Assignment 4
SPM course a.a 23/24

Spiridon Dragoș

Problem:

Consider the sequential code provided *nkeys.cpp*. It implements a numerical computation on a set of keywords. The number of keywords is an input parameter of the program (*nkeys*), and so is the number of keyword pairs it generates (*length*).

Solution:

Before discussing any solution, the data dependency and what the actual parallelization can be made for this problem should be presented. By going with a naïve perspective of the problem, the first assumption would be to parallelize the first "for" loop met, the one who generates the keys, and the problem would be a simple data parallel task. However, upon a closer inspection, it is not possible to do that because a global dictionary is constantly updated, and different tasks are made with the values from it, which does not let a possible parallelization of the first "for" loop. Also, the data is randomly generated, so the assumption of "One process will have to generate all of the keys" will be made. From this perspective, the problem is actually a stream parallel task, and not a data parallel one.

From these assumptions and conclusions, the problem has to be analyzed from the point of view of time service, for finding out which of the tasks costs more and parallelize it. In this problem, it is clear that the matrix multiplication is the costliest one. A simple solution would be to apply a parallelization on the rows of the first matrix and afterwards applying a reduction of the summation from each of the processes.

As a first solution, and also the simplest case, each matrix multiplication met would be separated in different processes by the row of the first matrix, and after that, a reduction on their sum will be made, like in Figure 1.

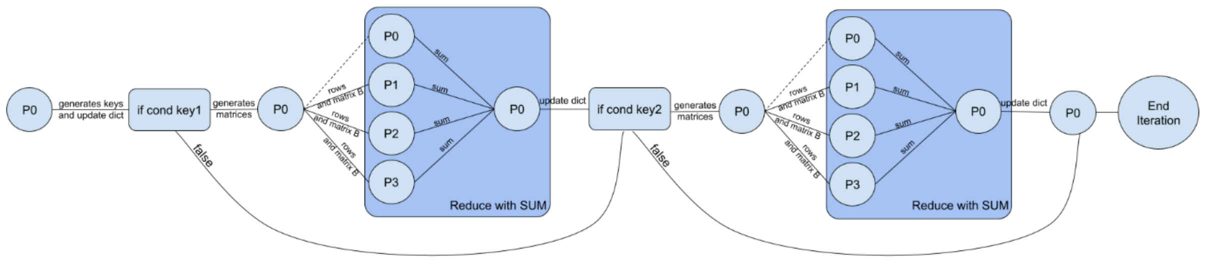

Figure 1. The skeleton of the first solution

The second solution implies that in a single generation of two keys, there are possible to happen two matrices multiplication, which gives the idea of using two communicators (created with the help of MPI_Comm_split) for each matrix multiplication that could happen: one of the communicators will have a process which will make the generation of the keys (calling him master), and the other communicator will have only a process which will wait a message from the master to tell him to start making a matrix multiplication (which will be done like in the first solution, but with less processes). After the second communicator is done with it, it will send the result (which is a float, due to only being a sum of other elements) back to the master process, which will update the global dictionary. A visual representation of the flaw of the program can be seen in Figure 2.

Note that for each solution presented, the same logic will apply to the last matrix multiplication, where the program will go through all of the dictionaries and do a matrix multiplication between each two different keys.

