Performance comparison of Java and C++
when sorting integers and writing/reading files.

Suraj Sharma
This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Bachelor of Science in Computer Sciences. The thesis is equivalent to 20 weeks of full-time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:
Author(s):
Suraj Sharma
E-mail: sush15@student.bth.se

University advisor:
Dr. Prashant Goswami
Department of Creative Technologies

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57
Abstract

This study is conducted to show the strengths and weaknesses of C++ and Java in three areas that are used often in programming: loading, sorting and saving data. Performance and scalability are large factors in software development and choosing the right programming language is often a long process. It is important to conduct these types of direct comparison studies to properly identify strengths and weaknesses of programming languages.

Two applications were created, one using C++ and one using Java. Apart from a few syntax and necessary differences, both are as close to being identical as possible. Each application loads three files containing 1000, 10000 and 100000 randomly ordered integers. These files are pre-created and always contain the same values. They are randomly generated by another small application before testing.

The test runs three times, once for each file. When the data is loaded, it is sorted using quicksort. The data is reset using the dataset file and sorted again using insertion-sort. The sorted data is then saved to a file. Each test runs 50 times in a large loop and the times for loading, sorting and saving the data are saved. In total, 300 tests are run between the C++ and the Java application.

The results show that Java has a total time that is faster than C++ and it is also faster when loading two out of three datasets. C++ was generally faster when sorting the datasets using both algorithms and when saving the data to files.

In general Java was faster in this study, but when processing the data and when under heavy load, C++ performed better. The main difference was when loading the files. The way that Java loads the data from a file is very different from C++, even though both applications read the files character by character, Java’s “Scanner” library converts data before it parses it. With some optimization, for example by reading the file line by line and then parsing the data, C++ could be comparable or faster, but for the sake of this study, the input methods that were chosen were seemingly the fairest.

Keywords: Java, C++, Programming languages
ACKNOWLEDGEMENTS

I would like to thank Dr. Prashant Goswami for his help during the writing of this thesis.
# CONTENTS

- ABSTRACT .................................................................................................................. III
- ACKNOWLEDGEMENTS .......................................................................................... IV
- CONTENTS ................................................................................................................ V
- LIST OF FIGURES ...................................................................................................... 6
- LIST OF TABLES ......................................................................................................... 7
- LIST OF GRAPHS ...................................................................................................... 8

1. INTRODUCTION ........................................................................................................... 9
   1.1 JAVA ..................................................................................................................... 10
   1.2 C++ ..................................................................................................................... 10
   1.3 AIM ......................................................................................................................... 10
   1.4 OBJECTIVES ....................................................................................................... 11
   1.5 RESEARCH QUESTIONS ................................................................................... 11

2. RELATED WORK ......................................................................................................... 12
   2.1 BACKGROUND ..................................................................................................... 13

3. METHOD .................................................................................................................... 14
   3.1 INTRODUCTION TO EXPERIMENT ..................................................................... 14
   3.2 THE TESTING ENVIRONMENT AND SETTINGS ................................................. 14
   3.3 THE EXPERIMENT ............................................................................................... 15
   3.4 TIME LIBRARIES FOR C++ AND JAVA ............................................................. 15
   3.5 THE QUICKSORT ALGORITHM ......................................................................... 16
   3.6 THE INSERTION-SORT ALGORITHM ................................................................. 17
   3.7 DIFFERENCES BETWEEN APPLICATIONS ......................................................... 18

4. RESULTS .................................................................................................................... 19
   4.1 C++ TIMES .......................................................................................................... 19
   4.2 JAVA TIMES ....................................................................................................... 19
   4.3 COMPARISON DIAGRAMS ................................................................................. 20

5. ANALYSIS AND DISCUSSION .................................................................................... 23

6. CONCLUSION AND FUTURE WORK ...................................................................... 24

REFERENCES .................................................................................................................. 25
LIST OF FIGURES

