CmpE 300
Algorithm Analysis

Programming Project

Image Denoiser Implementation in a Parallel Computing Manner

Computer Engineering
Boğaziçi University
Turkey
26 December 2018

Contents

1 Introduction

This project is an image denoiser utilizing Ising Model using Metropolis-Hastings algorithm. The main goal of the project is to implement this algorithm in a parallel computing manner. For this purpose, MPI library is used with with C language.

2 Program Interface

This program does not provide a graphical user interface, but a command line interface. There are some prerequisites:

1. C compiler
2. MPI Library
3. A command line tool
4. (Optional) `make` tool.

Note that C compiler and MPI Library source are not required if a compiled version for the target computer exists. In order to run the program, some arguments should be given:

1. Name of the input file. (Please note that the raw image resolution should be 200px by 200px. This can be alterable, but requires rebuild of the code.)
2. Name of the output file.
3. β value.
4. π value.

To run the program, one can directly give input, or edit `Makefile` with intended configurations. Manual compilation is as following:

```
mpicc -g img_denoiser.c -o img_denoiser -lm
```

Manual running with arguments input file `yinyang_noisy.txt`, output file `output.txt`, $\beta = 0.4$ and $\pi = 0.2$ using 5 workers (4 child processes and 1 master):

```
mpiexec -n 5 ./img_denoiser yinyang_noisy.txt output.txt 0.4 0.2
```

In case the user want to terminate the program, `CTRL+Z` command should be used simply.

3 Program Execution

Execution of the program is quite handy for the end user even though it has no graphical user interface. It's advised to edit the `Makefile` and use `make`. After entering the parameters and saving `Makefile`, compile the code:

```
make
```

And run:

```
make run
```

4 Input and Output

Input file is a black and white representation of the image consisting of -1 and 1 values. All lines should have 200 values and there should be 200 lines. `.txt` files are acceptable but no file extension is strictly required.

Likewise, output file is also a 200 by 200 array of arrays having 40000 values. Initial values are changed according to program parameters and the Ising Model. In order to convert this values table to an image, some helper methods may be utilized.

```
-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 -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 -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 -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 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 -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 -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 -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 -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 -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 -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 ...
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 -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 -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 -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 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 -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 -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 -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 -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 -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 -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 ...
...
...
```

Listing 1: Example input file (clipped)

```
-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 -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 -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 -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 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 -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 -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 -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 -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 -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 -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 -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 -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 -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 -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 -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 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 -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 -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 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 ...
...
...
```

Listing 2: Example output file (clipped)

5 Program Structure

First, some definitions:

- β **value:** Beta is parameter is Ising Model and simply means how the image will behave more solid meaning that the more the beta value is the less noises there will be, yet the shape gets more distorted as beta increases. Beta values can be from 0 to any positive number.
- π **value:** Pi is an image specific value denoting the noise probability varying between 0 to 1 due to the fact that it's a probabilistic value. The more noisy the image is, the higher the value of π. It's worth to note that 0 value of π means no noise at all and gives the algorithm zero chance to denoise. On the contrary, 1 value of π means maximum noise probability, and in this case, the algorithm transforms almost all pixels to their opposite value.

The program was written in C language, and since C language has no object orientation, there no objects and structures but an array of arrays of score table used.

At the beginning of the program, the master takes given parameters and opens input file, transfers data in, outputs information to the console, divides and sends the partitioned data to child processes using `MPI_Send( args )` method. Whereas, child processes starts and waits for the partitioned data to be transferred. After competion of the transfer with `MPI_Recv( args )`, they detect where their positions are. This is important, because the rank of the process determines where they are going to transfer data such as the highest one cannot pass data to the upper one, since there is no process there.

The next step for child processes is that they start loop. All child processes have the same iteration count and they progress synchronously. A loop starts with sending the top line to the upper process (except the highest process) and the bottom line to the lower process (except the lowest process). The important point is that avoiding deadlocks, this is done by sending the data first and retrieving next.

