Laboratory No. 03:
HPC Lab Report

By: Sam Partee

Instructor: John Dougherty

Haverford High Performance Computing 287

March 8, 2017

I. OVERVIEW

In this laboratory experiment we once again use OpenMP to make a sequential program multi-threaded and explore the difference between sequential and parallel execution of a computationally intensive task, Matrix Multiplication. The program, coded in C, takes in a text file of two matrices and multiplies them together to form a single matrix. Matrix Multiplication, used in many fields, is an extremely applicable experiment for the purpose of High Performance Computing.

A. Purpose

The purpose of this laboratory experiment was to examine the difference in complexity between a sequential matrix multiplication program and a parallel one. Data is to be gathered that shows both the speedup and efficiency of added processors.

B. Equipment

1. Linux Ubuntu 16.04 64 bit Machine
2. Intel Core i7-5500U CPU @ 2.40GHz(4 cores)
3. Open Mp and C programming language.

C. Procedure

The procedure of the computational experiment went as follows.

1. A sequential single-threaded matrix multiplication program was programmed and run.
2. Data was gathered on sequential runs using 1 core and 1 thread for computation.
3. Using OpenMP, the program was modified to run in parallel
4. Data was gathered running the matrix multiplication with up to 4 cores and threads.
5. Graphs and experimental analysis was conducted on complexity performance.

D. Estimated Results

The hypothesis of the experiment is that as the number of the processors is increased, the time complexity of the overall program will decrease. This is expected as 4 threads running concurrently on multiple cores should run faster than 4 threads running sequentially on one core. Matrix Multiplication, in itself, is a taxing program for a CPU. In this program, we expect up to a matrix of 10000 by 10000 to be able to successfully run a processor with 4 cores with a time complexity that remains reasonable.(within an upper bound of three times the complexity of a 1000 by 1000 matrix)

E. Actual Results

The results of the experiment showed that an increase in processor count does combat time complexity as input grows. An interesting observation, however, was the difference in results between a 100 x 100 matrix and a 1000 x 1000 matrix. When N=100 there was little difference between the complexity of the sequential program and the program running on 2, 3, and 4 cores of the CPU. In contrast, when N was increased to 1000 the difference in time complexity was strikingly apparent in that the sequential program was significantly slower than the multi-threaded versions. The difference between complexity of 2, 3, and 4 cores was less than expected but still was consistent with the given hypothesis. The results of the computations when N = 10000 were heavily skewed as they were only able to run on different hardware and are therefore not included in the experiment. The graphs below show the difference in complexity varying with input and processor count as discussed above.

