Chapter 8

Matrix-vector Multiplication
Chapter Objectives

- Review matrix-vector multiplication
- Propose replication of vectors
- Develop three parallel programs, each based on a different data decomposition
Outline

- Sequential algorithm and its complexity
- Design, analysis, and implementation of three parallel programs
  - Rowwise block striped
  - Columnwise block striped
  - Checkerboard block
Sequential Algorithm

\[
\begin{array}{cccc}
2 & 1 & 0 & 4 \\
3 & 2 & 1 & 1 \\
4 & 3 & 1 & 2 \\
3 & 0 & 2 & 0 \\
\end{array}
\times
\begin{array}{c}
1 \\
3 \\
4 \\
1 \\
\end{array}
= 
\begin{array}{c}
9 \\
14 \\
19 \\
11 \\
\end{array}
\]
Storing Vectors

- Divide vector elements among processes
- Replicate vector elements
- Vector replication acceptable because vectors have only \( n \) elements, versus \( n^2 \) elements in matrices
Rowwise Block Striped Matrix

- Partitioning through domain decomposition
- Primitive task associated with
  - Row of matrix
  - Entire vector
Phases of Parallel Algorithm

Inner product computation

All-gather communication
Agglomeration and Mapping

- Static number of tasks
- Regular communication pattern (all-gather)
- Computation time per task is constant

Strategy:
- Agglomerate groups of rows
- Create one task per MPI process
Complexity Analysis

- Sequential algorithm complexity: $\Theta(n^2)$
- Parallel algorithm computational complexity: $\Theta(n^2/p)$
- Communication complexity of all-gather: $\Theta(\log p + n)$
- Overall complexity: $\Theta(n^2/p + \log p)$
Isoefficiency Analysis

- Sequential time complexity: $\Theta(n^2)$
- Only parallel overhead is all-gather
  - When $n$ is large, message transmission time dominates message latency
  - Parallel communication time: $\Theta(n)$
- $n^2 \geq Cpn \Rightarrow n \geq Cp$ and $M(n) = n^2$

$$M(Cp)/p = C^2 p^2 / p = C^2 p$$

- System is not highly scalable
Block-to-replicated Transformation
MPI_Allgatherv

Before

0

1

2

3

After

Allgatherv
MPI_Allgatherv

```c
int MPI_Allgatherv (  
    void          *send_buffer,  
    int           send_cnt,  
    MPI_Datatype  send_type,  
    void          *receive_buffer,  
    int           *receive_cnt,  
    int           *receive_disp,  
    MPI_Datatype  receive_type,  
    MPI_Comm      communicator)
```
MPI_Allgatherv in Action

**Process 0**
- send_buffer: con
- send_cnt: 3
- receive_buffer: concatenate
- receive_cnt: 3
- receive_disp: 0 3 7

**Process 1**
- send_buffer: cate
- send_cnt: 4
- receive_buffer: concatenate
- receive_cnt: 3
- receive_disp: 0 3 7

**Process 2**
- send_buffer: nate
- send_cnt: 4
- receive_buffer: concatenate
- receive_cnt: 3
- receive_disp: 0 3 7
Function `create_mixed_xfer_arrays`

- **First array**
  - How many elements contributed by each process
  - Uses utility macro `BLOCK_SIZE`

- **Second array**
  - Starting position of each process’ block
  - Assume blocks in process rank order
Function replicate_block_vector

- Create space for entire vector
- Create “mixed transfer” arrays
- Call \texttt{MPI\_Allgatherv}
Function read_replicated_vector

- Process $p-1$
  - Opens file
  - Reads vector length
- Broadcast vector length (root process = $p-1$)
- Allocate space for vector
- Process $p-1$ reads vector, closes file
- Broadcast vector
Function print_replicated_vector

- Process 0 prints vector
- Exact call to `printf` depends on value of parameter `datatype`
Run-time Expression

