Matrix Computations

The basic approach is to partition the matrix into submatrices.

Figure 10.4 and Figure 10.5.

### 10.1.1 Complete Parallelization

If we parallelize completely, with  $n^2$  processors each doing one element of the matrix, then

#### **Communication:**

Each processor gathers in  $n - 1$  row elts and  $n - 1$  col elts.

Thus  $n^2 \cdot (2n - 2) = O(n^3)$  messages.

This is bad.

#### **Computation:**

Each processor does  $n$  mults, and all processors act in parallel.

Thus  $O(n)$  computation time.

This is good.

If we had  $n^3$  processors, we could do  $n$  mults with a binary tree in  $\lg n$  time.

Thus  $n^3$  processors gives  $O(\lg n)$  compute time.

### 10.1.2 Parallelization Into Submatrices

Let  $m = n/s$ .

If we parallelize into  $s^2$  submatrices of size  $m \times m$ , then

**Communication:**

Each processor gathers in  $(s - 1) \cdot m^2$  row elements and the same number of column elements.

Thus  $2s - 2$  messages, each of  $m^2$  elements.

This is good or bad, depending on  $s$  and  $m$ , and clearly part of getting high performance is load balancing the number of messages with their size.

**Computation:**

Each processor computes  $m^2$  elements of the product matrix.

Each element of the product matrix takes the usual  $n$  mults.

All processors act in parallel.

Thus  $O(nm^2) = O(n^3/s^2)$  computation time.

This is good if  $s$  is large—we have  $s^2$  processors and we make full use of them in the asymptotics of the computation (if not the communication).

### 10.1.3 Recursive Divide and Conquer

The previous idea suggests that an optimal parallel implementation might be that of recursively breaking up the matrix into four submatrices. Keep doing this as long as we have processors. That is, with 64 processors, break up the matrix three times into blocks of four...

## 10.2 Mesh Based Matrix Multiply

Consider the dot product that produces  $c_{i,j}$  in a matrix.

$$c_{i,j} = a_{i,0}b_{0,j} + a_{i,1}b_{1,j}\dots + a_{i,i+j}b_{i+j,j}\dots + a_{i,n}b_{n,j}$$

Let's look at the  $a_{i,i+j}b_{i+j,j}$  term.

If we have  $n^2$  processors arranged in a mesh, and we assign each  $c_{i,j}$  to a processor, then we can do the multiply as follows:

1. Shift the  $i$ -th row left (circularly)  $j$  places.
2. Shift the  $j$ -th row up (circularly)  $i$  places.
3. This brings  $a_{i,i+j}$  in row  $i$  to the processor for  $c_{i,j}$ .
4. And it brings  $b_{i,i+j}$  in col  $j$  to the processor for  $c_{i,j}$ .
5. We multiply and accumulate
6. And in subsequent steps we
  - (a) shift the row left one place
  - (b) shift the col up one place
  - (c) multiply the elements and add in to the running sum.

**Communication:** Don't do this one element at a time. Do it with  $s^2$  submatrices of size  $m \times m$ .

At most  $4(s-1)$  comms steps with  $m^2$  elements each.

**Computation:**

Same as before.  $m^3$  mults for each submatrix, and  $s$  such, so  $O(nm^2)$  mults.

## 11 Solutions of Linear Equations

We have as a problem the solution of the linear matrix equation

$$Ax = b$$

There are *dense matrices*  $A$  and there are *sparse matrices*  $A$ .

Solution methods differ.

Many sparse matrices have a structure that can be exploited in addition to the sparsity of coefficients.

There are *direct methods* for solving the equation and there are *iterative methods*.

### 11.1 Gaussian Elimination

A direct method usually applied to dense matrices.

Can be parallelized.

But the numerical and asymptotic properties often aren't very good.

### 11.2 LU Decomposition

You've all done Gauss, and you realize that what you're doing is converting

$$Ax = b$$

into

$$A^{-1}Ax = Ix = A^{-1}b$$

Except that in practice we usually don't go this far. In practice we compute

$$MAx = A'x = Mb$$

where  $A'$  is upper triangular, and then we can solve by *back-substitution* for the answer.

So let's go after this directly. Can we factorize  $A$  into  $A = LU$  where  $L$  is lower triangular and  $U$  is upper triangular? If we can do this, then we can solve

$$Ly = b$$

for  $y$  and then solve

$$Ux = y$$

to get the  $x$  vector of answers. And since both  $L$  and  $U$  are triangular, we can solve the first equation by forward substitution and the second by back substitution.

