Image Denoising with MPI

Candidate number 15202
(Dated: May 10, 2019)

Research regarding image denoising is presented. A denoising algorithm based on isotropic diffusion were implemented in C and parallelized by using MPI. The aim of the code was to achieve good performance and faster execution by parallelization. By denoising an image (4289×2835) with 100 iterations, the most efficient code ran on 4 processes, which executed 2.3 times faster than by using only one process.

I. INTRODUCTION

A very simple denoising algorithm for grayscale images (jpg) were written in C. The algorithm was based on the isotropic diffusion equation in 2D. There were two implementations: a serial code and a distributed memory parallelized code using MPI. The latter included communication between the different processes during each iteration.

II. THEORY

The isotropic diffusion equation is a linear differential equation. In two dimensions it reads

$$\frac{\partial u(x,y,t)}{\partial t} = D\nabla^2 u(x,y,t), \tag{1}$$

where D is constant in the case of the *isotropic* diffusion equation. Equation 1 can be discretized and solved numerically fairly straight-forward, by using the three point formula for the second derivatives:

$$\begin{aligned}
\frac{\partial u_{i,j}^t}{\partial t} &\approx \frac{u_{i,j}^{t+1} - u_{i,j}^t}{h_t} + \mathcal{O}(h_t), \\
\frac{\partial^2 u_{i,j}}{\partial x^2} &\approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(h_x)^2} + \mathcal{O}(h_x^2), \\
\frac{\partial^2 u_{i,j}}{\partial y^2} &\approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{(h_y)^2} + \mathcal{O}(h_y^2).
\end{aligned}$$

$\mathcal{O}$ denotes the order of the truncation error by this formula. Hence the discretize algorithm is

$$\bar{u}_{i,j} = u_{i,j} + \kappa(u_{i+1,j} + u_{i-1,j} - 4u_{i,j} + u_{i,j+1} + u_{i,j-1}). \tag{2}$$

$\kappa \leq 0.2$ is a constant, and $\bar{u}$ is the image which is one iteration ahead of u.

III. HARDWARE

Operative system: *Linux Mint 19.1.* The following terminal print shows the *lscpu* command (CPU specs):

```
$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 142
Model name:            Intel(R) Core(TM) i7-8550U
    CPU @ 1.80GHz
Stepping:              10
CPU MHz:               800.021
CPU max MHz:           4000.0000
CPU min MHz:           400.0000
BogoMIPS:              3984.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0-7
```

IV. ALGORITHM

A. Serial code

The image were imported to the C program by using an external library, and it was saved as an underlying 1D array of type unsigned char (integers 0-255). The import also specified the size of the image ($m \times n$) and the number of colors (c, which always was 1 in this project). The image were converted to a struct which contained the size of the image, as well as the image data, but now stored as a 2D array of type float: which is necessary for more accurate computation in the algorithm, and also more easily indexed.

Because the algorithm uses the four neighboring points of a pixel, only the inner pixels of an image is denoised and the pixels along the outer edge are not changed. At the end of each iteration, u is set to $\bar{u}$ (and not the other way around!).

Lastly, the image is converted back to a 1D array of type unsigned char and exported as a jpg-file.

B. Parallel code

The parallelized code runs in the same fashion as the serial code but with some additional communication. The flow of data can be summarized in these steps:

- Import: rank 0 imports image and broadcasts the size of the image: height m and width n.
- Domain decomposition: an int array dims is created where the second index is fixed to 1. This means that the image will be split in horizontal strips, and the

domain decomposition is one dimensional. Each process will find its own size: starting and stopping index in m-direction is found. Note: if the image can't be split evenly, all processes except the last will have the slightly larger image chunk and the last rank will have a slightly smaller one. The alternative, would have been to have the last rank have a few extra rows, but this would have resulted in more idle worker time.

