Second Assignement - FHPC course

Giulio Crognaletti

January 2021

1 Algorithm implementation

According to the assignment requests, I provided two version of a parallel algorithm used to do convolution of two matrices denoted here M and K. Hereafter the respective dimensions will be denoted as m_x,m_y,k_x and k_y.[1] The algorithms differ on the parallelization technique (one based upon OMP and the other on MPI) but originate from the same convolution algorithm. In particular, the algorithm implemented is a straight forward application of the definition of convolution, i.e. this sum by means of for loops:

$$C_{ij} = \sum_{u=0,v=0}^{u=k_x,v=k_y} M_{i-s_x+u,j-s_y+v} K_{u,v}$$

where $s_x = \lfloor k_x/2 \rfloor$ and $s_y = \lfloor k_y/2 \rfloor$.
The need for the algorithm to be independent from starting point is matched by not using an in-place algorithm but storing the result of each entry C_{ij} in a different result matrix and not M itself.
The normalization of the kernel K is performed just once at the generation, then the already normalized entries are stored in a floating point number matrix and used without the need of any successive step. The produced code can accept kernel of type: Uniform, Weighted, Gaussian and Custom, where Custom is a user defined matrix fed to the program, as a PGM matrix of integers to be normalized.
After the kernel has been correctly initialized the convolution proceeds, taking into account the border effect: this has been done by re-normalizing the kernel inside the convolution algorithm. Obviously since the body of M does not need these extra calculations, the border and body area have been split ad convoluted differently. All technical details are well explained with comments in the code itself and the README file.
Finally the code is thought to work well under the working assumption $k_x, k_y \ll m_x$ and $k_x, k_y \ll m_y$, but no major restrictions prevent the code to run and give correct results otherwise (it surely will be less efficient, since for stater most of the image if not all is now on the border, and so the re-normalization has to be performed in the majorty of steps). Some additional assumption on the number of processors are also required for the parallel versions, as stated in the paragraphs below.

1.1 OMP parallelization

In the parallelization with OMP a multi-threaded shared memory approach is used, so each thread can access and read the matrix M without the need of any communication.
In particular, after the I/O operations and the kernel generation/normalization (done by the main thread only), a parallel region is opened in the application and calls to the newly defined functions `OMP_swap()` and `OMP_Convolve()` are made. Inside these functions are present some orphaned OMP directives to parallelize the five main for loops that perform the convolution and swapping of M. This is done to use the same team of threads for each of the operations therefore reducing the overhead associated with the creation/initialization of the team itself.
Since the workload in the main loop is balanced the `static` schedule is the more appropriate. In the border loops small imbalances in the workload can be present (especially in the first and last parts) due to the uneven amount of operations to be done in each entry. For this reason the `nowait` clause is used, to try to make early finishing threads in the previous loop start the next first and hence acquiring the heavier parts. Anyway the number of operations is completely dominated by the body in the working assumptions so this should not

[1] Generalizing the text of the assignment, thy can be all different, but both k_x and k_y must be *odd* integers.

be a problem. Also the `collapse(2)` clause is used to reduce the matrix scanning to a single for loop to be parallelized. Under the working assumptions the two innermost for loops in both body and border parts are significantly less expensive than the outermost ones, so they are left as are to each thread.
Finally, the number of threads P here has no major constraint, anyway the best results should arises when $P \ll m_x m_y$, so that the workload is significant for each and the overhead is negligible.

1.2 MPI parallelization

In the parallezation with MPI, multiple processes (each with private memory) are spawned by the application, and so a communication algorithm must be implemented. After the image is read by the master process, a chunk of it must be sent to each worker, containing all the information needed to process its share of work. If we denote as P the number of processes used, the strategy adopted here is to divide M in P sub domains, each formed by the full number of columns m_x of M and (when divisible) m_y/P rows (so in "horizontal stripes"). This strategy arises from the wish to simplify as much as possible the memory management, with the aim of reducing the number of operations needed and ease the readability of the code: this in fact is the result of splitting the matrix M into pieces that are contiguous in memory[2]. By doing this, the calculation of the amount of extra data needed to take into account the "border effect" between sub domains (hereafter halo layer) and its location in memory becomes trivial: they are simply s_y rows above and below each chunk, which in terms of memory is easily translatable into $s_y m_x$ positions before and after the the beginning and end of each worker share of M (obviously master and last process only have one halo layer). Note that this is easier to implement but implies that the border is split unevenly between the processes. Since working on the border requires more FLOPs (need for renormalization), the workload is adjusted: if P does not divide m_y, the work is distributed evenly among the "middle processes" which have a small amount of border, and the remaing (lesser) amount of work is divided between master and last which have a bigger amount (the least workload is always given to the master) accordinig to the formulas:

$$\begin{aligned}
&\text{Middle}(m_x, m_y, P) = m_x(m_y + P - m_y \bmod P)/P \\
&\text{First}(m_x, m_y, P) = m_x(m_y - (P-2)(\text{Middle}(m_x, m_y, P)/m_x))/2 \\
&\text{Last}(m_x, m_y, P) = m_x(\text{First}(m_x, m_y, P) + ((m_y - (P-2)(\text{Middle}(m_x, m_y, P)/m_x)) \bmod 2))
\end{aligned}$$

The sending and receiving of the chunks of M has been implemented by means of non-blocking `MPI_Isend()` (of the workload + halo layer) and `MPI_Recv()`, so in a linear fashion. This way, while sending, the master process can work on setting up the requirements for the collective operation `MPI_Gatherv()`, used when all workers return their workload elaborated and without halo layers. In virtue of our split, putting the matrix together is very simple and actually is done automatically by `MPI_Gatherv()`. Master process work all the time in the final array, in this way it has not to move his processed chunk, since it is already in place (codewise this is achieved by the `MPI_IN_PLACE` token).
In the working assumptions is assumed among the rest that $k_y \ll m_y$: this is however insufficient to guarantee the proper working of the code as is. In fact, the actual submatrix S_P processed by each worker is possibly non square and has dimensions m_x and $\sim m_y/P$, so to ensure that the halo layer is smaller than the chunk itself [3] the final assumption of $s_y < m_y/P$ is needed. This does not seems a very strict assumption, but the limit can be approached pretty frequently with small to medium matrices and a lot of processes, and so, to relax it, some corrections have been done on the code. Instead of sending the whole "theoretical" halo layer at each step, the algorithm uses as much space as it has, checking each time not to go out of boundary when splitting the workload. Anyway, the weaker requirement that $P < \min(m_y, m_x)$ cannot be dropped using this approach[4] and the code works best when $P \ll m_y$.

2 Performance Model

In this section a model for computational time required for execution is made, depending on the relevant quantities defined above. First, we restrict our attention to the MPI code: the total time depend on communication time, computation time and in the end I/O and overhead. Communication time (which by algorithmic choice

[2]The rows are used since C implements a row-major order.
[3]This is necessary to avoid having an halo layer that extends beyond the limits of the complete matrix M.
[4]If $P > m_y$ but $P < m_x$, M can be transposed before and after to get a matrix compliant with the requirements.

should be the sum of all the $P-1$ communications between master and workers) may be calculated using a latency-bandwidth model as follows:

$$T_{comm} = (P-1)\left(\lambda + \left(\frac{m_x m_y}{P} + 2m_x s_y\right)/B\right) \tag{1}$$

where λ is latency of the network and B its bandwidth[5].
Computation time may be split into swapping and convolution, using a coarse estimation of the FLOPs required in this way[6]:

$$T_{conv} = f\frac{m_x m_y k_x k_y}{P} \quad \text{and} \quad T_{swap} = s(\frac{m_x m_y}{P} + 2m_x s_y) \tag{2}$$

where f and s are supposed to represent the frequency at which such operations can be done by the machine. The swapping has more operations to do with respect to the total size of M because of the fact that also the halo layers must be swapped by each process, so some areas are swapped more than once. Finally I/O and overhead can be estimated as:

$$T_{I/O} = am_x m_y + bk_x k_y \quad \text{and} \quad T_{overhead} = oP \tag{3}$$

The I/O time represents the reading/writing of M and reading/initialization of K, while the overhead is supposed to be due to the creation/destruction of the processes and so proportional to their number.
For the OMP version a very similar model can be built, just discarding eq. (1) and removing the halo layer dependency in the swap time of eq. (2).

Once the time dependency on P, m_x, m_y, k_x and k_y has been modeled, also the scalabilities, here defined as $S(P) = T(1)/T(P)$, can be calculated. They are just differing one to another by means of what is kept constant and what varies.

