Introduction

In this report we describe how we implemented a microbenchmark as part of a Mergesort algorithm in C and x86 assembly. Mergesort is a “divide and conquer” algorithm that sorts elements in an array. It has a O(n log n) complexity which makes it significantly faster than most other sorting algorithms. After profiling our algorithm using “gprof” we found that the heaviest workload of the microbenchmark was the sorting of the elements and adding them in a single array which is what we are going to implement in assembly.

C-implementation:

The C-implementation of our microbenchmark starts with copying the elements from the array being sorted into two different temporary arrays. One of the temporary arrays contains all the elements in the left part of the array to be sorted, while the other temporary array contains elements of the right side of the array.
When all the elements have been copied from the main array, the algorithm starts comparing the elements in both the temporary arrays against each other. The smaller value of the two compared gets added into the main array, overwriting the value on the current index of the main array.

Whenever one of the temporary arrays have added all their elements into the main array, the loop breaks. As there still can be elements left in one of the arrays, we check whether any of the arrays still have elements inside themselves that has not been added into the main array. If so, we iterate through that array and add the elements into the main array. The elements inside this array does not have to be compared against each other, since they already have been sorted earlier recursively.

Assembly implementation:

The main part of the assembly code is the comparing of the elements in both the temporary arrays and putting the smallest value back into the main array again. For each element we put into the main array we are writing back to the memory, which is one of the costliest instructions we perform in the code.

The part of the assembly code which “reset” the registers and arrays before running the sorting-loop over again is implemented in a way that the stack has to be cleared of all the values we pushed (except the values we preserved in the beginning of the assembly code). We also must “reset” some of the registers so that they always contain the same variables when the loop starts over. This result in a lot of instructions where we pop and push elements on/off the stack, which could have been worked around making us push and pop less on constant variables, and instead overwritten values from the base pointer to the corresponding registers.

Methodology

How we measured hotspots:

We profiled our algorithm using *gprofiler* which provided us with an analysis of our program. The result shows that the main hotspot was the *Sort*-function. As seen below 94,44% of the run time is used inside the *sort*-function.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 94.44      0.17     0.17   999999     0.00     0.00  sort
  5.56      0.18     0.01        1    10.00   180.00  mergesort
  0.00      0.18     0.00        2     0.00     0.00  gettime

