Parallel Patterns

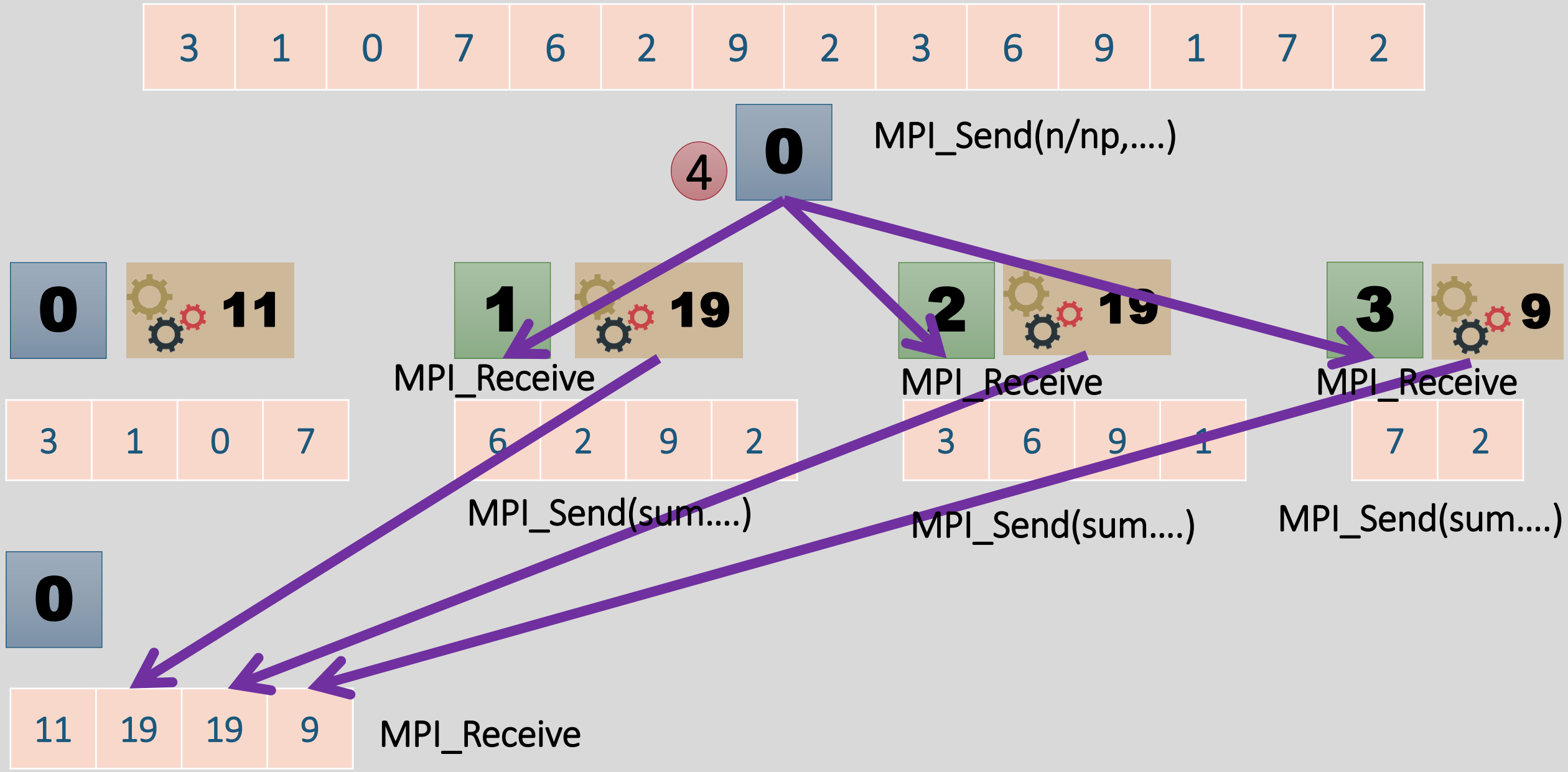
Array Sum, Trapezoidal Rule, Matrix Vector Multiplication

This Week….