So let's set up the *LU decomposition*. For illustration, we'll deal with a  $5 \times 5$  matrix. What we want is the following.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ \alpha_{21} & 1 & 0 & 0 & 0 \\ \alpha_{31} & \alpha_{32} & 1 & 0 & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & 1 & 0 \\ \alpha_{51} & \alpha_{52} & \alpha_{53} & \alpha_{54} & 1 \end{pmatrix} \cdot \begin{pmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} & \beta_{15} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} & \beta_{25} \\ 0 & 0 & \beta_{33} & \beta_{34} & \beta_{35} \\ 0 & 0 & 0 & \beta_{44} & \beta_{45} \\ 0 & 0 & 0 & 0 & \beta_{55} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{pmatrix}$$

Do the math. For every column  $j = 1, \dots, n$ , solve both steps of

$$\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik} \beta_{kj} \quad (*)$$

$$\alpha_{ij} = \frac{1}{\beta_{jj}} \cdot \left( a_{ij} - \sum_{k=1}^{j-1} \alpha_{ik} \beta_{kj} \right) \quad (**)$$

Note that this is in effect a column-by-column Gaussian elimination.

Two other wrinkles that improve the numerical stability.

First, we do *partial pivoting*, that is, we choose to pivot on the largest element in the column rather than the element that happens to be on the diagonal. We can do this either by actually exchanging rows or by keeping track of a permutation matrix and doing an implicit exchange.

Second, dividing is generally bad, so we don't do the division. Rather than dividing by the diagonal element and subtracting off multiples, Instead we implicitly multiply up the rows below the diagonal and then subtract off. NOTE: If the original matrix is banded, then the  $L$  and  $U$  matrices are also banded, although usually with more fill than the original.

### 11.3 (Jacobi) Iterative Methods

(Assume that  $A$  is a square matrix.) We have a set of equations: for  $i = 1, \dots, n$ , we have

$$a_{i1}x_1 + a_{i2}x_2 + \dots + a_{in}x_n = b_i$$

Solve this equation for  $x_i$ .

$$x_i = \frac{\left(b_i - \sum_{j \neq i} a_{ij}x_j\right)}{a_{ii}}$$

The basic iterative step is the following: start with a set of initial approximations

$$x_i^0, \quad i = 1, \dots, n$$

and then iterate the above formula to get subsequent iterations

$$x_i^{(k)} = \frac{\left(b_i - \sum_{j \neq i} a_{ij}x_j^{(k-1)}\right)}{a_{ii}}$$

for  $k = 0, 1, \dots$  until we get convergence.

The method is not guaranteed to converge, but if the matrix is *strongly diagonally dominant*, meaning

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|$$

then in fact the process is guaranteed to converge.

### 11.3.1 Gauss-Seidel Iteration

One can get faster convergence by substituting the new values for  $x_i^{(k)}$  as soon as they become available. That is, use

$$x_i^{(k)} = \frac{\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^n a_{ij}x_j^{(k-1)}\right)}{a_{ii}}$$

### 11.3.2 Parallelism

Note that Jacobi vs. Gauss-Seidel has a major affect on parallelism. Jacobi is naturally parallelizable, but Gauss-Seidel seems to be inherently sequential.

## 12 Finite Differences and Parallelized Linear Algebra Computations

Consider Laplace's heat equation

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0$$

Solve for  $f(x, y)$  in the 2-D space.

Don't solve the *differential* equation. Convert it to a *difference* equation. We start with the central difference formula

$$\frac{\partial^2 f}{\partial x^2} \approx \frac{f(x + \Delta, y) - 2f(x, y) + f(x - \Delta, y)}{\Delta^2}$$

$$\frac{\partial^2 f}{\partial y^2} \approx \frac{f(x, y + \Delta) - 2f(x, y) + f(x, y - \Delta)}{\Delta^2}$$

Substitute in, and rearrange to pull out the  $f(x, y)$  term.

$$f(x, y) \approx \frac{f(x - \Delta, y) + f(x, y - \Delta) + f(x + \Delta, y) + f(x, y + \Delta)}{4}$$

So. We can solve this iteratively by turning the  $(x, y)$  space into a standard grid with increments of size  $\Delta$ . Number the grid points in a natural order, and the finite difference equation above then becomes

$$x_{i-n} + x_{i-1} - 4x_i + x_{i+1} + x_{i+n} = 0$$

once we look at the locations as in the diagram.

This turns the problem into one dealing with a *sparse* and *banded* matrix.

## 12.1 Parallelizing the Computation

We would like to get the convergence behavior of Gauss-Seidel with the parallelism that one could get from Jacobi.

### 12.1.1 Red-Black Ordering

Go back to the grid. Notice that we have locality of reference in the star shape.

We can adopt a *red-black* ordering scheme, which will allow us to parallelize.

### 12.1.2 Wavefront Ordering

Or we can use a wavefront approach.