Algorithmic Performance Comparison Between C, C++, Java and C# Programming Languages

Cherrystone Software Labs
http://www.cherrystonesoftware.com

Version 1.6
March 29th 2010

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.

The latest version of this document is available at the following URL:
http://www.cherrystonesoftware.com/doc/AlgorithmicPerformance.pdf

Table of Contents

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)

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 to execute 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.

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.

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.

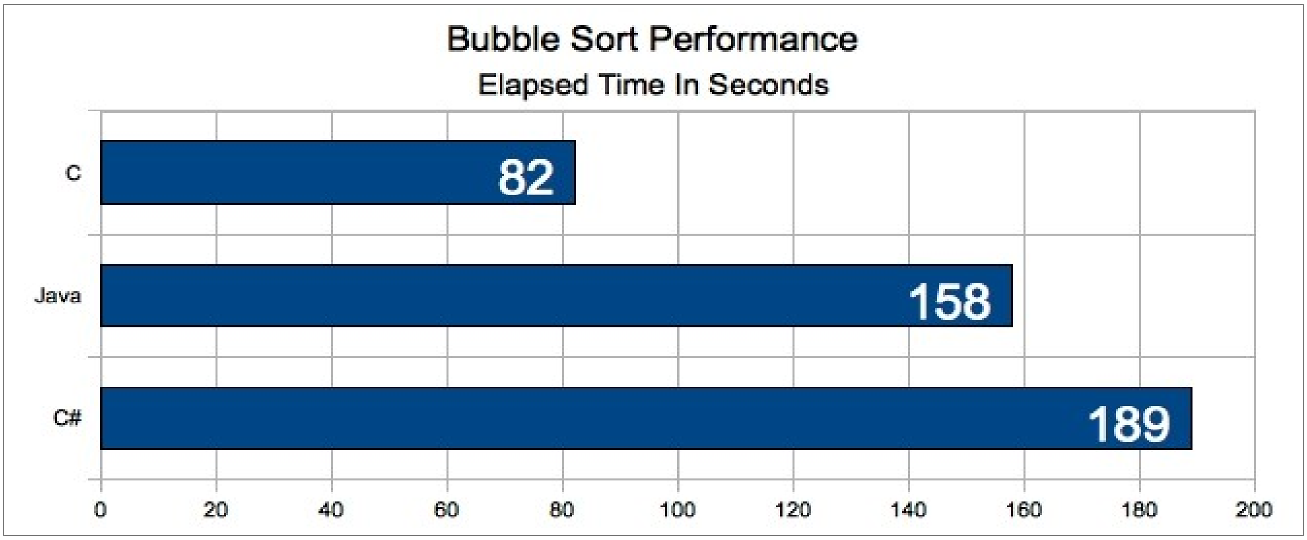
2.1 Workload

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.

