High-Performance Computing: Matrix Multiplication

Adrian F. Clark: alien@essex.ac.uk

2016–17

Matrix multiplication

High-performance computing generally involves large numbers of calculations or a lot or data (or both)

For many problems, we end up with large (say $> 20,000 \times 20,000$ element) matrices of numbers that have to be multiplied together

This is a pesky problem because matrix multiplication is not a 'nice' operation for distributed computing

Let us imagine that

- $\mathbf{A}$ is a $N \times L$ matrix
- $\mathbf{B}$ is a $L \times M$ matrix

These mean that $\mathbf{C} = \mathbf{AB}$ is a $N \times M$ matrix

Each element of **C** is calculated as

$$C_{i,j} = \sum_{k=0}^{L} A_{i,k} B_{k,j}$$

This translates into code as

```
for (i = 0; i < n; i++)
  for (j = 0; j < m; j++)
    C[i][j] = 0.0;
    for (k = 0; k < L; k++)
      C[i][j] += A[i][k] * B[k][j];
```

which clearly scales as $O(n^3)$ — for each element in **C**, we do L multiplications and additions

Memory access pattern of matrix multiplication

In C-like languages (C, C++, Java), elements along a row are stored at adjacent memory locations

Hence, accesses to **A** and **C** in memory are efficient

However, adjacent accesses in **B** are separated by the row length M (we say they have a *stride length* of M)

For decent-sized arrays, we get a cache miss for every element of **B** accessed, so the processor spends most of its time waiting for memory accesses

To be able to maximise cache hits, matrices need to be allocated as a contiguous chunk of memory

Modern processors have *prefetching*: when regular memory access is detected, the next cache line is automatically brought into main memory before access to the data is needed — so we should try to arrange memory access to exploit this

Static memory allocation

```
int n;
printf ("Value of N? ");
scanf ("%d%, &n);
double A[N][N];
```

Elements of `A` are stored in row-major order as a contiguous block

Allocation takes place from the stack, which places an upper limit on the amount that can be allocated

Dynamic memory allocation

Using `malloc` in C or `new` in C++ allocates memory from the heap

There are no limits to the amount of memory that can be allocated (up to the amount of memory in the system)

Allocation from the heap is a comparatively slow process

```
double *mat;
mat = (double *)malloc (nrows * ncols * sizeof (double));
for (i = 0; i < nrows; i++);
  for (j = 0; j < ncols; j++)
    mat[i*cols+j] = 0.0;
```

We can make this less ugly

```
#define MAT(i,j) (mat[i*cols+j])

for (i = 0; i < nrows; i++);
  for (j = 0; j < ncols; j++)
    MAT(i, j) = 0.0;
```

Allocating 2D arrays

For more elegant subscripting, we can allocate a row at a time

```
double **mat;
mat = (double **)malloc (nrows * ncols * sizeof (double *));
for (i = 0; i < nrows; i++)
  mat[i] = (double *)malloc (ncols * sizeof (double));
for (i = 0; i < nrows; i++)
  for (j = 0; i < ncols; j++)
    mat[i][j] = 0.0;
```

but this gives potentially non-contiguous allocations, so we get a cache miss between adjacent rows and we cannot transfer the entire matrix to another process without first copying it

Contiguous 2D array allocation

```
double **mat, *m_data;
mat = (double **)malloc (nrows * ncols * sizeof (double *));
m_data = (double *)malloc (nrows * ncols * sizeof (double));
// Initialise row pointers
for (i = 0; i < nrows; i++)
  mat[i] = m_data + i * nrows;
for (i = 0; i < nrows; i++)
  for (j = 0; i < ncols; j++)
    mat[i][j] = 0.0;
```

This gives the best of both worlds, though with a small memory overhead

Parallel matrix multiplication

Needless to say, there has been a great deal of investigation into parallel matrix multiplication algorithms

We shall work towards an algorithm based on block decomposition suitable for use with message-passing systems such as MPI (*Fox's algorithm*)

All the elements $C_{i,j}$ of the result are computed in parallel

There are different methods intended for shared memory machines, typically using row decomposition

To make things easier, let us assume that

- the matrices are of size $N \times N$
- we have N^2 processors arranged in a 2D grid
- the data have already been distributed to the processes

Fox's algorithm

A and **B** are $N \times N$ matrices

Let $q = \sqrt{p}$ be an integer such that it divides N evenly, where p is the number of processes

Create a Cartesian topology with processes $(i, j), i, j = 1, \ldots, q - 1$

Denote $m = N/q$

Distribute **A** and **B** by blocks on p processes such that $A_{i,j}$ and $B_{i,j}$ are $m \times m$ blocks stored on process (i, j)