- $\chi$: inner product loop iteration time
- Computational time: $\chi \lfloor \frac{n}{p} \rfloor$
- All-gather requires $\lceil \log p \rceil$ messages with latency $\lambda$
- Total vector elements transmitted:
  \[
  \frac{(2^{\lceil \log p \rceil} - 1) / 2^{\lceil \log p \rceil}}{}
  \]
- Total execution time:
  \[
  \chi \lfloor \frac{n}{p} \rfloor + \lambda \lceil \log p \rceil + \frac{(2^{\lceil \log p \rceil} - 1) / (2^{\lceil \log p \rceil} \beta)}{}
  \]
## Benchmarking Results

<table>
<thead>
<tr>
<th>$p$</th>
<th>Predicted</th>
<th>Actual</th>
<th>Speedup</th>
<th>Mflops</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>63.4</td>
<td>63.4</td>
<td>1.00</td>
<td>31.6</td>
</tr>
<tr>
<td>2</td>
<td>32.4</td>
<td>32.7</td>
<td>1.94</td>
<td>61.2</td>
</tr>
<tr>
<td>3</td>
<td>22.3</td>
<td>22.7</td>
<td>2.79</td>
<td>88.1</td>
</tr>
<tr>
<td>4</td>
<td>17.0</td>
<td>17.8</td>
<td>3.56</td>
<td>112.4</td>
</tr>
<tr>
<td>5</td>
<td>14.1</td>
<td>15.2</td>
<td>4.16</td>
<td>131.6</td>
</tr>
<tr>
<td>6</td>
<td>12.0</td>
<td>13.3</td>
<td>4.76</td>
<td>150.4</td>
</tr>
<tr>
<td>7</td>
<td>10.5</td>
<td>12.2</td>
<td>5.19</td>
<td>163.9</td>
</tr>
<tr>
<td>8</td>
<td>9.4</td>
<td>11.1</td>
<td>5.70</td>
<td>180.2</td>
</tr>
<tr>
<td>16</td>
<td>5.7</td>
<td>7.2</td>
<td>8.79</td>
<td>277.8</td>
</tr>
</tbody>
</table>
Columnwise Block Striped Matrix

- Partitioning through domain decomposition
- Task associated with
  - Column of matrix
  - Vector element
Matrix-Vector Multiplication

\[ c_0 = a_{0,0} b_0 + a_{0,1} b_1 + a_{0,2} b_2 + a_{0,3} b_3 + a_{0,4} b_4 \]

\[ c_1 = a_{1,0} b_0 + a_{1,1} b_1 + a_{1,2} b_2 + a_{1,3} b_3 + a_{1,4} b_4 \]

\[ c_2 = a_{2,0} b_0 + a_{2,1} b_1 + a_{2,2} b_2 + a_{2,3} b_3 + a_{2,4} b_4 \]

\[ c_3 = a_{3,0} b_0 + a_{3,1} b_1 + a_{3,2} b_2 + a_{3,3} b_3 + a_{3,4} b_4 \]

\[ c_4 = a_{4,0} b_0 + a_{4,1} b_1 + a_{4,2} b_2 + a_{4,3} b_3 + a_{4,4} b_4 \]

Processor 0’s initial computation

Processor 1’s initial computation

Proc 2

Proc 3

Proc 4
All-to-all Exchange (Before)
All-to-all Exchange (After)

P0 | P1 | P2 | P3 | P4
---|----|----|----|----
□  | □  | □  | □  | □
□  | □  | □  | □  | □
□  | □  | □  | □  | □
□  | □  | □  | □  | □
Phases of Parallel Algorithm

1. Column $i$ of $A$
2. Column $i$ of $A$
3. Column $i$ of $A$
4. Column $i$ of $A$

Multiplications

All-to-all exchange

Reduction
Agglomeration and Mapping

- Static number of tasks
- Regular communication pattern (all-to-all)
- Computation time per task is constant

Strategy:
- Agglomerate groups of columns
- Create one task per MPI process
Complexity Analysis

- Sequential algorithm complexity: $\Theta(n^2)$

- Parallel algorithm computational complexity: $\Theta(n^2/p)$

- Communication complexity of all-to-all: $\Theta(p + n/p)$

- Overall complexity: $\Theta(n^2/p + \log p)$
Isoefficiency Analysis

- **Sequential time complexity**: $\Theta(n^2)$
- Only parallel overhead is all-to-all
  - When $n$ is large, message transmission time dominates message latency
  - Parallel communication time: $\Theta(n)$
- $n^2 \geq Cpn \Rightarrow n \geq Cp$
- Scalability function same as rowwise algorithm: $C^2p$
Reading a Block-Column Matrix
MPI_Scatterv

