Statement of Problem:

The purpose of this program is to create a simple benchmarking program. The intent is to display how performance is affected by machine architecture, compilation settings, operating systems, and through how many instances run at once.

Workload

The floating point operation used in my program is matrix multiplication. The integer operation used is merge sort. Both operations utilize control structures, mathematical operations, and require shifting of data values. The stated operations are non-trivial and labor intensive and would take considerable time if done by hand.

The merge sort component works on an array of size 6000 that is initialized to a repeating sequence of the four values 0, 1, 2, 3.

The matrix multiplication is applied to matrices of dimensions 100x100

The workload was set so that both would achieve an approximately equal amount of operations per minute.

I wanted to utilize a matrix operation for floating point and a sorting algorithm for integer values. Additionally, I wanted to do something other than what was presented in class; most of my peers are likely doing matrix inversion and quick sort.

Program Structure

The program was created with ease of building, debugging, and reading in mind. There are three primary components of the program: CPUben, SortingOperations, and MatrixOperations. CPUben is used to explain the program, run the tests, and display the results. SortingOperations contains components for the merge sort. MatrixOperations contains components for the matrix operations.

The following entities were present in the source:

MatrixOperations - .h/.cpp

SortingOperation - .h/.cpp

CPUben.cpp

A stadfx.cpp and stadfx.h were also included, but those are defaults for Visual Studio projects and not exclusive to this project in particular.

The idea behind my organization was that MatrixOperations and SortingOperations would be able to work on their own and stand alone. All the materials needed to test, display, and work with the operations are available within the respective files. They also both have constants for size that can be easily changed towards the top of the header files for tuning. I know that my implementation is not the best for pure performance, but it yields consistent results that can be used for comparison between various machines and operating systems.

I do not run checks for correctness during the runtime of the program because I tested the methods as I was making them through files separate from the current benchmarking main program. I also did not account for many outside conditions, as the program's proper use does not depend on user input. I probably should have thrown a statement to check when allocating memory for the arrays, but with today's computers I feel as if it will likely not be a problem.

MatrixOperations:

A "Matrix" class is contained within to facilitate the operations that were performed.

Space is allocated for a square matrix of the dimensions defined by the SIZE constant.

The '+' , '-' , '*', and '<<' operators are overloaded to perform their given operations.

All of these operations are non-destructive and return a resulting Matrix object.

zero(): zeros out a matrix and another

initialize(): seeds the matrix with default values (2.00001 across the diagonal and 1.00001 elsewhere). This ensures a normal and diagonally-dominant matrix.

The size of matrix is indicated by a constant named SIZE and can be easily changed in the .h file.

-Initially, I wanted to try a 250x250 matrices but the size appeared to be too large and was subsequently reduced to 100x100.

The functions were verified to be working as implemented.

SortingOperations:

Memory is allocated for an array according to the SIZE constant on creation.

initializeArray(): initializes the array with: 0, 1, 2, 3, 0, 1, 2, 3 … until the array is filled. This distribution of values mimics random symmetric distribution without the downside of having non-identical arrays for the purpose of testing comparisons. This method can be repeated upon the same array multiple times and does not require the array to be deleted and remade.

commenceSort(): initializes the sort function

checkArray(): returns true if the array is sorted, false otherwise

sort(int[], int, int) is the main sorting method.

merge(int[], int, int) is the interior merge component of sort.

If more details about the implementation of merge sort and the matrix methods are desired my source code is appended to the end of this paper. The code is well-commented.

4.Measurements:

COMPILERS

I tried g++ with levels 0, 1, 2, 3, fast, and short as well as compiling with Visual Studio's debug and release features. The results are in a table below.

The differences between debug mode compiling and more performance-based compiling was absolutely astonishing. Compiling within Visual Studio had both the highest and lowest score, 356 for the release option and 84 for the debug option—the release version ran over 4.2 times faster.

