UPPSALA
UNIVERSITET

Parallel and Distributed Programming
Project: Shear sort

Niklas Wik

980227

Uppsala

May 24, 2021

Introduction

Imagine that you want to sort a square matrix in a snake-like fashion, that is, even and odd rows are sorted in ascending and descending order respectively, and where no number in a subsequent row is smaller than the numbers in an arbitrary row. One simple algorithm for this task is the Shear sort algorithm. The goal of this project was to implement the algorithm in a parallel computer environment with distributed memory using MPI, with particular focus on communication and load balancing between the processing elements (PE:s).

Shear sort

The Shear sort algorithm on a $n \times m$ matrix starts by sorting all even rows in ascending order and each odd row in descending order, followed by sorting all columns in ascending order. After the columns are sorted, the rows are sorted in the same way again as before. The algorithm requires $\log(n) + 1$ iterations of sorting the rows, and $\log(n)$ iterations of sorting the columns. For simplicity however, the implementation discussed sorts a $n \times n$ matrix instead, but the algorithm is identical. A second implementation working for $n \times m$ matrices was also made, but all experiments were done for the $n \times n$ version.

Implementation

The initial implementation first generates a $n \times n$ matrix with a uniform distribution between 1 - n^2, where n is an input parameter. The approach to load balancing is simple; the rows are divided equally between all PE:s so that each PE can sort its provided rows undisturbed. It is trivial that this will be faster than letting the PE:s sort each row together in parallel since that requires more communication. The actual sorting, both row-wise and column-wise, is done with Quicksort as it is one of the fastest algorithms for 1D sorting.

After the rows have been sorted each iteration, the columns are in the same way divided evenly between the PE:s. Since each PE have parts from all columns, every PE has to send to and receive values from every PE. This is done with MPI_Alltoallw. While receiving the columns, the values are stored column-wise, compared to the row-wise stored rows, so that during 1D sorting, stride one access is used. To achieve this, blocks of rows are sent which are received as multiple row vectors instead of blocks. Similarly, when the columns have been sorted, all to all communication is done once again, sending multiple row vectors which are received as blocks, so that every PE receives the updated rows, stored row-wise. Since every PE has both some of the rows and some of the columns, the overall space complexity is $\mathcal{O}(2 \cdot n^2)$, and for each PE $\mathcal{O}(2 \cdot \frac{n^2}{p})$, where p is the number of PE:s. The implementation sorts integers, which means that each PE uses roughly $4 \cdot 2 \cdot \frac{n^2}{p}$ bytes of memory.

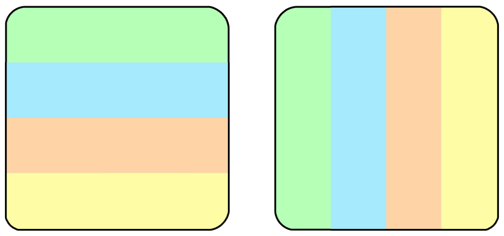


Figure 1: Distribution of rows and columns. Each color represent the rows and columns that belong to one PE.

Now, if n is not evenly divisible by the number of PE:s p, $n \bmod p$ of the PE:s receive one more row and column respectively. This complicates the communication somewhat, as equally sized blocks no longer can be sent to all PE:s. This is why MPI_Alltoallw is used instead of MPI_Alltoallv so that different block sizes can be sent to and received from different PE:s. In particular, the 4 different blocks differ by 1 row and/or column. This is illustrated bellow in Figure 2. Note that there are only two different sizes of row vectors, which differ by 1 in width, however, the number of row vectors received also differ by 1 depending on which PE that sent and received. Finally, sending rows to columns and columns to rows is symmetric, which simplifies the implementation somewhat.

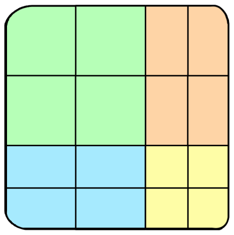


Figure 2: Each color represent one block size. Here $n \bmod p = 2$, that is, two of the PE:s have one more row and column than the other two PE:s. Only the green block is used if $n \bmod p = 0$.

