Evaluation of the Performance of Parallel Programming using OpenMP

Duwal Shrestha, Mahendra
mduwalsh@utk.edu

Nash, Drew
anash4@utk.edu

30 April 2015

Introduction

Parallel algorithms, which execute by running multiple threads that perform simultaneous operations, can dramatically decrease the amount of time necessary to calculate the result of a given problem. The effectiveness of threads, utilizing OpenMP, is examined by comparing the parallel and serial runtimes of matrix addition, matrix multiplication, and the *three-body problem*.

Matrix Addition and Multiplication

Serial Implementation

Serial Matrix Addition

The following pseudocode is an algorithm that adds the square matrices $M1$ and $M2$, each with n rows and n columns, and stores the sum in *result*:

```
for i = 1 to n
     for j = 1 to n
          result[i][j] = M1[i][j] + M2[i][j]
return result
```

This straightforward algorithm always performs n^2 operations. Therefore, the time complexity is: $\Theta\left(n^2\right)$.

The C implementation of this algorithm is in the `serial_add()` function of `matrix-serial.c`, which is in the Appendix.

Serial Matrix Multiplication

The following pseudocode is an algorithm that multiplies the square matrices $M1$ and $M2$, each with n rows and n columns, and stores the product in *result*:

```
for i = 1 to n
     for j = 1 to n
          result[i][j] = 0
          for k = 1 to n
```

```
                result[i][j] = result[i][j] + M1[i][k] * M2[k][j]
return result
```

With the three nested loops, the time complexity of this algorithm is: $\Theta\left(n^3\right)$.

The C implementation of this algorithm is in the `serial_multiply()` function of `matrix-serial.c`, which is in the Appendix.

Parallel Implementation

Parallel Matrix Addition

The following pseudocode is a parallel algorithm that adds the square matrices $M1$ and $M2$, each with n rows and n columns, and stores the sum in *result*:

```
parallel for i = 1 to n
     parallel for j = 1 to n
          result[i][j] = M1[i][j] + M2[i][j]
return result
```

The *work* of this algorithm, which is the same as the complexity of the serial version, is $\Theta\left(n^2\right)$. By utilizing two parallel loops, the *span*, which is the time to execute the commands within the loops in parallel, is $\Theta\left(\lg n\right) + \Theta\left(\lg n\right) = \Theta\left(\lg n\right)$. Therefore, the *parallelism* of this algorithm, defined as the work divided by the span, is:

$$\Theta\left(n^2\right) / \Theta\left(\lg n\right) = \Theta\left(\frac{n^2}{\lg n}\right)$$

This algorithm is implemented in the C language, utilizing OpenMP, in the `parallel_add()` function of `matrix-parallel.c`, which is in the Appendix.

Parallel Matrix Multiplication

The following pseudocode is a parallel algorithm that multiplies the square matrices $M1$ and $M2$, each with n rows and n columns, and stores the product in *result*:

```
parallel for i = 1 to n
     parallel for j = 1 to n
          result[i][j] = 0
          for k = 1 to n
               result[i][j] = result[i][j] + M1[i][k] * M2[k][j]
return result
```

The work is $\Theta\left(n^3\right)$. By utilizing two parallel loops and one regular loop, the span is $\Theta\left(\lg n\right) + \Theta\left(\lg n\right) + \Theta\left(n\right) = \Theta\left(n\right)$. Therefore, the parallelism of this algorithm is:

$$\Theta\left(n^3\right) / \Theta\left(n\right) = \Theta\left(n^2\right)$$

This algorithm is implemented in the C language, utilizing OpenMP, in the `parallel_multiply()` function of `matrix-parallel.c`, which is in the Appendix.

Note that the third-most interior loop in this algorithm is not run in parallel. This is to prevent conflicting variable accesses by different threads. If two or more threads attempt to update `result[i][j]` simultaneously, the algorithm could produce inaccurate results and become unreliable. There are algorithms, such as Strassen's Method for matrix multiplication, that attempt to optimize the multiplication as much as possible without leading to race conditions. However, those algorithms were not utilized since the purpose of this research is to demonstrate the efficiency of OpenMP alone. Furthermore, while it is possible to utilize the OpenMP *lock*, which is a form of a mutex, to allow for a third parallel loop, the locking and unlocking steps lead to the actual runtimes being worse. Therefore, this parallel algorithm was chosen to allow for an examination of the direct impact of OpenMP.

Three-Body Problem

Serial Simulation

The following pseudocode is an algorithm that simulates the three-body, or more generally, n-body problem, for a step of a given simulation period dt, where bd is a pointer array of the Body `struct`, containing properties mass (m), positions (x, y), velocities (vx, vy) and accelerations (ax, ay):

```
for(i = 0; i < n; i++){
       bd[i].ax = 0; bd[i].ay = 0;
       // calculate acceleration due to forces
       for(j = 0; j < n; j++){
            if(i == j) continue;    // no sense in calculating force from itself
            // calculate acceleration due to other body
            f = sqrt ( pow( square( b[i].x - b[j].x) + square(b[i].y - b[j].y), 3 ) );
            // cube of distance between two bodies
            bd[i].ax = bd[i].ax - b[j].m * ( b[i].x - b[j].x ) / f;
            bd[i].ay = bd[i].ay - b[j].m * ( b[i].y - b[j].y ) / f;
        }
        // update position of body      s = ut + 1/2*a*t^2
        bd[i].x += (bd[i].vx + 0.5*bd[i].ax*dt)*dt;
        bd[i].y += (bd[i].vy + 0.5*bd[i].ay*dt)*dt;
        // update velocity of a body    v = u + at;
        bd[i].vx += bd[i].ax * dt;
        bd[i].vy += bd[i].ay * dt;
}
```

