CPU Performance Benchmark

Written by: Juan Ledezma Gomez

CSC 641 Computer Performance Evaluation

Homework #1

February 13, 2018

Content

Introduction

The purpose of this project was to create a benchmark program to test a computer's CPU performance. A CPU's performance depends on different factors, like the processor speed, memory speed, and bus performance. To measure its performance, this program needed to reproduce workload that is similar to the most frequent actual workload.

Proposed Solution

To accurately calculate a computer's processor performance two different types of operations were used, numeric and combinatorial. For this project I used matrix inversion for the numeric operations, using double precision floating points, and the sorting algorithm quicksort to measure integer operations.

For matrix inversion, in order to ensure the calculations are correct across different systems, I used a matrix that was composed of three parts. The first is a square matrix that contains the float value "2.0001" in the main diagonal and "1.0001" elsewhere. The second region is a vector that contains the square root of 1 through n, where n is the size of the square matrix. Lastly, the identity matrix was attached at the end. I used the Gauss-Jordan algorithm on the whole matrix which resulted in regenerating the original matrix containing float values and the vector. The determinant of both the original matrix and its inverse was also calculated. This whole process was looped many times so that the elapsed time of this set of operations was approximately 2 seconds. These operations were embedded inside another loop that ran for 10 seconds. Finally, I computed the processor speed for the floating point numeric operations part of the benchmark.

Similar in structure to the matrix inversion process, I computed the speed for integer operations by using quicksort. First, I populated a large array with a repeating sequence of integers decreasing from 5 to 1. The array was then sorted with the quicksort function, in a

manner of increasing values. The quicksort algorithm uses recursion to sort the array. [1] The process of populating the array and sorting it was also looped for approximately 2 seconds, and then this was looped for 10 seconds. Using the same formula as the floating-point operations, I computed the processor speed for integer operations. Ultimately, the average processor speed was calculated using the harmonic means of both speeds based on equal weights.

Experiments

This benchmark was ran on three machines, two using Windows OS and one using Linux. The results of the average processor speed have been multiplied by 10 and rounded to the nearest whole number to create a clearer score:

1st Computer Tested (by third party)

Processor: Intel Core i5 4570 @ 3.2 GHz

Number of cores: 4

OS: Windows 10

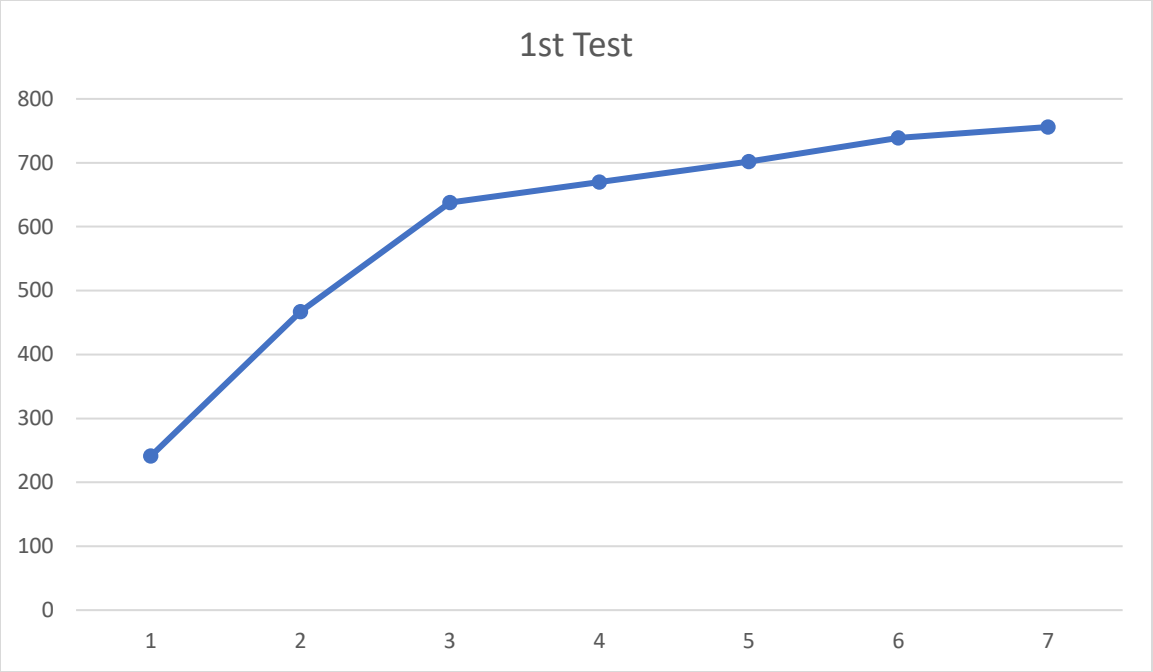
Results:

# of copies	1	2	3	4	5	6	7	∑
1	241							**241**
2	234	233						**467**
3	211	212	215					**638**
4	167	177	155	171				**670**
5	142	146	134	137	143			**702**
6	110	131	110	135	134	119		**739**
7	112	107	107	113	106	106	105	**756**

467 / 241 = 1.9

90% improvement with a second core

Parallelism ratio = 756 / 3.2 = 236.25



Actual values calculated by program for the number of copies ran:

1. 24.1063

2. 23.3884, 23.3452

3. 21.1147, 21.1584, 21.4602

4. 16.6636, 17.7198, 15.4566, 17.0883

5. 14.2483, 14.5655, 13.4043, 13.7478, 14.2948

6. 11.036, 13.1339, 11.0336, 13.461, 13.3924, 11.9435

7. 11.1836, 10.7306, 10.7349, 11.3019, 10.6411, 10.5862, 10.4603

2nd Computer Tested

Processor: Intel Core i5 4670K @ 3.4 GHz

Number of cores: 4

OS: Windows 10

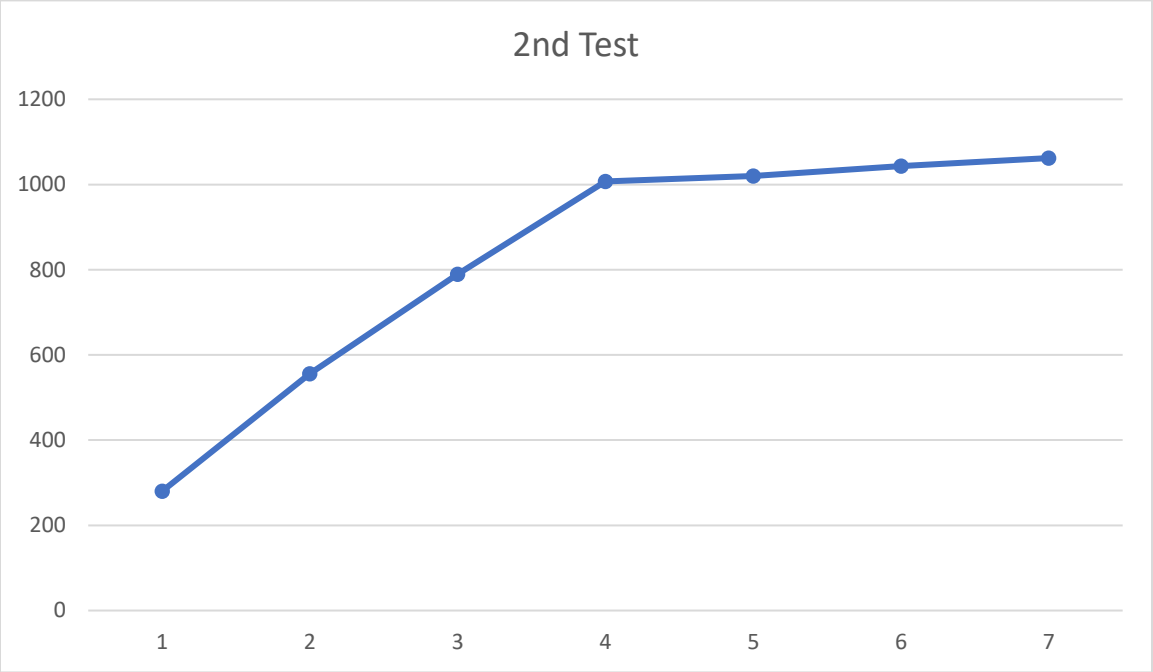
Results:

# of copies	1	2	3	4	5	6	7	∑
1	280							**280**
2	280	276						**556**
3	263	263	263					**789**
4	253	252	250	252				**1007**
5	199	212	196	213	200			**1020**
6	174	166	165	192	172	174		**1043**
7	149	152	149	147	159	147	159	**1062**

556 / 280 = 1.99

99% improvement with a second core

Parallelism ratio = 1062 / 3.4 = 312.35



Actual values calculated by program for the number of copies ran:

1. 27.9839

2. 28.0138, 27.5926

3. 26.2973, 26.2582, 26.2605

4. 25.2791, 25.1625, 25.0086, 25.1857

5. 19.9046, 21.1612, 19.5862, 21.2889, 19.9443

6. 17.3603, 16.5593, 16.4896, 19.1678, 17.1813, 17.4385

7. 14.9007, 15.1924, 14.9118, 14.7239, 15.9476, 14.7047, 15.8905

3rd Computer Tested

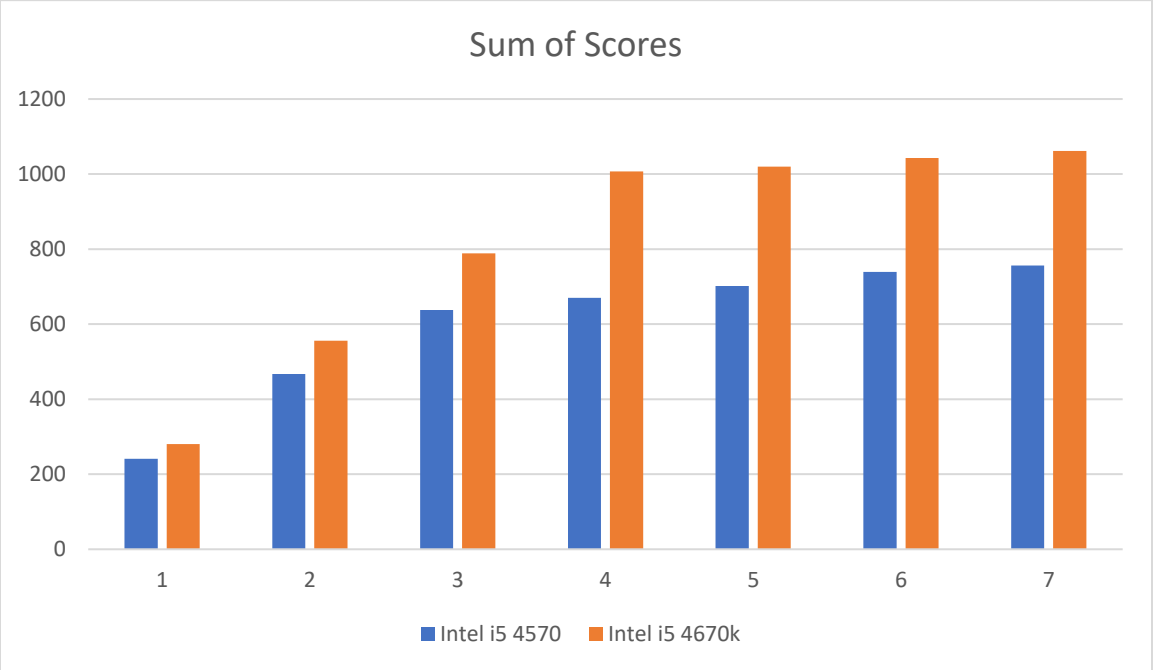
Processor: Intel Celeron N3050 @ 1.6 GHz