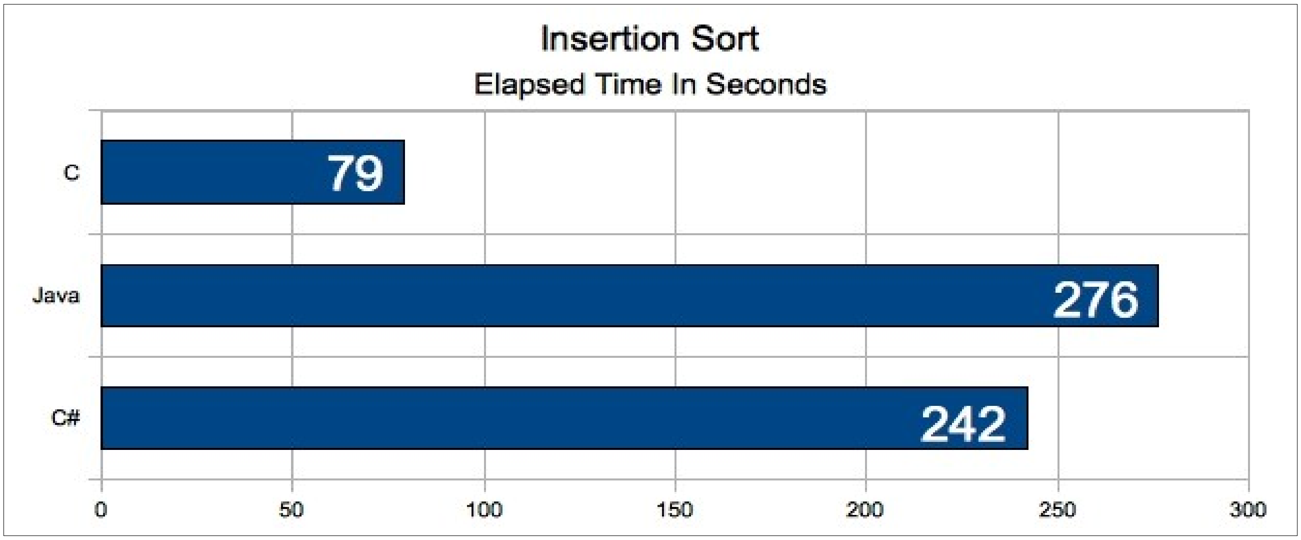
3.1 Algorithm

```
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 Fletcher 32bit CRC

This algorithm produces a 32bit CRC checksum value for the given input. The input must be 16 bit values as the fletcher algorithm does not work on 32 bit values.

4.1 Workload

The workload that the algorithms were tasked to complete were to produce a checksum from an array of length 500,000 short (16 bit) values. We call fletcher32() to produce a checksum on a byte array of data. We do this 1,000,000 times with a different set of data each time.

4.2 Algorithm

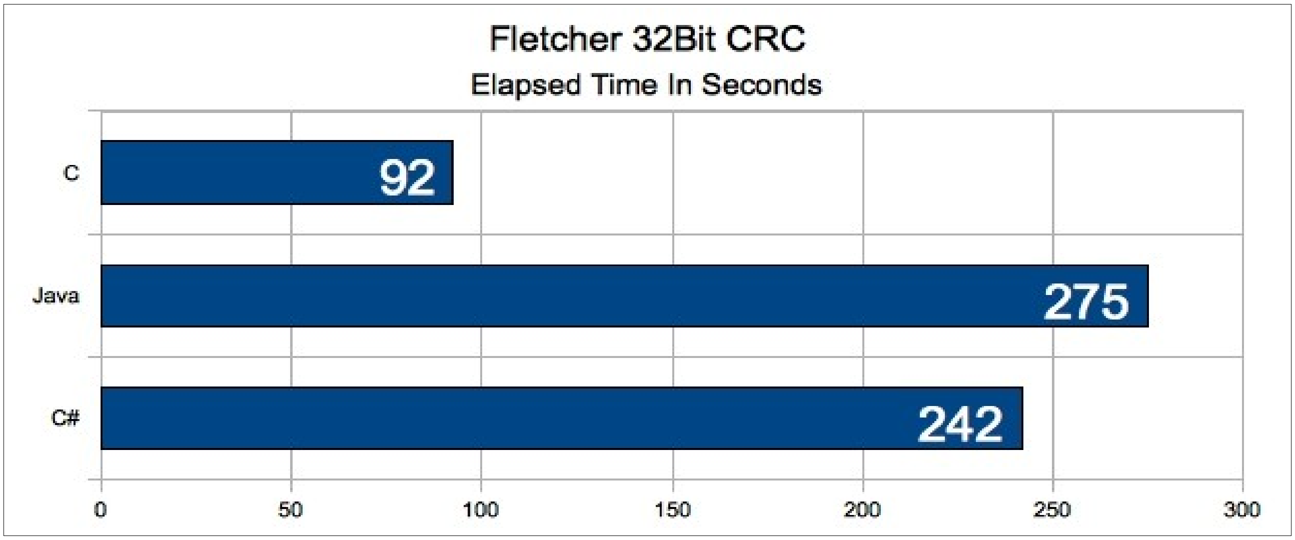
```
int
fletcher32(short *data, size_t len)
{
int         sum1 = 0xffff, sum2 = 0xffff;
int         i=0;

    while (len != 0)
        {
        int         tlen = len > 360 ? 360 : len;
                    len -= tlen;
                    do
                    {
                            sum1 += *data++;
                            sum2 += sum1;
                    } while ((--tlen) != 0);
        sum1 = (sum1 & 0xffff) + (sum1 >> 16);
        sum2 = (sum2 & 0xffff) + (sum2 >> 16);
    }
    sum1 = (sum1 & 0xffff) + (sum1 >> 16);
    sum2 = (sum2 & 0xffff) + (sum2 >> 16);
    return sum2 << 16 | sum1;
}
```

The fletcher32() function is called each time with a different set of data to compute a checksum. We get our data from a 1,000,000 integer array filled with random numbers.

Before each call to fletcher32(), we fill in our array with some of these random numbers starting from a random location with the random number array. Here's a general outline of what happens during setup time:

