Implementation of Optimised Counting Sort / Radix Sort in Assembly

Marcus Søndergaard, Jonathan Kilhof, and Olivia Jespersen

November 3, 2024

Abstract

In this report we will investigate an efficient implementation of counting sort / radix sort in x86-64 assembly on Linux. We will discuss the trade-offs between different optimisations together with their implications. The sorting will be of coordinates by their y-components.

1 Introduction

In this project, we have chosen the task of making a program for sorting coordinates in x86-64 assembly on Linux. Our input will be a file of 2-dimensional coordinates which we will have to sort by the y-coordinates with the correct x-coordinate following the y-coordinate. Our program should not only sort correctly but it should also sort as fast as possible. In this report, we will explain how we achieved this goal, why we chose counting / radix sort, and how we optimized our program for speed.

The format of our input file will be such that there is one coordinate per line. Each line will be arranged in such a way that there is a number from 0 to 32767, the x-coordinate, followed by a tab character, then a new number also from 0 to 32767, the y-coordinate, and finally a new line character.

```
17028	7296
8647	13573
31908	13573
5248	0
23630	467
28379	32767
23550	6874
10002	10710
5936	21504
20029	9134
```

Figure 1: An example of the contents of an input file.

2 Design Decisions

2.1 Selection of Sorting Algorithm

The creation of any sorting program starts with the selection of a sorting algorithm. Since we are interested in fast sorting even for large numbers, we would like to choose an efficient algorithm. To this end, we can use the fact that we will only be sorting numbers and choose a non-comparative sorting algorithm. This could for example be counting sort or radix sort that both sort in linear time. Because we had the course Algorithms and Data Structures in a previous semester we have been introduced to many different sorting algorithms, each with their own advantages and disadvantages. When choosing a sorting algorithm, different attributes need to be taken into account. Mainly we need to think about the runtime of the algorithm, how much memory the algorithm needs, and how challenging the algorithm would be to implement.

We chose to start with counting sort as it is simple and can be used as a component of radix sort. Counting sort runs in $\Theta(n + k)$ time where n is the count of elements to sort and k is the largest number to sort. In our case, we will have a rather large overhead as k will be 32767. This, however, did not turn out to be much of an issue in practice.

Later, we changed our sorting algorithm to a modified radix sort. Instead of using a set radix, it "splits" each number into a higher and lower part. The lower part is the least significant byte of the number – the first digit using a radix of 256 so to speak – while the higher is everything above – the other digits.[1] It then sorts, using counting sort, first the lower, then the higher part. It would therefore seem that it should be slower than counting sort since it must do two of them, however, the opposite is true. This is likely due to the fact that there are far fewer buckets in each counting sort and thus they fit in the cache.

2.2 Selection of Number Encoding

A different, but less obvious choice is the choice of the encoding of the numbers. The straightforward approach would be to store the numbers as 16-bit unsigned integers and leave it at that. This would probably be the fastest choice for just the sorting. However, as we are interested in the overall performance of the program, it will be worthwhile to consider the effect on parsing and printing too.[2]

Before looking at alternatives, it would be advantageous to examine what properties we need from a number encoding. Since we have chosen counting sort / radix sort, the only property we need is for the bit pattern of the encoding to be in order. I.e., all numbers in the encoding should still have the same ordering when interpreted as an unsigned integers.

One choice might be a binary-coded decimal (BCD) encoding. In this encoding each digit of a decimal number is stored in a fixed amount of bits. Each digit is usually given 4-bits – called packed BCD – or 8-bits – called unpacked BCD. The advantage that BCD has over unsigned integers is a very simple conversion to and from ASCII. Whereas converting an unsigned integer requires multiplication when parsing and division when printing – both relatively expensive CPU operations – BCD only requires bit shifting. Additionally, it allows for more radical optimisations. More on this in the section on implementation.

Another choice might be to simply not convert from ASCII and store each digit as its ASCII code. This would be make for an efficient conversion but lead to a lot of wasted space.

