Chapter 3

Distributed Memory Programming with MPI
Roadmap

- Writing your first MPI program.
- Using the common MPI functions.
- The Trapezoidal Rule in MPI.
- Collective communication.
- MPI derived datatypes.
- Performance evaluation of MPI programs.
- Parallel sorting.
- Safety in MPI programs.
A distributed memory system

![Diagram of a distributed memory system with multiple CPUs and memories connected through an interconnect.](image-url)
A shared memory system
Hello World!

```c
#include <stdio.h>

int main(void) {
    printf("hello, world\n");

    return 0;
}
```

(a classic)
Identifying MPI processes

- Common practice to identify processes by nonnegative integer ranks.

- \( p \) processes are numbered 0, 1, 2, .. \( p-1 \)
Our first MPI program

```c
#include <stdio.h>
#include <string.h>  /* For strlen */
#include <mpi.h>    /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
    char greeting[MAX_STRING];
    int comm_sz; /* Number of processes */
    int my_rank; /* My process rank */

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank != 0) {
        printf(greeting, "Greetings from process %d of %d!",
                        my_rank, comm_sz);
        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
                        MPI_COMM_WORLD);
    } else {
        printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
        for (int q = 1; q < comm_sz; q++) {
            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
                        0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", greeting);
        }
    }

    MPI_Finalize();
    return 0;
} /* main */
```
Compilation

mpicc  -g  -Wall  -o  mpi_hello  mpi_hello.c

wrapper script to compile

source file

produce debugging information

turns on all warnings

create this executable file name (as opposed to default a.out)

Copyright © 2010, Elsevier Inc. All rights Reserved
### Execution

```
mpiexec -n <number of processes> <executable>
```

- `mpiexec -n 1 ./mpi_hello` *run with 1 process*
- `mpiexec -n 4 ./mpi_hello` *run with 4 processes*
Execution

mpiexec -n 1 ./mpi_hello

Greetings from process 0 of 1 !

mpiexec -n 4 ./mpi_hello

Greetings from process 0 of 4 !
Greetings from process 1 of 4 !
Greetings from process 2 of 4 !
Greetings from process 3 of 4 !
MPI Programs

- Written in C.
  - Has main.
  - Uses stdio.h, string.h, etc.
- Need to add mpi.h header file.
- Identifiers defined by MPI start with “MPI_”.
- First letter following underscore is uppercase.
  - For function names and MPI-defined types.
  - Helps to avoid confusion.
MPI Components

- **MPI_Init**
  - Tells MPI to do all the necessary setup.

```c
int MPI_Init(
    int* argc_p /* in/out */,
    char*** argv_p /* in/out */);
```

- **MPI_Finalize**
  - Tells MPI we’re done, so clean up anything allocated for this program.

```c
int MPI_Finalize(void);
```
Basic Outline

```c
#include <mpi.h>

int main(int argc, char* argv[]) {
    /* No MPI calls before this */
    MPI_Init(&argc, &argv);
    /* No MPI calls after this */
    MPI_Finalize();
    return 0;
}
```
Communicators

- A collection of processes that can send messages to each other.
- MPI_Init defines a communicator that consists of all the processes created when the program is started.
- Called MPI_COMM_WORLD.
Communicators

```c
int MPI_Comm_size(
    MPI_Comm  comm    /* in */,
    int*      comm_sz_p /* out */);
```

number of processes in the communicator

```c
int MPI_Comm_rank(
    MPI_Comm  comm    /* in */,
    int*      my_rank_p /* out */);
```

my rank

(the process making this call)
SPMD

- Single-Program Multiple-Data
- We compile **one** program.
- Process 0 does something different.
  - Receives messages and prints them while the other processes do the work.