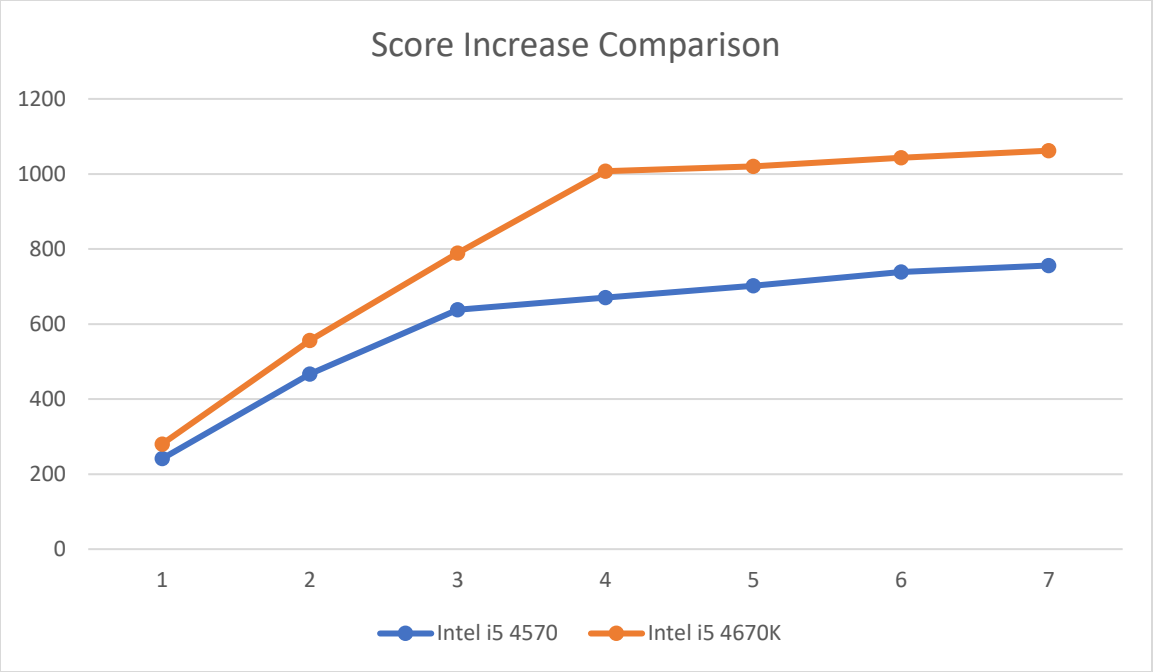
Number of cores: 2

OS: Linux Mint 18.1 Cinnamon 64-bit

*** The program ran, but it stalled doing the matrix inversions. Perhaps it is caused from the size of the matrix and number of times the operation is ran.

Comparison





Conclusion

This benchmark is able to show a processor's performance increase with multiple cores and how the performance decreases when the number of programs running exceeds the number of cores. The program seems to work well with better performing processors, but when tested with a slower processor the program was not able to run properly. This may be due to inappropriate sizes for the structures used in the operations.

Appendix

References

[1] http://www.algolist.net/Algorithms/Sorting/Quicksort

Source Code

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

using namespace std;

void matrixInversion(double[][241], int);
void fillMatrix(double[][241], int);
void linq(double[][241], double&, int);

void quickSort(int[], int, int);
void fillArray(int[], int);
double sec(void);
int testArray(int[], int);
```

```
int main() {
    int numIntOps = 0;
    int numFloatOps = 0;
    int i;
    double start;
    double intSpeed, floatSpeed, averageSpeed;

    int sortArray[120000];
    double matrix[120][241];

    double timeTest1;
    double timeTest2;

    cout << "** Benchmarking CPU Performance **\n\n";

    start = sec();
    while (sec() < start + 10) {
        timeTest1 = sec();

        for (i = 0; i < 1000; i++) {
            fillMatrix(matrix, 120);
            matrixInversion(matrix, 120);
        }

        numFloatOps++;
        timeTest2 = sec();
        cout << "Matrix inversion time: " << timeTest2 - timeTest1 << " sec\n\n";
    }
    floatSpeed = 60 * numFloatOps / (sec() - start);

    start = sec();
    while (sec() < start + 10) {
        timeTest1 = sec();

        for (i = 0; i < 1000; i++) {
              fillArray(sortArray, 120000);
            quickSort(sortArray, 0, 119999);

