Michigan State University

CMSE/CSE 822 – Parallel Computing **Fall 2019**

Homework 6

Alexander Harnisch

Parallel Bucket Sort with MPI

Part 1

General comment: In the instructions it says "These are the default names in the git repository. It is essential that you do not change the directory structure or file names.". However, that is not the case. I provided the files as asked for, and additionally a file called *bucket_sort_v2_squared.c* which is the version used in Part 2. If you don't pull that one and run my makefile without specifying the target, it will fail. My all directive also tries to build the squared version.

1.

See code in repository.

2.

Figure 1 and Figure 3 shows the speedup and efficieny for *bucket_sort_v1.c* and *bucket_sort_v2.c* respectively. The base case is the one using one full socket with 14 cores. The total and partial execution times can be seen in Figure 2 and Figure 4 all parts of the code, that are executed on multiple cores are reported as averages in these plots. The plots have many interesting features, but since you don't ask for a description I won't type it out. The only question the assignment asks is:

Which phases in your code are the root causes of the performance difference that you observe between your bucket sort implementations?

The root causes are the generation, binning and distribution phases. Everything else is unchanged after all. *bucket_sort_v2.c* is much faster for large n because the binning and generation are also parallelized. However, as always, that creates additional overhead (in this case especially in distribution time) that is too expensive for small problem sizes. We can clearly see that by comparing the execution time for the smallest problem size between the two versions.

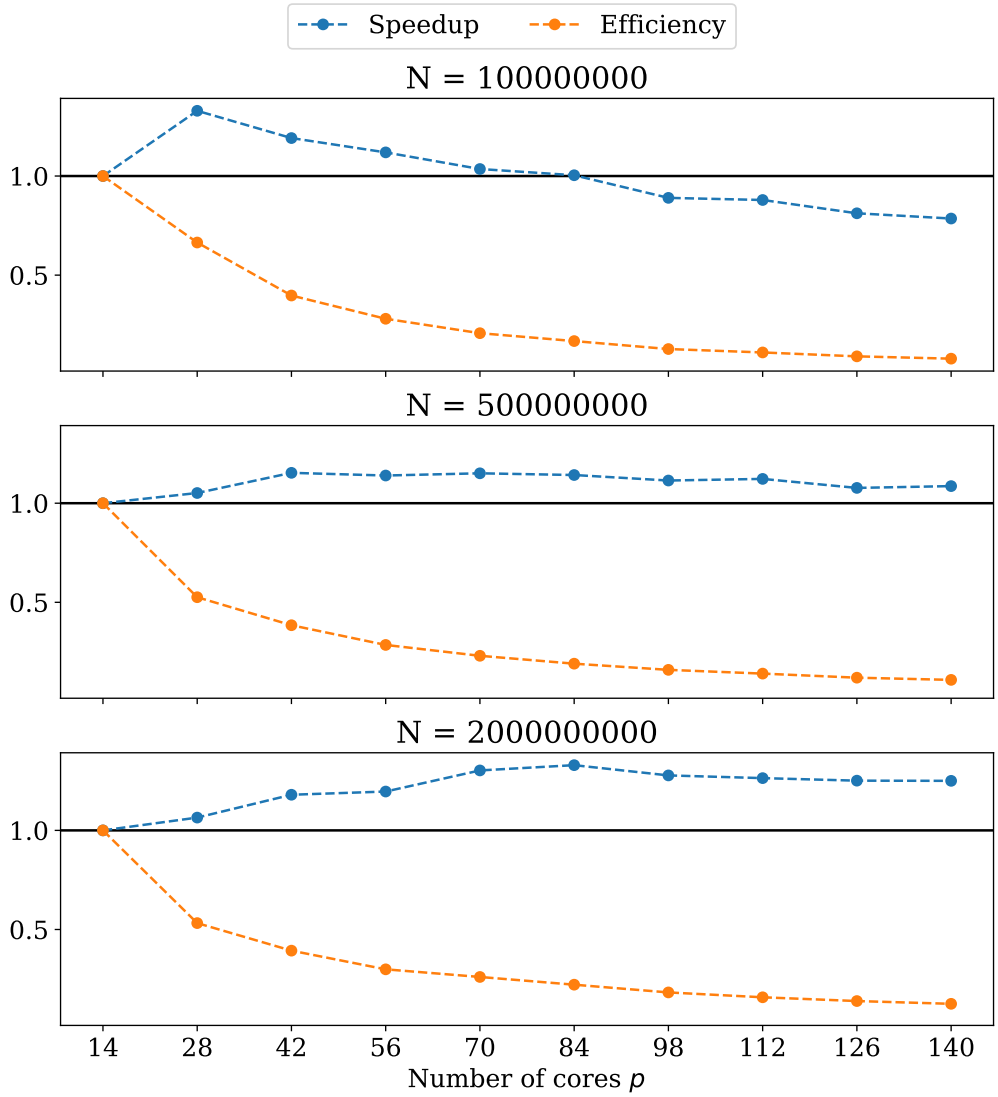

Figure 1: Speedup and efficiency of *bucket_sort_v1.c* for different number of cores p.

