This 2d stencil code was based on the previous 1d stencil code with only several differences. The first difference was the way to distribute data. There were 2 new parameters called Xprocs and Yprocs which indicate the number of processes on x dimension and on y dimension. I also declared xsize to be X_limit/Xprocs and ysize to be Y_limit/Yprocs. A small block of size xsize x ysize will be the matrix computed in each process.

The data was still read in the root process and stored in a X_limit x Y_limit sized matrix called life. Then I declared a 1d send buffer of length X_limit*Y_limit. I cut the life matrix into small blocks such that on each row there will be ysize blocks and on each column there will be xsize blocks. Then I store the elements inside the upper left block (with indices (0,0)) in row-major ordering into the first xsize*ysize space in the send buffer. Seconly, I put the elements in the next block with indices (0,1) into the send buffer. The blocks were also iterated using row-major ordering.

After filling the send buffer, the small blocks were sent to each process using MPI_Scatter function.

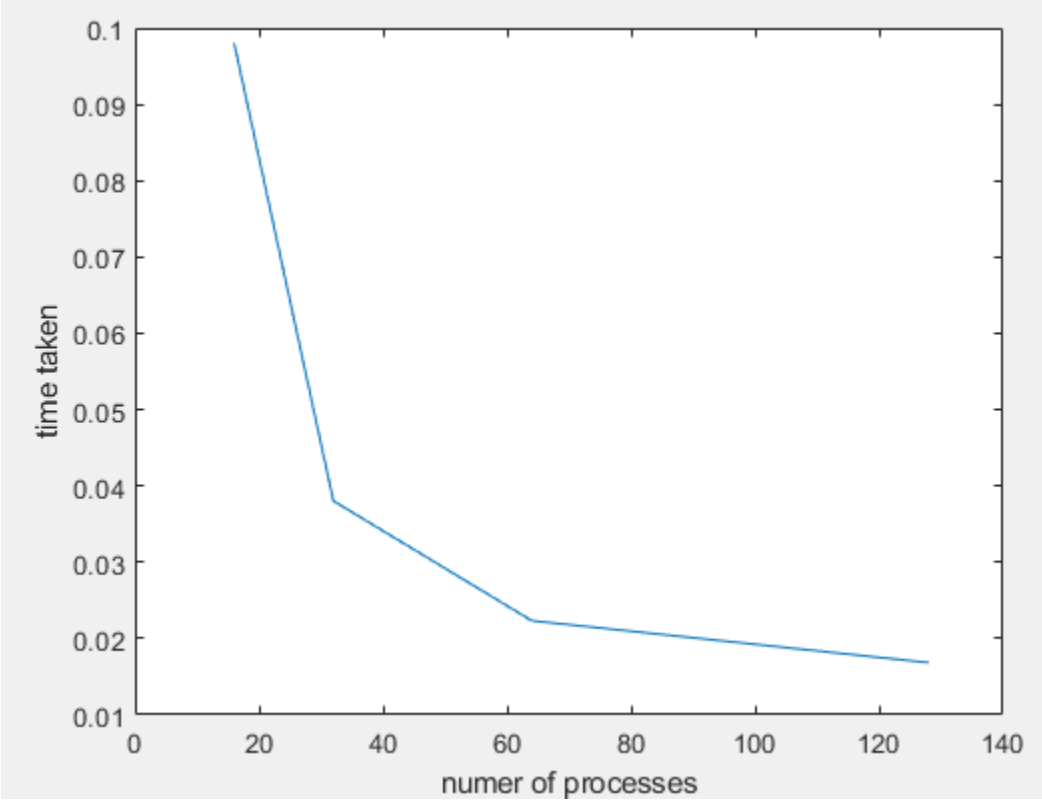
```
for (int numg = 0; numg < num_of_generations; numg++) {
  //if not first row
  if (myid >= Yprocs){
    //cout << "id: " << myid << "\n";
    MPI_Irecv(ydata2, ysize, MPI_INT, myid-Yprocs, 0, MPI_COMM_WORLD, &requests[0]);
    MPI_Isend(block[0], ysize, MPI_INT, myid-Yprocs, 0, MPI_COMM_WORLD, &requests[1]);
  }
  if (myid<numprocs-Yprocs){
    MPI_Irecv(ydata1, ysize, MPI_INT, myid+Yprocs, 0, MPI_COMM_WORLD, &requests[2]);
    MPI_Isend(block[xsize-1], ysize, MPI_INT, myid+Yprocs, 0, MPI_COMM_WORLD, &requests[3]);
  }
  if (myid % Yprocs != 0){
    MPI_Irecv(xdata2, xsize, MPI_INT, myid-1, 0, MPI_COMM_WORLD,&requests[4]);
    for (int i=0;i<xsize;i++){
      xsend1[i] = block[i][0];
    }
    MPI_Isend(xsend1, xsize, MPI_INT, myid-1, 0, MPI_COMM_WORLD, &requests[5]);
  }
  if(myid % Yprocs !=Yprocs-1){
    MPI_Irecv(xdata1, xsize, MPI_INT, myid+1, 0, MPI_COMM_WORLD, &requests[6]);
    for (int i=0;i<xsize;i++){
      xsend2[i] = block[i][ysize-1];
    }
    MPI_Isend(xsend2, xsize, MPI_INT, myid+1, 0, MPI_COMM_WORLD, &requests[7]);
  }
```

There are 4 if conditions to decide what a process should do within each generation. A process that was not at the boundary would send 4 messages to its neighbor and also receive 4 messages. If a process was at the boundary, it will send and receive fewer messages with respect to its specific position. Then I use MPI_Wait to ensure all the send and receive requests are done. Each process would only updated the

local block information after it successfully received the corresponding message.

```
if (myid<numprocs - Yprocs){
  MPI_Wait(&requests[2],MPI_SUCCESS);
  for (int i = 0; i < ysize; i++) {
    previous_block[xsize+1][i+1] = ydata1[i];}
  MPI_Wait(&requests[3],MPI_SUCCESS);
}
```

Finally, after all the processes finished their job, they need to send blocks back to the root process using MPI_Gather. The order was similar to that used before MPI_Scatter.



The performance of my code is shown in the plot above. As the number of processes increase exponentially, the time taken also tends to decrease exponentially. This is expected because the efficiency tends to be a little bit less than 1 because communications weaken some efficiency. Then it is reasonable to see that the ratio between speed up and number of processes is of O(1) .

The runtime using 4x16 alignment,16x4 alignment and 8x8 alignment are 0.0225191, 0.0223068 and 0.0223331 respectively. This means the alignment does change the performance so much.