

An Application: Numerical Integration

NOW THAT WE KNOW HOW TO SEND MESSAGES with MPI, let's write a program that uses message passing to solve a problem: calculate a definite integral with the trapezoidal rule. If you remember the trapezoidal rule, you can skip section 4.1.

4.1 The Trapezoidal Rule

Recall that the definite integral from a to b of a nonnegative function $f(x)$ can be thought of as the area bounded by the x -axis, the vertical lines $x = a$ and $x = b$, and the graph of the function $f(x)$. See Figure 4.1.

One approach to estimating this area or integral is to partition the region into regular geometric shapes and then add the areas of the shapes. In the trapezoidal rule, the regular geometric shapes are trapezoids; each trapezoid has its base on the x -axis, vertical sides, and its top edge joining two points on the graph of $f(x)$. See Figure 4.2.

For our purposes, we'll choose all the bases to have the same length. So if there are n trapezoids, the base of each will be $h = (b - a)/n$. The base of the leftmost trapezoid will be the interval $[a, a + h]$; the base of the next trapezoid will be $[a + h, a + 2h]$; the next, $[a + 2h, a + 3h]$; etc. In general, the base of the i th trapezoid will be $[a + (i - 1)h, a + ih]$, $i = 1, \dots, n$. In order to simplify notation, let x_i denote $a + ih$, $i = 0, \dots, n$. Then the length of the left side of the i th trapezoid will be $f(x_{i-1})$, and its right side will be $f(x_i)$. See Figure 4.3. Thus, the area of the i th trapezoid will be

Parallel Programming with MPI
by Peter S. Pacheco, Morgan Kaufmann
1997

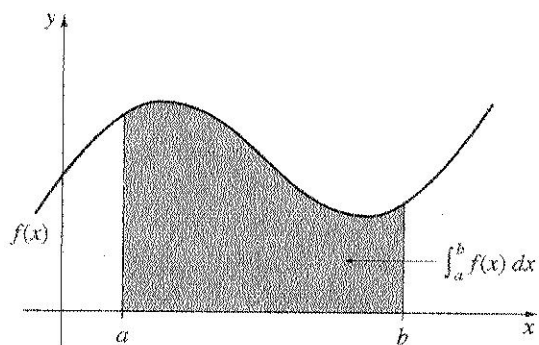


Figure 4.1 Definite integral of a nonnegative function

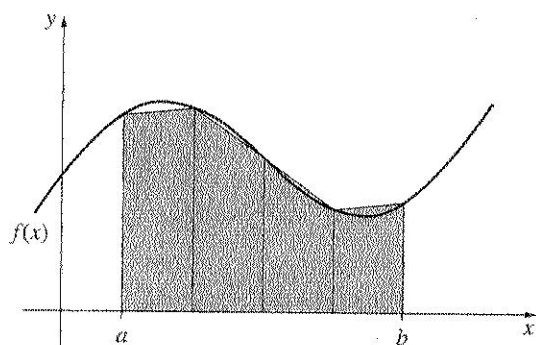


Figure 4.2 Trapezoids approximating definite integral

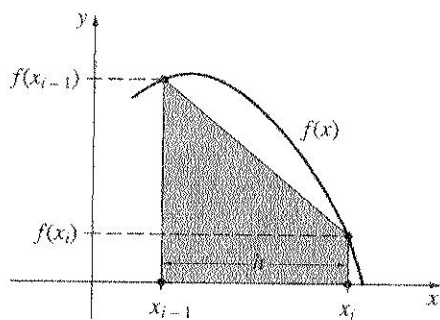


Figure 4.3 The i th trapezoid

$$\frac{1}{2}h[f(x_{i-1}) + f(x_i)],$$

and the area of our entire approximation will be the sum of the areas of the trapezoids:

$$\begin{aligned} & \frac{1}{2}h[f(x_0) + f(x_1)] + \frac{1}{2}h[f(x_1) + f(x_2)] + \cdots + \frac{1}{2}h[f(x_{n-1}) + f(x_n)] \\ &= \frac{h}{2}[f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + f(x_n)] \\ &= [f(x_0)/2 + f(x_n)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1})]h. \end{aligned}$$

