# Table of Contents

1. **MPI Collective Communications**
   - Parallel Example: Trapezoid Rule
   - Tree Structured Communications
   - Tree-Structure Sums
   - Collective vs Point-to-Point Communications
     - Collective Communication: MPI\_Reduce
     - Collective Communication: MPI\_Allreduce
     - Collective Communication: MPI\_Broadcast
     - Data Distribution: Vector Add
     - Collective Communication: MPI\_Scatter and MPI\_Gather
   - MPI Derived Data Types
COLLECTIVE COMMUNICATION
Recall: The Trapezoid Rule for Numerical Integration

Solve the Integral: \( \int_{a}^{b} F(x) \, dx \)

The Trapezoidal Rule

Where \( F(x) \) can be any function of \( x \): \( f(x^2) \), \( f(x^3) \) See Pacheco (2011), Ch3.
Parallelizing the Trapezoidal Rule

1. Partition problem solution into tasks.
2. Identify communication channels between tasks.
3. Aggregate tasks into composite tasks.
4. Map composite tasks to cores.
MPI Collective Communications
Parallel Example: Trapezoid Rule

Master Node collects sums using MPI_SEND / MPI_RECV
\[ \vartheta(np - 1), \text{ where } np \text{ is the number of } PEs \]
Output shows 7 Send/Recv pairs: \( \vartheta (np - 1) = \vartheta (7) \), for \( np = 8 \)

[mthomas] % mpirun -np 8 ./mpi_trap2
Enter a, b, and n
1 50 100
PE [1] SEND: For 12 trapezoids, local estimate = 584.199266
PE [4] SEND: For 12 trapezoids, local estimate = 4451.000162
PE [5] SEND: For 12 trapezoids, local estimate = 6553.123682
PE [0] RECV from PE [1]: local estimate = 584.199266
PE [0] RECV from PE [2]: local estimate = 1466.537954
PE [2] SEND: For 12 trapezoids, local estimate = 1466.537954
PE [6] SEND: For 12 trapezoids, local estimate = 9061.842146
PE [7] SEND: For 12 trapezoids, local estimate = 11977.155554
PE [3] SEND: For 12 trapezoids, local estimate = 2755.471586
PE [0] RECV from PE [3]: local estimate = 2755.471586
PE [0] RECV from PE [4]: local estimate = 4451.000162
PE [0] RECV from PE [5]: local estimate = 6553.123682
PE [0] RECV from PE [6]: local estimate = 9061.842146
PE [0] RECV from PE [7]: local estimate = 11977.155554
With \( n = 100 \) trapezoids, our estimate of the integral from 1.000000 to 50.000000 = 3.695778587200000e+04
How to reduce number of communications?

1. In the first phase:
   (a) Process 1 sends to 0, 3 sends to 2, 5 sends to 4, and 7 sends to 6.

2. Next phase:
   (a) Processes 0, 2, 4, and 6 add in the received values.
   (b) Processes 2 and 6 send their new values to processes 0 and 4, respectively.
   (c) Processes 0 and 4 add the received values into their new values

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

Compare with communication pattern used by 1st version of trap.c algorithm: 7 messages, 7 adds by node 0. Here the master has 3 messages and 3 adds.
An alternative tree-structured global sum
Collective Communication: MPI_Reduce

MPI_Reduce

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

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

```c
double local_x[N], sum[N];

MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Operator passed as an argument. Count > 1 supports arrays
Predefined reduction operators in MPI

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

- All communicator processes must call the same collective function.
  - e.g.: the program attempts to match a call to MPI_Reduce on PE(i) with a call to MPI_Recv on PE(j) will cause the program to hang or crash.

- Arguments passed by each process to an MPI collective communication must be compatible.
  - e.g.: PE(i) passes in 0 as the dest_process and another passes in 1, then the outcome of a call to MPI_Reduce causes the code to hang or crash.

- The output_data_p argument is only used on dest_process.

- All processes need to pass an argument corresponding to output_data_p.

- Point-to-point communications are matched on the basis of tags and communicators.

- Collective communications matched by communicator and order called.
Example (1)

<table>
<thead>
<tr>
<th>Time</th>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>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
Example (2)

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

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

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

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

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

```c
int MPI_Allreduce(
    void* input_data_p /* in */,
    void* output_data_p /* out */,
    int count /* in */,
    MPI_Datatype datatype /* in */,
    MPI_Op operator /* in */,
    MPI_Comm comm /* in */);
```
MPI AllReduce: Butterfly communication pattern, $O(n)$
A butterfly-structured global sum.

\[ O \log_2 (n) \]
Broadcast

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

