COMS4040A & COMS7045A: High Performance Computing & Scientific Data Management
Introduction to MPI II

Hairong Wang

School of Computer Science & Applied Mathematics
University of the Witwatersrand, Johannesburg

Semester one 2025

Contents

WITS UNIVERSITY

Outline

Outline

Blocking vs. Non-blocking

- Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode.
- Blocking:
  - A blocking send routine will only “return” after it is safe to modify the application buffer (your send data) for reuse.
  - A blocking send can be synchronous which means there is handshaking occurring with the receive task to confirm a safe send.
  - A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.
  - A blocking receive only “returns” after the data has arrived and is ready for use by the program.

WITS UNIVERSITY

Blocking vs. Non-blocking cont.

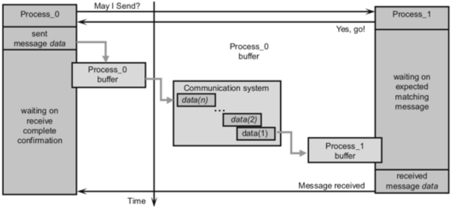


Figure: Communication between two processes awakes both of them while transferring data from sender to receiver, possibly with a set of shorter sub-messages.

Blocking vs. Non-blocking cont.

- Non-blocking:
  - Non-blocking send and receive routines behave similarly - they will return almost immediately. They do not wait for any communication events to complete.
  - Non-blocking operations simply “request” the MPI library to perform the operation when it is able.
  - It is unsafe to modify the application buffer (your variable space) until you know for a fact the requested non-blocking operation was actually performed by the library.
  - Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.

Outline

MPI Message Passing Routine Arguments

MPI point-to-point communication routines generally have an argument list that takes one of the following formats:

Blocking sends	`MPI_Send(buffer,count,type,dest,tag,comm)`
Non-blocking sends	`MPI_Isend(buffer,count,type,dest,tag,comm,request)`
Blocking receive	`MPI_Recv(buffer,count,type,source,tag,comm,status)`
Non-blocking receive	`MPI_Irecv(buffer,count,type,source,tag,comm,request)`

Outline

1. Point to Point Communication
   - Blocking vs. Non-blocking
   - MPI Message Passing Routine Arguments
   - Avoiding Deadlocks
   - Sending and Receiving Messages Simultaneously
   - Overlapping Communication with Computation
2. Collective Communications
   - MPI Collective Communication Illustrations
3. Examples
4. Summary

WITS UNIVERSITY

Avoiding Deadlocks

- The semantics of `MPI_Send` and `MPI_Recv` place some restrictions on how we can mix and match send and receive operations.
- Sources of 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)
  - Mismatched send and receive – unsafe.

- What happens with

Process 0	Process 1
Send(1)	Send(0)
Recv(0)	Recv(1)

- Order the operations.

Process 0	Process 1
Send(1)	Recv(1)
Recv(0)	Send(0)

WITS UNIVERSITY

Avoiding Deadlocks

Example 1

Process 0 sends two messages with different tags to process 1, and process 1 receives them in reverse order.

```c
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
  MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
  MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
} else if (myrank == 1) {
  MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
...
```

Avoiding Deadlocks

Example 2

Consider the following piece of code, in which process i sends a message to process $i+1$ (modulo the number of processes) and receives a message from process $i-1$ (modulo the number of processes).

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
                  MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
                  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
...
```

WITS UNIVERSITY

Avoiding Deadlocks

Example 2 cont.

We can break the circular wait to avoid deadlocks as follows:

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 1) {
  MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
     MPI_COMM_WORLD);
  MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
} else {
  MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
     MPI_COMM_WORLD);
}
...
```

Outline

Sending and Receiving Messages Simultaneously

To exchange messages, MPI provides the following function that both sends and receives a message:

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, int dest, int sendtag,
        void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
        int source, int recvtag, MPI_Comm comm,
        MPI_Status *status)
```

The arguments include arguments to the send and receive functions.

WITS UNIVERSITY

Using MPI_Sendrecv in Example 2

Example 2 can be made “safe” by using `MPI_Sendrecv`:

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Sendrecv(a, 10, MPI_INT, (myrank+1)%npes, 1, b, 10,
     MPI_INT, (myrank-1+npes)%npes, 1,
     MPI_COMM_WORLD, &status);
...
```

Outline

Overlapping Communication with Computation

- In order to overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations.

