IN-4200
High-Performance Computing and Numerical Projects

Project 2 - Image Denoising

Fred Marcus John Silverberg

10 May 2019

Contents

1 Introduction **2**

2 Method **2**
- 2.1 Import and Export of Data 2
- 2.2 Allocate and De-allocate Arrays 2
- 2.3 Conversion between JPEG and Image 3
- 2.4 Message Passing Interface [Open MPI] 4
- 2.5 Image Denoising Algorithm 6
- 2.6 Compilers and Hardware 7

3 Results **7**

4 Discussion **9**

1 Introduction

As a photographer the perfect moment can be destroyed by the phenomena of white noise, which presents itself as graininess in the final picture. The aim of this project is to implement a smoothing operation suited for a JPEG file in grayscale format, that will reduce the white noise. Achieving this in an efficient manner will require theoretical understanding and implementation of memory allocation and the message passing interface [MPI]. The operator will consist of the proven isotropic diffusion algorithm. This operation will be implemented as a serial approach as well as in a parallel approach. Further, the structure will be extended into handling a JPEG file in rgb format.

2 Method

2.1 Import and Export of Data

By the use of an internal c-library function 'simple-jpeg', we can import and export JPEG data. Given an image in JPEG format the internal function 'void_import_JPEG_file' will return the following structure:

For grayscale format (components=1):
1D-array = [g(0,0),g(0,1),g(0,2),g(0,3),g(0,4),g(0,5)...]
For rgb format (components=3):
1D-array = [r(0,0),g(0,0),b(0,0),r(0,1),g(0,1),b(0,1)...]
(x,y) = pixel position , (0,0) representing the top-left corner of the image

Given the same 1D array structure that represents an image, will by the internal function 'void_export_JPEG_file' return it as a JPEG file in the given directory.

2.2 Allocate and De-allocate Arrays

By defining memory before we operate on our data, we assure that there exists sufficient memory on our working platform. We also assure that there exists a specific address corresponding to our elements. The 'malloc()' function provide us with a dynamically allocated block of memory of wanted size and returns a pointer to the first byte of the block. This can be used for both grayscale (components=1) and rgb (components=3) as follows:

Allocating a 2D array, used for grayscale:

```
array_name = (float**)malloc(y_size*sizeof(float*))
for (i=0;i<y_size;i++)
  array_name[i] = (float*)malloc(x_size*sizeof(float))
```

Allocating a 3D array, used for rgb:

```
array_name = (float***)malloc(components_size*sizeof(float**))
for (i=0;i<components_size;i++)
   array_name[i] = (float**)malloc(y_size*sizeof(float*))
   for (j=0;j<y_size;j++)
      array_name[i][j] = (float*)malloc(x_size*sizeof(float))
```

The allocated blocks can after their contribution be delivered back to the working station as free memory space by using the function 'free()'. By demand one 'free()' for each 'malloc()' we avoid memory leaks. Implemented as follows:[1]

De-allocating a 2D array, used for grayscale:

```
for (i=0;i<y_size;i++)
   free(array_name[i])
free(array_name)
```

De-allocating a 3D array, used for rgb:

```
for (i=0;i<components_size;i++)
   for (j=0;j<y_size;j++)
      free(array_name[i][j])
   free(array_name[i])
free(array_name)
```

2.3 Conversion between JPEG and Image

The imported 1D array of grayscale or rgb values are structured into representing allocated memory block by the following procedure:

Conversion between 1D JPEG grayscale data and 2D Image data.

```
1. index = 0
2. for (i=0;i<y_size;i++)
3.    for (j=0;j<y_size;j++)
4.       index = x_size*i + j    // Keep track of rows
5.       array_name[i][j] = imported_data_array[index]
```

Conversion between 1D JPEG rgb data and 3D Image data.

```
1. index_x = 0
2. index_y = 0
3. for (i=0;i<components_size;i++)
4.    array_name[0][index_y][index_x] = imported_data_array[i]
5.    array_name[1][index_y][index_x] = imported_data_array[i+1]
6.    array_name[2][index_y][index_x] = imported_data_array[i+2]
7.    ix += 1
8.    if (index_x == x_size)    // Keep track of rows
9.       index_x = 0
10.   index_y +=1
```

In the examples above, the conversion is from the imported data [1D $\rightarrow$ 2D/3D]. In order to convert into import format [2D/3D $\rightarrow$ 1D] one have to swap the objects in row (4,5,6) for the rgb format and only row 5 for the grayscale format. Note that for the rgb case, there exists three 2D images holding 'r','g' or 'b' values, while for the grayscale case, there exists only one 2D image, holding the grayscale values.