- MPI – Array Sum
  - Reduction in MPI
- Trapezoidal Rule
- Matrix - Vector Multiplication

Array Sum in MPI using send and receive

The big picture



Sending Local size first then array

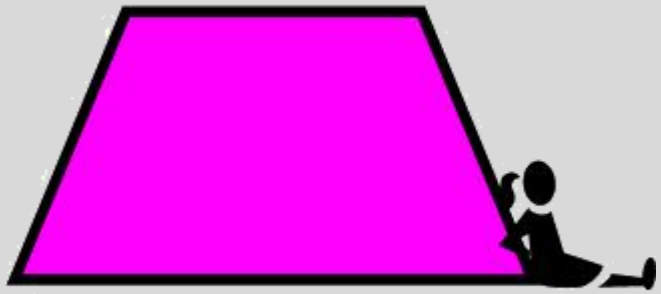
```
1.    for (i = 1; i < np - 1; ++i) {
2.    index = i * local_size;
3.    MPI_Send(&local_size, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
4.    MPI_Send(&arr[index], local_size, MPI_DOUBLE,
5.             i, 1, MPI_COMM_WORLD);
6.    }//end for
7.    // last process adds remaining elements
8.    index = i * local_size;
9.    int elements_left = size - index;

10.   MPI_Send(&elements_left, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
11.   MPI_Send(&arr[index], elements_left, MPI_DOUBLE,
12.            i, 1, MPI_COMM_WORLD);
```

Slave Code receiving local size then elements

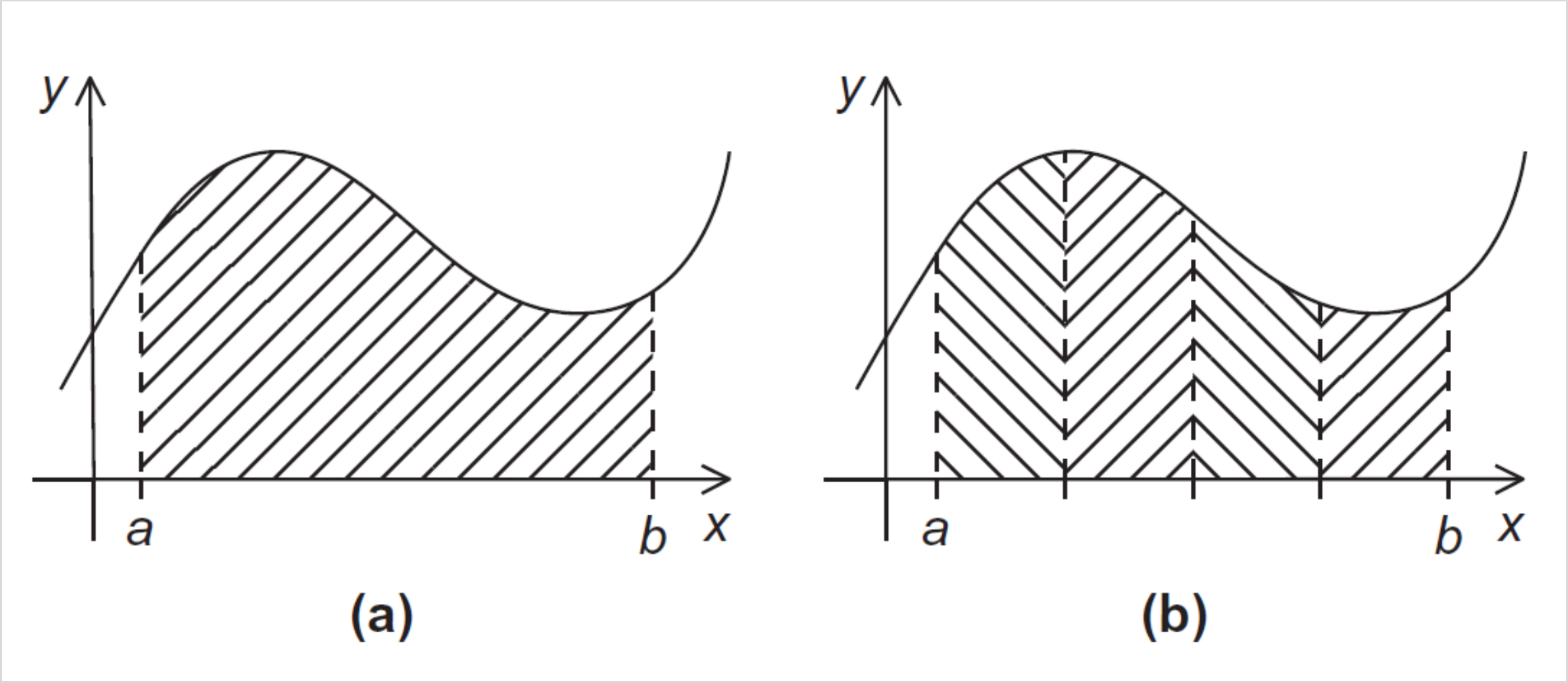
```
1.    else {//slave code
2.    printf("I am slave to He-Man!");
3.    MPI_Recv(&local_size, 1, MPI_INT,
4.          0, 0, MPI_COMM_WORLD, &status);
5.    printf(" He said %d!\n", local_size);
6.    // stores the received array segment in local array
      arr2
7.    double* arr2 = malloc(sizeof(double) * local_size);
8.    MPI_Recv(arr2, local_size, MPI_DOUBLE,
9.                0, 1, MPI_COMM_WORLD, &status);
10.   printf(" Just Received my %d elements! Calculating
      Local sum....\n", local_size);
```

