Matrix Multiplication

CPS343

Parallel and High Performance Computing

Spring 2018
1 Matrix operations
   - Importance
   - Dense and sparse matrices
   - Matrices and arrays

2 Matrix-vector multiplication
   - Row-sweep algorithm
   - Column-sweep algorithm

3 Matrix-matrix multiplication
   - “Standard” algorithm
   - $ijk$-forms
1. **Matrix operations**
   - Importance
   - Dense and sparse matrices
   - Matrices and arrays

2. **Matrix-vector multiplication**
   - Row-sweep algorithm
   - Column-sweep algorithm

3. **Matrix-matrix multiplication**
   - “Standard” algorithm
   - $ijk$-forms
A matrix is a rectangular two-dimensional array of numbers. We say a matrix is $m \times n$ if it has $m$ rows and $n$ columns. These values are sometimes called the dimensions of the matrix. Note that, in contrast to Cartesian coordinates, we specify the number of rows (the vertical dimension) and then the number of columns (the horizontal dimension). In most contexts, the rows and columns are numbered starting with 1. Several programming APIs, however, index rows and columns from 0. We use $a_{ij}$ to refer to the entry in $i^{th}$ row and $j^{th}$ column of the matrix $A$. 
Matrices are extremely important in HPC

- While it’s certainly not the case that high performance computing involves only computing with matrices, matrix operations are key to many important HPC applications.

- Many important applications can be “reduced” to operations on matrices, including (but not limited to)
  1. searching and sorting
  2. numerical simulation of physical processes
  3. optimization

- The list of the top 500 supercomputers in the world (found at http://www.top500.org) is determined by a benchmark program that performs matrix operations.

- Like most benchmark programs, this is just one measure, however, and does not predict the relative performance of a supercomputer on non-matrix problems, or even different matrix-based problems.
Outline

1. Matrix operations
   - Importance
   - Dense and sparse matrices
   - Matrices and arrays

2. Matrix-vector multiplication
   - Row-sweep algorithm
   - Column-sweep algorithm

3. Matrix-matrix multiplication
   - “Standard” algorithm
   - $ijk$-forms
The $m \times n$ matrix $A$ is *dense* if all or most of its entries are nonzero.

Storing a dense matrix (sometimes called a *full* matrix) requires storing all $mn$ elements of the matrix.

Usually an array data structure is used to store a dense matrix.
Dense matrix example

Find a matrix to represent this complete graph if the \( ij \) entry contains the weight of the edge connecting node corresponding to row \( i \) with the node corresponding to column \( j \). Use the value 0 if a connection is missing.

\[
\begin{bmatrix}
0 & 1 & 2 & 3 & 4 & 5 \\
1 & 0 & 6 & 7 & 8 & 9 \\
2 & 6 & 0 & 10 & 11 & 12 \\
3 & 7 & 10 & 0 & 13 & 14 \\
4 & 8 & 11 & 13 & 0 & 15 \\
5 & 9 & 12 & 14 & 15 & 0
\end{bmatrix}
\]
Dense matrix example

\[
\begin{bmatrix}
0 & 1 & 2 & 3 & 4 & 5 \\
1 & 0 & 6 & 7 & 8 & 9 \\
2 & 6 & 0 & 10 & 11 & 12 \\
3 & 7 & 10 & 0 & 13 & 14 \\
4 & 8 & 11 & 13 & 0 & 15 \\
5 & 9 & 12 & 14 & 15 & 0
\end{bmatrix}
\]

Note:

- This is considered a dense matrix even though it contains zeros.
- This matrix is symmetric, meaning that \( a_{ij} = a_{ji} \).
- What would be a good way to store this matrix?
A matrix is *sparse* if most of its entries are zero.

Here “most” is not usually just a simple majority, rather we expect the number of zeros to far exceed the number of nonzeros.

It is often most efficient to store only the nonzero entries of a sparse matrix, but this requires that location information also be stored.

