HW2-Report

Shengyuan Wang

CS445

Exploration of Serial Version

Here I type in the following code chunk to compile the serial version of the program, and try a tiny problem size to test the serial version.

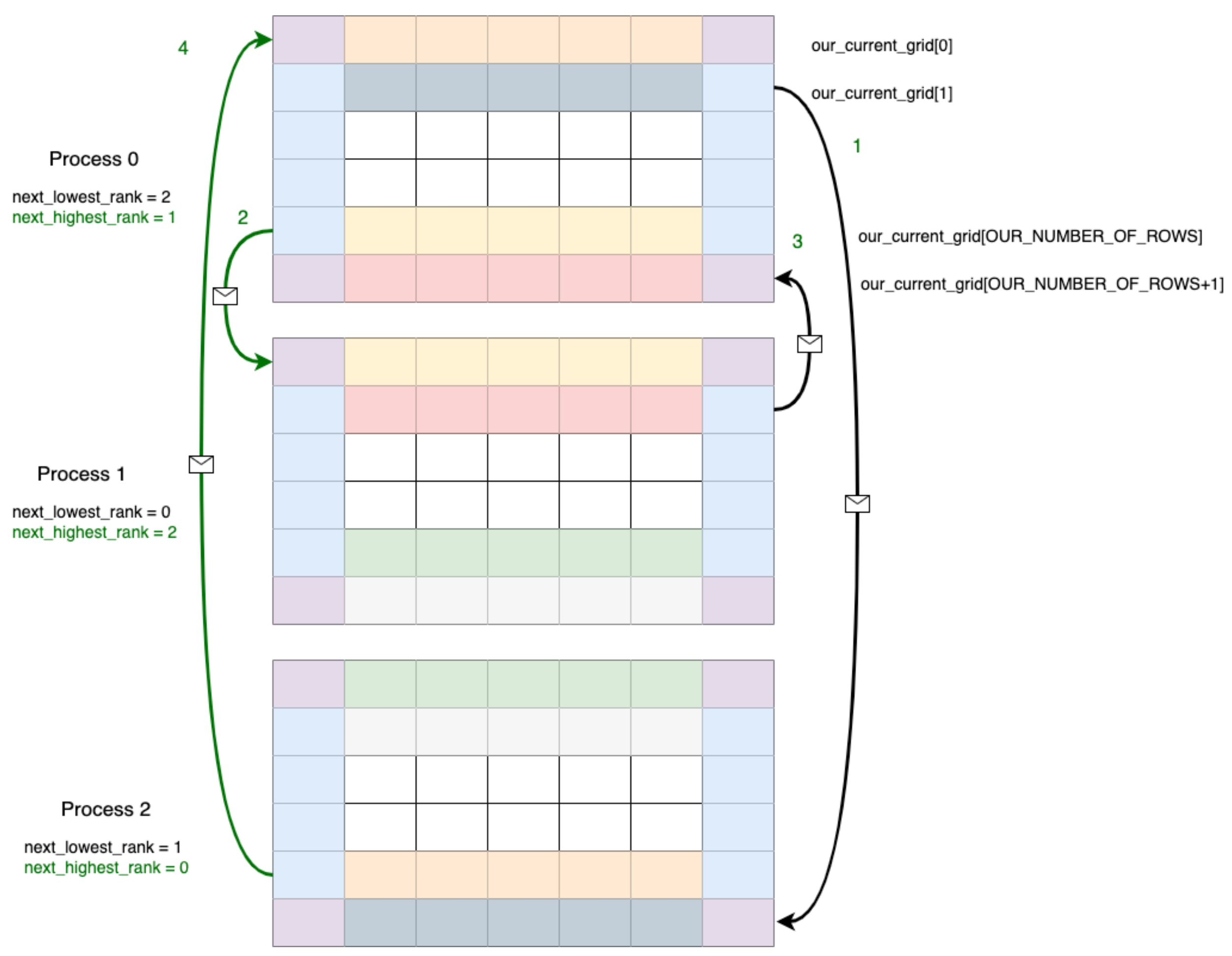
```
make life.serial
./life.serial -r 10 -c 10 -t 20
```