 %         the percentage of the total running time of the
time       program used by this function.
```

Computer(s) used for the experiment

The lab-computers are the only computers we used during the implementation, profiling, and benchmarking of the described algorithm.

How we measured execution time for the benchmark

We measured the execution time for our benchmark using the *gettime*-function. During an earlier assignment we were provided this function and it is therefore not our work. The

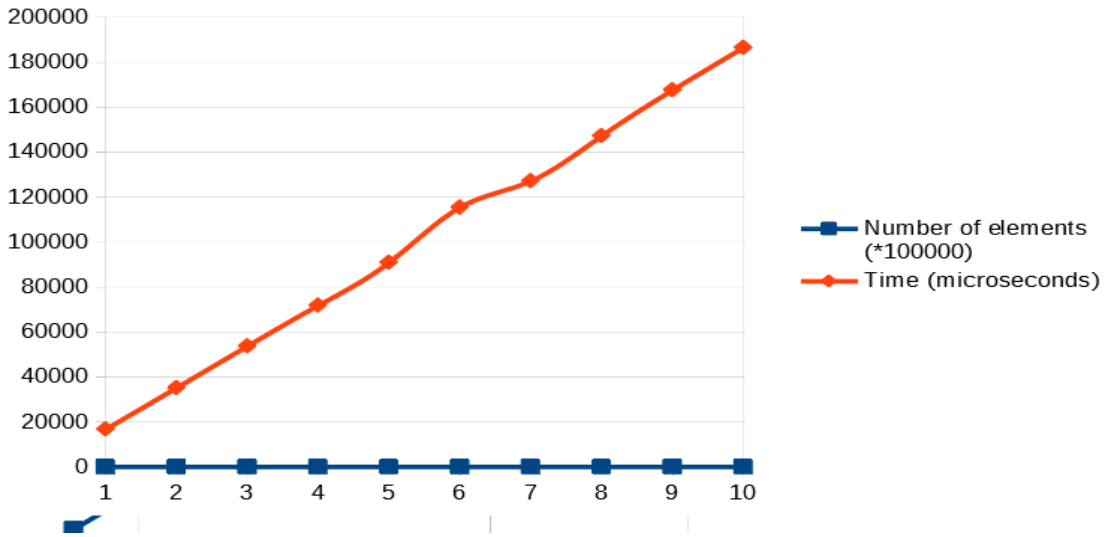
gettime-function provides results measured in microseconds which represents a time-resolution of one millionth of a second.

Experiment parameters

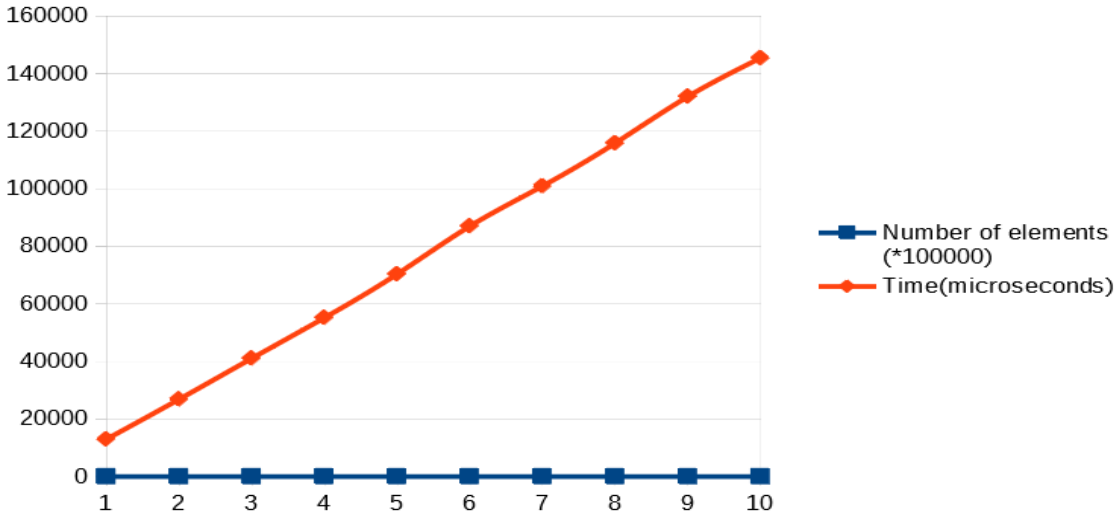
We benchmarked our program using both the assembly implementation and the regular C implementation and compared both results. We run each implementation three times with 100 000 – 1 000 000 elements being only integers and used the average time to create a line diagram as illustrated bellow. We found that the assembly implementation beat the C implementation at level 0 but was slower when the level was 1 or higher.

Results:

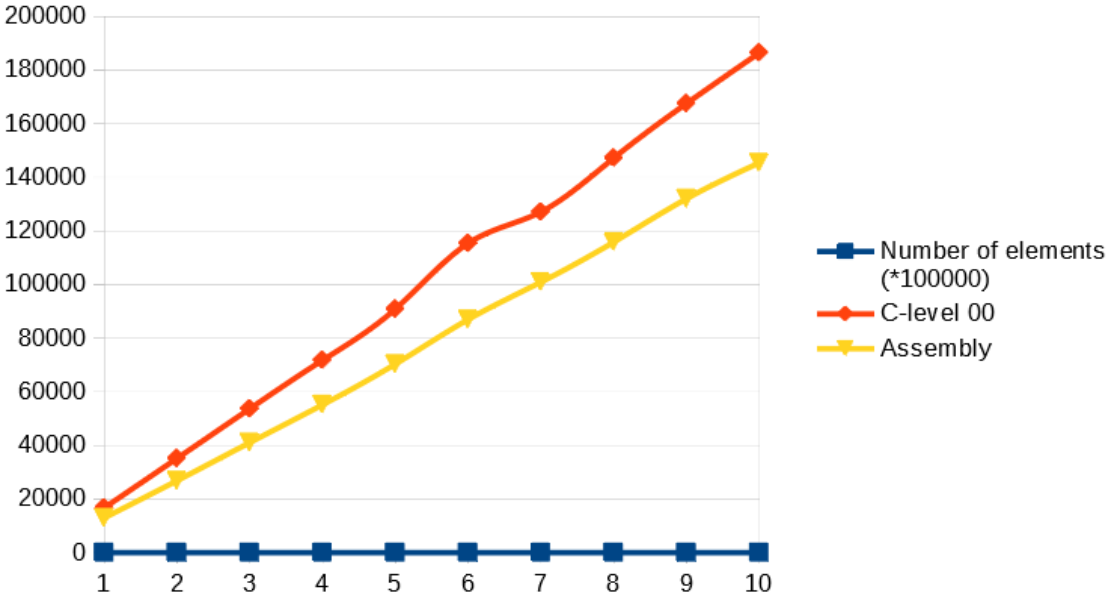
Measured time for assembly



Measured time for C-code at level 00:



Comparison: Assembly VS C



Discussion

We found that our assembly implementation beats the C implementation at level 0, but is slower when the level is 1 or higher. The main reasons for this may be that we make too many unnecessary memory accesses. For example, we copy a constant variable from the stack to use it for comparison. After the comparison we then push this copy to another location on the stack which is later popped and pushed to or from the stack for later comparisons. This process is repeated for each element in the array and may be repeated millions of times depending on the array size. This is very unnecessary since writing to or from memory is a slow process. A better solution would be to only copy this constant variable from the stack and overwrite a register when we need it, and not think about pushing it back to another location on the stack

A last honorable mention of what we have learned is that not popping %ebp from the stack at the end of the code will result in a lot of painful *illegal instruction* warnings and *segmentation faults.*