Arrays and lists are most commonly used to store sparse matrices.
Find a matrix to represent this graph if the $ij$ entry contains the weight of the edge connecting node corresponding to row $i$ with the node corresponding to column $j$. As before, use the value 0 if a connection is missing.

$$
\begin{bmatrix}
0 & 0 & 0 & 3 & 0 & 5 \\
0 & 0 & 6 & 0 & 0 & 9 \\
0 & 6 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 14 \\
0 & 0 & 0 & 0 & 0 & 15 \\
5 & 9 & 0 & 14 & 15 & 0 \\
\end{bmatrix}
$$
Sparse matrix example

Sometimes it's helpful to leave out the zeros to better see the structure of the matrix

\[
\begin{bmatrix}
0 & 0 & 0 & 3 & 0 & 5 \\
0 & 0 & 6 & 0 & 0 & 9 \\
0 & 6 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 14 \\
0 & 0 & 0 & 0 & 0 & 15 \\
5 & 9 & 0 & 14 & 15 & 0
\end{bmatrix}
= 
\begin{bmatrix}
3 & 5 \\
6 & 9 \\
6 \\
3 & 14 \\
15 \\
5 & 9 & 14 & 15
\end{bmatrix}
\]

- This matrix is also symmetric.
- How could it be stored efficiently?
Banded matrices

- An important type of sparse matrices are *banded matrices*.
- Nonzeros are along diagonals close to main diagonal.
- Example:

\[
\begin{bmatrix}
3 & 1 & 6 & 0 & 0 & 0 & 0 \\
4 & 8 & 5 & 0 & 0 & 0 & 0 \\
1 & 2 & 1 & 1 & 3 & 0 & 0 \\
0 & 1 & 0 & 4 & 2 & 6 & 0 \\
0 & 0 & 6 & 9 & 5 & 2 & 5 \\
0 & 0 & 0 & 7 & 1 & 8 & 7 \\
0 & 0 & 0 & 0 & 4 & 4 & 9 \\
\end{bmatrix}
= 
\begin{bmatrix}
3 & 1 & 6 \\
4 & 8 & 5 & 0 \\
1 & 2 & 1 & 1 & 3 \\
1 & 0 & 4 & 2 & 6 \\
6 & 9 & 5 & 2 & 5 \\
7 & 1 & 8 & 7 \\
4 & 4 & 9 \\
\end{bmatrix}
\]

- The *bandwidth* of this matrix is 5.
Outline

1. Matrix operations
   - Importance
   - Dense and sparse matrices
   - Matrices and arrays

2. Matrix-vector multiplication
   - Row-sweep algorithm
   - Column-sweep algorithm

3. Matrix-matrix multiplication
   - “Standard” algorithm
   - $ijk$-forms
It is natural to use a 2D array to store a dense or banded matrix. Unfortunately, there are a couple of significant issues that complicate this seemingly simple approach.

1. Row-major vs. column-major storage pattern is language dependent.
2. It is not possible to dynamically allocate two-dimensional arrays in C and C++; at least not without pointer storage and manipulation overhead.
Row-major storage

Both C and C++ use what is often called a *row-major* storage pattern for 2D matrices.

- In C and C++, the last index in a multidimensional array indexes contiguous memory locations. Thus \( a[i][j] \) and \( a[i][j+1] \) are adjacent in memory.

- Example:

  \[
  \begin{array}{cccccc}
  0 & 1 & 2 & 3 & 4 \\
  5 & 6 & 7 & 8 & 9 \\
  \end{array}
  \begin{array}{cccccc}
  0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
  \end{array}
  \]

- The *stride* between adjacent elements in the same row is 1. The stride between adjacent elements in the same column is the row length (5 in this example).
In Fortran 2D matrices are stored in *column-major* form.

- The first index in a multidimensional array indexes contiguous memory locations. Thus \( a(i,j) \) and \( a(i+1,j) \) are adjacent in memory.

- Example:

  \[
  \begin{array}{cccc}
  0 & 1 & 2 & 3 & 4 \\
  5 & 6 & 7 & 8 & 9 \\
  \end{array}
  \begin{array}{cccc}
  0 & 5 & 1 & 6 & 2 & 7 & 3 & 8 & 4 & 9 \\
  \end{array}
  \]

- The stride between adjacent elements in the same row is the column length (2 in this example) while the stride between adjacent elements in the same column is 1.

