Dense Matrix-Matrix Multiplication in Parallel
(see chap. 8 of the text book)

A.Y. 2013-14
Salvatore Orlando
Introduction

• Due to their regular structure, parallel computations involving matrices and vectors readily lend themselves to data-decomposition.
• Typical algorithms rely on input, output, or intermediate data decomposition.
• Most algorithms use one- and two-dimensional block, cyclic, and block-cyclic partitionings
Basics of Matrixes

- A matrix is a rectangular array of numbers, called elements

\[
A = \begin{bmatrix}
    a_{11} & a_{12} & \cdots & \cdots & a_{1m} \\
    a_{21} & a_{22} & \cdots & \cdots & a_{2m} \\
    a_{31} & a_{32} & \cdots & \cdots & a_{3m} \\
    \vdots & \vdots & \ddots & \ddots & \vdots \\
    a_{n1} & a_{n1} & \cdots & \cdots & a_{nm}
\end{bmatrix}
\]

- \( A \) is an \( n \times m \) matrix, also denoted by \( A = [a_{ij}] \)
Basics of Matrixes (Blocks/Submatrices)

- We can partition a matrix into submatrixes / blocks

\[
A = \begin{bmatrix}
    a_{11} & a_{12} & \ldots & \ldots & a_{1m} \\
    a_{21} & a_{22} & \ldots & \ldots & a_{2m} \\
    a_{31} & a_{32} & \ldots & \ldots & a_{3m} \\
    \vdots & \vdots & \ddots & \ddots & \vdots \\
    a_{n1} & a_{n1} & \ldots & \ldots & a_{nm}
\end{bmatrix} = \begin{bmatrix}
    A_{11} & A_{12} \\
    A_{21} & A_{22}
\end{bmatrix}
\]
Basics of Matrixes (Multiplication)

- $A$ is an $n \times m$ matrix, also denoted by $A = [ a_{ij} ]$
- $B$ is an $m \times p$ matrix, also denoted by $B = [ b_{ij} ]$
- $C$ is an $n \times p$ matrix, also denoted by $C = [ c_{ij} ]$

\[ C = A \cdot B = [ c_{ij} = \sum_{k=1}^{m} a_{ik} \cdot b_{kj} ] \]

- Square matrixes
  - $A$ is an $n \times n$ matrix, also denoted by $A = [ a_{ij} ]$
  - $B$ is an $n \times n$ matrix, also denoted by $B = [ b_{ij} ]$
  - $C$ is an $n \times n$ matrix, also denoted by $C = [ c_{ij} ]$

\[ C = A \cdot B = [ c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} ] \]
Matrix-Matrix Multiplication

- Consider the problem of multiplying two $n \times n$ dense, square matrices $A$ and $B$ to yield the product matrix $C = A \times B$.
- The serial complexity is $O(n^3)$.
- We do not consider better serial algorithms (Strassen's method), although, these can be used as serial kernels in the parallel algorithms.

- A useful concept in this case is called block operations. In this view, an $n \times n$ matrix $A$ can be regarded as a $q \times q$ array of blocks/submatrixes $A_{i,j}$ ($0 \leq i, j < q$) such that each block is an $(n/q) \times (n/q)$ submatrix.

- In this view, we perform $q^3$ matrix multiplications, each involving $(n/q) \times (n/q)$ matrices.
Example of Matrix-Matrix Multiplication (blocks)

\[
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix}
= 
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\times
\begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
\]

\[
C = A \cdot B = [ C_{ij} = \sum_{k=1,2} A_{ik} \cdot B_{kj} ]
\]

Matrix multiplication & Matrix sum

\[
\begin{bmatrix}
A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{bmatrix}
= 
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\times
\begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
\]
Example of Matrix-Matrix Multiplication (blocks)

\[
\begin{align*}
C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\
C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\
C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\
C_{22} &= A_{21}B_{12} + A_{22}B_{22}
\end{align*}
\]

\[
C = A \cdot B = \begin{bmatrix}
C_{ij} = \sum_{k=1,2} A_{ik} \cdot B_{kj}
\end{bmatrix}
\]

- If \(A, B,\) and \(C\) are matrices 4 x 4, each sub-matrix is 2 x 2
- Example:

\[
\begin{align*}
A_{11}B_{11} &= a_{11}b_{11} + a_{12}b_{21} \\
A_{12}B_{21} &= a_{21}b_{11} + a_{22}b_{21} \\
A_{11}B_{12} &= a_{11}b_{12} + a_{12}b_{22} \\
A_{12}B_{22} &= a_{21}b_{12} + a_{22}b_{22}
\end{align*}
\]
• Consider two \( n \times n \) matrices \( A \) and \( B \) partitioned into \( p^2 \) sub-matrixes
  – \( A_{i,j} \) and \( B_{i,j} \) (\( 0 \leq i, j < p \)) of size \( n/p \times n/p \) (\( n \) is a multiple of \( p \))
• Process \( P_{i,j} \) (of \( p^2 \) processes)
  – initially stores \( A_{i,j} \) and \( B_{i,j} \)
  – computes block \( C_{i,j} \) of the result matrix.
• Computing submatrix \( C_{i,j} \) requires all submatrices \( A_{i,k} \) and \( B_{k,j} \) for \( 0 \leq k < p \)
• All-to-all broadcast blocks of \( A \) along rows and \( B \) along columns.
• Perform local submatrix multiplications.
Cannon Algorithm (2)

