UPPSALA
UNIVERSITET

Parallel and Distributed Programming
Individual Project

Author:

Moa Li Lekander

Uppsala

May 25, 2021

Contents

1 Introduction

In many areas of science today, great computational resources are necessary for being able to perform the heavy computations that are required within the fields. Parallel programming can in many of these cases be of great use in order to minimize run times and make programs more efficient, as the power of several computers can be combined.

In this project, the aim is to implement a parallelized code on a distributed memory, and then analyze its performance. In this report, the shearsort algorithm, a sorting algorithm that is applied to two dimensional lists, is implemented and parallelized using C and MPI. The program makes the assumption that the matrix size is divisible by the number of processes.

2 Problem Description

Sorting lists is a basic feature that is widely used within computer science, and there exist many different sorting algorithms with varying efficiencies. The shearsort algorithm sorts the values in a matrix in a "snake-like" order, as presented in Figure 1, where the smallest value is situated in the top left corner and the largest value in the bottom right corner. The shear sort algorithm begins by sorting all even rows in ascending order and all odd rows on descending order. Then, every column is sorted in ascending order. This procedure is repeated $log_2 N + 1$ times for the row sorting part, and $log_2 N$ times for the column sorting part, where N is the matrix size.

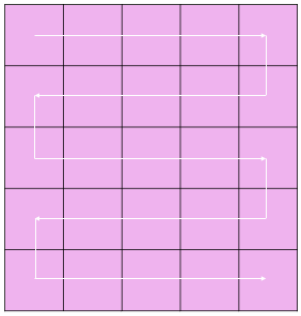

Figure 1: Visualization of the "snake-like" order of a sorted 2D array of size 5x5, where the smallest element is at the top left corner and the largest element at the bottom right corner.

The goal of this project, is to implement a parallelized shearsort algorithm, using C and MPI. Its performance is then evaluated by studying both the strong and weak scaling of the program.

3 Solution Approach

The implemented program is presented in Appendix A. The program takes one input argument that specifies the size of the matrix to sort, meaning that if `N` is entered a matrix of size NxN will be generated.

The program uses a row-wise partitioning strategy so that the row sorting phases of the algorithm are possible without any communications between the processes. Thus, the program begins with initializing the MPI and then the assumptions that the size of the matrix is divisible by the number of processes and that it is larger than 0 are checked, and sends an error message and terminates the program if those conditions are not fulfilled. Then the matrix is filled with randomly generated values on the process of rank 0 in row-major order. Thereafter, an equal amount of rows of the matrix is distributed to all processes using `MPI_Scatter`, where the values are stored in the local array `local_data`. The matrix then gets sorted in the "snake-like" order described in section 1, as all processes call the function `shearsort`.

The `shearsort` function takes the local array (which in the function is called `data`) with its number of rows and columns, together with the total amount of processes and the rank of the process calling the function, as input. This function follows the shearsort algorithm, and firstly it has an outer for-loop specifying the number of phases to be performed. Inside this loop, every process sorts its designated even rows in ascending order, and its designated odd rows in descending order (according to the row numbering in the full NxN matrix). This sorting is performed by the function `quicksort` (which also uses the functions `partition_a` or `partition_d`), which performs a one dimensional list sorting according to the quicksort algorithm. This algorithm was chosen for this part since it has a time complexity of $\mathcal{O}(n \cdot log(n))$, which is advantageous for large lists. Thereafter, if the iteration is not the last one, all columns are sorted. As this implementation uses a row-wise partitioning strategy, the columns are distributed among the processes by transposing the whole matrix so that every column gets represented as a row on every process. This is done by firstly reordering the local array into p blocks, where p is the total number of processes. The result of this reordering is stored in the local array `temp`. This `temp` array is then transposed using three nested for-loops, and the result is stored in the local `data` array. Then the transposed blocks in `data` are distributed among all processes by using `MPI_Alltoall` so that every process also receives data in the `temp` array. These two arrays are then merged together in `data` so that the transposition is complete. The local data on every process (now representing the rows of the matrix), is then sorted in ascending order using the `quicksort` function once again. When that is finished, the matrix is transposed back in the same manner as before, and finally the allocated memory for the `temp` array is freed.

