INF236 Parallel Programming: Assignment 1

*by **Lars Erik Storbukås** (lst111@student.uib.no)*

Task 1

Solution file: `task1.c`

I started with the typical radix-sort algorithm, and starting to change obvious things like, from least significant digit to some type of least significant bit(s). I quickly found out that using masking on the power of to b to 2 minus one will give me a perfect masking after I'd performed the bit-shifting by b time (iteration * b) to extract the last b-bit(s) values from the 32-bit number.

In the end I found out that I had to change my way of finding the mask which was performed in this manner:

```
int mask = pow(2,b) – 1;
```

which ended up giving a lot slower running time since this was used multiple places in the code, this had to be replaced by:

```
int mask = (1 << b) – 1;
```

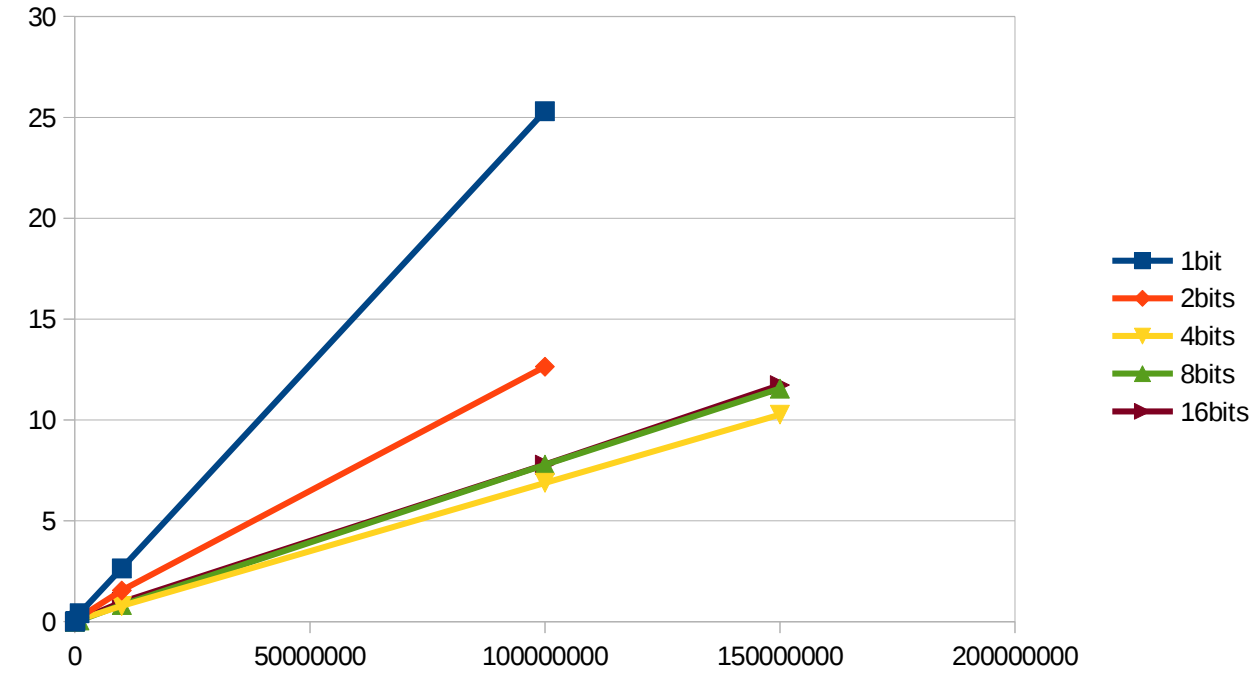
which gave me the exactly same result and was much more quicker than the pow()-function.

My final algorithm is able to manage around 225 000 000 random int numbers just under 10 seconds when 8 bit is interpreted as one digit (of course running sequentially) on my laptop with a fair Intel i7 processor.

Though I tried for some time to get the program to also accept input b to be 32, I had to focus on other parts of the assignment, since I think it would have taken more time than I should spend on this task and proceeded to work on the parallel task.

