UPPSALA
UNIVERSITET

Parallel and Distributed Programming
Parallelized Shearsort

Author:
Victoria Sedig

Uppsala
June 5, 2021

1 Introduction

1.1 Background

In computer science it is useful to have programs that perform as fast as possible to not cost time. To parallelize a code is a good way to improve the efficiency as a computer often has more than one core. In this project a parallel shearsort algorithm was developed.

1.2 Problem Description

A shearsort algorithm is a sorting algorithm that sorts elements in a matrix. The elements are sorted 'snake-wise' as seen in the before and after matrix.

$$\text{Before: } \begin{bmatrix} 12 & 2 & 7 & 84 \\ 90 & 70 & 28 & 70 \\ 60 & 13 & 35 & 46 \\ 100 & 20 & 30 & 2 \end{bmatrix} \text{ After: } \begin{bmatrix} 2 & 2 & 7 & 12 \\ 30 & 28 & 23 & 20 \\ 35 & 46 & 60 & 70 \\ 100 & 90 & 84 & 70 \end{bmatrix}$$

In this project the matrix is of size $n * n$ with N elements. The algorithm is as following looped $ceil(log2(n)$ times:

1. Sort odd rows in ascending order
2. Sort even rows in descending order
3. Sort all columns in ascending order

[1]

2 Method

2.1 Solution Approach

The first step of the algorithm is to read the data. This is done in processor zero whom will read the data row-wise into an array. The numbers are integers between 0 and 100. Integers are used due to simplicity. In the whole code all data will be handled in one dimensional arrays to make the communication easy. After the data is generated the data is scattered so that each processor gets equal amount of columns. It is assumed that the matrix size

n is divisible with the number of processors. When all processors has their data the odd and even rows are sorted as described in section 1.2. The data is sorted using quicksort because it is a well performing sorting algorithm. Then the data is sent to be distributed column wise over the processors. The columns will be sorted in ascending order. After step 1, 2 and 3 (from section 1.2) is done the data is sent back to the row-wise storing so that the loop can start over. The steps are looped ceil(log2(n)) times. After all loops are done the data is gathered by process zero whom will have the final result.

Only one dimensional arrays are used in the program. That is because it is easier for memory allocations and simplifies the communications.

2.2 Verification of Correctness

To verify the correctness a small loop was made which went through all the elements in snake-wise order and checked that the numbers where are larger or equal to the previous element for each step. This was done for different matrix sizes and different number of processors.

2.3 Experiments

The tests were run with bash files on the snowy cluster. There were tests with growing numbers of processors (to give speedup). The same thing was done but with rising number of data and number of processors to see weak scalability.

3 Results

3.1 Strong Scalability

In Table 1 and Figure 1 the result for the speedup is seen. The number of processors is put to values were the matrix size n is evenly divided by the number of processors.

Table 1: The time and speedup for increasing numbers of processors. The size of the list was constantly $n = 3600$. The speedup was calculated with the sequential time $t_{seq} = 174.3272$.

# Cores	Time	Speedup
1	174.3272	1
2	88.4786	1.9703
3	60.0021	2.9054
4	45.3831	3.8412
5	37.5727	4.6397
6	31.2241	5.5831
8	23.6936	7.3576
9	21.7649	8.0095
10	19.6118	8.8889
12	16.6238	10.4866

# Cores	Time	Speedup
12	16.6238	10.4866
15	13.9627	12.4852
16	13.0316	13.3773
18	11.8156	14.7540
20	10.8698	16.0377
24	9.4376	18.4715
25	9.1926	18.9639
30	8.0521	21.6499
36	7.4871	23.2837
40	7.3679	23.6603