We decided to first implement the program using unsigned integers and then later switched to a packed BCD encoding. The switch lead our numbers to grow from 16-bit to 32-bit which lead to higher memory usage. This subsequently lead to slower performance for the sorting since more memory had to loaded. Nevertheless, the improvements this allowed to the printing far outweighed any performance penalty elsewhere. The reason the size grew is that we need 4-bits per decimal digit and our highest number, 32767, has 5. Thus it requires 20-bits but, to have nice alignment, we increased the size to 32-bit.[3]

[1] You would be forgiven for thinking the higher part must also be 1 byte since the numbers fit in 16-bit. However, as we will explain in the next section, it is not actually.

[2] As it turns out, both parsing and printing (and allocation of memory) is slower than sorting. The optimizations have therefore been pointed this way. Additionally, while this report is presented as if the design was decided before implementing, the actual development was more back and forth.

[3] Later we will also use some of this additional space for an optimisation.

3 Implementation

In this section we will discuss interesting parts of the implementation. Each section will focus on a specific section of the code and give an overview or explanation of the interesting parts of that section – most of which are optimisations.

3.1 Guessing the Number of Coordinates

When parsing the numbers from ASCII, a space is needed to place the resulting numbers into. This space needs to be big enough to hold all the parsed numbers. The straightforward approach would be to count the number of lines – and thus the number of coordinates – and use that to allocate enough memory. This, however, requires reading through the entire file, a not insignificant overhead. It would therefore be better if there was a way to know beforehand how many numbers the file has.

A possibility presents itself through the file size. However, even though it is possible to get the file size using the system call `fstat`, it is unfortunately impossible to calculate the number of coordinates thereof – we do not know how many bytes each number takes up. Nevertheless, it is possible to calculate the maximum number of numbers in the file (and the minimum, for that matter). The argument is as follows. Firstly, the smallest a number can be is one digit and thus one byte. Secondly, all numbers are followed by either a vertical tab or a newline, also occupying one byte. Consequently, there is a maximum of one number per two bytes. This means that the number of numbers can be found by dividing the filesize by two. With this, it is simple to calculate the maximum needed space. The question that remains is then: is allocating (what is possibly) too much space faster than counting the number of lines and then allocating? In our benchmarks, the answer is yes. Our parsing function is about a third faster when guessing.

3.2 Unrolling the Parsing Loop

Unrolling a loop is a common optimisation technique where certain properties of a loop are exploited to reduce its overhead. This is a technique we have used in our `parseFile` function. Here we have a loop that iterates over each character in the file and parses them to BCD numbers. To best explain the optimizations done, it might be good to take a step back and look at the first version of the loop. See the below figure for a rough flowchart of the initial loop.

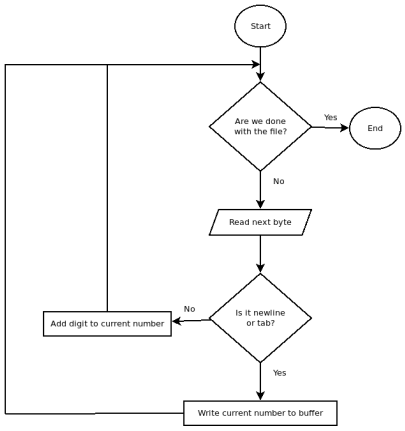

Figure 2: A flowchart roughly showcasing the first version of the loop.

While this is by no means a bad implementation, it does not exploit any of the properties unique to our problem. One such property is that all lines end with a newline and thus the file must also end with a newline. This means that we only need to check if we are at the end of the file when a newline has been read. Another is that that no number is longer than 5 digits – remember that the max value is 32767 – which means that after 5 digits either a tab or newline must follow. The last property we use is that the file must start with a number – unless it is empty[4] – and any tab or newline must also be followed by a number, if the end has not been reached.

Using all of these properties to our advantage leads to the final version of the loop. In this, the loop does not iterate over each byte in the file, instead it iterates over each number in the file, producing one number for each iteration of the loop. This means that, at the beginning of each

[4]We had actually entirely forgot to implement this case in our program until writing this paragraph.

iteration, the first byte we parse must be a number. Underneath the corresponding code from the `parseFile` function is presented (with some comments removed).

```
parse_start_of_number:
    movzbq (%rsi), %rax  # First digit into %rax, clearing it
    movq $1, %rcx        # Reset digit counter to 1
    subq $48, %rax       # Convert from ASCII to value
```