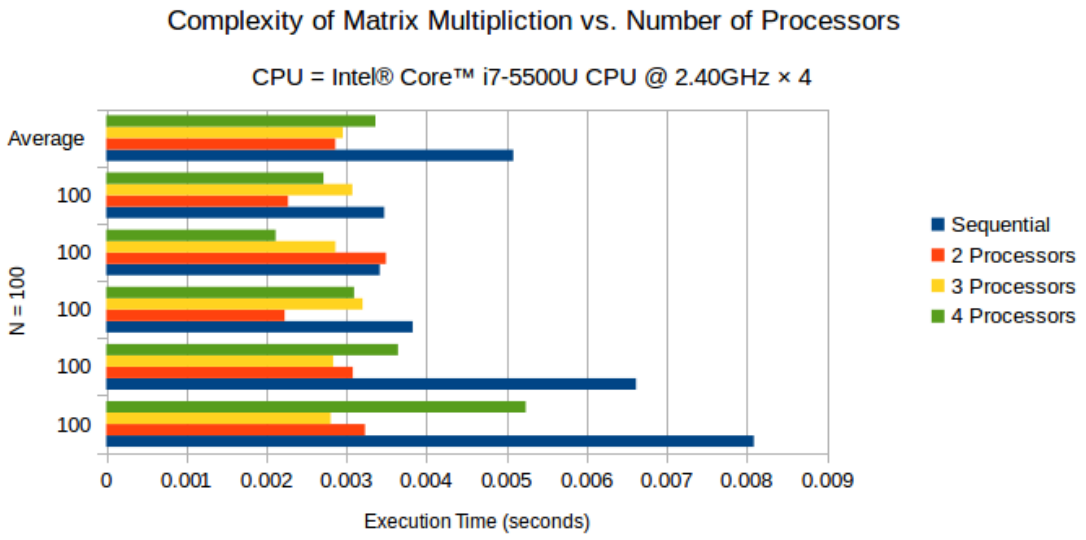

FIG. 1: Showing data collection from 100 by 100 Matrix Multiplication

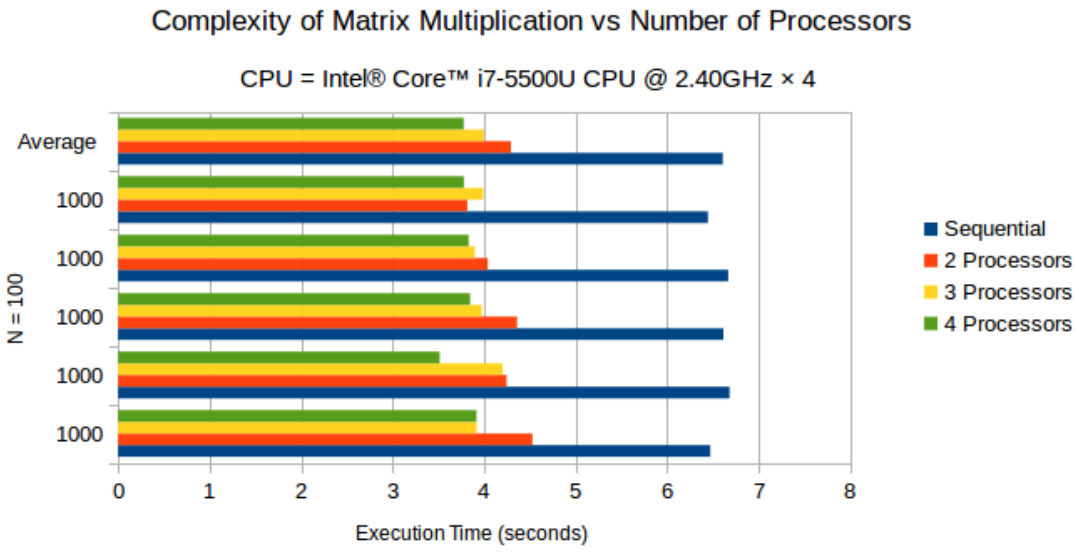

FIG. 2: Showing data collection from 1000 by 1000 Matrix Multiplication

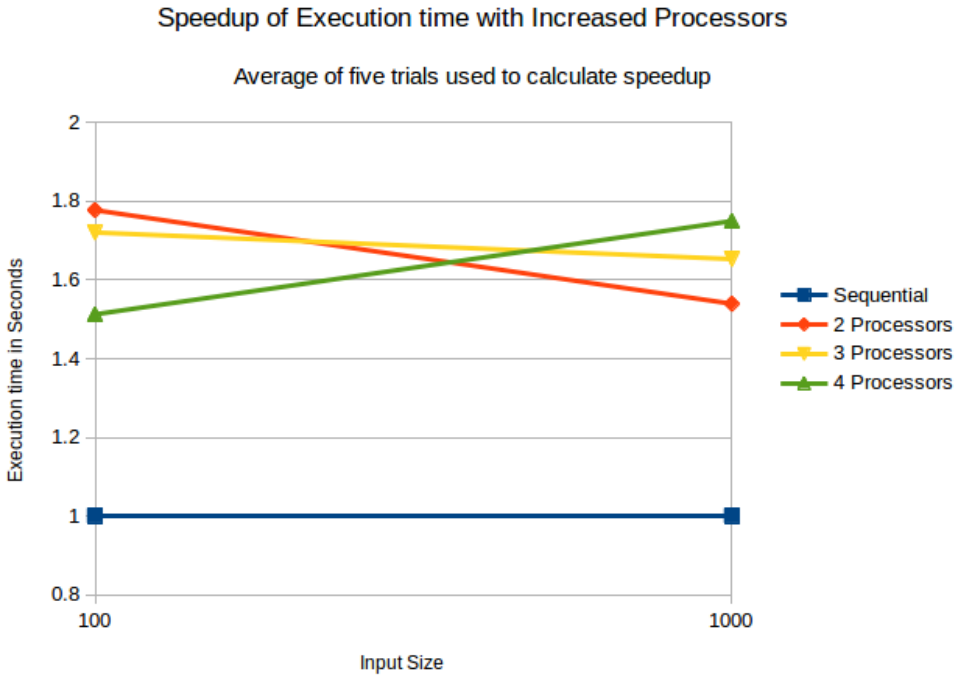

FIG. 3: Speedup of sequential program with increased processor count

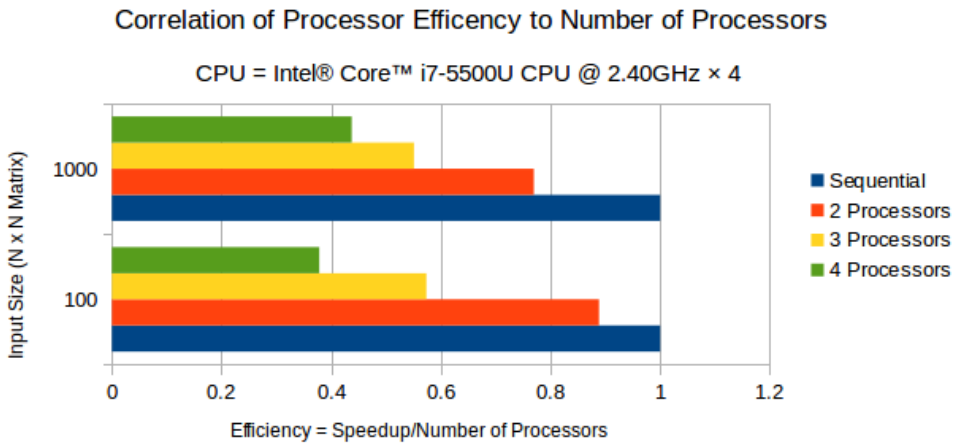

FIG. 4: A graph showing efficiency of added processors.

II. DISCUSSION & CONCLUSION

A few interesting observations came from this experiment. First, when input size was a 100 x 100 matrix or smaller the processor count had little impact on the processing speed, and actually computations running on 2 processors has the fastest average execution time. However, as input size grew the speedup of the program increased with added processors as expected and stated in the hypothesis. Efficiency was also seen to increase in multi-threaded programs as the size of the matrix grew.

In comparison to the dart simulation, one can tell that matrix multiplication carries a much greater time complexity as when input increased to 10000 the program ran exponentially slower. This is due to the nested loops in matrix multiplication.

If I were to do this experiment again, I think the most important part would be to increase the number of processors available in my hardware as being limited to 4 processors does not completely show the impact of added processing power as much as I would like it to.

In conclusion, The most important takeaway from this experiment is that the complexity of a program in terms of its software is far more important the hardware that it runs on, however, one can run more computationally intensive experiments if given more processing power. Therefore, if decreasing time complexity in software is not an option, such as the nested loops in matrix multiplication, adding processors can speedup the execution times to an acceptable level. Once again, concurrency is key.