```
random = 1,000,000 randomly generated numbers;
start = get current time;
count = 0;
do
        random_index = random() % 1,000,000
        data = copy 500,000 random numbers from the random number array starting from random_index;
        crc = fletcher32(data, 500,000);
        count++;
until count == 50,000;
endtime = get current time;
print out elapsed time
```

4.3 Performance Graph


Fletcher 32Bit CRC
Elapsed Time In Seconds
C
Java
C#
92
275
242
0
50
100
150
200
250
300

5 Run Length Encoding and Decoding

Run length Encoding/Decoding is a basic computer science algorithm used for compression. It's CPU bound like all of the other algorithms tested so far, so it's a good algorithm to use for testing performance of a programming language.

The first thing we need to do was make sure that the data each algorithm worked on was identical between all languages. Two ways to do this. Either generate a large data file and have each language read from the data file, or generate a string dynamically such that the string is identical in each language. We chose the latter. So we have a function called ***mkstr***() that returns the same string for all languages, although it's written a little differently for each language. Returns a char* in C and a string in C++, and a String class in Java and CSharp.

```
#define   RLEINIT "340899012923238756387902375786489357893938523808573148915349831984539134751058"
#define   RALPHA              "abcdefghijklmnopqrstuvwxyz"

char*
mkstr(int elements)
{
char      *p=NULL, *q=NULL, *pinit, *alpha;
int       i, j, count=0, subcount, letter=0, buflen=0, ninit = 0, initlen, nalpha;
          nalpha = strlen(RALPHA);
          buflen = elements;
          initlen = strlen(RLEINIT);
          alpha = RALPHA;
          p = q = calloc(1, buflen + 1);
          if(p == NULL)
                    return p;
          pinit = RLEINIT;
          while(count < elements && pinit)
          {
                    subcount = pinit[ninit++ % initlen] - '0';
                    if(subcount == 0)
                              subcount += pinit[ninit++ % initlen] - '0' + 10;
                    for(j=0; j<subcount && (count + j < buflen); j++)
                              *q++ = alpha[letter % nalpha];
                    letter++;
                    count += subcount;
          }
          return p;
}
```

As a small example of the string returned from the ***mkstr***() function will look something like the following in each language:

aaabbbbccccccccccccccccccddddddddd …..