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
        int dest, int tag, MPI_Comm comm,
        MPI_Request *request)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
        int source, int tag, MPI_Comm comm,
        MPI_Request *request)
```

- These operations return before the operations have been completed. Function `MPI_Test` tests whether or not the non-blocking send or receive operation identified by its `request` has finished.
-
```
int MPI_Test(MPI_Request *request, int *flag,
        MPI_Status *status)
```

- `MPI_Wait` blocks until a specified non-blocking send or receive operation has completed. For multiple non-blocking operations, the programmer can specify `any`, `all` or `some` completions.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

- `MPI_Request` handle is used to determine whether an operations has completed.
    - Non-blocking wait: `MPI_Test`
    - Blocking wait: `MPI_Wait`
- Anywhere you use `MPI_Send` or `MPI_Recv`, you can use the pair of `MPI_Isend/MPI_Wait` or `MPI_Irecv/MPI_Wait`.
- It is sometimes desirable to wait on multiple requests:
    - ```
      MPI_Waitall(int count,
      MPI_Request array_of_requests[],
              MPI_Status array_of_statuses[])
      ```
    - The corresponding version of `MPI_Test`

      ```
      int MPI_Testall(int count,
      MPI_Request array_of_requests[], int *flag,
      MPI_Status array_of_statuses[])
      ```

      `flag`: true if all the requests are completed, otherwise false.
      ```

Example 3

```
int main(int argc, char *argv[]){
    int myid, numprocs, left, right, flag=0;
    int buffer1[10], buffer2[10];
    MPI_Request request; MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    /* initialize buffer2 */
    ......
    right = (myid + 1) % numprocs;
    left = myid - 1;
    if (left < 0)
        left = numprocs - 1;
    MPI_Irecv(buffer1, 10, MPI_INT, left, 123, MPI_COMM_WORLD,
        &request);
    MPI_Send(buffer2, 10, MPI_INT, right, 123, MPI_COMM_WORLD);
    MPI_Test(&request, &flag, &status);
    while (!flag){
        /* Do some work ... */
        MPI_Test(&request, &flag, &status);
    }
    MPI_Finalize();
}
```

Example 4

```
int main(int argc, char *argv[]){
  int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
    MPI_Request reqs[4]; MPI_Status stats[4];
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    prev = rank-1; next = rank+1;
    if (rank == 0)  prev = numtasks - 1;
    if (rank == (numtasks - 1))  next = 0;
    MPI_Irecv(&buf[0],1,MPI_INT,prev,tag1,MPI_COMM_WORLD,
      &reqs[0]);
    MPI_Irecv(&buf[1],1,MPI_INT,next,tag2,MPI_COMM_WORLD,
      &reqs[1]);

    MPI_Isend(&rank,1,MPI_INT,prev,tag2,MPI_COMM_WORLD,
      &reqs[2]);
    MPI_Isend(&rank,1,MPI_INT,next,tag1,MPI_COMM_WORLD,
      &reqs[3]);
    MPI_Waitall(4, reqs, stats);
    MPI_Finalize();
}
```

Example 5 (The Trapezoidal Rule)

- We can use **the trapezoidal rule** to approximate the area between the graph of a function, $y = f(x)$, two vertical lines, and the x-axis.

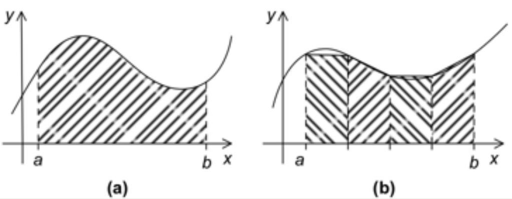


Figure: The trapezoidal rule: (a) area to be estimated, (b) estimate area using trapezoids

WITS UNIVERSITY

Example 5 cont.

- If the endpoints of the subinterval are x_i and x_{i+1}, then the length of the subinterval is $h = x_{i+1} - x_i$. Also, if the lengths of the two vertical segments are $f(x_i)$ and $f(x_{i+1})$, then the area of the trapezoid is

$$\text{Area of one trapezoid} = \frac{h}{2}(f(x_i) + f(x_{i+1})).$$

- Since we chose the $\mathtt{N}$ subintervals, we also know that the bounds of the region are $x = a$ and $x = b$ then

$$h = \frac{b - a}{N}$$

WITS UNIVERSITY

Example 5 cont.

- The pseudo code for a serial program:

```
h = (b-a)/N;
approx = (f(a) + f(b))/2.0;
for(i=1; i<=n-1; i++){
    x_i = a + i * h;
    approx += f(x_i);
}
approx = h * approx;
```

Recall we can design a parallel program using four basic steps:

1. Partition the problem solution into tasks.
2. Identify the communication between the tasks.
3. Aggregate the tasks into composite tasks.
4. Map the composite tasks to cores.

Example 5: Parallel Algorithm for the Trapezoidal Rule

Assuming `comm_sz` evenly divides *n*, the pseudo-code for the parallel program looks like the following:

```
Get a, b, n;
h = (b - a)/n;
local_n = n/comm_sz;
local_a = a + my_rank * local_n * h;
local_b = local_a + local_n * h;
local_integral = Trap(local_a, local_b, local_n, h);
if (my_rank != 0)
  Send local integral to process 0;
else {/* my_rank == 0 */
  total_integral = local_integral;
  for (proc = 1; proc < comm_sz; proc++) {
    Receive local_integral from proc;
    total_integral += local_integral;
  }
}
if (my_rank == 0)
  print result;
```

Dealing with I/O

- In most cases, all the processes in `MPI_COMM_WORLD` have access to `stdout` and `stderr`.
- The order in which the processes' output appears is indeterministic.
- For the input, i.e., `stdin`, usually, only process 0 has access to.
- If an MPI program uses `scanf` function, then process 0 reads in the data, and sends it to the other processes.

Outline

WITS UNIVERSITY

Collective Communications

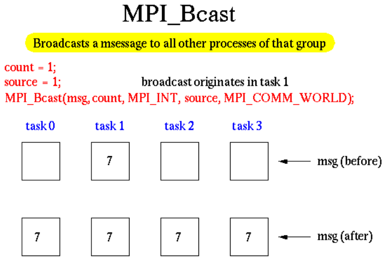
- Communication is coordinated among a group of processes, as specified by a communicator.
- All collective operations are blocking and no message tags are used.
- All processes in the communicator must call the collective operation.
- Three classes of collective operations
  - Data movement
  - Collective computation
  - Synchronization

Outline

WITS UNIVERSITY

MPI_Bcast

- A collective communication in which data belonging to a single process is sent to all of the processes in the communicator is called a broadcast — `MPI Bcast`.



- The process with rank `source` sends the contents of the memory referenced by `msg` to all the processes in the communicator `MPI_COMM_WORLD`.

Example 5 cont.

1. In the example `mpi_trapezoid_1.c`, we are using

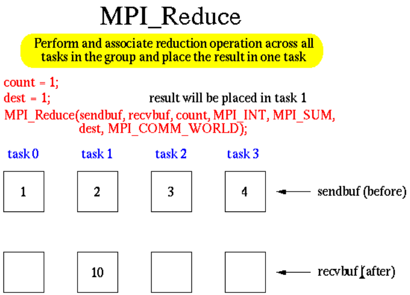
```
if (my_rank == 0) {
  for (dest = 1; dest < comm_sz; dest++){
    MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
    MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
    MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
  } else {/* my rank != 0 */
    MPI_Recv(a_p, 1, MPI_DOUBLE, 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);
  }
}
```

2. Instead of using point-to-point communications, you can use collective communications here. Write another function to implement this part using — `MPI_Bcast()`.

WITS UNIVERSITY

MPI_Reduce

- `MPI_Reduce` combines data from all processes in the communicator and returns it to one process.
- In many numerical algorithms, `Send/Receive` can be replaced by `Bcast/Reduce`, improving both simplicity and efficiency.



MPI_Reduce cont.

- When the `count` is greater 1, `MPI_Reduce` operate on arrays instead of scalars.

```
double local_x[N], sum[N];
...
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0, \
    MPI_COMM_WORLD);
```

Example 5 cont.

1. In the example `mpi_trapezoid_1.c`, we are using