On process (i, j), we want to compute

$$C_{i,j} = \sum_{k=0}^{q-1} A_{i,k} B_{k,j}$$

$$= A_{i,0} B_{0,j} + A_{i,1} B_{1,j} + \cdots + A_{i,i} B_{i,j}$$

$$+ A_{i,i+1} B_{i+1,j} + \cdots + A_{i,q-1} B_{q-1,j}$$

stage	compute
0	$C_{i,j} = A_{i,i} B_{i,j}$
1	$C_{i,j} = C_{i,j} + A_{i,i+1} B_{i+1,j}$
⋮	⋮
	$C_{i,j} = C_{i,j} + A_{i,q-1} B_{q-1,j}$
	$C_{i,j} = C_{i,j} + A_{i,0} B_{0,j}$
⋮	⋮
	$C_{i,j} = C_{i,j} + A_{i,i-1} B_{i-1,j}$
⋮	⋮

Stage 0

Each process computes in stages:

- process (i, j) has $A_{i,j}$ and $B_{i,j}$ and needs $A_{i,i}$
- process (i, i) broadcasts $A_{i,i}$ along row i
- process (i, j) computes $C_{i,j} = A_{i,i}B_{i,j}$

Stage 1

- process (i, j) has $A_{i,j}$ and $B_{i,j}$ and needs $A_{i,i+1}$ and $B_{i+1,j}$
- shift the jth block column of $\mathbf{B}$ by one block upwards cyclically
- process $(i, i+1)$ broadcasts $A_{i,i+1}$ along row i
- process (i, j) computes $C_{i,j} = C_{i,j} + A_{i,i+1}B_{i+1,j}$

. . . and so on

Consider multiplying the 3×3 matrices $\mathbf{C} = \mathbf{AB}$

Stage 0

Broadcast $A_{i,i}$ along row i

$$\begin{array}{lll} A_{00}, B_{00} & A_{00}, B_{01} & A_{00}, B_{02} \\ A_{11}, B_{10} & A_{11}, B_{11} & A_{11}, B_{12} \\ A_{22}, B_{20} & A_{22}, B_{21} & A_{22}, B_{22} \end{array}$$

Process (i, j) computes

$$\begin{array}{lll} C_{00} = A_{00}B_{00} & C_{01} = A_{00}B_{01} & C_{02} = A_{00}B_{02} \\ C_{10} = A_{11}B_{10} & C_{11} = A_{11}B_{11} & C_{12} = A_{11}B_{12} \\ C_{20} = A_{22}B_{20} & C_{21} = A_{22}B_{21} & C_{22} = A_{22}B_{22} \end{array}$$

Cyclically shift the columns of **B**

$$\begin{array}{lll} A_{00}, B_{10} & A_{00}, B_{11} & A_{00}, B_{12} \\ A_{11}, B_{20} & A_{11}, B_{21} & A_{11}, B_{22} \\ A_{22}, B_{00} & A_{22}, B_{01} & A_{22}, B_{02} \end{array}$$

Stage 1

Process $(i, (i+1) \mod 3)$ broadcasts along row i

$$\begin{array}{lll} A_{01}, B_{10} & A_{01}, B_{11} & A_{01}, B_{12} \\ A_{12}, B_{20} & A_{12}, B_{21} & A_{12}, B_{22} \\ A_{20}, B_{00} & A_{20}, B_{01} & A_{20}, B_{02} \end{array}$$

Process (i, j) computes

$$\begin{array}{lll} C_{00} += A_{01} B_{10} & C_{01} += A_{01} B_{11} & C_{02} += A_{01} B_{12} \\ C_{10} += A_{12} B_{20} & C_{11} += A_{12} B_{21} & C_{12} += A_{12} B_{22} \\ C_{20} += A_{20} B_{00} & C_{21} += A_{20} B_{01} & C_{22} += A_{20} B_{02} \end{array}$$

Cyclically shift the columns of **B**

$$\begin{array}{lll} A_{01}, B_{20} & A_{01}, B_{21} & A_{01}, B_{22} \\ A_{10}, B_{00} & A_{10}, B_{01} & A_{10}, B_{02} \\ A_{21}, B_{10} & A_{21}, B_{11} & A_{21}, B_{12} \end{array}$$

Stage 2

Process $(i, (i+1) \mod 3)$ broadcasts along row i

$$\begin{array}{lll} A_{02}, B_{20} & A_{02}, B_{21} & A_{02}, B_{22} \\ A_{10}, B_{00} & A_{10}, B_{01} & A_{10}, B_{02} \\ A_{21}, B_{10} & A_{21}, B_{11} & A_{21}, B_{12} \end{array}$$

Process (i, j) computes

$$\begin{array}{lll} C_{00} += A_{02} B_{20} & C_{01} += A_{02} B_{21} & C_{02} += A_{02} B_{22} \\ C_{10} += A_{10} B_{00} & C_{11} += A_{10} B_{01} & C_{12} += A_{10} B_{02} \\ C_{20} += A_{21} B_{10} & C_{21} += A_{21} B_{11} & C_{22} += A_{21} B_{12} \end{array}$$