Figure 4: The application design on page 14.
LIST OF TABLES

Table 1: The resulting times for the C++ application on page 19.
Table 2: The resulting times for the Java application on page 19.
LIST OF GRAPHS

Graph 1: Comparison of load times 20.
Graph 2: Comparison of quicksort times 20.
Graph 3: Comparison of insertion-sort times 21.
Graph 4: Comparison of save times 21.
Graph 5: Comparison of total times 22.
1. **INTRODUCTION**

Programming languages come in many different variations with each being unique in its area of use. When it comes to how well an application performs, and scales, depends highly on what language is used, and the programmers experience with that language. Learning the strengths and weaknesses of different languages is key to making an application perform and scale well on the target platform. The purpose of this study is to compare two languages, C++ and Java, that have many similarities but are executed in a very different way. There are many parts to a language, therefore comparing them in their entireties is out of the scope of this study. Rather, the focus will lie on the functionalities that are often used in almost all types of applications; loading or otherwise initializing, sorting and saving data. The results gathered from the comparison will show how these languages differ in these specific aspects and what their strengths and weaknesses are.

There are several levels of programming languages ranging from low to high. The higher level a language is, the higher the level of abstraction is, and these are often interpreted languages that are not compiled. They are executed on a line-by-line basis by an interpreter that is running in the background, and thus require no compilation times and have a less complicated syntax. Programmers can easily change values mid-execution to see differences. These scripting languages, as they are also called, are also more accessible to programmers with less experience because of their more simplified syntax and more flexible and forgiving framework. Larger applications often combine higher and lower level languages for the most balanced and flexible results. The added overhead of the interpreted language can often be overseen when looking at the bigger picture and what is gained. A good example of the flexibility such higher level languages provide is often a “garbage collector”. This background process quietly but efficiently clears the dynamic memory allocated by the programmer so that the memory does not have to be deallocated manually which can often lead to unintended memory leaks. Such luxuries in programming surely come with the cost of overhead, but once again, this can be overseen in the light of the development being faster and less prone to errors.

Lower level languages often have a less programmer friendly syntax and offer few to no luxuries in terms of background helper processes. Memory must be deallocated manually otherwise the application will have memory leaks which can often be hard to find. New programmers that are not accustomed to the specific lower level language will find it hard to debug and update the code since more experience is required to master the syntax. The style of programming and the knowledge of performance-boosting algorithms is always important, no matter what language is used, but there are often smaller, language-specific keywords and types that can be used to greatly increase the performance and scalability of an application when using lower level languages. These are often not found in higher level languages as the interpreter handles most of the optimization by itself and only leaves a simple set of functions to be used by the developer. Lower level languages are compiled to machine or hardware code which is as close to the hardware as can be. This language is numerical and contains the set of instructions that are sent to the Central processing unit, CPU, for execution. Machine code is easily read by the computer, so the speed of execution is extremely quick, but to compile a larger program can take several minutes to hours. Making changes can sometimes be tedious and time consuming since the application must be re-compiled each time a change is made.

In this study, the comparison of C++ and Java is essentially a comparison of a higher and lower level language. C++ being on the lower end, and Java being on the higher. Java and C++ are popular languages that are the basis for many backend applications and games. They are both object-oriented and very flexible in terms of the syntax and libraries available.

There are many opinions on C++ and Java and which one is more flexible or faster. Specially in the past there were many differences between languages and there were fewer optimizations as well as compiler tweaks like the Java JIT compiler which speeds up the compilation to bytecode. This can be seen in the opinions from a 1999 study by Prechelt. L, where C++ and Java’s efficiency was compared [4]: "The relative efficiency of Java programs is much discussed today, particularly in comparison to well-
established implementation languages such as C or C++. Java is often considered very slow and memory-intensive”. In another study by Nikishkov, G. P., Nikishkov, Y. G., & Savchenko, V. V. (2003) it was stated that [13]: “Although Java has attractive features for producing portable, architecturally neutral code, it is not widely used in engineering computations. Slower speed of Java codes is usually considered its main disadvantage”.

