Assignment 2

Davide Roznowicz

Description for the MPI implementation

A 1D decomposition is implemented. The possibility of a 2D decomposition was carefully taken into consideration: its improvements over a 1D decomposition are clear when the number of cores gets very high as this case would imply less memory buffering for 2D compared to 1D as well as a more balanced structure and likely less time to perform the computations. However, due to the nature of the cluster (the number of cores is not that high) and the specific requests for the assignment (using the 24 physical cores in one node), it looked like a 2D decomposition would not guarantee any actual benefit. Moreover, a standard 2D decomposition in the case of the number of cores used in the scalability studies would often turn into a 1D decomposition, or close to it (just think about prime number of cores).

The explanation for the implementation of the blurring of an image performed by the file called blur.mpi.c consists of the following parts:

- Initialization of the MPI process via **MPI_Init** ; after declaration/initialization of variables for subsequent usages, command-line arguments are properly parsed according to the specific requests of the assignment.
- To avoid unnecessary communication and further buffering, we decided to let each process open its own file containing the pgm image, which is accessible across the whole cluster by each core. Thus, each process parses the header and then reads it all up to the piece of matrix of interests, saving in a proper array only the part of it which will be useful for its own computations. More specifically, the distribution among cores is such that each process receives a number of lines of the image equal to: ny/P, where ny is the height of the image, P is the number of cores and the operation is intended as an integer division. If ny%P != 0, then an additional line is assigned to each core starting from the first one (the master) and continuing until the extra lines have all been distributed. This way, the additional workload tends to be charged on the cores which should finish earlier their own reading. It is relevant to say that each separate matrix, where every process saves its own read data, has a profile (a contour) all around of size $half\text{-}size$ (as defined in the assignment) that serves as the halo layers: this solution seemed more suitable for the problem, as a Jacoby-style buffer-sending-approach looked inappropriate (there are no iterations that require continuous exchange of halo layers).
- The kernel matrix (be it average, weight, gaussian) is computed by the master core, which should be the first one to finish reading. Since this matrix tends to be quite small, this workload should be well absorbed by the master while others are still reading. Then, an **MPI_Bcast** algorithm sends the kernel matrix to all the processes.
- The actual blurring is now performed by every process involved, which saves its own final result in a new matrix (without contour this time). After preparing the required input, a **Gatherv** algorithm actually gathers the data, in an ordered way, in the matrix that will be written down as blurred image.
- The master core produces the output name for the image file, then opens it: after the header, the whole matrix is written down. Lastly, an **MPI_Finalize** ends it all.

Description for the OpenMP implementation

For reasons that are similar to what we discussed about MPI, a 1D decomposition has been adopted in OpenMP, too: every thread is in charge of a subset of lines of the matrix that is proportionate among threads.

The specific number of lines each thread gets depends on the round-robin policy adopted in the **for ordered** clause.

The explanation for the implementation of the blurring of an image performed by the file called blur.omp.c consists of the following parts:

- After declaration/initialization of variables for subsequent usages, command-line arguments are properly parsed by the master thread according to the specific requests of the assignment.
- Only the master thread parses the header. Afterwards, the parallel region begins: a specified **proc_bind(close)** clause allows each thread to reach out the shared memory banks belonging to the other cores in a faster way.
- A previously allocated buffer, necessary to contain the image to be blurred, is initialized by each thread with the data just read from the input file (in a **for ordered** clause): it happens in an ordered way (almost sequential), according to the $first\ touch$ policy concepts. This way, every thread will soon be working on a big bunch of data that is local. It's important to notice that the buffer is bigger than the image matrix: a contour of entity $half\text{-}size$ allows each thread to perform the blurring on the borders, too. This contour is zero-initialized in parallel according to $first\ touch$, again. Please note: as this buffer is shared among threads, the halo layers often don't belong to the local memory bank of the thread; luckily, using the close binding policy, the data is still quite close. However this effect should not be particularly relevant if the kernel matrix is not very large, like in our cases. This looked like a good trade-off between amount of buffered data and performance.
- The kernel matrix is performed in parallel in a **schedule(dynamic)** mode. If the kernel is gaussian, a reduction is needed on the accumulator to avoid a data race. The kernel matrix is initialized without any respect to $first\ touch$: in fact, being the kernel matrix quite small, it will be put **firstprivate** at the point of computing the blurring, allowing full exploitation of data locality.
- The actual blurring is now performed in parallel by every thread involved, saving the final result in a new matrix (without contour this time). Some useful temporary variables are put **private**, while a **reduction** clause is necessary to avoid data race on the accumulator.
- A single thread produces the output name for the image file, then opens it and writes down the header.
- The master writes the final matrix and closes the file. Then, the parallel region is closed.

