Collective Communication in MPI and Advanced Features

Pacheco. Chapter 3

T. Yang, CS140 2014

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

• Collective group communication
• Examples
• Advanced features
• **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

<table>
<thead>
<tr>
<th></th>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Broadcast</strong></td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>P0</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Gather</strong></td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>P0</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>P1</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td></td>
<td></td>
<td>D</td>
</tr>
</tbody>
</table>
• 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\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: Allgater and AlltoAll

Allgater

P0
P1
P2
P3

A
B
C
D

A
B
C
D

P0
P1
P2
P3

A0 A1 A2 A3
B0 B1 B2 B3
C0 C1 C2 C3
D0 D1 D2 D3

Alltoall

A0 B0 C0 D0
A1 B1 C1 D1
A2 B2 C2 D2
A3 B3 C3 D3
Collective Computation: Reduce vs. Scan

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

**Reduce**

- Reduce operation involves combining data from multiple processors into a single result. Here, the operations `R(ABCD)`, `R(ABC)`, `R(AB)`, and `R(A)` represent the cumulative results from each processor.

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

**Scan**

- Scan operation involves sorting or aggregating data sequentially. Here, the operations `R(A)`, `R(AB)`, `R(ABC)`, and `R(ABCD)` represent the cumulative results from each processor.
MPI_Reduce

Before MPI_Reduce

Process 1
1

Process 2
2

Process 3
3

Process 4
4

After MPI_Reduce

Process 1
10

Process 2

Process 3

Process 4

```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 */);
```

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
An alternative tree-structured global sum
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, /* in */
    MPI_Datatype datatype, /* in */
    MPI_Op operator, /* in */
    MPI_Comm comm /* in */) ;
```
A global sum followed by distribution of the result.
MPI Collective Routines: Summary

- Many Routines: Allgather, 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
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

    Slide source: Bill Gropp, ANL

\[
\pi = 4 \int_0^1 \frac{1}{1 + x^2} \, dx
\]
Example: PI in C - 2

```c
h   = 1.0 / (double) n;   // Compute local pi values
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
}
mypi = h * sum;

MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);  // Compute summation
if (myid == 0)
    printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
}
MPI_Finalize();
return 0;
```
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);
```
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

<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>MPI_Reduce(&amp;a, &amp;b, ...)</td>
<td>MPI_Reduce(&amp;c, &amp;d, ...)</td>
<td>MPI_Reduce(&amp;a, &amp;b, ...)</td>
</tr>
<tr>
<td>2</td>
<td>MPI_Reduce(&amp;c, &amp;d, ...)</td>
<td>MPI_Reduce(&amp;a, &amp;b, ...)</td>
<td>MPI_Reduce(&amp;c, &amp;d, ...)</td>
</tr>
</tbody>
</table>

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?
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 $2+1+2 = 5$. 
Summation of Two Long Vectors

Collective Communication Application
Textbook p.109-111
Application Example: Distributed Summation of Two Long Vectors

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

Sequential code for computing a vector sum.

```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 */
```
Parallel implementation of vector addition

1. Divide each vector into $n$ subvectors and distribute data
2. Add two subvectors at each process in parallel.
3. Gather the sum subvector from each process.

Read two input vectors

Add #1 subvectors

Add #2 subvectors

... Add #n subvectors

Gather subvectors
Partitioning options for distributing vectors

• 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.
Each vector is divided into 12 subvectors and then distributed to 3 processes.