• In this algorithm, as in the previous example
  – each process $P_{i,j}$ has to perform the computation of block $C_{i,j}$ of the result matrix
  – Process $P_{i,j}$ initially stores $A_{i,j}$ and $B_{i,j}$

• No broadcast, but systematically rotations and computations

• In particular
  – We schedule the computations of the $p$ processes of the $i$th row such that, at any given time, each process is using a different block $A_{i,k}$ (a different block $B_{i,k}$)
  – These blocks can be systematically rotated among the processes after every submatrix multiplication so that every process gets a fresh $A_{i,k}$ after each rotation (a fresh $B_{i,k}$ after each rotation)
Cannon Algorithm (2)

- The initial step of the algorithm regards the alignment of the matrixes:
  - Align the blocks of $A$ and $B$ in such a way that each process can independently multiply its local submatrices
- At the end $P_{i,j}$ will stores the computed sub-matrix $C_{i,j}$
- The computation works in $p$ steps
- Initial alignment
  - the blocks are shifted vertically and horizontally, along the two dimensions of the virtual 2D mesh
  - by shifting all submatrixes $A_{i,j}$ to the left (with wraparound) by $i$ steps
  - by shifting all submatrixes $B_{i,j}$ up (with wraparound) by $j$ steps
- At each step (following the 1st one)
  - the blocks are shifted up and left (with wraparound), along the two dimensions of the virtual 2D mesh, by $l$ steps
Cannon Algorithm (2): Communication Steps

- This step leaves $P_{ij}$ with sub-matrix $A_{i, (j+i) \mod p}$ and sub-matrix $B_{(i+j) \mod p, j}$
Cannon Algorithm (2): Communication Steps

- After a local submatrix multiplication step, each block of $A$ moves one step left and each block of $B$ moves one step up (again with wraparound)

<table>
<thead>
<tr>
<th></th>
<th>A₀,₀</th>
<th>A₀,₁</th>
<th>A₀,₂</th>
<th>A₀,₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>B₀,₀</td>
<td>A₂,₂</td>
<td>A₂,₁</td>
<td>A₂,₀</td>
<td>A₂,₁</td>
</tr>
<tr>
<td>A₁,₁</td>
<td>A₁,₂</td>
<td>A₁,₃</td>
<td>A₁,₀</td>
<td>A₁,₀</td>
</tr>
<tr>
<td>B₁,₀</td>
<td>B₀,₂</td>
<td>B₀,₁</td>
<td>B₁,₂</td>
<td>B₁,₃</td>
</tr>
<tr>
<td>A₂,₂</td>
<td>A₂,₃</td>
<td>A₂,₀</td>
<td>A₂,₁</td>
<td>A₂,₂</td>
</tr>
<tr>
<td>B₂,₀</td>
<td>B₃,₁</td>
<td>B₀,₂</td>
<td>B₁,₃</td>
<td>B₂,₃</td>
</tr>
<tr>
<td>A₃,₃</td>
<td>A₃,₀</td>
<td>A₃,₁</td>
<td>A₃,₂</td>
<td>A₃,₃</td>
</tr>
<tr>
<td>B₃,₀</td>
<td>B₀,₁</td>
<td>B₁,₂</td>
<td>B₂,₂</td>
<td>B₃,₃</td>
</tr>
</tbody>
</table>

(c) A and B after initial alignment

(d) Submatrix locations after first shift
### Cannon Algorithm (2): Communication Steps

#### (a) Initial alignment of A

<table>
<thead>
<tr>
<th></th>
<th>A_{0,2}</th>
<th>A_{0,3}</th>
<th>A_{0,0}</th>
<th>A_{0,1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_{2,0}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_{3,1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_{1,3}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_{3,0}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_{2,0}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_{0,0}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_{3,1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_{1,0}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### (b) Initial alignment of B

<table>
<thead>
<tr>
<th></th>
<th>A_{0,0}</th>
<th>A_{0,1}</th>
<th>A_{0,2}</th>
<th>A_{0,3}</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_{0,2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_{0,1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_{1,0}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_{1,1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_{2,1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_{2,2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_{3,2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_{3,2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### (c) A and B after initial alignment

<table>
<thead>
<tr>
<th></th>
<th>A_{1,2}</th>
<th>A_{1,1}</th>
<th>A_{1,3}</th>
<th>A_{1,0}</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_{1,2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_{1,1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_{2,2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_{2,2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_{3,2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_{3,2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### (d) Submatrix locations after first shift

<table>
<thead>
<tr>
<th></th>
<th>A_{2,3}</th>
<th>A_{2,2}</th>
<th>A_{2,1}</th>
<th>A_{2,0}</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_{3,3}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_{3,2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_{3,3}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_{3,1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_{3,0}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_{0,3}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### (e) Submatrix locations after second shift