Then there can be up to four more digits after which we know either a tab or newline must follow. This is utilised by unrolling the loop for the next four bytes, jumping if any non-number is encountered. To do this we make use of the `.rept` directive repeat the same code four times.

```
.rept 4
    movzbq (%rsi, %rcx), %rdi # Read the next byte (zero extending)
    subq $48, %rdi       # Convert the ASCII digit to its value
    js parse_end_of_number # Jump if sign is negative (\n or \t)
    addq $1, %rcx        # Increase counter
    shlq $4, %rax        # Move 4 bits for the new digit
    addq %rdi, %rax      # Add the new digit
.endr
```

After this we must be at either a tab or a newline, so we store the number of digits in the lower byte (which we use when printing) and write the number to our buffer. Then we check if we are at the end of the file and if not, we start from the top again, knowing that the next byte must be the first digit of a number.

```
parse_end_of_number:
    # Save the count of digits in the lower byte
    shlq $8, %rax        # Shift out of the lower byte
    addq %rcx, %rax      # And save the count there
    # Save the number
    movl %eax, (%r11, %rdx, 4)
    addq $1, %rdx        # Increase the counter
    addq $1, %rcx        # Count the \t or \n we read
    addq %rcx, %rsi      # Move to the start of the next number
    # Have we read the entire file?
    cmpq %rsi, %r8
    jg parse_start_of_number
```

This concludes our optimisations of the parsing loop. We also tried some other optimisations, none of which improved the performance. One improvement we tried was having each iteration parse a coordinate, however, this only complicated the code and lead to a small decrease in performance. We also tried a rather complex scheme using SIMD where we would concatenate the lower nibbles of the characters, though this was also found to not improve performance. The difference between parsing unsigned integers and packed BCD numbers was also minimal.[5] As a final remark, it would be possible to improve the performance of the parsing by using threads. Each thread would be assigned a section of the file to parse and a place to write to. The reason we have chosen not to use threads is complexity and lack of time.

3.3 Sorting and Quirks of BCD

Our implementation of radix sort, found in the function `radixSort`, is mostly standard except for the aforementioned lack of strict radix. The reason behind this is best explained by examining what would happen were we to use a strict radix.

Let us start with the choice of radix. We want it to be reasonably high, so that the number of iterations made is kept low. We would also prefer for it to be a nice number in terms of bytes. If we want to stick strictly to digits matching bytes, then we have two choices 256 – corresponding to one byte – or 65536 – corresponding to two. The latter is very high and would likely not bring much improvement over a simple counting sort.[6] The former does seem to be a good fit, however. It will lead to three iterations of counting sort – the most a number will take up is three bytes – and it is very small likely leading to good cache usage. Even so, an improvement can be made. Inspecting

[5] It should be noted that we have tried to come up with optimisations using packed BCD – for example the SIMD solution mentioned – however, we have not been able to find any.

[6] Even if, by closer examination, the actual max value of two bytes of packed BCD is 39321.

the "digits" that a radix of 256 would give us, we find that the most significant digit will always be from zero to three. Simply extending the "digit" before to include these four values would save an entire round of counting sort. We have therefore chosen to implement this optimisation, although it introduces some complexity to the sorting. The two rounds of counting sort will be slightly different since we no longer use a set radix. We have used a macro in an effort to mitigate the repetition this leads to.

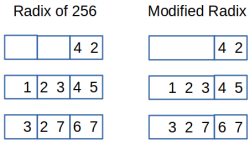

Figure 3: A figure showing the "digits" of three numbers in packed BCD using a radix of 256 and a "modified" radix.

3.4 Printing and SIMD Instructions

The printing, or more accurately, the conversion to ASCII found in the function `print`, is the section of the code that has been optimised most heavily. It is also the section with the furthest reaching optimisations, e.g., the change to BCD and saving the number of digits of each number. In total, the optimisations have made the printing about seven times faster than the initial version. On the other hand, the final is also quite complex and less intuitive. Nevertheless, this section will explain how it functions.

Let us start with a rough overview of the steps. We start with our numbers in packed BCD where each byte contains two digits. In ASCII each byte contains one digit, so we extract each digit into its own byte. Now, we convert each digit to its ASCII value by adding 48. Our number is now in ASCII so we simply move it into memory. We move our counter by the number of digits in the number – which we saved back when parsing – and repeat. This is of course a simplification – we are missing tabs and newlines and it does one coordinate at a time – but the actual code follows very similar steps.