So by putting $f(x)$ into a subprogram, we can write a *serial* program for calculating an integral using the trapezoidal rule.

```
/* Calculate definite integral using trapezoidal rule.
 * The function f(x) is hardwired.
 * Input: a, b, n.
 * Output: estimate of integral from a to b of f(x)
 *        using n trapezoids.
 */

#include <stdio.h>

main() {
    float integral; /* Store result in integral */
    float a, b;     /* Left and right endpoints */
    int n;          /* Number of trapezoids */
    float h;        /* Trapezoid base width */
    float x;
    int i;

    float f(float x); /* Function we're integrating */

    printf("Enter a, b, and n\n");
    scanf("%f %f %d", &a, &b, &n);

    h = (b-a)/n;
    integral = (f(a) + f(b))/2.0;
    x = a;
    for (i = 1; i <= n-1; i++) {
        x = x + h;
        integral = integral + f(x);
    }
    integral = integral*h;

    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %f\n",
        a, b, integral);
} /* main */
```

Table 4.1

Assignment of subintervals to processes

Process	Interval
0	$[a, a + \frac{n}{p}h]$
1	$[a + \frac{n}{p}h, a + 2\frac{n}{p}h]$
$\vdots$	$\vdots$
i	$[a + i\frac{n}{p}h, a + (i + 1)\frac{n}{p}h]$
$\vdots$	$\vdots$
$p - 1$	$[a + (p - 1)\frac{n}{p}h, b]$

```

float f(float x) {
    float return_val;
    /* Calculate f(x). Store calculation in return_val. */
    ;
    return return_val;
} /* f */

```

4.2 Parallelizing the Trapezoidal Rule

As we saw in Chapter 2, there are several approaches to parallelizing a serial program. Perhaps the simplest approach distributes the data among the processes, and each process runs essentially the same program on its share of the data. In our case, the data is just the interval $[a, b]$ and the number of trapezoids n . So we can parallelize the trapezoidal rule program by assigning a subinterval of $[a, b]$ to each process, and having that process estimate the integral of f over the subinterval. In order to calculate the integral over $[a, b]$, the processes' local calculations are added.

An obvious question here is, How does each process know which subinterval it should integrate over, and how many trapezoids it should use? In order to answer this, suppose there are p processes and n trapezoids, and, in order to simplify the discussion, also suppose that n is evenly divisible by p . Then it is natural for the first process to calculate the area of the first n/p trapezoids, the second process to calculate the area of the next n/p , etc. Recall that MPI identifies each process by a nonnegative integer. So if there are p processes, the first is process 0, the second process 1, ..., and the last process $p - 1$. Using the notation we developed in our discussion of the serial program, we have each process calculating integrals over the subintervals indicated in Table 4.1.

Thus each process needs the following information:

- The number of processes, p
- Its rank
- The entire interval of integration, $[a, b]$
- The number of subintervals, n

Recall from Chapter 3 that the first two items can be found by calling the MPI functions `MPI_Comm_size` and `MPI_Comm_rank`. The last two items should probably be input by the user. But this (perhaps surprisingly) can raise some difficult problems. So for our first attempt at calculating the integral, let's "hardwire" these values by simply setting their values with assignment statements.

A second obvious question is, How are the individual processes' calculations added up? One straightforward approach would be to send each process's result to, say, process 0, and have process 0 do the final addition.

With these assumptions we can write our first "real" MPI program.

```

/* Parallel Trapezoidal Rule
*
* Input: None.
* Output: Estimate of the integral from a to b of f(x)
*         using the trapezoidal rule and n trapezoids.
*
* Algorithm:
*   1. Each process calculates "its" interval of
*      integration.
*   2. Each process estimates the integral of f(x)
*      over its interval using the trapezoidal rule.
*   3a. Each process != 0 sends its integral to 0.
*   3b. Process 0 sums the calculations received from
*       the individual processes and prints the result.
*
* Note: f(x), a, b, and n are all hardwired.
*/
#include <stdio.h>

/* We'll be using MPI routines, definitions, etc. */
#include "mpi.h"

main(int argc, char** argv) {
    int    my_rank;    /* My process rank          */
    int    p;          /* The number of processes */
    float  a = 0.0;    /* Left endpoint           */
    float  b = 1.0;    /* Right endpoint          */
    int    n = 1024;   /* Number of trapezoids    */
    float  h;          /* Trapezoid base length   */
    float  local_a;    /* Left endpoint my process */

