THE TRAPEZOIDAL RULE IN MPI

- Recall that 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. See Figure 2 below.

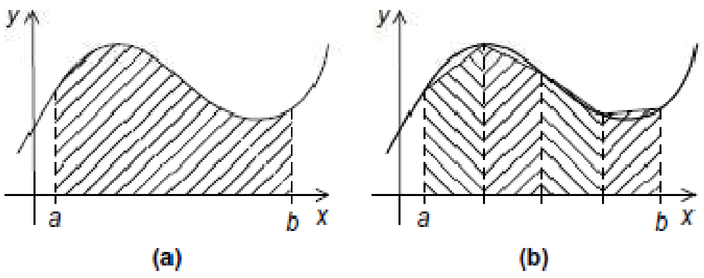

Fig 2: Trapezoidal rule (a) Area to be estimated (b)Approximate Areas using trapezoids

- The basic idea is to divide the interval on the x-axis into n equal subintervals. Then we approximate the area lying between the graph and each subinterval by a trapezoid whose base is the subinterval, whose vertical sides are the vertical lines through the endpoints of the subinterval, and whose fourth side is the secant line joining the points where the vertical lines cross the graph.

- See Figure 3 . If the endpoints of the subinterval are xi and xi+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 (.xi) and f (.x+1), then the area of the trapezoid is

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

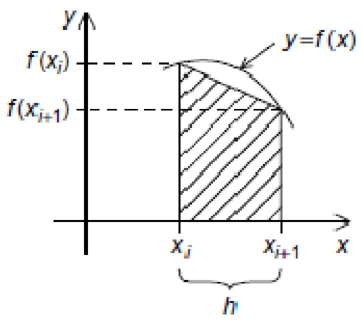

Fig 3: One Trapezoid

- Since we chose the n subintervals so that they would all have the same length, we also know that if the vertical lines bounding the region are x = a and x= b, then

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

- Thus, if we call the leftmost endpoint x0, and the rightmost endpoint xn, we have

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

- and the sum of the areas of the trapezoids our approximation to the total area is

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

- Thus, pseudo-code for a serial program might look something like this:

```
/*Input: a, b, n */
        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;
```

Parallelizing the trapezoidal rule:

- We can design a parallel program using four basic steps:
  1. Partition the problem solution into tasks.
  2. Identify the communication channels between the tasks.
  3. Aggregate the tasks into composite tasks.
  4. Map the composite tasks to cores.

- In the **partitioning phase**, we usually try to identify as many tasks as possible. For the trapezoidal rule, we might identify two types of tasks:
  - one type is finding the area of a single trapezoid,
  - the other is computing the sum of these areas.

- The **communication channels** will join each of the tasks of the first type to the single task of the second type. See Figure 4.

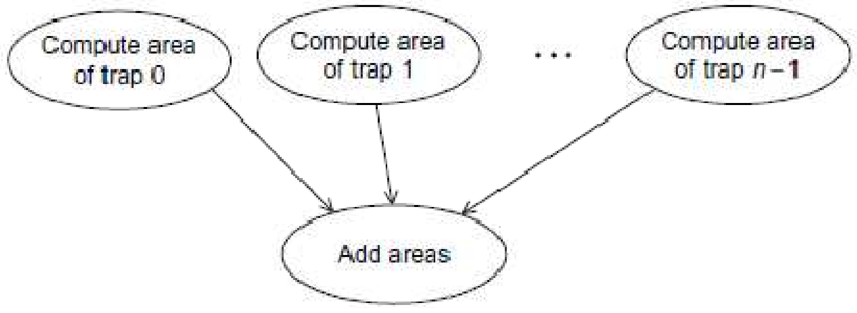

Fig 4: Tasks and communications for the trapezoidal rule

- We **aggregate** the tasks and map them to the cores. we should use many trapezoids, and we will use many more trapezoids than cores. Thus, we need to aggregate the computation of the areas of the trapezoids into groups.

- A natural way to do this is to split the interval [a,b] up into comm_sz subintervals. If comm_sz evenly divides n, the number of trapezoids, we can simply apply the trapezoidal rule with n/ comm_sz trapezoids to each of the comm_sz subintervals.

- To finish, we can have one of the processes, say process 0, add the estimates.
  Let's make the simplifying assumption that comm_sz evenly divides n. Then pseudo-code for the program might look something 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*/
 tota_ 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;
```

- Let's defer, for the moment, the issue of input and just "hardwire" the values for a, b, and n. When we do this, we get the MPI program shown in Program 2.

```
int main(void) {
   int my_rank, comm_sz, n = 1024, local_n;
   double a = 0.0, b = 3.0, h, local_a, local_b;
   double local_int, total_int;
   int source;

   MPI_Init(NULL, NULL);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

   h = (b-a)/n;          /* h is the same for all processes */
   local_n = n/comm_sz;  /* So is the number of trapezoids  */

   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   local_int = Trap(local_a, local_b, local_n, h);

   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;
      }
   }

   if (my_rank == 0) {
      printf("With n = %d trapezoids, our estimate\n", n);
      printf("of the integral from %f to %f = %.15e\n",
          a, b, total_int);
   }
   MPI_Finalize();
   return 0;
} /*  main  */
```

Program 2: First version of the MPI trapezoidal rule

- The Trap function is just an implementation of the serial trapezoidal rule. See Program 3

```
double Trap(
      double left_endpt  /* in */,
      double right_endpt /* in */,
      int    trap_count  /* in */,
      double base_len    /* in */) {
   double estimate, x;
   int i;

   estimate = (f(left_endpt) + f(right_endpt))/2.0;
   for (i = 1; i <= trap_count-1; i++) {
      x = left_endpt + i*base_len;
      estimate += f(x);
   }
   estimate = estimate*base_len;

   return estimate;
} /*  Trap  */
```