Collective Communication in MPI and Advanced Features

Pacheco’s book. Chapter 3

T. Yang, CS240A. Part of slides from the text book, CS267 K. Yelick from UC Berkeley and B. Gropp, ANL
Outline

• Collective group communication
• Application examples
  ▪ Pi computation
  ▪ Summation of long vectors
• More applications
  ▪ Matrix-vector multiplication
    – performance evaluation
  ▪ Parallel sorting
• Safety and other MPI issues.
What MPI Functions are commonly used

For simple applications, these are common:

- **Startup**
  - MPI_Init, MPI_Finalize

- **Information on the processes**
  - MPI_Comm_rank, MPI_Comm_size, MPI_Get_processor_name

- **Point-to-point communication**
  - MPI_Irecv, MPI_Isend, MPI_Wait, MPI_Send, MPI_Recv

- **Collective communication**
  - MPI_Allreduce, MPI_Bcast, MPI_Allgather

http://mpitutorial.com/mpi-broadcast-and-collective-communication/
MPI Collective Communication

- **Collective routines** provide a higher-level way to organize a parallel program
  - Each process executes the same communication operations
  - Communication and computation is coordinated among a group of processes in a communicator
  - Tags are not used
  - No non-blocking collective operations.

- **Three classes of operations:** synchronization, data movement, collective computation.
Synchronization

- **MPI_Barrier(comm)**
- Blocks until all processes in the group of the communicator `comm` call it.
- Not used often. Sometime used in measuring performance and load balancing.
Collective Data Movement: Broadcast, Scatter, and Gather

Broadcast

P0  A
P1  A
P2  A
P3  A

Scatter

P0  A B C D
P1  A
P2  B
P3  C

Gather

P0  A
P1  A
P2  B
P3  C
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 */);
```
Comments on Broadcast

- All collective operations must be called by *all* processes in the communicator.
- MPI_Bcast is called by both the sender (called the root process) and the processes that are to receive the broadcast.
  - MPI_Bcast is not a “multi-send.”
  - “root” argument is the rank of the sender; this tells MPI which process originates the broadcast and which receive.
Implementation View: A tree-structured broadcast of a number 6 from Process 0
A version of `Get_input` that uses `MPI_Bcast` in the trapezoidal program

```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
");
        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 */
```
Collective Data Movement: Allgather and AlltoAll

<table>
<thead>
<tr>
<th>P0</th>
<th>A</th>
<th></th>
<th></th>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>P2</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>P3</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

Allgather

<table>
<thead>
<tr>
<th>P0</th>
<th>A0 A1 A2 A3</th>
<th>A0</th>
<th>B0</th>
<th>C0</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>B0 B1 B2 B3</td>
<td>A1</td>
<td>B1</td>
<td>C1</td>
<td>D1</td>
</tr>
<tr>
<td>P2</td>
<td>C0 C1 C2 C3</td>
<td>A2</td>
<td>B2</td>
<td>C2</td>
<td>D2</td>
</tr>
<tr>
<td>P3</td>
<td>D0 D1 D2 D3</td>
<td>A3</td>
<td>B3</td>
<td>C3</td>
<td>D3</td>
</tr>
</tbody>
</table>
Collective Computation: Reduce vs. Scan

P0  
    A  
P1  
    B  
P2  
    C  
P3  
    D  

\[ \text{Reduce} \]

\[ R(ABCD) \]

P0  
    A  
P1  
    B  
P2  
    C  
P3  
    D  

\[ \text{Scan} \]

\[ R(A) \]
\[ R(AB) \]
\[ R(ABC) \]
\[ R(ABCD) \]
MPI_Reduce

Before MPI_Reduce

Process 1  2  3  4

After MPI_Reduce

Process 1

10

Process 2

Process 3

Process 4

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 */);

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

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>
Implementation View of Global Reduction using a tree-structured sum
Implementation view: Tree-structured global reduction with sum operator

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.
An alternative tree-structured global sum

Processes

0 1 2 3 4 5 6 7

5 2 -1 -3 6 5 -7 2

11 7 -8 -1

3 6

9
MPI Scan

MPI_Scan( void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm );
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
    MPI_Datatype datatype
    MPI_Op operator
    MPI_Comm comm
);
```
A global sum followed by distribution of the result.
A butterfly-structured global sum.
MPI Collective Routines: Summary

• Many Routines: Allgatherv, Allgatherv, Allreduce, Alltoall, Alltoallv, Bcast, Gather, Gatherv, Reduce, Reduce_scatter, Scan, Scatter, Scatterv

• All versions deliver results to all participating processes.