Methodology used for Parallelize

Before we do parallel, we first need to record the start time and the end time to calculate how much time it takes for the parallel chunk to run. Since the time we record starts with the first process beginning its work and ends with all processes finishing their work. For the start of timing, we decide to use a barrier here to signify a point where all processes must reach before continuing. As for the end of timing, we will use a barrier and conductor node 0 to get the end time.

```
MPI_Barrier(MPI_COMM_WORLD);
if (OUR_RANK == CONDUCTOR) {
  startTime = MPI_Wtime();
}

/***
* The parallel part
*/

MPI_Barrier(MPI_COMM_WORLD);
if (OUR_RANK == CONDUCTOR) {
  totalTime = MPI_Wtime() - startTime;
  printf("%f", totalTime);
}
```

Then, we will update the code for the parallel version. Since for “distribution memory” parallelism, in which execution flows keep their own memories, the data must be communicated through message passing. In this program, we need to separate the large grid into several small grids. Thus, we will need to use ghost rows of parts of the grids above and below it so that other processes are computing.



```
/* 10.1.1.  Send our second-from-the-top row to the process with the
  *  next-lowest rank */
exit_if((MPI_Send(our_current_grid[1], NUMBER_OF_COLUMNS + 2,
                MPI_INT, next_lowest_rank, 0, MPI_COMM_WORLD) !=
            MPI_SUCCESS),
        "MPI_Send(top row)", OUR_RANK);


/* TODO 10.1.2.  Send our second-from-the-bottom row to the process
  *  with the next-highest rank */
exit_if((MPI_Send(our_current_grid[OUR_NUMBER_OF_ROWS], NUMBER_OF_COLUMNS + 2,
                MPI_INT, next_highest_rank, 0, MPI_COMM_WORLD) !=
            MPI_SUCCESS),
        "MPI_Send(bottom row)", OUR_RANK);


/* TODO 10.1.3.  Receive our bottom row from the process with the
  *  next-highest rank */
exit_if((MPI_Recv(our_current_grid[OUR_NUMBER_OF_ROWS+1], NUMBER_OF_COLUMNS + 2,
                MPI_INT, next_highest_rank, 0, MPI_COMM_WORLD, &status) !=
            MPI_SUCCESS),
        "MPI_Recv(bottom row)", OUR_RANK);

/* TODO 10.1.4.  Receive our top row from the process with the
  *  next-lowest rank */
exit_if((MPI_Recv(our_current_grid[0], NUMBER_OF_COLUMNS + 2,
                MPI_INT, next_lowest_rank, 0, MPI_COMM_WORLD, &status) !=
            MPI_SUCCESS),
        "MPI_Recv(top row)", OUR_RANK);
```

In this way, we successfully set up the ghost rows, and the next step is to determine what the next grid is. In this part, we will utilize the four rules in the given document of the project.

- If a cell has fewer than 2 ALIVE neighbors, it will be DEAD in the next time step.
- If an ALIVE cell has 2 or 3 ALIVE neighbors, it will be ALIVE in the next time step.
- If a cell has more than 3 ALIVE neighbors, it will be dead in the next time step.
- If a DEAD cell has 3 ALIVE neighbors, it will be ALIVE in the next time step.

```
/* 10.4.3.  Apply Rule 1 of Conway's Game of Life */
if (my_number_of_alive_neighbors < 2)
{
    our_next_grid[our_current_row][my_current_column] = DEAD;
}

/* 10.4.3.  Apply Rule 2 of Conway's Game of Life */
if (our_current_grid[our_current_row][my_current_column] == ALIVE
        && (my_number_of_alive_neighbors == 2
            || my_number_of_alive_neighbors == 3))
{
    our_next_grid[our_current_row][my_current_column] = ALIVE;
}

/* 10.4.3.  Apply Rule 3 of Conway's Game of Life */
if (my_number_of_alive_neighbors > 3)
{
    our_next_grid[our_current_row][my_current_column] = DEAD;
}

/* 10.4.3.  Apply Rule 4 of Conway's Game of Life */
if (our_current_grid[our_current_row][my_current_column] == DEAD
        && my_number_of_alive_neighbors == 3)
{
    our_next_grid[our_current_row][my_current_column] = ALIVE;
}
```