- Collection of sizes and distribution of image: the image chunk sizes are gathered using MPI_Gather and rank 0 distributes chunks of the image by MPI_Scatterv.
- Denoising algorithm: the function iso_diffusion_denoising_parallel() is called by all ranks. Essentially each rank have two arrays which will work as its ghost layer on the top and bottom of its own chunk. Before a sweep is done, the necessary communication takes place:
  - Even ranks send their bottom layer down, odd ranks receive these.
  - Odd ranks send their bottom layer down, even ranks receive these.
  - Even ranks send their top layer up, odd ranks receive these.
  - Odd ranks send their top layer up, even ranks receive these.
- Note: rank 0 will not send up and the last rank will not send down due to not using periodic boundary conditions. This communication form can be optimal when using blocking communication (MPI_Send and MPI_Recv), by splitting tasks so that every other rank does something different at a time. The reason behind this form was that sent messages would not have to wait long until they reached the confirmation that they've been received (in fact the Recv call was always done before the send so that it was ready).
- The sweep is done in each rank's internal image.
- Lastly, in the main function, the images converted to unsigned char arrays, gathered to rank 0 (using MPI_Gatherv) and exported as a jpg-file.

Timing measurements were done by using clock_gettime(CLOCK_MONOTONIC) over the function iso_diffusion_denoising_parallel(). The number of processes were varied from 1-16. The result was gathered as the mean of 5 runs, with an uncertainty taken as the standard deviation.

The codes were attempted to be optimized with compiler flags (such as -O1 -O2 and so on). By trial and error, -O2 turned out to provide the fastest executable.

V. RESULTS

Figure 1 shows the runtime of the function iso_diffusion_denoising_parallel() in milliseconds vs. the number of processes used.

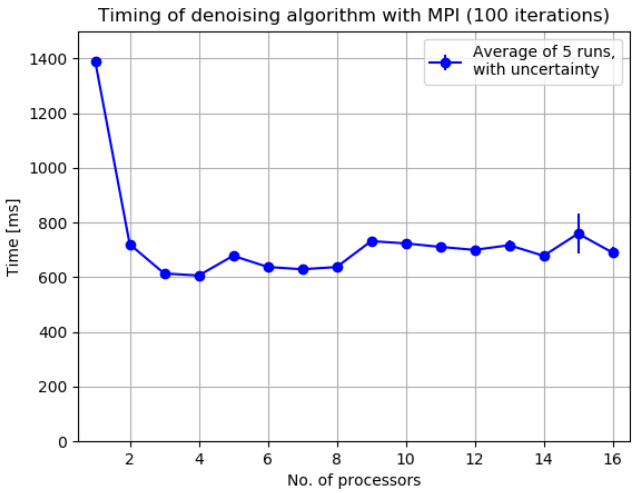

FIG. 1. Parallelized denoising algorithm on a 4289×2835 image, with $\kappa = 0.15$, 100 iterations. CPU: Intel i7-8550U with 8 cores and two threads per core.

The run on 4 processes were 2.3 times faster than the run on a single process.

VI. DISCUSSION

As seen in figure 1 the performance quickly increases when using more than 1 process. The fastest execution time is achieved with 4 processes.

A slight trend that can be seen is a performance drop from 4-5 processes and from 8-9 processes. The latter is easily explained by the hardware, which contains 8 cores and when running on 9 processes, two of those have to share a core. The drop at using 5 is less subtle. When monitoring active cores whilst running the program, 4 cores are active when running on 4 processes and 5 when running on 5 as expected. Therefore the performance drop can not be explained by cores beeing shared. A possible explanation is that overhead communication costs more than what the extra core gains. This is also a possible explanation for why 4 processes achieved the best result: for this image size the speedup gained by using 4 processes minus the time lost to overhead was optimal.

Figure 1 also includes uncertainty based on 5 runs, which were surprisingly small for all runs, except for when running with 15 processes: this specific no. of processes had a rather large uncertainty. The reason for this is unknown.

VII. CONCLUSION

Image denoising is a perfect example of an algorithm which can be optimized by using parallelization. By us-

ing MPI, the denoising of an image of 12.2 Mpixels were sped up 2.3 times, by using 4 processes, rather than only one.