COMP5426 Parallel and Distributed Systems

Semester 1, 2016

Assignment 1 Report

Ken Koh

SID: 312146310

1. Problem Definition and Requirements

The Red / Blue computation involves movement of two cell types in a two-dimensional grid according to a rule for each type. Grid cells may be red, blue or empty. The grid may be viewed as made up of a number of tiles which divides the grid size equally. The parameters for the problem are as follows:

- The size of the grid, *n*, which forms an *n x n* matrix
- The size of the tiles, *t,* where n is divisible by *t*
- The maximum ratio of either type of coloured cells, *c*, for any tile
- The maximum number of complete iterations, *num_iters*
- The number of processes to run the computation, *np*

For this problem, it is required to maximise the level of parallelism to divide the computation amongst each process as evenly as possible. The number of messages sent must also be minimised in order to reduce the overhead cost of using message passing in the algorithm. The design decisions to achieve this are detailed in the following section.

2. Parallel Algorithm Description

For this algorithm, the rows of the grid are first distributed evenly according to the tile size. For example, if *n = 9, t = 3* and *np = 2*, then the first process will receive two tiles (or 6 rows) and the last process will receive one tile. This is done to help balance the workload amongst the available processes and simplify checking the tiles at the end of the iteration. If the number of processes exceeds the number of tiles per row / column, the extra processes exit the program and a new MPI communicator is created for only the active processes.

Once the rows are distributed, each process completes the red computation by sequentially going through their local grid from left to right, top to bottom, and moving red cells to the right if there is an empty space. The local grid is then iterated over to clear any ' just moved' flags which indicate that a red cell moved out of an edge column.

Each process then sends the top row of its local grid to the process before it (ie the process with rank *r – 1,* except for the master and last process which must be handled as a corner case to wraparound). This row is stored in a temporary buffer so that each process has enough information to move the blue cells for its local grid. After receiving data into the buffer, the blue computation can be completed. This limits the amount of data that must be sent as each process only needs to receive and send 1 row. This also means that each process only communicates with its immediate neighbours for the red/blue computation step. For the blue step, each process can move a blue cell if there is an empty space below it. When checking the bottom row of the local grid, blue cells must be compared to the corresponding cell in the temporary buffer, as it represents the cell below.

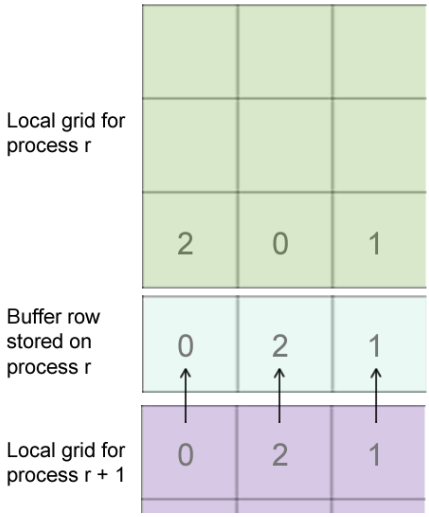

Fig 1. The top row is sent to process r – 1, so they can complete the blue step. It is sent back after the step.

Once the blue step is complete, each process must send back the results to the process it received the row from so that it can update the values of its top row for use in the next iteration. After the blue step, each tile must be checked to see if the number of blue or red cells exceeds the threshold for the maximum number of coloured cells in a tile. For this step, each process iterates through the local grid and counts the number of cells in the tiles they know about. Tiles are given an index from 0 to ($t^2 – 1$), as shown below (Fig 2).

These values are stored in two arrays, one for blue and one for red values. When a process gets to the bottom, right corner of a tile, it will check the threshold for that tile. If any tile exceeds the limit, a result integer -1 is stored.

The result integer is then all-reduced to get the minimum value. If it is -1, each process will end the loop and exit the program. Otherwise, they will go on to the next iteration.

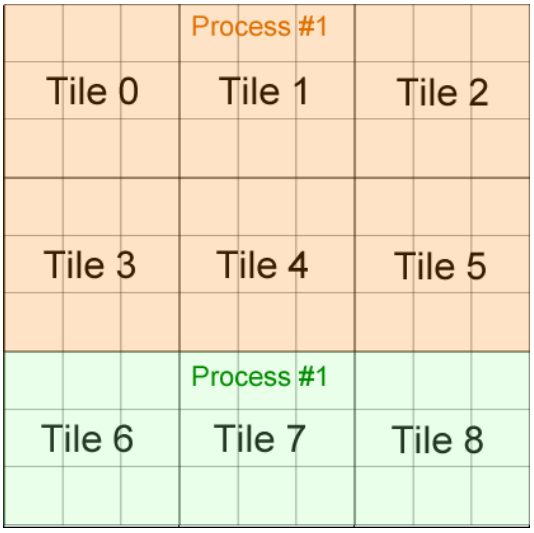

Fig 2. Example grid with n = 9, np = 2, t = 3

3. Parallel Algorithm Analysis

The costs of the algorithm in terms of number of messages and their size is analysed below.

