Report on the MPI Assignment

Sam van Kampen

October 19, 2022

1 Introduction

In this assignment, we were asked to parallelize an implementation of red-black successive over-relaxation (RB-SOR), measure its performance, and explain the design choices made in our implementation.

2 Implementation

All processes start by initializing the grid and determining the rows they own, based on the total number of rows, their own ID and the number of processes. The first process is designated the main process, and in addition to acting like a regular worker process it performs the execution timing and final data gathering roles.

Every process then starts the RB-SOR loop; in each iteration of the loop, for each phase, the processes perform an RB-SOR step and exchange elements (detailed below). Then, at the end of each iteration, the maximum difference among all processes is computed using `MPI_Allreduce`. If the maximum difference is below the threshold, the processes exit the loop.

To gather the final grid, the processes send each of the rows they own to the main process with `MPI_Send`, using a modified version of the row index as a tag. Standard send is used, as at worst this is as bad as synchronous send and at best as good as buffered send, without having to deal with buffering ourselves. The main process, then, probes for messages in order to get the row index and starts receives using `MPI_Irecv` for these rows. Receives are batched in groups of 20, which are waited for using `MPI_Waitall`.

2.1 Transmission of halo region elements

After performing the computation for a phase, the elements that are needed by neighboring processes are exchanged. These are the rows on a boundary between two processes. Additionally, because the computation progresses in two phases, after each step only the elements for a single phase need to be exchanged.

The elements are stored in the grid in a checkered fashion, where each row has alternating red and black elements. MPI supports sending these kinds of strided vectors through defining a custom data type with a given size, stride, and extent. In the case of RB-SOR, this data type is defined as in listing 1.

By using such a custom data type, a single-phase row can be exchanged using a send function such as `MPI_Send`, as can be seen in listing 2.

In the implementation, the elements are exchanged by performing buffered sends with `MPI_Bsend` (a slightly oversized send buffer is used, being large enough to contain 8 full rows), and non-blocking receives with `MPI_Irecv`. The data is sent from and received into the grid rows directly. This does not cause conflicts as there is no overlap between communication and computation.

For reasons unknown to me, the use of non-blocking receives here results in a significant performance improvement as compared to blocking reads. Performing non-blocking receives allows processes to start receiving whichever row arrives first, of course, but as processes wait for both rows to arrive immediately after starting the receives, I did not expect such a large performance improvement.

3 Experiments

The implementation was tested by running the sequential and parallel version on grids sized between 1250x1250 and 40000x40000 elements and between 1 and 8 processes, where each step up the grid size and number of processes doubled. These runs were repeated five times, and the mean runtime was then taken for each combination of (grid size, number of processes).

4 Results

The results of the experiments can be seen in figures 1, 2, and 3. Figure 1a shows the runtime of RB-SOR under various grid sizes and numbers of processes. In this graph, we can already see that the scaling is not ideal; in that case, in this logarithmic plot, we should expect to see nicely straight lines and evenly spaced sample points. Instead, the lines are wonky and the sample points are at varying distances from one another.

Figure 2a, which shows the speedup, paints a clearer picture. First of all, the performance impact of running a parallel version on 1 process is not very large; the orange and blue line practically overlap. At very small grid sizes, the parallel version suffers from the fact that there is too little data to overcome the communication overhead. At larger grid sizes, this is mitigated, and the 2-process runs get a nearly 2x speedup. This speedup is not linear, however; at larger numbers of processes, efficiency drops off.

The efficiency figures are given in figure 3a. At sufficiently large grid sizes, efficiency at 2 processes stays above 90%, at 4 processes it hovers around 85% and at 8 processes it hovers around 70%.

These figures are slightly puzzling – why does the efficiency dip at larger grid sizes and at larger numbers of processes? Shouldn't the efficiency improve – we're using larger grids, so processes should have less communication overhead. This can be explained by looking at the communication overhead for the final gather operation; at larger grid sizes, more time is spent on this communication. If we ignore the time taken to perform this gathering operation, speedup and efficiency are significantly better and improve as grid size increases. These results can be seen in figures 1b, 2b and 3b.

Another dip is visible around the 5000 element point. This may have to do with caching effects. At 2000 elements per row, three rows fit into level 1 cache of the processors in the DAS cluster ($2000 \cdot 8 \cdot 3 = 60.000$ bytes, smaller than the 64K L1 cache). At 5000 elements/row, less than two rows fit into L1 cache. This doesn't explain the dip on its own, however, as these caching effects also apply to the sequential version.