<table>
<thead>
<tr>
<th>Process</th>
<th>Block</th>
<th>Components</th>
<th>Block-cyclic Blocksize = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1 2 3</td>
<td>0 3 6 9</td>
<td>0 1 6 7</td>
</tr>
<tr>
<td>1</td>
<td>4 5 6 7</td>
<td>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>
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 */
• 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 */
);  
```
void Read_vector(
    double local_a[] /* out */,
    int local_n /* in */,
    int n /* in */,
    char vec_name[] /* in */,
    int my_rank /* in */,
    MPI_Comp comm /* in */) {

double* a = NULL;
int i;

if (my_rank == 0) {
    a = malloc(n*sizeof(double));
    printf("Enter the vector \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 data from everybody

- 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 */);
```
MPI code to gather and print a distributed vector

```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;
```
```c
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 */
```
• 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 */);
```
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:

\[
\text{for } i = 1 \text{ to } n \text{ do } \\
\quad S_i : \quad y_i = 0; \\
\quad \quad \text{for } j = 1 \text{ to } n \text{ do } \\
\quad \quad \quad y_i = y_i + a_{i,j} \times x_j; \\
\quad \quad \text{endfor} \\
\quad \text{endfor}
\]

\[S_i : \text{ Read row } A_i \text{ and vector } x.\]

Write element \( y_i \)

Task graph:

\[S_1, S_2, S_3, \ldots, S_n\]
Execution Schedule and Task Mapping

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

Write element \( y_i \)

Task graph:

\begin{center}
\begin{tabular}{ccc}
S1 & S2 & S3 \\
\hline
S1 & Sr+1 & Sr+2 \\
S2 & & S2r \\
Sr & & Sn \\
\end{tabular}
\end{center}

Schedule:

\begin{center}
\begin{tabular}{cccc}
0 & 1 & \ldots & p-1 \\
\hline
S1 & S2 & \ldots & Sn \\
\end{tabular}
\end{center}

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 $A_1, A_2, \cdots A_n$.

![Local space partitioning diagram]

**Data mapping:**

Row $A_i$ is mapped to processor $\text{proc}_\text{map}(i)$, the same as task $i$. The indexing function is:

$\text{local}(i) = (i - 1) \mod r$. Vectors $x$ and $y$ are replicated to all processors.
int x[n], y[n], a[r][n];
me=mynode();
for $i = 1$ to $n$ do
    if $\text{proc\_map}(i) == me$, then do $S_i$:
        $S_i$ : $y[i] = 0$
        for $j = 1$ to $n$ do
            $y[i] = y[i] + a[\text{local}(i)][j] \times x[j];$
        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

Data requirement for other processes

MPI_Gather

MPI_Allgather
MPI Code for Gathering Data

Data gather for Process 0

```c
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:

```c
MPI_All_gather(local_x, n/p, MPI_FLOAT,
               global_x, n/p, MPI_FLOAT,
               MPI_COMM_WORLD);
```
MPI Code for $y=A\times x$

```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 Code for \( y = A \times x \)

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

\( \mathbf{x} \) is a vector with \( n \) components

\[
\begin{array}{cccc}
a_{00} & a_{01} & \cdots & a_{0,n-1} \\
a_{10} & a_{11} & \cdots & a_{1,n-1} \\
\vdots & \vdots & \ddots & \vdots \\
a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
\vdots & \vdots & \ddots & \vdots \\
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\end{array}
\]

\[
\begin{array}{cccc}
x_0 \\
x_1 \\
\vdots \\
x_{n-1} \\
\end{array}
\]

\[
\begin{array}{cccc}
\mathbf{y}_0 \\
\mathbf{y}_1 \\
\vdots \\
\mathbf{y}_{m-1} \\
\end{array}
\]

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

\( i \)-th component of \( \mathbf{y} \)

Dot product of the \( i \)-th row of \( \mathbf{A} \) with \( \mathbf{x} \).
Use one dimensional C array to represent 2D matrix

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

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

stored as

0 1 2 3 4 5 6 7 8 9 10 11
Sequential code for $y = Ax$

```c
void Mat_vect_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_vect_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;
Textbook MPI code for $y = A \times x$

```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);
```

Diagram:

```
   A
```

```
   x
```

```
Proc 0
Proc 1
Proc 2
Proc 3
```
PERFORMANCE EVALUATION
How to measure elapsed parallel time

- Use MPI_Wtime() that 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);
```
Measure elapsed sequential time

- In this case, you don’t need to link in the MPI libraries.
- 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

```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);
```
Use of MPI_Barrier before time measurement

• 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 */);
```
Example of using MPI_Barrier

```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>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 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>( \text{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>

Copyright © 2010, Elsevier Inc. All rights Reserved
### Efficiencies of 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>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.89</td>
</tr>
<tr>
<td>4</td>
<td>0.51</td>
</tr>
<tr>
<td>8</td>
<td>0.30</td>
</tr>
<tr>
<td>16</td>
<td>0.15</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**.
A PARALLEL SORTING ALGORITHM

Textbook p. 127-136
Sorting

- \( n \) keys and \( p = \# \) of processes.
- \( n/p \) keys assigned to each process.
- When the algorithm terminates:
  - The keys assigned to each process should be sorted in (say) increasing order.
  - If \( 0 \leq q < r < p \), then each key assigned to process \( q \) should be less than or equal to every key assigned to process \( r \).
Simple bubble sort with pairwise swaps

