Project 4

Kevin Siraki

April 20, 2021

1 Introduction

For this project, we had a similar problem to tackle as in Project 3: making functions run in parallel and measuring the overall run-time of both the original serial code and the parallel code. We began by running the original code with "gettimeofday()" in our main method in order to see the serial run-time. We then proceeded to convert the code to parallel in 3 different ways depending on the type of loop/iterator data type we were dealing with in each case. The results of this experimentation process will be further discussed with in depth code analysis in the following subsections.

2 Numerical Integration Serial:

	Serial:	Parallel:	
Small:	0.001001	0.004004	
Medium:	0.250227	0.041038	
Large:	9.115155	2.22297	
			X: Approach Avg. Run-time (Seconds)
			Y: Parameter Size

Numerical Integration featured 5 different techniques of integration. Each of these techniques were split into functions that took in the boundaries of integration, precision, and a function pointer as parameters. The important parameter we manipulated to test run-times was the precision parameter. As the above graph indicates, the serial method had an easy time with small-to-medium precision when calculating integrals, but suffers when we aim for a more precise calculation. These parameter sizes can be seen within the comments on lines 170-177. The corresponding run-times are displayed according to parameter size in the above graphical representation.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <unistd.h>
5  #include <float.h>
6  #include <sys/time.h>
7
8  double int_leftrect(double from, double to, double n, double (*func)())
9  {
10    double h = (to-from)/n;
11    double sum = 0.0, x;
12    for(x=from; x <= (to-h); x += h)
13       sum += func(x);
14    return h*sum;
15 }
```

```
16
17 double int_rightrect(double from, double to, double n, double (*func)())
18 {
19    double h = (to-from)/n;
20    double sum = 0.0, x;
21    for(x=from; x <= (to-h); x += h)
22      sum += func(x+h);
23    return h*sum;
24 }
25
26 double int_midrect(double from, double to, double n, double (*func)())
27 {
28    double h = (to-from)/n;
29    double sum = 0.0, x;
30    for(x=from; x <= (to-h); x += h)
31      sum += func(x+h/2.0);
32    return h*sum;
33 }
34
35 double int_trapezium(double from, double to, double n, double (*func)())
36 {
37    double h = (to - from) / n;
38    double sum = func(from) + func(to);
39    int i;
40    for(i = 1;i < n;i++)
41        sum += 2.0*func(from + i * h);
42    return  h * sum / 2.0;
43 }
44
45 double int_simpson(double from, double to, double n, double (*func)())
46 {
47    double h = (to - from) / n;
48    double sum1 = 0.0;
49    double sum2 = 0.0;
50    int i;
51
52    double x;
53
54    for(i = 0;i < n;i++)
55       sum1 += func(from + h * i + h / 2.0);
56
57    for(i = 1;i < n;i++)
58       sum2 += func(from + h * i);
59
60    return h / 6.0 * (func(from) + func(to) + 4.0 * sum1 + 2.0 * sum2);
61 }
62 /* test */
63 double f3(double x)
64 {
65   return x;
66 }
67
68 double f3a(double x)
69 {
```

```
70   return x*x/2.0;
71 }
72
73 double f2(double x)
74 {
75   return x*x;
76 }
77
78 double f2a(double x)
79 {
80   return x*x*x/3.0;
81 }
82
83 double f1(double x)
84 {
85   return x*x*x;
86 }
87
88 double f1a(double x)
89 {
90   return x*x*x*x/4.0;
91 }
92
93 typedef double (*pfunc)(double, double, double, double (*)());
94 typedef double (*rfunc)(double);
95
96 #define INTG(F,A,B) (F((B))-F((A)))
97
98 int main()
99 {
100         struct timeval start, end;
101
102         gettimeofday(&start, NULL);
103      int i, j;
104      double ic;
105
106      pfunc f[5] = {
107        int_leftrect, int_rightrect,
108        int_midrect,  int_trapezium,
109        int_simpson
110      };
111      const char *names[5] = {
112        "leftrect", "rightrect", "midrect",
113        "trapezium", "simpson"
114      };
115      rfunc rf[] = { f1, f2, f3, f3 };
116      rfunc If[] = { f1a, f2a, f3a, f3a };
117      double ivals[] = {
118        0.0, 1.0,
119        1.0, 100.0,
120        0.0, 5000.0,
121        0.0, 6000.0
122      };
123         // double approx[] = { 100.0, 100.0, 100.0, 100.0 }; //very small
```