This string derives from the 2 constants, RLEINIT and RALPHA. The first character in RLEINIT is '3', and the first character in RALPHA is an 'a', so there will be 3 a's in our string. The next character is RLEINIT is '4', and the next character in RALPHA is 'b', so 4 b's will be appended to the string. Now, the next character in RLEALPHA is '0'. When we hit a zero, we will then go and read the next number and multiply that by 10, which means that anytime we hit a '0', we'll be treating it as a '1'. So will then append 18 c's to our string. Etc Etc … When we come to the end of either string, we wrap around and star reusing the characters from the beginning. So basically, the RLEINIT string is used as probably the dumbest random number generator on the planet. But it does serve it's purpose, and that is to make sure that each language has the exact same input data so that we can more be assured that any differences in the timing data are not because of a different set of input data. This is extremely important.

The encoding of this input string would look something like this:

3a4b18c9d ….

That is how we get compression of the data, to find a sequence of like characters and compress them by indicating how many time a character repeats by placing a repeat character in front of it.

The decoding of the string will yield the original string.

So we generate a constant input string, and each language has it's own ***encode***() and ***decode***() functions that are timed in how long it takes to encode the string plus how long it takes to decode it back to it's original state. One twist on what we decided to do here from previous performance tests is break out the C and C++ algorithms to compare them against each other. So in this performance test there will be 4 languages in the timing charts, C, C++, Java and CSharp. What this comes down to is the C language using dynamic memory and pointers to encode and decode, and C++, Java and CSharp using their respective String classes.

5.1 Workload

The workload will be a 200,000 byte input string. The task will be to encode and then decode the string 10,000 times. The input string is the same on each run.

5.2 Algorithm

We'll not display all 4 source functions for our 4 languages of C, C++, Java and CSharp. We'll display the C++ code here, and the C, Java and CSharp will be in Appendix A at the back of this document.

C++ encode Function

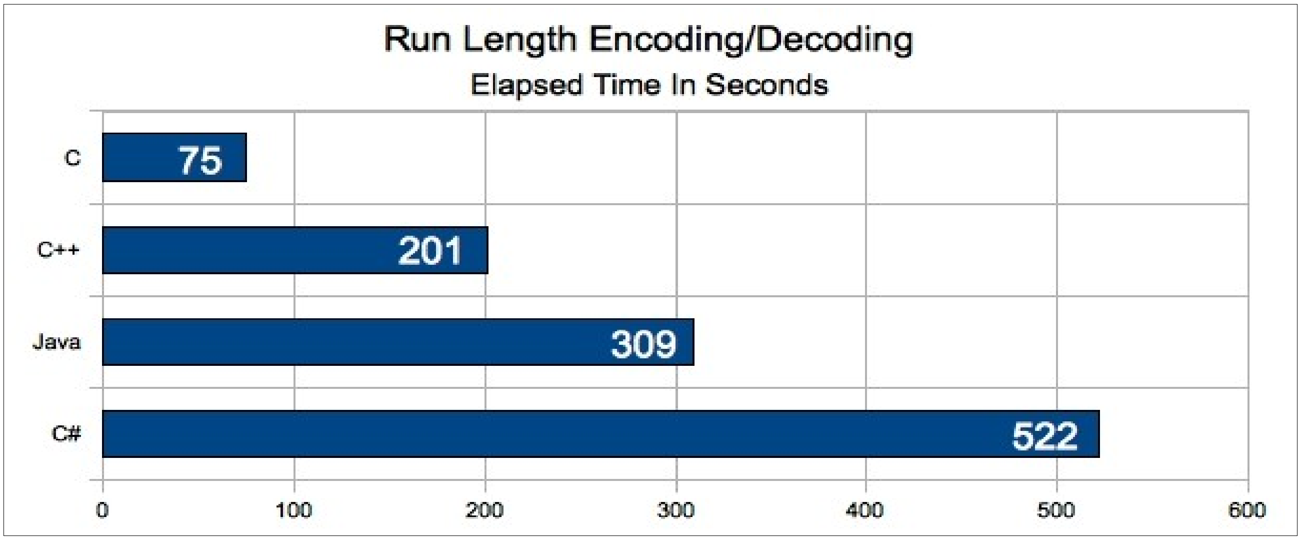
```
string*
encode(string *source, int n)
{
char    c, cur=' ';
int     i, runlength=1;
string  *dest;