After data exchange, all processes choose two random numbers, one for horizontal and one for vertical with min and max limits given to the `getRandom(int min, int max)` method.

```
int getRandom(int min, int max)
{return rand() % (max - min + 1) + min;}
```

At every iteration, the program calculates a score value:

```
double score = exp(- 2.0 * GAMMA * slice[y][x] * slice_orig[y][x]
           - 2.0 * BETA * slice[y][x] * neighbourSum);
```

And decides to a flip using a random value and score:

```
bool flip = flipRandomly(MIN(1, score), k);
```

`flipRandomly(double score)` works like this:

```
bool flipRandomly(double score)
{double r = (double)rand() / (double)RAND_MAX;
return r  score ?  true :  false;}
```

The completion of loops are non-trivial. When the loops finishes, all processes send their partitioned data to the parent back and parent merges and writes the data to the output file.

6 Improvements and Extensions

The program is running on multiprocessor architectures better. Execution of parallel algorithms on single core computers or using more processes than the actual core number of the computer makes it slower due to context switching time. For this reason, observance of great performance effects are not that much possible. Using computers having more cores definitely gives better run time results.

The program can be improved by making it size-generic meaning that it can accept not only 200px by 200px images, but all sizes. In C language, this requires dynamic allocation and some care to internal processes, which are doable.

One more improvement can be utilization of GPUs, as they can calculate numeric operations faster.

There may also be some improvements if the program can detect number of iterations itself. This might be done by utilizing an escape value which holds consecutive non flip count and finalizing the program if escape value passes some predefined or calculated value.

7 Difficulties Encountered

The main difficulty of the program is that using random numbers. The algorithm is heavily based on random numbers as all random selection of locations and probability to flip them are requires them. Generating uniform random numbers on computers are somehow a difficult problem. There are some better algorithms and libraries around, but no external libraries are allowed and used in this project. Alternatively, seeding the random generator method was tried using `srand(seed)`. Without seeding the `rand()` method, closely related numbers are returned. Seeding using time value is not a good idea, since the algorithm works in parallel, so even though processes run on different processors, they generate the same 'random values' and the output will have a repeating pattern. Adding or multiplying with the processor id of the process is also not enough as the output still has noticeable patterns in it. There may not be an ultimately perfect and simple way of doing this random number generation other than buying hundreds of thousand of them from industry-level random number sellers. Fortunately, using larger integers than the size of the image makes no possible pattern visible and that gives a persuasive solution. It also worths mentioning that using `srand()` in every loop is not a good idea, since `srand()` takes some time to process and makes the program noticeably slower.

Another difficulty is to select the best values for β, π and iterations count value. For minimalist images, higher β values give better results. However, if the original image has more details, high β is not a good choice. Iteration count should not be that much high, since after some time, it has diminishing effects. Detection of probabilistic value π is also somehow problematic, because it should be a close value to the noise level.

8 Conclusion

It was a worthwhile experience to have an experience on parallel programming. As the computing load increasing day by day and needs to process big data increases, parallel programs utilizing parallel or parallel-able algorithms are gaining importance.

9 Appendices

9.1 Makefile

```
all: img_denoiser.c
	mpicc -g img_denoiser.c -o img_denoiser -lm

run:
	time mpiexec -n 5 ./img_denoiser yinyang_noisy.txt output.
	    ↪ txt 0.4 0.2

show:
	time mpiexec -n 5 ./img_denoiser yinyang_noisy.txt output.
	    ↪ txt 0.4 0.2
	../scripts/env/bin/python ../scripts/text_to_image.py
	    ↪ output.txt output.jpg

clean:
	$(RM) img_denoiser
	$(RM) output.txt
    $(RM) output.jpg
```

9.2 img_denoiser.c

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>

/* Predefined values */
#define IMG_HEIGTH 200 // Image height
#define IMG_WIDTH 200 // Image width
#define ITERATIONS 500000 // Total number of iterations

/* Macro for minimum operations */
#define MIN(a,b) (((a)<(b))?(a):(b))

/* C does not have bool type, so defined here. */
typedef enum { false, true } bool;


/* Get random number between minimum and maximum values
    given as arguments. */
```

```
int getRandom(int min, int max) {
    return rand() % (max - min + 1) + min;
}

