Parallel and Distributed Programming – Spring 2025 Presentation for Individual Project on Shear Sort

Oskar Perers

June 3, 2025

The Good, the Bad and the Ugly

Outline

What is Shearsort?

Shearsort is a sorting algorithm for sorting a matrix of data. The data is, at the end of the algorithm stored in Snakewise order.

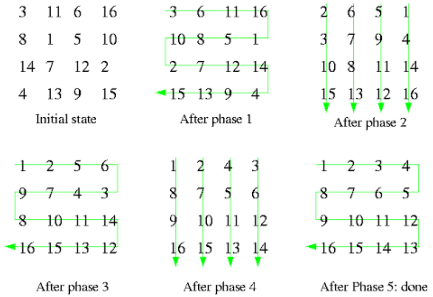


Figure: Example execution of the shear sort algorithm for a matrix of size 4×4[1].

[1] Roman Iakymchuk. *Parallel sorting algorithms*. Lecture notes, 1TD070 Parallel and Distributed Programming, Uppsala University. Accessed: May 2025. 2025.

Parallel algorithm

```
Require: $A$ is a $n \times n$ matrix where $n = \sqrt{N}$
 1: Distribute rows in $A$ over processes
 2: for $i = 1$ to $\log_2(n) + 1$ do
 3:     Sort local odd rows ascending
 4:     Sort local even rows descending
 5:     if $i \leq \log_2(n)$ then
 6:         Distribute columns in $A$ over processes
 7:         Sort local columns ascending
 8:         Distribute rows in $A$ over processes
 9:     end if
10: end for
```

Setup

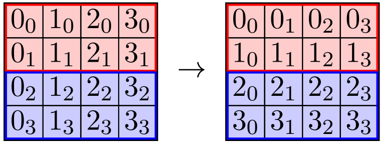
- **Input** - Possible to read from file or generate based on matrix order.
- **Storage class** - Helper class `matrix2D` stores data and supports the reuse of functionality through abstraction.
- **Distribution** - Each process is assigned a block of rows and columns they are responsible for throughout the algorithm.
- **Scatter** - The local rows for each process are distributed onto the processes using `MPI_Scatterv()`.

Finilization

- **Gather** - The sorted rows in each process are gathered back to root using `MPI_Gatterv()`.
- **Check** - The sorted matrix is stepped through to check correctness.
- **Output** - The sorted matrix is optionally written to a provided output file.

Global Sort

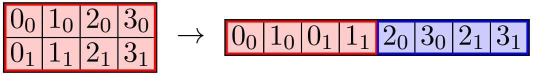
- **Sorting** - The sorting of local rows and columns is done using `std::sort()`.
- **Redistribution** - After each row and column phase, the data was redistributed so that each process got their assigned columns using `MPI_Alltoallw`.
- **Transpose** - The matrix is transposed during redistribution using `MPI Derived Datatypes` to improve data locality during sorting.



Send type

The goal is to group the local parts of the columns that shall be sent to each process. We use `MPI_Type_vector` for this purpose with the following parameters:

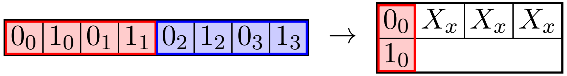
```
for (int i = 0; i < num_proc; ++i) {
    MPI_Type_vector(
        local_rows,                      // count
        rows_cols_per_proccess[i],       // blocklength
        global_dim,                      // stride
        MPI_INT,                         // basetype
        &send_types[i]
    );
    MPI_Type_commit(&send_types[i]);
}
```



Recive type

Read the incoming columns from all processes and store them as a transposed, column-wise matrix. This is done in multiple steps and the first is to read the incoming data as a column instead of a row. For this we use `MPI_Type_vector` with the following parameters:

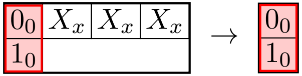
```
MPI_Type_vector(
    local_rows,                    // count
    1,                             // blocklength
    global_dim,                    // stride
    MPI_INT,                       // basetype
    &column_vector);
MPI_Type_commit(&column_vector);
```



Recive type

Next we have to resize the extent of this new datatype. By default the extent is the difference between the first and last element but that is not correct in our case. We want to store the columns immediately after one another. We do this with `MPI_Type_create_resized()` with the following parameters:

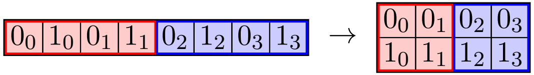
```
MPI_Type_create_resized(
        column_vector,                    // old_datatype
        0,                                // lower_bound
        sizeof(int),                      // extent
        &resized_column
    );
```



Recive type

Finally, we group all of the columns received from each process in a contiguous datatype using `MPI_Type_contiguous()` with the following parameters:

```
for (int i = 0; i < num_proc; ++i) {
    MPI_Type_contiguous(
        rows_cols_per_proccess[i],   //count
        resized_column,              //basetype
        &recv_types[i]
    );
    MPI_Type_commit(&recv_types[i]);
}
```



MPI_Alltoallw()