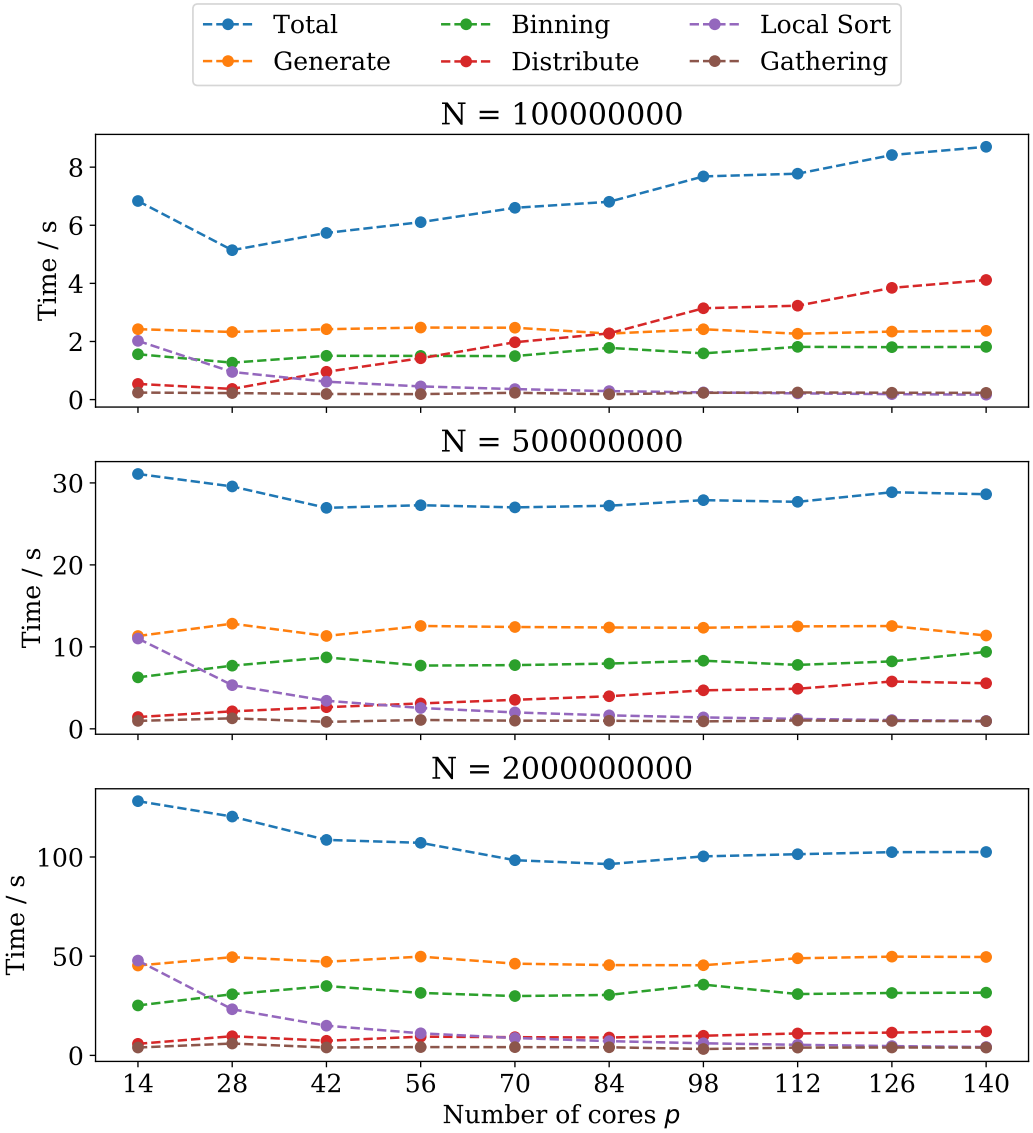

Figure 2: Execution times of *bucket_sort_v1.c* for different number of cores p.

