Computerarkitektur og systemprogrammering

DM548

Assignment 1 - Sorting in assembler

Authors:
Kramer , Philip
Nørgaard, Anton
Porsteinsson , Vilmundur
Gyldenbrand, Jeff
Zdunek, Karol

Student mail:
phkra16@student.sdu.dk
antno16@student.sdu.dk
vipor16@student.sdu.dk
jegyl16@student.sdu.dk
kazdu16@student.sdu.dk

1 Introduction

The aim of the project was to implement a sorting algorithm of our choice in the $x86-64$ assembly language, with $AT\&T$ syntax. The software must take a file as argument, open the file, and read it, and afterwards sort the numbers in ascending order. The result will be printed as std out. We were then tasked with analyzing the efficiency of our algorithm, using various measurements, including MCIPS, runtime and the total amount of comparisons made. Additionally, these calculations had to be depicted in a meaningful manner to evaluate the speed of the algorithm. In our assignment, we choose Bubble Sort as the algorithm of choice.

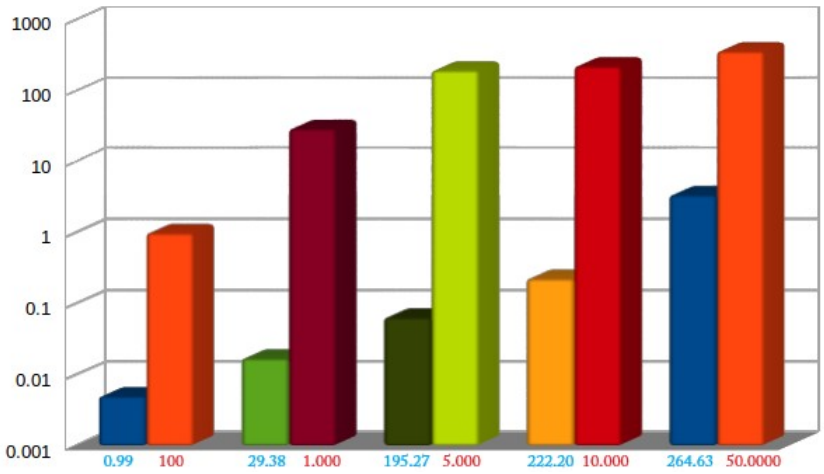
2 Time and MCIPS measurements

We generated 10 different replicates with different random numbers between 0 and 32767 to test our implemented code. The files are with given amount of numbers: 100, 1000, 5000, 10000, 50000

Antal	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.
100 time	0.005	0.005	0.005	0.005	0.010	0.011	0.003	0.004	0.005	0.004
100 MCIPS	0.99	0.99	0.99	0.99	0.49	0.45	1.65	1.23	0.99	1.23
1000 time	0.017	0.090	0.014	0.020	0.008	0.009	0.009	0.013	0.074	0.025
1000 MCIPS	29.38	5.55	35.67	24.97	62.43	55.50	55.50	38.42	6.75	19.98
5000 time	0.064	0.063	0.314	0.068	0.066	0.262	0.066	0.047	0.063	0.062
5000 MCIPS	195.27	198.37	39.80	183.78	189.35	47.70	189.35	265.90	198.37	201.57
10000 time	0.225	0.240	0.162	0.145	0.153	0.151	0.144	0.184	0.245	0.153
10000 MCIPS	222.20	208.31	308.61	344.79	326.76	331.09	347.18	271.71	204.06	326.76
50000 time	3.428	3.252	3.432	3.467	3.266	3.433	3.403	3.429	3.607	3.499
50000 MCIPS	364.63	384.37	364.21	360.53	382.72	364.10	367.31	364.53	346.54	357.23

Table 1: Time and MCIPS

As shown in Table 1 the running time for the files with a amount of numbers between 100-10.000 has a visible time difference. For example the 5.th file with 1000 numbers has running time which is over 10 times less than the 2.nd file, where some other took half the time. The time for sorting a file with up to 10.000 numbers is not that bad, comparing to files with 50.000 numbers.



As shown in the graph above, the respective tests of the files, are represented in columns for the count of numbers in the file and associated MCIPS. Because difference in the MCIPS for 100-digit files and 50000-digit files, is so large, the graph is represented logarithmically.

3 Discussion

Our implementation of the bubble sort-algorithm makes the same amount of comparisons on each file with the same number-count. Meaning that any file with e.g. 100 arbitrary numbers would always have the same amount of comparisons. Our measurements on respectively files with 100-, 1.000-, 5.000-, 10.000- and 50.000 numbers resulted in:

Number-count in file	Comparisons (unsorted file)	Comparisons (sorted file)
100	4.950	4.950
1.000	499.500	499.500
5.000	12.497.500	12.497.500
10.000	49.995.000	49.995.000
50.000	1.249.975.000	1.249.975.000

Normally bubble sort has the asymptotic running times; Worst case $O(n^2)$, average case $O(n^2)$ and best case $O(n)$. However, the way we've implemented the algorithm, it has $O(n^2)$ in all three cases. That means, even though if the input-numbers is already in sorted order, our algorithm still would need to make $n * n$ comparisons.

4 Motivation

We had three sorting-algorithms in mind; Quicksort, insertion sort and bubble sort. We choose the bubble sort algorithm because it was the most simple and easy to implement. Our ambitions were to understand and implement a sorting-algorithm in assembly, we didn't feel the need to compete for the fastest sorter, or smallest program. That is also the reason, we choose to use the given code-snippets, and not reinvent everything from scratch. The given snippets gave us a lot of help while implementing our sorting algorithm, but also by studying each snippet line by line we could understand and learn a lot.

5 Conclusion

Considering that our algorithm makes 4.950 comparisons on just 100 numbers, we have $4.950/100 = 49.5$ times more comparisons than actual numbers. For larger amounts, the ratio is even worse. We have $124.997.500/50.000 = 2.4999.5$ times more comparisons than actual numbers if we sort a file with 50k numbers. This is obviously a very inefficient way to sort numbers, but not surprising for Bubble Sort. Additionally, considering how powerful even a lower-end modern computer is, the fact that it took over three seconds to sort 50.000 numbers is an indicator that the performance of the algorithm is very poor. Our MCIPS aren't any better. If we look at the average MCIPS for 1.000 numbers, we get an average of 33.4175 million compare instructions per second, which again is disproportionately high for our algorithm when we only have 1.000 numbers to sort. In our conclusion, the inefficiency of Bubble Sort is not surprising and is noticeable in the measurements.