Report

Siver Al-khayat (sialk16 : Exam Number 439492)
Julia Schawaller (jscha18 : Exam Number 463169)

-

Computer Architecture and System Programming
Assignment 1

November 18, 2018

1 Introduction

In this project our objective is to program a assembly program, that takes a file of character numbers, sorts the numbers and outputs them to a file. We also do some analysis on the program, comparing run time of different files sizes, look at the ramp up time for different file sizes, and look at the million compare instructions per second, for the program and chosen algorithm.

2 Design

Here is a general description of the functions in the assembly code "myprogram.s". We use selection sort in the program to sort the numbers.

In the first "_start" function we load the numbers from the file into a buffer and then make a new buffer to contain the number characters from the first buffer, but as integers instead, so we can do calculations and comparisons with them as integers. We then call the "selection_sort" function.

In the "selections_sort" function we setup the relevant variables/data in the registers and run a loop of two function, "find_smallest_num" and "swap_num_in_buffer". We run this loop until we reach the end of the number buffer.

In "find_smallest_num" we loop through the unsorted part of the buffer, find the smallest number.

When the "swap_num_in_buffer" is called we first print the smallest number found in the unsorted part of the buffer, then we swap the smallest number with the first number of the unsorted part of the number buffer, and increment the divider that divides the sorted from the unsorted numbers.

This is how we implemented the selection sort algorithm in the assembly code "myprogram.s".

3 Analysis

We measured the runtime of our sorting algorithm without printing the numbers to stdout, because this takes relatively long compared to the sorting itself. Without this overhead, the evaluation of the algorithm performance compared to the theoretical runtime can be done in an appropriate way. You can see the results in Table 1 and 2. The algorithm was tested on a personal machine using the Windows Subsystem for Linux.

Numbers	Instance 1	Instance 2	Instance 3	Instance 4	Instance 5
100	0.018	0.017	0.015	0.020	0.016
1000	0.017	0.018	0.017	0.017	0.016
5000	0.043	0.041	0.046	0.051	0.043
10000	0.119	0.128	0.124	0.136	0.120
50000	2.279	2.681	2.802	2.362	2.421

Table 1: Runtime in seconds for instances 1-5

Numbers	Instance 6	Instance 7	Instance 8	Instance 9	Instance 10
100	0.018	0.017	0.017	0.017	0.015
1000	0.017	0.026	0.025	0.017	0.016
5000	0.053	0.048	0.043	0.050	0.056
10000	0.129	0.124	0.130	0.126	0.122
50000	2.709	2.676	2.380	2.401	2.317

Table 2: Runtime in seconds for instances 6-10

Using the selection sort algorithm, we always have

$$\sum_{i=1}^{n-1} i$$

comparisons for sorting n numbers. This is because in the first iteration we search for the smallest number and therefor make $n-1$ comparisons, in the second iteration we search for the smallest number in the left over $n-1$ numbers and therefor make $n-2$ comparisons and so on. In Table 3 you can see the average runtime for the number of comparisons. As the number of comparisons increases, the runtime also increases. We have to notice, that we have some overhead for the file handling and we need time for the swaps that are performed during the sorting. One can see with regression, that we can represent the overall runtime as a second degree polynomial of the number of comparisons.

Numbers	Number of comparisons	Average overall runtime
100	$0.00505 \cdot 10^6$	0.017
1000	$0.5005 \cdot 10^6$	0.0186
5000	$12.5025 \cdot 10^6$	0.0474
10000	$50.005 \cdot 10^6$	0.1258
50000	$1250.025 \cdot 10^6$	2.5028

Table 3: Relation of runtime and number of comparisons

Measuring the ramp up time for each instance shows that there is a logarithmic relation between the file size and the ramp up (Table 4):

$$\text{RampUpT}(size) \approx 0.0005 \cdot ln(size) + 0,0113.$$

Numbers	Ramp up
100	0.0134
1000	0.015
5000	0.0159
10000	0.016
50000	0.0165

Table 4: Average ramp up time in seconds

In order to relate the number of comparisons to the execution time instead of the overall runtime, we subtract the ramp up times from the overall runtime and display the result as MCIPS (million compare instructions per second) in Table 5 and 6. We have a high slope if we compare the smaller files of 100, 1000 and 5000 numbers but for the greater files, the slope is not that high.

Numbers	Instance 1	Instance 2	Instance 3	Instance 4	Instance 5
100	1.098	1.403	3.156	0.765	1.942
1000	250.25	166.83	250.25	250.25	500.5
5000	461.347	498.108	415.365	356.197	461.347
10000	485.485	446.473	463.009	416.708	480.817
50000	552.497	469.141	448.761	532.946	519.869

Table 5: MCIPS for instances 1-5

Numbers	Instance 6	Instance 7	Instance 8	Instance 9	Instance 10
100	1.098	1.403	1.403	1.403	3.156
1000	250.25	45.5	50.05	250.25	500.5
5000	336.995	389.486	461.347	366.642	311.783
10000	442.522	463.009	438.640	454.591	471.745
50000	464.262	470.023	528.887	524.229	543.37

Table 6: MCIPS for instances 6-10

4 Discussion

We chose to implement the selection sort algorithm, because it is an in-place algorithm which helps us avoid additional memory management.

The selection sort algorithm is an relatively simple algorithm, therefore make the implementation more straight forward, compared to for example Radix Sort, where we have to keep tack of a radix point.

As we see there is a significant increase in MCIPS, when the file becomes bigger. We suspect that the reason for this increase, is because some data is cached, but we cant say for sure.