Performance Model

By carefully studying the codes both for MPI and OpenMP, a theoretical model can be derived. We try to find out the major terms, i.e. the ones that somehow tend to prevail over the others (taking most of the time), even if we change some conditions (size of the square kernel, size of the square image...). This implies that we ignore or at least give less credit to certain pieces of code (or operations) that are believed to be some orders of magnitude less than others. This allows us to produce an approximated model to forecast the time by varying the number of cores/threads as well as the sizes of the image/kernel.

We define $T_{read\ image}$ as the time requested for reading and saving into the buffer; T_k as the time necessary to initialize the kernel matrix; $T_{broadcast}$ as the time needed to send the kernel to all of the processes and save into the local buffers; T_{calloc} as the time it takes to initialize to zero the local buffers for the piece of image belonging to each core; $T_{write\ image}$ as the time required to write down the whole image in the output file; $T_{gatherv}$ as the time to gather together all the pieces of the blurred image, plus saving it into the buffer; $T_{compute}$ as the time for computing the whole blurred image; N_m, N_k as the number of bytes of the whole image and the kernel, respectively; T_{save} as the time to save one unit in the buffer (let's say one byte); T_{comm} as the time to send one unit (again one byte of data); T_{read} as the time to read one unit (one byte); T_{write} as the time to write down one byte of the image; λ as the latency concerning the communication among cores; P as the number of cores/threads:

We now estimate the model for MPI (fl stands for floating point, int for integer):

- $T_{read\ image} \propto N_m(T_{save} + T_{read})$
- $T_k \propto N_k T_{save}$: this term is present only if k is sufficiently large, otherwise it should be masked by the reading time as it was previously explained in the description.

- $T_{broadcast} \propto \lambda + (\log_2(P)N_k)T_{comm}$
- $T_{calloc} \propto (\sqrt{N_k} + \frac{\sqrt{N_m}}{P})^2 T_{save}$
- $T_{gatherv} \propto \lambda + \frac{N_m}{P} \log_2(P) T_{comm}$
- $T_{compute} \propto \frac{N_k N_m \quad fl\ mult + (2N_k N_m + 4\sqrt{N_k} N_m) \quad int\ sums + N_k N_m \quad fl\ sums + 2\sqrt{N_k} N_m \quad int\ mult}{P}$
- $T_{write\ image} \propto N_m T_{write}$

Similarly, in the case of OpenMP all the main terms are kept except the communication times (shared memory here), while $T_k \propto \frac{N_k}{P} T_{save}$ is initialized in parallel. Moreover, $T_{calloc} \propto \frac{d(2\sqrt{N_m}+d)}{P} T_{save}$ because only the contour is initialized to zero (this term can be neglected, though). Summing all the terms up, we obtain an estimate of the theoretical model for our code.