- The **if-else** construct makes our program **SPMD**.
int MPI_Send(

void* msg_buf_p /* in */,
int msg_size /* in */,
MPI_Datatype msg_type /* in */,
int dest /* in */,
int tag /* in */,
MPI_Comm communicator /* in */);

Copyright © 2010, Elsevier Inc. All rights Reserved
## Data types

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>C datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_LONG_LONG</td>
<td>signed long long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>
int MPI_Recv(
    void*         msg_buf_p    /* out */,
    int           buf_size     /* in  */,
    MPI_Datatype  buf_type     /* in  */,
    int           source       /* in  */,
    int           tag          /* in  */,
    MPI_Comm      communicator /* in  */,
    MPI_Status*   status_p    /* out */);

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
</tr>
</tbody>
</table>
status_p argument

MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag, recv_comm, &status);

MPI_Status* status;

status.MPI_SOURCE
status.MPI_TAG
status.MPI_ERROR
Message matching

MPI_Send(src = q, dest = r, send_tag, send_comm);

MPI_Recv(dest = r, src = q, recv_tag, recv_comm, &status);
Receiving messages

- A receiver can get a message without knowing:
  - the amount of data in the message,
  - the sender of the message,
  - or the tag of the message.
How much data am I receiving?

```c
int MPI_Get_count(
    MPI_Status* status_p /* in */,
    MPI_Datatype type    /* in */,
    int* count_p         /* out */);
```
Issues with send and receive

- Exact behavior is determined by the MPI implementation.
- MPI_Send may behave differently with regard to buffer size, cutoffs and blocking.
- MPI_Recv always blocks until a matching message is received.
- Know your implementation; don’t make assumptions!
Dealing with I/O

```c
#include <stdio.h>
#include <mpi.h>

int main(void) {
    int my_rank, comm_sz;

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    printf("Proc %d of %d > Does anyone have a toothpick?\n", my_rank, comm_sz);

    MPI_Finalize();
    return 0;
} /* main */
```

Each process just prints a message.
Running with 6 processes

Proc 0 of 6 > Does anyone have a toothpick?
Proc 1 of 6 > Does anyone have a toothpick?
Proc 2 of 6 > Does anyone have a toothpick?
Proc 4 of 6 > Does anyone have a toothpick?
Proc 3 of 6 > Does anyone have a toothpick?
Proc 5 of 6 > Does anyone have a toothpick?

unpredictable output
Most MPI implementations only allow process 0 in MPI_COMM_WORLD access to stdin.

Process 0 must read the data (scanf) and send to the other processes.

```c
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

Get_data(my_rank, comm_sz, &a, &b, &n);

h = (b-a)/n;
```

Function for reading user input

```c
void Get_input(
    int my_rank /* in */,
    int comm_sz /* in */,
    double* a_p /* out */,
    double* b_p /* out */,
    int* n_p /* out */
) {
    int dest;
    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
        for (dest = 1; dest < comm_sz; dest++) {
            MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
        }
    } else { /* my_rank != 0 */
        MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
    }
} /* Get_input */
```
COLLECTIVE COMMUNICATION
Tree-structured communication

1. In the first phase:
   (a) Process 1 sends to 0, 3 sends to 2, 5 sends to 4, and 7 sends to 6.
   (b) Processes 0, 2, 4, and 6 add in the received values.
   (c) Processes 2 and 6 send their new values to processes 0 and 4, respectively.
   (d) Processes 0 and 4 add the received values into their new values.

2. (a) Process 4 sends its newest value to process 0.
   (b) Process 0 adds the received value to its newest value.
A tree-structured global sum

Processes

0  1  2  3  4  5  6  7
5  2  -1  -3  6  5  -7  2

7  -4  11  -5

3  6

9
An alternative tree-structured global sum
MPI_Reduce

```c
int MPI_Reduce(
    void* input_data_p,  /* in */
    void* output_data_p, /* out */
    int count,            /* in */
    MPI_Datatype datatype,/* in */
    MPI_Op operator,      /* in */
    int dest_process,     /* in */
    MPI_Comm comm,        /* in */
);```

