MSc in High-Performance Computing

Message-Passing Programming

Coursework Part1

B142302

Introduction

Edition to the Source Code

In this 1D decomposition experiment, we utilise the C-MPI/automaton.c and insert explicit timing calls into the code to exclude the initialisation and IO overheads. The start time is recorded after the *allcell* initialisation and before the root processor scatters the cells. The end time is recorded after the root processor gathers all the pieces of *allcell* from other processors and before it writes the result to '*cell.pbm'*. The serial version of the 1D decomposition automaton is changed under the same idea.

For the density issue later discussed in the report, we added another timer to record the time on an every 100 iterations basis which simply makes a change to the print frequency *printfreq* to 100.

In order to test the time scale with the number of processors for the parallel version, the global variable NPROC in file '*automaton.h*' is changed to the number of processors to which we would like to split the work.

Test Environment

The tests are run on Cirrus by submitting the script to the sbatch workload manager. The seeds of allcell initialisation are fixed to 1234 for both versions. We tested on density=0.49 and density = 0.61.

1D-decomposed Case Study Evaluation

Correctness Test

Every result for the parallel automatons is exactly as same as the serial automaton at each step for our two density settings. The two output images are also exactly the same by comparing by:

```
diff C-MPI/cell.pbm C-SER/cell-ser.pbm
```

Performance Test

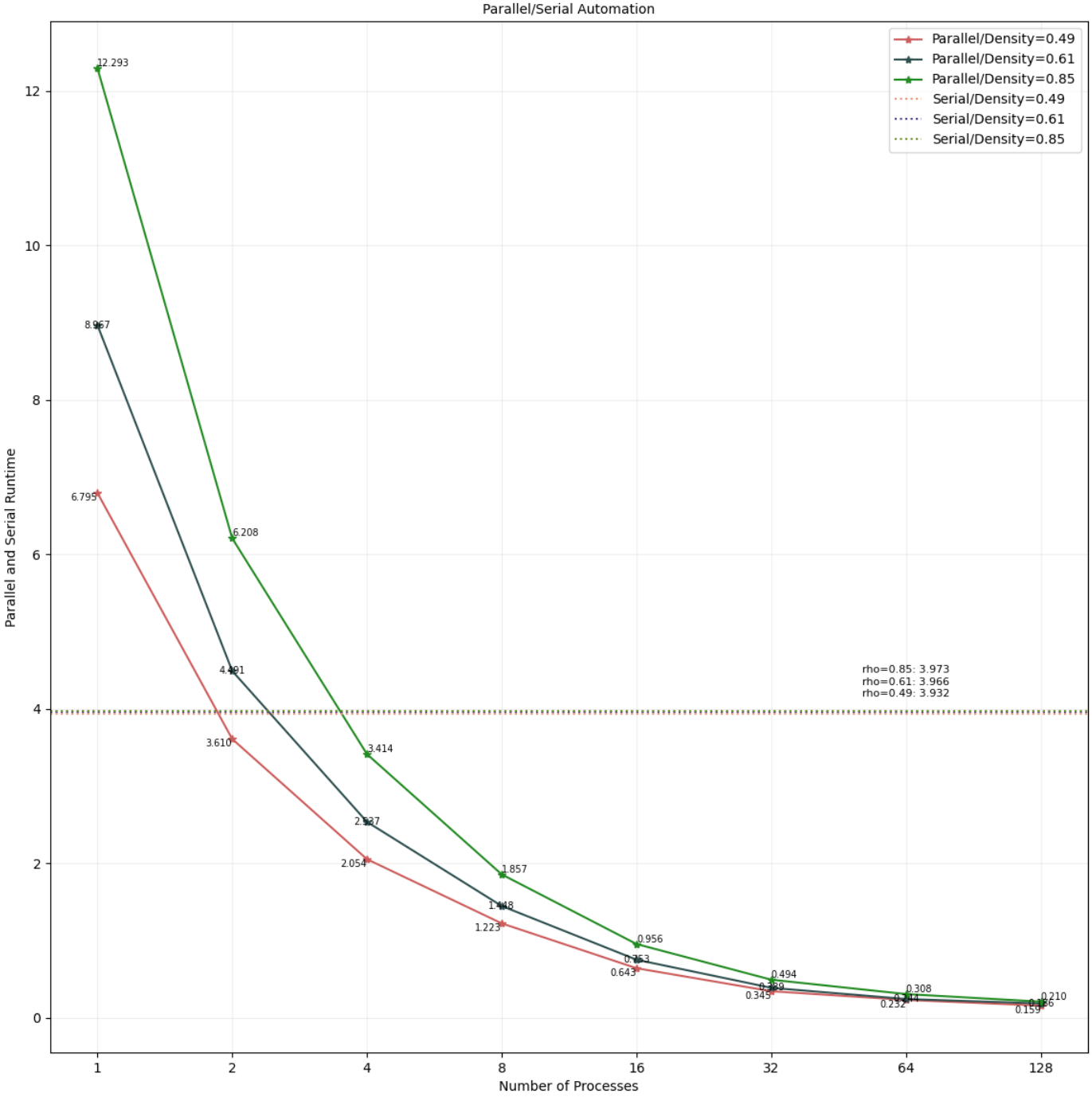
We compare the serials code runtime with different parallel settings.