1.1 Java

Java is not a scripting language, but much like one, it runs on top of an interpreter. The interpreter is more of an engine that reads the code and executes it in a platform independent way. This engine consists of two parts, the Java runtime environment, JRE, and the Java virtual machine, JVM. These two parts work together to interpret the code and run it on any platform that supports Java. This engine is written in different lower level languages that are supported on the different platforms, for example C or even C++. The main reason for this, as mentioned before, is to make the code platform independent so that Java can run on many different computers and devices without programmers needing to learn a new language for each of them. This engine has helper functions running in the background, for example the garbage collector that was mentioned before. All of this adds overhead but also makes the language more flexible and easier to use. On today’s powerful machines, this overhead can be considered minimal, especially with the number of cores and the amount of memory that is available. Although, for larger applications, this might still cause a dip in general performance.

In a 2011 study by Oancea, B., Rosca, I. G., Andrei, T., & Iacob, A. I. [3] Java was tested as a backend application language and although it has more overhead, the results were favorable. They stated that it is a common belief that Java is still slower than C and C++ or even Fortran in performance, especially for computationally intensive numerical applications. But they developed a library for matrix computations using a set of optimized techniques, and it was compared to its competitors. The results showed that Java can achieve a performance comparable with other libraries developed in C, C++ or Fortran. This shows that Java does not lose by default when its performance is compared to lower level languages.

1.2 C++

C++ is the object-oriented successor to the popular language C. It is considered quite a low to mid-level language since it offers little in the way of abstraction and simplified syntax. C++ is compiled to machine code that contains the translated instructions which are then sent to the processor for execution. This makes C++ quite a low-level language that offers a lot in the way of micro-managing almost every aspect of the application. Managing memory allocation and deallocation is up to the programmer and there are many ways to create bugs if one is not careful. There are many libraries that offer extended functionality, like threads. Mastering C++ takes many years of experience, and it is constantly evolving and being optimized. Many applications have their backend built using C++ as it offers both great flexibility and an abundance of functionalities, although it is not multiplatform in the same way as Java is, rather there are different compilers for all the platforms that support the language, like Linux, Microsoft Windows and Mac Os. There are also some syntax and library differences that need to be considered when porting an applications between platforms. Usually a C++ application is more complex and not as easy to program. As was seen in the 1999 study by Phipps, G. [1], C++ code is more prone to bugs, longer development times and more bugs per minute than Java. But it has more features in its standard library [5] and thus a larger ability to optimize the code.

1.3 Aim

This study aims to compare the performance and scalability of C++ and Java when sorting, saving and loading large datasets.
1.4 **Objectives**

- Constructing two identical applications for C++ and Java.
- Loading three increasingly large datasets into the applications.
- Sorting the large datasets and saving them into files.
- Running the applications and gathering and comparing the timings for each dataset.
- Identifying and explaining the reasons for performance and scalability differences.

1.5 **Research questions**

- What are the differences between C++ and Java’s performance and scalability in terms of loading and saving large datasets to and from files?
- What differences are there between C++ and Java performance when sorting large datasets using the sorting algorithms quick sort and insertion-sort?
2 Related Work

Not much recent work has been done to specifically compare C++ and Java in this way. Research is rather done on specific libraries and their functionalities within the same language. But that is not to say that no work has been done. Although some of the work is not very recent, it still shines a light on the strengths and weaknesses of the languages that are tested.

There are a few different types of studies that have been conducted. One type is where languages are tested for their ease of use and flexibility like this older study from 1999 by Phipps, G. [1], an experiment was conducted to compare C++ programming and Java programming cleanliness and ease of use. Their results were that a typical C++ program had about two to three times as many bugs as a typical Java program and generated between 15 to 50 percent more defects per line. Java was also between 30 to 200 percent more productive. C++ also had about two to three times as many bugs per hour.