Parallel Simulation

The following pseudocode is a parallel algorithm that simulates the three-body, or more generally, n-body problem, for a step of a given simulation period dt, where bd is a pointer array of the Body `struct`, containing properties mass (m), positions (x, y), velocities (vx, vy) and accelerations (ax, ay):

```
      parallel for(i = 0; i < n; i++){
          b[i].ax = 0; b[i].ay = 0;
          // calculate acceleration due to forces
          for(j = 0; j < n; j++){
```

```
            if(i == j) continue;    // no sense in calculating force from itself
            // calculate acceleration due to other body
            f[i] = sqrt ( pow( square( bd[i].x - bd[j].x) + square(bd[i].y - bd[j].y), 3 ) );
            // cube of distance between two bodies
            b[i].ax += - bd[j].m * ( bd[i].x - bd[j].x ) / f[i];
            b[i].ay += - bd[j].m * ( bd[i].y - bd[j].y ) / f[i];
        }
        // update position of body      s = ut + 1/2*a*t^2
        b[i].x += (b[i].vx + 0.5*B[i].ax*dt)*dt;
        b[i].y += (b[i].vy + 0.5*B[i].ay*dt)*dt;
        // update velocity of a body    v = u + at;
        b[i].vx += b[i].ax * dt;
        b[i].vy += b[i].ay * dt;
    }
```

The outer loop for multiple numbers of simulation steps could not be parallelized because properties of the body, updated in a step, depend on properties in the preceding step. But properties of a body can be updated independently of other bodies in a step of the simulation if the state of all other bodies in the previous step is accessible.

Testing

Matrix Addition and Multiplication

Compiling and Usage

To compile the `matrix-serial.c` and `matrix-parallel.c` programs to binary executable files respectively named `matrix-serial` and `matrix-parallel`, run the following commands:

```
UNIX> gcc matrix-serial.c -o matrix-serial
UNIX> gcc -fopenmp matrix-parallel.c -o matrix-parallel
```

To execute the binary files, run the commands with the following arguments, all of which must be integers:

```
UNIX> ./matrix-serial mode size print
UNIX> ./matrix-parallel mode size print
```

To add matrices, `mode` should be `1`; to multiply, it should be `2`. The integer passed as `size` should be between `1` and `10000`, inclusive. This argument will cause the program to generate two random, square matrices with `size` by `size` dimensions. Pass `0` if printing the results to `stdout` is not desired and `1` if it is.

Executing the Code

The matrices generated by these programs consist of floating point numbers between 0.00 and 100.00, inclusive. Executing the programs with `print` set as `1` will cause the values in the matrices to print, showing two digits after the decimal. Note, however, that the values are *not* truncated to two decimal digits in memory. Since the multiplication and addition operations are applied to the

whole floating point number, there could be some slight discrepancies between the printed results and those calculated by hand with only two decimal places, due to rounding errors.

The following is sample output from both programs:

```
UNIX> ./matrix-serial 2 2 1
Matrix 1:
      84.02      78.31
      91.16      33.52

Matrix 2:
      39.44      79.84
      19.76      76.82

Product:
    4860.58   12724.40
    4257.62    9854.24

UNIX> ./matrix-parallel 2 3 1
Matrix 1:
      84.02      78.31      91.16
      33.52      27.78      47.74
      36.48      95.22      63.57

Matrix 2:
      39.44      79.84      19.76
      76.82      55.40      62.89
      51.34      91.62      71.73

Product:
   14009.96   19399.00   13123.70
    5906.97    8589.23    5833.44
   12017.71   14012.01   11268.87
```

Testing Methodology

To test the runtime and performance of each algorithm, the UNIX `time` command was utilized. Furthermore, commands were run with `nohup` since the matrix multiplication could take several hours. An example syntax is as follows:

```
UNIX> nohup time ./matrix-serial 2 10000 0 > output.txt
```

The command above generates two matrices of size 10,000 by 10,000 and multiplies them. The results are not printed so that time is not spent printing to `stdout`, in order to collect accurate runtimes. With the command above, the statistics regarding time and CPU usage are stored in `output.txt` after the execution of `matrix-serial` is complete.

Note that in the testing for this research, the `srand()` function in the code was

commented out; see the code in the Appendix. This prevents the C `rand()` function from being seeded, so the "random" matrices that are generated are the same every time the program is run. This allows for consistency among the testing; if the matrices change on each iteration, runtimes could be skewed if especially large numbers are generated in one test case.

Three-Body Problem

Compiling and Usage

To compile the `threebody-serial.c` and `threebody-parallel.c` programs to binary executable files respectively named `threebody-serial` and `threebody-parallel`, run the following commands:

```
UNIX> gcc threebody-serial.c -o threebody-serial
UNIX> gcc -fopenmp threebody-parallel.c -o threebody-parallel
```

To execute the binary files, run the commands with the configuration filename as the argument:

```
UNIX> ./threebody-serial configuration-filename
UNIX> ./threebody-parallel configuration-filename
```

The configuration file should strictly follow the format in the sample configuration file `threebody.config`. Its configurable parameters are `step length`, `period`, and the following body properties: `mass`, `position` and `velocity`.

Testing Methodology

To test the runtime and performance of each algorithm, each serial and parallel implementation was run ten times, and the average runtime was taken for different settings of step length values. The times taken for each set of step lengths for both serial and parallel implementations were stored into a file.

Results

Matrix Addition and Multiplication

The code was compiled and executed in the Hydra Lab at the University of Tennessee. The technical specifications of the Hydra machines are identical, with each having an Intel Xeon quad-core processor. Despite the name, each machine actually has eight cores that can be utilized by executable programs. Each algorithm was run on the following machines: *Hydra0, Hydra3, Hydra6, Hydra7, Hydra12,* and *Hydra24*. Multiple machines were used, and the results were averaged, in order to minimize system usage by other simultaneous users. Figure 1 shows the actual time it takes the addition algorithms to complete, as a function of matrix size. The serial and parallel runtimes are plotted as two different lines. It is clear that the serial and parallel algorithms have very similar runtimes for sizes less than 1,000. However, with sizes of 1,000 and 10,000, the parallel algorithm completes the operations significantly faster than the serial version.

