Project 4: Parallel Task in OpenMP

Tristin Greenstein Anthony M Vladimir S Jefferson Perez Diaz

April 25, 2021

1 Numerical Integration:

Numerical Integration is a collection of algorithms used to find the value of a numerical integral. The method broken down can be described as combining evaluations of the integrand to get an approximate value of the integral. The general mathematical formula is shown below:

$$\int_a^b p(x)f(x)dx \cong \sum_{j=1}^{N} C_j f(x_j).$$

We programmed the Numerical Integration in C. We then created a copy of the code to function in parallel by implementing pragma openmp.

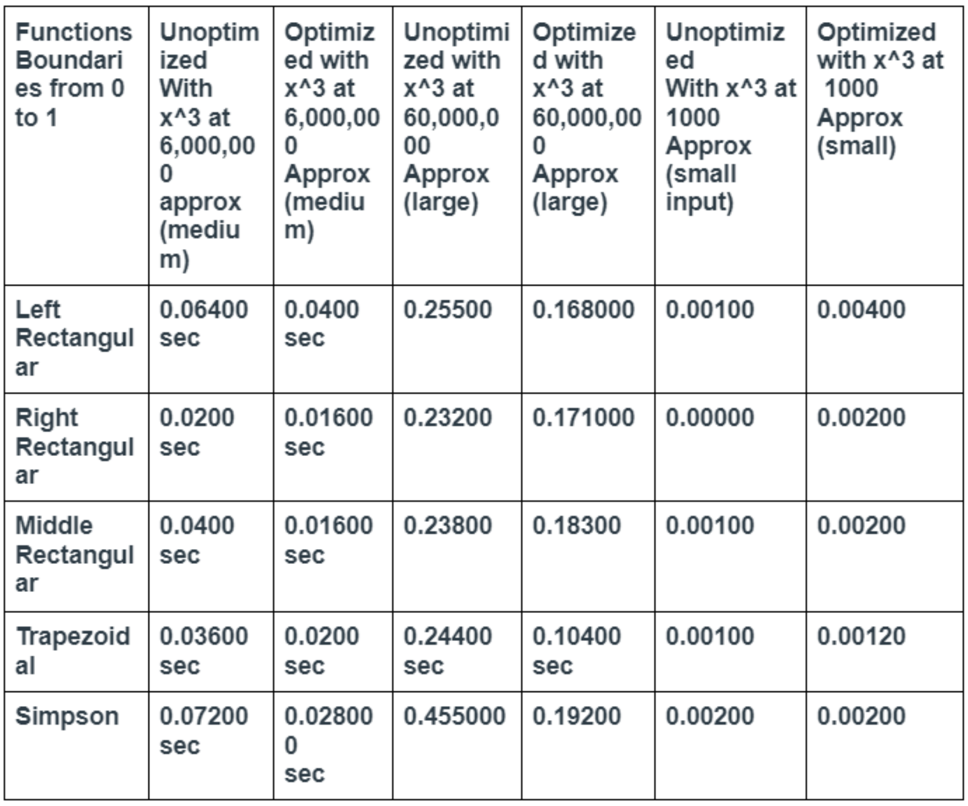
For the functions trapezoidal and simpson; the fact that both functions were utilizing integer numbers as iterators, It facilitated us the use of multi threading in both functions. To use this mentioned technique - in specific, for the trapezoidal function- the use of the reduction work-share construct was needed to have a local variable that would store a partial sum so that it can then be added to the total sum. Reduction allowed us to store a partial sum to a local thread that would then be combined to our actual output, so that the final total sum would not be affected by running the code in parallel, since whenever this happens the code runs in an unordered fashion leading to unexpected results which can be avoid by using said work share construct. Now for the simpson function we decided to utilize the same method, but this time, due to the code having two independent outer loops, we managed to run both loops in parallel but in different segments, using the same concept of reduction so that the total sum would not be affected by running the code in parallel.

For function left, right and mid rectangular we added a parallel region that works in two different loops. In the first loop we iterate until the last value of n and a partial sum which is given by a function (in our case x^3) will be running n times. The no wait work share construct allow us to run the threads without waiting for other threads to finish their job and by using the schedule(Static) we allow each thread to have its separate value of the partial sum which will then be needed for the second loop, where we are going to add all the partial sums to the total sum in an ordered fashion, that way we do not run with associativity rule issues that floating points may have whenever we are adding floating points in an unordered fashion.

An issue arises whenever we run the serial version of the functions where we iterate using double values. The issue consists: whenever we want to iterate using these floating point values , when this code gets compiled, the compiler converts this code to assembly and then to machine code. When this code gets converted to machine code so that the code can be run by the machine, the values that get converted to 0's and 1's are not precisely the same values that we thought we using to iterate, they are slightly different due to errors during conversion of floating point numbers to binary floating points, as these conversions cannot be represented correctly as binary floating points. This leads to the code to run less times then we expected.