```c
int MPI_Bcast(
    void* data_p,  /* in/out */
    int count,    /* in */
    MPI_Datatype datatype, /* in */
    int source_proc, /* in */
    MPI_Comm comm); /* in */
```
A tree-structured broadcast.
A version of Get_input that uses MPI_Bcast

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

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
    }

    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);

    /* Get_input */
```
Data distributions

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

*Compute a vector sum.*
Serial implementation of vector addition

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

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

<table>
<thead>
<tr>
<th>Process</th>
<th>Block</th>
<th>Cyclic</th>
<th>Block-cyclic Blocksize = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1 2</td>
<td>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>
Partitioning options

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

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

- **Block-cyclic partitioning**
  - Use a cyclic distribution of blocks of components.
Block Partitioning of a 20 element vector across 4 processors.
Cyclic Partitioning of a 20 element vector across 4 processors.

1D Vector Data Distribution: Cyclic (Note: using Fortran indexing)
Parallel implementation of vector addition

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

    for (local_i = 0; local_i < local_n; local_i++)
        local_z[local_i] = local_x[local_i] + local_y[local_i];

    /* Parallel_vector_sum */
```
Scatter

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

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

Note: the calculation of buffers and counts must be done by programmer.
All nodes call `MPI_Scatter`; only the master node has a value for `a`; the destination processors store the value of `a` into local variable `local_n`
Gather

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

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

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

    double* b = NULL;
    int i;
```
Print a distributed vector (2)

```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 */
```
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 */);
```
Grouping Data for Communication: Derived DataTypes
Derived datatypes

- Used to represent any collection of data items in memory by storing both the types of the items and their relative locations in memory.
- The idea is that if a function that sends data knows this information about a collection of data items, it can collect the items from memory before they are sent.
- Similarly, a function that receives data can distribute the items into their correct destinations in memory when they’re received.
Derived datatypes

- Formally, consists of a sequence of basic MPI data types together with a displacement for each of the data types.
- Trapezoidal Rule example:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>24</td>
</tr>
<tr>
<td>b</td>
<td>40</td>
</tr>
<tr>
<td>n</td>
<td>48</td>
</tr>
</tbody>
</table>

\{ (MPI\_DOUBLE, 0), (MPI\_DOUBLE, 16), (MPI\_INT, 24) \}
MPI_Type create_struct

- Builds a derived datatype that consists of individual elements that have different basic types.

```c
int MPI_Type_create_struct(
    int count,
    int array_of_blocklengths[],
    int array_of_displacements[],
    MPI_Aint array_of_types[],
    MPI_Datatype* new_type_p
);
```
**MPI_Get_address**

- Returns the address of the memory location referenced by `location_p`.
- The special type `MPI_Aint` is an integer type that is big enough to store an address on the system.

```c
int MPI_Get_address(
    void* location_p /* in */,
    MPI_Aint* address_p /* out */);
```
MPI_Type_commit

- Allows the MPI implementation to optimize its internal representation of the datatype for use in communication functions.

```c
int MPI_Type_commit(MPI_Datatype* new_mpi_t_p /* in/out */);
```
MPI_Type_free

- When we’re finished with our new type, this frees any additional storage used.

```c
int MPI_Type_free(MPI_Datatype* old_mpi_t_p /* in/out */);
```
Example from mpi-trap4.c

Get input function with a derived datatype (1)

```c
void Build_mpi_type(
    double* a_p /* in */,
    double* b_p /* in */,
    int* n_p /* in */,
    MPI_Datatype* input_mpi_t_p /* out */) {

    int array_of_blocklengths[3] = {1, 1, 1};
    MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
    MPI_Aint a_addr, b_addr, n_addr;
    MPI_Aint array_of_displacements[3] = {0};
```
Example from mpi-trap4.c

Get input function with a derived datatype (2)

```c
MPI_Get_address(a_p, &a_addr);
MPI_Get_address(b_p, &b_addr);
MPI_Get_address(n_p, &n_addr);
array_of_displacements[1] = b_addr - a_addr;
MPI_Type_create_struct(3, array_of_blocklengths,
                          array_of_displacements, array_of_types,
                          input_mpi_t_p);
MPI_Type_commit(input_mpi_t_p);
} /* Build_mpi_type */
```
Example from mpi-trap4.c

Get input function with a derived datatype (3)

```c
void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p,
               int* n_p) {
    MPI_Datatype input_mpi_t;
    Build_mpi_type(a_p, b_p, n_p, &input_mpi_t);
    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, input_mpi_t, 0, MPI_COMM_WORLD);
    MPI_Type_free(&input_mpi_t);
} /* Get_input */
```