```c
void Bubble_sort(
    int a[] /* in/out */,
    int n /* in */
) {
    int list_length, i, temp;

    for (list_length = n; list_length >= 2; list_length--)
        for (i = 0; i < list_length - 1; i++)
            if (a[i] > a[i+1]) {
                temp = a[i];
                a[i] = a[i+1];
                a[i+1] = temp;
            }

} /* Bubble_sort */
```
Odd-even sort

- Expose more parallelism with pairwise swaps
  - Also called Odd-even transposition sort or brick sort.
- Algorithm: Repeat at most n phases
  - Even phases, compare swaps:
    \[(a[0], a[1]), (a[2], a[3]), (a[4], a[5]), \ldots\]
  - Odd phases, compare swaps:
    \[(a[1], a[2]), (a[3], a[4]), (a[5], a[6]), \ldots\]
  - Complexity: best case $O(n)$. Worst case $O(n^2)$
Example of odd-even sort

Start: 3, 6, 2, 1, 4, 7, 5, 0

Even phase: compare-swap (3,6), (2,1), (4,7), (5,0)
getting the list 3, 6, 1, 2, 4, 7, 0, 5

Odd phase: compare-swap (6,1), (2,4), (7,0)
void Odd_even_sort(
    int a[] /* in/out */,
    int n    /* in */)
{
    int phase, i, temp;

    for (phase = 0; phase < n; phase++)
        if (phase % 2 == 0) /* Even phase */
            for (i = 1; i < n; i += 2)
                if (a[i-1] > a[i]) {
                    temp = a[i];
                    a[i] = a[i-1];
                    a[i-1] = temp;
                }
        else /* Odd phase */
            for (i = 1; i < n-1; i += 2)
                if (a[i] > a[i+1]) {
                    temp = a[i];
                    a[i] = a[i+1];
                    a[i+1] = temp;
                }
} /* Odd_even_sort */
Communications among tasks in odd-even sort

Computation contains a set of tasks
Each task handling a[i] is labeled with “a[i]”.

![Diagram showing communications among tasks]

- Phase j
- Phase j+1
Parallel odd-even sort for n keys and p processes (n=p)

- Weakness
  - When n>>p, modify to let each process handle n/p keys
  - Too much communication overhead with key-level fine-grain data exchange/swap
Parallel odd-even sort for n keys and p processes \((n \gg p)\)

\[
\begin{align*}
&\text{Local sort} \\
&P_0: \begin{bmatrix} 13 & 7 & 12 \end{bmatrix} \\
&P_1: \begin{bmatrix} 8 & 5 & 4 \end{bmatrix} \\
&P_2: \begin{bmatrix} 6 & 1 & 3 \end{bmatrix} \\
&P_3: \begin{bmatrix} 9 & 2 & 10 \end{bmatrix} \\
\end{align*}
\]

\[
\begin{align*}
&\text{Process-level exchange/swap} \\
&P_0: \begin{bmatrix} 13 & 7 & 12 \end{bmatrix} \\
&P_1: \begin{bmatrix} 8 & 5 & 4 \end{bmatrix} \\
&P_2: \begin{bmatrix} 6 & 1 & 3 \end{bmatrix} \\
&P_3: \begin{bmatrix} 9 & 2 & 10 \end{bmatrix} \\
\end{align*}
\]

SORTED: \(1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ 12 \ 13\)
Parallel odd-even sort of n keys with p processes

• Each process owns \( \frac{n}{p} \) keys.
• First each process sorts its keys locally in parallel.
  ▪ E.g. call C library qsort for quick sorting
• Repeat at most p phases
  ▪ Even phases, process with even ID exchanges data with odd ID and swaps keys
    – (P0, P1), (P2, P3), (P4, P5) …
  ▪ Odd phases, compare swaps:
    – (P1, P2), (P3, P4), (P5, P6) …
### Textbook example of parallel odd-even sort

<table>
<thead>
<tr>
<th>Time</th>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
<th>Process 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>15, 11, 9, 16</td>
<td>3, 14, 8, 7</td>
<td>4, 6, 12, 10</td>
<td>5, 2, 13, 1</td>
</tr>
<tr>
<td>After Local Sort</td>
<td>9, 11, 15, 16</td>
<td>3, 7, 8, 14</td>
<td>4, 6, 10, 12</td>
<td>1, 2, 5, 13</td>
</tr>
<tr>
<td>After Phase 0</td>
<td>3, 7, 8, 9</td>
<td>11, 14, 15, 16</td>
<td>1, 2, 4, 5</td>
<td>6, 10, 12, 13</td>
</tr>
<tr>
