Course Id:	CMPE 300
Course Name:	ANALYSIS OF ALGORITHMS
Term:	FALL 2018
Student Id:	2016400141
Student Name:	Halit Özsoy
Project Type:	Programming Project
Project Topic:	Parallel Programming with MPI
Submission Date:	26th December, 2018

icindekiler

INTRODUCTION

This project is about solving the problem of denoising noisy images. It's specifically implemented for images that are noised with respect to Ising model. Also, it's using the functions of Metropolis – Hastings Markov Chain Monte Carlo. What's more the solution is supposed to be running parallel by using MPI library. So, the actual problem is parallel image denoising.

Note that by images, black-white images (Images that consist of black and white pixels only) will be meant. And these black and white pixels will also be referred as -1 and 1, which is the representation of these images with text, provided in the project.

PROGRAM INTERFACE

REQUIRED ENVIRONMENT

In order to run this program, you must own a machine that implements POSIX standards. (It may be run in machines with linux or MacOS)

OpenMPI or a similar MPI library must be installed.

HOW TO COMPILE

- Open a terminal
- Change directory into the path of the project.
- Compile the program using command
  `mpicc -g denoiser.c -o denoiser`
-

HOW TO RUN

- Run the program using command (by default it will be run in grid mode)
  `mpiexec -n <nof_processors> denoiser <input_file> <output_file> <beta> <pi>`
- Optionally, it can run by the following command to run in row mode
  `mpiexec -n <nof_processors> denoiser <input_file> <output_file> <beta> <pi> row`

HOW TO TERMINATE

- Press `ctrl+c` on Linux or `cmd+c` on macOS in terminal.

INPUT AND OUTPUT

PROGRAM INPUTS

- `<nof_processors>` Nof processors must satisfy a special condition, let's say nof proccessors is `n`, then, in row mode the width and height of the image must be divisible by (`n-1`), in grid mode the width and height of the image must be divisible by `sqrt(n-1)` and `sqrt(n-1)` must be an integer.
- `<input_file>` A text file which includes a n * n black-white image described with 1 and -1 values Each row must be terminated by a line ending and each column must be separated by a space.
- `<output_file>` A file name to output denoised image, again as a n * n black-white image described with 1 and -1 values where each row is terminated by a line ending and each column is separated by a space.
- `<beta>` The beta value to be used as the beta value in the Metropolis-Hastings Markov Chain Monte Carlo functions.
- `<pi>` The pi value to be used as the pi value in the Metropolis-Hastings Markov Chain Monte Carlo functions.
- `row` That's an optional parameter to run the program in row mode, by default the program will be run in the grid mode. (An exact string "row" must be provided as the argument)

PROGRAM OUTPUT

- Program outputs the denoised version of the initial image to the given `<output_file>` path. It is the text version of a black-white image represented as a grid of 1 and -1 values.
- In addition, each slave process logs into stdout that it is working and tells how many million more iterations for it to complete.
- In addition, program logs its current status such as slave xx finished onto stdout.

IMPLEMENTATION DETAILS

- Instead of synching each process after each iteration by blocking calls, this implementation uses a much more advanced method for much faster program execution.
- Instead of using blocking MPI_Send and MPI_Recv methods, when the slaves are running their denoising logic, instead they use MPI_Isend and MPI_Irecv along with MPI_Test methods which are all nonblocking, meaning the proccesses can continue in their iterations without being blocked by sending / receiving messages when such messages are not needed.
- To achive such behaviour each slaves follows the below pseudo code:

```
begin:
selected_pixel = get_random_pixel()
askNeighbourProccessesForPixelInfo(selected_pixel);
while(!testNeighbourProccessesAnswerForPixelInfoReady(selected_p
ixel)) {
    answerNeighbourProccessesForAnyPixelInfo();
}
pixel_info =
readPixelInfoResponsesFromNeighbours(selected_pixel);
calculateAndMaybeFlipPixel(selected_pixel, pixel_info);

goto begin;
```

- Original code is more complex, but basically it uses `MPI_Isend` and `MPI_Irecv` methods in `askNeighbourProccesesForPixelInfo` step, and tests the requests in these `MPI_Isend` and `MPI_Irecv` in `testNeighbourProccessesAnswerForPixelInfoReady` step.
  Similarly it tests for previously initialized requests from `MPI_Irecv` to neighbours about asking about a pixel location in the current process in the `answerNeighbourProcessesForAnyPixelInfo();` and if the test is success (that neighbour did ask the current process about a pixel), sends a `MPI_Isend` with the calculated answer to that neighbour as well as reinitializing the request of `MPI_Irecv` to a new one for that neighbour so that future questions can be catched.

- While a process is waiting for answers from their neighbours, it keeps on answering any question sent to it, so it is never possible for any deadlock to happen.
- Also a process doesn't start to check another pixel until its current one is finished, so a process can at most send 1 question to all of its neighbours and it can be waiting for at most 1 question from each of its neighbours at any time, so that the neighbours can always catch the current request. (It is guarenteed that a neighbour process won't be sent a new question from the same process until the previous question gets answered)
- Overall, instead of waiting to sync after each iteration, the proccesses work asynchronously, yielding super fast execution times. (<1 second for 5million iterations on the sample input/output provided) as well as achieving 89% pixel similarity as average with the 640x640 sample input.

- Please, see the source code for more details.