• V versions allow the hunks to have variable sizes.

• Allreduce, Reduce, Reduce_scatter, and Scan take both built-in and user-defined combiner functions.

• MPI-2 adds Alltoallw, Exscan, intercommunicator versions of most routines
Example of MPI PI program using 6 Functions

\[ \pi = 4 \int_{0}^{1} \frac{1}{1 + x^2} \, dx \]

- **Using basic MPI functions:**
  - `MPI_INIT`
  - `MPI_FINALIZE`
  - `MPI_COMM_SIZE`
  - `MPI_COMM_RANK`

- **Using MPI collectives:**
  - `MPI_BCAST`
  - `MPI_REDUCE`

Slide source: Bill Gropp, ANL
Midpoint Rule for

\[ \pi = 4 \int_0^1 \frac{1}{1 + x^2} \, dx \]

\[ \int_a^b f(x) \, dx \approx (b - a) f(x_m) \]

\[ \int_{x=0}^1 \frac{1}{1 + x^2} \approx \sum_{i=1}^{n} \frac{1}{1 + \left( \frac{i-0.5}{n} \right)^2} \]
Example: PI in C - 1

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

int main(int argc, char *argv[]) {
    int done = 0, n, myid, numprocs, i, rc;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, a;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    while (!done) {
        if (myid == 0) {
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d", &n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0) break;
    }
    
    Input and broadcast parameters
```
Example: PI in C - 2

\[ \int_{x=0}^{1} \frac{1}{1 + x^2} \approx \sum_{i=1}^{n} \frac{1}{1 + \left(\frac{i-0.5}{n}\right)^2} \]

\[
h = \frac{1.0}{\text{(double) } n}; \quad \text{Compute local pi values}
\]
\[
\text{sum} = 0.0;
\]
\[
\text{for} \ (i = \text{myid} + 1; \ i \leq n; \ i += \text{numprocs}) \ {\}
\]
\[
\quad x = h \times ((\text{double})i - 0.5);
\]
\[
\quad \text{sum} += 4.0 / (1.0 + x \times x);
\]
\[
\text{mypi} = h \times \text{sum};
\]
\[
\text{MPI\_Reduce(&mypi, &pi, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD); \quad \text{Compute summation}
\]
\[
\text{if} \ (\text{myid} == 0)
\]
\[
\quad \text{printf("pi is approximately %.16f, Error is .16f
",}
\]
\[
\quad \quad \pi, \text{fabs(pi - PI25DT))} ;
\]
\[
}\]
\[
\text{MPI\_Finalize();}
\]
\[
\text{return} \ 0;
\]
Collective vs. Point-to-Point Communications

- **All** the processes in the communicator must call the same collective function.
  - Will this program work?

```c
if(my_rank==0) MPI_Reduce(&a,&b,1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
else MPI_Recv(&a, MPI_INT, MPI_SUM,0,0, MPI_COMM_WORLD);
```
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.

```c
if(my_rank==0) MPI_Reduce(&a,&b,1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
else MPI_Recv(&a, MPI_INT, MPI_SUM,0,0, MPI_COMM_WORLD);
```
The arguments passed by each process to an MPI collective communication must be “compatible.”

- Will this program work?

```c
if(my_rank==0) MPI_Reduce(&a,&b,1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
else MPI_Reduce(&a,&b,1, MPI_INT, MPI_SUM, 1, MPI_COMM_WORLD);
```
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.