The second implementation working for $n \times m$ matrices is very similar, as there still will be up to 4 different block sizes used. However, if, for instance, the number of rows is divisible by the number of PE:s and the number of columns is not, only two block sizes are used. The blocks are sent end received in the same way between the PE:s.

A priori time estimate - $n \times n$

As mentioned above, the algorithm requires $\log_2(n) + 1$ iterations. Every iteration, each PE sorts $\frac{n}{p}$ rows and $\frac{n}{p}$ columns with the Quicksort algorithm, which itself is $\mathcal{O}(n \log(n))$. Moreover, each process send and receive $\frac{n^2}{p}$ values twice per iteration respectively. We then get

$$T_p = \mathcal{O}_{comp}\left(\log_2(n)\log(n)\frac{n^2}{p}\right) + \mathcal{O}_{comm}\left(\log_2(n)\frac{n^2}{p}\right) \tag{1}$$

However, since all PE:s need to synchronise before each communication, as well as initialise the communication, there will be some added time which depend on p which is why it is not expected to see perfect strong or weak scaling. Moreover, for $n \to \infty$ we see that eq 1 goes towards $\mathcal{O}_{comp}\left(\log_2(n)\log(n)\frac{n^2}{p}\right)$.

Experiments

The experiments done include a complexity plot, runtime evaluation using Allinea MAP, strong scaling, and weak scaling. As mentioned, all experiments were done on the $n \times n$ implementation. In all experiments, only the part of the implementation that executes the Shear sort algorithm was measured as number generation, initiation, and gathering of the final matrix is not of interest. All runs were executed on UPPMAX Snowy cluster.[1]

The complexity plot was done using only $p = 1$ and can be seen in Figure 3 and Table 1.

The runtime evaluation using Allinea MAP was done using $p = 4$ for $n = 4000$ to see what parts of the implementation that took the longest time, as well as to gain a better understanding of the final results. Screenshots of the most important parts of the evaluation are shown in Figure 4.

The strong scaleability experiments were done using $n = 4800$ for $p = 1$ up to $p = 32$. Only relative speedup was measured as no serial algorithm was implemented. The result can be seen in Figure 5 and Table 2.

Lastly, the weak scaling experiments were done by letting the computational work load estimate per PE in eq. 1 stay constant while increasing the number of PE:s p. In particular, $\log_2(n_1)\log(n_1)\frac{n_1^2}{1} = \log_2(n_p)\log(n_p)\frac{n_p^2}{p}$, where $n_p \times n_p$ matrices were sorted for p number of PE:s. The result is shown in Figure 6 and Table 3.

Result

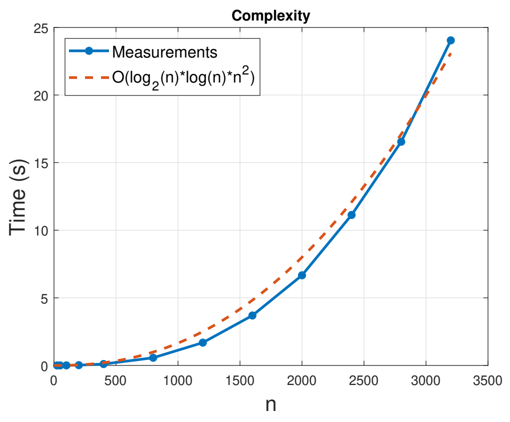
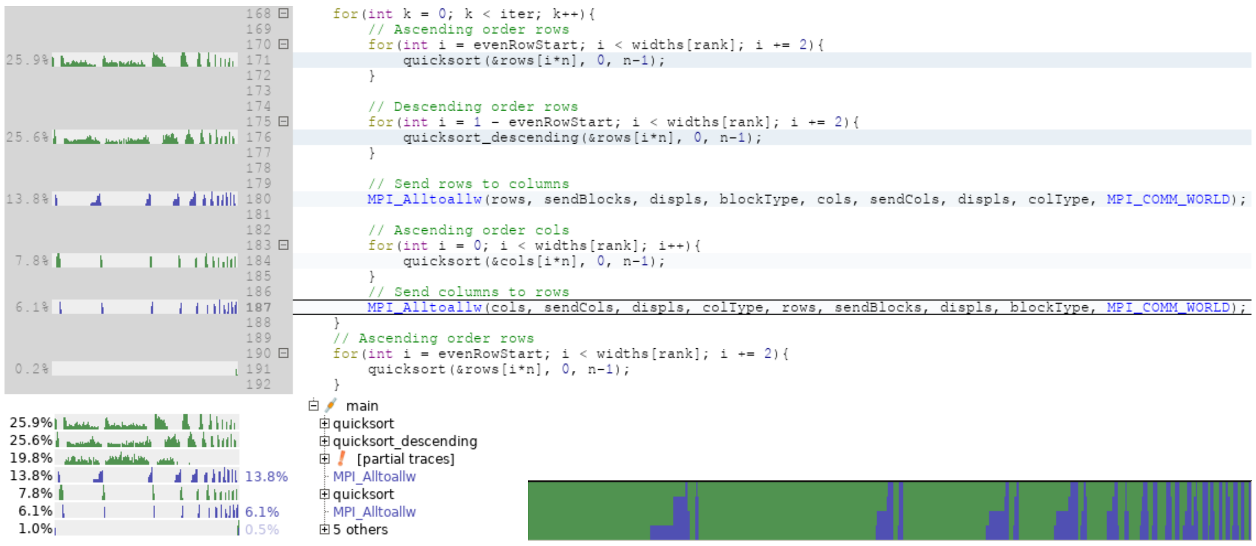