NPROC	1	2	4	8	16	32	64	128	Serial
rho=49	6.795	3.610	2.054	1.223	0.643	0.345	0.232	0.159	3.932
rho=61	8.967	4.491	2.537	1.448	0.753	0.389	0.244	0.186	3.966
rho=85	12.293	6.208	3.414	1.857	0.956	0.494	0.308	0.210	3.973

Table1. Parallel runtime on different number of processors with serial runtime at different initial density.

Variant 1 Density

The density of initial cell density does not affect the serial runtime much. There's a slight increase in the serial runtime as the density of initial live cells increases. However, it has strongly pulled down the parallel performances. The percentage increase in parallel compilation time when



NPROC=2 is nearly directly proportional to the percentage increase in density. Even though the number of iterations and number of communicating processes does not change, the execution slows down. The increase in density has increased the problem size.

The conjecture is that:

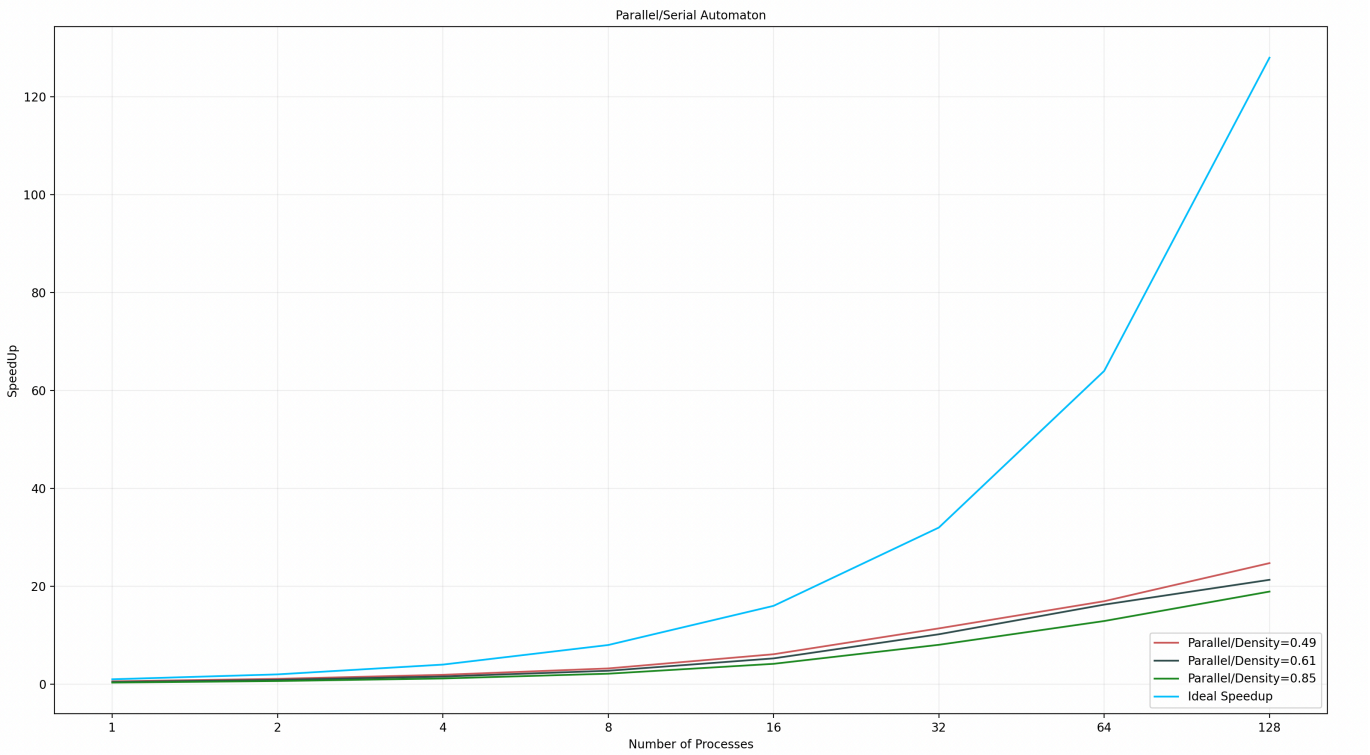
1) For serial code, there is an unbalanced path caused by the number of updates on *localcell*. The higher density that the initial alive cell is, the higher probability that the cell is alive in the following iterations. This causes a slight runtime rise.
2) For parallel code, the unbalanced path still exists. However, the major difference is caused by the increased number of intra-process communications. After the local process computes the number of alive cells in the current iteration, the number should be reported to rank0 to compute the total number of changes by MPI_Reduce. We found the interesting thing that even process calls MPI_Reduce( ) in every iteration, the communication won't happen if there's no change on the value pointed by the send address pointer. This means that for lower-density settings, the graphs are quicker, automating to a stable status which the live cell rarely updates.