Visual Studio Release:

```
================================================================================
This is a simple benchmark program utilizing double and integer operations.
================================================================================
Integer operations are checked using merge sort.
Double operations are checked using matrix multiplication.
10 seconds will be allocated to each type of operation.


Double Values:A total of 13927 operations were performed in 10 seconds.
Operations per minute: 83562
Operations per second: 1393


Integer Values:A total of 14530 operations were performed in 10 seconds.
Operations per minute: 87180
Operations per second: 1453

Harmonic mean: 1422
==================================================
Benchmark Score: 356
==================================================
Press any key to continue . . .
```

Visual Studio Compiler – DEBUG

```
================================================================================
This is a simple benchmark program utilizing double and integer operations.
================================================================================
Integer operations are checked using merge sort.
Double operations are checked using matrix multiplication.
10 seconds will be allocated to each type of operation.


Double Values:A total of 3688 operations were performed in 10 seconds.
Operations per minute: 22124
Operations per second: 369


Integer Values:A total of 3073 operations were performed in 10 seconds.
Operations per minute: 18434
Operations per second: 307

Harmonic mean: 335
==================================================
Benchmark Score: 84
==================================================
```

The default compiling setting for g++ (O0) was rather slow with a score of 97 as well, all of the versions above that level performed similarly; this was surprising to me seeing how much the performance varied with the debug and release versions. The compilations for g++ 01,02,03, fast, and short all scored around 250. This is still significantly slower than the Visual Studio release build, which was an unexpected result. I anticipated that the g++ fast build would perform similarly to the Visual Studio release, but it is not even close for overall performance. The g++ builds, however varied immensely on the speed for floating point or integer values. g++ O3 and g++ Ofast scored extremely high on the double operations, around 2460 operations per second, but performed terribly on the integer operations where they only got 620 operations per second. Currently, I do not understand why this would be the case; maybe it is due to my code being poorly written for performance, or perhaps there is some trade-off that will occur no matter how immaculate the code is. It is something that I will look into.

g++ default

```
================================================================================
This is a simple benchmark program utilizing double and integer operations.
================================================================================
Integer operations are checked using merge sort.
Double operations are checked using matrix multiplication.
10 seconds will be allocated to each type of operation.


Double Values:A total of 2729 operations were performed in 10 seconds.
Operations per minute: 16374
Operations per second: 273


Integer Values:A total of 6616 operations were performed in 10 seconds.
Operations per minute: 39696
Operations per second: 662

Harmonic mean: 386
================================================
Benchmark Score: 97
================================================
```

g++ -O1

```
$ ./CPUben01
================================================================================
========
This is a simple benchmark program utilizing double and integer operations.
================================================================================
========
Integer operations are checked using merge sort.
Double operations are checked using matrix multiplication.
10 seconds will be allocated to each type of operation.


Double Values:A total of 13790 operations were performed in 10 seconds.
Operations per minute: 82740
Operations per second: 1379


Integer Values:A total of 8038 operations were performed in 10 seconds.
Operations per minute: 48228
Operations per second: 804

Harmonic mean: 1016
================================================
Benchmark Score: 254
================================================
```

g++ -O2

```
$ ./CPUben2
================================================================================
This is a simple benchmark program utilizing double and integer operations.
================================================================================
Integer operations are checked using merge sort.
Double operations are checked using matrix multiplication.
10 seconds will be allocated to each type of operation.


Double Values:A total of 13454 operations were performed in 10 seconds.
Operations per minute: 80724
Operations per second: 1345


Integer Values:A total of 8083 operations were performed in 10 seconds.
Operations per minute: 48498
Operations per second: 808

Harmonic mean: 1010
================================================
Benchmark Score: 252
================================================
```

- Note the Extreme Values For Double/Integer Operations in the 3 and fast builds

g++ -O3

```
Ficha@DESKTOP-5TO6GEN ~/Benchmark Source
$ ./CPUben3
================================================================================
This is a simple benchmark program utilizing double and integer operations.
================================================================================
Integer operations are checked using merge sort.
Double operations are checked using matrix multiplication.
10 seconds will be allocated to each type of operation.


Double Values:A total of 24599 operations were performed in 10 seconds.
Operations per minute: 147594
Operations per second: 2460


Integer Values:A total of 6192 operations were performed in 10 seconds.
Operations per minute: 37152
Operations per second: 619

Harmonic mean: 989
================================================
Benchmark Score: 247
================================================
```