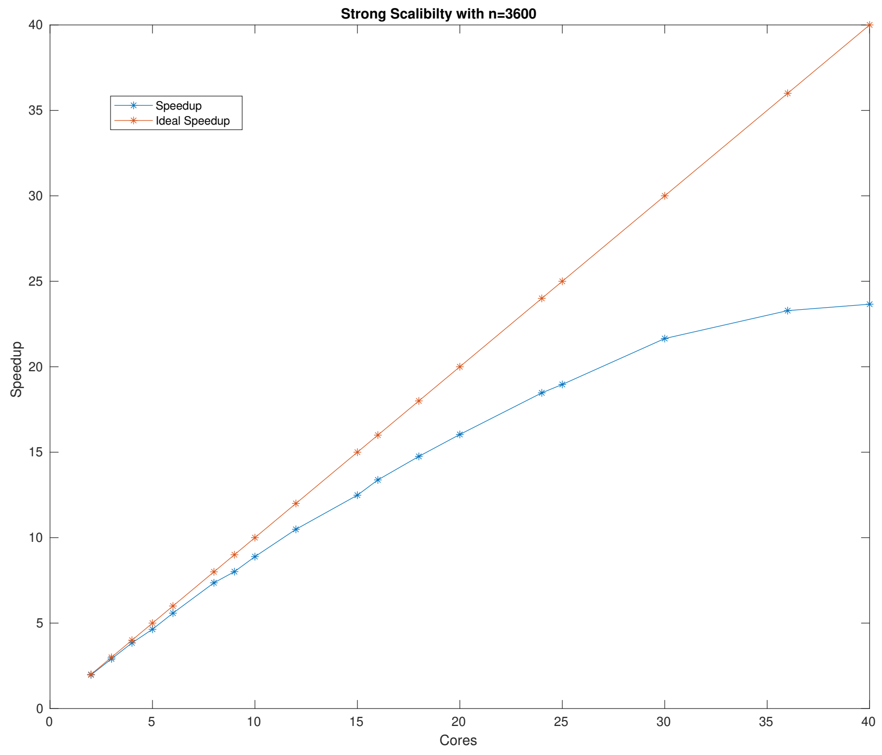

Figure 1: The speedup and the ideal speedup for the parallelized program plotted against the number of cores used. The size of the matrix was kept constant at $n = 3600$.

3.2 Weak Scalability

In Table 2 and Figure 2 the result for the weak scalability is seen. The size of the matrix n is if necessary approximated to be evenly divided by the number of processors.

Table 2: The time for increasing numbers of processors and increasing matrix size. The matrix size varies to maintain the workload constant for the processors. The matrix size was calculated with $N = sqrt(p) * n$ where n is the matrix size for 1 processor and p is the number of processes. For example for 8 cores the matrix size is $sqrt(8) * 2000 \approx 5600$

# Cores	1	2	4	8	10	12	16
Matrix size n	2000	2800	4000	5600	6300	6900	8000
Time	27.8620	42.4187	61.8066	97.1456	114.2477	125.8736	151.1549

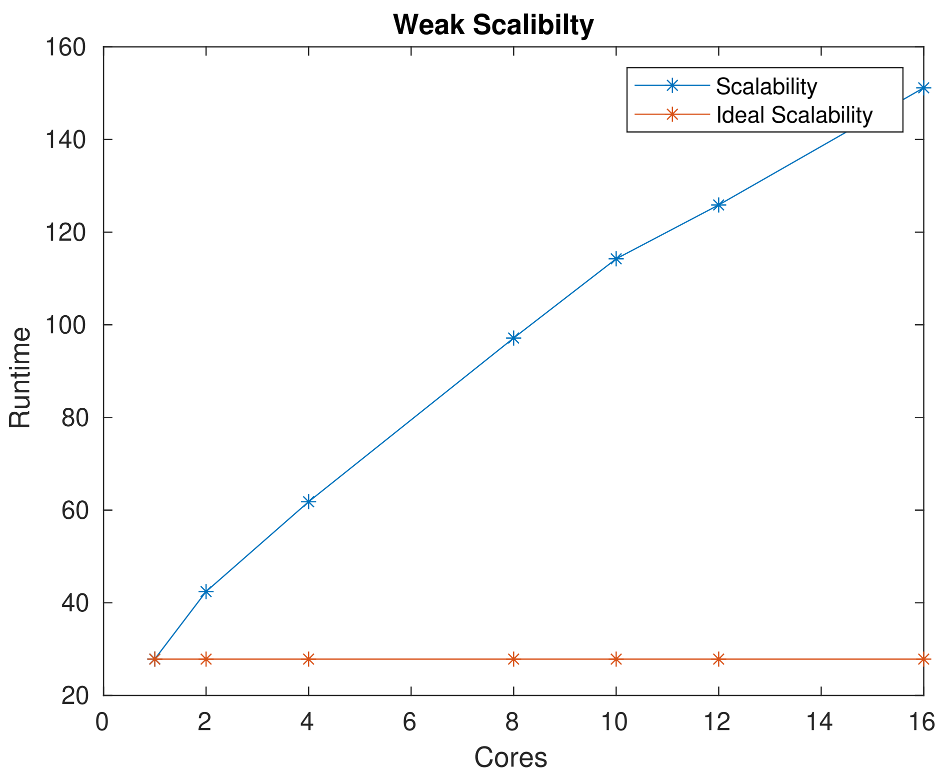

Figure 2: The weak scalability and the ideal weak scalability plotted against the number of cores used. The size of the matrix is increased with number of cores to keep the work constant for each processor. See table 2 for details.

4 Conclusions

In table 1 and figure 1 the result for the strong scalability is shown. The program has a good speedup up to around 20 cores where it still has a linear behaviour. After that speedup stops to increase. The graph follows a typical speedup. A reason to why the speedup works well for the program may be that shearsort algorithm used is made to work well in parallelization.

In table 2 and figure 2 the weak scalability is shown. The weak scalability does not appear to be good since the time increases significantly between the different number of cores. This may be because there are more communication needed the more processes you have, as all processes communicates with all others at two occasions in the loop. Also the number of times the loop is repeated depends on how large n is. The matrix is increasing with the number of processes which also could be a contribution to why the scaling is not good.

An improvement of the code would be to optimize the code to have a better weak scalability performance. It would also be interesting to explore if there is a more efficient way to do the communications or distribution of elements between processors.

References

[1] . Parallel Sorting Algorithms. `https://www.cpp.edu/~gsyoung/CS370/14Sp/Parallel%20Sorting%20Algorithms%20Matthew.pdf`, 2021. Online; accessed 20 May 2021.