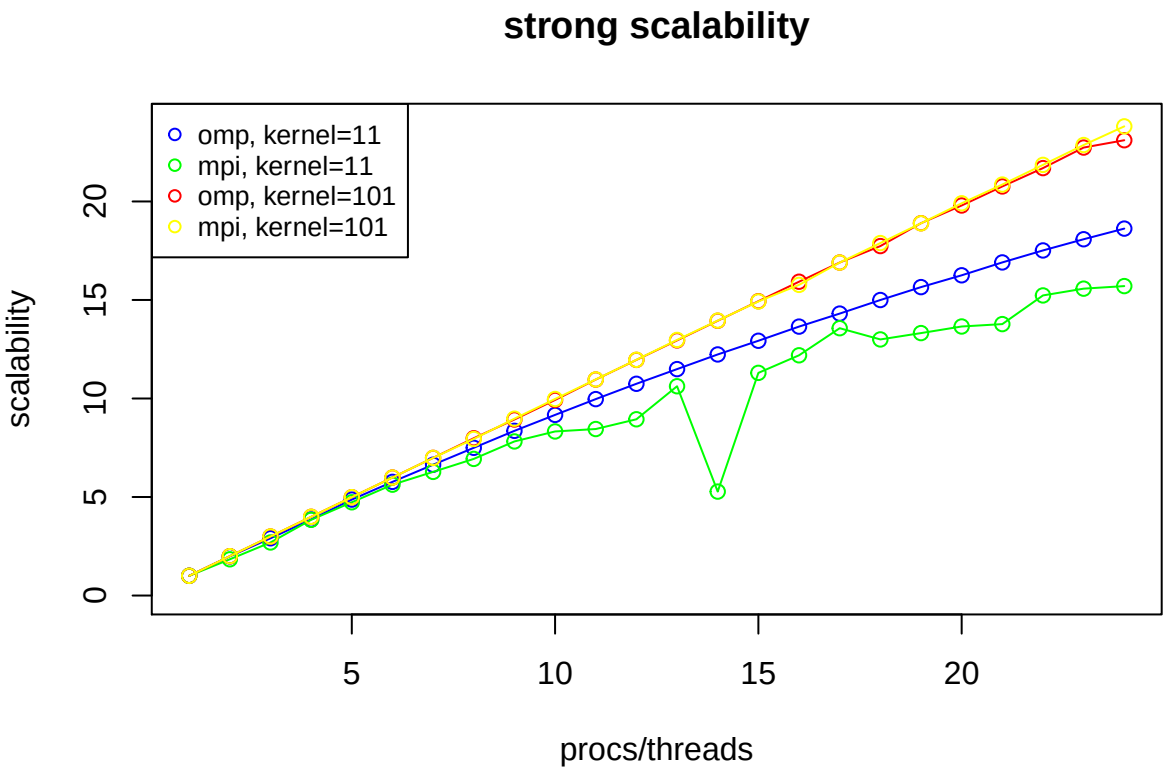
Scalability Analysis

The scalability analysis were performed on one single thin node, i.e. 24 physical cores. The files were compiled using -O1 for MPI and -O3 for OpenMP, after making sure that these optimizations don't change the real structure of the code. Three repetitions were made to better assess the trend (the .csv files for the timings can be found in this folder, or by reading the README) while varying the number of cores/threads: the average was then used to complete the study and plot the charts. Time begins just before reading the image/initializing the buffer.

Please notice: everything was performed according to the details of the assignment BEFORE the relaxation of the workload was granted.