Task 2

We see that the running time is in the range 0 – 25 seconds and that the number of bits were able to sort with b bits under 10 seconds is approximately 150000000 (which works for 4,8,16 bits).



Sequential

n \ b	1	2	4	8	16
1000	0.000337	0.000173	0.000093	0.000051	0.000621
10000	0.003602	0.001831	0.000934	0.000487	0.001055
100000	0.036881	0.018548	0.009487	0.005117	0.008301
1000000	0.416813	0.209865	0.105428	0.056790	0.091511
10000000	2.648258	1.549873	0.780986	0.833718	0.975879
100000000	25.307088	12.639077	6.885782	7.785478	7.787112
150000000			10.265871	11.556057	11.732726

The running time of your typical radix-sort is **O(b * n)** where **b** is size of digit. Since our program is based on bits representing a digit, our **b** is **32** (*bits*), but it's also divided by how many bits you want to represent a digit. So our running time is equal to **O((32/ b) * n)**. Since in our case the k is so small, we can consider the algorithm to be **O(n)** or in other words *linear*.

This conclusion of a linear running time is backed up from the output data, and shown in the graph.

Task 3

Solution file: `task3.c`

I started this task with a clear vision of what I needed to implement in my previous solution to make it run parallel: *parallelize the radix-function()* in the code. After many different versions of trying to get this working correctly, I realized this couldn't be the correct way of implementing it. My results from this was either not substantial to even call it speedup (maybe a second on really large numbers). So I started thinking in another way, and used both the Internet and previous lecture slides to give me ideas on how to get the “correct” solution.

The way I ended up implementing parallel radix-sort, was by *splitting up the actual number input into equally sized chunks* (with maybe a few more in the last chunk due to n%threads) and then running the loop for bit-shifting from the *main loop (for 0-32 with b increments*). So then for each iteration of a new bit-shift value (used with the mask to retrieve the next bit-value we want to sort), the algorithm splits and parallelizes the chunks all based on this bit-shift value, and then when all the chunks, individually are sorted correctly in regards to the current bit-shift value, all you have to do is merge the result by *pulling out all the buckets from each chunk in ascending order* (so for example: first take all the '01' buckets from each chunk, before you take all the '10' buckets from each chunk, and so on).

During my work on this task, one of the most difficult things to wrap my head around was how I was supposed to merge the buckets back into the main array/pointer of data without increasing the time so that my parallelized version ran just as slow as my sequential version. I tried a lot of stupid things to get out the correct data from the chunks before I realized that I needed to *send pointer-pointers in to the radix-function* so that I would be able to retrieve information like local bucket prefix in the chunks and also the local bucket size in each chunk. After this was implemented, I was able to do some simple calculation of which memory address in the 'dummy' array should be pulled/copied back into the data array (did this using *memcpy()-function*).

