Huu Danh Nguyen
Gloria Tabarelli

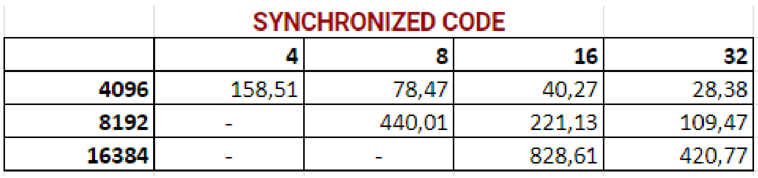
MPI assignment
Parallel MPI solution of 2D-laplace equation

2019/12/17

1 Basic MPI solution

The code is contained in the file `lapFusion-task1.c` while the script we use to run the code in WILMA cluster is in the file named `submit_job_script.sub` (with the parameters initialized for a specific test case).

In the following table we report the time-performance results (in seconds) of this code version:



SYNCHRONIZED CODE

	4	8	16	32
4096	158,51	78,47	40,27	28,38
8192	-	440,01	221,13	109,47
16384	-	-	828,61	420,77

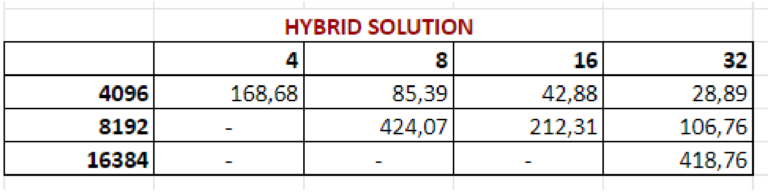
Figure 1: Performances of synchronized code

2 Improving the solution

We choose to implement the first hint solution and the hybrid one with OpenMP, the latter including also the previous.

Focusing on the "Block partitioning" solution, we can notice that the total number of communications needed is increased (each block needs to send between 2 and 4 lines to others blocks and to receive the same amount), but the size of the data communicated each time is reduced. Actually, if we have N processes, and we assume that A is a square matrix, with each block having a square form, each communication between blocks will carry $\sqrt{N}$ elements (assuming also that N is a perfect square). In our perception this will result in a decreasing of the time needed for each communication, but, since the total number of communications is higher, this advantage could be deleted by the time needed to complete all the communications between blocks (probably processes will have a bigger latency time). Moreover, if the matrix size is not big enough with respect to N, processes will have few work to do, and this could increase their latency time, contributing to delete the advantage of the reduced amount of data carried by each communication. Also the fact that we need some condition command to decide which is the number of lines each processor needs to send and receive will take time, increase the complexity of the code and make the program slower. All of these together lead us to discard this possible optimization.

About the "Block communications" improvement, we noticed that the total number of communications is reduced (actually divided by k), but the data size of each communication is multiplied by k. Moreover, as said in the suggestions, each process computes 2*(k-1)*n elements more than in the first implementation. So we have less communications, but bigger, and we have to compute more things. This suggested us that the structure of this implementation is similar to the previous one, indeed both require to play and find a balance between the total number of communications and the size of each of them to obtain a faster execution time. This, added to the observation that in this solution there is a lot of memory consumed just to store the communications' data, lead us to choose to implement the first and the fourth suggestions, which in the choosing phase, apart from the time needed to create the threads in each node with OpenMP for the fourth one, and the need to define the inner matrix and the outer matrix for the first one, did not seem (to us) to have any particular disadvantage.
The code for the first optimization is in the file `lapFusion-firstHint.c` while the one of the fourth is in the file `lapFusion-combine-first-fourth-hint.c`.
Here follow the results of the time performance of the hybrid solution.



	HYBRID SOLUTION			
	4	8	16	32
4096	168,68	85,39	42,88	28,89
8192	-	424,07	212,31	106,76
16384	-	-	-	418,76

Figure 2: Performances of hybrid code

3 Assesment of the solutions

From Figure 1 we can analyze the strong and weak scalability of the synchronized program: the weak scalability results can be read on the diagonal, while for the strong scalability we can choose to read columns (for the matrix size variations) or rows (for the cores number variation).
We can observe that the weak scalability seems to be linear once the problem size increases, indeed multiplying by 4 the matrix size and by 2 the available resources we obtain an (almost) doubled execution time only for the bigger sizes of the matrix.
The same holds for the strong scalability when fixing the number of cores: 4 times the matrix size implies a 4 times execution time only when the problem size grows.
This could be explained by the fact that if the problem is not big enough the communication-overhead is significant.
Instead, if we look at the strong scalability for the fixed matrix size, we see that doubling the number of cores will make the computation time the half.
This, together with the previous analysis, could lead us to the conclusion that the size of each communication, and not their total number, is a bottle-neck for the execution time in this problem.

From the analysis of Figure 2, we can make a comparison of the performances results we obtained and we see that for the not syncronized communications, in a couple of small cases, the total time is almost the same as the syncronized case. This could be explained by the fact that if the matrix is not big, the total amount of time to send and receive the data is almost zero. In this code we want to calculate inner elements in the waiting time, but there is almost no waiting time, and so the effect of the first hint is reduced. Instead, if we increase the matrix size, we see only a little improvement on the elapsed time with respect to the synchronized version: this can be explained by the fact that the time to send and receive data increases since there are more data to transfer, and even if the processors do not have a consistent latency time in waiting for the communication to start (because they are calculating the inner elements in the meantime), they need time to send and receive a consistent amount of data.