Before

0

1

2

3

After

Scatterv
int MPI_Scatterv (  
    void         *send_buffer,  
    int          *send_cnt,  
    int          *send_disp,  
    MPI_Datatype  send_type,  
    void         *receive_buffer,  
    int           receive_cnt,  
    MPI_Datatype  receive_type,  
    int           root,  
    MPI_Comm      communicator)
Printing a Block-Column Matrix

- Data motion opposite to that we did when reading the matrix
- Replace “scatter” with “gather”
- Use “v” variant because different processes contribute different numbers of elements
Function MPI_Gatherv

Before

0
1
2
3

After

Gatherv
Header for MPI_Gatherv

```c
int MPI_Gatherv (  
    void         *send_buffer,  
    int           send_cnt,  
    MPI_Datatype  send_type,  
    void         *receive_buffer,  
    int          *receive_cnt,  
    int          *receive_disp,  
    MPI_Datatype  receive_type,  
    int           root,  
    MPI_Comm      communicator)  
```

Functional Call MPI_Alltoally

<table>
<thead>
<tr>
<th>Processes</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram:
- Before: Each process has some data.
- After: The data is redistributed among all processes.
- The function call MPI_Alltoally transfers data between processes.
Header for MPI_Alltoallv

```c
int MPI_Gatherv ( 
    void         *send_buffer,  
    int          *send_cnt,    
    int          *send_disp, 
    MPI_Datatype  send_type,  
    void         *receive_buffer, 
    int          *receive_cnt,  
    int          *receive_disp, 
    MPI_Datatype  receive_type, 
    MPI_Comm      communicator)
```
Count/Displacement Arrays

- MPI_Alltoallv requires two pairs of count/displacement arrays
  - First pair for values being sent
    - `send_cnt`: number of elements
    - `send_disp`: index of first element
  - Second pair for values being received
    - `recv_cnt`: number of elements
    - `recv_disp`: index of first element

`create_mixed_xfer_arrays` builds these
Function `create_uniform_xfer_arrays`

- **First array**
  - How many elements received from each process (always same value)
  - Uses ID and utility macro `block_size`

- **Second array**
  - Starting position of each process’ block
  - Assume blocks in process rank order
Run-time Expression

- $\chi$: inner product loop iteration time
- Computational time: $\chi \, n\lceil n/p \rceil$
- All-gather requires $p-1$ messages, each of length about $n/p$
- 8 bytes per element
- Total execution time:
  $$\chi \, n\lceil n/p \rceil + (p-1)(\lambda + (8n/p)/\beta)$$
## Benchmarking Results

<table>
<thead>
<tr>
<th>$p$</th>
<th>Predicted</th>
<th>Actual</th>
<th>Speedup</th>
<th>Mflops</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>63.4</td>
<td>63.8</td>
<td>1.00</td>
<td>31.4</td>
</tr>
<tr>
<td>2</td>
<td>32.4</td>
<td>32.9</td>
<td>1.92</td>
<td>60.8</td>
</tr>
<tr>
<td>3</td>
<td>22.2</td>
<td>22.6</td>
<td>2.80</td>
<td>88.5</td>
</tr>
<tr>
<td>4</td>
<td>17.2</td>
<td>17.5</td>
<td>3.62</td>
<td>114.3</td>
</tr>
<tr>
<td>5</td>
<td>14.3</td>
<td>14.5</td>
<td>4.37</td>
<td>137.9</td>
</tr>
<tr>
<td>6</td>
<td>12.5</td>
<td>12.6</td>
<td>5.02</td>
<td>158.7</td>
</tr>
<tr>
<td>7</td>
<td>11.3</td>
<td>11.2</td>
<td>5.65</td>
<td>178.6</td>
</tr>
<tr>
<td>8</td>
<td>10.4</td>
<td>10.0</td>
<td>6.33</td>
<td>200.0</td>
</tr>
<tr>
<td>16</td>
<td>8.5</td>
<td>7.6</td>
<td>8.33</td>
<td>263.2</td>
</tr>
</tbody>
</table>
Checkerboard Block Decomposition