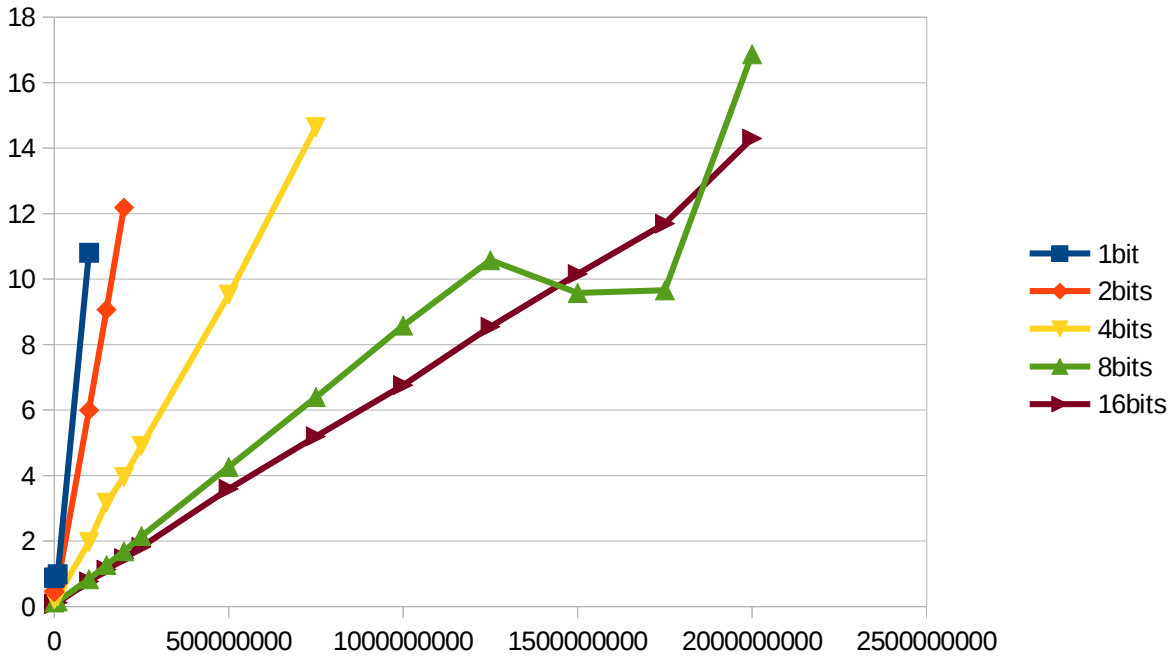
The solution I ended up with works surprisingly well, and gives a pretty decent speedup compared to my sequential solution (see graph/table in task 4).

I alternate the number of cores by using this command:

```
export OMP_NUM_THREADS=40
```

Task 4

We see that the running time is in the range 0 - 17 seconds and that the number of bits were able to sort with b bits under 10 seconds is approximately 1750000000 (which works for 8,16 bits).



Parallel (threads 40)

b	1	2	4	8	16
b					
1000	0.870831	0.450867	0.225458	0.117838	0.084767
10000	0.872527	0.445712	0.228883	0.115556	0.083572
100000	0.890267	0.448446	0.225631	0.115942	0.081172
1000000	0.859269	0.465116	0.232012	0.119172	0.085462
10000000	0.985897	0.705004	0.312460	0.159013	0.120125
100000000	10.802773	5.996789	1.967471	0.832247	0.765357
150000000		9.068662	3.171984	1.264037	1.125301
200000000		12.181648	3.953206	1.685068	1.475074
250000000			4.905119	2.147791	1.820741
500000000			9.523975	4.261344	3.593965
750000000			14.643703	6.395768	5.195709
1000000000				8.577041	6.759318
1250000000				10.577812	8.546984
1500000000				9.573353	10.156055
1750000000				9.656471	11.692796
2000000000				16.867722	14.295108

The algorithm still performs in linear time when it's parallel. So **O(n)** running time, but you can see that it correlates in a stronger way to the number of bits to be interpreted as one digit than the sequential algorithm. You can also notice that the running time for small numbers in the parallel algorithm is more or less the same value, this is due to all the overhead and sequential work in between the parallelization of the algorithm (and possibly the memory allocation within the parallelization).

Here is a small table showing difference between the running time of parallel merge sort and parallel radix sort with 16 bit (interpreted as one digit).

n	Merge	Radix
100000000	4.915968	0.765357
200000000	9.013584	1.475074
300000000	14.492550	2.273581
400000000	17.638451	3.007567
500000000	21.341792	3.893426

And we can see that Radix-sort performs way better than the Merge-sort algorithm when it runs in parallel (compared the Radix-sort algorithm has a speedup of roughly 6 compared to Merge-sort algorithm).

The speedup of the comparison between our sequential radix-sort and parallel radix-sort algorithm of the 16-bit is show below:

n	Radix-seq	Radix-parallel	Speedup
100000	0.008301	0.081172	0.102264328
1000000	0.091511	0.085462	1.0707799958
10000000	0.975879	0.120125	8.1238626431
100000000	7.787112	0.765357	10.17448328
150000000	11.732726	1.125301	10.426300163

We can see that from 10000000 elements and upwards it stabilizes at a speedup of about 10, which is fairly good considering there is some overhead in between the parallelization.

The table and graph-data is located in the document: `graph-data.ods`