The other types of studies compare the general performance of the languages by creating applications much like this study, but on a larger scale. Like this study from 2008 Fourment, M. and Gillings, M.R. [2], they compared Java and C++ in bioinformatics. Here they found that: "implementations in C and C++ were fastest and used the least memory. Programs in these languages generally contained more lines of code. Java and C# appeared to be a compromise between the flexibility of Perl and Python and the fast performance of C and C++." This reflects the foundation that this study is built upon; that the complexity versus functionality problem of choosing a language is important depending on the required system being built.

In a study from 1999 by Prechet, L. [4], Java, C and C++ were compared by creating similar applications 40 times using different programmers and the general efficiency noted. They found that Java had a higher memory consumption and slower processing speeds compared to C and C++, but they mentioned that the skill of the programmers could affect the results, so Java could very well be as efficient as, or more than its competitors. This made the programming style and the experience of the programmers as important as the languages capabilities. In a direct quote from the study: “The programming problem investigated here required a non-trivial algorithm and data structure design. However, the data clearly shows that the importance of an efficient technical infrastructure (such as language/compiler, operating system, or even hardware) is often vastly overestimated compared to the importance of a good program design and an economical programming style”.

In another study by Saiedian, H., & Hill, S. in 2003 [5], the general programming libraries of C++ and Java were compared. These libraries are the core of these languages as they allow programmers to pick from an array of standard functionalities that are well optimized and maintained by their respective development teams. These libraries are the Standard Template Library (STL) and the Java Development Kit (JDK) for C++ and Java respectively. The libraries were tested for compile size, runtime memory usage and performance. The results were: “Based on the results, we conclude that the support provided for generic programming in C++’s STL is superior to that provided by JDK.”. These results are important because the libraries used strongly affect the general performance of an application as the more flexible library has more ways to optimize the system.

In the year 2000 study by Prechet, L. [6] they compared seven programming languages (C, C++, Java, Perl, Python, Rexx and Tcl) directly to one another by creating 80 similar applications using different programmers. The comparison investigated several aspects of each language, including program length, programming effort, runtime efficiency, memory consumption, and reliability. They found that scripting languages are written in about half the time as their lower-level competitors and the amount of code is also half as long, but their memory consumption was in general twice as high.

No clear differences in readability were observed. C++ and C were almost two to three times faster than Java in the different tasks performed and about five to ten times faster than the scripting languages. Here, the skill of the programmers and the language used highly impact the results, but generally the results clearly show that scripting languages are slower but easier to write and test. Lower level
languages are harder to master but are generally faster, and Java, being a low-mid level language does suffer drawbacks because of its interpreted nature.

There is a study from 2013 by Nuzman, D., Eres, R., Dyshel, S., Zalmanovici, M., & Castanos, J. [12] which highlights a very good point about languages which can be extremely optimized and have the fastest speed versus more easily accessible languages that are still fast but much easier to understand and modify. It asks if it is possible to retain the performance of C++ while compiling it in a similar way to Java using a JIT compiler that interprets the code. They found that: “Dynamic optimization has the premise of taking advantage of runtime information to dramatically boost performance; however, in the domain of statically compiled languages, this approach has so far had limited success, due to the costs associated with dynamic profiling and recompilation”.

2.1 Background

This study is conducted to show the strengths and weaknesses of C++ and Java in three areas that are used often in programming; loading, sorting and saving data.

It is important to compare languages directly as there is too little research done in this area. This is also the opinion of the year 2000 study by Prechelt, L. [6]: "Often heated, debates regarding different programming languages' effectiveness remain inconclusive because of scarce data and a lack of direct comparisons.”.

Performance and scalability are large factors in software development and choosing the right programming language is often a long process. Companies often choose higher level languages like python or JavaScript because of their ease of use and large flexibility, but when it comes to performance a low-level language is always preferred. Some backend systems need to handle lots of file input and output effectively, and others might need to process data in different ways, and some might do both. This study will compare C++ and Java in a reliable and fair way to see how they perform in these aspects of programming.
3 Method

To answer the research questions, this thesis conducted a comparative study which used quantitative measurements to compare two programming languages in their ability to load, sort and save increasing amounts of data. Two identical applications were created in both C++ and Java. Loading sorting and saving are individually timed in both applications and these results are then compared with each other.