```
/* Add up the integrals calculated by each process */
  if (my_rank != 0) {
    MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
  } else {
    total_int = local_int;
    for (source = 1; source < comm_sz; source++) {
      MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
           MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      total_int += local_int;
    }
```

2. Instead of using point-to-point communications, you can also use collective communications here. Rewrite this part using appropriate collective communication.

MPI_Reduce

- Suppose that each process calls `MPI_Reduce` with operator `MPI_SUM`, and destination process 0. What happens with the following multiple calls of `MPI_Reduce`? What are the values for `b` and `d`?

Time	Process 0	Process 1	Process 2
0	`a=1; c = 2;`	`a=1; c = 2;`	`a=1; c = 2;`
1	`MPI_Reduce(&a,&b,...)`	`MPI_Reduce(&c,&d,...)`	`MPI_Reduce(&a,&b,...)`
2	`MPI_Reduce(&c,&d,...)`	`MPI_Reduce(&a,&b,...)`	`MPI_Reduce(&c,&d,...)`

- The order of the calls will determine the matching.
- What will happen with the following code?

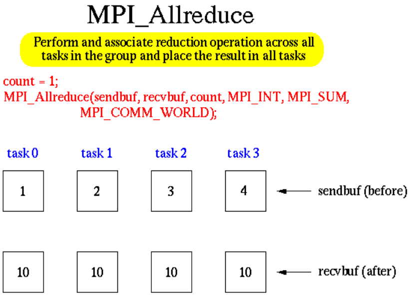
```
MPI_Reduce(&x, &x, 1, MPI_DOUBLE, MPI_SUM, 0, comm);
```

MPI_Allreduce

- If the result of the reduction operation is needed by all processes, MPI provides:

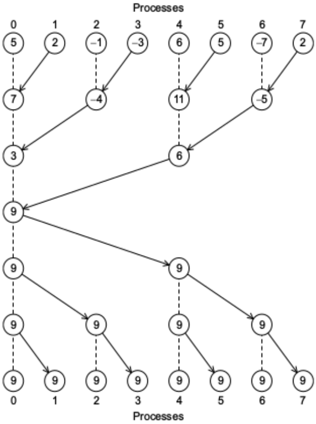
```
int MPI_Allreduce(void *sendbuf,void *recvbuf,
    int count,MPI_Datatype datatype,MPI_Op op,
    MPI_Comm comm)
```

- This is equivalent to an `MPI_Reduce` followed by an `MPI_Bcast`.



WITS UNIVERSITY

MPI_Allreduce cont.



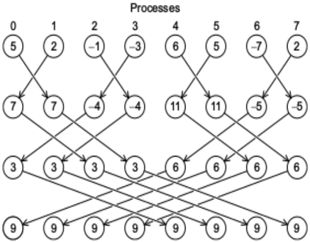


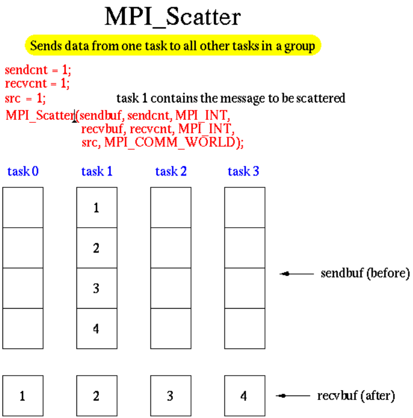
Figure: (a) A global sum followed by a broadcasting; (2) A butterfly structured global sum.

MPI_Scatter

- The scatter operation is to distribute distinct messages from a single source task to each task in the group.

```
int MPI_Scatter(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf,
        int recvcount, MPI_Datatype recvdatatype,
        int source, MPI_Comm comm)
```

MPI_Scatter cont.



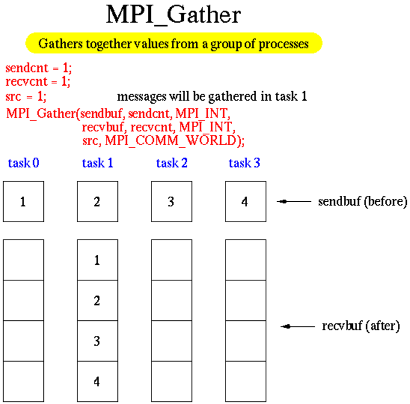
MPI_Gather

- The gather operation is performed in MPI using `MPI_Gather`.
  - Gathers distinct messages from each task in the group to a single destination task.
  - Reverse operation of `MPI_Scatter`.

```
int MPI_Gather(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf,
        int recvcount, MPI_Datatype recvdatatype,
        int target, MPI_Comm comm)
```

WITS UNIVERSITY

MPI_Gather cont.



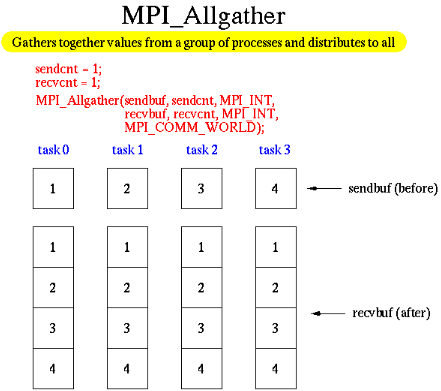
MPI_Allgather

- MPI also provides the `MPI_Allgather` function in which the data are gathered at all the processes.