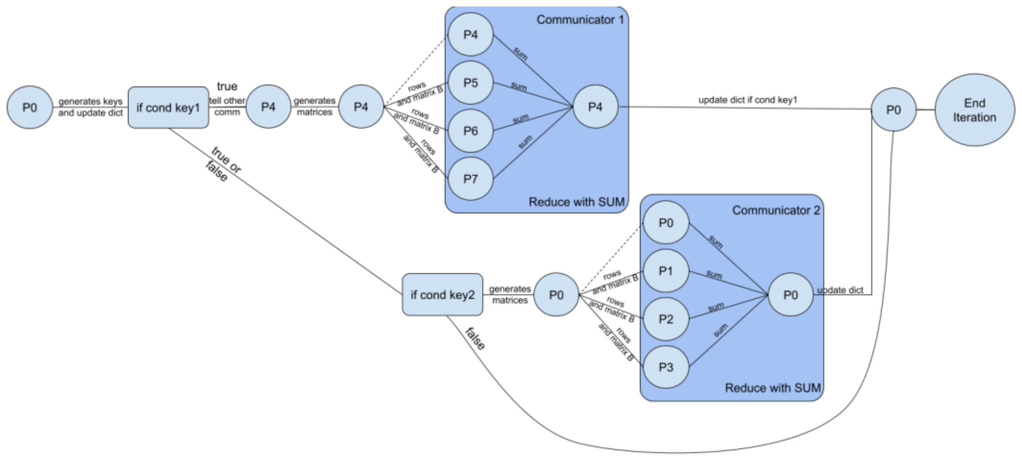

Figure 2. The skeleton of the second solution

Compile and run:

This code was run on the university cluster: spmcluster.unipi.it, using SLURM. Unlike the other assignments, there is no need to build it directly, you can only call the slurm.sh file like this:

```
sbatch slurm.sh
```

which will build and run the program with different number of processes: `2 4 8 16 32 64`, by creating 4 nodes (the split of the processes between the nodes will be made automatically by SLURM).

The results will be saved in a file called "solution_%j.cpp", where %j is the number of the job allocations.

Results:

Solutions results: in Table 1, the results are presented after building each one of them with the batch created on `slurm.sh`. It is obviously that the second solution, which is abbreviated as S2, has better results overall than the first solution (abbreviated as S1) as it always gets the best results overall (S2 on "8 processes"), and it does not increase a lot when the number of threads is too high, unlike S1. From the perspective of speed-up and efficiency, which can also be observed in Figure 3, we have two cases two evaluate. In the case of `nkeys=2`, the second solution is clearly achieving better results in any metrics, it even overcome the efficiency of the sequential one in the case of 2 processes. The other one escalates well, but not after 16 threads in usage, as it is starting to become really high. In the case of `nkeys=100`, the first solution presents better results when the number of processes lower than 8, but after that, the outcome is the similar to the other case: it starts to increase really fast. For the solution 2, even if it achieves the highest result in terms of nkeys, the efficiency goes low really fast, and also the time increases similarly as the other solution. In terms of speedup in this case, they are pretty similar, with a faster speedup coming from the Solution 1.

processes / nkeys	seq	2		4		8		16		32		64	
		S1	S2	S1	S2	S1	S2	S1	S2	S1	S2	S1	S2
2	12.946	6.72	6.27	4.59	3.79	4.12	**2.45**	6.99	2.60	16.19	5.32	57.5	18.2
100	7.8491	4.21	6.63	3.04	4.17	3.03	**2.86**	5.55	3.37	13.4	6.66	61.3	28.0

Table 1. Results from both of the solution given different nkeys.

Better approaches: for this problem, we made a strong assumption of dependency between the keys generated, which would not let us separate the generation part of the code. By this assumption, anything else could not have been parallelized except the matrix multiplication, as everything else is sequential and with efficiency of O(1). Meanwhile, the calculus of the matrix multiplication could have been made much

