Parallel Computing - Homework 3

Oleguer Canal Anton
Raphaël Widdershoven

March 2020

1

1.1

We can use N processors to compute each r_i. Having the set of processors: $\{P_1^i, P_2^i, ..., P_N^i\}$ we can assign each comparison $a_j < a_i$ to P_j^i. We thus have that aech of these processors just gets two elements to compare. If we assign to 1 the condition $a_j < a_i$ (0 otherwise), and perform a reduce operation with recursive doubling to compute the sum of the N elements all processors $\{P_1^i, P_2^i, ..., P_N^i\}$ will hold r_i position of i-th element in $\log(P = N)$ time.

1.2

In total we need $P = N^2$ processors: $\{P_1^i, ..., P_N^i\}, i = 1 : N$. The best sequential time is $O(N \cdot Log(N))$, the parallel time is $O(log(N))$ (recursive doubling time), we thus have a speedup of $S_p = \frac{N \cdot Log(N)}{log(N)} = O(N) = O(\sqrt{P})$. As we have N^2 processors the efficiency is $1/N$, which is really bad.

1.3

We would not use this algorithm in practice. The assumption of having that many processors with a shared memory is unrealistic. Just think already that a list of 1000 elements can perfectly be solved on one processor quickly and would need 1.000.000 processors sharing information. Not to mention the time to initially distribute the data.

2

The code is not correct, here we list some mistakes:

- The code does not take the boundaries into account. Because of this, process 0 will try to communicate with process -1 and process P-1 will try to communicate with process P. Process -1 and P of course don't exist and this will rise an error.
- Each process first sends (blocking send) and secondly receives. This will deadlock. The communication should be alternated so for example the send and receive in the else clause should be inverted.

3

3.1

The program was implemented in C and can be found in the submission.

N is given as a command line argument. Each processor then generates a sub-array of $\lceil N/P \rceil$ or $\lceil N/P \rceil - 1$ elements with its own seed. It is programmed in a way that each processor will always have an equal number or 1 more element than all processors with higher rank to keep the data balanced. We tried to minimise

memory usage and thus each process only stores at most $3 \cdot \lceil N/P \rceil$. This memory space is used in the compare phase. Each processor needs to hold its list of elements, the list of elements of its compared neighbour, and a third array to store the selected elements. The copy of elements from one array to another is avoided by swapping the array pointers after comparison.

It is important to notice that the array comparison depends on whether processor p is exchanging information with $p-1$ or $p+1$. If its receiving a sorted array from $p-1$ it has to keep the half largest elements of the merged array. Analogously, if exchanging with $p+1$ the smallest ones. In addition its important to take into consideration that when receiving information from a larger process, the array might have one element less than current process array (analogous in the opposite direction).

The odd even sort algorithm then just loops just as in exercise 2. It alternates between sending and receiving left and right and then compares both arrays to only keep the pertinent largest or smallest elements. This loop must be run $P+1$ times where P is the number of processes. In the slides is written that P is enough, but this is only if all processors hold the same amount of data. An example of this can be seen when attempting to to sort the arrays [6,5,4,3,2,1] or [5,4,3,2,1] on 4 processors. 5 steps are needed even though 4 processors are used.

The program was tested on a different number of processors for different amounts of data (always below 20 because it was tested manually). Results seem promising as it always worked even when P didn't divide N.

3.2

Figure 1 shows the speedup for different amount of processors and data sizes. Each timing was computed 5 times on Tegner through a batch job and the median value was taken. The time was measured with the time command in Linux, so the startup of all processes and the execution of the whole program was taken into account. The code was compiled with optimisation flag "-O3".

The most interesting part is the left side, where few processors are being used. For a big amount of processes more data should be used to have more meaningful speedup measurements. The sequential time is based on C's built in algorithm to sort (Quick Sort) and was around 10s, 20s and 40s respectively for the three curves.

