THE TRAPEZOIDAL RULE

- The trapezoidal rule for estimating the area under a curve.
  If y = f(x) is a function, and a < b are real numbers, then we can estimate the area between the graph of f(x), the vertical lines x =a and x = b, andthe x-axis by dividing the interval **[a,b] into n** subintervals and approximating thearea over each subinterval by the area of a trapezoid as shown in Fig 1 below.

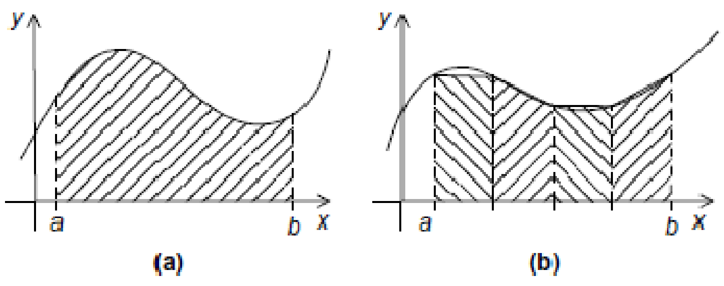
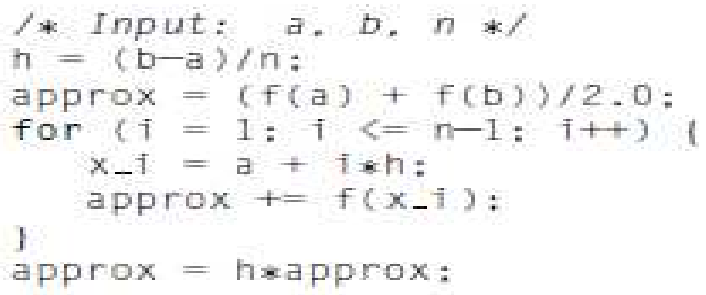


Fig1: THE TRAPEZOIDAL RULE   a) Area to be estimated b) Approx Area using Trapezoids

- If each subinterval has the same length and if we define
  h =(b-a)/n, x_i=a+ih, i= 0, 1, : : : ,n, then our approximation will be

$$h[f(x_0)/2+f(x_1)+f(x_2)+\cdots+f(x_{n-1})+f(x_n)/2].$$

- Thus, we can implement a serial algorithm using the following code:



```
/* 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;
```

A first OpenMP version:

1. We identified two types of tasks:
   a. Computation of the areas of individual trapezoids, and
   b. Adding the areas of trapezoids.
2. There is no communication among the tasks in the first collection, but each taskin the first collection communicates with task 1(b).

3. We assumed that there would be many more trapezoids than cores, so we aggregatedtasks by assigning a contiguous block of trapezoids to each thread (and asingle thread to each core). Effectively, this partitioned the interval [a,b] intolarger subintervals, and each thread simply applied the serial trapezoidal rule toits subinterval as shown in Figure 2.

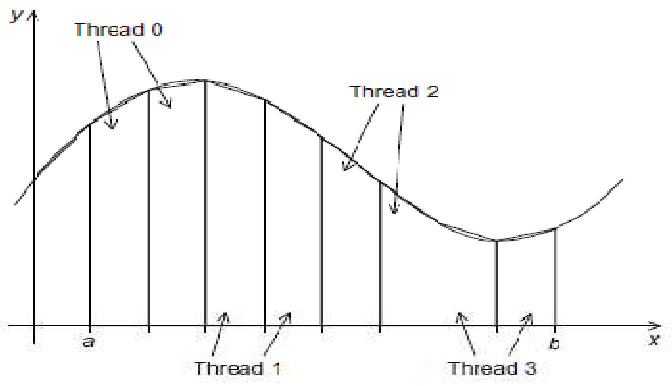
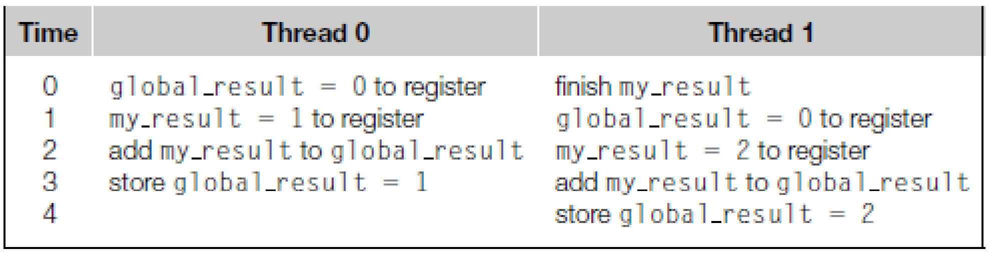