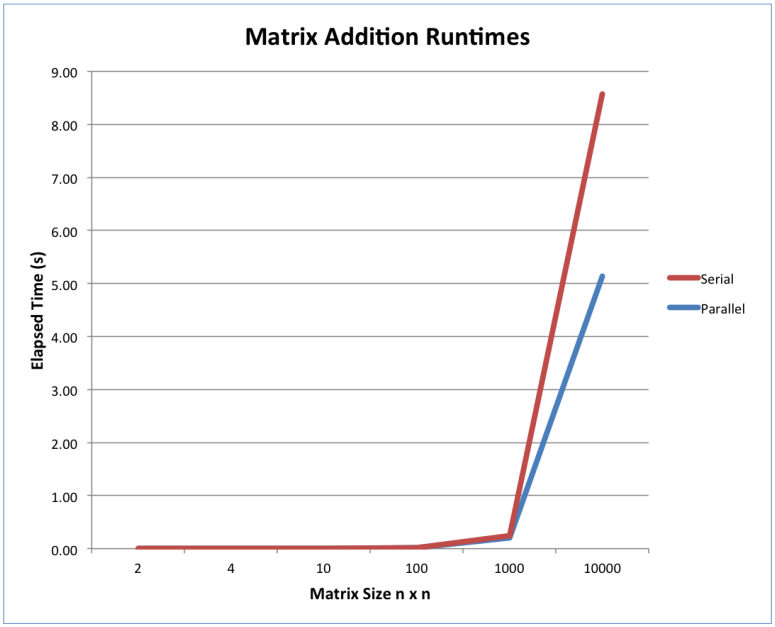

Figure 1: Comparison of parallel and serial matrix addition runtimes.

Figure 2 shows the actual time it takes the multiplication algorithms to complete, as a function of matrix size. The serial and parallel runtimes are plotted as two different lines, and this illustrates the dramatic improvement of the parallel algorithm when used for multiplication of matrices with size greater than 100.

Figures 3 and 4 show the central processing unit (CPU) usage of the multiplication and addition algorithms, with the serial and parallel values graphed alongside each other.

The serial algorithm has minimal CPU usage with matrix sizes of less than 1,000, since the result can be calculated with few operations. However, with sizes of 1,000 or greater, the CPU usage is very close to 100%, which means that the program is utilizing all available computing power within one core of the processor. Since the parallel algorithm can utilize multiple cores simultaneously, the CPU usage can exceed 100%. Since the Hydra machines on which the tests were run have eight cores, the theoretical maximum CPU usage is 800%. This threshold is almost reached in the parallel matrix multiplication with size 1,000 or greater, which demonstrates the power of parallel algorithms when utilized with computing-intensive applications. However, the usage of threads is not as beneficial when given smaller matrix sizes of 1,000 or less, considering the usage of around 400%, and the slower runtimes as compared to the times of the serial algorithms. Figures 5 and 6 contain the runtime and CPU usage of each algorithm tested with a certain size, as shown in the table. This data contains averages of six different iterations of the same test. The User and System Time, when added together, is essentially the amount of processing time that the

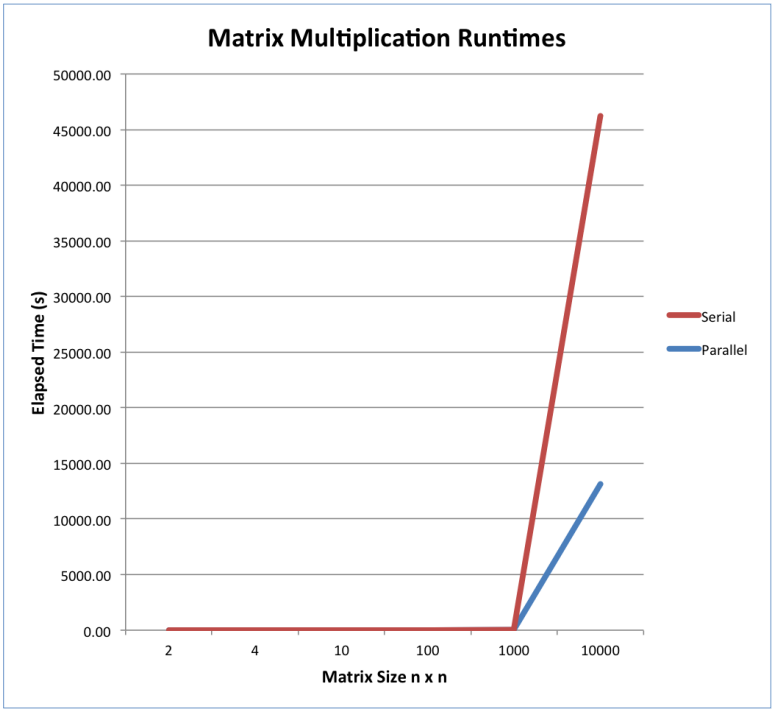

Figure 2: Comparison of parallel and serial matrix multiplication runtimes.

computer commits to executing the program. With the serial algorithms, the sum of these two values should be very close to the Elapsed Time, since these programs only utilize one core. However, with the parallel algorithms, the sum of the User and System Time can be up to eight times higher than the Elapsed Time, since the algorithm can utilize up to eight cores simultaneously. The CPU Usage indicates how many cores the program utilized. A value of 99% would indicate that one core is being fully used, while a value of 770% indicates that almost all eight cores were simultaneously utilized by the algorithm.