g++ -Ofast

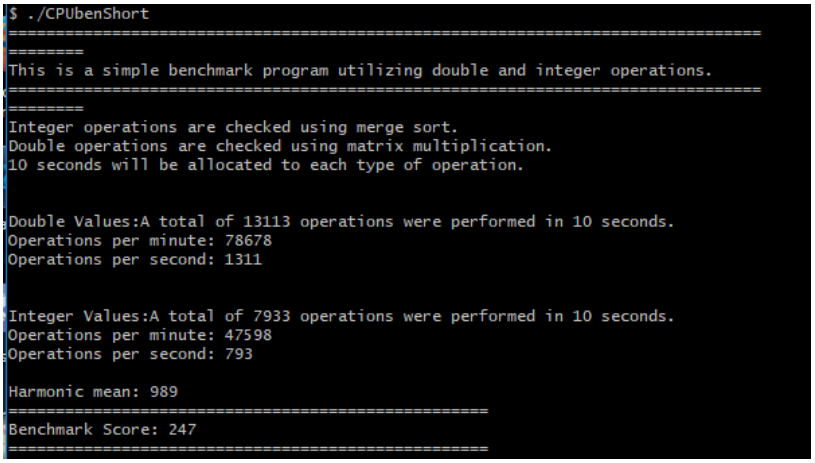
```
$ ./CPUbenFast
================================================================================
========
This is a simple benchmark program utilizing double and integer operations.
================================================================================
========
Integer operations are checked using merge sort.
Double operations are checked using matrix multiplication.
10 seconds will be allocated to each type of operation.


Double Values:A total of 24716 operations were performed in 10 seconds.
Operations per minute: 148296
Operations per second: 2472


Integer Values:A total of 6197 operations were performed in 10 seconds.
Operations per minute: 37182
Operations per second: 620

Harmonic mean: 991
================================================
Benchmark Score: 248
================================================
```

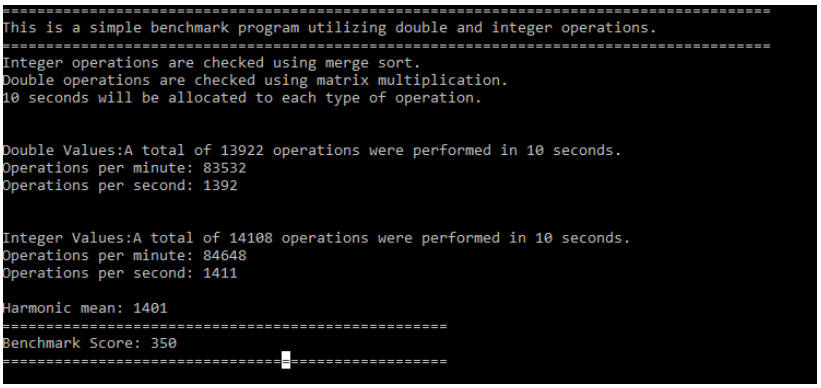
g++ -Oshort



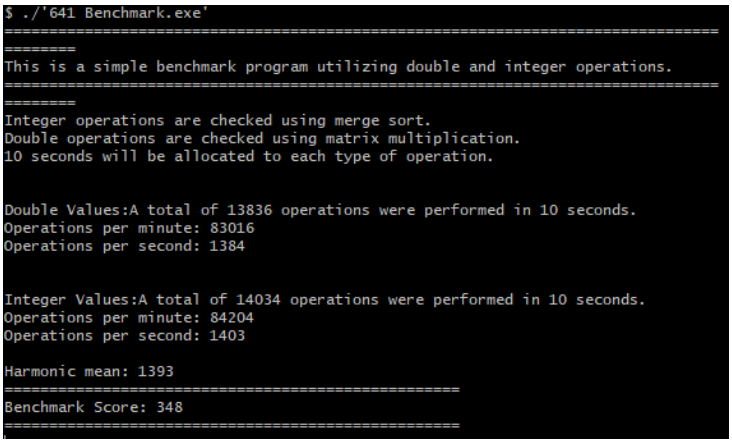
OS

I ran the program normally through Windows and through Cygwin. Both performed equivalently.

Windows:



Cygwin:



I was surprised by these results, as I would assume that a virtual machine would run slower than the primary OS, but this was not the case. The instances run through Cygwin were consistently slower, but

only slightly. The slowdown was generally between .5% and 2%. None of my computers dual boot currently so I decided on using Cygwin.