```
        //double approx[] = { 1000.0, 1000.0, 5000000.0, 6000000.0 }; //medium
   double approx[] = {//large
        99999999,
        99999999,
        99999999,
        99999999
    };
     for(j=0; j < (sizeof(rf) / sizeof(rfunc)); j++)
     {
       for(i=0; i < 5 ; i++)
       {
         ic = (*f[i])(ivals[2*j], ivals[2*j+1], approx[j], rf[j]);
         printf("%10s [ 0,1] num: %+lf, an: %lf\n",
                names[i], ic, INTG((*If[j]), ivals[2*j], ivals[2*j+1]));
       }
       printf("\n");
     }
         gettimeofday(&end, NULL);
         double time_taken = end.tv_sec + end.tv_usec / 1e6 -
                        start.tv_sec - start.tv_usec / 1e6; // in seconds


        printf("Integration took %f seconds to execute in Serial\n", time_taken);
}
```

3 Numerical Integration Parallel:

	Serial:	Parallel:	
Small:	0.001001	0.004004	
Medium:	0.250227	0.041038	
Large:	9.115155	2.22297	
			X: Approach Avg. Run-time (Seconds)
			Y: Parameter Size

The additions made to the original code to allow this program to run properly in parallel were quite meticulously implemented. To begin with, we utilized fork-and-join with a parallel section in the first 3 integration methods due to the fact that a simple reduction would not suffice. This is due to the fact that the iterator in these functions is a double and the OMP directives within the compiler cannot deal with floating point numbers. Moreover, a simple cast of the iterator to an integer would not be possible because we would be losing enormous amounts of accuracy in our calculations. With this approach, we were able to "fork" the work onto multiple threads, allowing the loops themselves to run in "serial", then be "joined" in a critical section where their total sums were aggregated. The final two integration functions were quite simple to convert to parallel as they only featured simple for loops that had an integer iterator. We converted these to parallel by using a simple work-share construct and reduction. The benefits of the parallel approach were quite apparent as the problem size was increased, as can be seen in the graph above. For a large problem size, the runtime was about 1/4 that of the serial approach. The small and medium problem sizes were still within a margin of error to the serial approach, so this program seems to benefit greatly from being ran in parallel when a very large parameter size is introduced.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
```

```c
#include <float.h>
#include <unistd.h>
#include <sys/time.h>

double int_leftrect(double from, double to, double n, double( * func)()) {
    double h = (to - from) / n;
    double sum = 0.0, x;
    double tstart;
    int tid;
    int nthreads;
    double tend;
    double tsum = 0.0;
    #pragma omp parallel shared(from, to, n, nthreads,sum) private(x, tid, tsum, tstart, tend)
    {
        tid = omp_get_thread_num();
        nthreads = omp_get_num_threads();
        tstart = tid * (int) ceil(((double)(to - h) / nthreads));
        tend = (tid + 1) * (int) ceil(((double)(to - h) / nthreads));
        if (tend <= (to - h))
                        for (x = tstart; x <= tend; x += h)
                tsum += func(x);
        else
            for (x = tstart; x <= (to - h); x += h)
                tsum += func(x);
        #pragma omp critical
            sum += tsum;
    }
    return h * sum;
}

double int_rightrect(double from, double to, double n, double( * func)()) {
    double h = (to - from) / n;
    double sum = 0.0, x;
    double tstart;
    int tid;
    double tend;
    int nthreads;
    double tsum = 0.0;
    #pragma omp parallel shared(from, to, n, nthreads,sum) private(x, tid, tsum, tstart, tend)
    {
        tid = omp_get_thread_num();
        nthreads = omp_get_num_threads();
        tstart = tid * (int) ceil(((double)(to - h) / nthreads));
        tend = (tid + 1) * (int) ceil(((double)(to - h) / nthreads));
        if (tend < (to - h))
            for (x = tstart; x <= tend; x += h)
                tsum += func(x + h);
        else
            for (x = tstart; x <= to - h; x += h)
                tsum += func(x + h);
        #pragma omp critical