Variant 2 Number of Parallel Processes

The tests are run from 1 parallel process to 128 parallel processes. As the number of processes increases, the execution time decreases. However, it takes longer to execute on only one parallel processor than executed by serial code when the initial cell density is 0.49. When we increase the density from 0.49 to 0.61 and 0.85, two parallel processes cannot run compete one single process work in serial.

NPROC	1	2	4	8	16	32	64	128
rho=49	0.579	1.089	1.914	3.215	6.115	11.397	16.948	24.730
rho=61	0.442	0.883	1.563	2.739	5.267	10.195	16.254	21.323
rho=85	0.323	0.640	1.164	2.139	4.156	8.043	12.899	18.919

Table2. The Speed Up of Parallel Execution of increasing number of processes



The speedup for parallel processing is increasing as the number of processes increases. The efficiency of using parallel processing for the lower-density graphs is higher. The reason is explained in the deduction above, the number of intra-processes communication increases. However, the parallel program speedup is far from the ideal speedup. The communication between ranks takes up a high amount of resources which pulls down the parallel performance.

2D-decomposed MPI Program Design

Program Structure

The basic idea is that we create a tree of ranks. The initial cells are split into slices. We scatter the data from ***allcell*** into ***slicecell***. Secondly, we split slices into dices ***dicecell*** that complete a 2D decomposition. Because the scatter can only handle continuous address space, so we use the matrix transpose of ***slicecell,*** scattering to the ranks in its child network of which the size of every message is LX*LY MPI_INT.

The source and destination address of halos swapping is calculated by introducing a Virtual Cartesian Topology. By assigning each rank a 2D coordination, they find their neighbours by looking up and down, left and right.

The gathering process is reverse engineering of the scattering process. The ***dicecell***s are first gathering

The steps to creating the first parallel program are as follows. Assume we have P_x* P_y numbers of processes.

1. LX = L/P_x, LY = L/P_y

2. Declarations:
   - Declare L*LX integer arrays called ***slicecell*** without halos,
   - Declare LY*LX integer arrays called ***dicecell*** without halos.
   - Declare (LX+2)*(LY+2) integer array called ***cell*** with halos,
   - Declare LX*LY integer array called ***neigh*** without halos.
   - Declare an L×L array ***allcell*** without halos

3. Initialise MPI, comm=***allranks*** compute the size and rank and check that size = Px*Py.

4. On the controller process, seed the random number generator from the command-line option and initialise ***allcell***

5. On the processors which have its rank modulo Px=0, initialise ***slicecell***. Check the total number of ranks has the ***slicecell*** initialised.