```c
if(my_rank==0)  MPI_Reduce(&a,&b,1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
else MPI_Reduce(&a,&b,1, MPI_INT, MPI_SUM, 1, MPI_COMM_WORLD);
```
Example of MPI_Reduce execution

Multiple calls to MPI_Reduce with MPI_SUM and Proc 0 as destination (root)

Is b=3 on Proc 0 after two MPI_Reduce() calls?
Is d=6 on Proc 0?

```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 */
);
```
Example: Output results

- 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 $2+1+2 = 5$.

<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>$a = 1; c = 2$</td>
<td>$a = 1; c = 2$</td>
<td>$a = 1; c = 2$</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>
Parallel Matrix Vector Multiplication

Collective Communication Application
Textbook p. 113-116
Matrix-vector multiplication: \( y = A \times x \)

\[
\begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
\end{pmatrix} \times 
\begin{pmatrix}
1 \\
2 \\
3 \\
\end{pmatrix} = 
\begin{pmatrix}
1 \times 1 + 2 \times 2 + 3 \times 3 \\
4 \times 1 + 5 \times 2 + 6 \times 3 \\
7 \times 1 + 8 \times 2 + 9 \times 3 \\
\end{pmatrix} = 
\begin{pmatrix}
14 \\
32 \\
50 \\
\end{pmatrix}
\]

Problem: \( y = A \times x \) where \( A \) is a \( n \times n \) matrix and \( x \) is a column vector of dimension \( n \).

Sequential code:

```
for i = 1 to n do
    y_i = 0;
    for j = 1 to n do
        y_i = y_i + a_{i,j} \times x_j;
    endfor
endfor
```
Partitioning and Task graph for matrix-vector multiplication

Partitioned code:

```plaintext
for i = 1 to n do
    \( S_i : \quad y_i = 0; \)
    for j = 1 to n do
        \( y_i = y_i + a_{i,j} \times x_j; \)
    endfor
endfor
```

\( S_i : \quad \text{Read row } A_i \text{ and vector } x. \)
Write element \( y_i \)

\( y_i = \text{Row } A_i \times x \)

Task graph:

- S1
- S2
- S3
- Sn
Execution Schedule and Task Mapping

\( S_i : \) Read row \( A_i \) and vector \( x \).

Write element \( y_i \)

Task graph:

\[
y_i = \text{Row } A_i \times x
\]

Schedule:

\[
\begin{array}{ccc}
S1 & S2 & S3 & \ldots & Sn \\
0 & 1 & \ldots & p-1 \\
S1 & S_r+1 & S2r & \ldots & Sn \\
S2 & S_r+2 & S2r & \ldots & Sn \\
S_r & S2r & S2r & \ldots & Sn \\
\end{array}
\]

Mapping function of tasks \( S_i \):

\[
\text{proc}_\text{map}(i) = \left\lfloor \frac{i-1}{r} \right\rfloor \text{ where } r = \left\lfloor \frac{n}{p} \right\rfloor.
\]
Data Partitioning and Mapping for $y = A^*x$

**Data partitioning:** for the above schedule:

Matrix $A$ is divided into $n$ rows $A_1, A_2, \cdots A_n$.

![Diagram showing data partitioning]

**Data mapping:**

Row $A_i$ is mapped to processor $proc\_map(i)$, the same as task $i$. The indexing function is:

$local(i) = (i - 1) \mod r$. Vectors $x$ and $y$ are replicated to all processors.
SPMD Code for $y = A^x$

int $x[n]$, $y[n]$, $a[r][n]$;
me=mynode();

for $i = 1$ to $n$ do
    if $proc\_map(i) == me$, then do $S_i$:
        $S_i : \; y[i] = 0$;
        for $j = 1$ to $n$ do
            $y[i] = y[i] + a[local(i)][j] * x[j]$;
        endfor
    endfor
endfor
Evaluation: Parallel Time

• Ignore the cost of local address calculation.

• Each task performs $n$ additions and $n$ multiplications.

• Each addition/multiplication costs $\omega$

• The parallel time is approximately $\frac{n}{p} \times 2n\omega$
How is initial data distributed?

Assume initially matrix $A$ and vector $x$ are distributed evenly among processes

Need to redistribute vector $x$ to everybody in order to perform parallel computation!

What MPI collective communication is needed?
Communication Pattern for Data Redistribution

Data requirement for Process 0

![Diagram showing data requirement for Process 0]

Data requirement for all processes

![Diagram showing data requirement for all processes]
MPI Code for Gathering Data

Data gather for Process 0

float local_x[]; /* local storage for x*/
float global_x[]; /* storage for all of x*/

MPI_Gather(local_x, n/p, MPI_FLOAT,
          global_x, n/p, MPI_FLOAT,
          0, MPI_COMM_WORLD);

Repeat for all processes

It is the same as:

MPI_All_gather(local_x, n/p, MPI_FLOAT,
               global_x, n/p, MPI_FLOAT,
               MPI_COMM_WORLD);
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 */);
```
MPI SPMD Code for $y=A^*x$

```c
void Parallel_matrix_vector_prod(
    LOCAL_MATRIX_T local_A
    int m
    int n
    float local_x[]
    float global_x[]
    float local_y[]
    int local_m
    int local_n)
{
    /* local_m = n/p, local_n = n/p */
    MPI_Allgather(local_x, local_n, MPI_FLOAT
                  global_x, local_n, MPI_FLOAT, MPI_COMM_WORLD);
```
MPI SPMD Code for y=A*x

```c
for (i = 0; i < local_m; i++) {
    local_y[i] = 0.0;
    for (j = 0; j < n; j++)
        local_y[i] = local_y[i] +
                    local_A[i][j]*global_x[j];
}
```
Performance Evaluation of Matrix Vector Multiplication
How to measure elapsed parallel time