When the `shearsort` function is complete, every local array should be sorted according to the final result. Thus, the next step in the main-function is to collect all `local_data` arrays to rank 0 by using `MPI_Gather`, which stores the final matrix in the original `data` array. For the purpose of verifying the correctness of the result, a piece of code is thereafter included that checks whether the matrix is sorted or not. Before the program terminates, all allocated memory is freed and a call to `MPI_Finalize` is made.

4 Experiments

In order to evaluate the performance of the parallel program, a set of experiments was carried out on the UPPMAX system Snowy. This system enabled so that a total of 40 processes were available when the strong and weak scaling analyses were made. The execution times for performing the shearsort algorithm was measured on every process, and then the longest measured time was considered as the final time measurement.

4.1 Strong Scaling

In the strong scaling performance experiment, a fixed matrix size of $N = 5000$ was used as the number of processes was increased. Since the total number of cores available was 40, this problem size made it possible to run the program on 1, 2, 4, 5, 8, 10, 20, 25 and 40 processes respectively. The execution times that were measured are presented in Table 1 and Figure 2 together with the computed relative speedup. The equation for computing the relative speedup $S(N, p)$, where N is the problem size and p is the number of processes, is presented in equation (1). Here, $T(N, 1)$ is the execution time for when the program is run on one process and $T(N, p)$ the execution time when run on p processes.

$$S(N, p) = \frac{T(N, 1)}{T(N, p)} \tag{1}$$

Table 1: Measured execution times and computed speedup for different number of processes in the strong scaling analysis.

Processes	Execution time (s)	Speedup
1	107.953	1.000
2	58.887	1.834
4	29.192	3.698
5	24.143	4.471
8	15.483	6.972
10	13.222	8.165
20	7.273	14.844
25	6.448	16.741
40	5.096	21.185

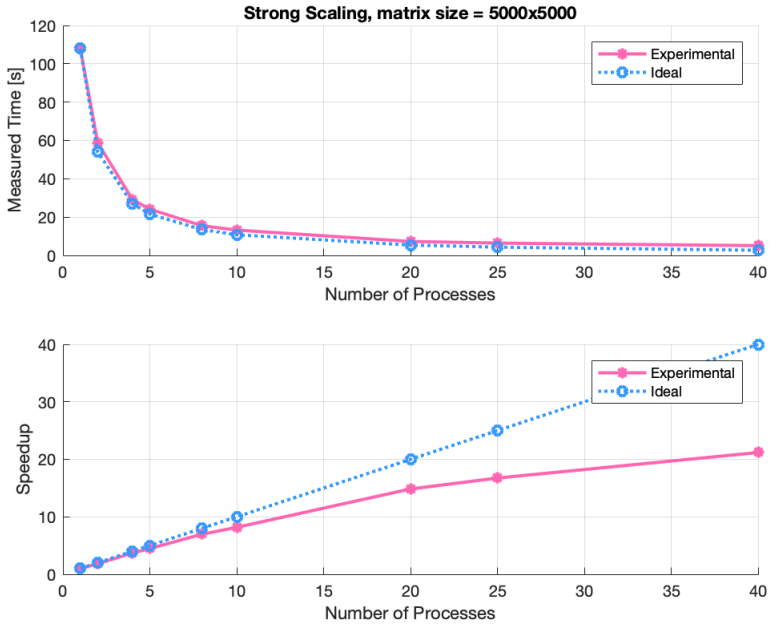

Figure 2: Execution times and calculated speedup depending on the number of processes for both the ideal and experimental case, in the strong scaling experiment.

4.2 Weak Scaling

In the weak scaling experiment, the amount of values for every process to sort was kept constant at a value of 4 000 000 as the number of processes increased. As a result, the program could be run with 1, 4, 16, and 25 processes, with a matrix size of 2000, 4000, 8000 and 10000 respectively, in order to fulfill the assumption that the matrix size must be divisible by the number of processes. The measured execution times are presented in Table 2. In Figure 3 the values are plotted against the number of processes used together with the ideal values.

