COMP90025 Parallel and Multicore Computing

Matrices and Guassian Elimination

Aaron Harwood

School of Computing and Information Systems
The University of Melbourne

2019 Semester II

Matrix multiplication

An $n \times m$ matrix, A, has n rows and m columns,

$$\mathrm{A} = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,m-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,m-1} \end{bmatrix},$$

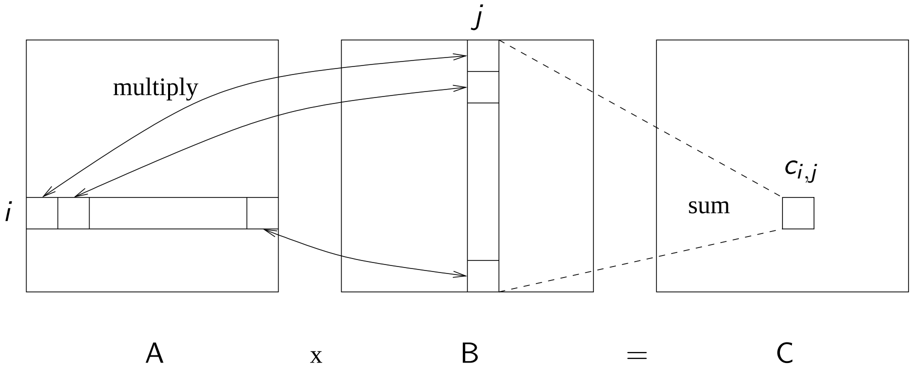
where $a_{i,j}$ is an element of A at the i-th row and j-th column.

An $m \times l$ matrix, A, can be multiplied by a $l \times n$ matrix, B, to produce an $m \times n$ matrix, C:

$$AB = C,$$

where

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



j
multiply
i
$c_{i,j}$
sum
A
x
B
=
C

For convenience, and without loss in generality, we can assume that all the matrices are square, $n \times n$. The sequential code explicitly shows the computations.

```
int A[n][n], B[n][n], C[n][n];
⋮
for(i=0;i<n;i++)
  for(j=0;j<n;j++){
    C[i][j]=0;
    for(k=0;k<n;k++)
      C[i][j]+=A[i][k]*B[k][j];
  }
```

The number of computational steps required is $O(n^3)$.

Unlike prefix sum, there is no flow dependence on data and all computations are independent of one another, i.e. no multiplication $a_{i,k}b_{k,j}$ is used twice. Consequently a large number of processors can be used to cost effectively speed up the computation.

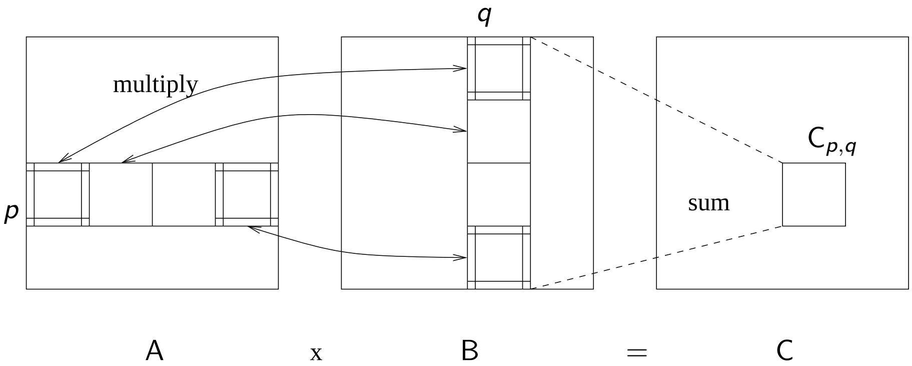
For $n \times n$ matrices and n processors, each row of the answer can be computed by a single processor. The total number of parallel computational steps is then $O(n^2)$.

For n^2 processors, each element of the answer can be computed by a single processor. The total number of parallel computational steps is then $O(n)$.

For n^3 processors it is possible to obtain a total number of parallel computational steps of $O(\log n)$. See that the summation is a reduction operation.

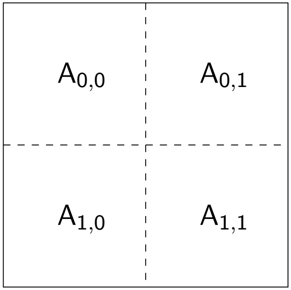
Usually the number of processors is much less than n when working with matrices of order $n \times n$. Although it may be possible to allocate n processes or threads, this approaches leads to excessive overheads.
Consider a large matrix, A, of order $n \times n$, such that the matrix can be divided into s^2 submatrices, $\mathrm{A}_{p,q}$, of order $m \times m$ where $m = \frac{n}{s}$. Thus $0 \leq p, q < s$.
For two large matrices, A and B, the matrix multiplication is equivalent to an element by element approach:

$$\mathrm{C}_{p,q} = \sum_{k=0}^{s-1} \mathrm{A}_{p,k} \mathrm{B}_{k,q}.$$


q
multiply
p
$C_{p,q}$
sum
A
x
B
=
C

Recursive subdivision

Consider matrices, A and B, of size $n \times n$ where $n = 2^t$. In this case the matrix can be conveniently subdivided until only single elements remain.



