Parallel computing - Game of Life

Andi Škrgat - r0876363

November 30, 2021

ROWS - total number of rows in the board
COLS - total number of columns in the board
N - total amount of points = ROWS*COLS
P - number of processors
NPROWS - number of processors in one row
NPCOLS - number of processors in one column
BLOCKROWS - number of rows of points in one processor
BLOCKCOLS - number of columns of points in one processor
n - number of points per processor
M - COLS or ROWS - depends on strip-wise partitioning strategy

Question 1

As partitioning strategy I used 2D block partitioning where each processor gets N / p data and processors are divided into $\sqrt(p) * \sqrt(p)$ grid. I used this sort of partitioning because in total there is less amount of data communicated when compared with strip-wise partitioning. However, with strip-wise partitioning we would have fewer messages communicated between processors during the algorithm's running and this would mean smaller startup cost so I used the assumption that startup time is not greatly dominating whole cost. This assumption holds when using processors from only one computer node which is enough for already quite large matrices(10^9 elements) I tested. For larger matrices, more computer nodes are going to increase startup cost and strip-wise partitioning would maybe be more appropriate. Communication overhead for 2D partitioning can be written in the following manner:

$$f_c = \frac{T_{comm}}{T_{calc}} \propto \frac{1}{\sqrt{n}}$$

We can see that communication overhead for 2D partitioning will be constant when n is constant but will increase(and with this, speedup and efficiency decrease) when number of

processors increase for same total amount of data.
Strip-wise partitioning(row-wise and column-wise)

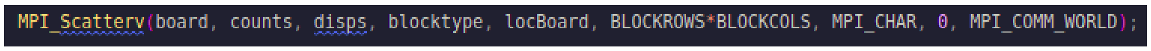
$$f_c = \frac{T_{comm}}{T_{calc}} \propto \frac{\sqrt{p}}{\sqrt{n}}$$

On the other side by using strip-wise partitioning communication overhead will increase even when n is constant but P grows and will increase even more when M is the same but p grows.

Question 2

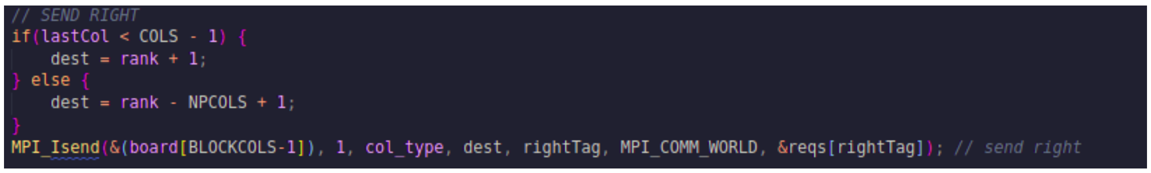
Processors are divided into 2D grid and it is not necessary that number of processors is squared number. Algorithm will first try to factorize number of processors e.g 6 into 2*3 so this means that after this step only prime numbers of processors are discarded. In the next step it tries to determine how many rows and columns should each processor have. For simplicity, if some processor should have different number of rows or columns, program ends. There are several approaches how one can solve this problem:

- Using MPI_alltoallw for completely general sending and receiving data
- Two step communication where we first scatter all except last row and then later scatter just last row.
- Scatter all the rows in a first phase, and scatter that data amongst the columns in a second phase



```
MPI_Scatterv(board, counts, disps, blocktype, locBoard, BLOCKROWS*BLOCKCOLS, MPI_CHAR, 0, MPI_COMM_WORLD);
```

Figure 1: Scatter local matrix array.



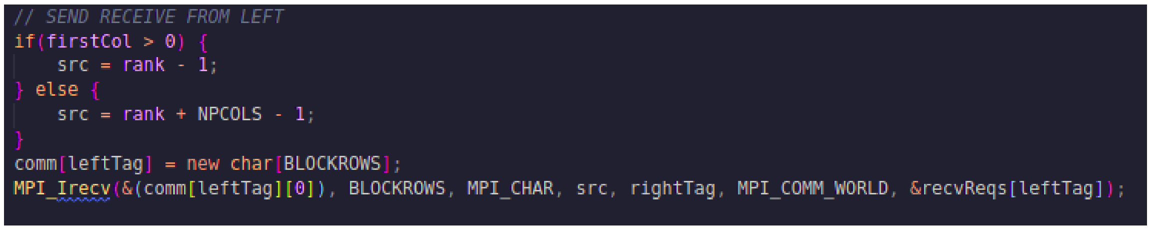
```
// SEND RIGHT
if(lastCol < COLS - 1) {
    dest = rank + 1;
} else {
    dest = rank - NPCOLS + 1;
}
MPI_Isend(&(board[BLOCKCOLS-1]), 1, col_type, dest, rightTag, MPI_COMM_WORLD, &reqs[rightTag]); // send right
```

Figure 2: Non-blocking send to right.