```
int MPI_Allgather(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf,
        int recvcount, MPI_Datatype recvdatatype,
        MPI_Comm comm)
```

WITS UNIVERSITY

MPI_Allgather cont.



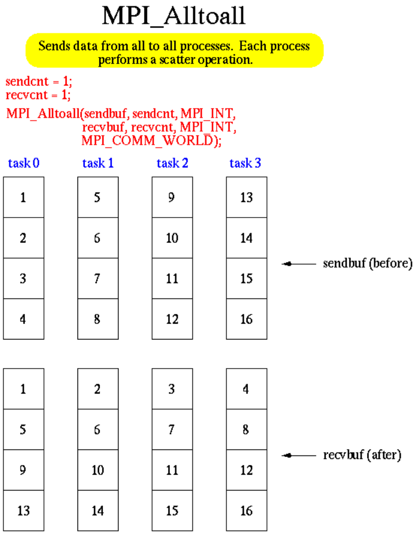
MPI_Alltoall

- The all-to-all communication operation is performed by:

```
int MPI_Alltoall(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf,
        int recvcount, MPI_Datatype recvdatatype,
        MPI_Comm comm)
```

- Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order by index.

MPI_Alltoall cont.



Example 6 (Matrix vector multiplication)

If $A = (a_{ij})$ is an $m \times n$ matrix and $\mathbf{x}$ is a vector with n components, then $\mathbf{y} = A\mathbf{x}$ is a vector with m components. Furthermore,

$$y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{i,n-1}x_{n-1}.$$

A serial code can be as simple as

```
    for (i = 0; i < m; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
        y[i] += A[i*n+j]*x[j];
    }
```

WITS UNIVERSITY

Example 6 cont.

Process 0 reads in the matrix and distributes row blocks to all the processes in communicator `comm`.

```
if (my_rank == 0) {
  A = malloc(m*n*sizeof(double));
  if (A == NULL) local_ok = 0;
  Check_for_error(local_ok, "Random_matrix",
    "Can't allocate temporary matrix", comm);
  srand(2018);
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      A[i*n+j] = (double)rand( ) / RAND_MAX;
    MPI_Scatter(A, local_m*n, MPI_DOUBLE,
      local_A, local_m*n, MPI_DOUBLE, 0, comm);
    free(A);
} else {
  Check_for_error(local_ok, "Random_matrix",
    "Can't allocate temporary matrix", comm);
  MPI_Scatter(A, local_m*n, MPI_DOUBLE,
    local_A, local_m*n, MPI_DOUBLE, 0, comm);
}
```

Example 6 cont.

Each process gathers the entire vector, then proceeds to compute its share of sub-matrix and vector multiplication.

```
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];
}
```

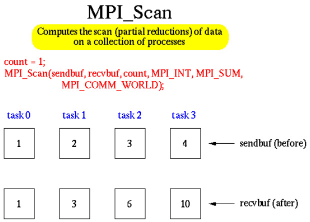
Exercise: Matrix Transpose

Implement matrix transpose using MPI scatter and gather operations.

MPI_Scan

- To compute prefix-sums, MPI provides:

```
int MPI_Scan(void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```



- Using this core set of collective operations, a number of programs can be greatly simplified.

MPI_Scatterv

Scatters a buffer in parts to all processes in a communicator, which allows different amounts of data to be sent to different processes.

```
int MPI_Scatterv(void *sendbuf, int *sendcounts, int *displs,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int source, MPI_Comm comm)
```

- `sendbuf`: address of send buffer (significant only at root)
- `sendcounts`: integer array (of length group size) specifying the number of elements to send to each processor
- `displs`: integer array (of length group size). Entry i specifies the displacement (relative to `sendbuf` from which to take the outgoing data to process i
- `sendtype`: data type of send buffer elements
- `recvcount`: number of elements in receive buffer (integer)
- `recvtype`: data type of receive buffer elements
- `root`: rank of sending process (integer)

MPI_Scatterv cont.

Example 7

Given an $N \times N$ matrix, A, of integers, write an MPI program that distributes the first M rows of the upper triangle of A to M processes by rows, where each process gets one row of the upper triangle of A (when $M = N$, it means each process gets one row of the upper triangle of A).