COMPUTERS

Intel Core i7-7700K : 4 Cores, (8 logical processors), 4.2GHz

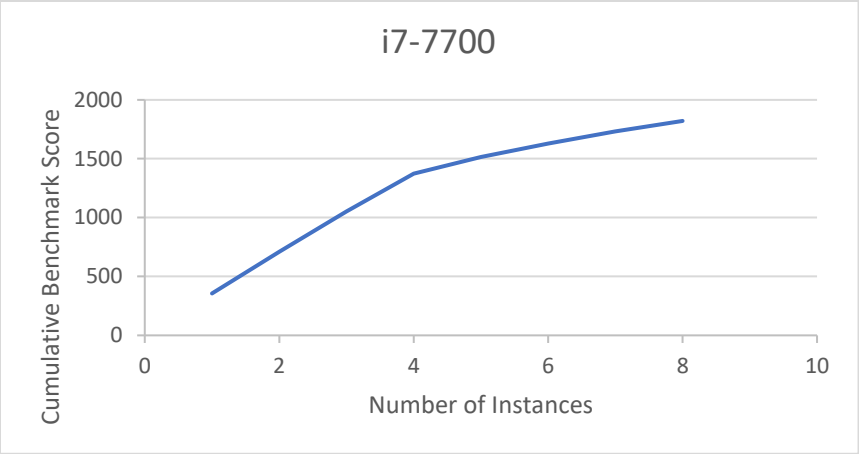
1	2	3	4	5	6	7	8	SUM
356								**356**
355	356							**711**
351	351	352						**1054**
346	342	342	344					**1374**
291	291	294	328	310				**1514**
256	284	257	260	272	300			**1629**
260	243	238	238	257	253	243		**1732**
228	228	228	227	228	227	228	227	**1821**

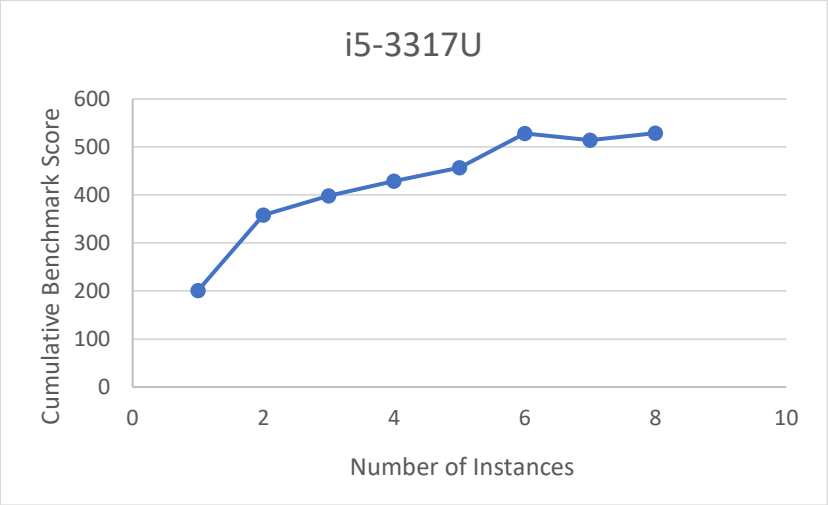
Intel Core i5-3317U: 2 Cores (4 logical processors), 1.7 GHz

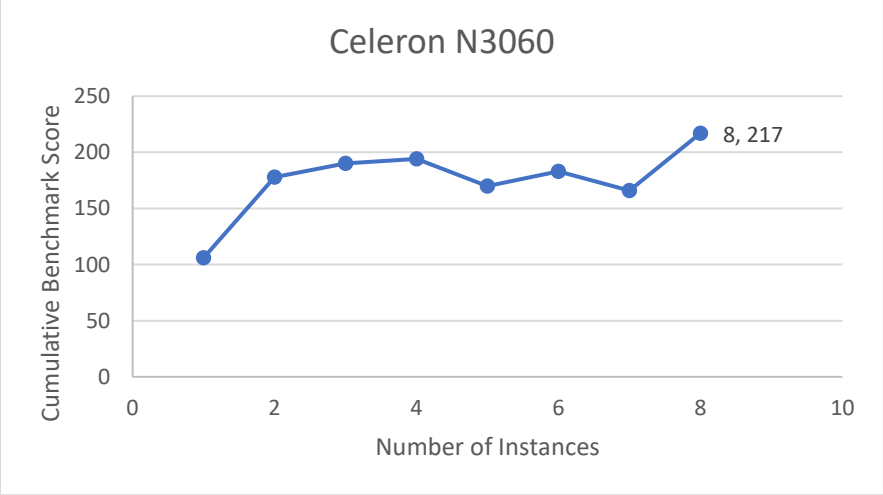
1	2	3	4	5	6	7	8	SUM
201								**201**
179	179							**358**
132	132	134						**398**
107	107	107	108					**429**
98	86	98	88	87				**457**
94	116	79	81	79	79			**528**
72	88	68	80	69	69	68		**514**
66	97	68	61	59	60	60	58	**529**