Table 2: Measured execution times for different problem sizes and different number of processes in the weak scaling analysis.

Processes	Matrix size	Execution time (s)
1	2000	8.066
4	4000	15.164
16	8000	34.636
25	10000	40.709

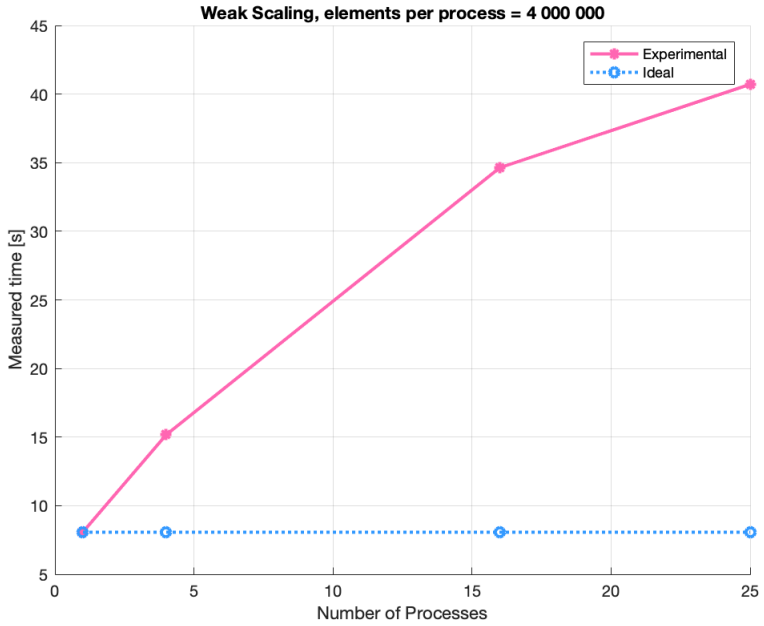

Figure 3: Ideal and measured execution time depending on the number of processes, in the weak scaling experiment.

4.3 Efficiency

The efficiency, that can be perceived as the speedup per process, was computed from the strong scaling execution times. Ideally, the efficiency should be kept constant at the value of 1, which would imply that the computer resources are being utilized effectively. The efficiency $E(N, p)$ was calculated according to equation (2), where $S(N, p)$ is the speedup when the program is run for a problem size of N run on p processes. The computed efficiencies are presented in Table 3 and displayed together with the ideal efficiency in Figure 4.

$$E(N,p) = \frac{S(N,p)}{p} = \frac{T(N,1)}{p \cdot T(N,p)} \tag{2}$$

Table 3: Computed efficiency for the strong scaling analysis.

Processes	Efficiency
1	1.000
2	0.917
4	0.925
5	0.894
8	0.872
10	0.816
20	0.742
25	0.670
40	0.530