Figure 3: Complexity plot with $p = 1$.

n	Time (s)
25	0.000
50	0.001
100	0.004
200	0.020
400	0.102
800	0.566
1200	1.685
1600	3.693
2000	6.667
2400	11.130
2800	16.544
3200	24.045

Table 1: Timings for different n with $p = 1$

[1]UPPMAX Snowy cluster.



```
168         for(int k = 0; k < iter; k++){
169             // Ascending order rows
170             for(int i = evenRowStart; i < widths[rank]; i += 2){
171                 quicksort(&rows[i*n], 0, n-1);
172             }
173 
174             // Descending order rows
175             for(int i = 1 - evenRowStart; i < widths[rank]; i += 2){
176                 quicksort_descending(&rows[i*n], 0, n-1);
177             }
178 
179             // Send rows to columns
180             MPI_Alltoallw(rows, sendBlocks, displs, blockType, cols, sendCols, displs, colType, MPI_COMM_WORLD);
181 
182             // Ascending order cols
183             for(int i = 0; i < widths[rank]; i++){
184                 quicksort(&cols[i*n], 0, n-1);
185             }
186             // Send columns to rows
187             MPI_Alltoallw(cols, sendCols, displs, colType, rows, sendBlocks, displs, blockType, MPI_COMM_WORLD);
188         }
189         // Ascending order rows
190         for(int i = evenRowStart; i < widths[rank]; i += 2){
191             quicksort(&rows[i*n], 0, n-1);
192         }
```

Figure 4: Result from Allinea Map with $p = 4$ and $n = 4000$ of line 160-192 of the implementation as seen the upper most plot. The plot in the lower right corner show time in the horizontal axis and what each PE does in the vertical axis, where green represent computation and memory access, and blue represent communication. [partial traces] include recursive calls to functions *quicksort()* and *quicksort_ descending()*

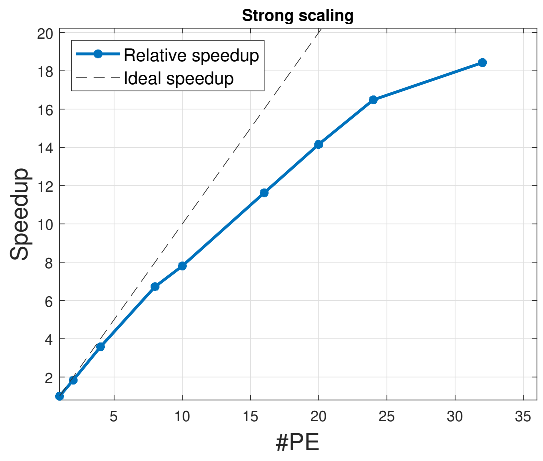


Figure 5: Relative speedup with $n = 4800$.

