A parallel algorithm for Schelling's model of segregation

Introduction

This algorithm uses a parallel programming approach, by exploiting distributed memory, for simulating Schelling's model of segregation. Its behavior allows it to satisfy all the agents in the grid in less time than usual by exploiting the concurrent execution over more processors or nodes.

The problem

In 1971, the American economist Thomas Schelling created an agent-based model that suggested inadvertent behavior might also contribute to segregation. His model of segregation shown that even when individuals (or "agents") didn't mind being surrounded or living by agents of a different race or economic background, they would still choose to segregate themselves from other agents over time! Although the model is quite simple, it provides a fascinating look at how individuals might self-segregate, even when they have no explicit desire to do so.

The problem's to create a simulation of Schelling's model by using a parallel programming approach, by exploiting distributed memory.

Definitions

There are two types of agents: (`B`)lue and (`R`)ed; and a cell can be populated or can be (`E`)mpty.

A `satisfied agent` is one that is surrounded by at least threshold `t` percent (30%) of agents that are like itself.

An `unsatisfied agent` is randomly moved `only` on a vacant location in the grid `owned by the corresponding processor`. This means if the grid `size`x`size` is divided by rows between *4* processors, when the agent in the cell *(0,0)* can be moved only in an empty cell in the first *`size`/4* rows.

The simulation is performed until `all agents are satisfied` or a maximum number of rounds `max_rounds` is reached.

The agents are initially placed into random locations of a neighborhood represented by a grid by assigning to the agent `x` the value = `{ B | (x % 2) == 0} ∪ { G | (x % 2) == 1}`.

The `MASTER` is the processor with the rank `0` and a generic `SLAVE x` has the rank = `{ p | p > 0 and p <= workers }` where `workers = P - 1` and `P` is the number of processors exploited.

The solution

The solution follows these steps:

1. The `MASTER` processor allocates the grid and place all the agents by using the `initialize_grid` and `initialize_agents` functions.

2. Until the max_rounds number is `NOT` reached `OR` all agents are `NOT` satisfied:

    1. The `MASTER` processor splits the grid rows over the `SLAVE` processors and `send`, by using the `MPI_Isend` routine, the corresponding portion to them. The grid is split by assigning `(size/workers)` rows to every `SLAVE` (except for the remaining rows that are assigned to the last `SLAVE` if the `size` is not divisible for `workers`).
    2. The `SLAVE` processor `receive`, by using the `MPI_Recv` routine, its protion of the grid and move the unsatisfied agents by using the `optimize_agents` function. Then it `send` back to the `MASTER` processor its modified grid.
    3. The `MASTER` processor `receives`, by using the `MPI_Recv` routine, from all the `SLAVE` processors their portions and updates its own grid.

3. The `MASTER` prints out the result of the simulation.

Implementation details

The MASTER code

A `partial` portion of the `MASTER` code is shown below; it emphasizes its behavior.

A `finished` variable, initially set to `0`, is sent to the `SLAVE` processors (*Row #6*) in order to take them awake and keep it receiving the grid's portions from the `MASTER` processor at every round. When the rounds have finished, or all the agents are satisfied, the `finished` variable will be set to `1` and sent to the `SLAVE` processors (*Row #12*) in order to sleep them down.

The `start_row` and `num_rows` variables are used to `send` the right rows to every `SLAVE` processor in order to give it the required data to determine agent satisfaction. This means if there are *3* `SLAVE` processors and the `size` of the grid is *9x9*, the grid will be sent as follow:

1. To the `SLAVE` *1* are sent the cells *(0, 0) -> (3, 8)* with a total of *3+1 = 4* rows. It requires *row 3* to determine the right satisfaction of the agents in *row 2*.
2. To the `SLAVE` *2* are sent the cells *(2, 0) -> (6, 8)* with a total of *1+3+1 = 5* rows. It requires *row 2* to determine the right satisfaction of the agents in *row 3*, and *row 6* to determine the right satisfaction of the agents in *row 5*.
3. To the `SLAVE` *3* are sent the cells *(5, 0) -> (8, 8)* with a total of *1+3 = 4* rows. It requires *row 5* to determine the right satisfaction of the agents in *row 6*.

