CmpE300 - Analysis of Algorithms
Fall 2018
Programming Project

due: 26.12.2018 - 23:59

1 Introduction

In this project, you are going to experience parallel programming with C/C++ using MPI library. You will implement a parallel algorithm for image denoising with the Ising model using Metropolis-Hastings algorithm. Throughout the document you will probably encounter models and methods you have never seen before. There will be things related to probability. But don't be afraid and embrace it. Because, surprisingly we will end up with a very simple algorithm with a single line equation.

2 The Ising Model

The Ising model, named after the physicist Ernst Ising, is a mathematical model of ferromagnetism in statistical mechanics. The model consists of discrete variables that represent magnetic dipole moments of atomic spins that can be in one of **two states (+1 or -1)**. The spins are arranged in a graph, usually a lattice, allowing each spin to **interact with its neighbors**.

Did you understand how The Ising model works? No, me neither. But there are two important things to note in the definition that we wrote in bold.

- The Ising model models things that have two states (+1 or -1)
- Things interact with its neighbors

Not understanding how The Ising model works in physics won't stop us to apply it to images. If we assume that a black and white image is generated using The Ising model, then it means that *if we take random black pixel from the image, it is more likely that this pixel is surrounded by black pixels (same for the white pixels).* This is all we need to know about The Ising model. Now let's apply this to image denoising.

3 Image Denoising

Let the Z be a $I \times J$ binary matrix that describes an image where each Z_{ij} is either $+1$ or -1 (i.e. white or black). By randomly flipping some of the pixels of Z, we obtain the noisy image X which we are observing. We assume that the noise-free image Z is generated from an Ising model (parametrized with β) and X is generated by flipping a pixel of Z with a small probability π:

$$Z \sim p(Z \mid \beta) \tag{1}$$
$$F_{ij} \sim \text{Be}(\pi) \tag{2}$$
$$X_{ij} = (-1)^{F_{ij}} Z_{ij} \tag{3}$$

where

$$p(Z \mid \beta) \propto e^{-E(Z|\beta)} \tag{4}$$
$$E(Z \mid \beta) = -\beta \sum_{(i,j)\sim(k,l)} Z_{ij} Z_{kl} \tag{5}$$

Then we want to find the posterior distribution of Z:

$$p(Z \mid X, \beta, \pi) = \frac{p(Z, X \mid \beta, \pi)}{p(X \mid \beta, \pi)} \tag{6}$$
$$\propto p(Z, X \mid \beta, \pi) \tag{7}$$
$$= p(Z \mid \beta) p(X \mid Z, \pi) \tag{8}$$
$$\propto \exp\left(\gamma \sum_{ij} Z_{ij} X_{ij} + \beta \sum_{(i,j)\sim(k,l)} Z_{ij} Z_{kl}\right) \tag{9}$$

where $\gamma = \frac{1}{2} \log \frac{1-\pi}{\pi}$. In this formulation, higher γ implies lower noise on the X. Similarly, we expect more consistency between the neighbouring pixels if β is higher.

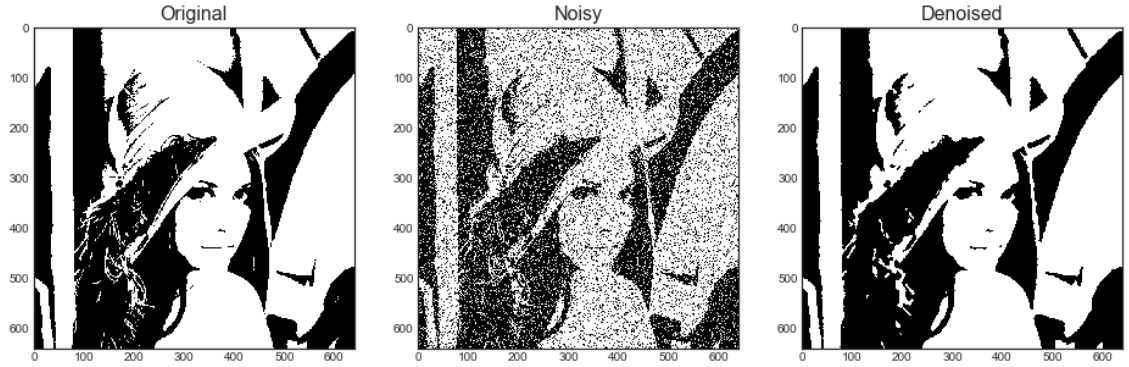
***"OK, what's going on"*, you thought probably. But, you really think that I'm on your side. Sorry, but these are the equations that you have to work with. ... ... Just kidding. Last part was true but let's go step by step and understand what is going on:**

1. There is a black and white image Z that we represent its pixels with $+1$ and -1 (i.e. black or white). We don't have this image and this is the image that we want to achieve.