As can be seen in the image, three pre-generated files containing randomly ordered integers were created using a separate application. These were then loaded into the application. The data was sorted using quicksort and insertion-sort and finally the sorted data was saved to a file. This was done 50 times for each of the dataset sizes. The times for each procedure in each iteration was saved and, in the end, an average time was calculated for all above mentioned procedures as well as the total iteration times.

3.1 Introduction to experiment

Two applications were created using an identical design, but one was programmed using C++ and the other one using Java. There are a few syntax differences that will be explained below, but each application aims to be as identical as possible. Both applications were programmed by the same programmer with high experience of both languages.

3.2 The testing environment and settings

The applications were created in different integrated development environments, or IDE’s. The Java application was created in Eclipse, and C++ application was programmed in Visual Studio 2017. Both projects were created as 32-bit. This test was done on a 64-bit Windows 10 machine with 16 GB of RAM, an i7 Processor running at 2.6 GHz.

Java does not have the same options as C++ when it comes to compiler optimization of the code since it is compiled by its own javac-compiler [11] and turned into bytecode which is then run on the JVM. This compiler optimizes the code automatically. To make this comparison as fair as possible, the C++ integrated development environment, or IDE, was set up to optimize the code using the O2 flag and run
the code in release mode, which skips the debugging overhead and optimizes the code as much as possible.

3.3 The experiment

The applications loaded pre-generated input data, that was stored in files as random integers ranging between 1 and the size of the dataset. This data was pre-generated by another application once before the tests were run. The loaded data was stored in a dynamically allocated array. The array was then sorted once using the quicksort algorithm. It was then reset using the current dataset file the experiment was using, and then it was sorted again using the insertion-sort algorithm. Finally, the sorted array was saved into a text file. The loading, sorting and saving methods were timed separately and any other statement which was not to be timed was ignored.

The dataset sizes were 1000 (one thousand), 10000 (ten thousand) and 100000 (one hundred thousand) and consisted of integers. The applications were run once using each of the datasets, and each loading, sorting and saving test was done 50 times. These were pre-generated to be the same throughout the entire testing process. Random values can give largely varying results while sorting and thus unfair advantages.

The reason for using two algorithms was that it would test the performance of recursively sorting data and sorting it using more simple means. There were other candidates for example heap-sort instead of quicksort, and bubble-sort instead of insertion-sort, but in the end, quicksort was chosen as it is widely used, easy to implement and it is one of the fastest sorting algorithms. Insertion-sort was chosen because it is not as slow as bubble-sort, it can be used with a larger dataset size like 100000 integers, it can be easily implemented, and it has acceptable performance when sorting smaller datasets.

After both applications were run three times each for all the datasets, there were six resulting files with the times. Both applications were run for 150 iterations each resulting in 300 total iterations combined.

3.4 Time libraries for C++ and Java

To time the C++ application, the standard library “ctime” [11] was used. This library contains the “clock_t” type and “clock” function. These are used in combination to get the current processor-ticks since the system started. To get the elapsed time for an event, for example the loading of data, the start time was subtracted from the end time. This was done by collecting the current time before the event and once after the event. The exact way this was done for one of the functions is shown in the below example from the C++ application code.

```cpp
start = std::clock();
loadFromFile(arr, SIZE, FILENAME);
duration = static_cast<long>((std::clock() - start) / (double)CLOCKS_PER_SEC) * 1000.0);
```

For the Java application, the standard “System” package was used which has a built-in time function. To gather the time in milliseconds for a specific event, the “currentTimeMillis” function [14] was used in the same way it was for C++. One measurement was taken at the start of an event and one at the end, these values were then subtracted to get the elapsed time. The exact way this was done for one of the functions is shown in the below example from the Java application code.
3.5 The quicksort algorithm

Quicksort is a recursive sorting algorithm that uses a divide and conquer process to sort smaller chunks of a larger dataset and then put them all together in the end. This algorithm is most effective for large arrays.