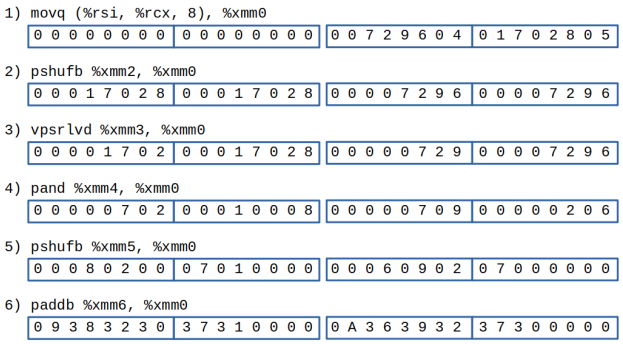

Figure 4: The instructions for the conversion of the coordinate (17028, 7296) from packed BCD to ASCII. The numbers shown are the contents of %xmm0 in hexadecimal after each step.

Above is a figure of the six instructions that loads a coordinate, (17028, 7296), from memory and converts it to its ASCII representation. This is the complex part of the loop in `print`, the rest is simply moving the result and common loop logic. Here follows a brief explanation of each instruction. 1) simply loads the coordinate from memory – here they are in packed BCD. 2) duplicates the numbers and drops the digit count. 3) bit shifts one copy of x and y 4 bits to the left. 4) clears the upper 4 bits of every byte – notice how the two quadwords now contain all the digits of x and y respectively. 5) shuffles the numbers into reverse order with one byte extra at the end – read the numbers backwards to see that they are correct. 6) adds 48 (`0x30`) to every number and tab (`0x09`) and newline (`0x0A`) in the empty bytes. It would seem that we have an issue with our y, 7296, which has gotten an extra 0 at the start and while this extra 0 will be written, it will be overwritten by the next number, that is, it is not counted as part of the number. This is the entire reason we keep track of the number of digits in our numbers – otherwise, we would end up with extra zeros or have to recount the size, neither of which are preferable.

4 Results and Discussion

By using standard command line utilities we can get the execution time our program and, by isolating the various parts of the code, the runtime of different parts of our program. We saw through analysing the data that the majority of the runtime was taken up by reading the numbers from the file and converting them from ASCII and by the conversion back to ASCII. The sorting part of our program was, in fact, the fastest part of our program. This is what lead to us switching to BCD and to the drastic optimisations of the conversion to ASCII.

4.1 Runtime on Different Instances

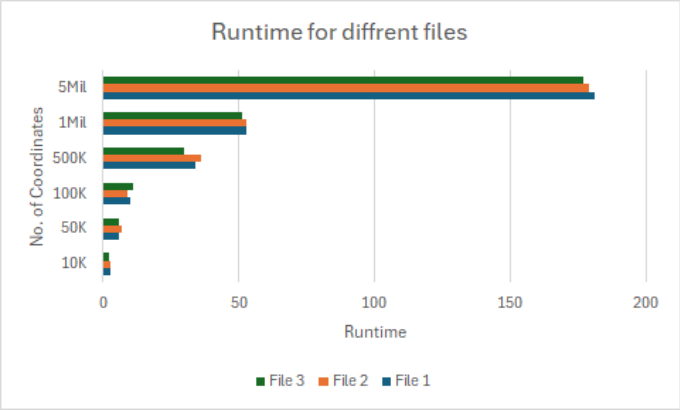

Figure 5: A graph showing the runtimes of our program on three instances of different numbers of coordinates.

We see here a graph that shows the runtimes of our program when run on different instances of a file full of uniformly random coordinates. We see that the runtime does not sway that much. While the runtime is dependant on the number of coordinates, all the files with the same amount of coordinates have similar runtimes. This is also supported theoretically, the runtime of radix sort using counting sort should not be any faster or slower on any input. We can also see that the runtime grows somewhat linearly with the number of coordinates, like it theoretically should. Strangely, it seems to grow somewhat slower than linearly, though this is probably due to some overhead becoming less significant as the number grows.