2. There is another black and white image X. This is the noisy version of Z. By noisy we mean that, some of the pixels of the image Z are flipped. Equations 2 and 3 model this noise relation. π is the probability of a random flip. If $\pi = 0.1$ then ten percent of the pixels of Z will be flipped to achieve the noisy image X. We start with this noisy image X, and try to reach the noise-free image Z.

3. We assume that the image Z is generated from an Ising model.

4. By combining all these 3 information, we can statistically show our model as $p(Z \mid X, \beta, \pi)$. It can be read as the probability of Z given X, β and π. It returns a score for a candidate Z image that we propose. The score is higher if the candidate image is more likely to the original image (or more fit to the model). So it is a distribution over all possible Z images and we want to find the most probable Z image, in another word the Z image that gives the highest score from the posterior probability. However, this is not straight forward process because the posterior probability doesn't have a close form like Gaussian. Thus we can't find the most probable image easily. We have to try all possible Z images, but this is not efficient due to that there are $2^{I \times J}$ possibilities. Do we give up then? Of course not! Instead of finding the original image in an instant, we will approach to it step by step.



4 Metropolis-Hastings Markov Chain Monte Carlo

We have a function (the posterior distribution) that tells us which candidate Z image is more similar to the original image. Instead of choosing random images, it is obvious that we should start with the image X which is the only thing that is given to us. To reach the noise free image Z we need to make some modification to X. Metropolis-Hastings algorithm gives us a very simple approach to do this:

1. Choose a random pixel from the image X
2. Calculate an acceptance probability of flipping the pixel or not.
3. Flip this pixel with the probability of the acceptance probability that is calculated in the second step.
4. Repeat this process untill it converges.

That is basically all you need to do. Simple as I promised, right? Just one last equation that shows how to calculate the acceptance probability. And actually this one will be the only equation you are going to use. So every step we will only calculate how flipping a pixel will effect our outcome. As you may guess, we will do it by dividing the posterior distributions of two possibilities (flipping or not):

Assume a bit flip is proposed in pixel (i, j) at time step t, (a bit flip on $Z_{ij}^{(t)}$, i.e. $Z'_{ij} \leftarrow -Z_{ij}^{(t)}$), then

$$\alpha^{(t)} = \frac{p(Z' \mid X, \beta, \pi)}{p(Z^{(t)} \mid X, \beta, \pi)} \tag{10}$$

$$= \frac{\exp\left(\gamma Z'_{ij} X_{ij} + \beta \sum_{(i,j)\sim(k,l)} Z'_{ij} Z'_{kl}\right)}{\exp\left(\gamma Z_{ij}^{(t)} X_{ij} + \beta \sum_{(i,j)\sim(k,l)} Z_{ij}^{(t)} Z_{kl}^{(t)}\right)} \tag{11}$$

$$= \frac{\exp\left(-\gamma Z_{ij}^{(t)} X_{ij} - \beta \sum_{(i,j)\sim(k,l)} Z_{ij}^{(t)} Z_{kl}^{(t)}\right)}{\exp\left(\gamma Z_{ij}^{(t)} X_{ij} + \beta \sum_{(i,j)\sim(k,l)} Z_{ij}^{(t)} Z_{kl}^{(t)}\right)} \tag{12}$$

$$= \exp\left(-2\gamma Z_{ij}^{(t)} X_{ij} - 2\beta \sum_{(i,j)\sim(k,l)} Z_{ij}^{(t)} Z_{kl}^{(t)}\right) \tag{13}$$