Intel Celeron N3060 : 2 cores, 1.6GHz

1	2	3	4	5	6	7	8	SUM
106								**106**
102	76							**178**
91	50	49						**190**
90	47	47	104					**288**
83	38	37	36					**194**
88	22	20	20	20				**170**
86	22	19	18	18	20			**183**
73	21	15	14	13	15	15		**166**
89	26	21	16	16	16	16	17	**217**


i7-7700
Cumulative Benchmark Score
2000
1500
1000
500
0
0
2
4
6
8
10
Number of Instances


i5-3317U
Cumulative Benchmark Score
600
500
400
300
200
100
0
0
2
4
6
8
10
Number of Instances


Celeron N3060
Cumulative Benchmark Score
250
200
150
100
50
0
8, 217
0
2
4
6
8
10
Number of Instances

My home desktop computer with the i7-7700 performed significantly better than the other computers tested. The sum of the benchmark values increased continued increasing all the way up to 8 cores, due to the 8 logical cores. For the first 4 instances, there was little to no loss of the program performance. Above this point the benchmark sum continued increased, but performance per instance slowed down, spreading the workload relatively evenly. The behavior was as expected. Its sum with 8 instances running was 1821.

The 2 core i5 had its sum increase quickly to the number of cores (2) then gradually increase to 6 instances before leveling off. I was surprised that it leveled off at 7 instances instead of at 4, as it only has 4 logical processes. The sum for this was 529.

The Celeron N3060 is a terrible computer, it is one of the worst computes I have had the displeasure of owning. It maxes out on computing power with one tab of Firefox open or with PowerPoint. This computer did increase up to t2 cores, but after it leveled off for the most part having a few ups and downs. It is an inconsistent machine and I am not surprised by the dips in the graph. The load distribution on the computer is terribly managed, the first instance seems to stay around 90 while the rest of them get spread relatively evenly.

When I made the benchmark program I made the number corresponding with my desktop, around the 350 range. I had expected the i5 to score around 200, which it did. This is what I would consider to be a lower end computer in modern times; 2 cores @ 1.7 GHz. I did not, however anticipate how terribly the N3060 performed. If I would have known that computers would score that low I would have set the benchmark on my PC to be around the 700 range instead of 350, as slight changes in numbers when an instance is only scoring a 16 is huge. The gap between the two computers was absolutely amazing, the i7 achieved a cumulative score with 8 processes that was almost 9 times higher.

5.Conclusions

The architecture of a computer matters more than anything else when it comes to performance. The number of cores and logical cores behaves as one would think when running simultaneous processes.

Operating systems have been refined and improved for many years. The three major operating systems used: Linux, Windows, and Mac all perform at a rather similar level.

The compiler makes a significant difference on performance. How it is compiled can affect different operations as well. Compiling by the debug option makes compile time quicker, but negatively affects performance on a large level. Compiling can optimize code for different purposes: one way may be well-rounded while another makes floating operations faster, but integer operations slower. If a program was to be released to the public time should be put into trying various compiling options to optimize the program for its particular purpose.

Discussion / Additions

I have been compiling code with default options most of the time but after seeing the degree of difference I will never do that again. The difference should have been displayed to students in intro level courses, instead of being introduced in my Senior year. This would have made a world of difference on some programs that I had to make in the past.

There was something I found by accident that significantly affected performance within my merge sort method. I define an array in one of my methods after compile time. I let this have a memory leak without deleting it and got results that were significantly faster by leaving out a delete operation, I thought this was interesting so I wanted to make a note of it. I was thinking about assigning a heap of memory for use of a set size during compile time, but the size was decided upon by trial (manipulating size to make it close to the same operations/second as the matrix multiplication score) and I was unsure of how much memory was needed at the time of writing. The operation would be significantly faster if clean up only had to be done upon termination.

This was an interesting project, the methods themselves were not incredibly difficult but the testing, data collection, and write up took quite a while.

I would like brutal feedback telling me everything wrong with my code and how to improve upon it, I know that I have a lot to learn. I would have loved to come in during office hours and asked there, but I waited until the final weekend to start this project.

CSC 641

September 18, 2017

Benchmark Analysis

Richard Robinson

Professor: Jozo Dujmovic