III. APPENDIX: PROGRAM CODE

```
1
2
3  // Sequential verison of Matric Multiplication program. Runs on a single thread
   // Prints to a text file.
5
   #include <math.h>
7  #include <string.h>
   #include <stdlib.h>
9  #include <time.h>
   #include <stdio.h>
11 #include "cur_time.h"

13 int main(void) {

15   double matrixMultiply(int x, int N) {


17
       FILE *fp;
19     if(x==2) {
         fp = fopen("Matrix2.txt","r");
21     }
       else if(x==3) {
23       fp = fopen("Matrix3.txt","r");
       }
25     else {
         fp = fopen("Matrix4.txt","r");
27       }
       int i,j,k; //variables for matrix for loops
29


31     // Memory Allocation
       double **a, **b; // a and b are pointers
33     a = (double**)malloc(N*sizeof(*a));
       b = (double**)malloc(N*sizeof(*b));
35
       // Memory Allocation
37     for (i=0; i<N; i++) {
         a[i] = malloc(N*sizeof(*a[i]));
39       b[i] = malloc(N*sizeof(*b[i]));
       }
41
       // Calcualate read_time
43     double start_read_time = cur_time();
        for (i=0; i<N; i++) {
45       for (j=0; j<N; j++) {
           fscanf(fp, "%lf", &a[i][j]);//read file element by element
47       }
       }
49     double read_time = 1.0 * cur_time() - start_read_time;
       printf("File read \n");
51     printf("Time taken for reading file: %f\n",read_time);
       fclose(fp);
53


55     // actual matrix Multiplication calculation
       double start_time = cur_time();
```

```
57      for (i=0; i<N; i++) {
          for (j=0; j<N; j++) {
59          double sum = 0;
            for (k=0; k<N; k++) {
61            sum += a[i][k] * a[k][j];
              }
63          b[i][j] = sum;
            }
65        }


67
      // free up memory space for re-allocation
69    for (i=0; i<N; i++){
        free(a[i]);
71      free(b[i]);
      }
73    free(a);
      free(b);
75
      // calculates multiply time
77    double multiply_time = 1.0 * cur_time() - start_time;
      printf("Execution time in seconds: %f\n",multiply_time);//tell user what the
          execution time was
79


81    // prints the results to a text file for data collection
      FILE *times;
83    times = fopen("sequentialtimes.txt","a");
      fprintf(times, "%d%s%f%s%f%s",N,"\t",read_time, "\t",multiply_time,"\n");
85    fclose(times);//returns 0;

87  }

89  remove("sequentialtimes.txt");
    FILE *times;
91  times = fopen("sequentialtimes.txt","w");
    fprintf(times,"N\tRead Time(seconds)\tMultiply Time(seconds)\n");
93  fclose(times);

95  int x, y; // helper variables for data collection
    for(x=2; x<=3; x++) {
97
      int N = (int)pow((double)10,(double)x); // n = 100, 1000, 10000
99    printf("N = %d\n",N);      // prints N

101   for (y=1; y<=5; y++) {    // prints 5 trials
        matrixMultiply(x,N);
103   }
    }
105 return 0;

107 }
```

```
1
// Program for Parallel Matrix Multiplication. Uses OpenMp to access multple threads
    .
3 // Program loops through input size and processor count for data collection purposes
    .
```

```
// Prints to a txt file.


#include <math.h>
#include <omp.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <stdio.h>
#include "cur_time.h"



int main(void) {

  double matrixMultiply(int x, int N, int nthreads) {

    FILE *fp;

    if(x==2) {
      fp = fopen("Matrix2.txt","r");
    }
    else if(x==3) {
      fp = fopen("Matrix3.txt","r");
    }
    else {
      fp = fopen("Matrix4.txt","r");
      }

    int i,j,k;
    double sum;             // init helper variables for matrix calculation
    double **a, **b;


    // Memory Allocation
    a = (double**)malloc(N*sizeof(*a));
    b = (double**)malloc(N*sizeof(*b));

    // Memory Allocation
    for (i=0; i<N; i++) {
      a[i] = malloc(N*sizeof(*a[i]));
      b[i] = malloc(N*sizeof(*b[i]));
    }

    // read in the file
    double read_start = cur_time(); // varaible linked to helper program
    for (i=0; i<N; i++) {           // for time calculation
      for (j=0; j<N; j++) {
        fscanf(fp, "%lf", &a[i][j]);
      }
    }

    // calculate read time and print to file
    double read_time = 1.0 * cur_time() - read_start;
    printf("File read \n");
    printf("Time taken for reading file: %f\n",read_time);
    fclose(fp);