```c
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

```c
double local_x[N], sum[N];
...
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```
Predefined reduction operators in MPI

<table>
<thead>
<tr>
<th>Operation Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical and</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bitwise and</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical or</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bitwise or</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical exclusive or</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Bitwise exclusive or</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum and location of maximum</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Minimum and location of minimum</td>
</tr>
</tbody>
</table>
Collective vs. Point-to-Point Communications

- All the processes in the communicator must call the same collective function.

- For example, a program that attempts to match a call to `MPI_Reduce` on one process with a call to `MPI_Recv` on another process is erroneous, and, in all likelihood, the program will hang or crash.
Collective vs. Point-to-Point Communications

- The arguments passed by each process to an MPI collective communication must be “compatible.”

- For example, if one process passes in 0 as the *dest_process* and another passes in 1, then the outcome of a call to MPI_Reduce is erroneous, and, once again, the program is likely to hang or crash.
Collective vs. Point-to-Point Communications

- The `output_data_p` argument is only used on `dest_process`.

- However, all of the processes still need to pass in an actual argument corresponding to `output_data_p`, even if it’s just `NULL`.
Collective vs. Point-to-Point Communications

- Point-to-point communications are matched on the basis of tags and communicators.
- Collective communications don’t use tags.
- They’re matched solely on the basis of the communicator and the order in which they’re called.
Example (1)

<table>
<thead>
<tr>
<th>Time</th>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>a = 1; c = 2</code></td>
<td><code>a = 1; c = 2</code></td>
<td><code>a = 1; c = 2</code></td>
</tr>
<tr>
<td>1</td>
<td><code>MPI_Reduce(&amp;a, &amp;b, ...)</code></td>
<td><code>MPI_Reduce(&amp;c, &amp;d, ...)</code></td>
<td><code>MPI_Reduce(&amp;a, &amp;b, ...)</code></td>
</tr>
<tr>
<td>2</td>
<td><code>MPI_Reduce(&amp;c, &amp;d, ...)</code></td>
<td><code>MPI_Reduce(&amp;a, &amp;b, ...)</code></td>
<td><code>MPI_Reduce(&amp;c, &amp;d, ...)</code></td>
</tr>
</tbody>
</table>

Multiple calls to MPI_Reduce
Example (2)

- Suppose that each process calls `MPI_Reduce` with operator `MPI_SUM`, and destination process 0.

- At first glance, it might seem that after the two calls to `MPI_Reduce`, the value of b will be 3, and the value of d will be 6.
Example (3)

- However, the names of the memory locations are irrelevant to the matching of the calls to MPI_Reduce.

- The order of the calls will determine the matching so the value stored in b will be 1+2+1 = 4, and the value stored in d will be 2+1+2 = 5.
### MPI_Allreduce

- Useful in a situation in which all of the processes need the result of a global sum in order to complete some larger computation.

```c
int MPI_Allreduce(
    void* input_data_p /* in */,
    void* output_data_p /* out */,
    int count /* in */,
    MPI_Datatype datatype /* in */,
    MPI_Op operator /* in */,
    MPI_Comm comm /* in */);
```
A global sum followed by distribution of the result.
A butterfly-structured global sum.
Broadcast

- Data belonging to a single process is sent to all of the processes in the communicator.

```c
int MPI_Bcast(
    void* data_p,          /* in/out */
    int count,             /* in */
    MPI_Datatype datatype, /* in */
    int source_proc,       /* in */
    MPI_Comm comm          /* in */
);```

A tree-structured broadcast.
A version of Get_input that uses MPI_Bcast

```c
void Get_input(
    int my_rank /* in */,
    int comm_sz /* in */,
    double* a_p /* out */,
    double* b_p /* out */,
    int* n_p /* out */) {

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
    }
    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
} /* Get_input */
```
Data distributions

\[ \mathbf{x} + \mathbf{y} = (x_0, x_1, \ldots, x_{n-1}) + (y_0, y_1, \ldots, y_{n-1}) \]
\[ = (x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1}) \]
\[ = (z_0, z_1, \ldots, z_{n-1}) \]
\[ = z \]

