DREXEL UNIVERSITY
Electrical and Computer Engineering
College of Engineering

Drexel University

Electrical and Computer Engineering Dept.

Parallel Computer Architecture ECEC-622

TITLE: PThreads Assignment #2: Trapezoidal Rule Integration

GROUP: Mark Klobukov and Greg Matthews

INSTRUCTOR: Dr. Kandasamy

DATE SUBMITTED: 2/13/2017

DATE DUE: 2/13/2017

Problem Description:

The objective of this assignment was to perform numerical integration using trapezoidal rule. The rule calculates the area under a curve using the following formula:

$$\int_a^b f(x)\,\mathrm{d}x = h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2].$$

The program accepts the number of threads, number of trapezoids to be used for calculation, and the bounds of integration.

Approach to parallelization:

In the above formula, $f(x_0)$ and $f(x_n)$ are divided by 2. In order to avoid unnecessary overhead introduced by evaluation of the if-statement by each thread, the computation of these values was placed outside the parallel region of the code. Each thread calculated a partial sum, and after the threads were joined, the total sum was found. The entire sum was multiplied by h, the trapezoid height, as indicated in the formula above.

Code explanation:

The following data structure was used to pass arguments to each worker thread:

```
//data struture for passing arguments for each worker thread
typedef struct args_for_thread_s {
    float a; //beginning of integration
    float b; //end of integration
    float h; //trapezoid height;
    int thread_id; //thread id
    double my_sum; //partial sum of the given thread
    unsigned int start_index; //where on the x-axis thread starts
    unsigned int end_index; //where on the x-axis thread ends
} ARGS_FOR_THREAD;
```

This includes the bounds of integration, the individual thread ID, the thread's partial sum, and its starting/ending indexes for the loop similar to the one provided in **compute_gold()**

Compute_using_pthreads() function explanation:

At the beginning of the function, the data structure for each thread's arguments was initialized.

```
/* Complete this function to perform the trapezoidal rule on the GPU. */
double compute_using_pthreads(float a, float b, int n, float h)
{
      double sum = 0.0;
      pthread_t thread_id[NUM_THREADS];
      pthread_attr_t attributes; //thread attr
      pthread_attr_init(&attributes); //init thread attributes to default
      ARGS_FOR_THREAD * args_for_thread[NUM_THREADS];
      unsigned int i;
      for (i = 0; i < NUM_THREADS; i++) {
        args_for_thread[i] = (ARGS_FOR_THREAD *)malloc(sizeof(ARGS_FOR_THREAD));
        args_for_thread[i]->a = a;
        args_for_thread[i]->b = b;
        args_for_thread[i]->h = h;
        args_for_thread[i]->thread_id = i; //thread id
        args_for_thread[i]->my_sum = 0.0; //init sum
        args_for_thread[i]->start_index = 1 + (NUM_TRAPEZOIDS/NUM_THREADS)*i;
        args_for_thread[i]->end_index = 1 + (NUM_TRAPEZOIDS/NUM_THREADS) * (i+1);
        }
```

Then, the NUM_THREADS threads are created, and each one executes the compute_my_traps() function. After the threads are joined, their partial sums are added together and multiplied by the trapezoid height.

```
   //create threads
   for (i = 0; i< NUM_THREADS; i++) {
     pthread_create(&thread_id[i], &attributes, compute_my_traps, (void*) args_for_th
read[i]);
   }


   //join threads
   for (i = 0; i < NUM_THREADS; i++) {
     pthread_join(thread_id[i], NULL);
   }

   for (i = 0; i < NUM_THREADS; i++) {
     sum += args_for_thread[i]->my_sum ;
   }
   sum = (sum + (f(a) + f(b)) /2.0) * h;

   return sum;
}
```

Compute_my_traps() function description:

This is the function that each worker thread executes. The start and end index arguments are provided to each thread during the function call. The if-statement is necessary to make sure that the last thread does not attempt to access elements outside of the upper limit of integration. The for-loop at the end of the function follows the syntax of **compute_gold()** exactly.

```
void * compute_my_traps(void * args) {
  //types cast the thread's arguments
  ARGS_FOR_THREAD * my_args = (ARGS_FOR_THREAD*) args;
  float h = my_args->h;
  float a = my_args->a;
  unsigned int end = my_args->end_index;
  unsigned int start = my_args->start_index;
  //take care of a potentially smaller chunk at the end
  if (my_args->thread_id == (NUM_THREADS - 1)) {
   end = NUM_TRAPEZOIDS;
  }

  for (unsigned int k = start; k < end; k++) {
    my_args->my_sum += f(a  + k*h);
  }
  //multiply each thread's sum by h and add to the overall sum in the thread-creating
 function
  pthread_exit((void*) 0);
}
                                                                166,1         Bot
```

Results:

Table 1: Serial vs Parallel Implementation Results

# Threads	Time Serial (s)	Time Parallel (s)	Speedup
2	4.052	2.066	1.961
4	4.052	1.923	2.107
8	4.050	1.270	3.188
16	4.056	0.890	4.556

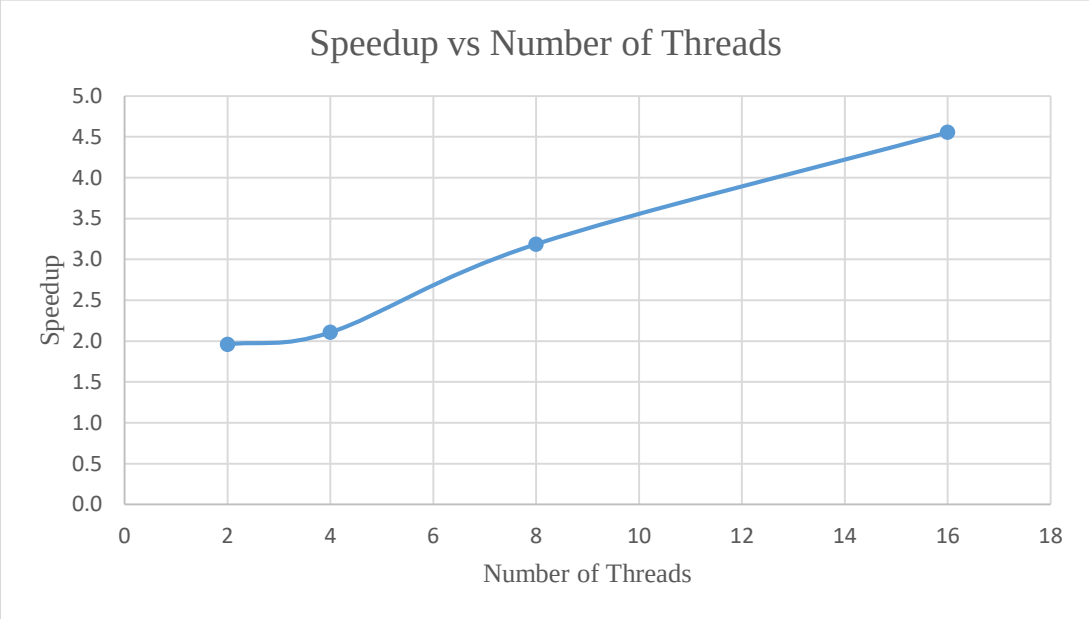

Figure 1: Graph of speedup vs number of threads

As per Table 1, the serial and parallel implementation of the trapezoidal integration algorithm were tested using different number of threads. In each case, the parallel implementation was efficient in its use of CPU time. As Figure 1 shows, an increase in the number of threads positively correlates to a greater computation speed.

Compile the program as follows:

gcc -o trap trap.c -lpthread -lm -std=c99

Example compilation and execution of the program with 100,000,000 trapezoids using 16 threads:

```
[mk3396@xunil-05 A4]$ gcc -o trap trap.c -lpthread -lm -std=c99
[mk3396@xunil-05 A4]$ ./trap
The height of the trapezoid is 0.000010
Reference solution computed on the CPU = 997.531321
CPU run time = 4.05582 s.
Solution computed using pthreads = 997.531321
Parallel run-time = 0.89016
Speedup = 4.55626
```