David Perry CS525 Parallel Computing January 30, 2014

1.)

Omp-trap-pil.c (Omp built-in reduction)

Number of Threads	Runtime	Speedup
1	4.52e+00	1.0
2	2.26e+00	2.0
4	1.13e+00	4.0
8	5.77e-01	7.8336
16	6.12e-01	7.8336

Trapezoid.c

Number of Threads	Runtime	Speedup
1	4.52e+00	1.0
2	3.86e+00	1.17098
4	2.81e+00	1.60854
8	2.48e+00	1.82258
16	2.75e+00	1.6436

Increasing the number of threads drastically reduced the runtime of the program that uses openmp's built-in reduction function close to a factor of the number of the threads being used. However, the runtime did not decrease at all and actually increased when using more than 8 threads. This is because the machines used for testing are only capable of running 8 threads at a time. Therefore, no speedup is seen but more overhead to setup the extra threads is required resulting in a longer runtime.

Increasing the number of threads did increase speedup on the trapezoid.c program. However, the speedup was not large and extremely inconsistent. Again, the largest speedup was seen while using 8 threads as a total of 8 threads can be ran at a time on the testing machine.

The program using openmp's built-in reduction function is much faster for each number of threads except 1. This occurs because my program's implementation of the reduction operation has not been optimized for parallelization like openmp's reduction function.

2.)

One Thread

Number of Intervals	Runtime	Error
16	1.01e-04	2.3649704417039175e-09
32	8.89e-05	3.6957104043722211e-11
64	1.03e-04	5.7820415122478153e-13
128	9.61e-05	9.7699626167013776e-15
256	1.02e-04	-8.8817841970012523e-16

Two Threads

Number of Intervals	Runtime	Error
16	1.57e-04	2.3649713298823372e-09
32	1.56e-04	3.6957104043722211e-11
64	1.64e-04	5.7820415122478153e-13
128	1.59e-04	8.8817841970012523e-15
256	1.52e-04	-8.8817841970012523e-16

There was little to no difference in the calculated error seen between the program while using 1 or 2 threads. This is expected as Simpson's rule requires less iterations through its calculation and therefore the calculations are not largely impacted by using another thread. However, the calculated error is largely reduced when the number of intervals is increased for both 1 and 2 threads. This occurs because when more intervals are used more area under the function curve is covered in the approximation. Therefore, the summation of these intervals will be closer to the actual value of the integral.

The runtime of the both 1 and 2 threads did not fluctuate much by increasing the intervals. This is due to the fact that Simpson's algorithm does not perform a large number of iterations of its summation loop and therefore does not gain a large benefit from parallelization. However, comparison between the runtimes while using 1 and 2 threads shows that 1 thread produces a faster runtime. Again, because Simpson's rule does not largely benefit from parallelization, the creation of extra threads adds overhead that actually makes its runtime worse.

3.)

(a)

```
#pragma omp parallel
for (i=0; i< k; i++) {
     A[i] = A[i] + A[i+k];
}
```

This program is very suited for parallelism. As the loop iterates no index in array A is accessed twice and no index in array A is needed once its new value has been assigned. Therefore, no race conditions occur. This can be seen in the following unrolling of the loop:

A[0] = A[0] + A[0+k]
A[1] = A[1] + A[1+k]
A[2] = A[2] + A[2+k]
.
.
.
A[k-1] = A[k-1] + A[2k-1](b)

While parallelization would work in this function where k < number of threads available this would rarely be the case. In a more practical case where k > number of threads this function is not suitable for parallelization. This is true because race conditions can occur when accessing the array A. Consider the following unrolling of the loop:

A[k] = A[k] + A[0]
A[k+1] = A[k+1] + A[1]
A[k+2] = A[k+2] + A[2]
.
.
.
A[k+100] = A[k+100] + A[100]

Now consider an execution where k=100 in the above unrolling. This execution would result in the first operation shown accessing and writing to A[100] as well as the last operation accessing A[100]. If these two operations were on two separate threads a race condition would occur. Therefore, this program is not well suited for parallelization.

4.)

```
#pragma omp parallel
for (i=0; i< m; i++) {
     cterm[i] = 0;
     for (j=0; j<n; j++)
          cterm[i] += A[i][2j] * A[i][2j+1];
}
```

The above function accesses A in a row oriented fashion. Parallelization is simple since every possible thread spawned gets a different set of values for i to iterate through. Therefore race conditions do not occur.

This can code be changed to work in a column oriented fashion. Consider the code below:

```
for (i = 0; i < m; i++)
     cterm[i] = 0;
for (j=0; j<n; j++) {
     for (i=0; i< m; i++)
          cterm[i] += A[i][2j] * A[i][2j+1];
}
```

However, this code is no longer easy to parallelize and modifications must be made. The code below shows the column oriented parallel program implemented using openmp:

```
for (i = 0; i < m; i++) {
     for (j =0; j < omp_get_max_threads(); j++)
          cterm[i][j] = 0;
#pragma omp parallel
for (j=0; j<n; j++) {
     for (i=0; i< m; i++)
          cterm[i][omp_get_thread_num()] += A[i][2j] * A[i][2j+1];
}
#pragma omp parallel
for (i=0; i<m; i++) {
     result[i] = 0;
     for (j=0; j < omp_get_max_threads(); j++)
          result[i] += cterm[i][j]
}
```

The basic idea for the above code is that an extra dimension of the original cterm array must be created to store the values that have been summed up on each individual thread. Once these summations are done this extra dimension must be iterated through for each i value calculating that specific i values sum which can be done in parallel.