<td>After Phase 1</td>
<td>3, 7, 8, 9</td>
<td>1, 2, 4, 5</td>
<td>11, 14, 15, 16</td>
<td>6, 10, 12, 13</td>
</tr>
<tr>
<td>After Phase 2</td>
<td>1, 2, 3, 4</td>
<td>5, 7, 8, 9</td>
<td>6, 10, 11, 12</td>
<td>13, 14, 15, 16</td>
</tr>
<tr>
<td>After Phase 3</td>
<td>1, 2, 3, 4</td>
<td>5, 6, 7, 8</td>
<td>9, 10, 11, 12</td>
<td>13, 14, 15, 16</td>
</tr>
</tbody>
</table>
Parallel time of odd-even sort

- **Total cost**
  - Local sorting using the best algorithm.
  - At most $p$ phases
    - Neighbor process data exchanges of $\frac{n}{p}$ keys
    - Merge and split two $\frac{n}{p}$ key lists
- $T_{\text{par}} = (\text{local sort}) + (p \text{ data exchanges}) + (p \text{ merges/splits})$
  - $= O\left(\frac{n}{p}\log(n/p)\right) + p*O(n/p) + p*O(n/p)$
  - $= O\left(\frac{n}{p}\log(n/p)\right) + O(2n)$
Comm\_sz= \# of processes

Sort local keys;

\textbf{for} (phase = 0; phase < comm\_sz; phase++) {
    partner = Compute\_partner(phase, my\_rank);
    \textbf{if} (I'm not idle) {
        Send my keys to partner;
        Receive keys from partner;
        \textbf{if} (my\_rank < partner)
            Keep smaller keys;
        else
            Keep larger keys;
    }
}
if (phase % 2 == 0) // Even phase */
if (my_rank % 2 != 0) // Odd rank */
    partner = my_rank - 1;
else // Even rank */
    partner = my_rank + 1;
else // Odd phase */
    if (my_rank % 2 != 0) // Odd rank */
        partner = my_rank + 1;
    else // Even rank */
        partner = my_rank - 1;
if (partner == -1 || partner == comm_sz)
    partner = MPI_PROC_NULL;
void Merge_low(
  int  my_keys[],          /* in/out */
  int  recv_keys[],         /* in */
  int  temp_keys[],         /* scratch */
  int  local_n              /* = n/p, in */)
{

  int  m_i, r_i, t_i;

  m_i = r_i = t_i = 0;
  while (t_i < local_n) {
    if (my_keys[m_i] <= recv_keys[r_i]) {
      temp_keys[t_i] = my_keys[m_i];
      t_i++;  m_i++;
    } else {
      temp_keys[t_i] = recv_keys[r_i];
      t_i++;  r_i++;
    }
  }

  for (m_i = 0; m_i < local_n; m_i++)
    my_keys[m_i] = temp_keys[m_i];
  /* Merge_low */
Safety in MPI programs

• The 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.
Buffers

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

Slide source: Bill Gropp, ANL
Avoiding Buffering

- Avoiding copies uses less memory
- May use more or less time

This requires that `MPI_Send` wait on delivery, or that `MPI_Send` return before transfer is complete, and we wait later.

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.
Will there be a deadlock?

- Assume tag/process ID is assigne 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>

Slide source: Bill Gropp, ANL
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)
- What happens with this code?

<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>

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

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

```c
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);
}
```
Uncertainty with five processes

Time 0

0 → 1

1 – 2

2 → 3

3 → 4

Time 1

0 → 1

1 → 2

2 – 3

3 – 4

Time 2

0 → 1

1 → 2

2 → 3

3 → 4

Copyright © 2010, Elsevier Inc. All rights Reserved
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 */ );
```
Use MPI_Sendrecv() in odd-even sort

- An alternative to scheduling deterministic communications
  - The dest and the source can be the same or different.
  - Send and receive datatypes may be different.
  - Can use Sendrecv with plain Send or Recv (or Irecv or Ssend_init, …)
- Ensure safer communication behavior so that the program won’t hang or crash.

