Parallel convolution for image blurring

Dario Coscia

September 2022

Contents

1 Introduction

Convolution is a mathematical operation applied in many area of science including, but not limiting, to deep learning, numerical analysis, physics, signal processing or computer vision [1, 9, 6, 3, 7, 5]. In particular, the convolution operator is widely used as blurring technique, i.e *smoothing* of a given quantity over a neighbourhood by means of a convolutional *kernel*.

In the present manuscript we present and analyze a parallel implementation of the convolutional operator which uses two standard frameworks for parallel programming, namely OpenMP [2] (shared memory) and MPI [4] (distributed memory). Specifically, we compare by a scalability study the two parallel algorithms in terms of strong and weak scaling. Furthermore, compiler optimization effects, as well as the consequences of kernel precision initialization (double against float) are analysed. Finally, we also provide a simple way to create an hybrid parallelization solution exploiting both OpenMP and MPI.

The present contribution is organised as follows: in Section 1.1, a small review of the convolutional operator is done, as well as introducing the different kernels. Section 2 is focused on the implementation of possible parallel algorithms, using MPI 2.1, OpenMP 2.2 and an hybrid approach 2.3. Last, in Section 3 we present scalability results in terms of strong and weak scaling and, additionally, some measurements regarding compiler flags and kernel precision type. Finally, conclusions follow in Section 4.

The source code is reported in the following GitHub page.

(a) Input image. (b) Blurred image.

Figure 1: Example of mean kernel with $D = 31$ application on a gray-scale image.

1.1 Convolutional operator

Convolutional is a mathematical technique which performs the smoothing of the input data $\mathcal{I}$ and the so called convolutive filter $\mathcal{K}$, such that

$$(\mathcal{I} * \mathcal{K})(x) = \int_{-\infty}^{\infty} \mathcal{I}(x+\tau)\mathcal{K}(\tau)d\tau. \tag{1.1.1}$$

Practically, in working with gray-scale (one-channel) images, i.e. two-dimensional matrices, we define convolution for an input matrix I of dimension $N \times M$, and a square kernel matrix K with dimension D as:

$$(I * K)(i,j) = \sum_{m=1}^{D}\sum_{n=1}^{D} I(i+m, j+n) \cdot K(m,n), \tag{1.1.2}$$

where $i \in \{1, ..., N\}$, $j \in \{1, ..., M\}$. It is important to notice that only square kernel with odd order D are considered[1]. The choice of the kernel is problem specific, and multiple kernels are available in literature e.g. Gaussian, Mean, Weight or Sharpen. The kernel matrix we will consider in the report are spherically symmetric with respect to the central entry $(\lfloor D/2 \rfloor, \lfloor D/2 \rfloor)$, and unit normalised, i.e. the sum of the kernel entries sum up to one. The available kernel choices in the software provided with the report are Gaussian, Mean and Weight kernel.

In Figure 3 an example of blurring using a Mean square kernel is reported. Notice that the blurred image has a vignetting effect due to the fact that while processing border pixel the remaining portion of the kernel is not re-normalized.

2 Parallel Algorithm

The convolutional operator, as reported in Equation 1.1, can be easily serially implemented by a four nested `for` loop, where the two outer loops scan the input matrix indices, while the inner two scan the kernel matrix and accumulate the sum. The accumulator can be either defined with `float` or `double` precision. Multiple works are done in literature to improve the reported näive implementation, see [8] for more information. Nevertheless, the improvement reported are kernel dimension dependent (loop enrollment for example), making it hard to generalize. Therefore, for the serial implementation, we adopted the näive convolution implementation. In the following sections three possible parallel implementations of the näive algorithm are reported using MPI, OpenMP and Hybrid. Practically, the code is developed using the `C++17` standard, where the matrices are saved as `std::vector`. The vector is initialised by means of the `reserve()` routine, since the (commonly used) `push_back()` operation is extremely costly. For one byte image `unsigned char` templates `std::vector`, while `unsigned short int` are used for templating if two bytes images are passed as input.