• Use MPI_Wtime() that returns the number of seconds that have elapsed since some time in the past.

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);
Measure elapsed sequential time in Linux

- This code works for Linux without using MPI functions
- Use GET_TIME() which returns time in microseconds elapsed from some point in the past.

- Sample code for GET_TIME()

```c
#include <sys/time.h>
/* The argument now should be a double (not a pointer to a double) */
define GET_TIME(now) {
    struct timeval t;
    gettimeofday(&t, NULL);
    now = t.tv_sec + t.tv_usec/1000000.0;
}
```
Measure elapsed sequential time

```
#include "timer.h"

...  

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

Start timing until every process in the communicator has reached the same time stamp

```c
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>Order of Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1024</td>
</tr>
<tr>
<td>1</td>
<td>4.1</td>
</tr>
<tr>
<td>2</td>
<td>2.3</td>
</tr>
<tr>
<td>4</td>
<td>2.0</td>
</tr>
<tr>
<td>8</td>
<td>1.7</td>
</tr>
<tr>
<td>16</td>
<td>1.7</td>
</tr>
</tbody>
</table>

(Seconds)
Speedup and Efficiency

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

\[ E(n, p) = \frac{S(n, p)}{p} = \frac{T_{\text{serial}}(n)}{p \times T_{\text{parallel}}(n, p)} \]
## 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.

• 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**.

•
Safety Issues in MPI programs
Safety in MPI programs

- Is it a safe program? (Assume tag/process ID is assigned properly)

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Send(1)</code></td>
<td><code>Send(0)</code></td>
</tr>
<tr>
<td><code>Recv(1)</code></td>
<td><code>Recv(0)</code></td>
</tr>
</tbody>
</table>

MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz, 0, comm, MPI_STATUS_IGNORE).
Safety in MPI programs

• Is it a safe program? (Assume tag/process ID is assigned properly)

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(1)</td>
<td>Send(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Recv(0)</td>
</tr>
</tbody>
</table>

• May be unsafe because MPI standard allows MPI_Send to behave in two different ways:
  ▪ it can simply copy the message into an MPI managed buffer and return,
  ▪ or it can block until the matching call to MPI_Recv starts.