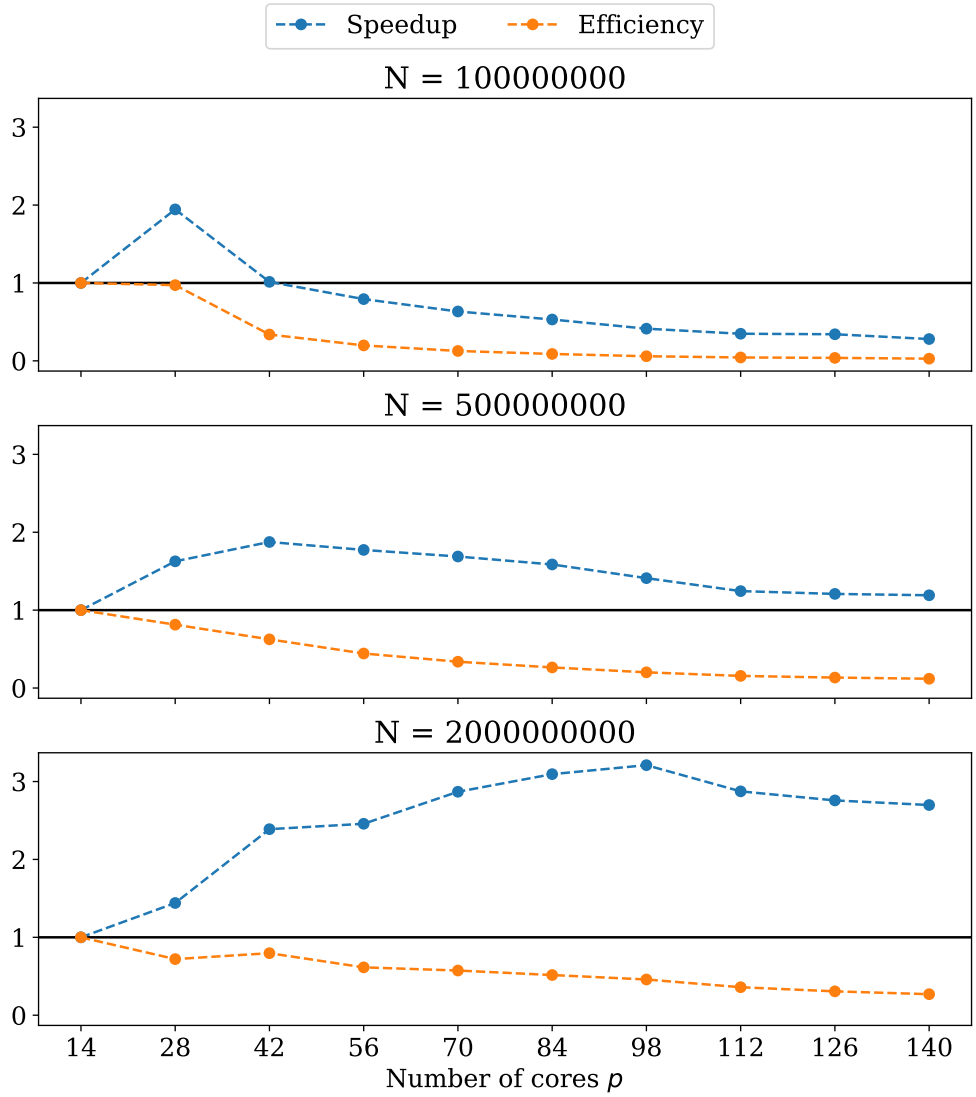

Figure 3: Speedup and efficiency of *bucket_sort_v2.c* for different number of cores p.

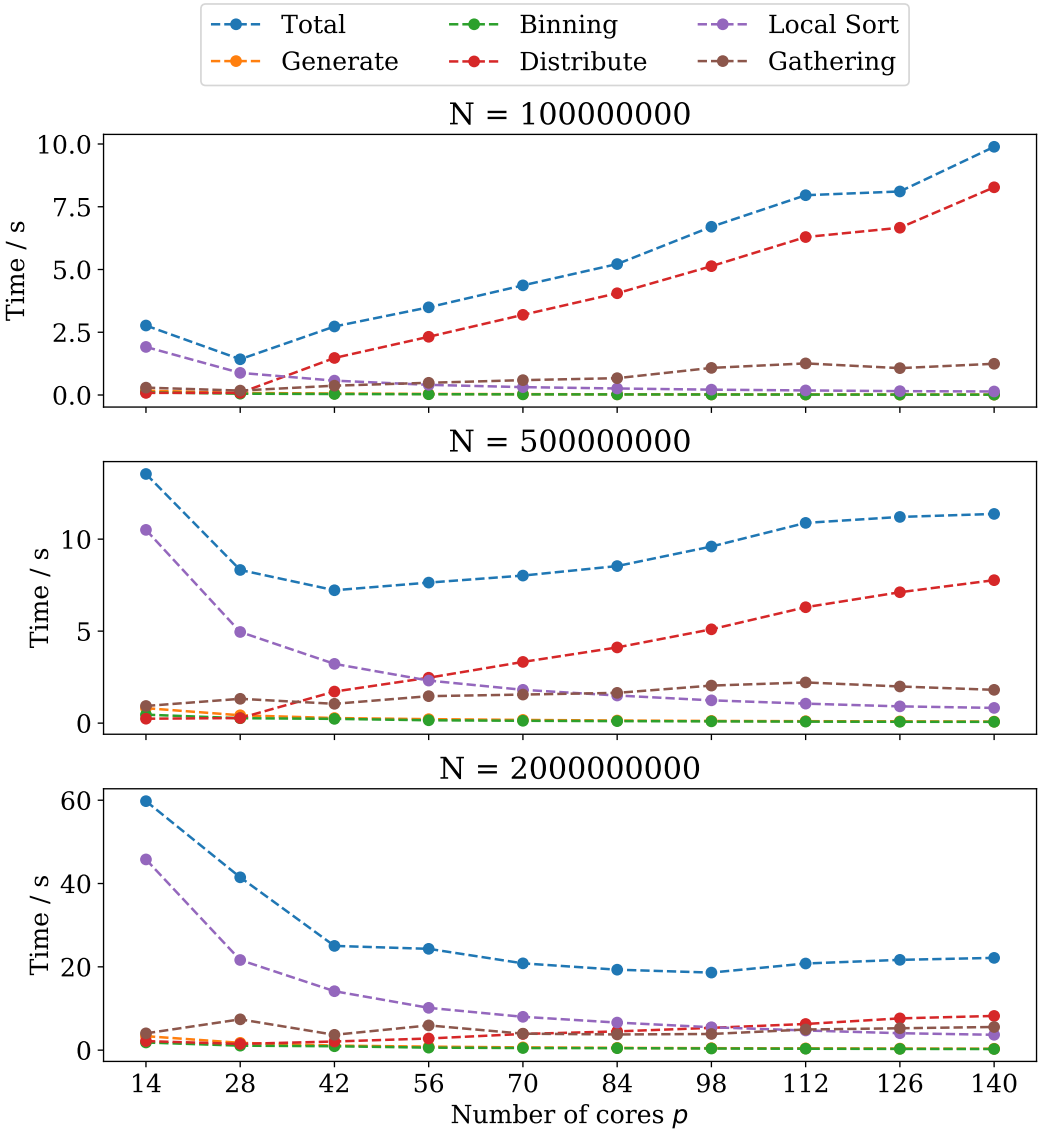

Figure 4: Execution times of *bucket_sort_v2.c* for different number of cores p.

Part 2

1.