Fig 2: Assignments of Trapezoids to Threads

- A solution is to use a shared variable for the sum of all the threads' results, and each thread can add its (private) result into the shared variable. We would like to have each thread execute a statement that looks something like

  global_result += my_result;

- This can result in an erroneous value forglobal result—if two (or more) threads attempt to simultaneously execute thisstatement, the result will be unpredictable.
- For example, suppose that global_resulthas been initialized to 0, thread 0 has computed my_result = 1, and thread 1has computed my result = 2. Furthermore, suppose that the threads execute the statementglobal_result += my_result according to the following timetable:



Time	Thread 0	Thread 1
0	global_result = 0 to register	finish my_result
1	my_result = 1 to register	global_result = 0 to register
2	add my_result to global_result	my_result = 2 to register
3	store global_result = 1	add my_result to global_result
4		store global_result = 2

- We see that the value computed by thread 0 (my result = 1) is overwritten by thread 1.
- We need some mechanism to make sure that once one thread has started executing global_result += my_result, no other thread can start executing this code until the first thread has finished. In Pthreads we used mutexes or semaphores. In OpenMP we can use the ***critical*** directive

  ```
  # pragmaomp critical
  global result += my result;
  ```

- This directive tells the compiler that the system needs to arrange for the threads tohave mutually exclusive access to the following structured block of code. That is,only one thread can execute the following structured block at a time.
- In the main function, prior to Line 16, the code is single-threaded, and it simply gets the number of threads and the input (a, b, and n).

- In Line 16 the ***parallel*** directive specifies that the ***Trap*** function should be executed by ***thread_count*** threads. After returning from the call to ***Trap***, any new threads that were started by the ***parallel*** directive are terminated, and the program resumes execution with
  only one thread. The one thread prints the result and terminates.

- In the ***Trap*** function, each thread gets its rank and the total number of threads in the team started by the ***parallel*** directive. Then each thread determines thefollowing:
  1. The length of the bases of the trapezoids (Line 32)
  2. The number of trapezoids assigned to each thread (Line 33)

3. The left and right endpoints of its interval (Lines 34 and 35, respectively)
4. Its contribution to global_result (Lines 36–41)