[1]The implementation of the convolution routines only, both parallel and serial, are templated and general (no need for square kernels), enhancing portability of the code.

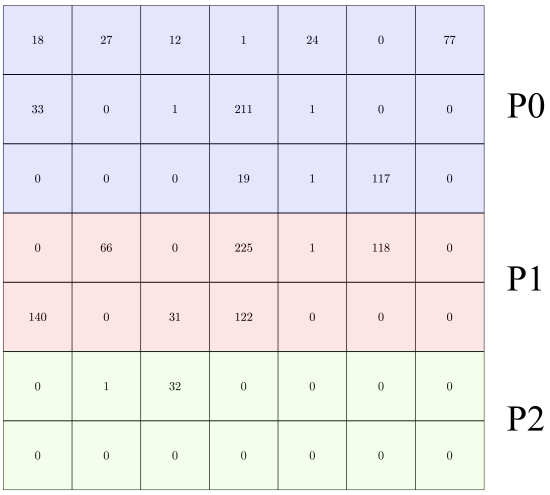


Figure 2: Stripe division of the input image, each color represents a different processor domain allocation.

2.1 MPI Algorithm

The strategy employed for the parallelization with MPI consists in dividing in stripes the input matrix and assigning each stripe to an available processor P, as shown in Figure 2. The use of horizontal stripes, instead of dividing the domain in blocks, is due to the fact that a block domain decomposition would require a matrix shuffling in order to ensure contiguous memory access. Once the matrix is divided into stripes, in order to ensure that the convolution is well executed at the domain borders, halo layers need to be exchanged between processors. The halo exchanged is done by means of `MPI_Sendrecv()` routine, while the stripe division and allocation is done by `MPI_Scatterv()`. The last mentioned routine is a collective operation that works on chunks of continuous data, thus avoiding the use of hard-coded send and receive routines through `MPI_Send()`, `MPI_Recv()`. In case the number of rows in the matrix is not perfectly divisible by the number of processors (unbalanced workload), a row is added for each processor till all the extra rows are divided. This procedure ensure that all processors have roughly the same amount of work. Finally, the näive convolution is done for the smaller scattered matrices in each processor, then the single processor results are gathered in the output matrix by `MPI_Gatherv()`.

2.2 OpenMP Algorithm

The OpenMP solution is more simplistic to the MPI solution. More specifically, the näive algorithm is parallellized by means of `#pragma omp for` for the outer-most loop. The choice of the loop order was done after performing scalability tests for different configurations. Furthermore, as suggested in [10], only the outer most loop is parallelized, and marked as `nowait` avoiding implicit barriers. It is important to notice that the provided solution can possibly increase the possibility of false sharing. In fact, when writing the output result of the convolution in the position (i, j), i.e. $(I * K)(i, j)$, it is possible that multiple threads access and modify simultaneously the same cache line(s), forcing write-back and reflush. Nevertheless, due to its simplicity of the implementation, we decided to implemented in this way and compare it to the MPI implementation. The results of scalability, as reported in Section 3.3, report a comparable speed up to the MPI implementation, thus we decided to remain with the presented OpenMP implementation.

2.3 Hybrid Algorithm

A possible hybrid implementation, which can be found in the source code provided with the present manuscript, is done by combining the MPI and OpenMP parallelization strategies. First, the input matrix is divided in stripes, and those stripes are scattered to different processors. Secondly, each processor performs convolution on the smaller scattered matrix, and the convolution is parallelized by the OpenMP strategy. Finally, once all processors are finished with the computations (implicit

barrier), the results are gathered, and the procedure follows the flow explained in the MPI sections. This procedure is very simple and safe, since OpenMP regions are opened in a MPI process for computation only.

3 Performance model and results

In this section we report a scalability study of our implementation, two simple performance models for the MPI and OpenMP parallelization strategies and additional measurements. In particular, we were interested in understanding how different compilation flags could affect the overall execution time, as well as studying the behaviour of the scalability for different kernel represented as `float` or `double`.