2.1 Strong Scalability Model

In the strong scalability scenario, the workload is kept constant, so it means that all the parameters m_x, m_y, k_x and k_y are also constant. The time equation can be therefore be rewritten in a simpler form keeping only the P dependence

$$T(P) = \alpha P + \frac{\beta}{P} + \gamma \tag{4}$$

This formulation keeps its validity in both OMP and MPI case, the important thing to keep in mind is that each formulation brings different meaning (and hence possibly different numerical values) to each term. Using this formulation the scalability curve should have the form

$$S(P) = T(1)/T(P) = \frac{\alpha + \beta + \gamma}{\alpha P + \beta/P + \gamma} \tag{5}$$

2.2 Weak Scalability Model

In the weak scalability scenario on the other hand, the workload is not constant, but increases linearly with the number of processors. This means that the total number of elements in the matrix M increases as P. A fair way of obtaining this is to keep the proportion m_x/m_y constant, and so increasing each dimension of M by a factor of $\sqrt{P}$. So, by keeping this dependence and all other parameters constant, the time equation can again be rewritten in a simpler form keeping only the P dependence

$$T(P) = \alpha P + \beta P\sqrt{P} + \gamma\sqrt{P} + \delta \tag{6}$$

[5]Of course since all the processes are executed in the same node these numbers are expected to be respectively small and large. Also in this study the communication time due to the collective operation `MPI_Gatherv()` are considered to follow the same scheme.

[6]The precise number of FLOPs during the convolution can be calculated analytically and yields the following result in the working assumptions:

$$\text{FLOPs} = (m_x - s_x)(m_x - s_x)k_x k_y + 2(k_x + s_x - 1)(k_x - s_x)((3s_y - 1)s_y + k_y(m_y - 2s_y) + k_y(m_x - 2(s_x - 1))) \sim O(m_x m_y k_x k_y)$$

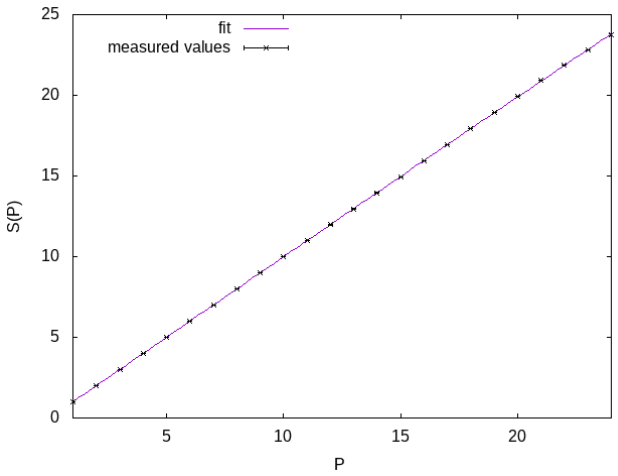
Of course this estimation will be good in the limit oh high workload and lose some accuracy otherwise

Again this formulation is valid in both OMP and MPI case with the previous concern. Finally, using this formulation, the scalability (or *efficiency* in this case) curve should have the form

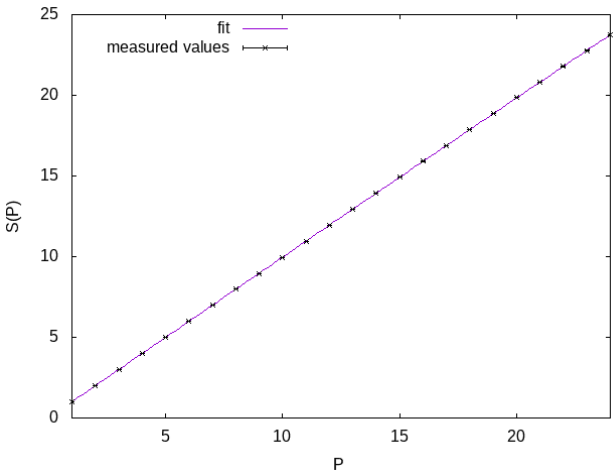
$$S(P) = T(1)/T(P) = \frac{\alpha + \beta + \gamma + \delta}{\alpha P + \beta P\sqrt{P} + \gamma\sqrt{P} + \delta} \tag{7}$$