Strong Scalability

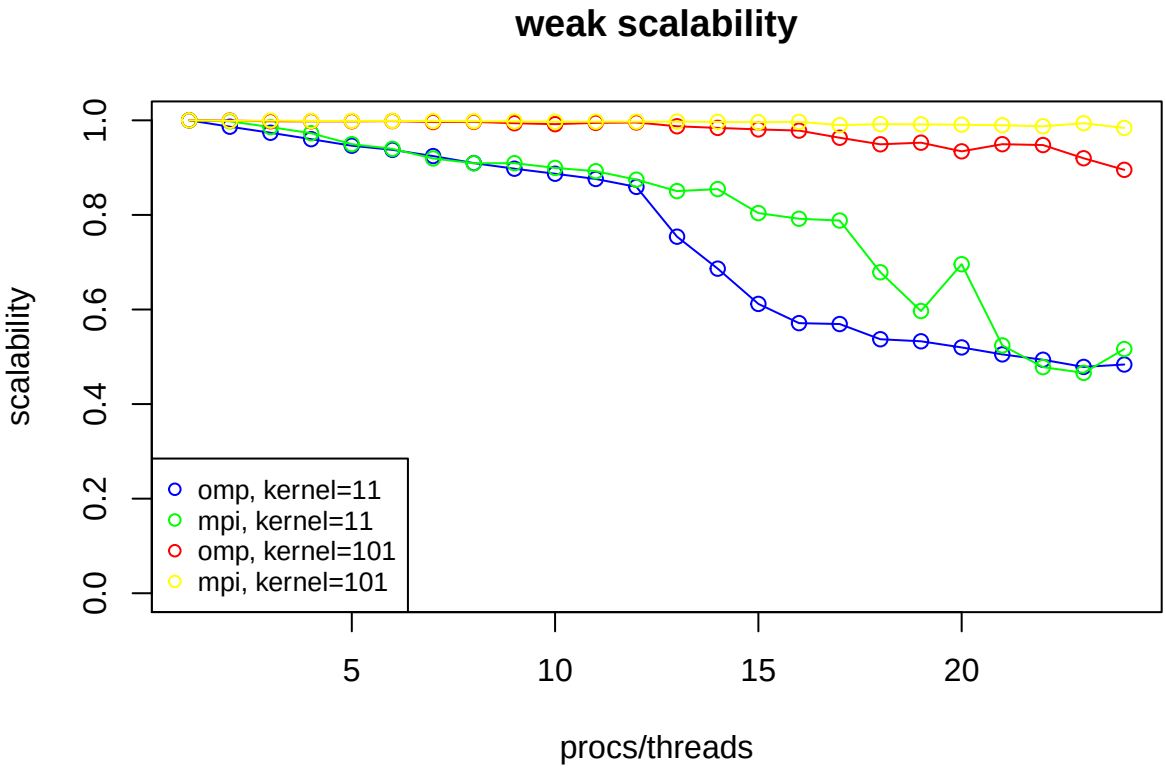
The image *earth-large.pgm* was used for the analysis.



Of course, the scalability lines related to the smaller kernel are less straight than the ones for the bigger kernel: this is because, in the case represented by the latter, $T_{compute}$ prevails over the others by a huge margin. This means that most of the time (almost all of the time) is spent crunching numbers in order

to perform the blurring. This is not true anymore when we consider smaller kernels: the other terms are not negligible anymore. $T_{compute}$ is still a relevant factor looking at time usage; however, T_{read} and T_{write}, which are sequential terms, get more and more relative importance as we increase the number of cores. Similarly, the communication times embedded in $T_{broadcast}$ and $T_{gatherv}$ increase their weight on the whole time performance. By collecting further data, we realize that $T_{read\ image}$ (in the case of *earth-large.pgm*) is between 0.3 s and 0.4 s, while $T_{write\ image} \approx 0.45$ s; $T_{broadcast} \approx 0.35$; $T_{gatherv}$ is between 0.35 s and 0.45 s. When using just one core/thread, the total time is around 65-66 s, thus the just mentioned terms are still quite small; when increasing the number of cores/threads, the computational time decreases but the other terms remain constant and start eroding the scalability.

Weak scalability



For the weak scalability, $earth - large.pgm$ was fed to 24 cores/threads. As a consequence, via a *create_image.c* file, a proportionate square image (or almost square) was created each time that the number of cores/threads changed. Therefore, P cores/threads were assigned an image containing a number of pixels equal to $\frac{P}{24}$ of the number of pixels of *earth-large.pgm*.

Space for Improvements

We might have gone through some further optimizations of the code, by digging more into the following:

- careful loop unrolling
- speeding up the code through more temporary variables, avoiding unnecessary integer operations especially in some expensive inner *for cycles*.
- producing less buffering/allocation not needed