Question 3

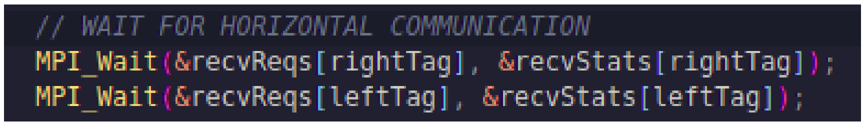
Communication goes as follows. Processor with rank 0 uses blocking send command to send to all other processors information about their first row and first col and other processors receive it with blocking receive. Processors then calculate displacements and call

MPI_Scatterv command to receive its local matrix. Each processor then enters the loop for number of iterations and in each iteration calls update board. Communication in this method is most important part of an algorithm. Each processor first sends non-blocking send request to up, down, left and right processor with its boundary elements. Up processor will receive first row, down processor last row, left processor first column and right processor last column of current processor. Processors also send its marginal elements to diagonal processors so processor i-1, j-1 will receive element at the position [0][0], processor i-1, j+1 will receive element at the position [0][BLOCKCOLS-1], processor i+1, j+1 will receive element at the position [BLOCKROWS-1][BLOCKCOLS-1] and processor i+1, j-1 will receive element at the position [BLOCKROWS-1][0]. I used non-blocking receive command to cover the case where some of the processors are slower than other so instead of waiting, each processor will use this time to update part of the local board that knows how to calculate. After it finished, it calls MPI_Wait command to wait for all receiving requests so it could update remaining, marginal part of the board. As last operation, it waits for all send requests to finish so every processor could have most updated version of other processors.



```
// SEND RECEIVE FROM LEFT
if(firstCol > 0) {
    src = rank - 1;
} else {
    src = rank + NPCOLS - 1;
}
comm[leftTag] = new char[BLOCKROWS];
MPI_Irecv(&(comm[leftTag][0]), BLOCKROWS, MPI_CHAR, src, rightTag, MPI_COMM_WORLD, &recvReqs[leftTag]);
```

Figure 3: Corresponding non-blocking receive from left.



```
// WAIT FOR HORIZONTAL COMMUNICATION
MPI_Wait(&recvReqs[rightTag], &recvStats[rightTag]);
MPI_Wait(&recvReqs[leftTag], &recvStats[leftTag]);
```

Figure 4: Synchronize receivings and sendings.

Question 4

MPI is a logical choice because for OpenMP we would have problem when some cores are in different iterations. Then we would need to use #pragma omp barrier to synchronize this.

Question 5

For assessing speedup and efficiency I run algorithms for 1000 iterations for several board sizes. Results are summarized in the plots below.

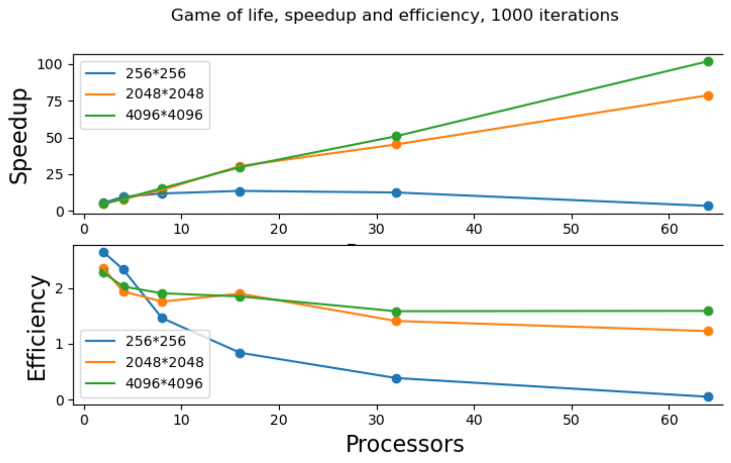


Figure 5: Speedup efficiency curve for smaller board sizes.

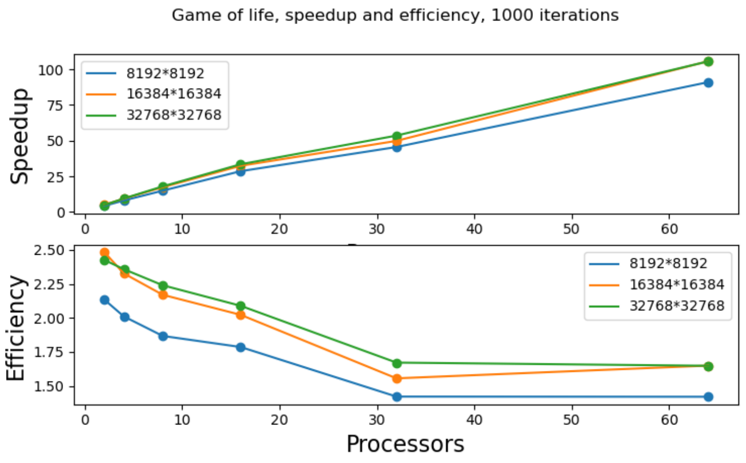


Figure 6: Speedup efficiency curve for larger board sizes.

In both cases we can see that after 32 processors we are not as efficient as before. One of the possible reasons could be that for running on 64 processors we have to use two computer nodes so there is higher communication cost because of more messages that are sent between processors.