MPI_Sendrecv(
    mykeys, n/comm_sz, MPI_INT, partner, 0,
    recvkeys,n/comm_sz, MPI_INT, partner, 0,
    comm, MPI_Status_ignore)
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>
Run-times of parallel odd-even sort

<table>
<thead>
<tr>
<th>Processes</th>
<th>Number of Keys (in thousands)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>200</td>
</tr>
<tr>
<td>1</td>
<td>88</td>
</tr>
<tr>
<td>2</td>
<td>43</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
</tr>
<tr>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>16</td>
<td>7.5</td>
</tr>
</tbody>
</table>

(times are in milliseconds)
Concluding Remarks (1)

• 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.
Backup Slides

• CS267 from J. Demmel/K. Yelick at Berkeley
MPI’s Non-blocking Operations

• Non-blocking operations return (immediately) “request handles” that can be tested and waited on:

  MPI_Request request;
  MPI_Status status;
  MPI_Isend(start, count, datatype, 
           dest, tag, comm, &request);
  MPI_Irecv(start, count, datatype, 
           dest, tag, comm, &request);
  MPI_Wait(&request, &status);
  (each request must be Waited on)

• One can also test without waiting:

  MPI_Test(&request, &flag, &status);
Communication Modes

MPI provides multiple modes for sending messages:

- Synchronous mode (**MPI_Ssend**): the send does not complete until a matching receive has begun. (Unsafe programs deadlock.)
- Buffered mode (**MPI_Bsend**): the user supplies a buffer to the system for its use. (User allocates enough memory to make an unsafe program safe.
- Ready mode (**MPI_Rsend**): user guarantees that a matching receive has been posted.
  - Allows access to fast protocols
  - undefined behavior if matching receive not posted

- **Non-blocking versions** (**MPI_Issend**, etc.)
- **MPI_Recv** receives messages sent in any mode.
- See [www.mpi-forum.org](http://www.mpi-forum.org) for summary of all flavors
Not Covered

- **Topologies**: map a communicator onto, say, a 3D Cartesian processor grid
  - Implementation can provide ideal logical to physical mapping
- **Rich set of I/O functions**: individual, collective, blocking and non-blocking
  - Collective I/O can lead to many small requests being merged for more efficient I/O
- **One-sided communication**: puts and gets with various synchronization schemes
  - Implementations not well-optimized and rarely used
  - Redesign of interface is underway
- **Task creation and destruction**: change number of tasks during a run
  - Few implementations available
Implementing Synchronous Message Passing

- Send operations complete after matching receive and source data has been sent.
- Receive operations complete after data transfer is complete from matching send.

1) Initiate send
2) Address translation on \( P_{\text{dest}} \)
3) Send-Ready Request
4) Remote check for posted receive
5) Reply transaction
6) Bulk data transfer
Implementing Asynchronous Message Passing

- Optimistic single-phase protocol assumes the destination can buffer data on demand.

1) Initiate send
2) Address translation on P_{dest}
3) Send Data Request

4) Remote check for posted receive
5) Allocate buffer (if check failed)
6) Bulk data transfer
Safe Asynchronous Message Passing

- Use 3-phase protocol
- Buffer on sending side
- Variations on send completion
  - wait until data copied from user to system buffer
  - don’t wait -- let the user beware of modifying data

1) Initiate send
2) Address translation on $P_{dest}$
3) Send-Ready Request
4) Remote check for posted receive
   - record send-rdy
5) Reply transaction
6) Bulk data transfer
Books on MPI

- *Designing and Building Parallel Programs*, by Ian Foster, Addison-Wesley, 1995.