Study of Algorithm Design Comparison between C, C++, Java and C# Programming Languages.

GURJOT KAUR
Research Scholar, Manav Bharti University, H.P., India

Abstract: This document compares the performance of various algorithms across various programming languages, namely, C, C++, Java and C# (C-Sharp). Using standard algorithms implemented in each language, we will compare the performance of the resulting executables from each language. Because the C# language only compiles and runs on Windows based operating systems, all performance tests will be run on a Windows Server 2003 based system.

-------------------------♦-------------------------

1. INTRODUCTION

One of the first decisions that needs to be made when embarking on a new software project is what programming language to use for development. There are many choices. C, C++, Java, C#, Pascal, Fortran, etc. There are many criteria that a developer can use to narrow down the choices somewhat. However, criteria that always seems to be high on the list is application performance. Performance is and should always be amongst the highest, if not the absolute highest, of criteria used to determine what language an application should be written in.
What that in mind, we will write a number of algorithms in various programming languages to see which language offers the best performance. All source code was written in house, and all performance tests were run on computers here in our labs.
The source code that lies within is not intended for commercialization nor for production environments. But is simply to gather timing information when developing a specific algorithm in different programming languages and seeing what language can yield the fastest time. No more, no less.

1.1 What Are We Testing?

What we're trying to test in each language is, at a very basic level, how fast can each language execute the code for various constructs such as:

- if statements
- while loops
- do loops
- for loops
- array accesses (reading and writing)
- mathematical statements (integer and floating point)
- Logical operations (and'ing, or'ing)

We'll test these basic constructs in a way that will give the reader a way of coming to terms with what to expect, performance wise, from each language. The way that we'll do this is by implementing various well known algorithms such that we can test all of the various performance aspects of the languages that will be benchmarked whilst accomplishing something useful.

1.2 How Are We Testing: Algorithms Utilized

The algorithms we will benchmark across the various languages will be the following:

- Bubble Sort
- Insertion Sort
- Fletcher 32 bit CRC Checksum
- Run Length Encoding
- Prime Number Generation (Floating Point and Integer Operations)
- Square Root

All we're looking for in these tests is how long does it take for the respective languages to execute a particular algorithm given a constant work load. The workload will be the same regardless of what language the algorithm is written in. Some of these algorithms are heavy on string processing, array processing, or math processing. Other areas that can be covered will be covered in future algorithms that we add to the benchmark suite and document.

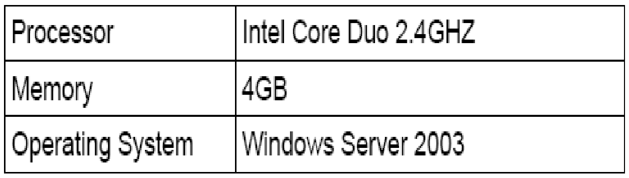
These benchmarks are meant to give an indication as to how long it takes each respective language toexecute the given set of instructions. For instance, the Bubble Sort algorithm does a lot of array processing. What you should get from this is not the fact that your application doesn't make use of the Bubble Sort algorithm, but rather, if your application does a lot of array processing, then that portion of your code within your application may fair the same if extracted and benchmarked against the same code written in a different programming language. That's all.

For instance, when you look at the algorithm for the Bubble Sort algorithm, for example, don't look at it and think that your applications doesn't implement the Bubble Sort algorithm, think of it as a function that does a for loop with some compare statements, array reads and array writes. The Bubble Sort algorithm is a great example of a piece of code that does these operations. So using this algorithm to measure the performance of these operations across multiple languages is valid.