3.1 Serial and parallel model

Let N and M be respectively the number of rows and columns of the input matrix, while D is the odd-dimension of the kernel matrix. Considering Equation 1.1, for each element of the input image and for each position of the kernel, two operations (one addition and one multiplication) are done. Hence the total number of operation OPS will be:

$$OPS \propto 2 \cdot N \cdot M \cdot D^2 \tag{3.1.1}$$

Therefore, the total serial execution time T_{serial} is:

$$T_{\text{serial}} \propto T_{\text{I/O}} + \frac{OPS}{\nu}, \tag{3.1.2}$$

where ν is the total number of floating point operation per second.
A simple parallel model for the OpenMP parallelization strategy is done by considering P threads that execute the workload simultaneously, hence:

$$T_{\text{OpenMP}} \propto T_{\text{I/O}} + \frac{OPS}{\nu} \cdot \frac{1}{P}. \tag{3.1.3}$$

Finally, in case of MPI the parallel model for P processors will be:

$$T_{\text{MPI}} \propto T_{\text{I/O}} + T_{\text{comm}}(M, P) + \frac{OPS}{\nu} \cdot \frac{1}{P}, \tag{3.1.4}$$

where $T_{\text{comm}}(M, P)$ is the communication cost that accounts for `MPI_Scatterv()`, `MPI_Gatherv()` and Halo exchange done by the `MPI_Sendrecv()` routine. Assuming that the scattering and gathering takes approximately the same time, and that the matrix is scattered (gathered) by means of binary tree divisions, the total scatter/gather time is:

$$T_{\text{scatter}} = T_{\text{gather}} = \lambda \log_2(P) + N \cdot \frac{2P - 1}{P}, \tag{3.1.5}$$

where λ is the latency of the infiniband network. Moreover, let b_{network} the bandwidth of the network, the total halo exchange time is:

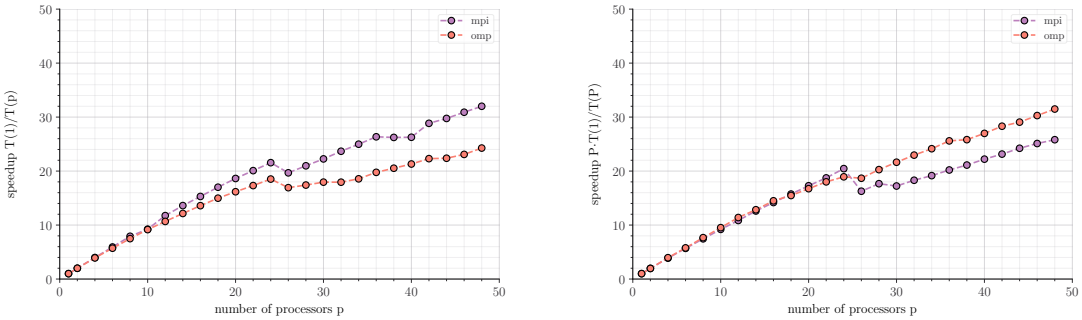
$$T_{\text{halo}} = 2\lambda + 2\frac{M \cdot \lfloor \frac{D}{2} \rfloor}{b_{\text{network}}}, \tag{3.1.6}$$

where $M \cdot \lfloor \frac{D}{2} \rfloor$ is the total halo size. Therefore, the final MPI performance model is:

$$T_{\text{MPI}} \propto T_{\text{I/O}} + T_{\text{scatter}} + T_{\text{halo}} + \frac{OPS}{\nu} \cdot \frac{1}{P}, \tag{3.1.7}$$

3.2 Software and hardware employed

The measurements we report have been carried out using a variable number of processors on a gpu node on Orfeo, 2 sockets of 12 physical cores each (24 logical cores due to the fact the processors have hyperthreading enabled). The code have been compiled using gcc 9.3.0, the MPI distribution taken into account is OpenMPI 4.1.1.



(a) Strong scaling for parallel models.

(b) Weak scaling for parallel models.

Figure 3: Different scaling models.