For this example, we can use `MPI_Scatterv`.

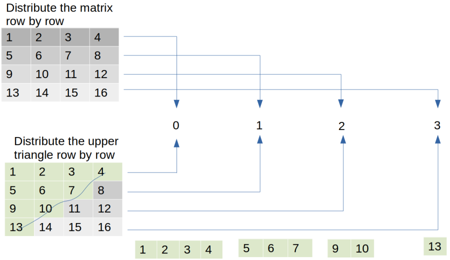


Figure: `MPI_Scatter` (top) and `MPI_Scatterv` (bottom) example

MPI_Scatterv cont.

For Example 7, assuming the matrix is only 4×4, and we are running the MPI code using 4 processes, then some of the arguments of calling `MPI_Scatterv`:

- `sendcounts[4] = {4, 3, 2, 1}`;
- `displs[4] = {0, 4, 8, 12}` which is with reference to `sendbuf`; these values can be expressed as `N * rank`, where `rank` is the rank of a process.
- note also that `recvcount` in `MPI_Scatterv` is a scalar; for process 0, `recvcount = 4 (=4-0)`; for process 1, `recvcount = 3 (=4-1)`; for process 2, `recvcount = 2 (=4-2)`; and for process 3, `recvcount = 1 (=4-3)`; so this value can be obtained as `N - rank` where `N` is the number of rows in the matrix, and `rank` is the rank of a process.

`scatterv_1.c` gives an example code for Example 7.

WITS UNIVERSITY

MPI_Alltoallv

Sends data from all to all processes; each process may send a different amount of data and provide displacements for the input and output data.

```
MPI_Alltoallv(void *sendbuf, int *sendcounts, int *sdispls,
MPI_Datatype sendtype, void *recvbuf, int *recvcounts,
int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)
```

- `sendbuf`: starting address of send buffer
- `sendcounts`: integer array equal to the group size specifying the number of elements to send to each processor
- `sdispls`: integer array (of length group size). Entry j specifies the displacement (relative to `sendbuf` from which to take the outgoing data destined for process j
- `sendtype`: data type of send buffer elements
- `recvcounts`: integer array equal to the group size specifying the maximum number of elements that can be received from each processor
- `rdispls`: integer array (of length group size). Entry i specifies the displacement (relative to `recvbuf` at which to place the incoming data from process i
- `recvtype`: data type of receive buffer elements

WITS UNIVERSITY

MPI_Alltoallv cont.

Example 8

Given the `MPI_Alltoallv` argument settings shown in the figure (the number of processes is 3), what is the content of `recvbuf` for each process?

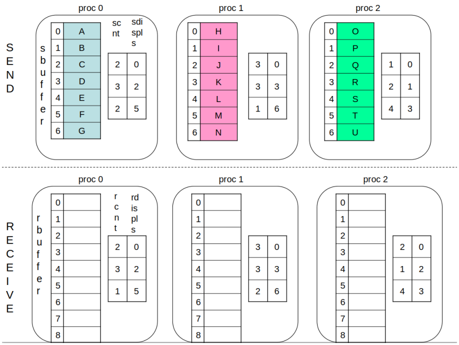


Figure: `MPI_Alltoallv` example

WITS UNIVERSITY

MPI_Alltoallv cont.

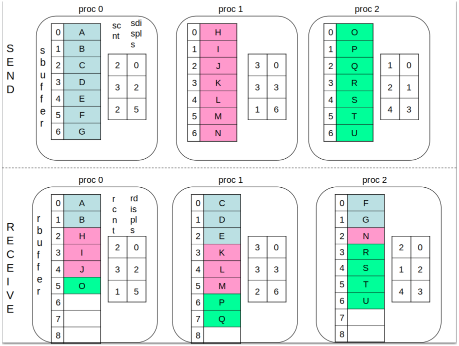


Figure: `MPI_Alltoallv` example

MPI_Gatherv

The following function allows a different number of data elements to be sent by each process by replacing `recvcount` in `MPI_Gather` with an array `recvcounts`

```
int MPI_Gatherv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int *recvcounts, int *displs,
MPI_Datatype recvtype, int target, MPI_Comm comm)
```