In our parallel version program, I use Data Decomposition, Message Passing, Barrier, Multiple Instructions Multiple Data(MIMD) to enable parallel program. And to compile the code, we can manually type in the following code in terminal. If we want to print the result for each step, we can use this line

```
mpicc -DSHOW_RESULTS -DMPI life_mpi.c -o life.mpi
```

Otherwise, we can just have the run time by typing in

```
mpicc -DMPI life_mpi.c -o life.mpi
```

Or we can use makefile to compile the code

```
make life.mpi
```

For example, when we want to see the result and the run time of 2 step with 20 rows 20 columns data through 4 processes, we can use the following line.

```
mpirun -mp 4 ./life.mpi -r 20 -c 10 -t 2
```

And the terminal output will be

```
Time Step 0:
1 | 0 0 0 0 1 0 0 0 0 1 | 0
- - - - - - - - - - - - - -
1 | 1 0 1 1 1 1 0 0 1 1 | 1
0 | 0 1 0 1 1 0 0 0 0 0 | 0
1 | 1 0 1 1 0 0 0 1 1 1 | 1
0 | 1 0 0 0 1 1 1 0 1 0 | 1
0 | 1 1 1 1 0 1 0 0 1 0 | 1
1 | 1 0 1 0 0 1 0 0 0 1 | 1
1 | 1 1 0 1 0 1 0 1 1 1 | 1
0 | 0 1 0 1 0 1 0 0 1 0 | 0
1 | 1 0 0 0 0 0 1 1 0 1 | 1
1 | 0 0 0 0 1 0 0 0 0 1 | 0
- - - - - - - - - - - - - -
1 | 1 0 1 1 1 1 0 0 1 1 | 1

Time Step 1:
0 | 0 1 0 0 1 0 1 1 0 0 | 0
- - - - - - - - - - - - - -
1 | 1 1 1 0 0 1 0 0 1 1 | 1
0 | 0 0 0 0 0 1 0 1 0 0 | 0
0 | 1 0 1 0 0 0 1 1 1 0 | 1
0 | 0 0 0 0 0 1 1 0 0 0 | 0
0 | 0 0 1 1 0 0 0 1 1 0 | 0
0 | 0 0 0 0 0 1 0 1 0 0 | 0
0 | 0 0 0 1 0 1 0 1 0 0 | 0
0 | 0 1 0 0 0 1 0 0 0 0 | 0
1 | 1 0 0 0 1 1 1 1 0 1 | 1
0 | 0 1 0 0 1 0 1 1 0 0 | 0
- - - - - - - - - - - - - -
1 | 1 1 1 0 0 1 0 0 1 1 | 1

...

Time Step 9:
1 | 0 0 1 0 1 1 1 1 1 1 | 0
- - - - - - - - - - - - - -
0 | 0 0 1 0 0 1 1 1 1 0 | 0
0 | 1 0 0 0 1 0 0 0 1 0 | 1
1 | 0 0 0 0 1 0 0 0 0 1 | 0
1 | 1 0 0 0 0 0 0 0 0 1 | 1
1 | 0 1 0 0 0 1 0 1 1 1 | 0
0 | 0 0 0 1 0 0 0 1 1 0 | 0
0 | 0 0 0 0 0 0 0 0 0 0 | 0
0 | 0 0 0 1 1 1 0 1 1 0 | 0
0 | 0 0 0 0 0 0 1 1 1 0 | 0
1 | 0 0 1 0 1 1 1 1 1 1 | 0
- - - - - - - - - - - - - -
0 | 0 0 1 0 0 1 1 1 1 0 | 0

time: 0.004684 seconds
```

Then we can try a larger problem size.

```
./life.serial -r 1000 -c 1000 -t 100
```

And the output will be

```
time: 4.498801 seconds
```

Data Collection From Parallel version

In the program, I use the function `MPI_Wtime()` to get the start and end time for each part in order to find out how time used for generating the final time step. Besides, the modification to the parallel version of the program mentioned above, I wrote one shell script file to help myself collect the data.