```

```

float    local_b;    /* Right endpoint my process */
int      local_n;    /* Number of trapezoids for */
                        /* my calculation */
float    integral;   /* Integral over my interval */
float    total;      /* Total integral */
int      source;     /* Process sending integral */
int      dest = 0;   /* All messages go to 0 */
int      tag = 0;
MPI_Status status;

float Trap(float local_a, float local_b, int local_n,
           float h); /* Calculate local integral */

/* Let the system do what it needs to start up MPI */
MPI_Init(&argc, &argv);

/* Get my process rank */
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

/* Find out how many processes are being used */
MPI_Comm_size(MPI_COMM_WORLD, &p);

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

/* Length of each process's interval of
 * integration = local_n*h. So my interval
 * starts at: */
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
integral = Trap(local_a, local_b, local_n, h);

/* Add up the integrals calculated by each process */
if (my_rank == 0) {
    total = integral;
    for (source = 1; source < p; source++) {
        MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
                 MPI_COMM_WORLD, &status);
        total = total + integral;
    }
} else {
    MPI_Send(&integral, 1, MPI_FLOAT, dest,
             tag, MPI_COMM_WORLD);
}

/* Print the result */
if (my_rank == 0) {
    printf("With n = %d trapezoids, our estimate\n",
           n);
}

```

```

        printf("of the integral from %f to %f = %f\n",
               a, b, total);
    }

    /* Shut down MPI */
    MPI_Finalize();
} /* main */

float Trap(
    float local_a /* in */,
    float local_b /* in */,
    int local_n /* in */,
    float h /* in */) {

    float integral; /* Store result in integral */
    float x;
    int i;

    float f(float x); /* function we're integrating */

    integral = (f(local_a) + f(local_b))/2.0;
    x = local_a;
    for (i = 1; i <= local_n-1; i++) {
        x = x + h;
        integral = integral + f(x);
    }
    integral = integral*h;
    return integral;
} /* Trap */

float f(float x) {
    float return_val;
    /* Calculate f(x). */
    /* Store calculation in return_val. */
    :
    return return_val;
} /* f */

```

Observe that this program also uses the SPMD paradigm. Even though process 0 executes an essentially different set of commands from the remaining processes, it still runs the same program. The different commands are executed by branching based on the process rank.

Also note that we were careful to distinguish between variables whose contents were significant on all the processes, and variables whose contents

were only significant on individual processes. Examples of the former are `a`, `b`, and `n`. Examples of the latter are `local_a`, `local_b`, and `local_n`. Variables whose contents are significant on all the processes are sometimes called **global variables**, and variables whose contents are significant only on individual processes are sometimes called **local variables**. If you learned to program in Pascal, this terminology may at first seem somewhat confusing. However, it's usually very easy to tell from the context which meaning is implied.

It's extremely important that we, as programmers, distinguish between global and local variables; it can be very difficult or impossible to decipher a program that makes no distinction between the two. One of the most insidious things a parallel programmer can do is to use the same variable for both global and local storage with no documentation. In general, separate variables should be allocated for global and local scalar variables. For composite variables it may be necessary to use the same storage for both global and local variables. However, if this is done, it should be clearly documented.

4.3 I/O on Parallel Systems

One obvious problem with our program is its lack of generality. The function, $f(x)$, and the input data, `a`, `b`, and `n`, are hardwired. So if we want to change any of these, we must edit and recompile the program. Different functions can be incorporated by revising the `Trap` function so that it takes an additional parameter—a pointer to a function. Since this has nothing to do with parallel computing, we'll leave it as an exercise to modify the program to use function pointers. However, the issue of changing the input data has everything to do with parallel computing. So we should take a look at it.

In our greetings and serial trapezoidal programs we assumed that process 0 could both read from standard input (the keyboard) and write to standard output (the terminal screen). Many parallel systems provide this much I/O. In fact, many parallel systems allow all processors to both read from standard input and write to standard output. So what's the problem?

In the first place, we were careful to say "many" (not "all") systems provide this much I/O. But even if we could say "all," there would still be issues that need to be resolved.

Let's look at an example. Suppose we modify the trapezoidal program so that each process attempts to read the values `a`, `b`, and `n` by adding the statement

```
scanf("%f %f %d", &a, &b, &n);
```

Suppose also that we run the program with two processes and the user types in

```
0 1 1024
```