5 Discussion and conclusion

As can be seen from the results, the current implementation can reach fairly good efficiency, especially at larger task sizes, although it is slowed down by the gather operation at the end. Additionally, the performance drop at around 5000 elements is not fully explained yet; this could be explored further. There is also room for improvement – one optimization that could be implemented is to break up the computation into two parts, where the center rows are computed while the edge elements are being received, as to overlap computation and communication, and the edge elements are computed afterwards.

A Code listings

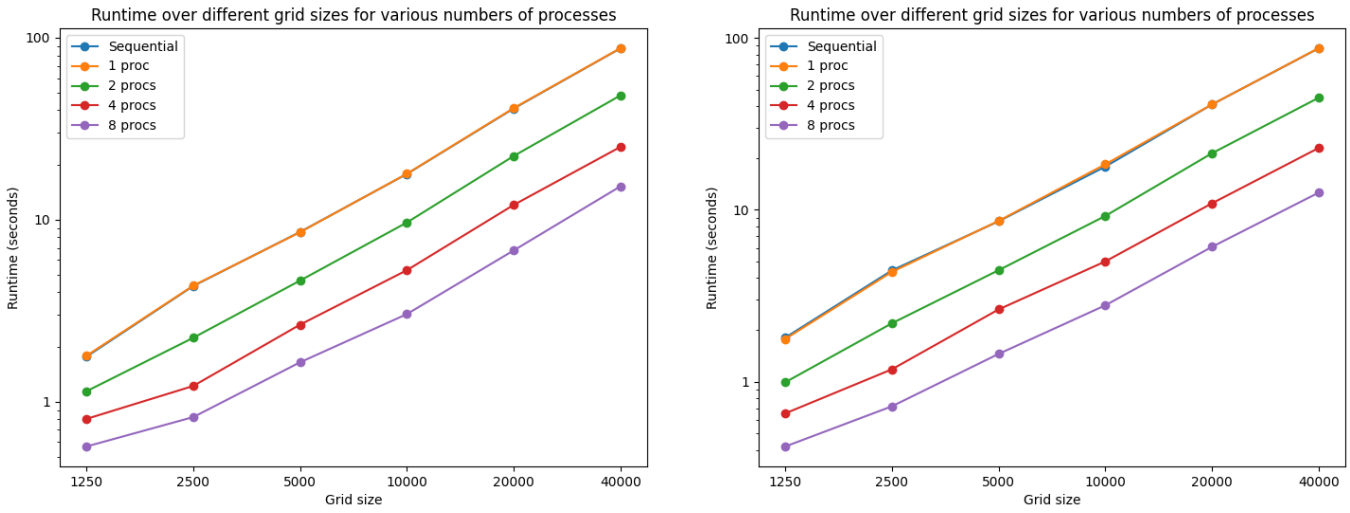
```c
/* Define a vector data type which represents a single-phase row
 * in the RB-SOR grid, i.e. N/2 doubles with stride 2 */
MPI_Datatype row_phase_t;
MPI_Type_vector(N / 2, 1, 2, MPI_DOUBLE, &row_phase_t);
MPI_Type_commit(&row_phase_t);
```

Listing 1: A definition for the `row_phase_t` datatype, holding a single phase of an RB-SOR row.

```c
MPI_Send(G[row] + 1 + (is_even(row) ^ phase), // Start of this phase.
         1,                                   // One element...
         row_phase_t,                         // ...of the vector type.
         ...)                                 // Destination, tag, communicator.
```

Listing 2: A send operation for a single phase of an RB-SOR row.