I find plots generally a lot more helpful than tables, so I went with that. Figure 5 and Figure 6 show the load balance in terms of spread in execution time for the uniform and squared versions respectively. Comparing the two clearly proofs we have a load imbalance problem: The spread between the minimal, average and maximal execution times is significantly larger for the squared distribution. This is of course the case because smaller numbers are quadratically more likely there, resulting in quadratically less work for higher ranked processes. The speedup and efficiency curves for the squared distribution can be seen in Figure 7 and the partial execution times are shown in Figure 8.

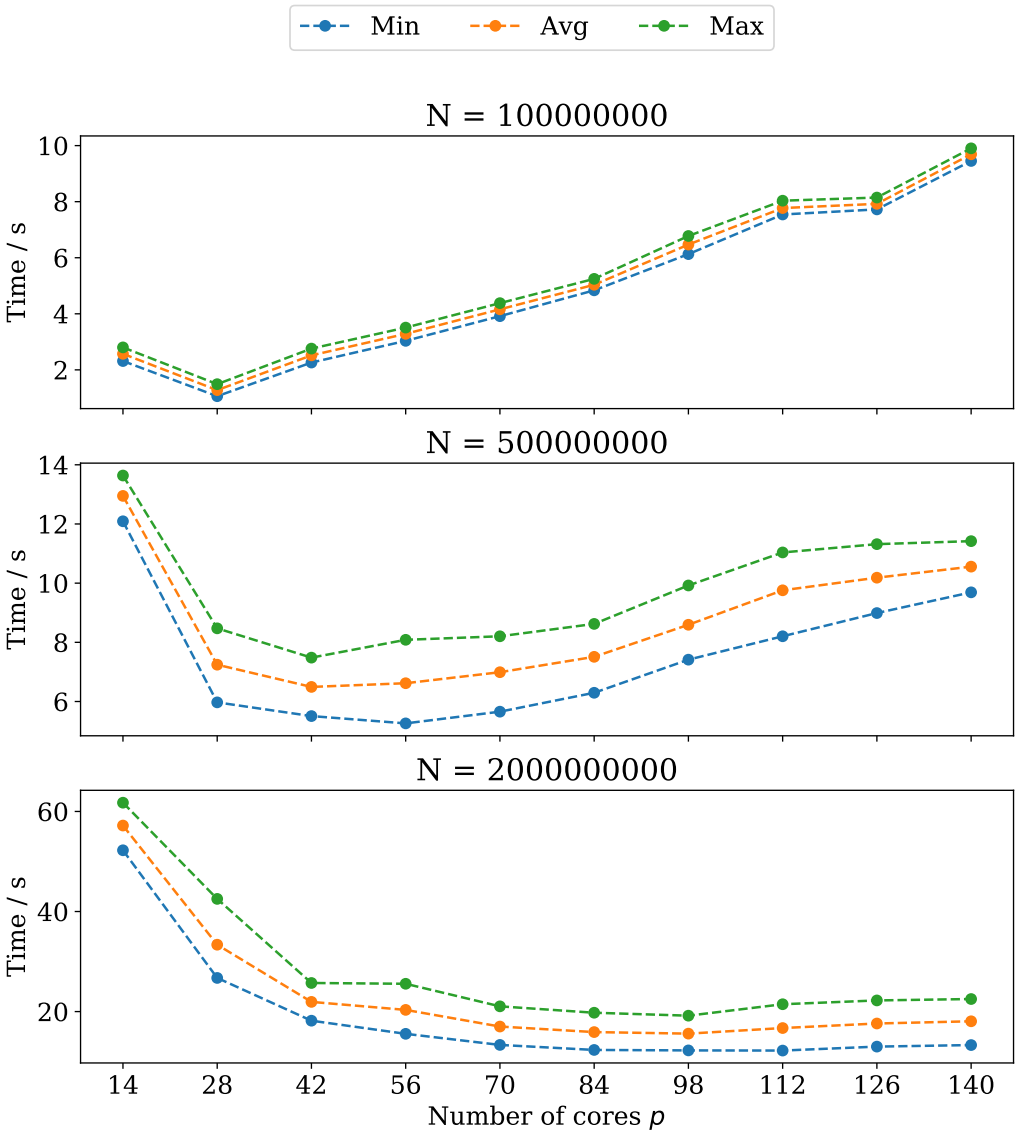

Figure 5: Load balance of *bucket_sort_v2.c* for different number of cores p.