1.3 Timing Graphs

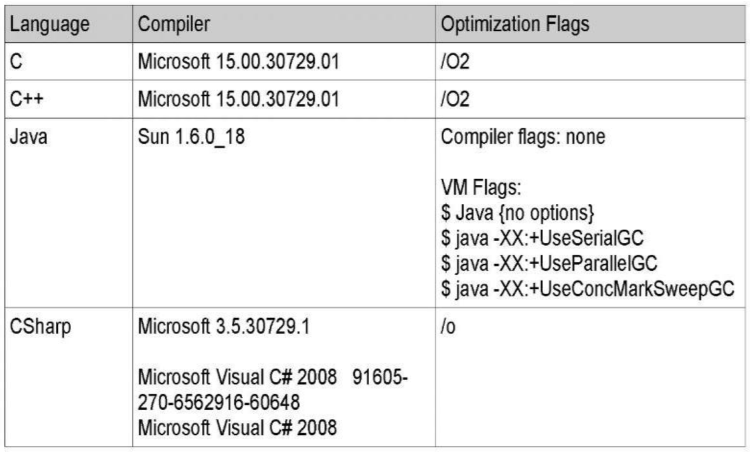
All graphs present the data in the amount of seconds it took to execute the workload. Shorter times are better, except where noted.

1.4 Testing Hardware and Software



Processor	Intel Core Duo 2.4GHZ
Memory	4GB
Operating System	Windows Server 2003

1.5 Developer Tools and Optimizations *Compilers and Optimization Flags*



Language	Compiler	Optimization Flags
C	Microsoft 15.00.30729.01	/O2
C++	Microsoft 15.00.30729.01	/O2

Java	Sun 1.6.0_18	Compiler flags: none
		
		VM Flags:
		$ Java {no options}
		$ java -XX:+UseSerialGC
		$ java -XX:+UseParallelGC
		$ java -XX:+UseConcMarkSweepGC

CSharp	Microsoft 3.5.30729.1 Microsoft Visual C# 2008 91605-270-6562916-60648 Microsoft Visual C# 2008	/o

For the Java tests, we ran individual tests with all of the listed optimizations and took the best times for the graphs. Also, for Java, we 'warmed up' the algorithms by calling them a number of times before timing started to give the JIT compiler a chance to possibly optimize them.

1.6 Future Work

This document will be updated as frequently as time permits with new algorithms, fixes and performance suggestions from readers.

1.7 Source Code Package: Algorithmic.zip

The source code for all benchmarked algorithms is available from the Cherrystone web site in the Documentation and White Papers section. Feel free to download and try them yourself. If you can get any of the programs to run faster, please let us know and we'll make the appropriate changes to our algorithm benchmark package and document. Send any comments/changes to feedback@cherrystonesoftware.com.

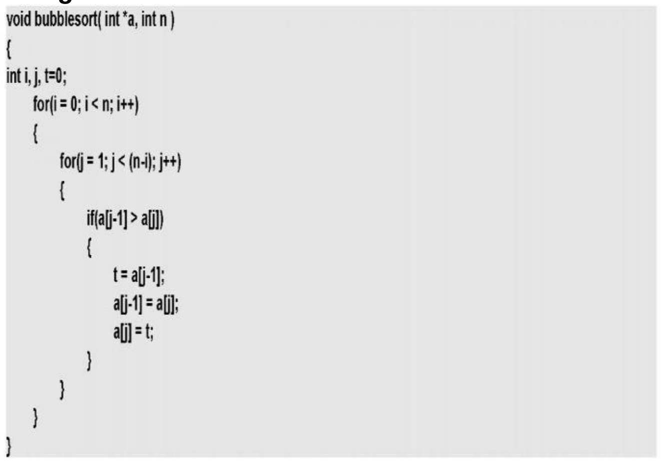
2 Bubble Sort Algorithm

The Bubble Sort algorithm is a sorting algorithm that incrementally steps through an array comparing each adjacent element and doing a swap if necessary. It is one of the slowest known sorting algorithms, so it was a great choice to use in this CPU intensive benchmark because basically what we're timing then is a nested loop comparing and swapping adjacent integer values from within an array.