- Notice that if C is used to read a matrix stored in Fortran (or vice-versa), the *transpose* matrix will be read.
Significance of array ordering

There are two main reasons why HPC programmers need to be aware of this issue:

1. Memory access patterns can have a dramatic impact on performance, especially on modern systems with a complicated memory hierarchy. These code segments access the same elements of an array, but the order of accesses is different.
   - Access by rows
     
     ```c
     for (i = 0; i < 2; i++)
     for (j = 0; j < 5; j++)
     a[i][j] = ...;
     ```
   - Access by columns
     
     ```c
     for (j = 0; j < 5; j++)
     for (i = 0; i < 2; i++)
     a[i][j] = ...;
     ```

2. Many important numerical libraries (e.g. LAPACK) are written in Fortran. To use them with C or C++ the programmer must often work with a transposed matrix.
Significance of array ordering

There are two main reasons why HPC programmers need to be aware of this issue:

1. Memory access patterns can have a dramatic impact on performance, especially on modern systems with a complicated memory hierarchy. These code segments access the same elements of an array, but the order of accesses is different.

   **Access by rows**
   ```c
   for (i = 0; i < 2; i++)
     for (j = 0; j < 5; j++)
       a[i][j] = ...  
   ```

   **Access by columns**
   ```c
   for (j = 0; j < 5; j++)
     for (i = 0; i < 2; i++)
       a[i][j] = ...  
   ```

2. Many important numerical libraries (e.g. LAPACK) are written in Fortran. To use them with C or C++ the programmer must often work with a transposed matrix.
Significance of array ordering

There are two main reasons why HPC programmers need to be aware of this issue:

1. Memory access patterns can have a dramatic impact on performance, especially on modern systems with a complicated memory hierarchy. These code segments access the same elements of an array, but the order of accesses is different.

   Access by rows
   
   ```c
   for (i = 0; i < 2; i++)
       for (j = 0; j < 5; j++)
           a[i][j] = ...
   ```

   Access by columns
   
   ```c
   for (j = 0; j < 5; j++)
       for (i = 0; i < 2; i++)
           a[i][j] = ...
   ```

2. Many important numerical libraries (e.g. LAPACK) are written in Fortran. To use them with C or C++ the programmer must often work with a transposed matrix.
Outline

1. Matrix operations
   - Importance
   - Dense and sparse matrices
   - Matrices and arrays

2. Matrix-vector multiplication
   - Row-sweep algorithm
   - Column-sweep algorithm

3. Matrix-matrix multiplication
   - “Standard” algorithm
   - $ijk$-forms
Row-sweep matrix-vector multiplication

Row-major matrix-vector product $y = Ax$, $A$ is $M \times N$:

```c
for (i = 0; i < M; i++)
{
    y[i] = 0.0;
    for (j = 0; j < N; j++)
    {
        y[i] += a[i][j] * x[j];
    }
}
```

- matrix elements accessed in row-major order
- repeated consecutive updates to $y[i]$...
- ...we can usually assume the compiler will optimize this
- also called *inner product form* since the $i^{th}$ entry of $y$ is the result of an inner product between the $i^{th}$ column of $A$ and the vector $x$. 
Outline

1. Matrix operations
   - Importance
   - Dense and sparse matrices
   - Matrices and arrays

2. Matrix-vector multiplication
   - Row-sweep algorithm
   - Column-sweep algorithm

3. Matrix-matrix multiplication
   - “Standard” algorithm
   - $ijk$-forms
Column-sweep matrix-vector multiplication

Column-major matrix-vector product $\mathbf{y} = A\mathbf{x}$, $A$ is $M \times N$:

```c
for (i = 0; i < M; i++)
{
    y[i] = 0.0;
}

for (j = 0; j < N; j++)
{
    for (i = 0; i < M; i++)
    {
        y[i] += a[i][j] * x[j];
    }
}
```

- matrix elements accessed in column-major order
- repeated updates to $y[i]$, but every element in $y$ array is updated before any element is updated again.
- also called *outer product form*. 
Which of these two algorithms will run faster? Why?