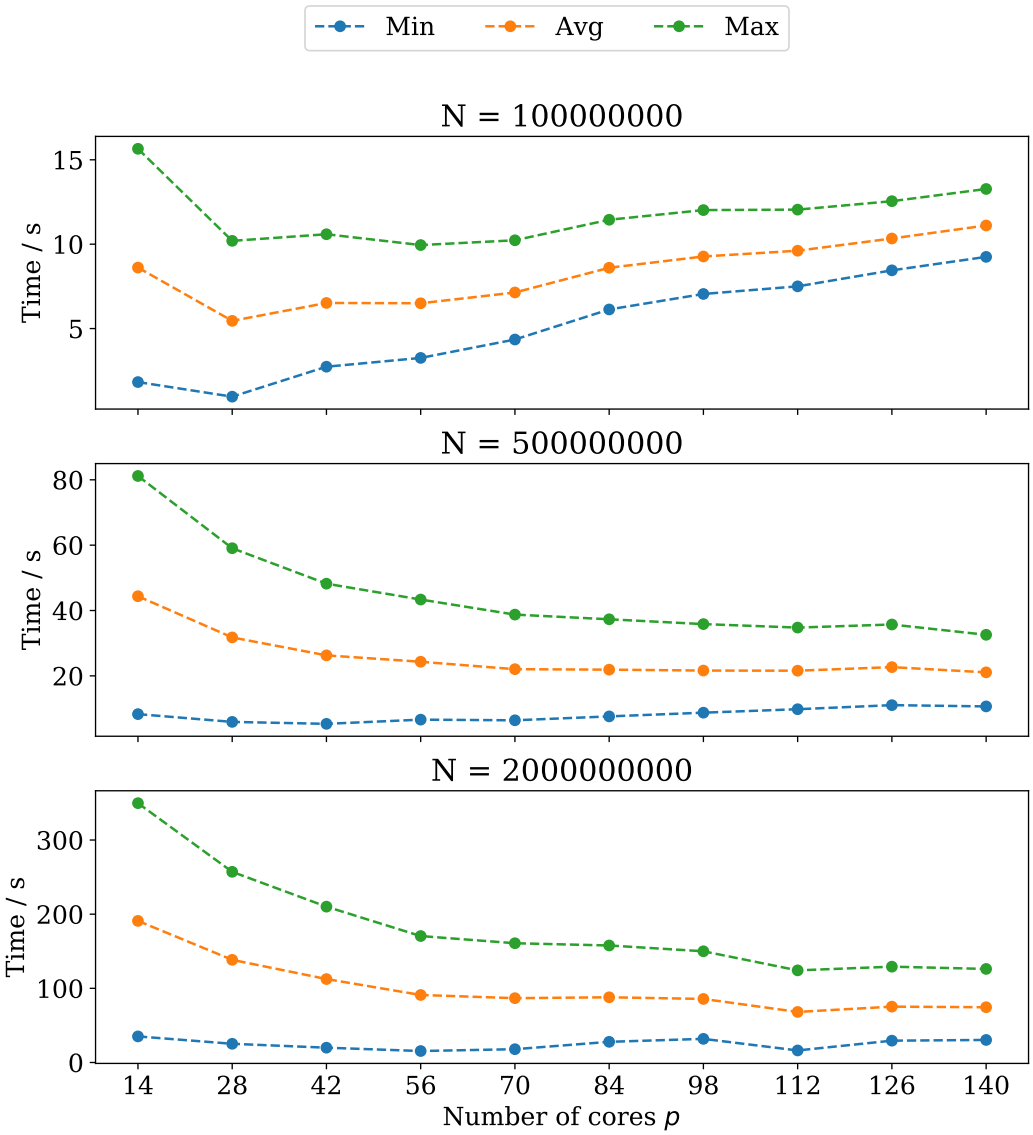

Figure 6: Load balance of *bucket_sort_v2_square.c* for different number of cores p.

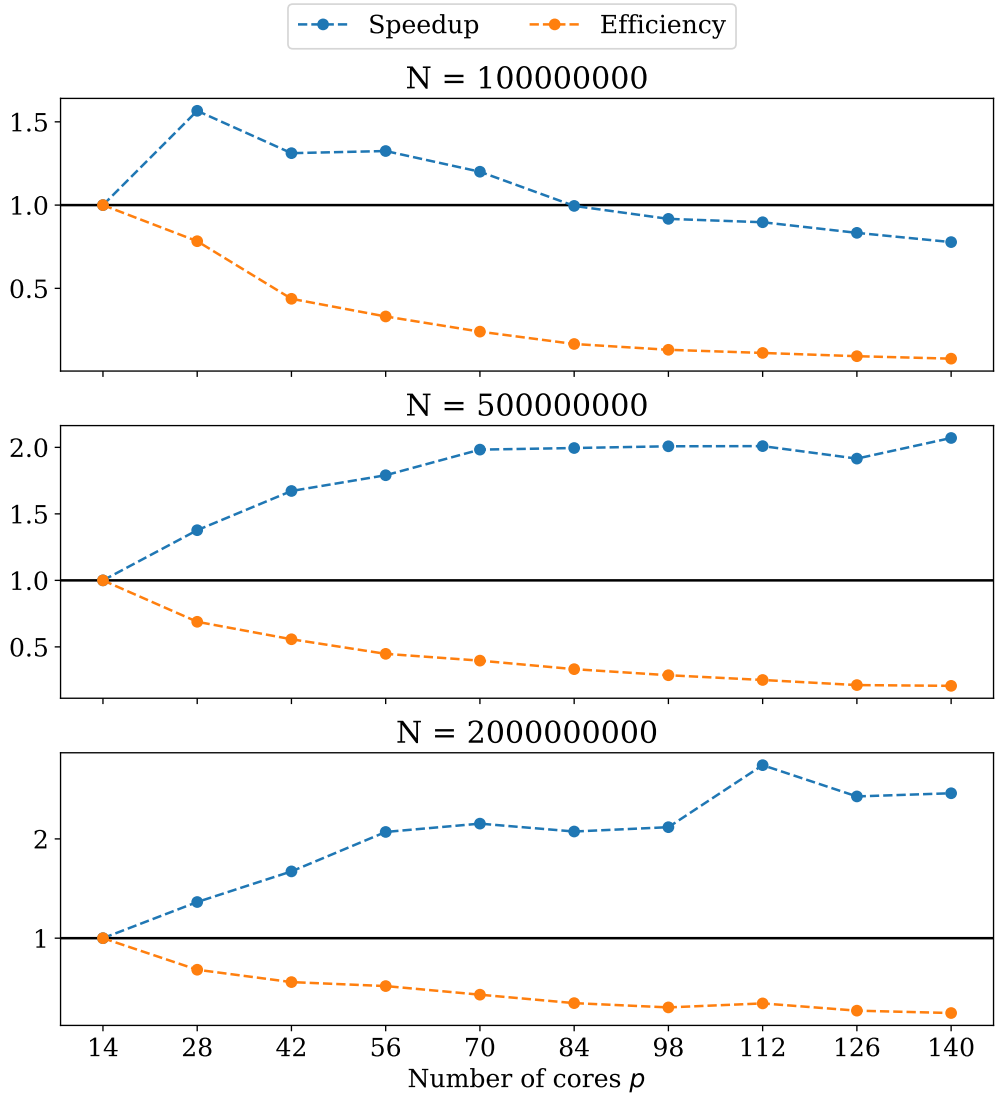

Figure 7: Speedup and efficiency of *bucket_sort_v2_square.c* for different number of cores p.