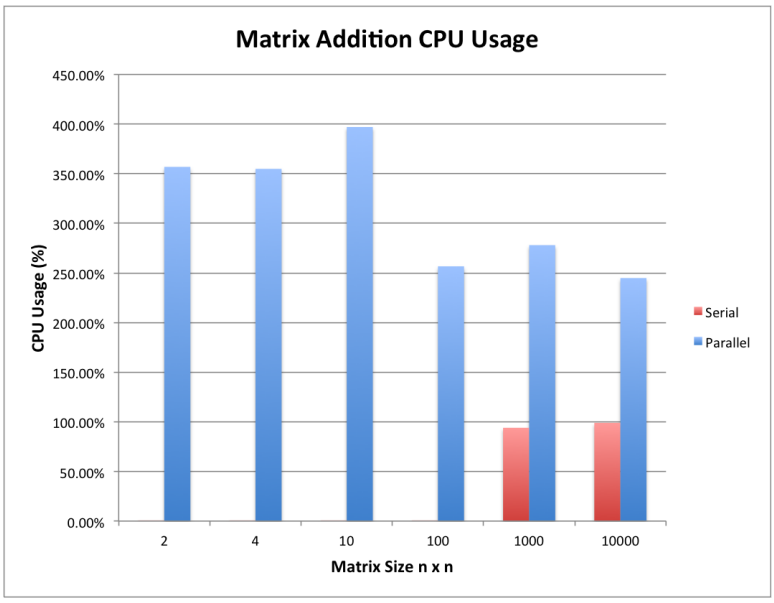

Figure 3: Comparison of CPU usage for parallel and serial matrix addition.

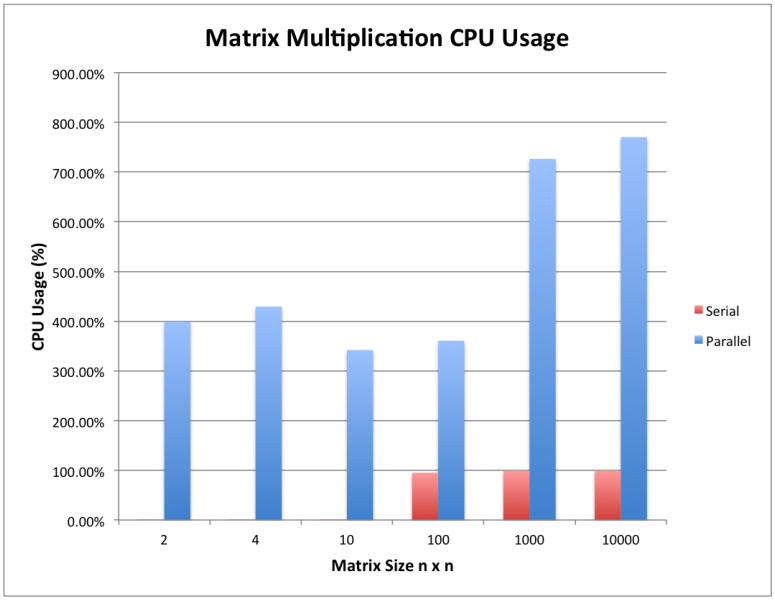

Figure 4: Comparison of CPU usage for parallel and serial matrix multiplication.

Process	Size n x n	User Time (s)	System Time (s)	Elapsed Time (s)	CPU Usage
Serial	2	0.00	0.00	0.00	0.00%
Parallel	2	0.02	0.00	0.00	356.83%
Serial	4	0.00	0.00	0.00	0.00%
Parallel	4	0.02	0.00	0.00	354.83%
Serial	10	0.00	0.00	0.00	0.00%
Parallel	10	0.02	0.00	0.00	397.00%
Serial	100	0.00	0.00	0.00	0.00%
Parallel	100	0.02	0.03	0.02	256.67%
Serial	1000	0.03	0.01	0.04	94.00%
Parallel	1000	0.15	0.43	0.21	278.00%
Serial	10000	2.38	1.04	3.43	99.00%
Parallel	10000	5.50	7.10	5.14	244.83%

Figure 5: Average results of running matrix addition algorithm.

Process	Size n x n	User Time (s)	System Time (s)	Elapsed Time (s)	CPU Usage
Serial	2	0.00	0.00	0.00	0.00%
Parallel	2	0.03	0.00	0.00	399.33%
Serial	4	0.00	0.00	0.00	0.00%
Parallel	4	0.02	0.01	0.01	429.33%
Serial	10	0.00	0.00	0.00	0.00%
Parallel	10	0.02	0.00	0.00	341.83%
Serial	100	0.01	0.00	0.01	95.00%
Parallel	100	0.05	0.04	0.02	360.67%
Serial	1000	23.55	0.01	23.64	99.00%
Parallel	1000	37.49	0.34	5.20	726.33%
Serial	10000	32977.41	2.24	33078.00	99.00%
Parallel	10000	100808.81	37.02	13162.50	770.17%

Figure 6: Average results of running matrix multiplication algorithm.

Three-Body Problem

For both the serial and parallel simulations of the three-body problem, each set of parameters was run ten times, and the average runtime of each simulation was computed. The three-body problem simulation was done for three different sets of values for step length, which is the total number of steps of simulation: `100000`, `1000000` and `10000000`.

OpenMP was used to parallelize the code for simulation. Parallelization was implemented to the simulation for each step and could not be used for multiple steps directly because the properties of each body depend on the properties of all bodies in the preceding step. With that dependency, it could not be parallelized.

Runtime was scaled linearly with increases in step length for both serial and parallel simulations.

In this simulation, as opposed to the ideal case, parallel implementations performed more poorly than serial implementations. The runtimes of parallel implementations were slightly lower than those of the serial run for each value of the step length. This is most likely because the overhead of creating threads and joining them for synchronization in each step overcame the benefit of threading.