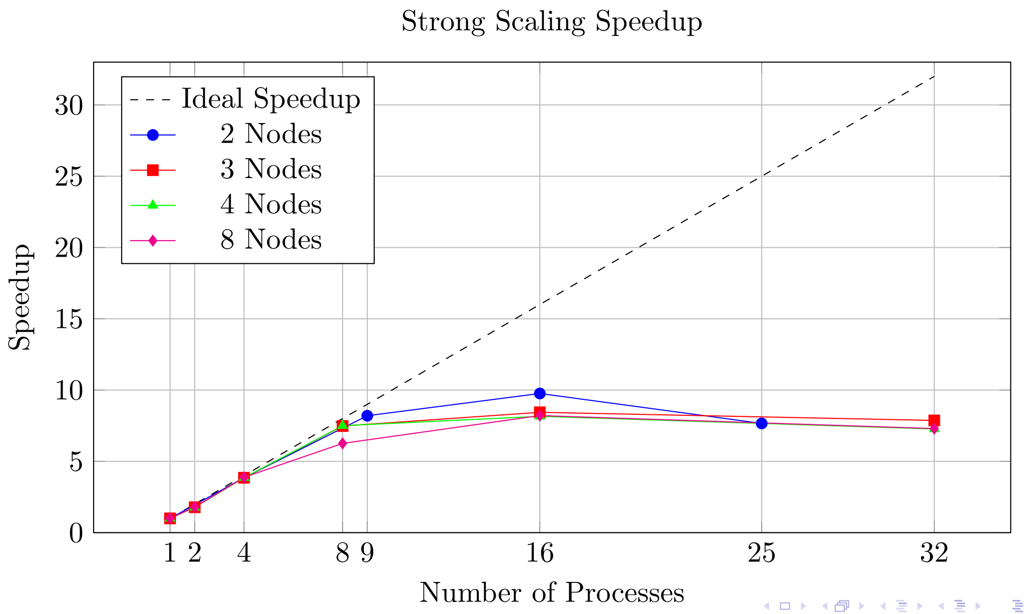
Using `MPI_Alltoallw()` we are able to use separate datatypes for each process we send to and receive from. This extension to `MPI_Alltoallw()` allows this program to work even if `n|nr_of_proceces`.

Strong Scalability

For strong scalability, I tested the program on a matrix of size $N = 8000 \times 8000$. The number of processes was generally set to $p = \{1, 2, 4, 8, 16, 32\}$ (except for the first test run). The speedup was calculated with the following equation and then plotted:

$$S_p[S] = \frac{T_1}{T_p}$$

Strong Scalability



Weak Scalability

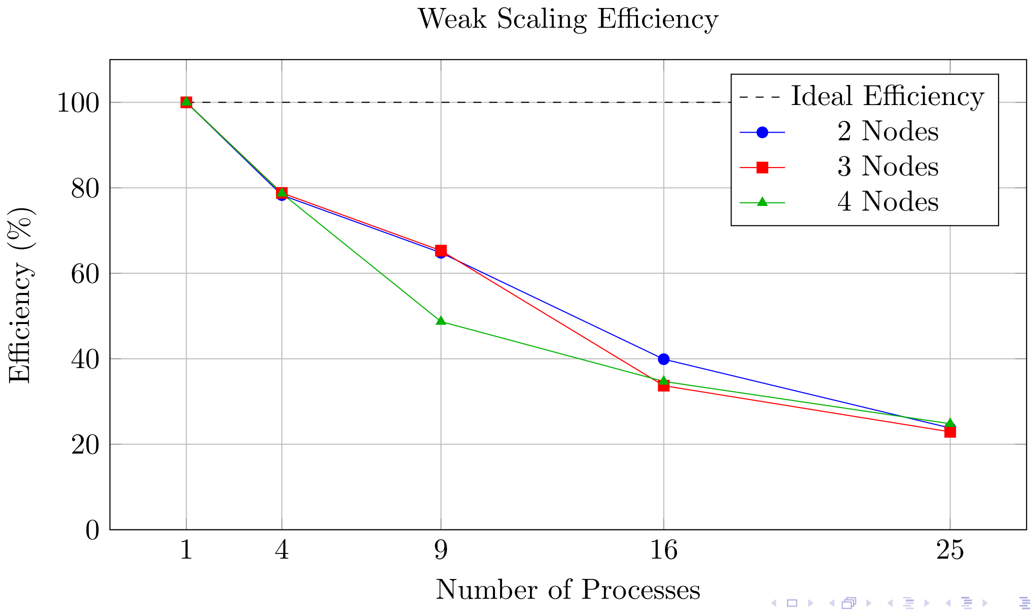
For strong scalability, I tested the program on a matrix of size $N_p = (2000 \times 2000) \times p$. To calculate the combinations between p and N_p the following equation was used:

$$\begin{aligned} N &= k \times P \\ n^2 &= 2000^2 \times P \\ \sqrt{n^2} &= \sqrt{2000^2 \times P} \\ n &= 2000\sqrt{P} \end{aligned}$$

This resulted in me setting the number of processes $p = \{1, 4, 9, 16, 25\}$ for $n = \{2000, 4000, 6000, 8000, 10000\}$ respectively. The efficiency was calculated with the following equation and then plotted:

$$E_p[\%] = \frac{T_1}{T_p} \cdot 100$$

Weak Scalability



Initial issues

My initial issues were quite mild. I figured out fast I wanted to use `MPI_Alltoall`. The issue was I could for a couple of days, not figure out how to handle that I wanted to transform the matrix during transfer with uneven data. This was solved when I found `MPI_Alltoallw`.
The second issue encountered had to do with extents. When I tried to place multiple column vectors next to one another, they were placed out of range. My first solution involved using `MPI_Type_create_hvector` and defining a byte offset between each column. This was later replaced with resizing.

The 12h debug

I encountered one bug that is the reason I'm still not done with all my courses for this semester. It was simple when I found it but it took a long time to figure out as it ran fine on my local machine:

```
MPI_Type_create_resized(
    column_vector,              // old_datatype,
    0,                          // lower_bound,
    sizeof(MPI_INT),            // extent
    &resized_column
);

for (int i = 0; i < num_proc; ++i) {
    send_counts[i] = 1;
    send_displs[i] = displs[i] * sizeof(MPI_INT);
    recv_counts[i] = 1;
    recv_displs[i] = displs[i] * sizeof(MPI_INT);
}
```