            sum += tsum;
    }
    return (h * sum);
```

```
59 }
60
61 double int_midrect(double from, double to, double n, double( * func)()) {
62     double h = (to - from) / n;
63     double sum = 0.0, x;
64     double tstart;
65     int tid;
66     double tend;
67     int nthreads;
68     double tsum = 0.0;
69     #pragma omp parallel shared(from, to, n, sum, nthreads) private(x, tid, tsum, tstart, tend)
70     {
71         tid = omp_get_thread_num();
72         nthreads = omp_get_num_threads();
73         tstart = tid * (int) ceil(((double)(to - h) / nthreads));
74         tend = (tid + 1) * (int) ceil(((double)(to - h) / nthreads));
75         if (tend < (to - h))
76             for (x = tstart; x <= tend; x += h)
77                 tsum += func(x + h / 2.0);
78         else
79             for (x = tstart; x <= to - h; x += h)
80                 tsum += func(x + h / 2.0);
81         #pragma omp critical
82             sum += tsum;
83     }
84     return h * sum;
85 }
86
87 double int_trapezium(double from, double to, double n, double( * func)()) {
88     double h = (to - from) / n;
89     double sum = func(from) + func(to);
90     int i;
91     #pragma omp parallel
92     {
93         #pragma omp for reduction(+: sum)
94             for (i = 1; i < (int) n; i++) {
95                 sum += 2.0 * func(from + i * h);
96             }
97     }
98     return h * sum / 2;
99 }
100
101 double int_simpson(double from, double to, double n, double( * func)()) {
102     double h = (to - from) / n;
103     double sum1 = 0.0;
104     double sum2 = 0.0;
105     int i;
106     #pragma omp parallel
107     {
108         #pragma omp for reduction(+: sum1)
109             for (i = 0; i < (int) n; i++) {
110                 sum1 += func(from + h * i + h / 2);
111             }
112         #pragma omp for reduction(+: sum2)
```

```
            for (i = 1; i < (int) n; i++) {
                sum2 += func(from + h * i);
            }
    }
    return h / 6 * (func(from) + func(to) + 4 * sum1 + 2 * sum2);
}

/* test */
double f3(double x) {
    return x;
}

double f3a(double x) {
    return x * x / 2.0;
}

double f2(double x) {
    return x * x ;
}

double f2a(double x) {
    return x * x * x / 3.0;
}

double f1(double x) {
    return x * x * x;
}

double f1a(double x) {
    return x * x * x * x / 4.0;
}

typedef double( * pfunc)(double, double, double, double( * )());
typedef double( * rfunc)(double);

#define INTG(F, A, B)(F((B)) - F((A)))

int main() {
    struct timeval start, end;

    gettimeofday( & start, NULL);
    int i, j;
    double ic;

    pfunc f[5] = {
        int_leftrect,
        int_rightrect,
        int_midrect,
        int_trapezium,
        int_simpson
    };
    const char * names[5] = {
        "leftrect",
        "rightrect",
```

```
        "midrect",
        "trapezium",
        "simpson"
    };
    rfunc rf[] = {
        f1,
        f2,
        f3,
        f3
    };
    rfunc If[] = {
        f1a,
        f2a,
        f3a,
        f3a
    };
    double ivals[] = {
        0.0,
        1.0,
        1.0,
        100.0,
        0.0,
        5000.0,
        0.0,
        6000.0
    };

        // double approx[] = { 100.0, 100.0, 100.0, 100.0 }; //very small
         //double approx[] = { 1000.0, 1000.0, 5000000.0, 6000000.0 }; //medium
   double approx[] = {//large
        99999999,
        99999999,
        99999999,
        99999999
    };
    for (j = 0; j < (sizeof(rf) / sizeof(rfunc)); j++) {
        for (i = 0; i < 5; i++) {
            ic = ( * f[i])(ivals[2 * j], ivals[2 * j + 1], approx[j], rf[j]);
            printf("%10s [ 0,1] num: %+lf, an: %lf\n",
                names[i], ic, INTG(( * If[j]), ivals[2 * j], ivals[2 * j + 1]));
        }
        printf("\n");
    }
    gettimeofday( & end, NULL);
    double time_taken = end.tv_sec + end.tv_usec / 1e6 -
        start.tv_sec - start.tv_usec / 1e6; // in seconds

    printf("Integration took %f seconds to execute in Parallel\n", time_taken);
}
```

4 Runge-Kutta Serial:

	Serial:	Parallel:	
Small:	0.001001	0.001001	
Medium:	0.005003	0.005005	
Large:	4.93593	4.14735	
			X: Approach Avg. Run-time (Seconds)
			Y: Parameter Size