<table>
<thead>
<tr>
<th></th>
<th>A_{3,0}</th>
<th>A_{3,1}</th>
<th>A_{3,2}</th>
<th>A_{3,3}</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_{0,0}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_{0,1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_{0,0}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_{0,2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_{0,1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_{1,0}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### (f) Submatrix locations after third shift

<table>
<thead>
<tr>
<th></th>
<th>A_{0,3}</th>
<th>A_{0,0}</th>
<th>A_{0,1}</th>
<th>A_{0,2}</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_{3,0}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_{0,1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_{1,1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_{1,1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_{2,2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_{2,2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_{3,3}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_{3,3}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **After a local submatrix multiplication step, each block of** $A$ **moves one step left and each block of** $B$ **moves one step up (again with wraparound).**
Cannon Algorithm (2) – example 2x2 matrixes

\[
\begin{align*}
C_{00} &= A_{00}B_{00} + A_{01}B_{10} \\
C_{10} &= A_{10}B_{00} + A_{11}B_{10} \\
C_{01} &= A_{00}B_{01} + A_{01}B_{11} \\
C_{11} &= A_{10}B_{01} + A_{11}B_{11}
\end{align*}
\]

\[
\begin{align*}
A_0 &
\end{align*}
\]

\[
\begin{align*}
B_0 &
\end{align*}
\]

\[
\begin{align*}
C_{00} &= A_{00}B_{00} + A_{01}B_{10} \\
C_{10} &= A_{10}B_{00} + A_{11}B_{10} \\
C_{01} &= A_{00}B_{01} + A_{01}B_{11} \\
C_{11} &= A_{10}B_{01} + A_{11}B_{11}
\end{align*}
\]

\[
\begin{align*}
A_0 &
\end{align*}
\]

\[
\begin{align*}
B_0 &
\end{align*}
\]

\[
\begin{align*}
C_{00} &= C_{00} + A_{01}B_{10} \\
C_{10} &= A_{10}B_{00} + C_{10} \\
C_{01} &= A_{00}B_{01} + C_{01} \\
C_{11} &= C_{11} + A_{11}B_{11}
\end{align*}
\]

\[
\begin{align*}
A_0 &
\end{align*}
\]

\[
\begin{align*}
B_0 &
\end{align*}
\]

\[
\begin{align*}
A_0 &
\end{align*}
\]

\[
\begin{align*}
B_0 &
\end{align*}
\]

1st step after alignment

2nd step after shift
Cannon Algorithm (2)

- Align the blocks of $A$ and $B$ in such a way that each process multiplies its local submatrices. This is done by shifting all submatrices $A_{i,j}$ to the left (with wraparound) by $i$ steps and all submatrices $B_{i,j}$ up (with wraparound) by $j$ steps.
- Perform local block multiplication.
- Each block of $A$ moves one step left and each block of $B$ moves one step up (again with wraparound).
- Perform next block multiplication, add to partial result, repeat until all $p$ blocks have been multiplied.
- More effective that algorithm (1) with respect to memory occupancy on each node
Processor farm (3)

- Replicate matrix B on all the workers
- Each task consists in computing $K$ rows of $C$, $K < n/p$
- The master dynamically assigns the task to a worker by sending $k$ rows of $A$
Increasing the computational cost

• The simple computation of a matrix-matrix multiplication cannot be enough for taking advantage of current parallel platforms

• Compute instead:
  – $C = f(A) \times B$.

• $f()$ is function that can independently be applied to all $A[i,j]$
  – emulate the application of an expensive function $f()$ by running a loop
  – this increases the minimum granularity of parallel tasks
Evaluate and comparing different algorithms

- Constraints
  - The input data (generate synthethic floating-point FP matrixes, stored row-major as ascii files) are stored on the disk, and can be accessed by processor 0 only
    - (don’t use the mounted HOME, but the local mounted disk)
    - data blocks must be transmitted/scattered from processor 0 to all the others (algorithm 1 and 2)
    - the master of the processor farm (algorithm 3), which is responsible for data distribution to all the slaves, must thus be mapped on processor 0 in order to permit data transmission / collection
  - Also the output matrix must be collected by processor 0 and stored on the disk
  - However, it must be possible to disable the input/output of data from/to the disk
    - to remove the influence of the input/output on the total execution cost
  - You can iterate the matrix-matrix multiplication several times
Evaluate and comparing different algorithms

- **Goals**
  - Evaluate speedups for the different versions
    - with different granularities (cost of function $f()$)
    - with different problem sizes
    - note that for very large data size, sequential version can incur in trashing (superlinear speedup)
  - Evaluate system imbalance or heterogeneity *(optional)*
    - for example, by running synthetic workloads on some processors
    - the algorithm based on the processor farm (3) is naturally able to cope with these issues
    - the other two algorithms (1 and 3), based on static block-wise distribution of arrays, may suffer from this imbalance
      - a possible solution: generate a program with much more MPI processes than processors
      - assign processes to processors on the basis of their load/computational capability