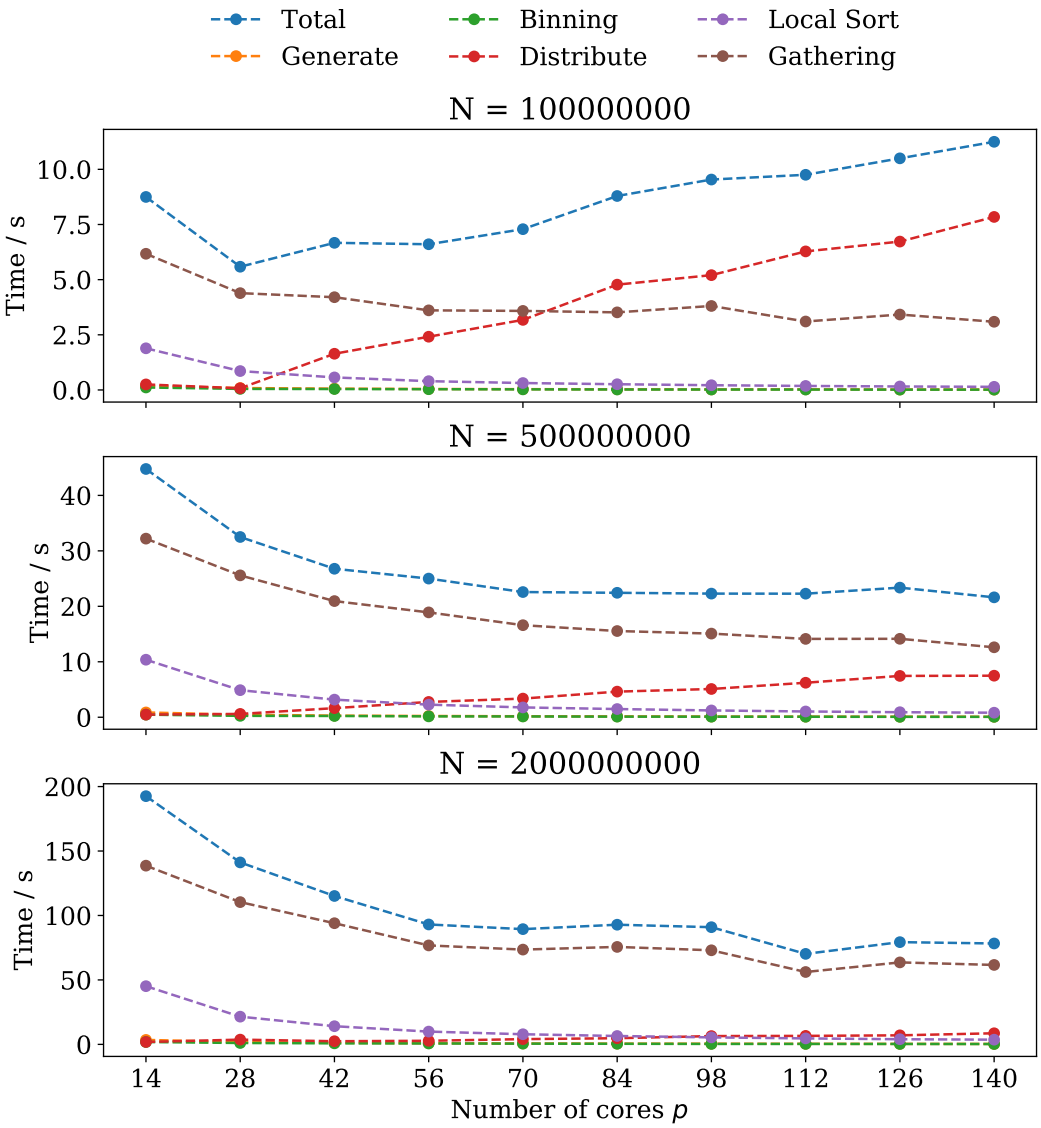

Figure 8: Execution times of *bucket_sort_v2_square.c* for different number of cores p.

2.

See code in repository. For some reason there is no 3. so let's continue with 4..

4.

I assume here you mean speedup over the v2 squared distribution case, which is provided by Figure 9. We can clearly see a large speedup, which indicates we remedied the load imbalance problem. I also provide a speedup and efficiency curve with using one socket with v3 as base case in Figure 10, which is not really helpful here though. It is pretty much the same as for v2 which just tells us it scales the same way, which makes sense.