Figure 7 shows the runtime comparison of the parallel and serial implementations of the three-body problem.

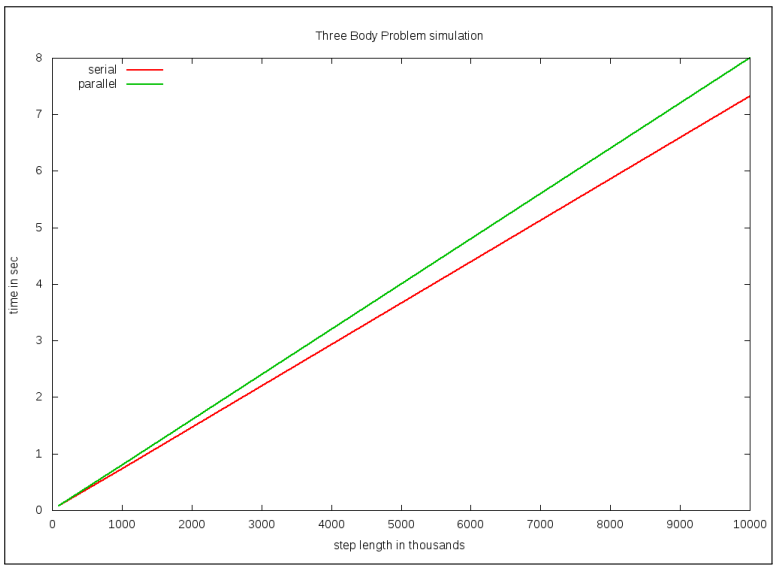

Figure 7: Comparison of the runtimes of parallel and serial implementations of the three-body problem simulation.

Conclusion

Utilizing parallel algorithms is very important when thousands, or even millions, of operations must be performed in order to calculate a result. With addition and multiplication of square matrices of size 10,000 or greater, the increased speed and efficiency, as shown in the results, demonstrates the usefulness of parallel algorithms and the effectiveness of OpenMP. The serial multiplication of a matrix of size 10,000 took over nine hours to complete, while the parallel algorithm took just over three hours to calculate the result. However, using OpenMP threads with small matrices and the three-body problem actually caused the parallel algorithm to take longer to complete than the serial version. This is because the time required to initialize and utilize the OpenMP threads offsets the benefit of the increased speed of the calculations.

Parallel algorithms, implemented with threads, are necessary to conduct simulations that process millions of operations in a short period of time. Overall, though, it is important to know when the usage of threads can hinder performance, as demonstrated in the results. This research proves that OpenMP is a powerful tool when applied to appropriate algorithms. However, in order to make the runtimes of processes that conduct few floating-point operations more efficient, OpenMP would need additional development.

Appendix

matrix-serial.c

```
/*****************************************************************************
 * FILE:    matrix-serial.c
 * AUTHOR:  Drew Nash - anash4@utk.edu
 * DATE:    30 April 2015
 * COURSE:  CS 560 - PA3
 * This program implements serial matrix addition and multiplication.
*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* This program generates random M x M square matrices, with entries being
   floating point numbers in the following range: [0, 100]. The syntax to
   run this program is as follows:
   ./matrix-serial mode size print
   with 'mode' being either '1' for sum or '2' for product, 'size' being
   an integer so that the generated matrices are 'size' x 'size', and
   'print' being '0' or '1' for false or true, respectively.
*/

/* Print the matrix that is passed. */
void print_matrix(int size, long double **M){
    int i, j;

    for (i = 0; i < size; i++){
        for (j = 0; j < size; j++){
            printf(" %10.2Lf", M[i][j]); /* 'Lf' is used with long doubles */
        }
        printf("\n");
    }
    return;
}

/* Add two passed matrices 'M1' and 'M2' and store the sum in 'result' */
void serial_add(int size, long double **M1, long double **M2, long double **result){
    int i, j;

    for (i = 0; i < size; i++){
        for (j = 0; j < size; j++){
            result[i][j] = M1[i][j] + M2[i][j];
        }
    }
    return;
}

/* Multiply two passed matrices 'M1' and 'M2' and store the product in 'result' */
void serial_multiply(int size, long double **M1, long double **M2, long double **result){
    int i, j, k;

    /* Algorithm is referenced from ``Introduction to Algorithms'' in Cormen et al. */
    for (i = 0; i < size; i++){
        for (j = 0; j < size; j++){
            result[i][j] = 0;
            for (k = 0; k < size; k++){
                result[i][j] += M1[i][k] * M2[k][j];
            }
        }
    }
    return;
}

int main(int argc, char **argv){
    int i, j, size, mode, print;

    /* Error checking to confirm correct syntax to run program */
    if (argc != 4){
        fprintf(stderr, "ERROR - Usage: matrix-serial mode size print\n");
        exit(1);
    }

//  srand(time(NULL));      /* Seed random number generator with time */
    print = atoi(argv[3]);
    size = atoi(argv[2]);
    mode = atoi(argv[1]);

    if (mode < 1 || mode > 2){
        fprintf(stderr, "ERROR - Modes: Sum = 1 | Product = 2\n");
        exit(1);
    }
    if (size < 1 || size > 10000){
        fprintf(stderr, "ERROR - Size: int between 1 and 10000\n");
        exit(1);
    }
    if (print < 0 || print > 1){
        fprintf(stderr, "ERROR - Print: False = 0 | True = 1\n");
        exit(1);
    }

    /* Matrices are 2-dimensional arrays of long doubles, so that large values
       can be stored, which is important when multiplying large matrices */
    long double **M1;
    long double **M2;
```

```
    long double **result;

    /* Memory must be allocated for the primary array of pointers to long doubles */
    M1 = (long double **) malloc(size * sizeof(long double *));
    M2 = (long double **) malloc(size * sizeof(long double *));
    result = (long double **) malloc(size * sizeof(long double *));

    /* Memory must be allocated for each secondary array of long doubles */
    for (i = 0; i < size; i++){
        M1[i] = (long double *) malloc(size * sizeof(long double));
        M2[i] = (long double *) malloc(size * sizeof(long double));
        result[i] = (long double *) malloc(size * sizeof(long double));
    }

    /* Load 'M1' and 'M2' with random floating point numbers from 0 to 100, inclusive */
    for (i = 0; i < size; i++){
        for (j = 0; j < size; j++){
            M1[i][j] = (long double) rand() / (long double) (RAND_MAX / 100);
            M2[i][j] = (long double) rand() / (long double) (RAND_MAX / 100);
        }
    }

    /* Print matrices 'M1' and 'M2', if requested */
    if (print){
        printf("Matrix 1:\n");
        print_matrix(size, M1);
        printf("\n");
        printf("Matrix 2:\n");
        print_matrix(size, M2);
        printf("\n");
    }

    if (mode == 1){ /* Add 'M1' and 'M2', then print 'result' */
        serial_add(size, M1, M2, result);
        if (print){ /* Print only if requested */
            printf("Sum:\n");
            print_matrix(size, result);
            printf("\n");
        }
    }
    else if (mode == 2){    /* Multiply 'M1' and 'M2', then print 'result' */
        serial_multiply(size, M1, M2, result);
        if (print){ /* Print only if requested */
            printf("Product:\n");
            print_matrix(size, result);
            printf("\n");
        }
    }

    return 0;
}
```