To start the sorting algorithm, a pivot value is chosen from the end of the array. The array is sorted by placing the numbers that are smaller than the pivot to the left and the numbers that are larger to the right. After this procedure, the smaller numbers and the larger numbers are sorted individually as two different arrays using their own pivot value. This process continues until the entire array is sorted by continuously dividing the array into smaller parts as needed. The algorithm is one of the most efficient, but it has a worst case of $\mathcal{O}(n^2)$. This can occur if the pivot value that is chosen for each partition does not divide the array somewhere in the middle, but rather it is either the smallest or largest number in the dataset. This in turn makes it so that the array cannot be divided into any parts-

The best case is $\mathcal{O}(n \log n)$. This can occur if the perfect pivot value is chosen at each partition. That value would be at the center of the array and it would divide the dataset perfectly each time.

The following pseudocode explains the algorithm:
INTEGER FUNCTION partition(array, low, high)
    pivot = array[high];
    i = (low - 1);
    FOR(j = low; j less than high; j++)
        IF arr[j] is less than or equal to pivot THEN
            i = i + 1;
            Swap arr[i] and arr[j];
        END IF
    END FOR
    Swap arr[i + 1] and arr[high];
END FUNCTION

FUNCTION qsort_custom(array, low, high)
    IF low is less than high THEN
        pi = partition(array, low, high);
        qsort_custom(array, low, pi - 1);
        qsort_custom(array, pi + 1, high);
    END IF
END FUNCTION

3.6 The insertion-sort algorithm

Insertion-sort is a simple sorting algorithm that requires many iterations and tests to sort data. It works by keeping one part of the entire dataset sorted, and the other part unsorted.

Each value from the unsorted part is compared to the sorted sub-array and inserted at the correct position. This way, the sorted array is guaranteed to always be sorted and there are less checks that are made compared to for example bubble-sort. After each insertion, each value that is larger than the inserted value needs to be shifted one step towards the end of the array. This shift is very expensive, specially for very large algorithm needs to iterate through unsorted elements as well as the larger sorted elements to move them.

The worst case is O(n^2) and the best case is O(n). Thus, for this study this algorithm limits the amount of data we can process to a dataset no larger than 100000 integers, as a dataset that is too large would take too much time to iterate through 50 times.

The following pseudocode explains the algorithm:
FUNCTION insertionsort(array, size)
    key, j, i;
    FOR(i = 1; i less than size; i = i + 1)
        key = array[i];
        j = i - 1;
        WHILE(j larger or equal to 0 AND arr[j] larger than key)
            arr[j + 1] = arr[j];
            j = j - 1;
        END WHILE
        arr[j + 1] = key;
    END FOR
END FUNCTION

3.7 Differences between applications

The main difference between the Java and C++ applications is the syntax for loading and saving a file. In C++ this is done by using the standard library “fstream”. In Java there are several ways to save and load files. In this study the “File” and “Scanner” libraries are used.

There is a difference in memory management as mentioned in the previous sections. C++ requires manual deallocation of memory while Java uses its garbage collector. These deallocations are almost instant and are done at the end of each iteration. This process does not affect the total time, so it is not timed. This was tested separately before the test was conducted.
4 RESULTS

The results show that Java has a total time that is faster than C++ and it is also faster when loading two out of three datasets. C++ was generally faster when sorting the datasets using both algorithms and when saving the data to files.

In terms of loading data from a file, character by character, Java was quicker using its “File” and “Scanner” object combination compared to C++ using its standard library “fstream” library. Java was faster loading the two larger datasets but slower loading the smallest one where it was about two seconds slower. The differences for the other two datasets were about one second for 10000 integers, and about 22 seconds for 100000 integers.

The loading part of the application slowed the C++ version down enough to fall behind a considerable amount. The average total time was less for the Java application because of its faster file reading. But when it comes to sorting and processing of the data, C++ was generally faster in all areas, specially sorting the largest dataset using insertion-sort where it was about eight seconds faster and using quicksort where it was more than one second faster.