We run the code using a laptop with a processor Intel(R) Core(™) i7-8750H CPU 2.20 ghz, with 8 threads. Since this code is being run in parallel using different threads, the performance will be reliant on the pc performance in general. The faster your pc is the better boost in performance the code will have.

The results are seen in the graph below.



Functions Boundaries from 0 to 1	Unoptimized With x^3 at 6,000,000 approx (medium)	Optimized with x^3 at 6,000,000 Approx (medium)	Unoptimized with x^3 at 60,000,000 Approx (large)	Optimized with x^3 at 60,000,000 Approx (large)	Unoptimized With x^3 at 1000 Approx (small input)	Optimized with x^3 at 1000 Approx (small)
Left Rectangular	0.06400 sec	0.0400 sec	0.25500	0.168000	0.00100	0.00400
Right Rectangular	0.0200 sec	0.01600 sec	0.23200	0.171000	0.00000	0.00200
Middle Rectangular	0.0400 sec	0.01600 sec	0.23800	0.18300	0.00100	0.00200
Trapezoidal	0.03600 sec	0.0200 sec	0.24400 sec	0.10400 sec	0.00100	0.00120
Simpson	0.07200 sec	0.028000 sec	0.455000	0.19200	0.00200	0.00200

Analyzing the above chart, we can clearly see how the optimized implementation outperformed the serial version of the code for medium and large inputs, as more threads are running the code in parallel without having to wait for a certain line of code to be read by the compiler, while with a small input there are not enough tasks to clearly see a boost in performance, hence the serial function is faster.

Unoptimized:

```
//Unoptimize left Integral

double int_leftrect(double from, double to, double n, double (*func)())
{
```

```c
    double h = (to-from)/n;
    double sum = 0.0, x;
    for(x=from; x <= (to-h); x += h)
          sum += func(x);
    return h*sum;
}

//Right Rectangular numerical Integration unoptimized function

double int_rightrect(double from, double to, double n, double (*func)())
{
   double h = (to-from)/n;
   double sum = 0.0, x;
   for(x=from; x <= (to-h); x += h){
     sum += func(x+h);
   }
   return h*sum;
}
//Double Mid Rectangular Numerical integration unoptimized function
double int_midrect(double from, double to, double n, double (*func)())
{
   double h = (to-from)/n;
   double sum = 0.0, x;
   for(x=from; x <= (to-h); x += h)
     sum += func(x+h/2.0);
   return h*sum;
}
//Trapezoidal Numerical Integration Unoptimized Function

double int_trapezium(double from, double to, double n, double (*func)())
{
   double h = (to - from) / n;
   double sum = func(from) + func(to);
   int i;
   for(i = 1;i < n;i++)
       sum += 2.0*func(from + i * h);
   return h * sum / 2.0;
}
//Simpson Numerical Integration unoptimized Function

double int_simpson(double from, double to, double n, double (*func)())
{
   double h = (to - from) / n;
   double sum1 = 0.0;
   double sum2 = 0.0;
   int i;
   double x;

   for(i = 0;i < n;i++)
      sum1 += func(from + h * i + h / 2.0);
```

```
  for(i = 1;i < n;i++)
    sum2 += func(from + h * i);

  return h / 6.0 * (func(from) + func(to) + 4.0 * sum1 + 2.0 * sum2);
}
```

Optimized:

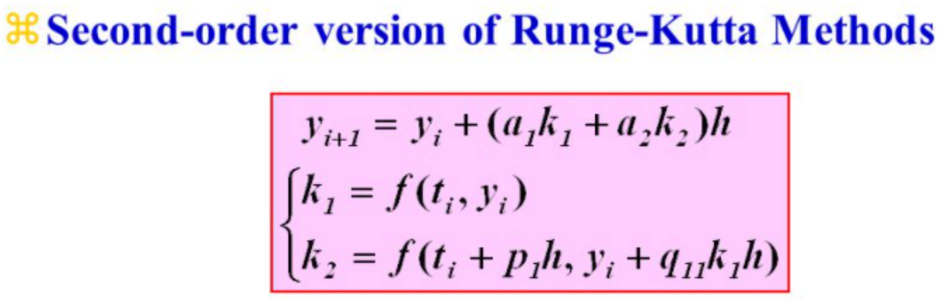
```
//Left Integral parallel function

double intlLeftRectParallel(double from, double to, double n, double (*func)())
{

    double h = to-from;
    double modeOffset = 0/2.0;
    double sum = 0.0;

    #pragma omp parallel
    {
      double psum;
      #pragma omp for schedule(static) nowait
        for(int i = 0; i< (int )n; i++){
          double x = from + h *(((double)(i)+modeOffset)/n);
          psum += func(x);
        }
      #pragma omp for schedule(static) ordered
        for(int i = 0 ; i< omp_get_num_threads() ;i++){
          #pragma omp ordered
            sum += psum;
        }
    }
    return sum*h/n;
}
//Right Rectangular numerical Integration parallel function

double intRightrectParallel(double from, double to, double n, double (*func)())
{
    double h = to-from;
    double modeOffset = 0/2.0;
    double sum = 0.0;
    #pragma omp parallel
    {
      double psum;
      #pragma omp for schedule(static) nowait
        for(int i = 0; i< (int )n; i++){
          double x = from + h *(((double)(i)+modeOffset)/n);
          psum += func(x+(h/n));
        }
      #pragma omp for schedule(static) ordered
        for(int i = 0 ; i< omp_get_num_threads() ;i++){
          #pragma omp ordered
          sum += psum;
        }

    }
    return sum*h/n;
}
//Mid Rectangular Numerical Integration Parallel Function

double intMidrectParallel(double from, double to, double n, double (*func)())
{


   double h = to-from;
   double modeOffset = 0/2.0;
    double sum = 0.0;
    #pragma omp parallel
    {
      double psum;
      #pragma omp for schedule(static) nowait
        for(int i = 0; i< (int )n; i++){
          double x = from + h *(((double)(i)+modeOffset)/n);
          psum += func(x+((h/n)/2.0));
        }
      #pragma omp for schedule(static) ordered
        for(int i = 0 ; i< omp_get_num_threads() ;i++){
          #pragma omp ordered
          sum += psum;
        }

    }
 return sum *(h/n);
}
//Trapezoidal Numerical Integration parallel Function
double intTrapeziumParallel(double from, double to, double n, double (*func)())
{
   double h = (to - from) / n;
   double sum = func(from) + func(to);
   int i;
   #pragma omp parallel for private(i) reduction(+:sum) shared(h,n)
      for(i = 1; i < ((int)n);i++){
        sum += 2.0*func(from + i * h);
      }
   return h * sum / 2.0;
}
//Simpson Numerical Integration Parallel Function
double intSimpsonParallel(double from, double to, double n, double (*func)())
{
   double h = (to - from) / n;
   double sum1 = 0.0;
   double sum2 = 0.0;
   int i;
   double x;
   #pragma omp parallel for private(i) reduction(+ : sum1) shared(from,h)
   for(i = 0;i < ((int)n);i++)
      sum1 += func(from + h * i + h / 2.0);

   #pragma omp parallel for private(i) reduction(+ : sum2) shared(from,h)
   for(i = 1;i < ((int)n);i++)
      sum2 += func(from + h * i);

   return h / 6.0 * (func(from) + func(to) + 4.0 * sum1 + 2.0 * sum2);
}
```

2 Runge-Kutta 2nd order method:

The Runge-Kutta 2nd order method is a technique used to solve differential equations that are in the form below:

$$\frac{dy}{dx} = f(x,y), y(0) = y_0$$

The equations for The Runge-Kutta 2nd order method is illustrated below:

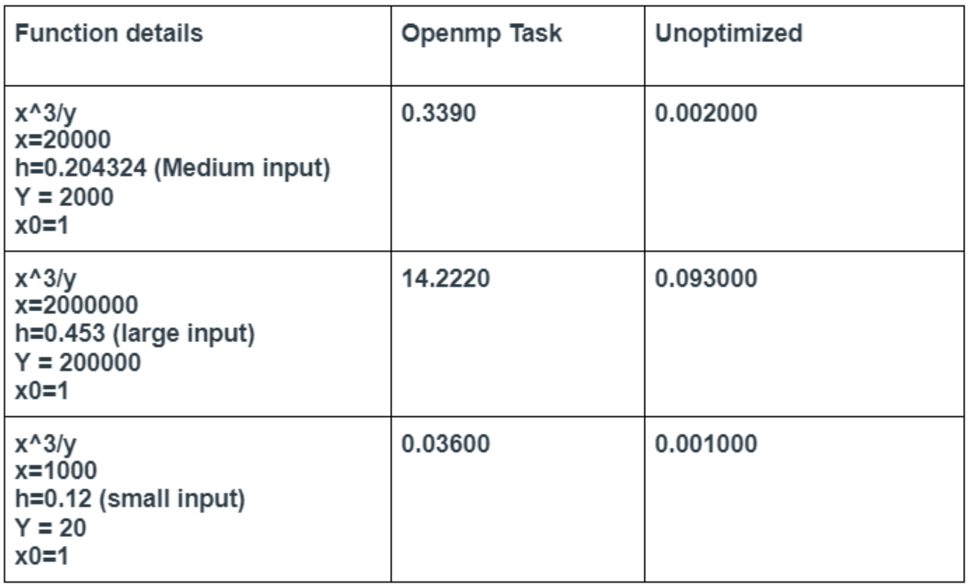


⌘ Second-order version of Runge-Kutta Methods

$$\boxed{\begin{aligned} &y_{i+1} = y_i + (a_1 k_1 + a_2 k_2)h \\ &\begin{cases} k_1 = f(t_i, y_i) \\ k_2 = f(t_i + p_1 h, y_i + q_{11} k_1 h) \end{cases} \end{aligned}}$$

It is to note that only first order differential equations can be solved by this particular method. Since Runge Kutta 2nd order function deals with too many dependencies on variables and values from outside a single set of brackets, it is not possible to use parallel programming, pragma openmp specifically, on this function. This is due to the fact in order to run the function in parallel each part must be completely independent of the others sections of the code so that it may run "side by side" instead of sequential. The function's different sections can be made independent of each other but that would require reworking the formula. With our teams current level of knowledge, we would not be able to recreate the formula to be able to run this in parallel.

We charted the runtime below. Keep in mind that the openmp task runtime is not fully run in parallel due to the explanation above, resulting in slower runtime than the unoptimized version.



Function details	Openmp Task	Unoptimized

x^3/y
x=20000
h=0.204324 (Medium input)
Y = 2000
x0=1
	0.3390	0.002000

x^3/y
x=2000000
h=0.453 (large input)
Y = 200000
x0=1
	14.2220	0.093000

x^3/y
x=1000
h=0.12 (small input)
Y = 20
x0=1
	0.03600	0.001000

You can see our attempt below at trying to run the method in parallel, marked by the comment OPTIMIZED.

Unoptimized and Optimized:

```
// C program to implement Runge
// Kutta method

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <omp.h>
// A sample differential equation
// "dy/dx = (x - y)/2"
double dydx(double x, double y)
{
    return (x + y -2);
}

// Finds value of y for a given x
// using step size h
// and initial value y0 at x0.
double rungeKutta(double x0, double y0,
                 double x, double h)//UNOPTIMIZED
{
    // Count number of iterations
    // using step size or
    // step height h
    int n = (int)((x - x0) / h);

    double k1, k2;

    // Iterate for number of iterations
    double y = y0;

    for (int i = 1; i <= n; i++) {
        // Apply Runge Kutta Formulas
        // to find next value of y
        k1 = h * dydx(x0, y);

        k2 = h * dydx(x0 + 0.5 * h,
                      y + 0.5 * k1);

        // Update next value of y
        y = y + (1.0 / 6.0) * (k1 + 2 * k2);

        // Update next value of x
        x0 = x0 + h;
    }

    return y;
}


 double rungeKuttaParallel(double x0, double y0,
                double x, double h)//OPTIMIZED
{
    // Count number of iterations
    // using step size or
    // step height h
    int n = (int)((x - x0) / h);

    double k1, k2;

    // Iterate for number of iterations
    double y = y0;
   #pragma omp parallel master
   {
    for (int i = 1; i <= n; i++) {
        // Apply Runge Kutta Formulas
        // to find next value of y

       #pragma omp task depend(out: k1)
        {
        k1 = h * dydx(x0, y);
        }

      #pragma omp task depend(in : k1)
       {
        k2 = h * dydx(x0 + 0.5 * h,
                      y + 0.5 * k1);
       }

       #pragma omp task depend(inout : k1,k2)
```

```
        // Update next value of y
       {
         y = y + (1.0 / 6.0) * (k1 + 2 * k2);
       }
        #pragma omp taskwait
        // Update next value of x
        x0 = x0 + h;
    }

   }

    return y;
}
// Driver Code
int main()
{
    double x0 = 20, y = 100,
          x = 20, h =0.2450;
    double timeSpentParallel, timeSpentSerial;
    clock_t begin = clock();
    printf("y(x) = %lf\n",
           rungeKutta(x0, y, x, h));
     clock_t end = clock();
        timeSpentSerial= (double)(end - begin) / CLOCKS_PER_SEC;

     clock_t begin1 = clock();
    printf("y(x) (Parallel) = %lf\n",
           rungeKuttaParallel(x0, y, x, h));
           clock_t end1 = clock();
        timeSpentParallel= (double)(end1 - begin1) / CLOCKS_PER_SEC;

        printf("TimeSpentParallel: %lf \nTimeSpentSequential: %lf", timeSpentParallel,
            timeSpentSerial);
    return 0;
}
```