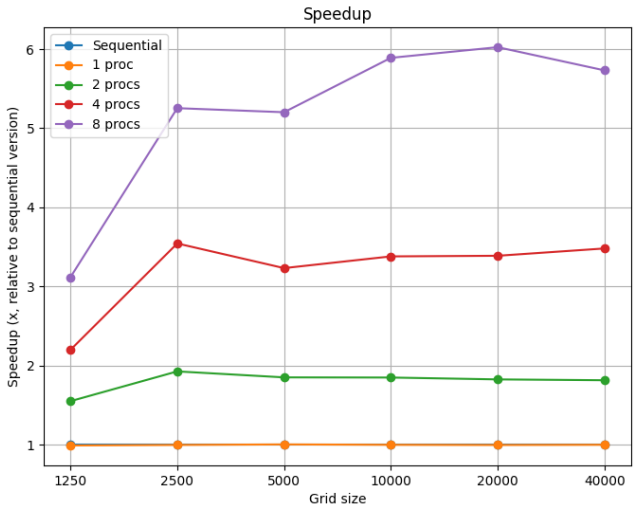
B Figures



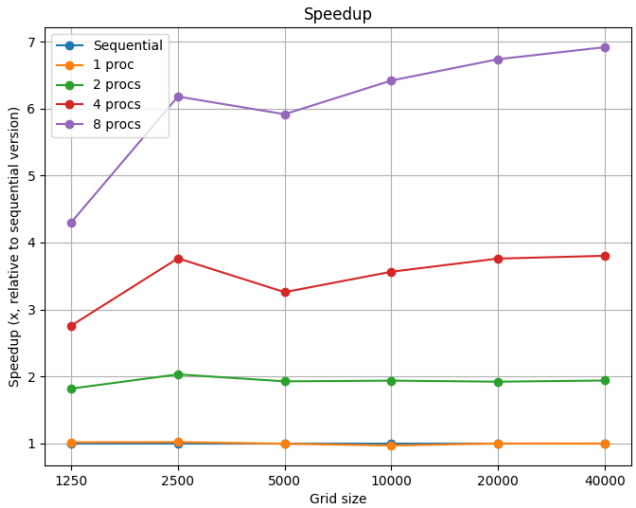
(a) including gather time

(b) excluding gather time

Figure 1: The runtime of the RB-SOR implementation under various grid sizes and numbers of processes.

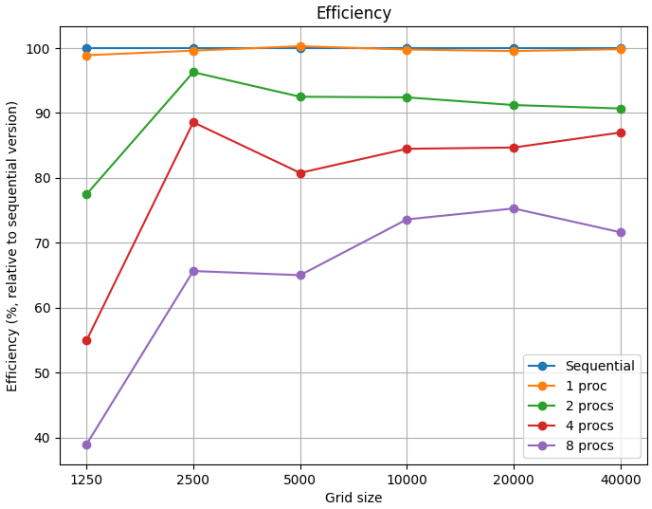


(a) including gather time

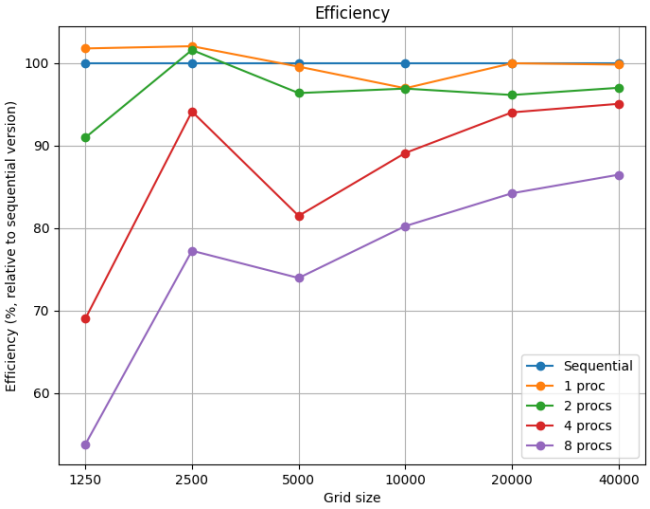


(b) excluding gather time

Figure 2: The speedup of the RB-SOR implementation under various grid sizes and numbers of processes.



(a) including gather time



(b) excluding gather time

Figure 3: The efficiency of the RB-SOR implementation under various grid sizes and numbers of processes.