Row-Sweep Form

```c
for (i = 0; i < M; i++)
{
    y[i] = 0.0;
    for (j = 0; j < N; j++)
    {
        y[i] += a[i][j] * x[j];
    }
}
```

Column-Sweep Form

```c
for (i = 0; i < M; i++)
{
    y[i] = 0.0;
}
for (j = 0; j < N; j++)
{
    for (i = 0; i < M; i++)
    {
        y[i] += a[i][j] * x[j];
    }
}
Matrix-vector algorithm comparison

Answer: *it depends...*

- Both algorithms carry out the same operations, but do so in a different order.
- In particular, the memory access patterns are quite different.
- The row-sweep form will typically work better using a language like C or C++ which access 2D arrays in row-major form.
- Since Fortran accesses 2D arrays column-by-column, it is usually best to use the column-sweep form when working in that language.
To compute the *computation rate* in FLOPS we need to know the number of floating point operations carried out and the elapsed time.

Divisions are usually the most expensive of the four basic operations, followed by multiplication. Addition and subtraction are equivalent in terms of time and faster than multiplication or division.

We usually count all four of these operations on floating point numbers but ignore integer operations (e.g. array subscript calculations).

In the case of a matrix-vector product, the innermost loop body contains a multiplication and an addition:

\[ y_i = y_i + a_{ij}x_j \]

The inner and outer loop are done \( M \) and \( N \) times respectively, so there are a total of \( 2MN \) FLOPs.
1. Matrix operations
   - Importance
   - Dense and sparse matrices
   - Matrices and arrays

2. Matrix-vector multiplication
   - Row-sweep algorithm
   - Column-sweep algorithm

3. Matrix-matrix multiplication
   - “Standard” algorithm
   - $ijk$-forms
The textbook algorithm

Consider the problem of multiplying two matrices:

\[
C = AB = \begin{bmatrix}
5 & 2 & 3 & 3 & 0 \\
1 & 8 & 4 & 2 & 6 \\
2 & 3 & 7 & 9 & 2
\end{bmatrix}
\begin{bmatrix}
3 & 8 & 2 \\
5 & 4 & 0 \\
1 & 3 & 6 \\
2 & 7 & 5 \\
4 & 0 & 2
\end{bmatrix}
\]

The standard “textbook” algorithm to form the product \(C\) of the \(M \times P\) matrix \(A\) and the \(P \times N\) matrix \(B\) is based on the inner product.

The \(c_{ij}\) entry in the product is the inner product (or dot product) of the \(i^{th}\) row of \(A\) and the \(j^{th}\) column of \(B\).
The textbook algorithm

\textit{ijk}-form Matrix-matrix product pseudocode:

\begin{verbatim}
    for i = 1 to M
        for j = 1 to N
            c(i,j) = 0
            for k = 1 to P
                c(i,j) = c(i,j) + a(i,k) * b(k,j)
            end
        end
    end
\end{verbatim}

- known as the \textit{ijk}-form of the product due to the loop ordering

\textbf{What is the operation count...?}

Number of FLOPs is $2MNP$. For square matrices $M = N = P = n$, so number of FLOPs is $2n^3$. 

CPS343 (Parallel and HPC)
The textbook algorithm

**ijk-form Matrix-matrix product pseudocode:**

```plaintext
for i = 1 to M
    for j = 1 to N
        c(i,j) = 0
        for k = 1 to P
            c(i,j) = c(i,j) + a(i,k) * b(k,j)
        end
    end
end
```

- known as the *ijk*-form of the product due to the loop ordering
- What is the operation count...?
The textbook algorithm

**ijk-form** Matrix-matrix product pseudocode:

```plaintext
for i = 1 to M
  for j = 1 to N
    c(i,j) = 0
    for k = 1 to P
      c(i,j) = c(i,j) + a(i,k) \* b(k,j)
    end
  end
end
```

- known as the *ijk*-form of the product due to the loop ordering
- What is the operation count...?
- Number of FLOPs is $2MN$. 
The textbook algorithm

**ijk-form Matrix-matrix product pseudocode:**