3.3 Strong and weak scalability

In the first place we report the strong scalability of our parallel implementation, i.e. the performance gain as the number of processors increase while maintaining constant the problem size. The image size used for testing strong scalability is $N = 2520$, $M = 4032$, which is represented using `unsigned short int`. The kernel size is kept fix at 51, and the kernel type used is the Mean kernel.
In Figure 3a the results of the experiment are depicted. In particular, we can notice from 0 to 24 processors a similar speed up between the MPI implementation and the OpenMP one. The jump presented at 25 processors, which decreases the speedup rate, could be due to hyperthreding. In fact, the node used for the tests contains 24 processors with hyperthreding enabled, thus 48 logical cores performing the workload. However, since all the threads have a similar workload, and since the instructions given are not independent (all threads performing same operation in writing to the output matrix), the latency between instructions is hardly optimized, since it is not possible scheduling overlapping instructions perfectly.
The second measure reported is a weak scalability study, hence the number of processors and the image size both increase. In order to perform the study, we have generated multiple gray-scale gradient images with increasing dimensions of rows and columns. In particular, starting from a 1200×800 image ($P = 1$), we tried to keep the $3 : 2$ ratio when scaling it up to the number of processors. For example, for $P = 4$ we would have a 2400×1600 image, while $P = 22$ a 5628×3752 image. The results of the weak scalability experiments are reported in Figure 3b. In particular, the OpenMP implementation seems to better weakly scale than the MPI one, after the 24 processors. Again, we believe that hyperthreding is responsible for such behaviour.

3.4 Additional measurements

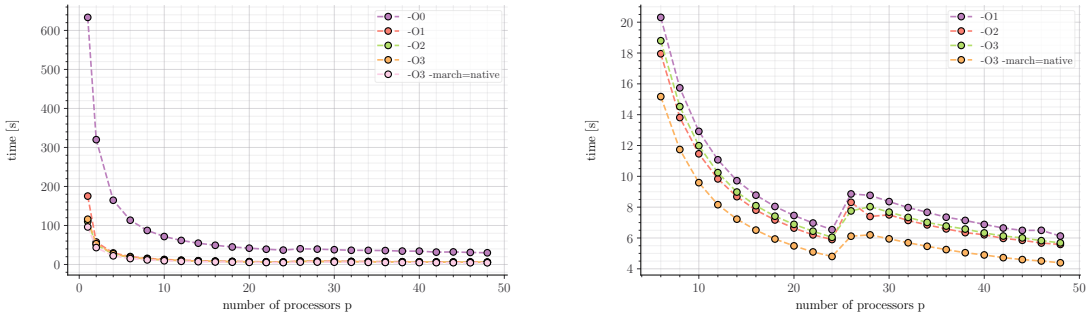
Some additional measurements regarding compiler optimization flags and kernel precision are reported. We decided to perform such measurements in order to understand first if there is a gain, and how much, in using a different compilation flag; and if there is a gain in using `float` instead of `double` for storing the values of the kernel matrix.

3.4.1 Compiler optimization

It is interesting to comment on the performance of the code for different compilation flags. As depicted in Figure 4, there is a clear difference in using the -O0 flag with respect to the others, which seem to perform all slightly similar (Figure 4a). Nevertheless, from Figure 4b, it is notable that while -O1, -O2 and -O3 tend to perform approximately the same, the -O3 -march=native flag is outperforming with respect to the others.

3.4.2 Kernel precision

Another interesting experiment is to see if the code perform worse using `double` instead of `float` for templating the `std::vector` of the kernel matrix. The results are depicted in Figure 5, where it is possible to see that using `double` increase the overall execution time. This could be due to



(a) All possible flags.

(b) Compilation flag excluding -O0.

Figure 4: Execution time for different compilation flags in MPI.