Saving the data at the end of each iteration was quicker in the C++ application by seconds, specially using the largest dataset where there was a difference of about two seconds.

The results show that as the dataset-sizes increase, the time differences increase considerably and exponentially. The scalability of the Java application is better than its C++ counterpart when it comes to loading, but when it comes to sorting data using the quicksort algorithm and saving data, the C++ application scales better. Insertion-sort did not scale well in both applications.

Tables and graphs in the sections below clearly show these differences. In this study, times have been mentioned as seconds to improve understanding, but all times were measured in milliseconds. The left-most column of the tables displays the number of integers used, and the remaining columns are the average times for the processes in milliseconds.

4.1 C++ times

<table>
<thead>
<tr>
<th>Dataset size</th>
<th>Load</th>
<th>Qsort</th>
<th>iSort</th>
<th>Save</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0,9 ms</td>
<td>0,04 ms</td>
<td>0,1 ms</td>
<td>1,64 ms</td>
<td>2,68 ms</td>
</tr>
<tr>
<td>10000</td>
<td>7,5 ms</td>
<td>0,5 ms</td>
<td>12,62 ms</td>
<td>8,52 ms</td>
<td>29,14 ms</td>
</tr>
<tr>
<td>100000</td>
<td>73,18 ms</td>
<td>6,04 ms</td>
<td>1208,58 ms</td>
<td>16,94 ms</td>
<td>1304,74 ms</td>
</tr>
</tbody>
</table>

Table 1 “C++ times.”

4.2 Java times

<table>
<thead>
<tr>
<th>Dataset size</th>
<th>Load</th>
<th>Qsort</th>
<th>iSort</th>
<th>Save</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>2,34 ms</td>
<td>0,06 ms</td>
<td>0,22 ms</td>
<td>2,18 ms</td>
<td>4,8 ms</td>
</tr>
<tr>
<td>10000</td>
<td>6,44 ms</td>
<td>0,76 ms</td>
<td>12,48 ms</td>
<td>8,08 ms</td>
<td>27,76 ms</td>
</tr>
<tr>
<td>100000</td>
<td>51,28 ms</td>
<td>7,66 ms</td>
<td>1216,06 ms</td>
<td>19,9 ms</td>
<td>1294,9 ms</td>
</tr>
</tbody>
</table>

Table 2 “Java times.”
4.3 Comparison diagrams

The y-axis of the graphs shows the size of the dataset and the x-axis displays the time taken in milliseconds.

Graph one clearly shows that C++ (orange) is much slower and scales less than Java (blue) when it comes to loading from a file. The only time it was faster was using the smallest dataset. Note that loading the largest file took almost seven seconds (70 milliseconds).

![Loading Java vs. C++](graph1)

Graph 1 “Loading comparison.”

Graph two displays the result of the quicksort comparison, and C++ (orange) was faster, even if it is with just a few milliseconds.

![Quicksort Java vs. C++](graph2)

Graph 2 “Quicksort comparison.”

In graph three the difference in time is not all that big, but it took almost 12 seconds (1200 ms). The insertion-sort algorithm did not scale well and it performed very poorly for both of the languages. C++ (orange) managed to be faster but just by a few milliseconds. The smallest dataset results cannot be seen.
in this graph, but C++ completed the operation in 0.1 milliseconds, and Java was done in 0.22 milliseconds.

Graph 3 “Insertion-sort comparison.”

Graph four shows that C++ (orange) was faster than Java when saving the data to a file. Java was faster (blue) in the second test.

Graph 4 “Saving comparison.”