The `start_row` and `num_rows` variables used to `send` and `receive` are different; in the `receiving` part of the `MASTER` processor, it is not considered the offset rows sent to the `SLAVE` processors, in order to substitute the correct cells of the `MASTER` processor grid.

```
initialize_grid(grid);
initialize_agents(grid, agents);

for (int round = 0; round < max_rounds && !all_satisfied; round++) {
    for (int i = 1; i <= workers; i++) {
        MPI_Isend(&(finished), 1, MPI_INT, i, MESSAGE_TAG, MPI_COMM_WORLD, 
&(requests[i - 1]));
        MPI_Isend(&(grid[start_row][0]), (num_rows * size), MPI_INT, i, 
MESSAGE_TAG, MPI_COMM_WORLD, &(requests[i - 1]));
        MPI_Isend(&(agents[0]), num_agents, MPI_CHAR, i, MESSAGE_TAG,
```

```
MPI_COMM_WORLD, &(requests[i - 1]));
    }
    MPI_Waitall(workers, requests, MPI_STATUSES_IGNORE);
    for (int i = 1; i <= workers; i++) {
        MPI_Recv(&(grid[start_row][0]), (num_rows * size), MPI_INT, i,
MESSAGE_TAG, MPI_COMM_WORLD, &status);
    }
    all_satisfied = all_agents_are_satisfied(grid, agents, size, size);
}

for (int i = 1; i <= workers; i++) {
    MPI_Isend(&(finished), 1, MPI_INT, i, MESSAGE_TAG, MPI_COMM_WORLD, &
(requests[i - 1]));
}
```

The SLAVE code

A `partial` portion of the SLAVE code is shown below; it emphasizes its behavior.

A `finished` variable is `received` from the SLAVE processor (*Row #1 and Row #7*) in order to take it awake and keep it receiving the grid's portions from the MASTER processor at every round. When the MASTER processor rounds have finished, or all the agents are satisfied in the MASTER processor grid, the `finished` variable `received` from the SLAVE processor will be equal to `1`, in order to sleep down the SLAVE processor (*exit from the `while` at Row 2*).

The `num_rows` variable used to `receive` and `send` are different; in the `sending` part of the SLAVE processor, it is not considered the offset rows received from the MASTER processor in order to `send` the correct cells to the MASTER processor.

The `optimize_agents` function (*Row 4*) is used to move the agents that are not satisfied in the SLAVE processor's grid.

```
MPI_Recv(&(finished), 1, MPI_INT, MASTER_RANK, MESSAGE_TAG,
MPI_COMM_WORLD, &status);
while (finished == 0) {
    MPI_Recv(&(grid[0][0]), (num_rows * size), MPI_INT, MASTER_RANK,
MESSAGE_TAG, MPI_COMM_WORLD, &status);
    MPI_Recv(&(agents[0]), num_agents, MPI_CHAR, MASTER_RANK, MESSAGE_TAG,
MPI_COMM_WORLD, &status);
    optimize_agents(rank, workers, grid, agents, start_row, 0, num_rows,
size);
    MPI_Send(&(grid[start_row][0]), (num_rows * size), MPI_INT,
MASTER_RANK, MESSAGE_TAG, MPI_COMM_WORLD);
    MPI_Recv(&(finished), 1, MPI_INT, MASTER_RANK, MESSAGE_TAG,
MPI_COMM_WORLD, &status);
}
```

The `optimize_agents` function

The `optimize_agents` function code, used in every `SLAVE` processor, is shown below; it receives in input a `SLAVE` processor's grid including the offset rows received from the `MASTER` processor.

The function iterates all the rows and cols of the `SLAVE` processor's grid where the agents have to be satisfied (*so, not including the offset rows*), and uses the offset rows received from the `MASTER` processor for verifying (*using the `is_satisfied` function*) the satisfaction on an agent.

At every iteration, the function decides to randomly move an agent (*using the `move_agent` function*) to another location of the `SLAVE` processor's grid (*not including the offset rows*) if the agent located in the cell `(i + start_row, j + start_column)` is not satisfied.

```
void optimize_agents(int rank, int workers, int **grid, char *agents, int
start_row, int start_column, int num_rows, int num_cols) {
    if (!has_free_cells(grid, start_row, start_column, num_rows,
num_cols)) {
        return;
    }
    for (int i = 0; i < num_rows; i++) {
        for (int j = 0; j < num_cols; j++) {
            if (grid[i + start_row][j + start_column] == -1) {
                continue;
            }
            if (!is_satisfied(grid, agents, start_row, start_column,
get_num_rows_of_worker(rank, workers), num_cols, i + start_row, j +
start_column)) {
                move_agent(grid, start_row, start_column, num_rows,
num_cols, i + start_row, j + start_column);
            }
        }
    }
}
```

The `is_satisfied` function

The `is_satisfied` function code is shown below; it receives in input a grid including its own size and the coordinates `(x,y)` of the cell to check.