A

```
mat_mult(A, B, s){
  if(s==1) return C=A*B;
  s=s/2;
  P0=mat_mult(A00, B00, s);
  P1=mat_mult(A01, B10, s);
  P2=mat_mult(A00, B01, s);
  P3=mat_mult(A01, B11, s);
  P4=mat_mult(A10, B11, s);
  P5=mat_mult(A11, B10, s);
  P6=mat_mult(A10, B01, s);
  P7=mat_mult(A11, B11, s);
  C00 = P0 + P1;
  C01 = P2 + P3;
  C10 = P4 + P5;
  C11 = P6 + P7; }
```

Gaussian elimination

A system of equations in n unknowns is of the form:

$$\begin{aligned}
a_{0,0}x_0 + a_{0,1}x_1 + \cdots + a_{0,n-1}x_{n-1} &= b_0 \\
a_{1,0}x_0 + a_{1,1}x_1 + \cdots + a_{1,n-1}x_{n-1} &= b_1 \\
&\vdots \\
a_{n-1,0}x_0 + a_{n-1,1}x_1 + \cdots + a_{n-1,n-1}x_{n-1} &= b_{n-1}
\end{aligned}$$

Written in matrix form this becomes:

$$\mathrm{Ax} = \mathrm{b}.$$

We are trying to find x that satisfies the equation.

Sparse matrices have many zeroes, while *dense* matrices have mostly non zero values.
Sparse matrices tend to admit more efficient methods for storing and computation, that may however also use approximations.
The *direct* method to solve a system of linear equations that have a dense coefficient matrix, A, is Gaussian elimination.

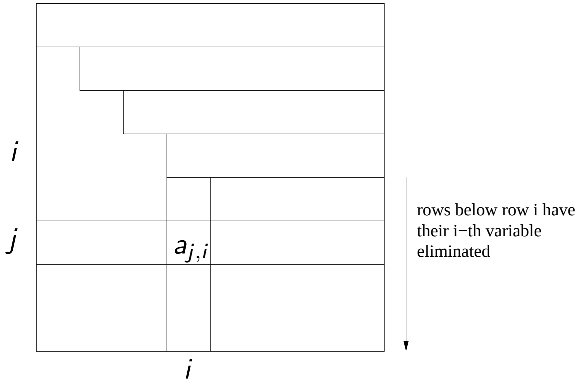
The objective of Gaussian elimination is to progressively "eliminate" variables from the system, until a single variable remains. This is the same method that you would use by hand.

Given an equation in the system, say the i-th equation, we can eliminate an unknown from another equation, j, by multiplying equation i by a constant factor, $-a_{j,i}/a_{i,i}$, and adding the resulting equation to equation j. For example:

$$2x + 3y = 7 \text{ ;Equation 0}$$
$$5x + y = -4 \text{ ;Equation 1}$$

Multiplying Equation 0 by $-a_{1,0}/a_{0,0} = -5/2$ and adding to Equation 1 yields:

$$\frac{-13y}{2} = \frac{-43}{2} \Rightarrow y = \frac{43}{13}$$


i
j
$a_{j,i}$
i
rows below row i have their i−th variable eliminated

One problem with Gaussian elimination is that when $a_{i,i}$ is zero or close to zero, the factor $-a_{j,i}/a_{i,i}$ is unstable.
In this case, the i-th row must be swapped, called *partial pivoting*, with the row, j, below i that has the greatest $a_{j,i}$. This does not affect the system of equations.
Partial pivoting can be done by finding the maximum in $\log_2 n$ steps.
If all the variables are zero or close to zero then the variables in that column are already eliminated.

Ignoring the partial pivoting requirement, the sequential code is:

```
for(i=0;i<n-1;i++)
  for(j=i+1;j<n;j++){
    m=a[j][i]/a[i][i];
    for(k=i;k<n;k++)
      a[j][k]=a[j][k]-a[i][k]*m;
    b[j]=b[j]-b[i]*m;
  }
```

The time complexity is $O(n^3)$.

A parallel implementation can assign one row to each processor. With n rows, processor P_0 has row 0, processor P_1 has row 1 and so on.
The implementation has $n-1$ phases. In the first phase P_0 broadcasts its row to all the other processors $\{\, P_i \mid i > 0 \,\}$. These processors eliminate the first variable. In the second phase P_1 broadcasts its row (apart from the eliminated variable) to processors $\{\, P_i \mid i > 1 \,\}$. These processors eliminate the second variable.
In the i-th phase, processor P_i broadcasts its remaining $n-i+1$ variables ($+1$ is the b_i constant) to $n-i-1$ remaining rows. Each processor must operate on $n-1+1$ elements in each computational step.

Clearly some processors are idle for a significant fraction of the overall computation, e.g. P_0 becomes idle after the first phase.
Partitioning the systems over a small number of processors is best done using *cyclic-striped partitioning*. This manages to keep all the processors busy for a greater period of time than block partitions.

processor 1
processor 2
processor 3
processor 1
processor 2
processor 3
processor 1
processor 2