According to Wikipedia, Runge-Kutta is a "In numerical analysis, the Runge–Kutta methods are a family of implicit and explicit iterative methods, which include the well-known routine called the Euler Method, used in temporal discretization for the approximate solutions of ordinary differential equations. " As we can see, the serial method for this function seems to run quite quickly for anything from small to large problem sizes. The size of the problems can be seen in lines 43-47. The parameter size on line 43 was quite small and resulted in too tiny of a run-time to be relevant in our observations. The corresponding run-times based on parameter sizes is present in the chart on top.

```c
// C program to implement Runge
// Kutta method
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
// A sample differential equation
// "dy/dx = (x - y)/2"

float dydx(float x, float y)
{
    return (x + y - 2);
}

// Finds value of y for a given x
// using step size h
// and initial value y0 at x0.
float rungeKutta(float x0, float y0, float x, float h)
{
    // Count number of iterations
    // using step size or
    // step height h
    int n = (int)((x - x0) / h),i;
    float k1, k2;
    // Iterate for number of iterations
    float y = y0;
    for (i = 1; i <= n; i++)
    {
        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;
}

```

```
// Driver Code
int main()
{
    struct timeval start, end;
    gettimeofday(&start, NULL);
    //float x0 = 0, y = 1, x = 2, h = 0.002; //too small

    //float x0 = 0, y = 10, x = 20, h = 0.001; //small
    //float x0 = 0, y = 10, x = 20, h = 0.0001; //medium
    float x0 = 0, y = 10, x = 20, h = 0.0000001; //large
    printf("y(x) = %f", rungeKutta(x0, y, x, h));
    gettimeofday(&end, NULL);
    double time_taken = end.tv_sec + end.tv_usec / 1e6 - start.tv_sec - start.tv_usec / 1e6; // in seco
    printf("\nRunge-Kutta took %f seconds to execute in Serial\n", time_taken);
    return 0;
}
```

5 Runge-Kutta Parallel:

	Serial:	Parallel:	
Small:	0.001001	0.001001	
Medium:	0.005003	0.005005	
Large:	4.93593	4.14735	
			X: Approach Avg. Run-time (Seconds)
			Y: Parameter Size

In our run-time analysis of the parallel version of Runge-Kutta, the performance was equivalent to or slightly worse with a smaller parameter size in comparison to the serial version. However, as excepted, a larger parameter size for the variable 'h' showed us the true benefits of running this code in parallel. The parallel version of this code was devised by utilizing tasks due to the nature of the for loop in the Runge-Kutta method. A simple reduction would not work here because parallelization of the for loop would result in an incorrect answer when a measly work-share construct was implemented.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
#include <omp.h>

float dydx(float x, float y)
{
    return (x + y - 2);
}

float rungeKutta(float x0, float y0, float x, float h)
{
    int n = (int)((x - x0) / h), i;
    float k1, k2;
    float y = y0;
    #pragma omp parallel reduction(+:x0)
    {
        #pragma omp master
        {
            for (i = 1; i <= n; i++)
```

```
            {
                #pragma omp task shared(x0,h,y) private(i,k1,k2)
                {
                    k1 = h * dydx(x0, y);
                    k2 = h * dydx(x0 + 0.5 * h, y + 0.5 * k1);
                    x0 = x0 + h; //x0 += h;
                    y = y + (1.0 / 6.0) * (k1 + 2 * k2);
                }
            }
        }
        #pragma omp taskwait
    }
    return y;
}

int main()
{
    struct timeval start, end;
    gettimeofday( & start, NULL);
    //float x0 = 0, y = 1, x = 2, h = 0.002; //too small

    //float x0 = 0, y = 10, x = 20, h = 0.001; //small
    //float x0 = 0, y = 10, x = 20, h = 0.0001; //medium
    float x0 = 0, y = 10, x = 20, h = 0.0000001; //large
    printf("y(x) = %f", rungeKutta(x0, y, x, h));
    gettimeofday( & end, NULL);
    double time_taken = end.tv_sec + end.tv_usec / 1e6 - start.tv_sec - start.tv_usec / 1e6; // in seco
    printf("\nRunge-Kutta took %f seconds to execute in Parallel\n", time_taken);
    return 0;
}
```

---

6 Conclusion:

To conclude, in conducting our analysis on these Numerical Integration and Runge-Kutta in parallel, we have come to the realization that both of these programs seem to benefit quite greatly from being run in parallel when a larger parameter size is introduced. Moreover, even on a small problem size, the parallel versions of these programs ran at nearly the same speed or within the margin of error of the serial versions. Overall, we utilized fork/join, work-share constructs/reductions, and tasks in Open-MP to run these programs in parallel and the behavior of the parallel code seemed superior to the serial, especially in the case of larger problems being solved.