/* Get a random number between 0 and 1
    and decide to flip. */
bool flipRandomly(double score) {
    double r = (double)rand() / (double)RAND_MAX;
    return r < score ? true : false;
}


int main(int argc, char** argv)
{
    // Check for argument count
    if (argc < 4) {
        printf("Error:␣Too␣few␣arguments.\n");
        exit(1);
    }

    // Define constant values, get parameters
    const char* IMAGE_FILE = argv[1];
    const char* OUTPUT_FILE = argv[2];
    const double BETA = strtod(argv[3], NULL);
    const double PI = strtod(argv[4], NULL);
    const double GAMMA = (1.0/2.0) * log((1.0-PI) / PI) / log(2.0)
        ↪ ;

    // MPI initial routine for all processes
    MPI_Init(NULL, NULL);

    int world_size; // Number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    int world_rank; // Rank of the working process
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);


    int N = world_size-1; // Number of slaves
    int S = IMG_HEIGTH / N; // Lines for every process
    int i, j; // Used for for loops

    // Check if image is evenly distributable to the slave
        ↪ processes.
    // It not, give exception and exit with code 1.
    if (IMG_HEIGTH % N != 0) {
```

```
    printf("Invalid␣number␣of␣processes␣for␣height␣%d.\n",
        ↪ IMG_HEIGTH);
    exit(1);
}

if (world_rank == 0) {
    /* ------------------------- MASTER PROCESSES
        ↪ ------------------------- */
    printf("\n
        ↪ -------------------------------------------------------------\
        ↪ n");
    printf("|␣␣Input␣file:␣%s,␣output␣file:␣%s\n", IMAGE_FILE,
        ↪  OUTPUT_FILE);
    printf("|␣␣Parameters:␣Pi:%f,␣Beta:%f,␣Gamma:%f\n", PI,
        ↪ BETA, GAMMA);
    printf("|␣␣Total␣iterations:␣%d\n", ITERATIONS);
    printf("|␣␣Number␣of␣processes:␣%d␣+␣1␣master␣process\n",
        ↪ N);
    printf("|␣␣Iterations␣per␣process:␣%d\n", ITERATIONS/N);
    printf("
        ↪ -------------------------------------------------------------\
        ↪ n\n");

    // Main image array or arrays
    int img[IMG_HEIGTH][IMG_WIDTH];

    // Read data
    FILE *imgf;
    imgf = fopen(IMAGE_FILE, "r");

    if (imgf) {
        for (i=0; i<IMG_WIDTH; i++) {
            for (j=0; j<IMG_HEIGTH; j++) {
                fscanf(imgf, "%d", &img[i][j]);
            }
        }
    }

    // Send data to child processes
    for (i=0; i<N; i++) {
        for (j=i*S; j<(i+1)*S; j++) {
            MPI_Send(img[j], IMG_WIDTH, MPI_INT, i+1, 0,
                ↪ MPI_COMM_WORLD);
        }
    }
```

```
    /* Here, the master process waits for childs to complete
        ↪ their inputs to be complete. */

    // Collect data
    for (i=0; i<N; i++) {
        for (j=0; j<S; j++) {
            MPI_Recv(img[i*S + j], IMG_WIDTH, MPI_INT, i+1, 0,
                ↪ MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        printf("Process␣%d␣data␣completely␣received␣back.\n",
            ↪ i+1);
    }

    // Print data to file
    printf("Writing␣into␣%s\n", OUTPUT_FILE);
    FILE *of = fopen(OUTPUT_FILE, "w");
    if (of == NULL) {
        printf("Error␣opening␣output␣file.\n");
        exit(1);
    }

    for (i=0; i<IMG_HEIGTH; i++) {
        for (j=0; j<IMG_WIDTH; j++) {
            fprintf(of, "%d␣", img[i][j]);
        }
        fprintf(of, "\n");
    }
    printf("Written␣into␣%s\n\n", OUTPUT_FILE);

    /* -------------------------- END MASTER PROCESSES
        ↪ ------------------------- */
}
else {
    /* -------------------------- CHILD PROCESSES
        ↪ ------------------------- */


    int slice[S][IMG_WIDTH]; // Partitioned data, will be used
        ↪  for flips
    int slice_orig[S][IMG_WIDTH]; // Original copy of
        ↪ partitioned data