6. Scatter the map from the controller to the processors, which has its rank modulo Px=0. Here send_buffer=***allcell*** to receive_buffer=***slicecell***,

7. For each process rank=R, which has the ***slicecell*** initialised, create a child communication network comm=***allslices***, which only includes rank number {R,…, R+LY}.

8. Loop over the ***allslices***:
   - Calculate the matrix transpose of those initialised ***slicecell;***
   - Scatter the map from send_buffer=***slicecell*** to receive_buffer=***dicecell***,
9. Initialise ***cell*** by looping over i=1,LX; j =1,LY
   - ***cell****[i][j]* = ***dicecell****[j - 1][i - 1]*

10. Initialise the halos by zeroing the bottom, top, left and right halos of ***cell***.

11. *** Start looping over the steps ***
    - Swap the halos with (up to) four neighbours by an own function ***My_MPI_Sendrecv( ).*** First, swap the up and down halos and the message size for each LX*4; Second, read the left and right halos in line to 4 temporary-preserve-halos arrays. Third, swap those left and right halos, and the message size for each is LY*4.
    - Set ***neigh****[i][j]* = ***cell****[i][j]* + ***cell****[i - 1][j]* + ***cell****[i + 1][j]* + ***cell****[i][j - 1]* + ***cell****[i][j + 1]*
    - Loop over i=1,LX; j =1,LY
      if ***neigh****[i][j]* = 2, 4 or 5 :
        cell*[i][j]* = 1
        Increment the number of live cells
      else
        cell*[i][j]* = 0
    - Report the number of changes every 500 steps

    *** End loop over steps ***

13. Calculate the ***dicecell*** by looping over i=1,LX; j =1,LY
    - ***dicecell***[*j - 1*][*i - 1*] = ***cell***[*i*][*j*]

14. Loop over the ***allslices***:
    - Gather the map from send_buffer=***dicecell*** to receive_buffer=***slicecell,***
    - Calculate the matrix transpose of ***slicecell;***

15. Gather the map from send_buffer=***slicecell*** to receive_buffer=***allcell,***

16. write out the result by passing ***allcell*** to ***cellwrite( )***

Changes to the Program:

- Different to the previous 1-D decomposition, here we declare the size of ***allcell***[*LY*][*LX*] to benefit general C-writing habit.

- Create a Cartesian virtual topology **as an object in another file**. Each processor's rank has a cartesian coordination (x, y) of which rank = y + x * Px. Use ***MPI_Cart_coords( )*** to calculate the coordinate (x, y) of the rank in the virtual Cartesian Topology.

- Using ***MPI_Cart_shift( )*** to get the rank number of up-neighbour, down-neighbour, left-neighbour, and right-neighbour. Swap with those existing neighbours who have a rank number greater or equal to 0.

- In the file definition Cartesian Virtual Topology, using the idea of Virtual Cartesian Topology to generate a non-blocking send and receive wrapped function.

```
//Up and down halos swapping
for (i=0; I<Py; i++) {
   MPI_Isend(&cell[1][1], LX, MPI_INT, UP, tag, MPI_COMM_WORLD, &reqs[i]);
   MPI_Irecv(&cell[LX+1][1], LX, MPI_INT, DOWN, tag, MPI_COMM_WORLD, &reqs[i+4]);
   MPI_Isend(&cell[LX][1], LX, MPI_INT, Down, tag, MPI_COMM_WORLD, &reqs[i+8]);
   MPI_Irecv(&cell[0][1], LX, MPI_INT, UP, tag, MPI_COMM_WORLD, &reqs[I+12]);
   Calculate the next neighbouring information;
}
MPI_Waitall(4, reqs, stats);
      }
```

File Documentation

- Automaton.c
- Automaton.h
- Cartesian.c
- cellio.c
- unirand.c