Compute a vector sum.
Serial implementation of vector addition

```c
void Vector_sum(double x[], double y[], double z[], int n) {
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
} /* Vector_sum */
```
Different partitions of a 12-component vector among 3 processes

<table>
<thead>
<tr>
<th>Process</th>
<th>Block</th>
<th>Cyclic</th>
<th>Block-cyclic Blocksize = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1 2</td>
<td>3 0 3 6 9</td>
<td>0 1 6 7</td>
</tr>
<tr>
<td>1</td>
<td>4 5 6</td>
<td>7 1 4 7 10</td>
<td>2 3 8 9</td>
</tr>
<tr>
<td>2</td>
<td>8 9 10 11</td>
<td>2 5 8 11</td>
<td>4 5 10 11</td>
</tr>
</tbody>
</table>
Partitioning options

- **Block partitioning**
  - Assign blocks of consecutive components to each process.

- **Cyclic partitioning**
  - Assign components in a round robin fashion.

- **Block-cyclic partitioning**
  - Use a cyclic distribution of blocks of components.
Parallel implementation of vector addition

```c
void Parallel_vector_sum(
    double local_x[] /* in */,
    double local_y[] /* in */,
    double local_z[] /* out */,
    int    local_n   /* in */)
{
    int local_i;

    for (local_i = 0; local_i < local_n; local_i++)
        local_z[local_i] = local_x[local_i] + local_y[local_i];
} /* Parallel_vector_sum */
```
Scatter

- MPI_Scatter can be used in a function that reads in an entire vector on process 0 but only sends the needed components to each of the other processes.

```c
int MPI_Scatter(
    void* send_buf_p /* in */,
    int send_count /* in */,
    MPI_Datatype send_type /* in */,
    void* recv_buf_p /* out */,
    int recv_count /* in */,
    MPI_Datatype recv_type /* in */,
    int src_proc /* in */,
    MPI_Comm comm /* in */);
```
Reading and distributing a vector

```c
void Read_vector(
    double local_a[] /* out */,
    int local_n /* in */,
    int n /* in */,
    char vec_name[] /* in */,
    int my_rank /* in */,
    MPI_Comm comm /* in */) {

    double* a = NULL;
    int i;

    if (my_rank == 0) {
        a = malloc(n*sizeof(double));
        printf("Enter the vector %s\n", vec_name);
        for (i = 0; i < n; i++)
            scanf("%lf", &a[i]);
        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
                    0, comm);
        free(a);
    } else {
        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
                    0, comm);
    }
} /* Read_vector */
```
Gather

- Collect all of the components of the vector onto process 0, and then process 0 can process all of the components.

```c
int MPI_Gather(
    void* send_buf_p  /* in */,
    int send_count /* in */,
    MPI_Datatype send_type /* in */,
    void* recv_buf_p /* out */,
    int recv_count /* in */,
    MPI_Datatype recv_type /* in */,
    int dest_proc /* in */,
    MPI_Comm comm /* in */);
```
Print a distributed vector (1)

```c
void Print_vector(
    double local_b[] /* in */,
    int local_n /* in */,
    int n /* in */,
    char title[] /* in */,
    int my_rank /* in */,
    MPI_Comm comm /* in */) {

    double* b = NULL;
    int i;
```
if (my_rank == 0) {
    b = malloc(n*sizeof(double));
    MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE, 0, comm);
    printf("%s\n", title);
    for (i = 0; i < n; i++)
        printf("%f ", b[i]);
    printf("\n");
    free(b);
} else {
    MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE, 0, comm);
}
/* Print_vector */
Allgather