```plaintext
for i = 1 to M
  for j = 1 to N
    c(i,j) = 0
    for k = 1 to P
      c(i,j) = c(i,j) + a(i,k) * b(k,j)
    end
  end
end
```

- known as the *ijk*-form of the product due to the loop ordering
- What is the operation count...?
- Number of FLOPs is $2MNP$.
- For square matrices $M = N = P = n$ so number of FLOPs is $2n^3$. 
**The textbook algorithm**

*i*:*j*:*k*-form Matrix-matrix product pseudocode:

```plaintext
for i = 1 to M
  for j = 1 to N
    c(i,j) = 0
    for k = 1 to P
      c(i,j) = c(i,j) + a(i,k) * b(k,j)
    end
  end
end
```

- Notice that *A* is accessed row-by-row but *B* is accessed column-by-column.
The textbook algorithm

**ijk-form Matrix-matrix product pseudocode:**

```
for i = 1 to M
    for j = 1 to N
        c(i,j) = 0
        for k = 1 to P
            c(i,j) = c(i,j) + a(i,k) * b(k,j)
        end
    end
end
```

- Notice that $A$ is accessed row-by-row but $B$ is accessed column-by-column.
- The column index for $C$ varies faster than the row index, but these are constant with respect to the inner loop so is much less significant.
The textbook algorithm

**$ijk$-form Matrix-matrix product pseudocode:**

```plaintext
for i = 1 to M
    for j = 1 to N
        c(i,j) = 0
        for k = 1 to P
            c(i,j) = c(i,j) + a(i,k) * b(k,j)
        end
    end
end
```

- Notice that $A$ is accessed row-by-row but $B$ is accessed column-by-column.
- The column index for $C$ varies faster than the row index, but these are constant with respect to the inner loop so is much less significant.
- Regardless of the language we use (C or Fortran), we have an efficient access pattern for one matrix but not for the other.
The textbook algorithm

**ijk-form Matrix-matrix product pseudocode:**

```plaintext
for i = 1 to M
    for j = 1 to N
        c(i,j) = 0
        for k = 1 to P
            c(i,j) = c(i,j) + a(i,k) * b(k,j)
        end
    end
end
```

- Notice that $A$ is accessed row-by-row but $B$ is accessed column-by-column.
- The column index for $C$ varies faster than the row index, but these are constant with respect to the inner loop so is much less significant.
- Regardless of the language we use (C or Fortran), we have an efficient access pattern for one matrix but not for the other.
- Could we improve things by rearranging the order of operations?
1 Matrix operations
   • Importance
   • Dense and sparse matrices
   • Matrices and arrays

2 Matrix-vector multiplication
   • Row-sweep algorithm
   • Column-sweep algorithm

3 Matrix-matrix multiplication
   • “Standard” algorithm
   • $ijk$-forms
From linear algebra we know that column $i$ of the matrix-matrix product $AB$ is defined as “a linear combination of the columns of $A$ using the values in the $i^{th}$ column of $B$ as weights.” The pseudocode for this is:

```plaintext
for j = 1 to N
    for i = 1 to M
        c(i,j) = 0.0
    end
    for k = 1 to P
        for i = 1 to M
            c(i,j) = c(i,j) + a(i,k) * b(k,j)
        end
    end
end
```

- the loop ordering changes but the innermost statement is unchanged
- the initialization of values in $C$ is done one column at a time
- the operation count is still $2MNP$. This is the $jki$ form.
Other loop orderings are possible...
Other loop orderings are possible...

How many ways can $i$, $j$, and $k$ be arranged?
Other loop orderings are possible...

How many ways can \(i, j,\) and \(k\) be arranged?

- Recall from discrete math that this is a *permutation* problem.
- “three ways to choose the first letter, two ways to choose the second, and one way to choose the third”: \(3 \times 2 \times 1 = 6\)
- There are six possible loop orderings.
- We’ll work with all six during our first hands-on exercise 😊.