- `sendbuf`: pointer, starting address of send buffer (or the data to be sent)
- `sendcount`: the number of elements in the send buffer
- `sendtype`: datatype of send buffer elements
- `recvbuf`: pointer, starting address of receive buffer (significant only at root)
- `recvcounts`: integer array (of length group size) containing the number of elements to be received from each process (significant only at root)
- `displs`: integer array (of length group size). Entry i specifies the displacement relative to `recvbuf` at which to place the incoming data from process i (significant only at root)
- `recvtype`: the datatype of data to be received (significant only at root)
- `target`: rank of receiving process (integer)

MPI_Allgatherv

Gather data from all processes and deliver the combined data to all processes

```
int MPI_Allgatherv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int *recvcounts, int *displs,
MPI_Datatype recvtype, MPI_Comm comm)
```

- `sendbuf`: pointer, starting address of send buffer (or the data to be sent)
- `sendcount`: the number of elements in the send buffer
- `sendtype`: datatype of send buffer elements
- `recvbuf`: pointer, starting address of receive buffer (significant only at root)
- `recvcounts`: integer array (of length group size) containing the number of elements to be received from each process (significant only at root)
- `displs`: integer array (of length group size). Entry i specifies the displacement relative to `recvbuf` at which to place the incoming data from process i (significant only at root)
- `recvtype`: the datatype of data to be received (significant only at root)

Outline

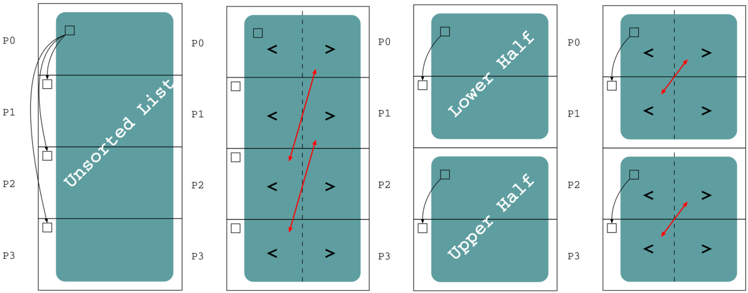
1. Point to Point Communication
   - Blocking vs. Non-blocking
   - MPI Message Passing Routine Arguments
   - Avoiding Deadlocks
   - Sending and Receiving Messages Simultaneously
   - Overlapping Communication with Computation
2. Collective Communications
   - MPI Collective Communication Illustrations
3. Examples
4. Summary

WITS UNIVERSITY

Parallel quicksort

- one process broadcast initial pivot to all processes;
- each process in the upper half swaps with a partner in the lower half
- recurse on each half
- swap among partners in each half
- each process uses quicksort on local elements

Parallel quicksort cont.



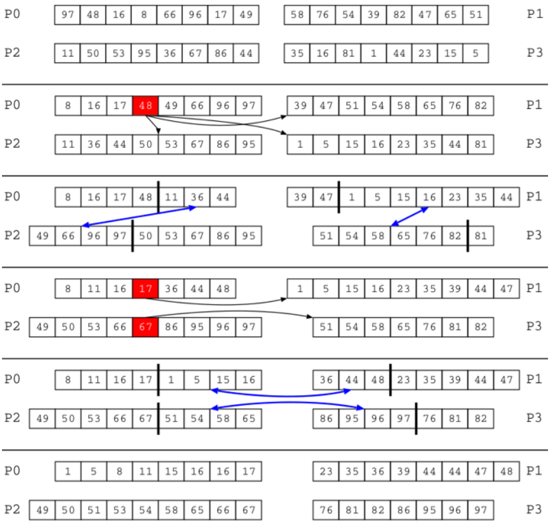
WITS UNIVERSITY

Hyperquicksort

Limitation of parallel quicksort: poor balancing of list sizes.
Hyperquicksort: sort elements before broadcasting pivot.

- sort elements in each process
- select median as pivot element and broadcast it
- each process in the upper half swaps with a partner in the lower half
- recurse on each half

Hyperquicksort cont.