- Concatenates the contents of each process’ `send_buf_p` and stores this in each process’ `recv_buf_p`.
- As usual, `recv_count` is the amount of data being received from each process.

```c
int MPI_Allgather(
    void* send_buf_p  /* in */,
    int send_count    /* in */,
    MPI_Datatype send_type  /* in */,
    void* recv_buf_p  /* out */,
    int recv_count    /* in */,
    MPI_Datatype recv_type  /* in */,
    MPI_Comm comm     /* in */);
```
Matrix-vector multiplication

\[ A = (a_{ij}) \text{ is an } m \times n \text{ matrix} \]

\[ x \text{ is a vector with } n \text{ components} \]

\[ y = Ax \text{ is a vector with } m \text{ components} \]

\[ y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{i,n-1}x_{n-1} \]

i-th component of y

Dot product of the ith row of A with x.
Matrix-vector multiplication

\[
\begin{bmatrix}
a_{00} & a_{01} & \cdots & a_{0,n-1} \\
a_{10} & a_{11} & \cdots & a_{1,n-1} \\
\vdots & \vdots & \ddots & \vdots \\
a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
\vdots & \vdots & \ddots & \vdots \\
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1}
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
\vdots \\
x_{n-1}
\end{bmatrix}
= 
\begin{bmatrix}
y_0 \\
y_1 \\
\vdots \\
y_{m-1}
\end{bmatrix}
\]

\[
y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}
\]

Copyright © 2010, Elsevier Inc. All rights Reserved
Multiply a matrix by a vector

/* For each row of A */
for (i = 0; i < m; i++) {
   /* Form dot product of ith row with x */
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i][j]*x[j];
}

Serial pseudo-code
C style arrays

\[
\begin{pmatrix}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11
\end{pmatrix}
\]

stored as

0 1 2 3 4 5 6 7 8 9 10 11
Serial matrix-vector multiplication

```c
void Mat_vec_mult(
    double A[] /* in */,
    double x[] /* in */,
    double y[] /* out */,
    int m /* in */,
    int n /* in */) {
    int i, j;

    for (i = 0; i < m; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i*n+j]*x[j];
    }
} /* Mat_vec_mult */
```
void Mat_vect_mult(
    double local_A[] /* in */,
    double local_x[] /* in */,
    double local_y[] /* out */,
    int local_m /* in */,
    int n /* in */,
    int local_n /* in */,
    MPI_Comm comm /* in */) {

double* x;
int local_i, j;
int local_ok = 1;
An MPI matrix-vector multiplication function (2)

```c
x = malloc(n*sizeof(double));
MPI_Allgather(local_x, local_n, MPI_DOUBLE,
              x, local_n, MPI_DOUBLE, comm);

for (local_i = 0; local_i < local_m; local_i++) {
    local_y[local_i] = 0.0;
    for (j = 0; j < n; j++)
        local_y[local_i] += local_A[local_i*n+j]*x[j];
}
free(x);

/* Mat_vect_mult */
```
PERFORMANCE EVALUATION
Elapsed parallel time

- Returns the number of seconds that have elapsed since some time in the past.

```c
double MPI_Wtime(void);

double start, finish;
...
start = MPI_Wtime();
/* Code to be timed */
...
finish = MPI_Wtime();
printf("Proc %d > Elapsed time = %e seconds\n"  
   my_rank, finish-start);
```
Elapsed serial time

- In this case, you don’t need to link in the MPI libraries.
- Returns time in microseconds elapsed from some point in the past.

```c
#include "timer.h"

...  
double now;
...  
GET_TIME(&now);
```
Elapsed serial time

```c
#include "timer.h"

    double  start, finish;
    ...
    GET_TIME(start);
    /* Code to be timed */
    ...
    GET_TIME(finish);
    printf("Elapsed time = %e seconds\n", finish-start);
```
MPI_Barrier

- Ensures that no process will return from calling it until every process in the communicator has started calling it.

```c
int MPI_Barrier(MPI_Comm comm); /* in */
```
double local_start, local_finish, local_elapsed, elapsed;
...
MPI_BARRIER(comm);
local_start = MPI_Wtime();
/* Code to be timed */
...
local_finish = MPI_Wtime();
local_elapsed = local_finish - local_start;
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE,
            MPI_MAX, 0, comm);

if (my_rank == 0)
    printf("Elapsed time = %e seconds\n", elapsed);
Run-times of serial and parallel matrix-vector multiplication