The function `explores the neighborhood` of the cell received in input, considering the size of the grid (*Row 7 and Row 11*), and determines, considering the threshold `t` percent, if the cell received in input is satisfied (*Row 22*).

```
bool is_satisfied(int **grid, char *agents, int start_row, int
start_column, int total_num_rows, int total_num_cols, int x, int y) {
    if (grid[x][y] == -1) {
        return true;
    }
    int i, j, neighbors = 1, siblings = 0;
    for (i = x - 1; i <= x + 1; i++) {
        if (i < 0 || i > total_num_rows - 1) {
```

```
                continue;
            }
            for (j = y - 1; j <= y + 1; j++) {
                if (j < 0 || j > total_num_cols - 1 || (i == x && j == y)) {
                    continue;
                }
                if (agents[grid[x][y]] == agents[grid[i][j]]) {
                    siblings++;
                }
                if (grid[i][j] != -1) {
                    neighbors++;
                }
            }
        }
        if ((siblings * 100) / neighbors >= t) {
            return true;
        }
        return false;
    }
```

Execution Tutorial

This tutorial shows how to locally deploy and run the algorithm.

- **Prerequisites**
- **Repository**
- **Build**
- **Test the correctness**
- **Test the algorithm**

Prerequisites

- Docker and Docker Compose (Application containers engine)

Repository

Clone the repository:

```
$ git clone https://github.com/IvanBuccella/parallel-schelling-s-model-of-segregation
```

Build

Build the local environment with Docker:

```
$ docker-compose build
```

Test the correctness

Test the correctness using the local environment with Docker:

```
$ docker-compose up correctness
```

Test the algorithm

Test the algorithm with `strong scalability` using the local environment with Docker:

```
$ docker-compose up runner-strong
```

Test the algorithm with `weak scalability` using the local environment with Docker:

```
$ docker-compose up runner-weak
```

The correctness

The correctness container executes the algorithm several times, from using `2` processors to `24` processors. It saves the output of the `2` processor execution in the file `two-processors.txt` that is compared to the file `x-processors.txt` (which corresponds to the output of the x processor `{ x-processor | x > 2 }`) in order to find differences in the results.

Benchmarking

The benchmark tests, for weak and strong scalability, have been executed over 4 instances with 4 cores (e2-standard-4) using the Google Cloud Plaform.

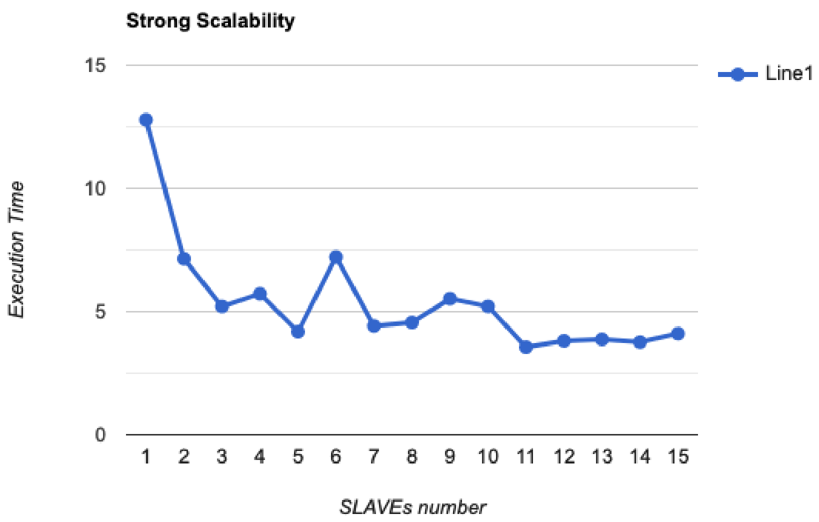
Strong scalability

Has been used a fixed grid size of `1900 x 1900`; the results are shown below.