Let p = number of processes, n = the grid size

Step	**Analysis**	**No. of messages**	**Message cost with size**
Dividing rows from master to each process.	Each row of size n must be sent to the owning process individually	n	n * n
Sending top row to the preceding process for the temporary blue step buffer	Each process *r* sends to process *r – 1* a row of size n	p	n * p
Receiving back the top row	Each process *r* sends to process *r + 1* a row of size n	p	n * p
All reduce the result of tile checking	All-to-one reduction, followed by a one-to-all broadcast. Only 1 integer is sent.	2p	2p * 1
Total		**4np + n**	$n^2 + 2np + 2p$

As the number of processes used is always less than or equal to n, this means that the costs are upper bounded by the grid size, n. Therefore the number of messages is O(n) and message cost with size is $O(n^2)$, which shows that message number grows linearly with grid input.

4. Implementation and Testing

The algorithm was tested using a number of different configurations and measured in terms of average CPU time consumed by the process. The number of max iterations was kept at 1 to minimise variation in number of runs between programs due to randomness of the generated grid.

np	n = 6	t = 3	c = 0.5	CPU time (Avg)	Min	Max
1				0.0000125	0.000029	0.00021
2				0.0034774	0.000126	0.0106
3				0.0127121	0.000112	0.045633
np	**n = 20**	**t = 5**	**c = 0.5**	**CPU time (Avg)**	**Min**	**Max**
1				4.43333E-05	0.000028	0.000061
2				0.006666333	0.000432	0.0418
3				0.0471	0.0107	0.0419
np	**n = 100**	**t = 5**	**c = 0.5**	**CPU time (Avg)**	**Min**	**Max**
1				0.000554	0.00022	0.001853
2				0.014324	0.000252	0.05762
5				0.036798	0.011919	0.089

The results showed a notable increase in CPU time as the number of processors increased. This reflects the expensive nature of communication for message passing between the different processes. However, the minimum and maximum values reported when running more than 1 process varied significantly. In some cases, multiple processes were able to complete the program faster than a single process, whilst in other cases multiple processes took many times longer to complete, as seen in n = 6. Scheduling for multiple processes and competing use of hardware may have contributed to this result. Further, the gap in execution time between different processes in the same program shows that processes were forced to wait for others in some cases.z

5. Manual

To run the program, it must first be compiled using the following command:

```
make redblue
```

After it has been compiled, you may run the program using the following structure:

```
mpirun –np <numberofprocesses> ./redblue <gridsize> <tilesize>
<terminating threshold> <maxiterations>
```

Parameter domains**:** t must divide n equally, c must be between 0 and 1 and max iterations must be greater than 0. For example:

```
mpirun –np 2 ./redblue 6 3 0.25 3
```

If there are any tiles which exceed the threshold, they will be printed to the console.

A debug version of the program which prints the final grid results and execution time can be compiled using:

```
make redbluedebug
```

After it has been compiled, it can be run using the following structure:

```
mpirun –np <numberofprocesses> ./redbluedebug <gridsize> <tilesize>
<terminating threshold> <maxiterations>
```

Appendix 1. Parallel Algorithm Pseudocode

If *rank* == master:

Initialise the global grid values

If *worldsize > 1:*

numrows = the number of rows this process will get

If *rank* == master:

For each process *p*:

Calculate the number of rows to send to each

MPI Send rows to *p*

Calculate the number of rows to be received

MPI Receive rows from the master into local grid

While *current iteration < max iterations*:

Move the red cells for the local grid rows

Update the local grid to clear the moved cell markers

Send the top row of the local grid to the previous process in the linear process array

Store the received row in a temporary buffer

Move the blue cells for the local grid rows, checking the temporary buffer to see if cells in the bottom row can move down

Update the local grid to clear the moved cell markers

Iterate through the local grid and count the number of red and blue cells for each tile. Tiles are assigned an index, starting from 0 in the top left corner, and increasing left to right.

All-reduce the result from the tile checking into *result*

If *result* == -1:

Break from the iteration loop

current_iteration++

Else:

// Solve the problem sequentially