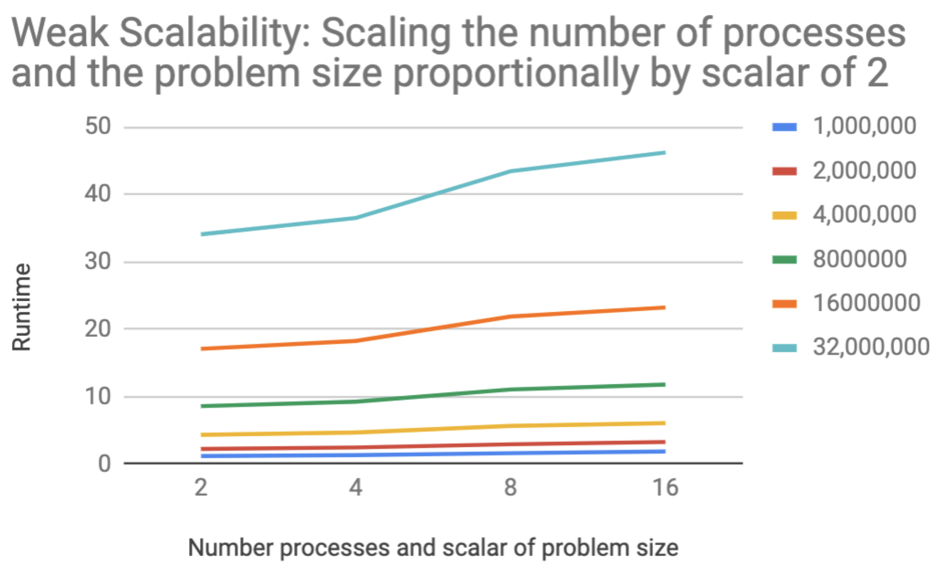
```
#!/bin/bash
num_times=$1
num_col=$2
weak_scale_lines=$3
num_row=$num_col
initial_size=$(($num_col*$num_row))
problem_size=$initial_size
double=0
printf "trial  \tproblem size \tprocesses \ttime\n"

for line in $(seq 1 $weak_scale_lines)
do
    echo "line: " $line

    for num_processes in 2 4 8 16
    do
        counter=1
        while [ $counter -le $num_times ]
        do
            printf "$counter\t$problem_size\t$num_processes\t"
            command="mpirun -np $num_processes ./life.mpi -r $num_row -c $num_col -t 100"
            $command
            printf "\n"
            ((counter++))
        done

        problem_size=$(($problem_size*2))
        num_col=$(echo $(echo | awk "{print sqrt($problem_size)}") | awk '{print int($0)}')
        num_row=$num_col
    done
    printf "\n"
    double=$((2**$line))
    problem_size=$(($initial_size*$double))
    num_col=$(echo $(echo | awk "{print sqrt($problem_size)}") | awk '{print int($0)}')
    num_row=$num_col
done
```

Our next goal is to try some experiments to show how this program behaves in terms of scalability for various problem sizes. Within the Linux shell script above, it can format the output data into tab-separated lines that can be written into a file.

```
bash ./run_weak_tests.sh 10 1000 6 > weak_test_whole.tsv
```

For the weak scalability test, the scrip created data designed for an existing spreadsheet. The number 10 is for the number of replications to run, and the number 1000 is the start size of row (or column), that is to say, the starting problem size is 1000*1000 = 1000000. The number 6 is the number of weak scalability lines to be graphed on the spreadsheet. Plus, in these experiments, we only run the program in 2/4/8/16 processors. For example, for line 1, the 4 cases are 1000*1000 problem size + 2 processes ; 1414*1414 problem size + 4 processes; 2000*2000 problem size + 8 processes; and 2828*2828 problem size + 16 processes.

Analysis of Scalability and Efficiency



From the graph and data above, we can find that the program demonstrates weak scalability for the problem size from 1,000,000 to 16,000,000. As for the problem size of 32,000,000, since the running time shifts from 34 seconds (problem size: 32,000,000; processors: 2) to 46 seconds (problem size: 256,000,000; processors:16), we can not say it has weak scalability for this problem size. So the reason why this happen is that as the problem size increasing, the data size of sending and receiving also increase, which will result in greater communication time. Both from the data result and the graph above, we can say Gustafson’s law come to rescue in terms of the efficacy of the MPI implementation.