Trapezoidal Rule



Trapezoidal rule in mpi

The Trapezoidal Rule



The Trapezoidal Rule

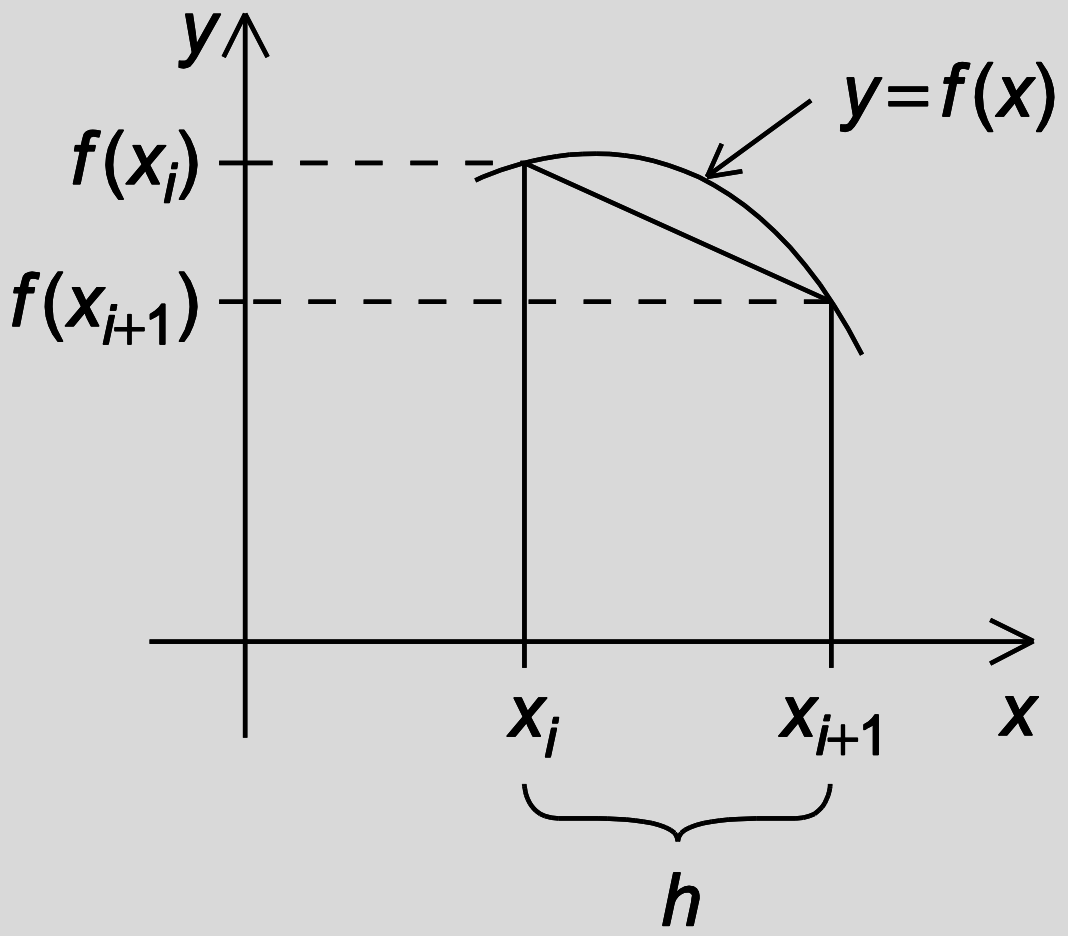
$$\text{Area of one trapezoid } = \frac{h}{2}[f(x_i) + f(x_{i+1})]$$