4.2 Comparison to First Version

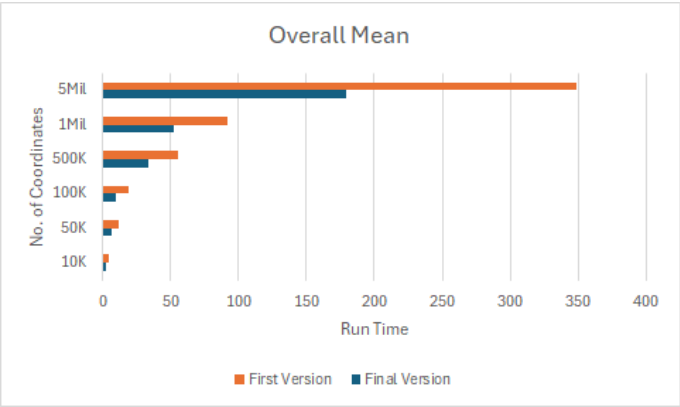

Figure 6: A graph showing the mean runtime of two versions our program on different numbers of coordinates.

Here, we see a graph showcasing the runtime of two versions of our program on different amounts of coordinates. The y-axis is the mean of how long the program took to run, measured in milliseconds. The x-axis is how many coordinates were in the files that was being sorted. The

runtime for each of the number of coordinates is the mean of running the program ten times on three different files, each distinct but containing the specified number of coordinates.

The first version is a version of the program that have the original parsing loop, uses counting sort, has a mildly optimised printing, and stores the numbers as unsigned integers.[7] The final version has the unrolled parsing loop, uses radix sort, stores the numbers using BCD, and uses the faster method of converting from BCD to ASCII. Here it is very plain to see that our various optimizations have lead to a faster program – especially for larger numbers of coordinates.

4.3 Runtime of the Program Parts

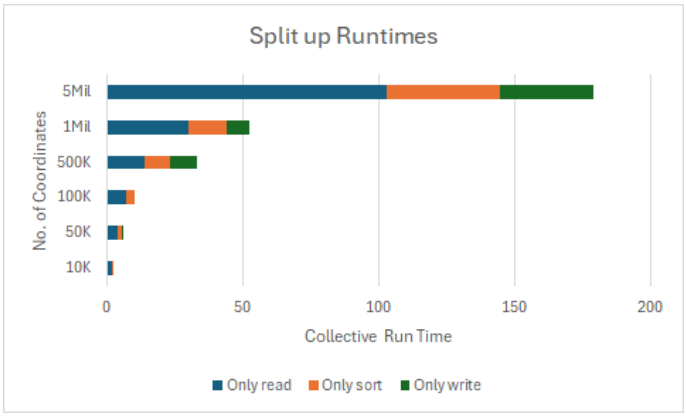

Figure 7: A graph showcasing the runtime of the different parts of the program.

Above is shown a graph over the runtime of the different parts of our program. We can see that the majority of the run time comes from reading the file and so that would be a focus should we bring the runtime down even further than we have now. We can also see that writing is the fastest part of our program. This is because of the BCD encoding we use on our numbers making it faster to convert to ASCII and write sacrificing some time in the sorting part making that slower.

We see that as the number of coordinates increases the runtime of the program also increases. The different versions of the program also have an impact on the runtime of the program with the final version having a better overall performance. We see that the difference in the times between the two programs increases as the file size increases. The optimizations have made it so that our final program runs about 20 percent faster on average than the first version of our program.

5 Summary

There are many ways to make a program faster or more optimized for its task. In this report we have show that, when it comes to finding ways to make a program faster, you need to look at the whole of the program and not just a single part of it. And what might slow down one part of the program may improve the performance in other places, leading to a net positive for the speed of the program. This we saw see when changing to the BCD encoding, our sorting got slower but our conversion from BCD to ASCII is considerably faster, making it so we gained more time than we lost. We also saw that it is important to choose the right algorithm for your goal. Because we wanted a fast sorting program for an input of any size, we chose counting / radix sort as it best fit our needs and goals for the project. Had we gone with another sorting algorithm – insertion sort for example – we might have had an easier time implementing it but we would also have been left with a slower program putting us further away from our goal. Overall, we would like to say that we succeeded in creating a program in x86-64 assembly on Linux that solves our chosen task – sorting coordinates – with a very reasonable performance. If we had had more time, we would have like to multi-thread the parsing so to improve the performance even further. But nevertheless, we are very pleased with our results.

[7] In truth, this is not the first version of the program. However, it does give a good indication of the runtime of early parts of the program.