Sample Program: Trapezoidal Rule

Following Foster's methodology, we

1. partition the problem (find the area of a single trapezoid, add them up)

2. identify communication (information about single trapezoid scattered, single areas gathered)

3. aggregate tasks (there are probably more trapezoids than cores, so split [a, b] into `comm_sz` subintervals)

4. map tasks to cores (subintervals to cores; results back to process 0)

Sample Program: Trapezoidal Rule

```
/* File:     mpi_trap1.c
 * Purpose:  Use MPI to implement a parallel version of the trapezoidal
 *           rule.  In this version the endpoints of the interval and
 *           the number of trapezoids are hardwired.
 *
 * Input:    None.
 * Output:   Estimate of the integral from a to b of f(x)
 *           using the trapezoidal rule and n trapezoids.
 *
 * Compile:  mpicc -g -Wall -o mpi_trap1 mpi_trap1.c
 * Run:      mpiexec -n <number of processes> ./mpi_trap1
 *
 * 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.
 *
 * IPP:   Section 3.2.2 (pp. 96 and ff.)
 */
#include <stdio.h>

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

/* Calculate local integral  */
double Trap(double left_endpt, double right_endpt, int trap_count,
   double base_len);
```

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

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

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

   /* 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, &comm_sz);

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

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

   /* Add up the integrals calculated by each process */
   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;
      }
   }

   /* Print the result */
   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);
   }

   /* Shut down MPI */
   MPI_Finalize();

   return 0;
} /*  main  */


/*--------------------------------------------------------------------
 * Function:     Trap
 * Purpose:      Serial function for estimating a definite integral
 *               using the trapezoidal rule
 * Input args:   left_endpt
 *               right_endpt
 *               trap_count
 *               base_len
 * Return val:   Trapezoidal rule estimate of integral from
 *               left_endpt to right_endpt using trap_count
 *               trapezoids
 */
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  */


/*--------------------------------------------------------------------
 * Function:    f
 * Purpose:     Compute value of function to be integrated
 * Input args:  x
 */
double f(double x) {
   return x*x;
} /* f */
```