    // Receive data
    for (i=0; i<S; i++) {
```

```
    MPI_Recv(slice[i], IMG_WIDTH, MPI_INT, 0, 0,
        ↪ MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    for (j=0; j<IMG_WIDTH; j++) {
        slice_orig[i][j] = slice[i][j];
    }
}

// Determine if the process it at the top or at the bottom
int highest = world_rank == 1 ? true : false;
int lowest = world_rank == N ? true : false;

printf("Process␣%d␣(isHighest:%d,␣isLowest:%d)␣is␣starting
    ↪ ...\n", world_rank, highest, lowest);


int top[IMG_WIDTH], bottom[IMG_WIDTH]; // Used for
    ↪ received data
int flipCount = 0; // Used for statictical console output.

// Seed random method with a relatively higher number
srand(pow(world_rank, 4) + world_rank);


// Main iteration cycle for child processes
for (int k=ITERATIONS/N; k>0; k--) {

    if (!highest) {
        // Send to above process
        MPI_Send(slice[0], IMG_WIDTH, MPI_INT, world_rank
            ↪ -1, 1, MPI_COMM_WORLD);
    }

    if (!lowest) {
        // Receive from below process
        MPI_Recv(bottom, IMG_WIDTH, MPI_INT, world_rank+1,
            ↪ 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    if (!lowest) {
        // Send to below process
        MPI_Send(slice[S-1], IMG_WIDTH, MPI_INT, world_rank
            ↪ +1, 2, MPI_COMM_WORLD);
    }

    if (!highest) {
        // Receive from above process
```

```
    MPI_Recv(top, IMG_WIDTH, MPI_INT, world_rank-1, 2,
        ↪ MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

// Pick a random cell
int y = getRandom(0, S-1);
int x = getRandom(0, IMG_WIDTH-1);

// Logical values if randomly selected cell has a
    ↪ critical point
bool atTop = y == 0;
bool atBottom = y == S-1;
bool atLeft = x == 0;
bool atRight = x == IMG_WIDTH-1;

/* Cell map:
[0] [1] [2]
[7] [C] [3]
[6] [5] [4] */
int neighbours[8];
int neighbourSum = 0;

// Fill top values
if (highest && atTop) {
    // The process is at the highest line, so no line
        ↪ above.
    neighbours[0] = 0;
    neighbours[1] = 0;
    neighbours[2] = 0;
}
else if (atTop) {
    // Use received data
    neighbours[0] = atLeft ? 0 : top[x-1];
    neighbours[1] = top[x];
    neighbours[2] = atRight ? 0 : top[x+1];
}
else {
    // Else, use self data
    neighbours[0] = atLeft ? 0 : slice[y-1][x-1];
    neighbours[1] = slice[y-1][x];
    neighbours[2] = atRight ? 0 : slice[y-1][x+1];
}

// Fill bottom values
if (lowest && atBottom) {
```

```
    // The process is at the lowest line, so no line
        ↪ below.
    neighbours[6] = 0;
    neighbours[5] = 0;
    neighbours[4] = 0;
}
else if (atBottom) {
    // Use received data
    neighbours[6] = atLeft ? 0 : bottom[x-1];
    neighbours[5] = bottom[x];
    neighbours[4] = atRight ? 0 : bottom[x+1];
}
else {
    // Else, use self data
    neighbours[6] = atLeft ? 0 : slice[y+1][x-1];
    neighbours[5] = slice[y+1][x];
    neighbours[4] = atRight ? 0 : slice[y+1][x+1];
}

// Fill right value
neighbours[3] = atRight ? 0 : slice[y][x+1];

// Fill left value
neighbours[7] = atLeft ? 0 : slice[y][x-1];

// Sum neighbours
for (i=0; i<8; i++) {
    neighbourSum += neighbours[i];
}

// Calculate probability
// Note that here, double values are utilized.
double score = exp(-2.0 * GAMMA * slice[y][x] *
   ↪ slice_orig[y][x] - 2.0 * BETA * slice[y][x] *
   ↪ neighbourSum);