2.1 Workload

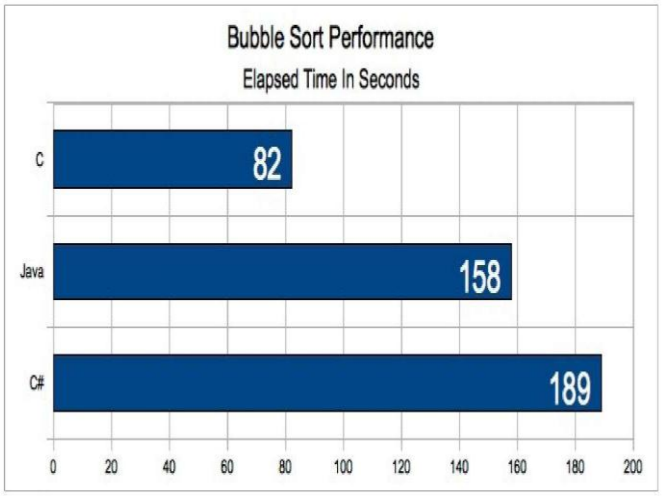
The workload that was given to each language to complete was to sort a 300,000 element array of integers. The initial array was setup such that every element would need to be swapped, The values in the array were initialized in descending order starting at 300000 and decreased in value to 0, The array was then sorted in ascending order. This would be considered a worst case scenario. The function called to sort the array is called exactly once.

2.2 Algorithm



```
void bubblesort( int *a, int n )
{
int i, j, t=0;
    for(i = 0; i < n; i++)
    {
        for(j = 1; j < (n-i); j++)
        {
            if(a[j-1] > a[j])
            {
                t = a[j-1];
                a[j-1] = a[j];
                a[j] = t;
            }
        }
    }
}
```

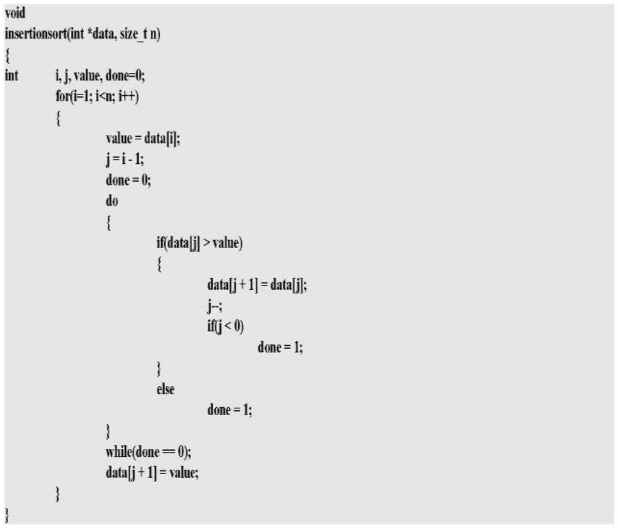
2.3 Performance Graph



3 Insertion Sort

Insertion sort is a simple sorting algorithm, comparison sort in which the sorted array (or list) is built one entry at a time. It is much less efficient on large lists than more advanced algorithms.

3.1 Algorithm

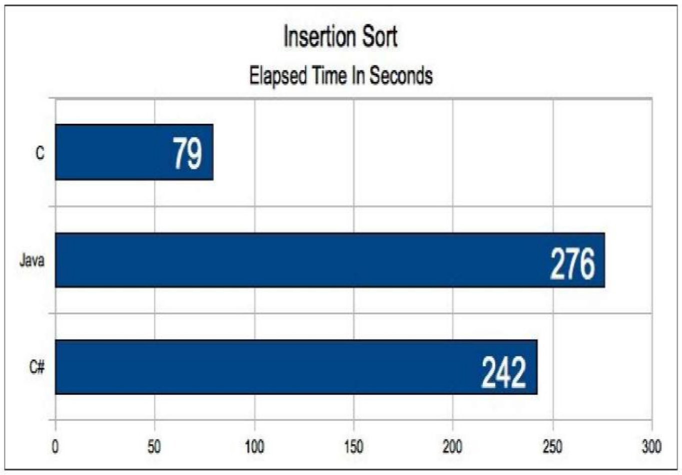


```
void
insertionsort(int *data, size_t n)
{
int         i, j, value, done=0;
            for(i=1; i<n; i++)
            {
                        value = data[i];
                        j = i - 1;
                        done = 0;
                        do
                        {
                                    if(data[j] > value)
                                    {
                                                data[j + 1] = data[j];
                                                j--;
                                                if(j < 0)
                                                            done = 1;
                                    }
                                    else
                                                done = 1;
                        }
                        while(done == 0);
                        data[j + 1] = value;
            }
}
```