matrix-parallel.c

```
/******************************************************************************
 * FILE:    matrix-parallel.c
 * AUTHOR:  Drew Nash - anash4@utk.edu
 * DATE:    30 April 2015
 * COURSE:  CS 560 - PA3
 * This program implements parallel matrix addition and multiplication.
******************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

/* This program generates random M x M square matrices, with entries being
   floating point numbers in the following range: [0, 100]. The syntax to
   run this program is as follows:
   ./matrix-parallel mode size print
   with 'mode' being either '1' for sum or '2' for product, 'size' being
   an integer so that the generated matrices are 'size' x 'size', and
   'print' being '0' or '1' for false or true, respectively.
*/

/* Print the matrix that is passed. */
void print_matrix(int size, long double **M){
    int i, j;

    for (i = 0; i < size; i++){
        for (j = 0; j < size; j++){
            printf(" %10.2Lf", M[i][j]); /* 'Lf' is used with long doubles */
        }
        printf("\n");
    }
    return;
}

/* Add two passed matrices 'M1' and 'M2' and store the sum in 'result' */
void parallel_add(int size, long double **M1, long double **M2, long double **result){
    int i, j;

    omp_set_nested(1); /* Set nested to true so two teams of threads are used */
    #pragma omp parallel for /* All iterations of for loop executed simulataneously */
    for (i = 0; i < size; i++){
        #pragma omp parallel for /* One thread handles each iteration of for loop */
        for (j = 0; j < size; j++){
            result[i][j] = M1[i][j] + M2[i][j]; /* No need for lock since value is updated once */
        }
    }
    return;
}

/* Multiply two passed matrices 'M1' and 'M2' and store the product in 'result' */
void parallel_multiply(int size, long double **M1, long double **M2, long double **result){
    int i, j, k;

    omp_set_nested(1); /* Set nested to true so two teams of threads are used */

    /* Algorithm is referenced from ''Introduction to Algorithms'' in Cormen et al. */
    #pragma omp parallel for /* All iterations of for loop executed simulataneously */
    for (i = 0; i < size; i++){
        #pragma omp parallel for private(k) /* One thread handles each iteration of for loop */
        for (j = 0; j < size; j++){
            result[i][j] = 0; /* No need for lock since value is updated once */
            /* 'k' is set to private so each thread can go through its own set of iterations */
            for (k = 0; k < size; k++){
                result[i][j] += M1[i][k] * M2[k][j];
            }
        }
    }
    return;
}

int main(int argc, char **argv){
    int i, j, size, mode, print;

    /* Error checking to confirm correct syntax to run program */
    if (argc != 4){
        fprintf(stderr, "ERROR - Usage: matrix-parallel mode size print\n");
        exit(1);
    }

//  srand(time(NULL));      /* Seed random number generator with time */
    print = atoi(argv[3]);
    size = atoi(argv[2]);
    mode = atoi(argv[1]);

    if (mode < 1 || mode > 2){
        fprintf(stderr, "ERROR - Modes: Sum = 1 | Product = 2\n");
        exit(1);
    }
    if (size < 1 || size > 10000){
        fprintf(stderr, "ERROR - Size: int between 1 and 10000\n");
        exit(1);
    }
    if (print < 0 || print > 1){
        fprintf(stderr, "ERROR - Print: False = 0 | True = 1\n");
        exit(1);
```

```
    }

    /* Matrices are 2-dimensional arrays of long doubles, so that large values
       can be stored, which is important when multiplying large matrices */
    long double **M1;
    long double **M2;
    long double **result;

    /* Memory must be allocated for the primary array of pointers to long doubles */
    M1 = (long double **) malloc(size * sizeof(long double *));
    M2 = (long double **) malloc(size * sizeof(long double *));
    result = (long double **) malloc(size * sizeof(long double *));

    /* Memory must be allocated for each secondary array of long doubles */
    /* NOT parallelized to accurately compare parallel and singular matrix operations */
    for (i = 0; i < size; i++){
        M1[i] = (long double *) malloc(size * sizeof(long double));
        M2[i] = (long double *) malloc(size * sizeof(long double));
        result[i] = (long double *) malloc(size * sizeof(long double));
    }

    /* Load 'M1' and 'M2' with random floating point numbers from 0 to 100, inclusive */
    /* NOT parallelized to accurately compare parallel and singular matrix operations */
    for (i = 0; i < size; i++){
        for (j = 0; j < size; j++){
            M1[i][j] = (long double) rand() / (long double) (RAND_MAX / 100);
            M2[i][j] = (long double) rand() / (long double) (RAND_MAX / 100);
        }
    }

    /* Print matrices 'M1' and 'M2', if requested */
    if (print){
        printf("Matrix 1:\n");
        print_matrix(size, M1);
        printf("\n");
        printf("Matrix 2:\n");
        print_matrix(size, M2);
        printf("\n");
    }

    if (mode == 1){ /* Add 'M1' and 'M2', then print 'result' */
        parallel_add(size, M1, M2, result);
        if (print){ /* Print only if requested */
            printf("Sum:\n");
            print_matrix(size, result);
            printf("\n");
        }
    }
    else if (mode == 2){    /* Multiply 'M1' and 'M2', then print 'result' */
        parallel_multiply(size, M1, M2, result);
        if (print){ /* Print only if requested */
            printf("Product:\n");
            print_matrix(size, result);
            printf("\n");
        }
    }

    return 0;
}
```