2.4 Message Passing Interface [Open MPI]

MPI is a method of writing applications that is based on message passing. It consists of a communicator which holds a closed group of processes that can communicate with each other. Each process has its own unique ID, this allows processes to perform point to point communication, where they can send and receive data between each other. As a user, one can then partitioning the task and distribute the sub-tasks to the processes. [3]

There exists both non-blocking and blocking 'send and receive' methods, this project has implemented the later. This imply that by calling MPI_Send the process do not return until the complete message has been sent and the buffer can be reused. Likewise, MPI_Recv do not return before it has stored the incoming data in its buffer. This may create a situation named 'deadlock', a point where a process is waiting for data that will never arrive.

Example of possible deadlock:

```
1. if (my_rank == 0)
2.     MPI_Send(x,count,MPI_FLOAT,1,tag,MPI_COMM_WORLD)
3.     MPI_Recv(x,count,MPI_FLOAT,1,tag,MPI_COMM_WORLD,&status)
4. else if (my_rank == 1)
5.     MPI_Send(x,count,MPI_FLOAT,0,tag,MPI_COMM_WORLD)
6.     MPI_Recv(x,count,MPI_FLOAT,0,tag,MPI_COMM_WORLD,&status)
```

Above, if MPI_Send does not complete its task before the corresponding MPI_Recv is called, then a deadlock will occur. To avoid this row 5 and 6 are swopped. The resulting implementation is used in this project.

Even if parallel programming is thought of as a speed-up or efficiency gain application, it always inject overhead in the program. It is due to internal communication, uneven workload and blocking operations. In general a parallel implementation is to favor in situations when the calculations are complicated or the amount of computation is large.[4]

Measurements of MPI:

Speed-up measurement:

$$S(P) = \frac{T(1)}{T(P)}$$

Efficency measurement:

$$n(P) = \frac{S(P)}{P}$$

P: Nr or processes, T(1): Execution time of serial computation, T(P): Execution time of parallel computation.

MPI Commands of usage:

MPI_Init()	This command initialize the MPI environment by taking pointers to the input arguments of the main function. A world communicator is then build called 'MPI_COMM_WORLD', which defines the group of MPI processes.
MPI_Comm_rank()	This assign an ID number starting from zero and increases in consecutive order, for the processes that enters the line.
MPI_Comm_size()	This returns the number of processes that are included in the group.
MPI_Bcast()	Communicate the same message from root process to all other process, essentially making sure the other processes see the same buffer. A buffer could for example be a constant or an array of a specific datatype, which also need to be determined along with how many elements (count) the buffer contains.
MPI_Send()	A command that sends data (buffer) with a certain elements (count) of a type (datatype) to a specific process (destination) in the group.
MPI_Recv()	A command that receives data (buffer) with wanted number of elements (count) of a type (datatype) from a specific process (source) in the group. Note that it can receive less elements than provided, but never more.
MPI_Barrier()	Waits for all processes to complete their work and arrive at this line.
MPI_Reduce()	Reads send_buffer on all processes and operate by choice of 'op' (MPI_MAX, MPI_AVG etc) then store the result in recv_buffer. Used for timing in this project.
MPI_Finalize()	A necessary call for exiting a MPI environment. This terminates the group. All processes must call this routine before exiting.