<table>
<thead>
<tr>
<th>comm_sz</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
<th>16,384</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.1</td>
<td>16.0</td>
<td>64.0</td>
<td>270</td>
<td>1100</td>
</tr>
<tr>
<td>2</td>
<td>2.3</td>
<td>8.5</td>
<td>33.0</td>
<td>140</td>
<td>560</td>
</tr>
<tr>
<td>4</td>
<td>2.0</td>
<td>5.1</td>
<td>18.0</td>
<td>70</td>
<td>280</td>
</tr>
<tr>
<td>8</td>
<td>1.7</td>
<td>3.3</td>
<td>9.8</td>
<td>36</td>
<td>140</td>
</tr>
<tr>
<td>16</td>
<td>1.7</td>
<td>2.6</td>
<td>5.9</td>
<td>19</td>
<td>71</td>
</tr>
</tbody>
</table>

(Seconds)
Speedup

\[ S(n, p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n, p)} \]
Efficiency

\[ E(n, p) = \frac{S(n, p)}{p} = \frac{T_{\text{serial}}(n)}{p \times T_{\text{parallel}}(n, p)} \]
# Speedups of Parallel Matrix-Vector Multiplication

<table>
<thead>
<tr>
<th>comm_sz</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
<th>16,384</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>1.8</td>
<td>1.9</td>
<td>1.9</td>
<td>1.9</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>2.1</td>
<td>3.1</td>
<td>3.6</td>
<td>3.9</td>
<td>3.9</td>
</tr>
<tr>
<td>8</td>
<td>2.4</td>
<td>4.8</td>
<td>6.5</td>
<td>7.5</td>
<td>7.9</td>
</tr>
<tr>
<td>16</td>
<td>2.4</td>
<td>6.2</td>
<td>10.8</td>
<td>14.2</td>
<td>15.5</td>
</tr>
</tbody>
</table>
 Efficiencies of Parallel Matrix-Vector Multiplication

<table>
<thead>
<tr>
<th>comm_sz</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
<th>16,384</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.89</td>
<td>0.94</td>
<td>0.97</td>
<td>0.96</td>
<td>0.98</td>
</tr>
<tr>
<td>4</td>
<td>0.51</td>
<td>0.78</td>
<td>0.89</td>
<td>0.96</td>
<td>0.98</td>
</tr>
<tr>
<td>8</td>
<td>0.30</td>
<td>0.61</td>
<td>0.82</td>
<td>0.94</td>
<td>0.98</td>
</tr>
<tr>
<td>16</td>
<td>0.15</td>
<td>0.39</td>
<td>0.68</td>
<td>0.89</td>
<td>0.97</td>
</tr>
</tbody>
</table>
Scalability

- A program is **scalable** if the problem size can be increased at a rate so that the efficiency doesn’t decrease as the number of processes increase.
Scalability

- Programs that can maintain a constant efficiency without increasing the problem size are sometimes said to be **strongly scalable**.

- Programs that can maintain a constant efficiency if the problem size increases at the same rate as the number of processes are sometimes said to be **weakly scalable**.
Concluding Remarks (1)

- MPI or the Message-Passing Interface is a library of functions that can be called from C, C++, or Fortran programs.
- A communicator is a collection of processes that can send messages to each other.
- Many parallel programs use the single-program multiple data or SPMD approach.
Concluding Remarks (2)

- Most serial programs are deterministic: if we run the same program with the same input we’ll get the same output.
- Parallel programs often don’t possess this property.
- Collective communications involve all the processes in a communicator.
Concluding Remarks (3)

- When we time parallel programs, we’re usually interested in elapsed time or “wall clock time”.
- Speedup is the ratio of the serial run-time to the parallel run-time.
- Efficiency is the speedup divided by the number of parallel processes.
If it’s possible to increase the problem size (n) so that the efficiency doesn’t decrease as p is increased, a parallel program is said to be scalable.