Notice that, the summation is over the all the pixels Z_{kl} that are connected to a fixed Z_{ij}. For example, if the randomly chosen pixel was $(i, j) = (3, 4)$. Then $(k, l) = \{(2, 3), (2, 4), (2, 5), (3, 3), (3, 5), (4, 3), (4, 4), (4, 5)\}$

The pseudocode for the whole process is as follows and this is actually the only part you need to understand to do this project:

1. Initialize your β and π priors. Then $\gamma = \frac{1}{2} \log \frac{1-\pi}{\pi}$
2. Initialize $Z^{(0)} \leftarrow X$
3. At time step t:
   - 3.1. Randomly choose a pixel (i, j).
   - 3.2. Propose a bit flip on $Z_{ij}^{(t)}$, i.e. $Z'_{ij} \leftarrow -Z_{ij}^{(t)}$.
   - 3.3. Calculate acceptance probability $\alpha^{(t)} = \min\left\{1, \frac{p(Z'|X,\beta,\pi)}{p(Z^{(t)}|X,\beta,\pi)}\right\}$
   - 3.4. $Z^{(t+1)} \leftarrow Z'$, with probability $\alpha^{(t)}$
   - 3.5. $Z^{(t+1)} \leftarrow Z^{(t)}$, otherwise

Remarks:

1. We don't know the true values of β and π. According to our prior knowledge (by looking at the image or via our spidey sense) we are just throwing wild guesses. To remind what these values represent:
   - We expect more consistency between the neighbouring pixels if β is higher.
   - π is our original prior to show the noise probability. So higher π implies higher noise on the X. γ is like the inverse of π and we just come up with it to make equations more readable.
2. A probability can't be higher than 1. That is why we have min function on step 3.3. while calculating the acceptance probability.
3. Steps 3.4 and 3.5, demonstrate the accepting of a flip with the probability of accepting probability. The most simple method to do that is to select a number from the uniform distribution between 0 and 1. If the selected number is less than the accepting probability, then flip.
4. The method is guaranteed to converge if it runs enough iteration. But the enough iteration can be huge according to your priors and how noisy and big the image is.

1

-1	-1	1	1	1
-1	1	1	-1	1
-1	1	1	1	1
-1	-1	-1	1	-1
1	-1	-1	1	1

Initial noisy image X

2

-1	-1	1	1	1
-1	1	1	**1**	1
-1	**-1**	1	1	1
-1	-1	-1	1	1
-1	-1	-1	1	1

The Image after (t-1) iterations

3

-1	-1	1	1	1
-1	1	1	1	1
-1	-1	1	1	1
-1	-1	-1	1	1
-1	-1	-1	1	1

At iteration t we randomly chose the blue squared pixel

4

-1	-1	1	1	1
-1	1	1	1	1
-1	-1	1	1	1
-1	-1	-1	1	1
-1	-1	-1	1	1

To calculate the acceptance probability we need to know its neighbouring pixels

5

$$\alpha^{(t)} = \exp\left(-2\gamma Z_{ij}^{(t)} X_{ij} - 2\beta \sum_{(i,j)\sim(k,l)} Z_{ij}^{(t)} Z_{kl}^{(t)}\right) \text{ where } \beta = 0.4 \text{ and } \gamma = 1.0$$

6

$$\begin{aligned} \alpha^{(t)} &= \exp\left(-2 \times 1.0 \times 1 \times 1 - 2 \times 0.4 \times 1 \times (5 \times -1 + 3 \times 1)\right) \\ &= 0.67 \end{aligned}$$

With probability 0.67 we will flip this pixel from 1 to -1.
So this is the random part. Assume that we accept to flip.

7

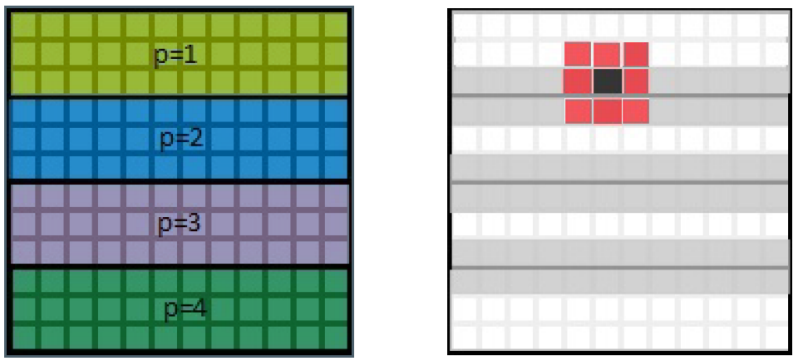
-1	-1	1	1	1
-1	**-1**	1	1	1
-1	-1	1	1	1
-1	-1	-1	1	1
-1	-1	-1	1	1