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

$$x_0 = a,\ x_1 = a+h,\ x_2 = a+2h,\ \ldots,\ x_{n-1} = a+(n-1)h,\ x_n = b$$

$$\text{Sum of trapezoid areas } = h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2]$$

One trapezoid



Pseudo-code for a serial program

```
1.  h = (b-a)/n
2.  estimate = (f(a) + f(b)) / 2.0

3.  for(i = 1; i<= n-1; i++) {
4.      x_i = a + i * h;
5.      estimate += f(x_i);
6.  }//end for

7.  estimate = h*estimate
```

- Height same for all
- 1st and last added once only
- Rest added twice
- Cancels divide by 2.0
- Multiply by height.

Parallelizing the Trapezoidal Rule

- Partition problem solution into tasks.
- Identify communication channels between tasks.
- Aggregate tasks into composite tasks.
- Map composite tasks to cores.

Parallel Code 1

```
1.    Get_input(my_rank, comm_sz, &a, &b, &n);
2.    h = (b - a) / n;
3.    /* h is the same for all processes */
4.    local_n = n / comm_sz;
5.    /* So is the number of trapezoids  */

6.    /* Length of each process' interval of integration
7.     * = local_n*h.  So my interval
8.     * starts at: */
9.    local_a = a + my_rank * local_n * h;
10.   local_b = local_a + local_n * h;
11.   local_int = Trap(local_a, local_b, local_n, h);
```

- Only process **0** gets from std_in, rest receives from **0**.
- Everyone can do it

Parallel Code 2

```
1.      /* Add up the integrals calculated by each process */
2.      if (my_rank != 0)
3.        MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);

4.      else {

5.        total_int = local_int;
6.        for (source = 1; source < comm_sz; source++) {
7.            MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
8.                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
9.            total_int += local_int;

10.       }
11.     }
```

- All process send local_int to process 0
- Process 0 receives and adds up all local_ints.

Get Input

```
1.  void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p,
2.      int* n_p) {
3.      int dest;

4.      if (my_rank == 0) {
5.          printf("Enter a, b, and n\n");
6.          scanf_s("%lf %lf %d", a_p, b_p, n_p);
7.          for (dest = 1; dest < comm_sz; dest++) {
8.              MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
9.              MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
10.             MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
11.         }
12.     } else { /* my_rank != 0 */
13.         MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
14.             MPI_STATUS_IGNORE);
15.         MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
16.             MPI_STATUS_IGNORE);
17.         MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
18.             MPI_STATUS_IGNORE);
19.     }
20. }  /* Get_input */
```

MPI_Reduce

```
1.  int MPIAPI MPI_Reduce(
2.  void           *sendbuf,
3.  void           *recvbuf,
4.  int            count,
5.  MPI_Datatype   datatype,
6.  MPI_Op         op,
7.  int            root,
8.  MPI_Comm       comm
9.  );
```

- The rank of the receiving process

MPI_Op

```
1.    typedef enum _MPI_Op {
2.      MPI_OP_NULL  = 0x18000000,
3.      MPI_MAX      = 0x58000001,
4.      MPI_MIN      = 0x58000003,
5.      MPI_SUM      = 0x58000003,
6.      MPI_PROD     = 0x58000004,
7.      MPI_LAND     = 0x58000005,
8.      MPI_BAND     = 0x58000006,
9.      MPI_LOR      = 0x58000007,
10.     MPI_BOR      = 0x58000008,
11.     MPI_LXOR     = 0x58000009,
12.     MPI_BXOR     = 0x5800000a,
13.     MPI_MINLOC   = 0x5800000b,
14.     MPI_MAXLOC   = 0x5800000c,
15.     MPI_REPLACE  = 0x5800000d
16.   } MPI_Op;
```

Parallel Code 2

```
1.      /* Add up the integrals calculated by each process */
2.      if (my_rank != 0)
3.        MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);

4.      else {

5.        total_int = local_int;
6.        for (source = 1; source < comm_sz; source++) {
7.            MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
8.                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
9.            total_int += local_int;

10.       }
11.     }
```

- All process send local_int to process 0
- Process 0 receives and adds up all local_ints.

Parallel Code 2 using reduce

```
1.    /* Add up the integrals calculated by each process */
2.    MPI_Reduce(
3.       &local_int,
4.       &total_int,
5.       1,
6.       MPI_DOUBLE,
7.       MPI_SUM,
8.       0,
9.       MPI_COMM_WORLD);
```