threebody-serial.c

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

// struct for body
typedef struct body{
  double m;                 // mass
  double x; double y;       // x and y position
  double vx; double vy;     // velocity x and y direction
  double ax; double ay;     // acceleration x and y direction
} Body;

//*** global variables ***//
Body B[3];
double DT;
int STEP_LENGTH;


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

// simulation for a step for n bodies
void simulation_nbody(Body *bd, int n, double dt)
{
  int i, j;
  double f;  // this could be made array and no. of calculation could be decreased but for now naive approach
  Body *b = malloc(sizeof(Body)*n);
  for(i = 0; i < n; i++){
    b[i].m = bd[i].m;
    b[i].x = bd[i].x;
    b[i].y = bd[i].y;
    b[i].vx = bd[i].vx;
    b[i].vy = bd[i].vy;
  }

  for(i = 0; i < n; i++){
    // initialize acceleration to 0 ; calculated as sum of acceleration due to other bodies
    bd[i].ax = 0; bd[i].ay = 0;
    // calculate acceleration due to forces
    for(j = 0; j < n; j++){
      if(i == j) continue;    // no sense in calculating force from itself
      // calculate acceleration due to other body
      f = sqrt ( pow( square( b[i].x - b[j].x) + square(b[i].y - b[j].y), 3 ) );  // cube of distance between two bodies
      bd[i].ax += - b[j].m * ( b[i].x - b[j].x ) / f;
      bd[i].ay += - b[j].m * ( b[i].y - b[j].y ) / f;
    }
    // update position of body      s = ut + 1/2*a*t^2
    bd[i].x += (bd[i].vx + 0.5*bd[i].ax*dt)*dt;
    bd[i].y += (bd[i].vy + 0.5*bd[i].ay*dt)*dt;
    // update velocity of a body    v = u + at;
    bd[i].vx += bd[i].ax * dt;
    bd[i].vy += bd[i].ay * dt;
  }
  // update other properties of body

  free(b);
}


#define EXPECT(a,b,c) if ((a) != fscanf(f,b "%*[ ^\n]\n",c)){ fclose(f); printf("Error: %s\n",b); return 1; }

int read_config(char *file_name)
{
  FILE *f;

  if (!(f = fopen(file_name,"r"))) return 1;
  EXPECT(1, "int    steps  = %d;", &STEP_LENGTH);
  EXPECT(1, "double period = %lf;", &DT);
  EXPECT(1, "double mass   = %lf;", &B[0].m);
  EXPECT(1, "double pos_x  = %lf;", &B[0].x);
  EXPECT(1, "double pos_y  = %lf;", &B[0].y);
  EXPECT(1, "double vel_x  = %lf;", &B[0].vx);
  EXPECT(1, "double vel_y  = %lf;", &B[0].vy);
  EXPECT(1, "double mass   = %lf;", &B[1].m);
  EXPECT(1, "double pos_x  = %lf;", &B[1].x);
  EXPECT(1, "double pos_y  = %lf;", &B[1].y);
  EXPECT(1, "double vel_x  = %lf;", &B[1].vx);
  EXPECT(1, "double vel_y  = %lf;", &B[1].vy);
  EXPECT(1, "double mass   = %lf;", &B[2].m);
  EXPECT(1, "double pos_x  = %lf;", &B[2].x);
  EXPECT(1, "double pos_y  = %lf;", &B[2].y);
  EXPECT(1, "double vel_x  = %lf;", &B[2].vx);
  EXPECT(1, "double vel_y  = %lf;", &B[2].vy);

  fclose(f); return 0;
}
#undef EXPECT


int main(int argc, char** argv)
{
  if (argc ^ 2){
    printf("Usage: ./threebody config_file\n");
```

```
    exit(1);
  }
  if (read_config(argv[1])){                    // read config file
    printf("READDATA: Can't process %s\n", argv[1]);
    return 1;
  }

  int i,j;
  clock_t start, end;
  start = clock();                      // start of simulation for all steps
  for( j = 0; j < 10; j++){
    for( i = 0; i < STEP_LENGTH; i++){
      simulation_nbody(B, 3, DT);
    }
  }
  end = clock();
  // print stats for each bodies
  for( i = 0; i < 3; i++){
    printf("Body #%d:\n", i+1);
    printf("mass: %.2f\n", B[i].m);
    printf("position: %.2f, %.2f\n", B[i].x, B[i].y);
    printf("velocity: %.2f, %.2f\n", B[i].vx, B[i].vy);
    printf("acceleration: %.8f, %.8f\n", B[i].ax, B[i].ay);
    printf("\n");
  }
  printf("time of execution: %.8f\n", (double)(end-start)/(double)(CLOCKS_PER_SEC*10));