3 Scalability Study

Both a strong and weak scalability study have been performed, to assess the code and check the validity of the models proposed. A total number of 8 runs have been done as requested, each with a different combination of parameters. All of them used a number of processes/threads starting from $P = 1$ (serial case) to the maximum number of physical cores available, in this case $P = 24$. Each configuration ran 3 times to average the times and eliminate some fluctuations. No particular strategies were used to enforce the pinning of the processes to a specific CPU. Due to the long queues on GPU nodes, each of these request run on THIN nodes. In particular, the following combination of parameters have been used: MPI program with $K_{dim} = k_x = k_y = 11$ and $K_{dim} = 101$, and OMP program with $K_{dim} = 11$ and $K_{dim} = 101$. In all (strong scalability) cases the the matrix M used was the PGM image "earth-large.pgm" with $m_x = m_y = 21600$, while for the weak scalability as sequence of PGM images $\{M_P\}_{P=1}^{24}$ of white noise have been generated following the relation $m_x(P) = m_y(P) = m_x(\text{"earth-large.pgm"})\frac{\sqrt{P}}{\sqrt{24}}$.
Firstly, we can observe that the parameter choice agrees with the working assumptions, so no major losses in the efficiency should be present. Moreover two different timings systems have been used: `/usr/bin/time` and internal wall-times, similarly to the previous assignment. They are pretty much equivalent in all the points[7], so the `/usr/bin/time` timings only are used here to ensure the presence of the parallel overhead already accounted in the model. As a second step, now we can plot the obtained result[8] alongside the corresponding fits for the models and comment[9]. As we can see, the two results resemble a straight line, meaning that the dominant part of the calculation is fully parallelizable and the term $\propto P$ is therefore the most important.



Strong scalability - OMP - 101



Strong scalability - MPI - 101

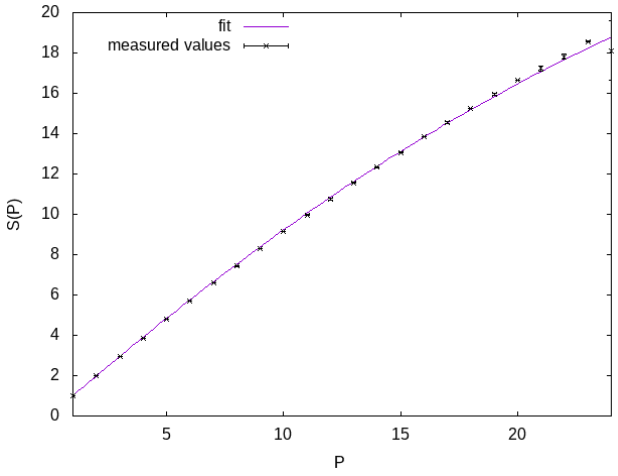
Using a smaller kernel instead, the serial part of the program starts to become important: this is due to the decrease in workload that reduced drastically the amount of FLOPs during the convolution, and so the parallelizable part of the code.

[7] Except for the last two points for the OMP 101 weak scalability, in which a sudden and (relatively) large discrepancy happens, thus making the plot less regular and resulting in a worse global fit.

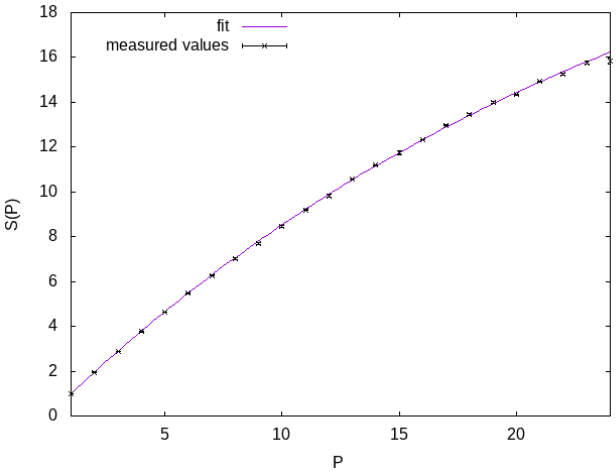
[8] The scalabilities are normalized to unity in the serial run. For reference, here are the absolute timings for these runs:

Timings	OMP - 11	OMP - 101	MPI - 11	MPI - 101
Strong Scalability	77.23 s	6132 s	78.62 s	6146 s
Weak Scalability	3.29 s	252.9 s	3.53 s	253.6 s

[9] The complete results for only strong and weak scalability are given separately and so they're not reported here.