    dest = new string();

    for(i = 0; i<n; i++)
    {
        c = source->at(i);

        if(c != cur)
        {
            *dest += runlength;
            *dest += c;

            runlength=1;
        }
        else runlength++;
    }

    return dest;
}
```

C++ decode Function

```
string*
decode(string *source)
{
```

```
char    c;
int     count, i, n=0, buflen, index=0;
string  *s, *dest;

        buflen = (int) source->length();
        dest = new string();
        while(index < source->length())
        {
                count = 1;
                c = source->at(index);
                if(c >= '0' && c <= '9')
                {
                int     m=1, val;
                        count = 0;
                        do
                        {
                                val = c - '0';
                                count = (count * m) + val;
                                m *= 10;
                                index++;
                                c = source->at(index);
                        } while(c >= '0' && c <= '9');
                }
                n += count;
                for(i=0; i<count; i++)
                        *dest += c;
                index++;
        }
        return dest;
}
```

5.3 Performance Graph



6 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. Integer Math
2. 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.

6.1 Prime Numbers - Floating Point Variables

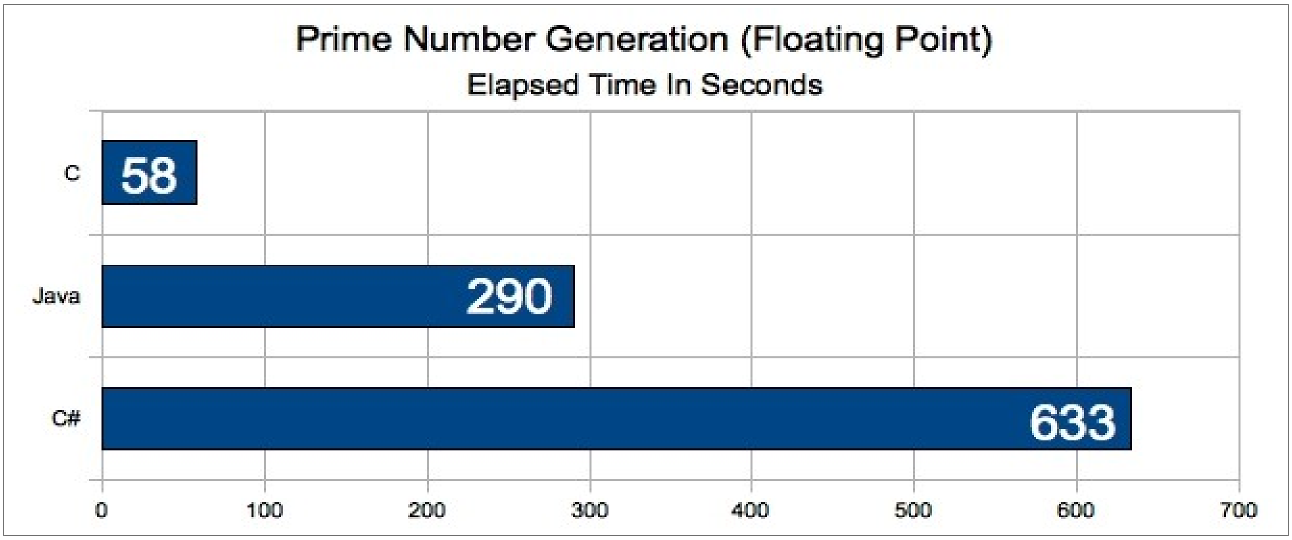
Workload

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



6.2 Prime Numbers - Integer Variables

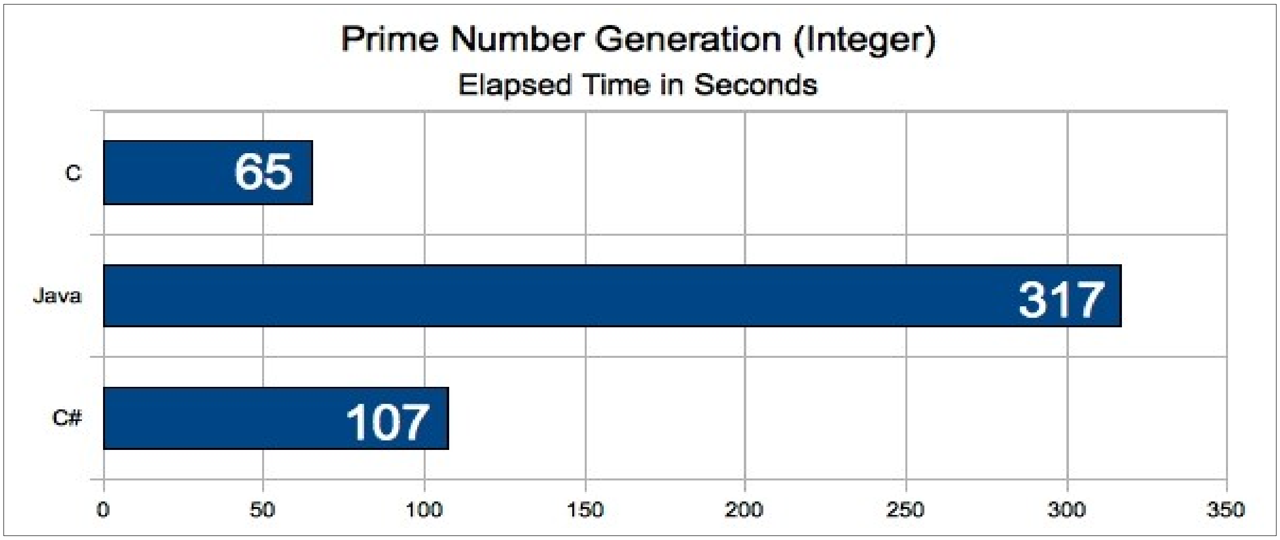
Workload

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



7 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.

7.1 Feedback

We'd love to hear your feedback from this performance white paper. Send any comments (constructive or deconstructive), flames, flowers and anything in between to the following email address:

feedback@cherrystonesoftware.com

We guarantee that whatever you send will be read, but we can't guarantee any response depending on how busy we are at that point in time.

If you have suggestions on how to increase the performance of any of these algorithms, please let us know. Please send along the source for the particular algorithm. If you have an algorithm that you'd like to see benchmarked in this document, let us know.