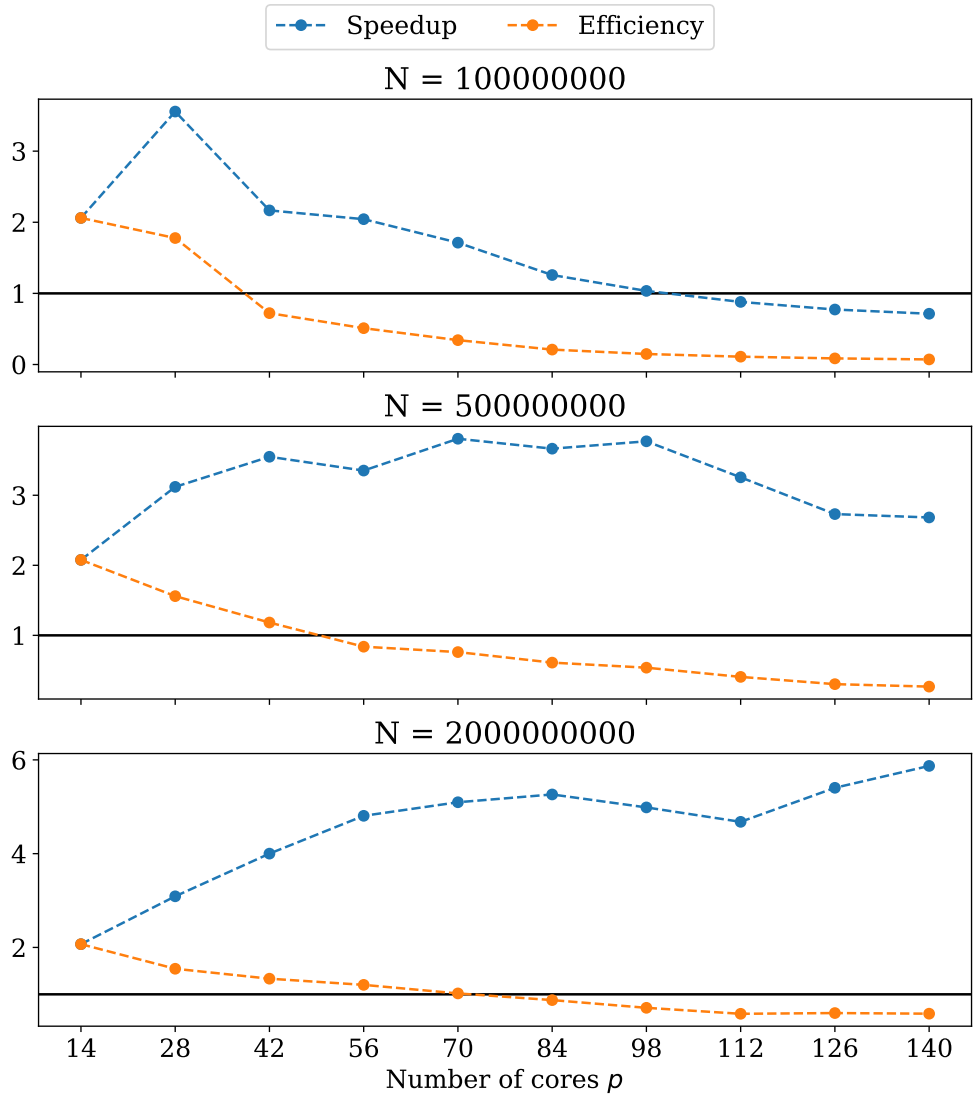

Figure 9: Speedup and efficiency of *bucket_sort_v3.c* for different number of cores p with $p = 14$ of *bucket_sort_v2_square.c* being the base case.