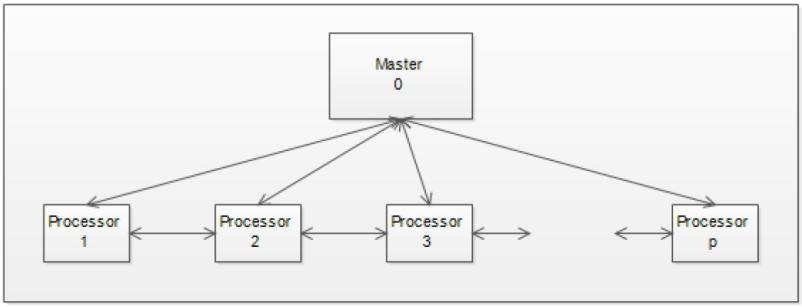
The image after the iteration t

5 Parallel Image Denoising

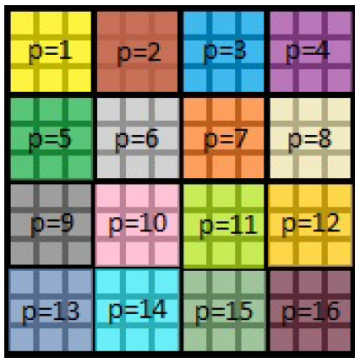
In this project, our aim is to implement the parallel image denoising using MPI. Two different parallel approaches will be discussed. We will assume that the image is a square of size $n \times n$. There will be 1 master and p slave processors and we will assume that n is divisible by the number of slave processors p.

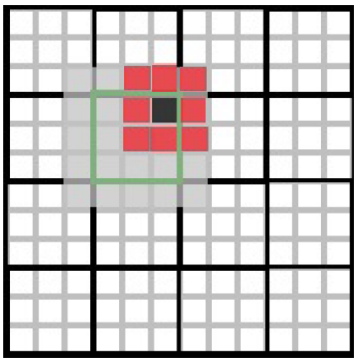
1. In the first approach, each processor is responsible from a group of n/p adjacent rows. Each processor works on $(n/p \times n)$ pixels and these pixels are stored locally by the processor. When a pixel is inspected, the processor should inspect all of its neighbors. When the pixel is not on the boundary of two adjacent processors, information about the neighbor pixels are present to the processor. When a pixel on the boundary is inspected, the processor needs to obtain information from the adjacent processor. Consider the black pixel in the below figure. Processor 1 should communicate with Processor 2 in order to learn the status of the south neighbor of the black pixel. Therefore, each processor should communicate with the adjacent processor at every iteration. Information about those neighbor pixels of the boundary can be stored locally and updated at every iteration.

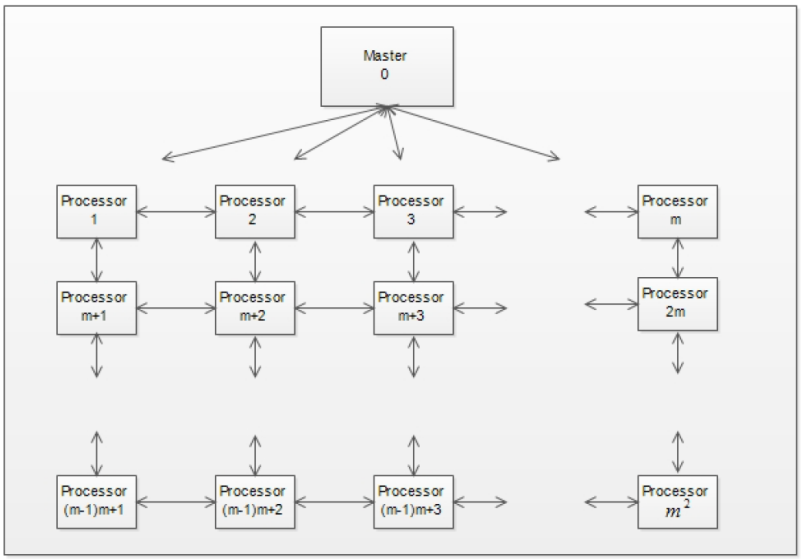




2. In the second approach, the grid is divided into $(n/p \times n/p)$ blocks and each process is responsible from a block. Now the updates are more trickier since each process has more than 1 adjacent process.