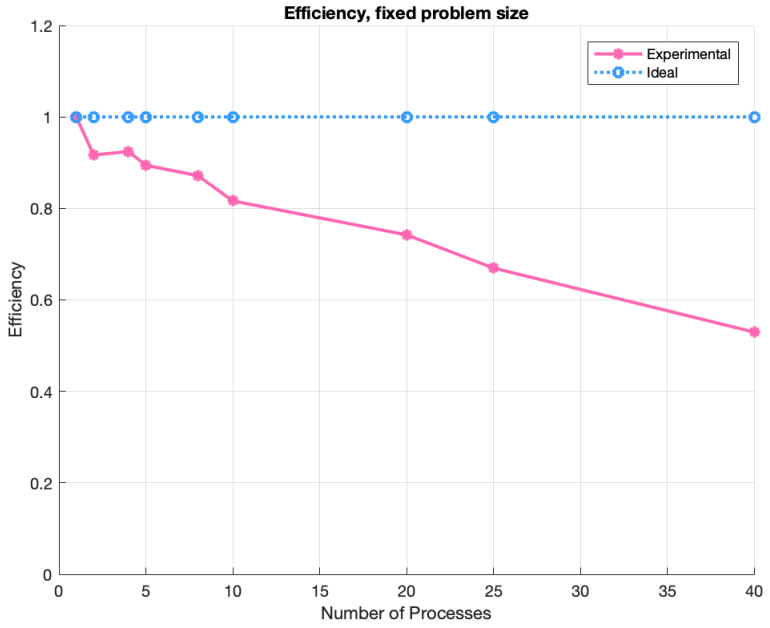

Figure 4: Ideal and calculated efficiency for the strong scaling experiment.

5 Conclusions

When analysing the strong scaling performance, presented in Figure 2, it can be observed that the experimental execution time very closely follows the trend showed by the ideal case. The execution time decreases as the amount of processes increase, which is the expected result. The experimental speedup follows the ideal speedup closely for a smaller number of processes, but it can be noted that as more processes are used the experimental speedup differ in an increasing amount. This result suggests that the program is not ideal for when a large amount of processes used, which can be because the amount of communications necessary becomes more time consuming relative to the time for the other parts of the algorithm. Another explanation could be that the memory requirement for the code increases a lot, which also can have a large impact on the performance.

Regarding the weak scalability test, it can be observed in Figure 3 that the measured time differ significantly from the ideal case. Hence, the conclusion that the program is not weakly scalable can be drawn. The time does not remain constant even though the workload for every process is kept the same, which suggests that the communications and memory requirement in the implementation presumably have a large impact on the performance. In order to fix this, focus has to be put on making the necessary communications more efficient and if possible, reducing the memory requirement of the program.

The efficiency of a program gives information regarding how well the computer resources are being utilized. When analysing the efficiency for the strong scalability test presented in Figure 4, it can be observed that the efficiency does not remain constant as the number of processes increase. This means that when the number of processes becomes larger, the computer resources are not being used as efficiently as desired. Hence, it can be concluded that the program is not strongly scalable.

In summary, the implemented program is not scalable. The efficiency does not remain constant as the number of processes increase, and the performance is not linearly proportional to the number of processes used, meaning that the strong scalability requirements are not fulfilled. Also, the execution time is not kept at a constant value when both the problem size and the number of processes increase which implies that the program is neither weakly scalable.

In order to improve the implementation, focus should be put on making the column sorting part of the algorithm more efficient. As this part of the algorithm require most communications and extra memory usage, it is presumably this part of the code that has most potential for reducing the time of the program, if optimized successfully. There may exist more effective ways to transpose the matrix, however in this project no such improvements could be detected.

Appendix A

```
/*****************************************************************
 * Parallelized shearsort
 * Usage: mpirun -np p shearsort N
 *****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

void printMatrix(double* M, int row, int col);
void shearsort(double *data, int rows, int col, int rank, int size);
void quicksort(double *data, int left, int right, int direction);
int partition_a(double *data, int left, int right, int pivotIndex);
int partition_d(double *data, int left, int right, int pivotIndex);

int main(int argc, char *argv[]){
  int N, size, rank, workload;
  double *data, *local_data, start_time, execution_time, max_time;

  if(argc != 2){
    printf("ERROR! Expected input: shearsort N\n");
    exit(0);
  }
  N = atoi(argv[1]); // matrix size (NxN)

  MPI_Init(&argc, &argv);               // Initialize MPI
  MPI_Comm_size(MPI_COMM_WORLD, &size); // Get the number of processors
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Get my number

  // Check assumptions
  if(N < 1){
    if(rank == 0) printf("ERROR! N must be larger than 0\n");
    MPI_Finalize();
    exit(0);
  } else if(N%size != 0){
    if(rank == 0) printf("ERROR! N must be divisible by # of processes\n");
    MPI_Finalize();
    exit(0);
  }

  // Fill 2D array
  if(rank == 0){
    data=(double *)malloc(N*N*sizeof(double));
    for(int i = 0; i < N*N; i++){
      data[i] = drand48();
    }

    // Print initial list
    /*printf("Initial List:\n");
    printMatrix(data, N, N);*/
  }

  // Start timer
  start_time = MPI_Wtime();

  // Distribute list to processes
  workload = N/size; // number of rows per process
  local_data = (double*)malloc(N*workload*sizeof(double));
  MPI_Scatter(&data[0], N*workload, MPI_DOUBLE, &local_data[0], N*workload, MPI_DOUBLE, 0, ...
      MPI_COMM_WORLD);