For full explanation of input arguments:[2] https://www.mpich.org/static/docs/latest

2.5 Image Denoising Algorithm

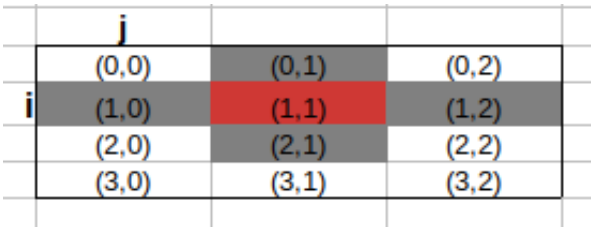


Figure 1: Illustrated image of algorithm

Isotropic diffusion algorithm:

$$\bar{u}_{i,j} = u_{i,j} + kappa(u_{i-1,j} + u_{i,j-1} - 4 * u_{i,j} + u_{i,j+1} + u_{i+1,j})$$

$\bar{u}$ represents the future pixel value in red, while u represents the current pixel value in red. (i.j) is again the positions of the pixels. The constant 'kappa' is a physical constant. As the algorithm zoom in on one pixel, it uses the neighbor pixel values in grey. This imply that for a 2D image the boundaries for $\bar{u}$ will be $(0 < j < x\_size - 1)$ and $(0 < i < y\_size - 1)$
The core implementation used for both serial and parallel approach:

```
1. for i (i=0;i<x_size;i++)
2.    do 'top and bottom boundaries'
3. for i (i=0;i<y_size;i++)
4.    do 'left and right boundaries'
5. for (i=1;i<y_size-1;i++)
6.    do 'isotropic diffusion algorithm'
7. for (i=1;i<y_size;i++)
8.    for (j=1;j<x_size;j++)
9.       do swap between u_bar and u.
```

Special implementation for serial approach:

1. Enclose the core directly with a for loop over 'iterations'.

Special implementation for parallel grayscale approach:

1. The for loop over 'iterations' is placed in the main code.
2. For each iteration a process receives data from master.
3. After receiving data the process call the core.

Special implementation for parallel rgb approach:

1. The for loop over 'iterations' is placed in the main code.
2. A for loop over 'components (r,g,b)' takes place for each iteration.
3. For each component (r,g,b) a process receives data from master.
4. After receiving data the process call the core.

2.6 Compilers and Hardware

```
* Compiler information:
mpicc for MPICH version 3.3a2
gcc - 4:7.3.0-3ubuntu2.1 - amd64 - GNU C compiler

* Hardware information:
Intel i5-6500 Skylake: 3.2 Ghz
Operation capacity:     16 DP operations/s
Max memory bandwidth: 34.1 GB/s
CPUs:                    4
Threads:                 4
Cache:                   6 MB SmartCache
```