Notice the logarithmic function shape that figure 1 presents, which was the expected speedup (as given in the slides). Time results seem very satisfactory as, for example, for 10 processors the efficiency is still above 0.5 (even larger for larger amounts of data). If we had an enormous problem, tackling it with parallel processors would really be a more feasible solution. As an example, if we wanted to sort 1e10 elements, which may take around 2 hours on a single processor, we can expect to solve it in 20 minutes or less on 10 processors using the implemented Odd-Even Transposition Sort.

What is also interesting to see is that more data improves the speedup for the same amount of processors. This is mostly because if we have more data the longest part becomes the initial sorting on each processor separately and this is purely parallel (speedup 1). The part that decreases the speedup is then the exchanging and comparing but this only increases linearly with the amount of data, while the initial sorting increases with O(N log N).

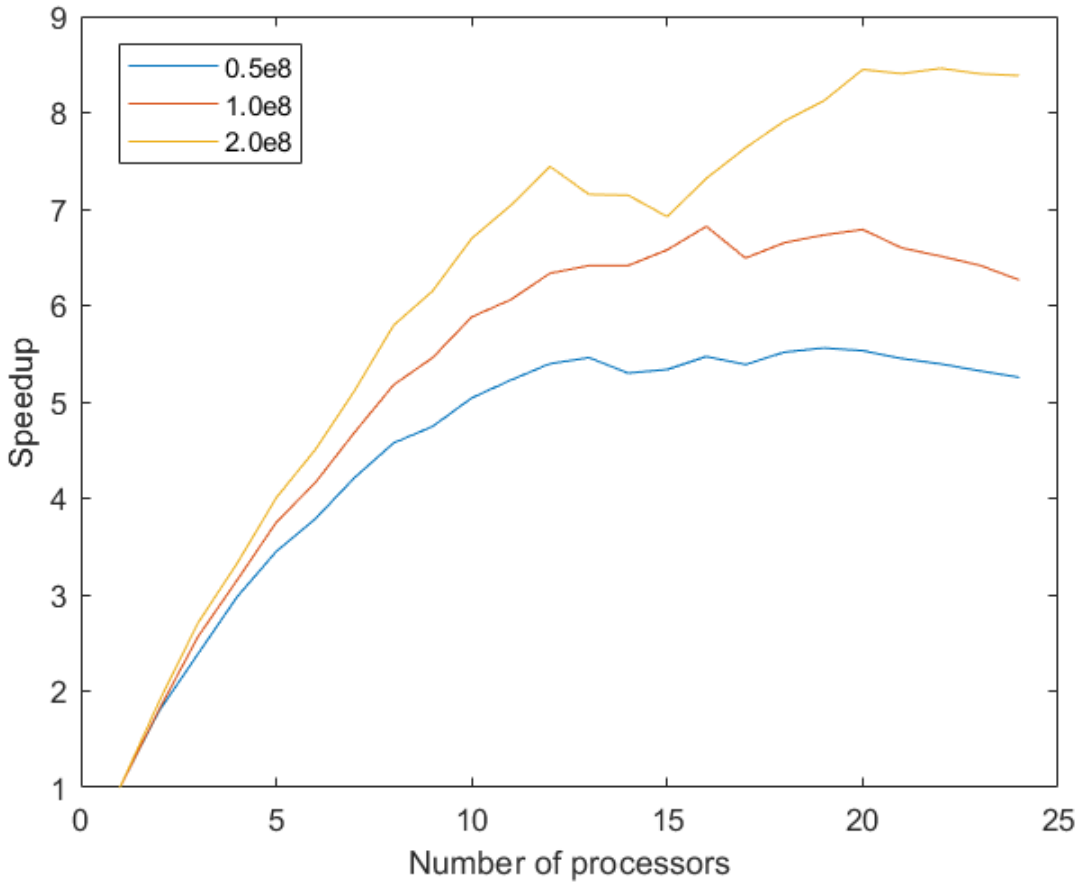

Figure 1: Speedup presented by using different number of processors for different sizes of data. Notice that, in this case, using more than 20 processors does not significantly improve speedup as communication and comparison begin to take a bigger proportion of the time.