Data reported are the average of three runs for every processor number change.

`SLAVE`s number	Relative Speed-up	Absolute Speed-up	Execution time
1	1.00	1.00	12.787611
2	1.78	1.78	7.144124
3	1.37	2.45	5.210431
4	0.91	2.23	5.722139
5	1.36	3.05	4.183748
6	0.57	1.77	7.215958

SLAVEs number	Relative Speed-up	Absolute Speed-up	Execution time
7	1.63	2.89	4.416550
8	0.96	2.80	4.557257
9	0.82	2.31	5.524770
10	1.05	2.45	5.213230
11	1.46	3.59	3.557619
12	0.93	3.35	3.812692
13	0.98	3.30	3.873288
14	1.03	3.40	3.760451
15	0.91	3.11	4.104836



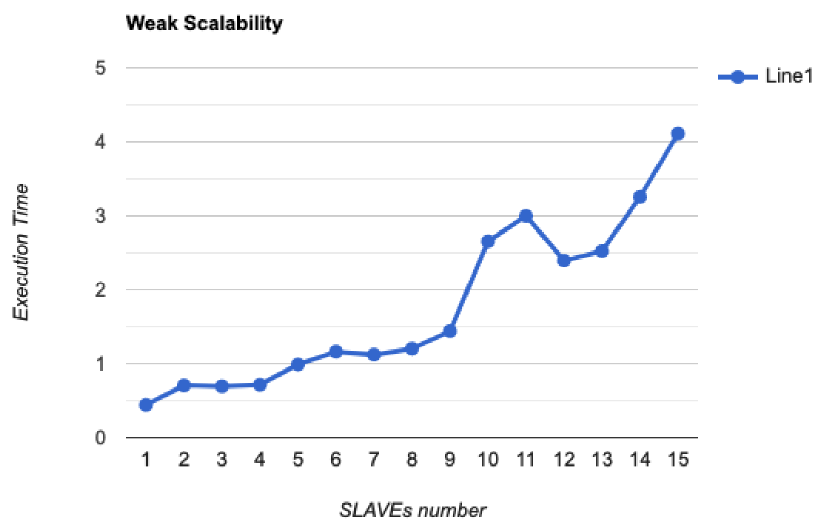
Weak scalability

Has been used a dynamic grid size starting from `500 x 500` to `1900 x 1900`, by augmenting the grid size of `100 x 100` for every new processor added; the results are shown below.

Data reported are the average of three runs for every processor number change.

SLAVEs number	Input size	Execution time
1	500	0.443342
2	600	0.706686
3	700	0.694799
4	800	0.715274
5	900	0.990585
6	1000	1.163156

SLAVEs number	Input size	Execution time
7	1100	1.122615
8	1200	1.203246
9	1300	1.440916
10	1400	2.653205
11	1500	2.998650
12	1600	2.393963
13	1700	2.522752
14	1800	3.254211
15	1900	4.111134



Conclusions

The analysis of the `strong scalability` graph shows an improvement in execution times as more processors are used. However, performance stabilizes by reaching the peak at `11 SLAVE` processors and then tends to degrade. This is due to the increase in the power of computation and to the smallest grid subdivision with the communication overhead between the processors reducing overall performance.

The analysis of the `weak scalability` graph shows that the execution time tends to augment with the augmentation of the number of processors and the input size per processor. This suggests an equal split of the workload between processors when the input size can be equally split among the processors.

In conclusion, the calculated accelerations show a sometimes negative and sometimes positive influence on the number of processors used; as the size of the grid and the number of processors do not always allow the grid to be equally divided among them causing a not-always-equal workload between processors. Another point to analyze is the `MASTER` doesn't optimize the agents on any portion of the grid in the current implementation; the code may be modified in order to keep the `MASTER` some work to do itself during waiting for the other portions of the grid from the `SLAVE` processors, in order to improve the performance.

Contributing

This project welcomes contributions and suggestions. If you use this code, please cite this repository.

Citation

Credit to Carmine Spagnuolo: Schelling's model of segregation & Ubuntu with OpenMPI and OpenMP & Docker image Ubuntu 18.04 OpenMPI.

Credit to Frank McCown: Schelling's Model of Segregation