Graph five displays the total average times for the applications to complete all 50 iterations. Java (blue) was slower than C++ in all but one test that used the smallest dataset of 1000 integers. The result for the smallest dataset is hard to read in this graph, but C++ had a total time of 2.68 milliseconds and Java had a total time of 4.8 milliseconds.
Graph 5 “Total comparison.”
5 ANALYSIS AND DISCUSSION

The results show that Java is faster in total when running an application like this, but there are a few important factors to look at. The processing of the data was faster in C++, and most applications are focused on this task, not loading and saving data each iteration. As seen in Graph two, C++ is faster at sorting the data using quicksort by about one and a half milliseconds in the test using the largest dataset. This is a large difference for applications that might run this code each iteration and where the dataset sizes fluctuate and become larger and thus C++ would scale better using this algorithm.

The loading of the larger datasets set C++ back in the larger scope of things but considering these types of operations are not often performed in an application in this manner, it is easy to say that C++ would be faster in a more conventional type of application and also if the more optimized type of loading would have been used, such as keeping the files in a binary format as loading binary files is faster than regular text files. Loading a file by reading one character at a time is very expensive and the input stream operator in C++ is also very expensive to use. The better way to do this would have been to read one line at a time and then parse this data in the application. C++ did not scale well as it was faster using the smallest dataset and ended up being more than 20 milliseconds slower using the largest dataset. This would be detrimental in a real-time application that loads data often, although as was mentioned before, this technique would probably be too inefficient to be used in regular applications.

In the case of Java, we saw that the loading was faster in most cases, but there are some large differences in the internals of the “Scanner” library compared to the “fstream” library. In Java, the “Scanner” object loads the file and turns the characters into tokens separated by white-spaces and they have their types defined so they are easier to fetch within the application, as can be seen in the reference manual [9]. This in turn acts as if the file was loaded line by line and then processed while C++ truly reads the data character by character, while the file is open in the stream, using iterators and then tries to cast this value to the desired type of the input variable, or to an integer in this case. If C++ would do the same as Java, it would most probably load the file contents faster.

C++ was faster when sorting and processing data, both using a slower insertion-sort and a quicker iterative quicksort. Saving data to a file was also faster. All memory was handled manually, there was more code and it was harder to write, but in the end the performance was not hindered by any virtual machine overhead and it resulted in faster speeds. This result was also found in most of the related works that have been done. When looking at the time that insertion-sort took, one can clearly see how badly this algorithm scales in general. The difference in times between the dataset size of 10000 and 100000 was more than 1000 milliseconds, and this poor scalability is not feasible to use in an application where dataset sizes grow quickly. If one looks at quicksort, the scalability was much better for both languages with a small difference of about six to seven milliseconds between the second and largest datasets.

Today, Java is more optimized using a “just in time”, or JIT compiler. This type of compiler speeds up the process by compiling bytecode to native machine code at runtime as can be seen in the official documentation [10]. Even though this is extremely fast, it does not remove the overhead of the virtual machine, but it does negate some of the performance setbacks of the regular compiler. This helps Java come close to the performance of C++, and with its ease of use it might often be chosen over its competitors. But sadly, with the rise of scripting languages, this speed gain is not enough compared to the flexibility of interpreted languages for main stream programming. Also, memory use is not lessened by the JIT compiler.
6 CONCLUSION AND FUTURE WORK

The results of this study and the research done in previous works shows that lower level languages are faster than higher level ones. It is important to know this when choosing the programming language for a specific task.

More new research is required in this field to properly document the key differences in all the programming languages that exist. Languages are updated frequently, and they change quickly. Open source libraries can be added that perform much better than standard library functions, and sometimes open source libraries are themselves added to the standard libraries. Because of this, research in this field needs to be done more often to update the results, and not only between languages of different types, but also similar languages. One can argue that comparisons between similar languages would be even more sought after and promote healthy growth of the languages themselves.

Comparing languages in key aspects like sorting, saving, loading and memory use using identical applications and experienced programmers would yield fair results that can then be used to properly organize languages in tiers, ranging from low level languages to higher level ones. The results can then be compared easily.

It is important to look at the internals of the libraries that are used for common processes like file input and output to see how it is done. Research in this field is also limited as there are not many studies that compare library functions from different languages as was done in this study.
REFERENCES