```

```
63      // variable for calucation of time complexity
        double start_multiply_time = cur_time();
65
        // set omp variables
67      omp_set_dynamic(0); //disable dynamic allocation
        omp_set_num_threads(nthreads); //use the specified number of threads
69
    // omp parallel section starts
71  #pragma omp parallel private(i,j,k,sum)
        {
73  #pragma omp for
        for (i=0; i<N; i++) {
75        for (j=0; j<N; j++) {
            sum = 0;
77          for (k=0; k<N; k++) {
              sum += a[i][k] * a[k][j];
79            }
            b[i][j] = sum;
81        }
        }
83      }

85      // free up memory space for re-allocation.
        for (i=0; i<N; i++){
87        free(a[i]);
          free(b[i]);
89      }
        free(a);
91      free(b);

93      //variable for complexity calculations.
        double multiplication_complexity = 1.0 * cur_time() - start_multiply_time;
95      printf("Execution time (sec): %f\n",multiplication_complexity);
        FILE *times;
97      times = fopen("paralleltimes.txt","a");
        fprintf(times, "%d%s%f%s%f%s",N,"\t", read_time, "\t",multiplication_complexity,
            "\n");
99      fclose(times);//returns 0;

101   }

103   // remove old file
      remove("paralleltimes.txt");
105   FILE *times;

107   //open new file and print results.
      times = fopen("paralleltimes.txt","w");
109   fprintf(times,"N\tRead Time(seconds)\tMultiply Time(seconds)\n");
      fclose(times);
111

113   int x, y, p; // helper variables for data collection
      for(x=2; x<=3; x++) {
115     int N = (int)pow((double)10,(double)x); // n = 100, 1000, 10000

117     // loop through number of threads
        for (p=2; p<=4; p++) {
119
          // print five trials of results of the program.
```

```
      for (y=1; y<=5; y++) {
        matrixMultiply(x,N, p);
      }
      printf("N = %d\n",N);                    // Print N
      printf("num_threads = %d\n",p);          // print Number of threads.
    }
  }
  return 0;
}
```

```
CC = /usr/bin/gcc
CFLAGS = -g -O0 -fopenmp
LD = /usr/bin/gcc
LDFLAGS = -g -fopenmp

PROGRAM1 = MatrixMultiplication
PROGRAM2 = MatrixMultiplication_Parallel

all:    ${PROGRAM1}.exe ${PROGRAM2}.exe

${PROGRAM1}.exe: ${PROGRAM1}.o cur_time.o
      ${LD} ${PROGRAM1}.o cur_time.o -o ${PROGRAM1}.exe -lm

${PROGRAM1}.o: ${PROGRAM1}.c
      ${CC} ${CFLAGS} -c $<

${PROGRAM2}.exe: ${PROGRAM2}.o cur_time.o
      ${LD} ${LDFLAGS} ${PROGRAM2}.o cur_time.o -o ${PROGRAM2}.exe -lm

${PROGRAM2}.o: ${PROGRAM2}.c
      ${CC} ${CFLAGS} -c $< -o ${PROGRAM2}.o

cur_time.o: cur_time.c
      ${CC} ${CFLAGS} -c $<

clean:
      rm -f ${PROGRAM1}.o ${PROGRAM1}.exe ${PROGRAM2}.o ${PROGRAM2}.exe
```

```
#include "cur_time.h"
#include <sys/time.h>


double cur_time(void)  /* From Dave or one of Dave's students */
{
        struct timeval tv;
        /* struct timezone tz; */
        gettimeofday(&tv, 0 /* &tz */);
        return tv.tv_sec + tv.tv_usec*1.0e-6;
}
```