3 Results

Figure 2: Smoothed grayscale result with kappa = 0.2 and iterations = 100

(a) Smoothed

(b) Noisy

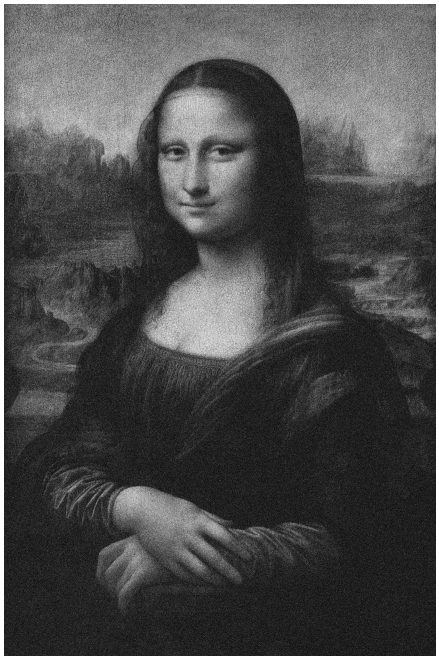


Figure 3: Smoothed rgb result with kappa = 0.03 and iterations = 20

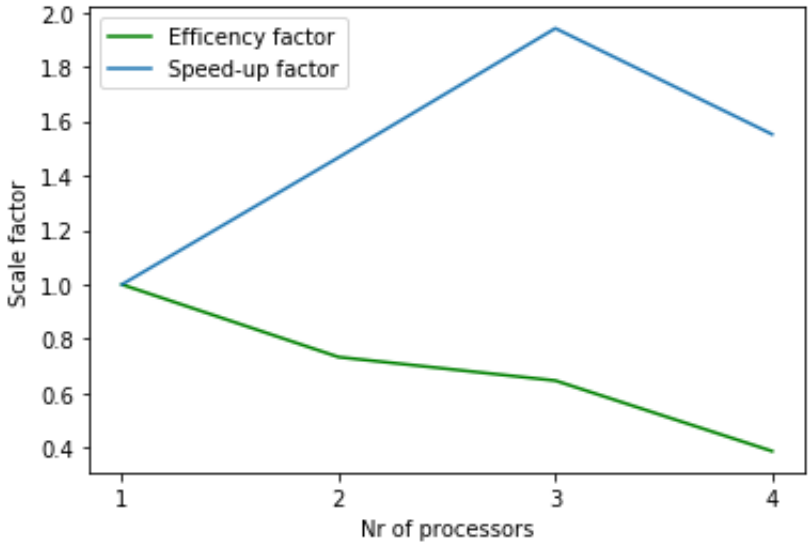
(a) Smoothed

(b) Noisy

Table 1: Measurements for grayscale code [Kappa = 0.2 , Iterations = 500]

Processes (n)	Avg_t1	Avg_t2	Avg_t3	Avg_t4	Avg_t5	Mean	Speed-up	Efficency
Serial	10.54s	10.99s	10.75s	10.82s	11.06s	10.83s	1.00	1.00
2	7.36s	7.40s	7.36s	7.43s	7.36s	7.38s	1.46	0.73
3	5.57s	5.60s	5.58s	5.58s	5.57s	5.58s	1.94	0.64
4	6.56s	6.95s	6.53s	8.20s	6.68s	6.98s	1.55	0.38

Figure 4: Graph for grayscale code [Kappa = 0.2 , Iterations = 500]



4 Discussion

A quick look at the result in figure (2) imply that the smoothing process of grayscale data succeeded. For a configuration of one master and two processes (n=3), the parallel implementation preformed the task in close to half the computational time as the serial (one process) implementation. The difference from exactly half could very well be explained by injected overhead. Figure (4) illustrates the values from table (1), there is a clear sign of a linear speed-up factor as a function of processes. Regarding efficiency, the trend indicate that it may start to flatten at (n=3). Since the number of processes are low, conclusions on such a few processes should only be considered indicative for further scaling. The fall for (n=4) is simply explained by the workstations limitations of four threads.

The partitioning of grayscale data (size_x, size_y) into workloads for each process (my_x,my_y) is also limited by the number of threads available. For a workstation with four threads, there is only three processes available for preforming the work. Hence, partitioning more than one dimension only adds overhead. However, with a larger amount of threads available a two dimensional partitioning would improve the performance. In such a case, a change could be made by implementing the same partitioning analogy on both dimensions, see function 'partitioning' in functions.c. As well as changing the boundaries in loops on lines [92,99,111,119] in parallel_main.c and on lines [80,87,101] in functions.c.

The rgb data was handled by separating the three components into three 2D batches, they were further partitioned with the same analogy as with the grayscale data. Even if this violate optimal boundary conditions it delivered decent result as can be seen in figure (3). Further improvement for the rgb implementation would be to add periodic boundary conditions.

References

[1] Manish Virmani. 2006. Pointers and memory leaks in C. [ONLINE] Available at: https://developer.ibm.com/articles/au-toughgame/ [Accessed 7 May 2019].

[2] MPICH. 2018. MPI Routines and Constants. [ONLINE] Available at: https://www.mpich.org/static/docs/latest/ [Accessed 7 May 2019].

[3] Hager, G. and Wellein, G. (2012). Introduction to high performance computing for scientists and engineers. Boca Raton, FL: CRC Press, p.205.

[4] Hager, G. and Wellein, G. (2012). Introduction to high performance computing for scientists and engineers. Boca Raton, FL: CRC Press, p.211-213.