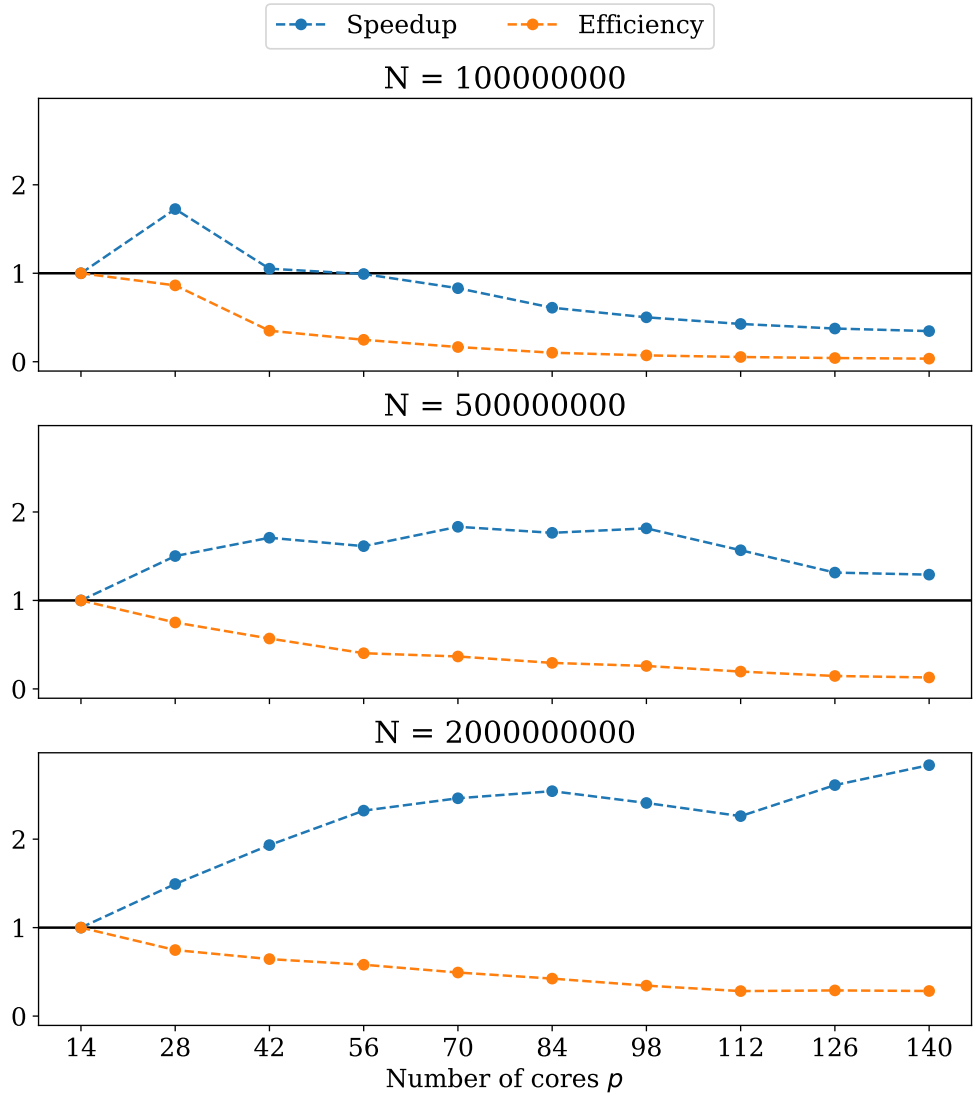

Figure 10: Speedup and efficiency of *bucket_sort_v3.c* for different number of cores p.

5.

The time spend in different parts of the program is shown in Figure 11. In all cases, selecting the pivots does not take any significant amount of time (about one second). This includes the communication time.

Taking all this into account: Yes, generally the time spent in selecting pivots is absolutely worth it. Once again, only for small problem sizes the overhead can become a problem (see Figure 9).

Additional comment: Obviously here it does make a lot more sense to just statically rescale the bin sizes instead of dynamically choosing pivots. However, I get the point you want to teach here, this only works in case one knows the actual distribution. For highly variable input data with unknown distribution the pivoting method will generally be of advantage.

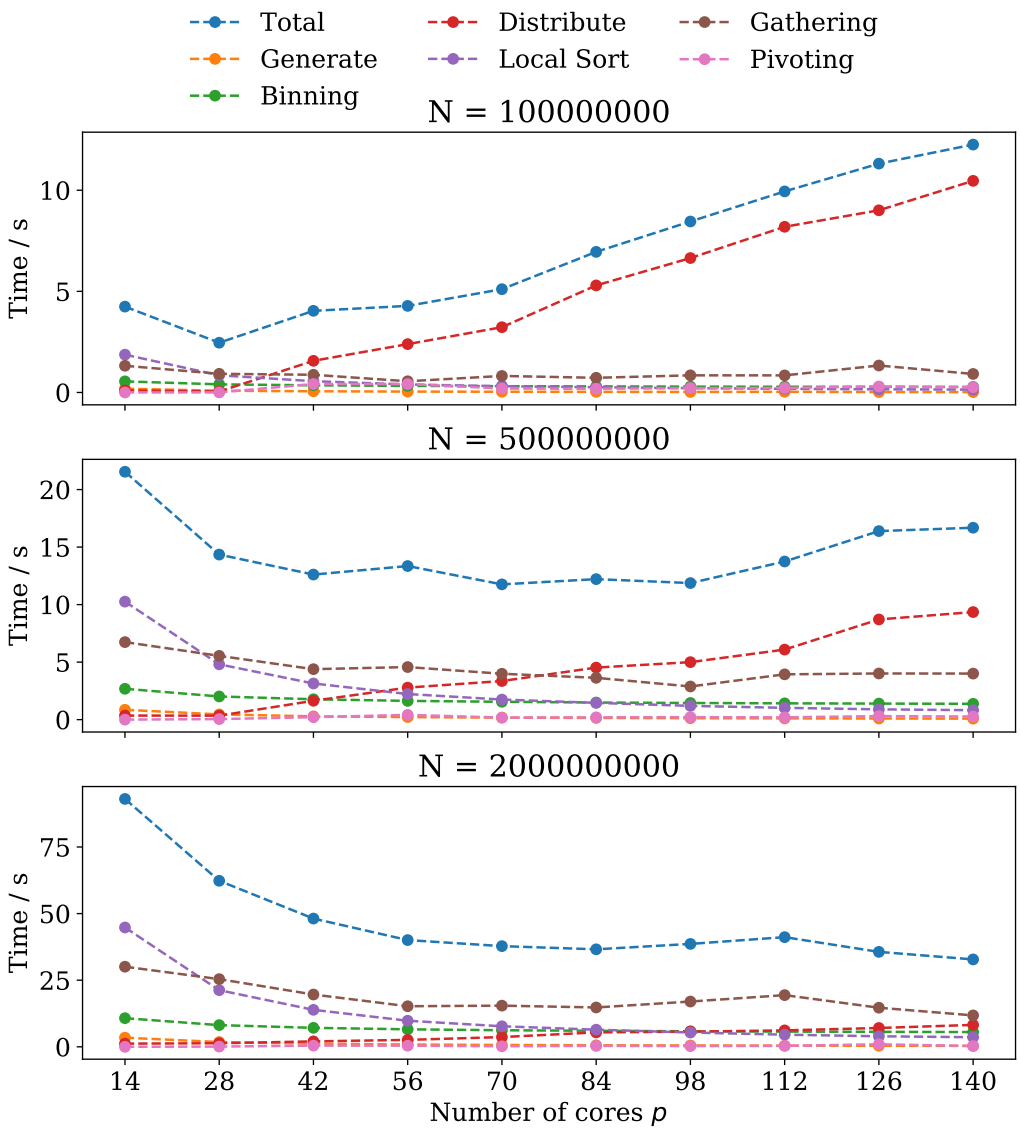

Figure 11: Execution times of *bucket_sort_v3.c* for different number of cores p.