```

```
61    // Sort local list
62    shearsort(local_data, workload, N, rank, size);
63
64    // Put all sorted local lists together
65    MPI_Gather(&local_data[0], N*workload, MPI_DOUBLE, &data[0], N*workload, MPI_DOUBLE, 0, ...
          MPI_COMM_WORLD);
66
67    // Compute time
68      execution_time = MPI_Wtime()-start_time; // stop timer
69      MPI_Reduce(&execution_time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
70
71    if(rank == 0){
72      // Display time result
73      printf("%f\n", max_time);
74
75      // Print sorted matrix
76      /*printf("\nFinal List:\n");
77      printMatrix(data, N, N);*/
78
79      // Check results
80      int OK = 1;
81      double prev = data[0];
82      for (int i = 0; i < N; i++) {
83        if(i % 2 == 0){ // if even row -> left to right
84          for(int j = 0; j < N; j++){
85            if(data[i*N+j] < prev){
86              OK = 0;
87            }
88          }
89        } else { // off row -> right to left
90          for(int j = N-1; j ≥ 0; j--){
91            if(data[i*N+j] < prev){
92              OK = 0;
93            }
94          }
95        }
96      }
97
98      if(OK){
99        printf("Data sorted correctly!\n");
100     } else {
101       printf("Data NOT sorted correctly...\n");
102     }
103   }
104
105   // Clean up
106   if(rank == 0) free(data);
107   free(local_data);
108   MPI_Finalize();
109
110   return 0;
111 }
112
113 void printMatrix(double* M, int row, int col){
114   for(int i = 0; i < row; i++){
115     for(int j = 0; j < col; j++){
116       printf("%10f ", M[i*col+j]);
117     }
118     printf("\n");
119   }
120 }
121
122 void shearsort(double *data, int rows, int col, int rank, int size){
123   int d = log(col)/log(2);
124
```

```
125     for(int l = 0; l < d+1; l++){
126       if(rank%2 == 0){ // first row even
127         for(int k = 0; k < rows; k += 2){ // even rows
128           quicksort(&data[k*col+0], 0, col-1, 0); // ascending order
129         }
130         for(int k = 1; k < rows; k += 2){ // odd rows
131           quicksort(&data[k*col+0], 0, col-1, 1); // descending order
132         }
133       } else { // first row odd
134         for(int k = 0; k < rows; k += 2){ // odd rows
135           quicksort(&data[k*col+0], 0, col-1, 1); // descending order
136         }
137         for(int k = 1; k < rows; k += 2){ // even rows
138           quicksort(&data[k*col+0], 0, col-1, 0); // ascending order
139         }
140       }
141
142       if(l ≤ d){
143         // Create size # of blocks (of size rows*rows)
144         double *temp = (double*)malloc(rows*col*sizeof(double));
145         for(int i = 0; i < rows; i++){
146           for(int block = 0; block < size; block++){
147             for(int j = 0; j < rows; j++){
148               temp[block*rows*rows+i*rows+j] = data[i*col+(block*rows+j)];
149             }
150           }
151         }
152
153         // Transpose local blocks
154         for(int block = 0; block < size; block++){
155           for(int i = 0; i < rows; i++){
156             for(int j = 0; j < rows; j++){
157               data[block*rows*rows+i*rows+j] = temp[block*rows*rows+j*rows+i];
158             }
159           }
160         }
161
162         // All to all communication
163         MPI_Alltoall(data, rows*rows, MPI_DOUBLE, temp, rows*rows, MPI_DOUBLE, MPI_COMM_WORLD);
164
165         // Merge
166         for(int i = 0; i < rows; i++){
167           for(int block = 0; block < size; block++){
168             for(int j = 0; j < rows; j++){
169               data[i*col+(block*rows+j)] = temp[block*rows*rows+i*rows+j];