- Associate primitive task with each element of the matrix $A$
- Each primitive task performs one multiply
- Agglomerate primitive tasks into rectangular blocks
- Processes form a 2-D grid
- Vector $b$ distributed by blocks among processes in the first column of grid
Tasks after Agglomeration
Algorithm’s Phases

- Redistribute \( b \)
- Matrix-vector multiply
- Reduce vectors across rows
Redistributing Vector $b$

- **Step 1:** Move $b$ from processes in first row to processes in first column
  - If $p$ square
    - First column/first row processes send/receive portions of $b$
  - If $p$ not square
    - Gather $b$ on process 0, 0
    - Process 0, 0 broadcasts to first row procs

- **Step 2:** First row processes scatter $b$ within columns
Redistributing Vector $b$

When $p$ is a square number

When $p$ is not a square number
Complexity Analysis

- Assume $p$ is a square number
  - If grid is $1 \times p$, devolves into columnwise block striped
  - If grid is $p \times 1$, devolves into rowwise block striped
Complexity Analysis (continued)

- Each process does its share of computation:
  \( \Theta(n^2/p) \)

- Redistribute \( b \): \( \Theta(n / \sqrt{p} + \log p(n / \sqrt{p})) = \Theta(n \log p / \sqrt{p}) \)

- Reduction of partial results vectors:
  \( \Theta(n \log p / \sqrt{p}) \)

- Overall parallel complexity:
  \( \Theta(n^3/p + n \log p / \sqrt{p}) \)
Isoefficiency Analysis

- Sequential complexity: $\Theta(n^2)$
- Parallel communication complexity: $\Theta(n \log p / \sqrt{p})$
- Isoefficiency function:
  
  $n^2 \geq Cn \sqrt{p \log p} \Rightarrow n \geq C \sqrt{p \log p}$

  
  \[ M(C \sqrt{p \log p}) / p = C^2 p \log^2 p / p = C^2 \log^2 p \]

- This system is much more scalable than the previous two implementations
Creating Communicators

- Want processes in a virtual 2-D grid
- Create a custom communicator to do this
- Collective communications involve all processes in a communicator
- We need to do broadcasts, reductions among subsets of processes
- We will create communicators for processes in same row or same column
What’s in a Communicator?

- Process group
- Context
- Attributes
  - Topology (lets us address processes another way)
  - Others we won’t consider
Creating 2-D Virtual Grid of Processes

- MPI_Dims_create
  - Input parameters
    - Total number of processes in desired grid
    - Number of grid dimensions
  - Returns number of processes in each dim

- MPI_Cart_create
  - Creates communicator with cartesian topology
MPI_Dims_create

int MPI_Dims_create (  
    int nodes,  
    /* Input - Procs in grid */  
    int dims,  
    /* Input - Number of dims */  
    int *size)  
    /* Input/Output - Size of each grid dimension */
MPI_Cart_create

int MPI_Cart_create (  
    MPI_Comm old_comm, /* Input - old communicator */  

    int dims, /* Input - grid dimensions */  

    int *size, /* Input - # procs in each dim */  

    int *periodic,  
        /* Input - periodic[j] is 1 if dimension j wraps around; 0 otherwise */  

    int reorder,  
        /* 1 if process ranks can be reordered */  

    MPI_Comm *cart_comm)  
        /* Output - new communicator */
Using MPI_Dims_create and MPI_Cart_create

```c
MPI_Comm cart_comm;
int p;
int periodic[2];
int size[2];
...
size[0] = size[1] = 0;
MPI_Dims_create (p, 2, size);
periodic[0] = periodic[1] = 0;
MPI_Cart_create (MPI_COMM_WORLD, 2, size,
    1, &cart_comm);
```
Useful Grid-related Functions

- **MPI_Cart_rank**
  - Given coordinates of process in Cartesian communicator, returns process rank