3.2 Workload

The work load is much like our bubble sort algorithm in that there is an initial array that is sorted in descending order and the task is to sort it in ascending order. The array size is 300,000 integer elements. The function insertionsort() is called twice to sort the array.

3.3 Performance Graph



4 Prime Number Generation

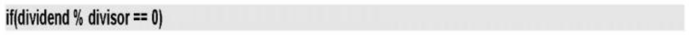
A prime number is defined as any number that can only be divided evenly by 1 and itself. The number of prime numbers is infinite. Therefore, writing a program to calculate prime numbers should have an upper limit. Here's a list of the first 20 primes:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 ...

There are several way to detect primality from within a computer application.

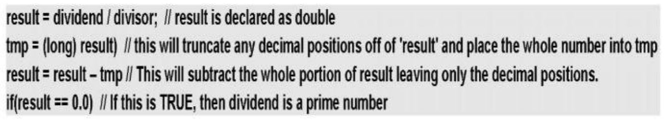
1.4 Integer Math
1.5 Floating Point Math

With integer math, you can see a division statement yields a remainder as follows:



```
if(dividend % divisor == 0)
```

if this yields TRUE, then the dividend is NOT a prime number. Using Floating Point arithmetic, you can do the following:



```
result = dividend / divisor;  // result is declared as double
tmp = (long) result)  // this will truncate any decimal positions off of 'result' and place the whole number into tmp
result = result – tmp // This will subtract the whole portion of result leaving only the decimal positions.
if(result == 0.0)  // If this is TRUE, then dividend is a prime number
```

We will utilize both methods to compute primes in separate performance test to show the speed difference between the 2 methods.

4.1 Prime Numbers - Floating Point Variables *Workload*

This performance test will calculate the first 2,500,000 prime numbers. The function that will generate the prime numbers will be called once and all prime numbers will be returned in an array.

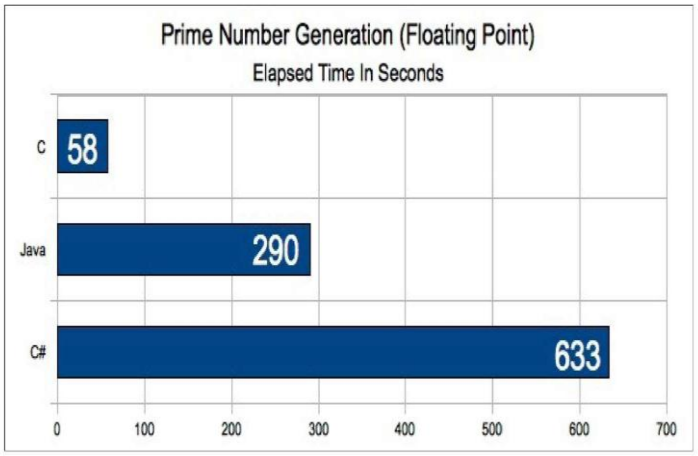
Algorithm

```
long*
computeprimes(int n)
{
double          divisor, result, squareroot;
long            *primes=NULL, i, half, tmp;
int             nprimes=0;
        primes = malloc(sizeof(double) * n);
        if(primes == NULL)
                return NULL;
        primes[nprimes++] = 2;
        for(i=3; nprimes<n; i+=2)
        {
        int     prime;
                squareroot = sqrt(i);
                divisor = 3;
                prime = 1;

                while(prime && divisor <= squareroot)
                {
                        result = i / divisor;
                        tmp = (long) result;
                        result -= tmp;
                        if(result == 0.00)
                                prime = 0;

                        divisor += 2.00;
                }
                if(prime)
                        primes[nprimes++] = i;
        }
        return primes;
}
```