Example 9 (Task 0 pings task 1 and awaits return ping)

```
#include "mpi.h"
#include <stdio.h>
int  main(int argc, char *argv[]) {
  int numtasks, rank, dest, source, rc, count, tag=1;
  char inmsg, outmsg='x';
  MPI_Status Stat;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0){
    dest = 1;
    source = 1;
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &
        Stat);
  }
  else if (rank == 1) {
    dest = 0;
    source = 0;
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &
        Stat);
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }
  rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
  printf("Task %d: Received %d char(s) from task %d with tag %d \n",
    rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);
  MPI_Finalize();
}
```

Example 10 (Nearest neighbor exchange in a ring topology)

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]){
  int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
  MPI_Request reqs[4];
  MPI_Status stats[4];

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  prev = rank-1;
  next = rank+1;
  if (rank == 0)  prev = numtasks - 1;
  if (rank == (numtasks - 1))  next = 0;
  MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
  MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

  MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
  MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

  MPI_Waitall(4, reqs, stats);

  MPI_Finalize();
}
```

WITS UNIVERSITY

Example 11 (Perform a scatter operation on the rows of an array)

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(int argc, char *argv[]) {
  int numtasks, rank, sendcount, recvcount, source;
  float sendbuf[SIZE][SIZE] = {
      {1.0, 2.0, 3.0, 4.0},
      {5.0, 6.0, 7.0, 8.0},
      {9.0, 10.0, 11.0, 12.0},
      {13.0, 14.0, 15.0, 16.0}  };
  float recvbuf[SIZE];
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  if (numtasks == SIZE) {
    source = 1;
    sendcount = SIZE;
    recvcount = SIZE;
    MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,
      MPI_FLOAT,source,MPI_COMM_WORLD);

    printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0],
      recvbuf[1],recvbuf[2],recvbuf[3]);
  }
  else
    printf("Must specify %d processors. Terminating.\n",SIZE);

  MPI_Finalize();
}
```

Example 12 (The Odd-Even Transposition Sort)

- Sorts n elements in n phases (n is even), each of which requires $n/2$ compare-exchange operations.
- The algorithm alternates between two phases — odd and even phases.
- Let $< a_0, a_1, ..., a_{n-1} >$ be the sequence to be sorted.
  - During the odd phase, elements with odd indices are compared with their right neighbours, and if they are out of sequence they are exchanged; thus, the pairs $(a_1, a_2), (a_3, a_4), \ldots, (a_{n-3}, a_{n-2})$ are compare exchanged.
  - During the even phase, elements with even indices are compared with their right neighbours, and if they are out of sequence they are exchanged; $(a_0, a_1), (a_2, a_3), \ldots, (a_{n-2}, a_{n-1})$.
- After n phases of odd-even exchanges, the sequence is sorted. Each phase requires $n/2$ compare-exchange operations (sequential complexity $O(n^2)$).

Example 12 cont. – The serial algorithm

```
for i = 0 to n-1 do
  if i is even then
    for j = 0 to n/2 - 1 do
      compare-exchange(a(2j), a(2j+1));
  if i is odd then
    for j = 0 to n/2 - 1 do
      compare-exchange(a(2j+1), a(2j+2));
```

WITS UNIVERSITY

Example 12 cont. – The parallel algorithm

```
void oddevensort(int n)
  id = process's label;
  for i =0 to n-1 do
    if i is odd then
      if id is odd then
        compare-exchange_min(id, id + 1);//increasing comparator
      else
        compare-exchange_max(id, id - 1);//decreasing comparator
    if i is even then
      if id is even then
        compare-exchange_min(id, id + 1);
      else
        compare-exchange_max(id, id - 1);
```

WITS UNIVERSITY

Outline

WITS UNIVERSITY

Summary

- Point-to-point communication
  - Blocking vs non-blocking
  - Safety in MPI programs
- Collective communication
  - Collective communications involve all the processes in a communicator.
  - All the processes in the communicator must call the same collective function.
  - Collective communications do not use tags, the message is matched on the order in which they are called within the communicator.
  - The meanings of *local variable* and *global variable* in MPI
  - Some important MPI collective communications we learned: `MPI_Reduce`, `MPI_Allreduce`, `MPI_Bcast`, `MPI_Gather`, `MPI_Scatter`, `MPI_Allgather`, `MPI_Alltoall`, `MPI_Scan` etc.

WITS UNIVERSITY

References

The resources used include

- *Introduction to Parallel Computing*
- *MPI Forum*
- *Using MPI: Portable Parallel Programming with the Message Passing Interface*
- *Parallel Programming in C with MPI and OpenMP*
- `https://hpc-tutorials.llnl.gov/mpi/`

WITS UNIVERSITY

References cont. I

- Grama, Ananth et al. *Introduction to Parallel Computing*. Addison Wesley, 2003.
- Gropp, William, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. Second. The MIT Press, Cambridge, Massachusetts, London, England., 1999.
- MPI. *MPI Forum*. https://www.mpi-forum.org/docs/. Accessed 2025-4-30. 2022.
- Quinn, Michael J. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.