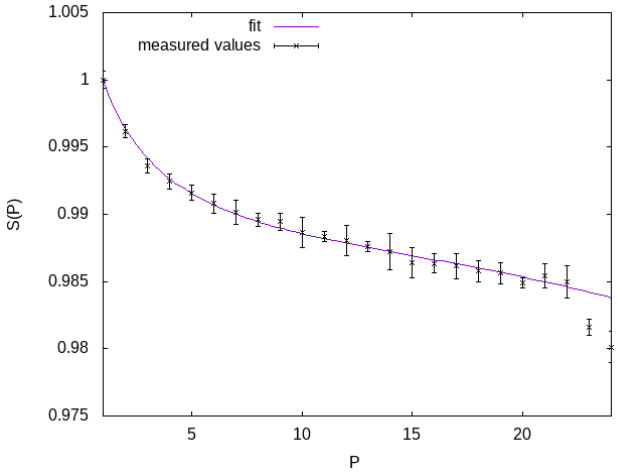


Strong scalability - OMP - 11

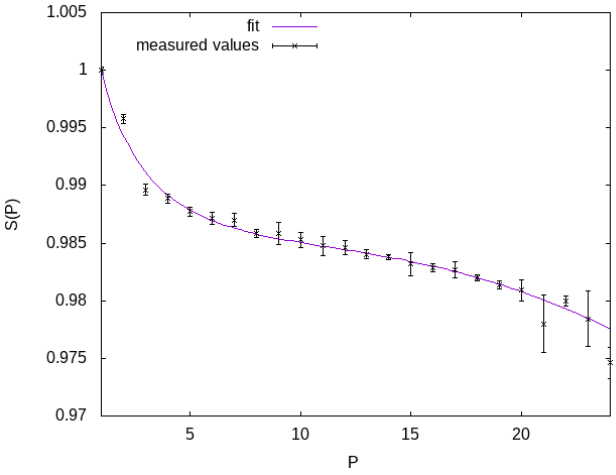


Strong scalability - MPI - 11

In the weak scenario the dominant part is still the calculation (so in this case the constant term), as stated above, and the main reason for which this pattern emerges (and can be predicted by the curve) is the scale: only about a 0.02 change happens in the scalability along the whole x-axis, meaning we are observing higher order variations!

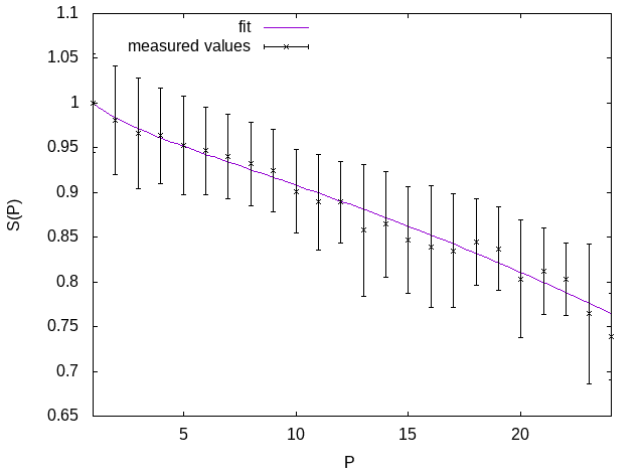


Weak scalability - OMP - 101

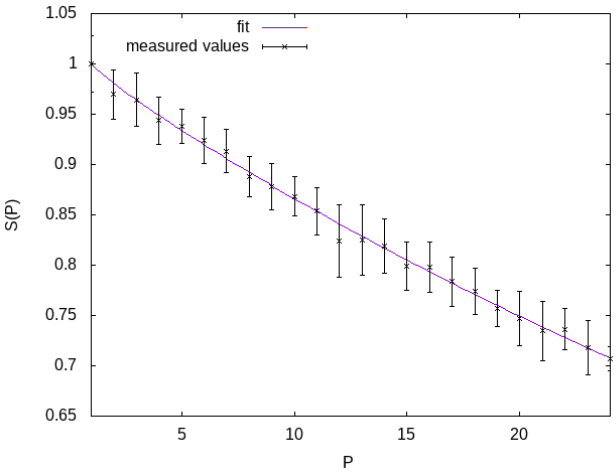


Weak scalability - MPI - 101

Finally, again by lowering the the workload, the serial part becomes important again, and this causes other factors, in this case the linear term (associated to communication, I/O and overhead), to play an important role, and deviate from the ideal case of constant efficiency.



Weak scalability - OMP - 11



Weak scalability - MPI - 11