The trapezoidal rule

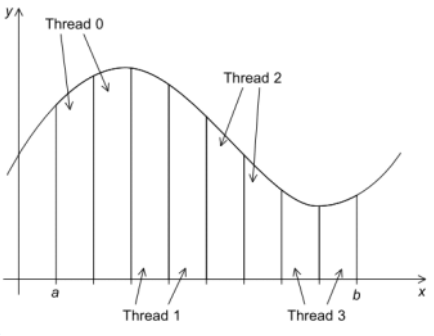
Summary of what the trapezoidal rule is:

Let's take a look at a somewhat more useful (and more complicated) example: the trapezoidal rule for estimating the area under a curve. Recall from Section 3.2 that if $y = f(x)$ is a reasonably nice 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 $x = b$, and the x-axis by dividing the interval $[a, b]$ into n subintervals and approximating the area over each subinterval by the area of a trapezoid. See Fig. 5.3.

Also recall that if each subinterval has the same length and if we define $h = (b - a)/n$, $x_i = a + ih$, $i = 0, 1, \ldots, n$, 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;
```

openMPI trapezoidal rule:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  void Trap(double a, double b, int n, double* global_result_p);
6
7  int main(int argc, char* argv[]) {
8     /* We'll store our result in global_result: */
9     double  global_result = 0.0;
10    double  a, b;  /* Left and right endpoints   */
11    int     n;     /* Total number of trapezoids */
12    int     thread_count;
13
14    thread_count = strtol(argv[1], NULL, 10);
15    printf("Enter a, b, and n\n");
16    scanf("%lf %lf %d", &a, &b, &n);
17 #  pragma omp parallel num_threads(thread_count)
18    Trap(a, b, n, &global_result);
19
20    printf("With n = %d trapezoids, our estimate\n", n);
21    printf("of the integral from %f to %f = %.14e\n",
22       a, b, global_result);
23    return 0;
24 }  /* main */
25
26 void Trap(double a, double b, int n, double* global_result_p) {
27    double  h, x, my_result;
28    double  local_a, local_b;
29    int  i, local_n;
30    int my_rank = omp_get_thread_num();
31    int thread_count = omp_get_num_threads();
32
33    h = (b-a)/n;
34    local_n = n/thread_count;
35    local_a = a + my_rank*local_n*h;
36    local_b = local_a + local_n*h;
37    my_result = (f(local_a) + f(local_b))/2.0;
38    for (i = 1; i <= local_n-1; i++) {
39      x = local_a + i*h;
40      my_result += f(x);
41    }
42    my_result = my_result*h;
43
44 #  pragma omp critical
45    *global_result_p += my_result;
46 }  /* Trap */
```

Program 5.2: First OpenMP trapezoidal rule program.

We use the prefix `local_` for some variables to emphasize that their values may differ from the values of corresponding variables in the `main` function—for example, `local_a` may differ from `a`, although it is the *thread's* left endpoint.

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, each thread will compute

```
local_n = n/thread_count = 14/4 = 3.
```

Thus each thread will only use 3 trapezoids, and `global_result` will be computed with $4 \times 3 = 12$ 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 doing something like this:

```
if (n % thread_count != 0) {
   fprintf(stderr,
      "n must be evenly divisible by thread_count\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_h*h
   . . .
```

So in Line 35, 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 right endpoint will just be

```
local_b = local_a + local_n*h;
```

Above is some clarifying info about what the program is doing, most notably…

- What does local_[something] mean?
- What code figures out what threads take care of what intervals [local_a,local_b]
- Error checking in case we don't get an evenly divisible n

Turning it into a parallel program using openMPI

- Using Fosters methodology to turn problem into a parallel solution

Recall that we applied Foster's parallel program design methodology to the trapezoidal rule as described in the following list (see Section 3.2.2):

1. We identified two types of jobs:
   a. Computation of the areas of individual trapezoids, and
   b. Adding the areas of trapezoids.
2. There is no communication among the jobs in the first collection, but each job in the first collection communicates with job 1b.
3. We assumed that there would be many more trapezoids than cores, so we aggregated jobs by assigning a contiguous block of trapezoids to each thread (and a single thread to each core).[2] Effectively, this partitioned the interval $[a, b]$ into larger subintervals, and each thread simply applied the serial trapezoidal rule to its subinterval. See Fig. 5.4.

- We aren't quite done, however, since we still need to add up the threads' results
  - We can use a shared variable that all the threads can put their results into like this:

```
global_result += my_result;
```

    - However this would cause a race condition
  - We also know that this line of code would be called a critical section.
  - We therefore 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:

```
#  pragma omp critical
   global_result += my_result;
```

    - This directive tells the compiler that the system needs to arrange for the threads to have mutually exclusive access to the following structured block of code.
- Summary of how the program works:
  In the main function, prior to Line 17, the code is single-threaded, and it simply gets the number of threads and the input (a, b, and n). In Line 17 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 the following:
    1. The length of the bases of the trapezoids (Line 33),
    2. The number of trapezoids assigned to each thread (Line 34),
    3. The left and right endpoints of its interval (Lines 35 and 36, respectively)
    4. Its contribution to `global_result` (Lines 37–42).

    The threads finish by adding in their individual results to `global_result` in Lines 44–45.