# Table of Contents

1. **MPI Collective Communications**
   - Parallel Example: Trapezoid Rule
   - Tree Structured Communications
   - Tree-Structure Global Sums
   - Collective Communication: MPI_Reduce
   - Collective Communication: MPI_Allreduce
     - Collective Communication: MPI_Broadcast
     - Data Distribution: Vector Add
     - Collective Communication: MPI_Scatter and MPI_Gather
Recall: The Trapezoid Rule for Numerical Integration

Solve the Integral: \( \int_a^b F(x)dx \)

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

- Partition problem solution into tasks.
- Identify communication channels between tasks.
- Aggregate tasks into composite tasks.
- Map composite tasks to cores
Tasks and communications for Trapezoidal Rule

Master Node collects sums using MPI\_SEND / MPI\_RECV

\( \vartheta (np - 1) \), where \( np \) 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[0] RECV from PE[1]: local estimate=584.199266
PE[0] RECV from PE[2]: local estimate=1466.537954
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=11977.15555
PE[1] SEND: For 12 trapezoids, local estimate=584.199266
PE[2] SEND: For 12 trapezoids, local estimate=1466.537954
PE[3] SEND: For 12 trapezoids, local estimate=2755.471586
PE[4] SEND: For 12 trapezoids, local estimate=4451.000162
PE[5] SEND: For 12 trapezoids, local estimate=6553.123682
PE[6] SEND: For 12 trapezoids, local estimate=9061.842146
PE[7] SEND: For 12 trapezoids, local estimate=11977.15555
With n = 100 trapezoids, our estimate of the integral from 1.000000 to 50.000000 = 3.695778587200000e+04
Reduces the 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.
Tree-structured Global Sum V1

Communication pattern used by 1st version of trap.c algorithm:
7 messages, 7 adds by node 0.
With tree-structured Comm: master has 3 messages and 3 adds.
Tree-structured Global Sum V2.
Collective Communication: MPI_Reduce

```
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);
```

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.: 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: What happens when we have multiple calls to MPI\_Reduce?**

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

- **tinit**: \( p0[a = 0, \ b = 0, \ c = 0, \ d = 0]; \ p1[a = 0, \ b = 0, \ c = 0, \ d = 0]; \ p2[a = 0, \ b = 0, \ c = 0, \ d = 0] \)
- **t0**: \( p0[a = 1, \ b = 0, \ c = 2, \ d = 0]; \ p1[a = 1, \ b = 0, \ c = 2, \ d = 0]; \ p2[a = 1, \ b = 0, \ c = 2, \ d = 0] \)
- **t1**: \( p0[a = 1, \ b = 1, \ c = 2, \ d = 0]; \ p1[a = 1, \ b = 0, \ c = 2, \ d = 2]; \ p2[a = 1, \ b = 1, \ c = 2, \ d = 0] \)
- **on p0**: \( b = P0(a) + P1(c) + P2(a) = 1 + 2 + 1 = 4, \ d = 0 \)
- **t2**: \( p0[a = 1, \ b = 2, \ c = 2, \ d = 2]; \ p1[a = 1, \ b = 1, \ c = 2, \ d = 0]; \ p2[a = 1, \ b = 0, \ c = 2, \ d = 2] \)
- **on p0**: \( d = P0(c) + P1(a) + P2(c) = c + a + c = 2 + 1 + 2 = 5 \)
Collective vs. Point-to-Point Communications

- 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.
- 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 global sum followed by distribution of the result.
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

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

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</td>
<td>1 4 7 10</td>
<td>2 3 8 9</td>
</tr>
<tr>
<td>2</td>
<td>8 9 10</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.

\[ A[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20] \]

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.
Reading and distributing a vector

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

    double* a = NULL;
    int i;

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

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 \n", 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 */;
```