6 Implementation

1. For the MPI environment and tutorials please visit the web page. You can find many other tutorials in the web.

2. You are expected to implement the first approach. There will be bonus points for those who also implement the second approach.

3. The program will read the input from a text file and print the result in another text file. **The input text file will be a 2D array representation of a black and white noisy image**. The input file must only be read by master processor and distributed to slave (rest) processors by the master processor. The whole array should not be stored in each processor locally.

4. Start by distributing the input among the processesors and let each processor work on its pixels without any communication. Once you accomplish, add the communication.

5. Any functioning of the program regarding the whole program such as printing the output should be done by the master processor.

6. When two processors are exchanging information about the boundary cells, be careful to avoid deadlock. Before processing a pixel, you need to share boundary pixels between the processors. Also, before moving on the next iteration, make sure that all processors have finished their jobs.

7. The names of the input and output files and the values of β and π priors will be given on the command line, as well as the number of processes. An example execution for a Windows user that runs on 4 processors and uses input.txt and output.txt files with $\beta = 0.6$ and $\pi = 0.1$ would be:
mpiexec -n 4 project.exe input.txt output.txt 0.6 0.1

7 Important Nice Things

1. We prepared two python scripts for you:

   - **image_to_text.py**: Converts your image into noise-free project ready text input file.
   python image_to_text.py input_image output_file
   - **make_noise.py**: Converts your noise-free project ready text input file into noisy project ready text input file. The code takes *pi* as a parameter that determines the noise rate. Try with your own images.
   python make_noise.py input_file pi_value output_file
   - **text_to_image.py**: Converts your project ready text file into image. So you can see your noisy image or the output of your code.
   python text_to_image.py input_file output_image

2. If you say "*What is this wall of text with some fancy images? I need code!*". Then don't worry, we covered you too. We prepared a Jupyter Notebook for you which includes a **REAL WORKING SEQUENTIAL VERSION of THIS METHOD**. Hoorraay. Go check it out: `https://github.com/suyunu/Markov-Chain-Monte-Carlo`

3. You need to run the main loop for a fairly long time. In the example on the Jupyter Notebook, to completely denoise a $15\%(\pi = 0.15)$ noisy image with 640×640 pixels we needed to iterate for nearly 5 million times.

4. Because this is somewhat a random algorithm you may not end up with the same outputs every time. Also more importantly, as our assumption of the real image coming from the Ising Model which is not really true, you will not end up with the exact same image with the original one, but a very close one.

5. Most importantly, I highly recommend you to take the course **CmpE 548 Monte Carlo Methods** to learn more about stuff like that. It is fun.

8 Submission

1. The deadline for submitting the project is 26.12.2018 - 23:59. The deadline is strict. We will have demo sessions the following days. In the demo session, you are required to bring your own laptop and explain how you implemented the project.

2. This is an individual project. Your code should be original. Any similarity between submitted projects or to a source from the web will be accepted as cheating.

3. Your code should be suficiently commented and indented.

4. You should write a header comment in the source code file and list the following information.
   /*
   Student Name: Ali Veli
   Student Number: 2008123456
   Compile Status: Compiling/Not Compiling
   Program Status: Working/Not Working
   Notes: Anything you want to say about your code that will be helpful in the grading process.
   */

5. You must prepare a document about the project, which is an important part of the project. You should talk about your approach in solving the problem. Follow the guidelines given in the "Programming Project Documentation" link on `https://www.cmpe.boun.edu.tr/~gungort/informationstudents.htm`

6. Submit your project and document as a compressed archive (zip or rar) to Moodle. Your archive file should be named in the following format: "name_surname.ext". You don't need to submit any hard copies.

7. If you have any further questions, send an e-mail to *burak.suyunu@boun.edu.tr*