  return 0;
}
```

threebody-parallel.c

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

// struct for body
typedef struct body{
  double m;                // mass
  double x; double y;      // x and y position
  double vx; double vy;    // velocity x and y direction
  double ax; double ay;    // acceleration x and y direction
} Body;

//*** global variables ***//
Body B[3];
double DT;
int STEP_LENGTH;


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

// simulation for a step for n bodies
void simulation_nbody(Body *b, int n, int step_length, double dt)
{
  int i,j,k,l;
  Body *bd = malloc(n*sizeof(Body));
  double *f = malloc(n*sizeof(double));
  #pragma omp parallel private(k)
  {
    for( k = 0; k < step_length; k++){
      #pragma omp for
      for(l = 0; l < n; l++){
bd[l].m = B[l].m;
bd[l].x = B[l].x;
bd[l].y = B[l].y;
bd[l].vx = B[l].vx;
bd[l].vy = B[l].vy;
      }

#pragma omp for private(j)
      for(i = 0; i < n; i++){
// initialize acceleration to 0 ; calculated as sum of acceleration due to other bodies
b[i].ax = 0; b[i].ay = 0;
// calculate acceleration due to forces
for(j = 0; j < n; j++){
  if(i == j) continue;    // no sense in calculating force from itself
  // calculate acceleration due to other body
  f[i] = sqrt ( pow( square( bd[i].x - bd[j].x) + square(bd[i].y - bd[j].y), 3 ) );  // cube of distance between two bodies
  b[i].ax += - bd[j].m * ( bd[i].x - bd[j].x ) / f[i];
  b[i].ay += - bd[j].m * ( bd[i].y - bd[j].y ) / f[i];
}
// update position of body      s = ut + 1/2*a*t^2
b[i].x += (b[i].vx + 0.5*B[i].ax*dt)*dt;
b[i].y += (b[i].vy + 0.5*B[i].ay*dt)*dt;
// update velocity of a body    v = u + at;
b[i].vx += b[i].ax * dt;
b[i].vy += b[i].ay * dt;
      }
    }
  }
  free(bd);
  free(f);
}

#define EXPECT(a,b,c) if ((a) != fscanf(f,b "%*[ ^\n]\n",c)){ fclose(f); printf("Error: %s\n",b); return 1; }

int read_config(char *file_name)
{
  FILE *f;

  if (!(f = fopen(file_name,"r"))) return 1;
  EXPECT(1, "int    steps  = %d;", &STEP_LENGTH);
  EXPECT(1, "double period = %lf;", &DT);
  EXPECT(1, "double mass   = %lf;", &B[0].m);
  EXPECT(1, "double pos_x  = %lf;", &B[0].x);
  EXPECT(1, "double pos_y  = %lf;", &B[0].y);
  EXPECT(1, "double vel_x  = %lf;", &B[0].vx);
  EXPECT(1, "double vel_y  = %lf;", &B[0].vy);
  EXPECT(1, "double mass   = %lf;", &B[1].m);
  EXPECT(1, "double pos_x  = %lf;", &B[1].x);
  EXPECT(1, "double pos_y  = %lf;", &B[1].y);
  EXPECT(1, "double vel_x  = %lf;", &B[1].vx);
  EXPECT(1, "double vel_y  = %lf;", &B[1].vy);
  EXPECT(1, "double mass   = %lf;", &B[2].m);
  EXPECT(1, "double pos_x  = %lf;", &B[2].x);
  EXPECT(1, "double pos_y  = %lf;", &B[2].y);
  EXPECT(1, "double vel_x  = %lf;", &B[2].vx);
  EXPECT(1, "double vel_y  = %lf;", &B[2].vy);

  fclose(f); return 0;
}
#undef EXPECT
```

```
int main(int argc, char** argv)
{
  if (argc ^ 2){
    printf("Usage: ./threebody config_file\n");
    exit(1);
  }
  if (read_config(argv[1])){                    // read config file
    printf("READDATA: Can't process %s\n", argv[1]);
    return 1;
  }

  int i;
  time_t start, end;
  time(&start);                    // start of simulation for all steps
  //Body bd[3]; // = malloc(sizeof(Body)*3);

  //double f[3];
  omp_set_num_threads(3);
  for(i = 0; i < 10; i++){
    simulation_nbody(B, 3, STEP_LENGTH, DT);
  }
  time(&end);

  // print stats for each bodies
  for( i = 0; i < 3; i++){
    printf("Body #%d:\n", i+1);
    printf("mass: %.2f\n", B[i].m);
    printf("position: %.2f, %.2f\n", B[i].x, B[i].y);
    printf("velocity: %.2f, %.2f\n", B[i].vx, B[i].vy);
    printf("acceleration: %.8f, %.8f\n", B[i].ax, B[i].ay);
    printf("\n");
  }
  printf("time of execution: %.7f\n", (double)difftime(end,start)/(double)10);
  //free(b);


return 0;
}
```

threebody.config

```
int    steps  = 10000000;
double period = 0.5;
double mass   = 0.5;
double pos_x  = 0;
double pos_y  = 1;
double vel_x  = 1;
double vel_y  = 1.2;
double mass   = 3;
double pos_x  = 6;
double pos_y  = 4;
double vel_x  = 1;
double vel_y  = 1.2;
double mass   = 2;
double pos_x  = 2;
double pos_y  = 3;
double vel_x  = 1;
double vel_y  = 1.2;
```

References

Barney, Blaise. *OpenMP*. Lawrence Livermore National Laboratory, 2014. Web. 30 Apr. 2015.

Burkardt, John. *THREE BODY SIMULATION* . Florida State University, 2012. Web. 30 Apr. 2015.

Cormen, Thomas H., et al. *Introduction to Algorithms*. Cambridge: MIT, 2009. Print.