170             }
171           }
172         }
173
174         // Sort local data (now columns) in ascending order
175         for(int k = 0; k < rows; k++){ // columns
176           quicksort(&data[k*col+0], 0, col-1, 0);
177         }
178
179         // TRANSPOSE BACK
180         // Create size # of blocks (of size rows*rows)
181         for(int i = 0; i < rows; i++){
182           for(int block = 0; block < size; block++){
183             for(int j = 0; j < rows; j++){
184               temp[block*rows*rows+i*rows+j] = data[i*col+(block*rows+j)];
185             }
186           }
187         }
188
189         // Transpose local blocks
```

```
190         for(int block = 0; block < size; block++){
191           for(int i = 0; i < rows; i++){
192             for(int j = 0; j < rows; j++){
193               data[block*rows*rows+i*rows+j] = temp[block*rows*rows+j*rows+i];
194             }
195           }
196         }
197 
198         // All to all communication
199         MPI_Alltoall(data, rows*rows, MPI_DOUBLE, temp, rows*rows, MPI_DOUBLE, MPI_COMM_WORLD);
200 
201         // Merge
202         for(int i = 0; i < rows; i++){
203           for(int block = 0; block < size; block++){
204             for(int j = 0; j < rows; j++){
205               data[i*col+(block*rows+j)] = temp[block*rows*rows+i*rows+j];
206             }
207           }
208         }
209         free(temp);
210       }
211     }
212   }
213 
214   void quicksort(double *data, int left, int right, int direction){
215       int pivotIndex, pivotNewIndex;
216 
217       if(direction == 0){ // ascending order
218         if (right > left){
219           pivotIndex = left+(right-left)/2;
220           pivotNewIndex = partition_a(data, left, right, pivotIndex);
221 
222           quicksort(data, left, pivotNewIndex - 1, 0);
223           quicksort(data, pivotNewIndex + 1, right, 0);
224         }
225       } else { // descending order
226         if (right > left){
227           pivotIndex = left+(right-left)/2;
228           pivotNewIndex = partition_d(data, left, right, pivotIndex);
229 
230           quicksort(data, left, pivotNewIndex - 1, 1);
231           quicksort(data, pivotNewIndex + 1, right, 1);
232         }
233       }
234   }
235 
236   int partition_a(double *data, int left, int right, int pivotIndex){
237     double pivotValue, temp;
238     int storeIndex, i;
239     pivotValue = data[pivotIndex];
240 
241     temp = data[pivotIndex];
242     data[pivotIndex] = data[right];
243     data[right] = temp;
244 
245     storeIndex = left;
246     for (i = left; i < right; i++)
247       if (data[i] ≤ pivotValue){
248         temp = data[i];
249         data[i] = data[storeIndex];
250         data[storeIndex] = temp;
251 
252         storeIndex = storeIndex + 1;
253       }
254 
```

```
255    temp = data[storeIndex];
256    data[storeIndex] = data[right];
257    data[right] = temp;
258
259    return storeIndex;
260  }
261
262  int partition_d(double *data, int left, int right, int pivotIndex){
263    double pivotValue, temp;
264    int storeIndex, i;
265    pivotValue = data[pivotIndex];
266
267    temp = data[pivotIndex];
268    data[pivotIndex] = data[right];
269    data[right] = temp;
270
271    storeIndex = left;
272    for (i = left; i < right; i++)
273      if (data[i] > pivotValue){
274        temp = data[i];
275        data[i] = data[storeIndex];
276        data[storeIndex] = temp;
277
278        storeIndex = storeIndex + 1;
279      }
280
281    temp = data[storeIndex];
282    data[storeIndex] = data[right];
283    data[right] = temp;
284
285    return storeIndex;
286  }
```