PE:s	Time (s)	Speedup
1	72.174	1.00
2	39.264	1.84
4	20.183	3.58
8	10.741	6.72
10	9.249	7.80
16	6.210	11.62
20	5.098	14.16
24	4.379	16.48
32	3.917	18.43

Table 2: Timings and relative speedup for $n = 4800$, where the timings are the fastest out of 4 runs.

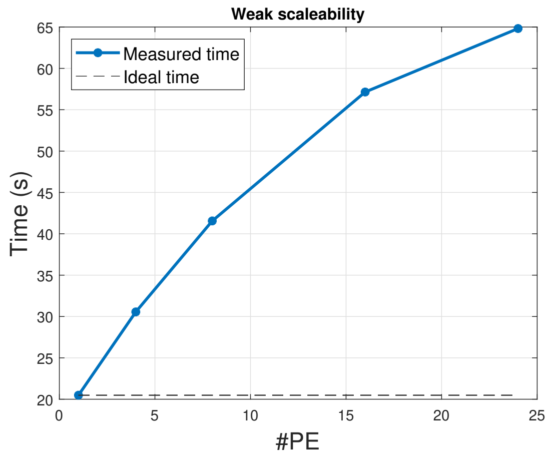


Figure 6: Average timings of 4 runs for same load per PE.

PE:s	n	$\log_2(n)\log(n)\frac{n^2}{p}$	Time (s)
1	3000	8.32e+08	20.485
4	5570	8.32e+08	30.558
8	7602	8.32e+08	41.556
16	10388	8.32e+08	57.140
24	12476	8.32e+08	64.817

Table 3: Number of PE:s relative to n, load, and timing

Discussion

Firstly, the complexity plot in Figure 3 show that the estimated computational work in eq. 1 seems correct. The fit is not perfect since the script for $p = 1$ still "send" all values from row-wise storage to column-wise storage.

Furthermore, we see from the result from the Allinea MAP evaluation in Figure 4 that the majority of the time is spent sorting the rows in ascending and descending order. Why it is slower to sort the rows than the columns is thought to be because the row sorting is divided into two for-loops compared to the single loop sorting the columns. An improvement here is to fuse the two row sorting loops into one loop. This can be done fast by using two *const int* to check if the first and last rows for each PE are odd.

Moreover, we can also observe from Figure 4 that some of the PE:s have to wait for the other PE:s before the communication of sending rows to columns can begin. In the contrary, all PE:s are very synchronised before the communication of sending columns to rows can begin. (In the bottom right plot in Figure 4 there are groups of two blue areas, where the wider blue area is the first *MPI_ Alltoallw* sending rows to columns).

From the result of the strong scaling in Figure 5 we can observe a piece-wise linear but non-ideal speedup for 1-8 PE:s, and 8-24 PE:s. The fact that the speedup is linear between 1-8 PE:s can be explained by the fact that the Snowy cluster exists of nodes with dual CPU:s with 8 cores per CPU. However, considering this we would expect to see piece-wise linearity between 8-16 PE:s instead of between 8-24. The fact that the speedup is non-ideal was mentioned under Section A priori time estimate - $n \times n$, the fact that the PE:s has to synchronise before, and initialise the communication each iteration. This was again observed from the Allinea MAP evaluation.

Lastly, from the result of the weak scaling experiments in Figure 6 we can see that the communication term in eq. 1 is present, but that the execution time seems to flatten as n increases as a consequence of the fact that the computation term $\mathcal{O}\left(\log_2(n)\log(n)\frac{n^2}{p}\right)$ scales faster than the communication term $\mathcal{O}\left(\log_2(n)\frac{n^2}{p}\right)$. This means that the algorithm scales reasonably well for larger and larger n.

Worth mentioning is that evaluating the $n \times m$ implementation would mainly change the a priori time estimate to

$$T_p = \mathcal{O}_{colSort}\left((\log_2(n)+1)\log(n)\frac{n^2}{p}\right) + \mathcal{O}_{rowSort}\left(\log_2(n)\log(m)\frac{m^2}{p}\right) + \mathcal{O}_{comm}\left((\log_2(n))\frac{nm}{p}\right) \tag{2}$$

This would however only complicate the evaluation of the algorithm and is out of the scope of the project.