            if (!testArray(sortArray, 120000)) {
                cout << "\nArray is not sorted! Exiting program.\n";
                exit(0);
            }
        }

        numIntOps++;
        timeTest2 = sec();
        cout << "Quicksort time: " << timeTest2 - timeTest1 << " sec\n\n";
    }
    intSpeed = 60 * numIntOps / (sec() - start);

    averageSpeed = 2 * floatSpeed * intSpeed / (floatSpeed + intSpeed);
    cout << "Floating point speed (using matrix inversion): " << floatSpeed << endl;
    cout << "Integer speed (using quick sort): " << intSpeed << endl;
    cout << "Average speed: " << averageSpeed << " operations per minute" << endl;

       system("pause");
    return 0;
}
```

```
double sec() {
    return double(clock()) / double(CLOCKS_PER_SEC);
}

void fillMatrix(double inversionMatrix[][241], int size) {
    int i, j;

    for (i = 0; i < size; i++) {
        for (j = 0; j < size; j++) {
            if (i == j) {
                inversionMatrix[i][j] = 2.0001;
                inversionMatrix[i][size + 1 + j] = 1.0;
            } else {
                inversionMatrix[i][j] = 1.0001;
                inversionMatrix[i][size + 1 + j] = 0.0;
            }
        }
        inversionMatrix[i][size] = sqrt(i + 1);
    }
}

void matrixInversion(double matrix[][241], int size) {
    double determinant;
    double determinantInverse;
    double vectorError;
    double temp;
    int i, j;

    linq(matrix, determinant, size);

    for (i = 0; i < size; i++) {
        for (j = 0; j < size; j++) {
            temp = matrix[i][j];
            matrix[i][j] = matrix[i][size + 1 + j];
            matrix[i][size + 1 + j] = temp;
        }
    }

    linq(matrix, determinantInverse, size);

    //cout << "Determinant of matrix: " << determinant << endl;
    //cout << "Determinant of matrix inverse: " << determinantInverse << endl;

       /*
    vectorError = 0.0;
    for (i = 0; i < size; i++) {
        vectorError = vectorError + abs(sqrt(i + 1) - matrix[i][size]);
    }
    cout << "Determinant error: " << 1.0 - determinant * determinantInverse << endl;
    cout << "Vector error: " << vectorError << endl;
       */

}

void linq(double matrix[][241], double& determinant, int size) {
    double TIK, TKK;
    int i, j, k;
```

```
    determinant = 1.0;

    for (k = 0; k < size; k++) {
        TKK = matrix[k][k];
        determinant = determinant * TKK;

        for (j = 0; j < size * 2 + 1; j++) {
            matrix[k][j] = matrix[k][j] / TKK;
        }

        for (i = 0; i < size; i++) {
            if (i != k) {
                TIK = matrix[i][k];

                for (j = 0; j < size * 2 + 1; j++) {
                    matrix[i][j] = matrix[i][j] - matrix[k][j] * TIK;
                }
            }
        }

    }
}

void fillArray(int quickSortArray[], int size) {
    int i;
    int fillValue = 5;

    for (i = 0; i < size; i++) {
        quickSortArray[i] = fillValue;
        fillValue--;

        if (fillValue == 0) {
            fillValue = 5;
        }
    }
}

void quickSort(int ary[], int left, int right) {
    int i = left;
    int j = right;
    int temp;
    int pivot = ary[(left + right) / 2];

    while (i <= j) {
        while (ary[i] < pivot) {
            i++;
        }
        while (ary[j] > pivot) {
            j--;
        }

        if (i <= j) {
            temp = ary[i];
            ary[i] = ary[j];
            ary[j] = temp;
            i++;
            j--;
```

```
        }
    }

    if (left < j) {
        quickSort(ary, left, j);
    }
    if (i < right) {
        quickSort(ary, i, right);
    }
}

int testArray(int ary[], int size) {
    for (int i = 0; i < size - 1; i++) {
        if (ary[i] > ary[i + 1]) {
            return 0;
        }
    }

    return 1;
}
```