Performance Graph



4.2 Prime Numbers - Integer Variables *Workload*

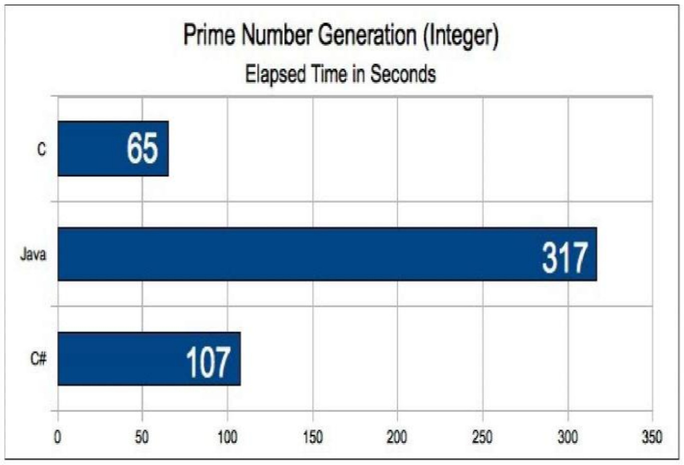
Same work load as in the Floating Point Math performance test

Algorithm

```
long*
computeprimes(int n)
{
long    divisor, result, squareroot;
long    *primes=NULL, i, half, tmp;
int     nprimes=0;
        primes = malloc(sizeof(double) * n);
        if(primes == NULL)
                return NULL;
        primes[nprimes++] = 2;
        for(i=3; nprimes<n; i+=2)
        {
        int     prime;
                squareroot = sqrt(i);

                divisor = 3;
                prime = 1;
                while(prime && divisor <= squareroot)
                {
                        if(i % divisor == 0)
                                prime = 0;
                        divisor += 2.00;
                }
                if(prime)
                        primes[nprimes++] = i;
        }
        return primes;
}
```

Performance Graph



5 Conclusions

Draw your own conclusions. We're just here to develop the algorithms and present the performance data.
What we tried to show here is what language performs

best on a number of different algorithms that use different language constructs such as arrays, strings, mathematical operations (floating point and integer). Things that all applications do, no matter the size.
Hopefully you found this document informative.

6. Bibliography

- "The Java Tutorials: Passing Information to a Method or a Constructor". Oracle. Retrieved 2010-12-07.
- Java and C++ Library a b Robert C. Martin (January 1997). "Java vs. C++: A Critical Comparison" (PDF).
- "Reference Types and Values". The Java Language Specification, Third Edition. Retrieved 9 December 2010.
- Deitel, Paul; Deitel, Harvey (2009). Java for Programmers. Prentice Hall. p. 223. ISBN 978-0-13-700129-3. "Unlike some other languages, Java does not allow programmers to choose pass-by-value or pass-by-reference—all arguments are passed by value. A method call can pass two types of values to a method—copies of primitive values (e.g., values of type int and double) and copies of references to objects (including references to arrays). Objects themselves cannot be passed to methods."
- "Java Language Specification 4.3.1: Objects". Sun Microsystems. Retrieved 2010-12-09.
- "Java memory leaks -- Catch me if you can" by Satish Chandra Gupta, Rajeev Palanki, IBM DeveloperWorks, 16 Aug 2005
- Boost type traits library Clark, Nathan; Amir Hormati, Sami Yehia, Scott Mahlke (2007). "Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping". HPCA'07: 216–227.