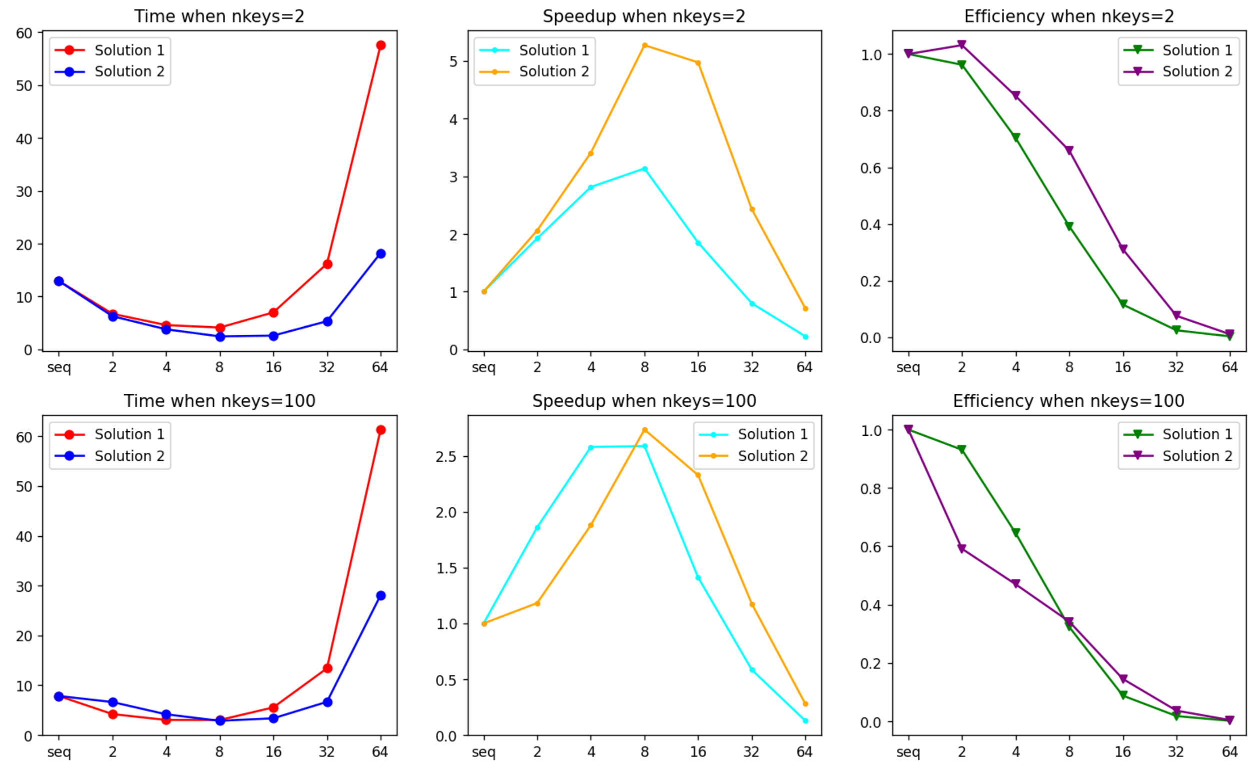

Figure 3. Results from both of the solutions

better, as it is not memory efficient to send an entire matrix to the other processes. After some research, a better method for solving it would be to apply Cannon's Algorithm, which separates in a clever way the matrices into square blocks and after that multiply them using a cartesian topology. Also, as we are interested only in the sum of each element, maybe there is a solution for the multiplying to be differently approached, as we are not interested in the matrix result, but in a sum of elements result.

Conclusion

By working with splits, working on this problem made it easier to solve the issue of solving two matrix multiplication tasks in the same time. Also, working with two communicators make the calls for each process easier as the index of them resets between 0 and half of the processes, and nonetheless the calling of `MPI_Reduce` is possible with two communicators by this split, therefore the management of this code does not overcomplicate.

From the perspective of MPI usage and how much I worked for the code to work, it took a really long time to understand some concepts of `MPI_Comm_split`, and also understand how to make two processes from different communicators interact within the `MPI_COMM_WORLD` communicator. There were a lot trials with `MPI_Scatter` and `MPI_Intercomm_create`, but all of them failed. In my case, simpler was better for splitting the data, and the inter-communicator was not needed after all.

In conclusion, the assignment is a stream parallel task, which can be solved by parallelizing the matrix multiplication inside them. By considering that in each iteration there can be made 2 matrix multiplications, a solution for efficiency is to split the number of processes into two communicators and give each one of them a matrix multiplication to calculate. With this solution, it seems that the results are improving a lot, and it helps divulging the other functions of MPI which could help us to improve the code from the point of view of efficiency and clean code. Even though better solutions exist, like applying a different algorithm for solving the matrix multiplication (Cannon's algorithm), the second solution shows good performances from the perspectives of efficiency and speedup, when it comes to `nkeys=2`, but also satisfactory results in the other case, even though the efficiency drops a lot from the start.