- **MPI_Cart_coords**
  - Given rank of process in Cartesian communicator, returns process’ coordinates
Header for MPI_Cart_rank

```c
int MPI_Cart_rank (  
    MPI_Comm comm,  
    /* In - Communicator */  
    int *coords,  
    /* In - Array containing process’ grid location */  
    int *rank)  
    /* Out - Rank of process at specified coords */
```
Header for MPI_Cart_coords

```c
int MPI_Cart_coords ( MPI_Comm comm,
    int rank,
    int dims,
    int *coords)
```

/ * Out - Coordinates of specified process in virtual grid */
MPI_Comm_split

- Partitions the processes of a communicator into one or more subgroups
- Constructs a communicator for each subgroup
- Allows processes in each subgroup to perform their own collective communications
- Needed for columnwise scatter and rowwise reduce
Header for MPI_Comm_split

```c
int MPI_Comm_split (  
    MPI_Comm old_comm,
    /* In - Existing communicator */

    int partition, /* In - Partition number */

    int new_rank,
    /* In - Ranking order of processes
    in new communicator */

    MPI_Comm *new_comm)
    /* Out - New communicator shared by
    processes in same partition */
```
Example: Create Communicators for Process Rows

MPI_Comm grid_comm; /* 2-D process grid */

MPI_Comm grid_coords[2];
    /* Location of process in grid */

MPI_Comm row_comm;
    /* Processes in same row */

MPI_Comm_split (grid_comm, grid_coords[0], grid_coords[1], &row_comm);
Run-time Expression

- Computational time: $\chi \left\lceil n/\sqrt{p} \right\rceil \left\lfloor n/\sqrt{p} \right\rfloor$
- Suppose $p$ a square number
- Redistribute $b$
  - Send/recv: $\lambda + 8 \left\lceil n/\sqrt{p} \right\rceil / \beta$
  - Broadcast: $\log \sqrt{p} \left( \lambda + 8 \left\lceil n/\sqrt{p} \right\rceil / \beta \right)$
- Reduce partial results:
  $\log \sqrt{p} \left( \lambda + 8 \left\lceil n/\sqrt{p} \right\rceil / \beta \right)$
## Benchmarking

<table>
<thead>
<tr>
<th>Procs</th>
<th>Predicted (msec)</th>
<th>Actual (msec)</th>
<th>Speedup</th>
<th>Megaflops</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>63.4</td>
<td>63.4</td>
<td>1.00</td>
<td>31.6</td>
</tr>
<tr>
<td>4</td>
<td>17.8</td>
<td>17.4</td>
<td>3.64</td>
<td>114.9</td>
</tr>
<tr>
<td>9</td>
<td>9.7</td>
<td>9.7</td>
<td>6.53</td>
<td>206.2</td>
</tr>
<tr>
<td>16</td>
<td>6.2</td>
<td>6.2</td>
<td>10.21</td>
<td>322.6</td>
</tr>
</tbody>
</table>
Comparison of Three Algorithms

![Graph showing comparison of speedup among different algorithms with respect to the number of processors.]
Summary (1/3)

- Matrix decomposition ⇒ communications needed
  - Rowwise block striped: all-gather
  - Columnwise block striped: all-to-all exchange
  - Checkerboard block: gather, scatter, broadcast, reduce

- All three algorithms: roughly same number of messages
- Elements transmitted per process varies
  - First two algorithms: \( \Theta(n) \) elements per process
  - Checkerboard algorithm: \( \Theta(n/\sqrt{p}) \) elements

- Checkerboard block algorithm has better scalability
Summary (2/3)

- Communicators with Cartesian topology
  - Creation
  - Identifying processes by rank or coords
- Subdividing communicators
  - Allows collective operations among subsets of processes
Summary (3/3)

- Parallel programs and supporting functions much longer than C counterparts
- Extra code devoted to reading, distributing, printing matrices and vectors
- Developing and debugging these functions is tedious and difficult
- Makes sense to generalize functions and put them in libraries for reuse
MPI Application Development

Application

Application-specific Library

MPI Library

C and Standard Libraries