// Choose for a flip
bool flip = flipRandomly(MIN(1, score));

if (flip) {
    // Flip the value
    slice[y][x] *= -1;

    // Add to flip count
    flipCount++;
}
```

```
    }

    // Send result data to the master process
    for (i=0; i<S; i++) {
        MPI_Send(slice[i], IMG_WIDTH, MPI_INT, 0, 0,
            ↪ MPI_COMM_WORLD);
    }

    printf("%d␣iterations␣finished.␣%d␣(%f%%)␣were␣flipped.\n"
        ↪ , world_rank, flipCount, (double)flipCount /
        ↪ ITERATIONS * 100);

    /* -------------------------- END CHILD PROCESSES
        ↪ -------------------------- */
  }

  MPI_Finalize();
  return 0;
}
```

10 Examples

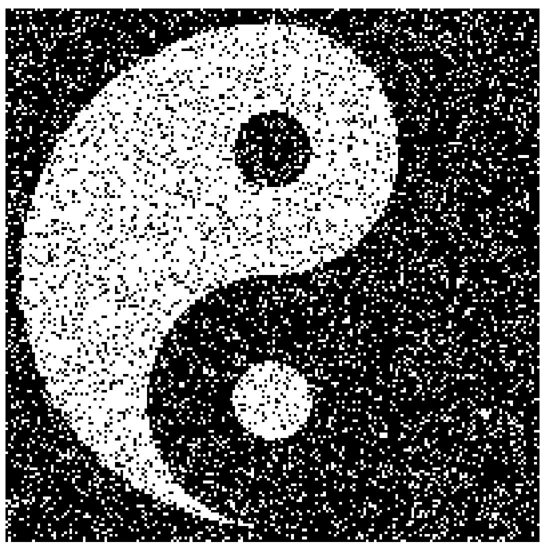

Figure 1: Noisy sample of `yinyang.png`

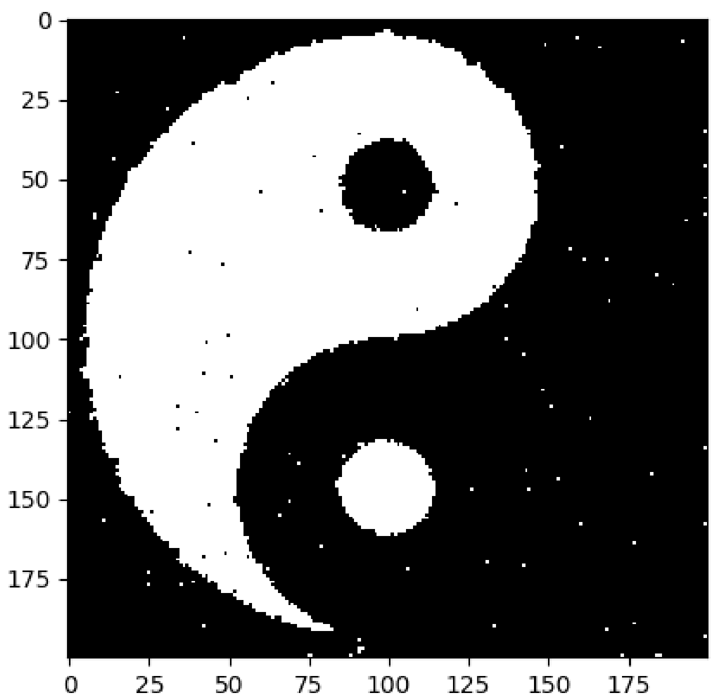

Figure 2: Denoised `yinyang` with $\beta = 0.4$ and$\pi = 0.2$

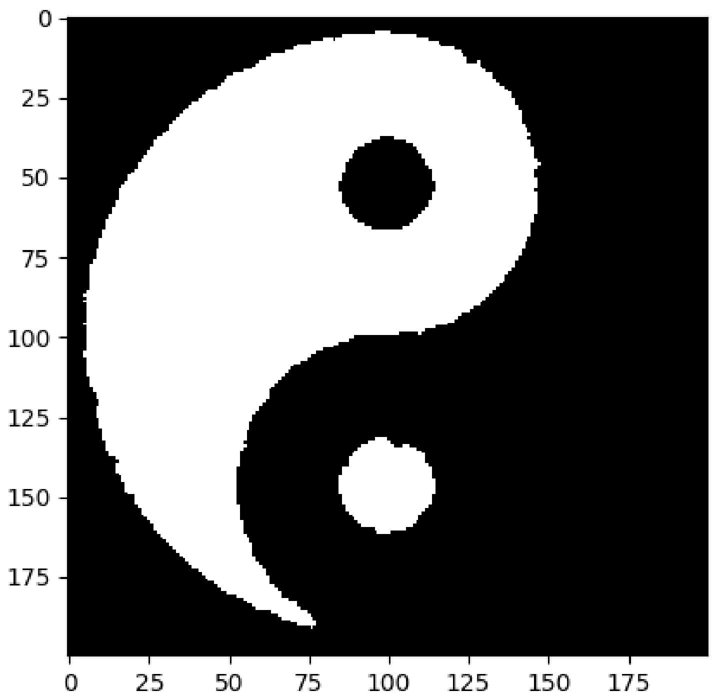