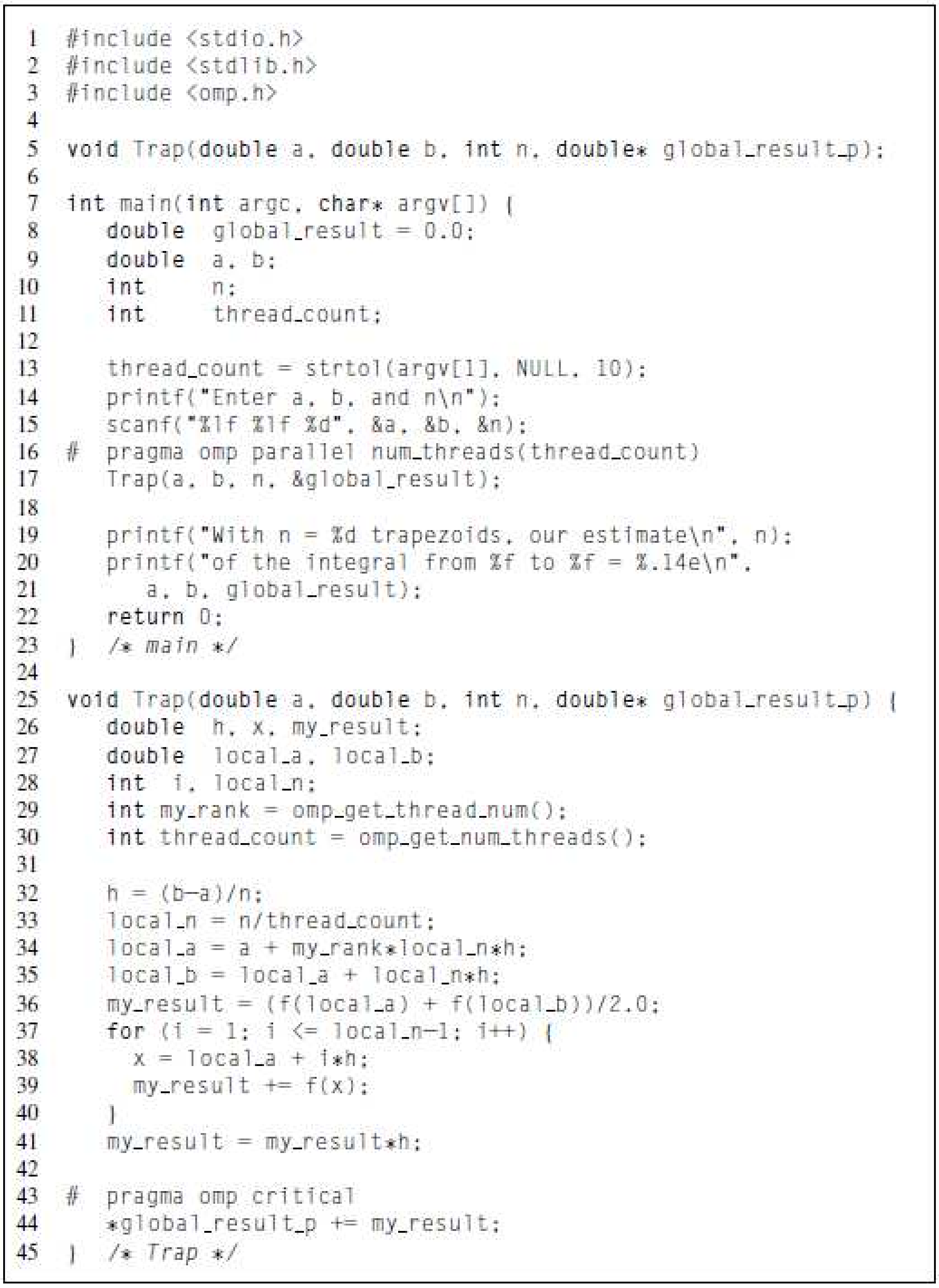
- The threads finish by adding in their individual results to global_result in Lines 43 and 44.

- Notice that unless ***n*** is evenly divisible by thread_count, we'll use fewer than ***n***trapezoids for global_result. For example, if n = 14 and thread_count = 4, eachthread will compute
  Local_n = n/thread _ount = 14/4 = 3.

- Thus each thread will only use 3 trapezoids, and global_result will be computedwith 4x3 = 12 trapezoids instead of the requested 14. So in the error checking(which isn't shown) we check that n is evenly divisible by thread count by doingsomething like this:

  ```
  if(n % thread_count != 0)
  {
  fprintf(stderr, "n must be evenly divisible by thread_countn \n");
  exit(0);
  }
  ```
- Since each thread is assigned a block of local_n trapezoids, the length of each
  thread's interval will be local_n*h, so the left endpoints will be
  thread 0: a + 0_local n*h
  thread 1: a + 1_local n*h
  thread 2: a + 2_local n*h
  . . .
  and in Line 34, we assign
  local_a = a + my rank*local_ n*h;

- Furthermore, since the length of each thread's interval will be local n*h, its rightendpoint will just be
  Local_b = local _a + local_n*h;



```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
   double  global_result = 0.0;
   double  a, b;
   int     n;
   int     thread_count;

   thread_count = strtol(argv[1], NULL, 10);
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", &a, &b, &n);
#  pragma omp parallel num_threads(thread_count)
   Trap(a, b, n, &global_result);

   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.14e\n",
      a, b, global_result);
   return 0;
}  /* main */

void Trap(double a, double b, int n, double* global_result_p) {
   double  h, x, my_result;
   double  local_a, local_b;
   int  i, local_n;
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   h = (b-a)/n;
   local_n = n/thread_count;
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   my_result = (f(local_a) + f(local_b))/2.0;
   for (i = 1; i <= local_n-1; i++) {
     x = local_a + i*h;
     my_result += f(x);
   }
   my_result = my_result*h;

#  pragma omp critical
   *global_result_p += my_result;
}  /* Trap */
```

Fig: OpenMP trapezoidal rule program