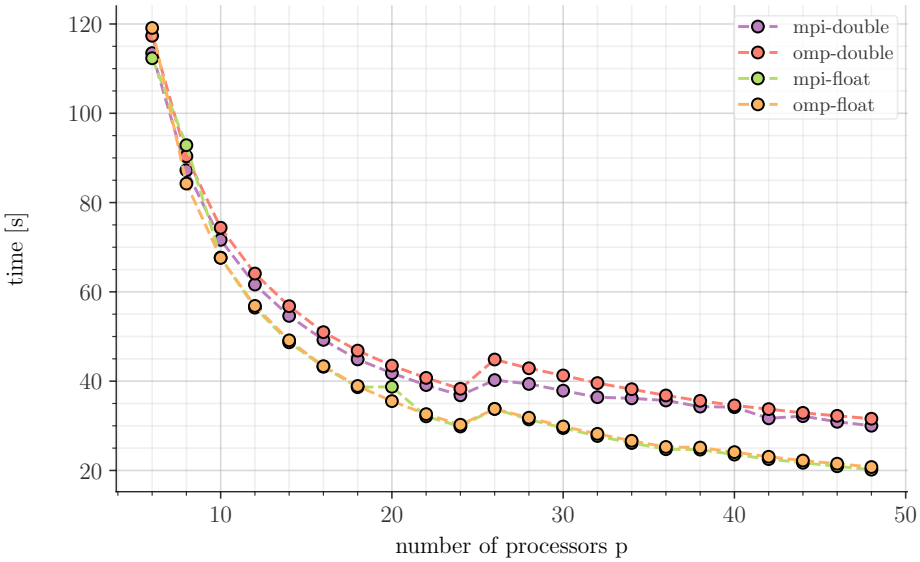


Figure 5: Execution time using different precision for kernel matrix.

the vector register capacity (holding one double or holding two floats), leading to an increase level of SIMD instructions if `float` are used instead of `double`. Nevertheless, `double` accounts for a greater precision, hence in our implementation we enable the user to choose which data type should be used at compile time, by means of a specific flag.

4 Conclusions

In the following report we present a way to parallelize the convolutional operator by means of OpenMP and MPI. The source code is available on GitHub, with a `README` file explaining how to compile the code using the provided makefile for the specific needs. Moreover, a performance model of the code, an analysis of strong and weak scalability and other additional measurements, namely compilation flags and kernel precision, are reported. We also presented an hybrid solution using both MPI and OpenMP, which is available on the GitHub but for which no performance study is done, due to an increase complexity in the analysis (too long for a short report). As suggested in Section 2.1, for the MPI implementation, we believe that a 2-dimensional decomposition of the input is better suited for the problem, however extra technicalities (e.g. shuffling matrices) need to be taken into account. The OpenMP implementation could also be improved by a different, more efficient way of parallelize the code, maybe trying a similar strategy as the MPI one.

References

[1] L. L. Ankile, M. F. Heggland, and K. Krange. Deep convolutional neural networks: A survey of the foundations, selected improvements, and some current applications, 2020.

[2] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[3] J. Dölz, H. Egger, and V. Shashkov. A convolution quadrature method for maxwell's equations in dispersive media, 2020.

[4] M. P. Forum. Mpi: A message-passing interface standard. Technical report, USA, 1994.

[5] C. B. Hyndman and P. O. Ngou. A convolution method for numerical solution of backward stochastic differential equations. *Methodology and Computing in Applied Probability*, 19(1):1–29, jun 2015.

[6] K. O'Shea and R. Nash. An introduction to convolutional neural networks, 2015.

[7] C. Shimmin. Particle convolution for high energy physics, 2021.

[8] A. Tousimojarad, W. Vanderbauwhede, and W. P. Cockshott. 2d image convolution using three parallel programming models on the xeon phi, 2017.

[9] X. Zhao, Z. Gong, Y. Zhang, W. Yao, and X. Chen. Physics-informed convolutional neural networks for temperature field prediction of heat source layout without labeled data, 2021.

[10] Z. Zheng, X. Chen, Z. Wang, L. Shen, and J. Li. Performance model for openmp parallelized loops. In *Proceedings 2011 International Conference on Transportation, Mechanical, and Electrical Engineering (TMEE)*, pages 383–387, 2011.