Figure 3: Denoised `yinyang` with $\beta = 0.9$ and$\pi = 0.2$

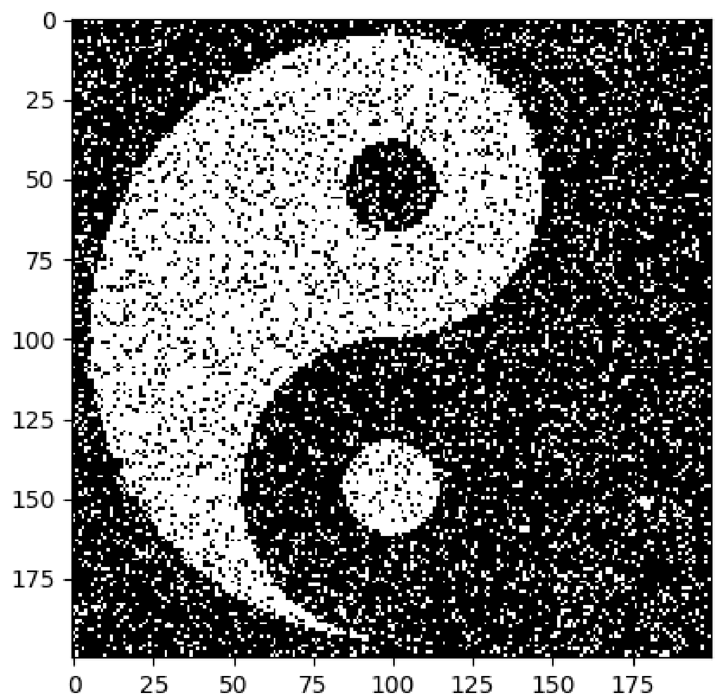

Figure 4: Denoised `yinyang` with $\beta = 0.4$ and$\pi = 0.001$

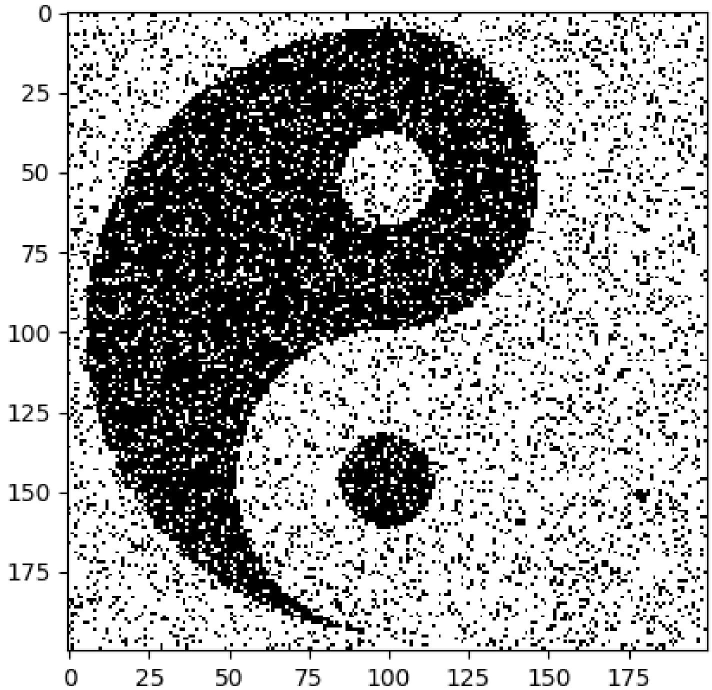

Figure 5: Denoised `yinyang` with $\beta = 0.4$ and$\pi = 1.0$

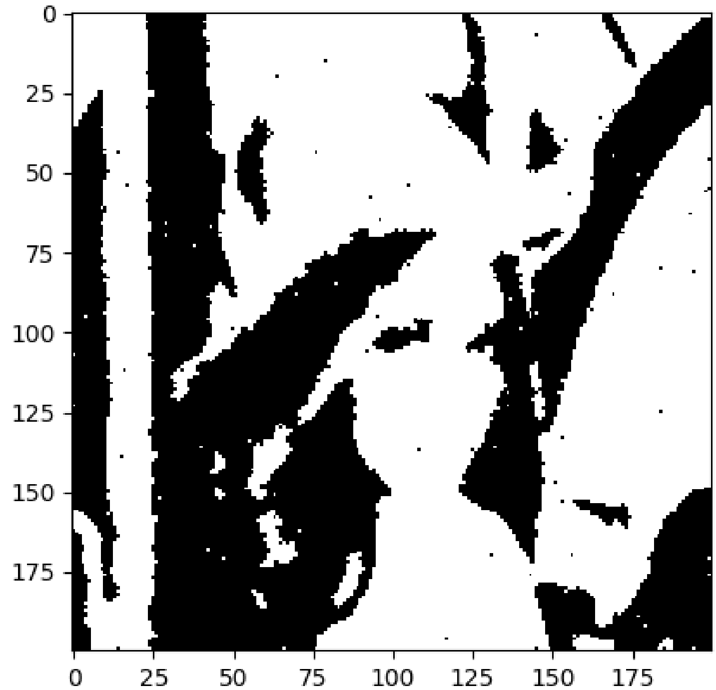

Figure 6: Denoised `Lena` with $\beta = 0.4$ and$\pi = 0.2$

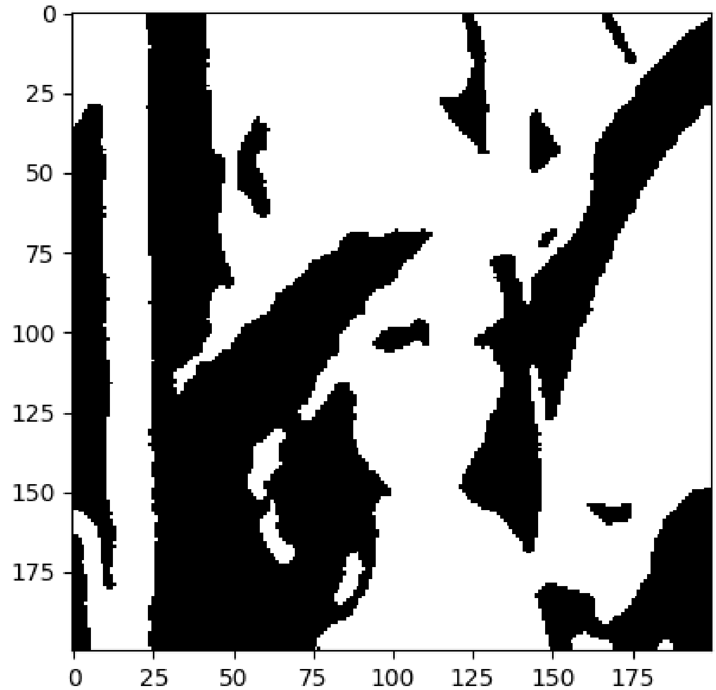

Figure 7: Denoised `Lena` with $\beta = 0.8$ and$\pi = 0.2$