Copyright © 2010, Elsevier Inc. All rights Reserved
Buffer a message implicitly during MPI_Send()

- When you send data, where does it go? One possibility is:
  
  Process 0
  
  User data
  
  Local buffer
  
  the network

  Process 1
  
  Local buffer
  
  User data
Avoiding Buffering

- Avoiding copies uses less memory
- May use more time

MPI_Send() waits until a matching receive is executed.

Slide source: Bill Gropp, ANL
Safety in MPI programs

- Many implementations of MPI set a threshold at which the system switches from buffering to blocking.
  - Relatively small messages will be buffered by MPI_Send.
  - Larger messages, will cause it to block.

If the MPI_Send() executed by each process blocks, no process will be able to start executing a call to MPI_Recv, and the program will hang or deadlock.

- Each process is blocked waiting for an event that will never happen.
Example of unsafe MPI code with possible deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)

```
Process 0            Process 1

Send(1)              Send(0)
Recv(1)              Recv(0)
```

- This may be “unsafe” because it depends on the availability of system buffers in which to store the data sent until it can be received
Safety in MPI programs

• A program that relies on MPI provided buffering is said to be **unsafe**.

• Such a program may run without problems for various sets of input, but it may hang or crash with other sets.
How can we tell if a program is unsafe

- Replace MPI_Send() with MPI_Ssend()
- The extra “s” stands for synchronous and MPI_Ssend is guaranteed to block until the matching receive starts.
- If the new program does not hang/crash, the original program is safe.
- MPI_Send() and MPI_Ssend() have the same arguments

```c
int MPI_Ssend(
    void* msg_buf_p /* in */,
    int msg_size /* in */,
    MPI_Datatype msg_type /* in */,
    int dest /* in */,
    int tag /* in */,
    MPI_Comm communicator /* in */);
```
Some Solutions to the “unsafe” Problem

- Order the operations more carefully:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(1)</td>
<td>Recv(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Send(0)</td>
</tr>
</tbody>
</table>

- Simultaneous send and receive in one call

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sendrecv(1)</td>
<td>Sendrecv(0)</td>
</tr>
</tbody>
</table>

Slide source: Bill Gropp, ANL
Restructuring communication in odd-even sort

MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,
0, comm, MPI_STATUS_IGNORE.

if (my_rank % 2 == 0) {
MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,
0, comm, MPI_STATUS_IGNORE.
} else {
MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,
0, comm, MPI_STATUS_IGNORE.
MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
}
Use MPI_Sendrecv() to conduct a blocking send and a receive in a single call.

```c
int MPI_Sendrecv(
    void* send_buf_p /* in */,
    int send_buf_size /* in */,
    MPI_Datatype send_buf_type /* in */,
    int dest /* in */,
    int send_tag /* in */,
    void* recv_buf_p /* out */,
    int recv_buf_size /* in */,
    MPI_Datatype recv_buf_type /* in */,
    int source /* in */,
    int recv_tag /* in */,
    MPI_Comm communicator /* in */,
    MPI_Status* status_p /* in */);
```
More Solutions to the “unsafe” Problem

- Supply own space as buffer for send

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bsend(1)</td>
<td>Bsend(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Recv(0)</td>
</tr>
</tbody>
</table>

- Use non-blocking operations:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isend(1)</td>
<td>Isend(0)</td>
</tr>
<tr>
<td>Irecv(1)</td>
<td>Irecv(0)</td>
</tr>
<tr>
<td>Waitall</td>
<td>Waitall</td>
</tr>
</tbody>
</table>
• MPI works in C, C++, or Fortran.
• A communicator is a collection of processes that can send messages to each other.
• Many parallel programs use the SPMD approach.
• 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 (2)

• **Performance evaluation**
  - Use elapsed time or “wall clock time”.
  - Speedup = sequential/parallel time
  - Efficiency = Speedup/ \( p \)
  - 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.

• **An MPI program is unsafe if its correct behavior depends on the fact that MPI_Send is buffering its input.**