Iterative Sparse Matrix-Vector Multiplication (iSpMV)

Buğra Sipahioğlu

Comp 429, Asst.Prof. Didem Unat

June 8, 2020

Introduction

Part 1 and 2 of the project were implemented but only the first part is tested. In the source code, the evaluator of this project can find developer comments by searching “BUĞRA:” in the files.

Part 1: Row-wise Partitioning and Parallelization with MPI

With respect to implementation, all the data which are subject to transfer was kept in the arrays. Scatter, scatterv, and broadcast functions are more effective than send and receive. Thus, the master thread distributed the workload to each process, and put the information to different arrays such that the ith index of an array was sent to ith process. For example, *nnzList* keeps the non-zero elements per each process, such that first element of the list corresponds the process with ID 0 (master). In the project, it is also assumed that number of total rows is a multiple of number of processes. If not, program exits.

Basically, each process needs to know total number of rows (for vector calculations), number of rows that it will handle, time steps, nnz (number of non zeros) and the displacement values for rows and nnzs. These displacement values were used due to non-equal sizes of nnzs per process, thus, the scatter-v function had the input of these displacements to scatter unequal data.

After all the necessary data is either scattered or broadcasted, computation starts. In the computation, after each time-step; *allgatherv* function was called because the local result array needs to be the same for every process after each iteration.

Part 2: Part 1 with MPI + OpenMP

Two for loops were parallelized for the optimization with OpenMP. First for loop is the loop where the local result is calculated. Second for loop is the loop where the rhs vector is updated with the latest results. Static scheduling was used since the workloads are approximately the same between processes. Also, local result must be firstPrivate because it is initialized outside of the loop and it must be specific to each process. Other variables are global variables which are reached from inside.

Part 1: Experiments

Experiment was conducted on a single node in the Intel® Xeon® Gold 6148 CPU with 2.40 Ghz clock frequency. Here are the steps to execute the code on the cluster once entered in the *sipahioglu* directory:

1- module load mpich/3.2.1

2- module load openmpi/3.0.0 (note that openmpi 4 has some problems, it should be 3)

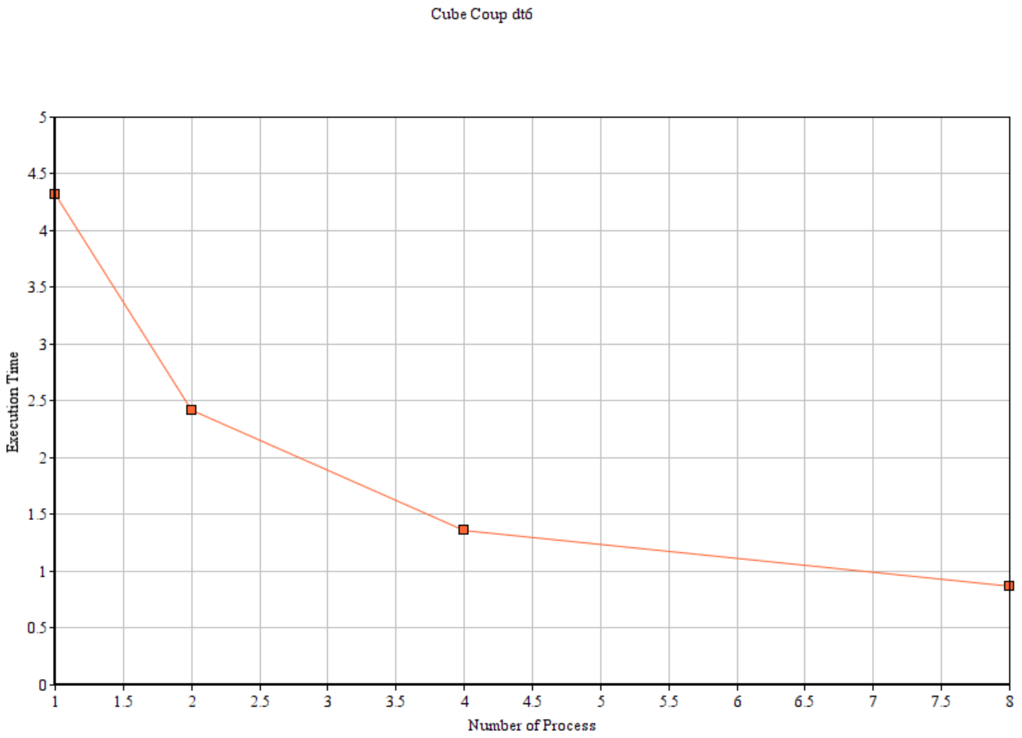
3- cd part-1 && make && cd ..

4- cd serial && make && cd

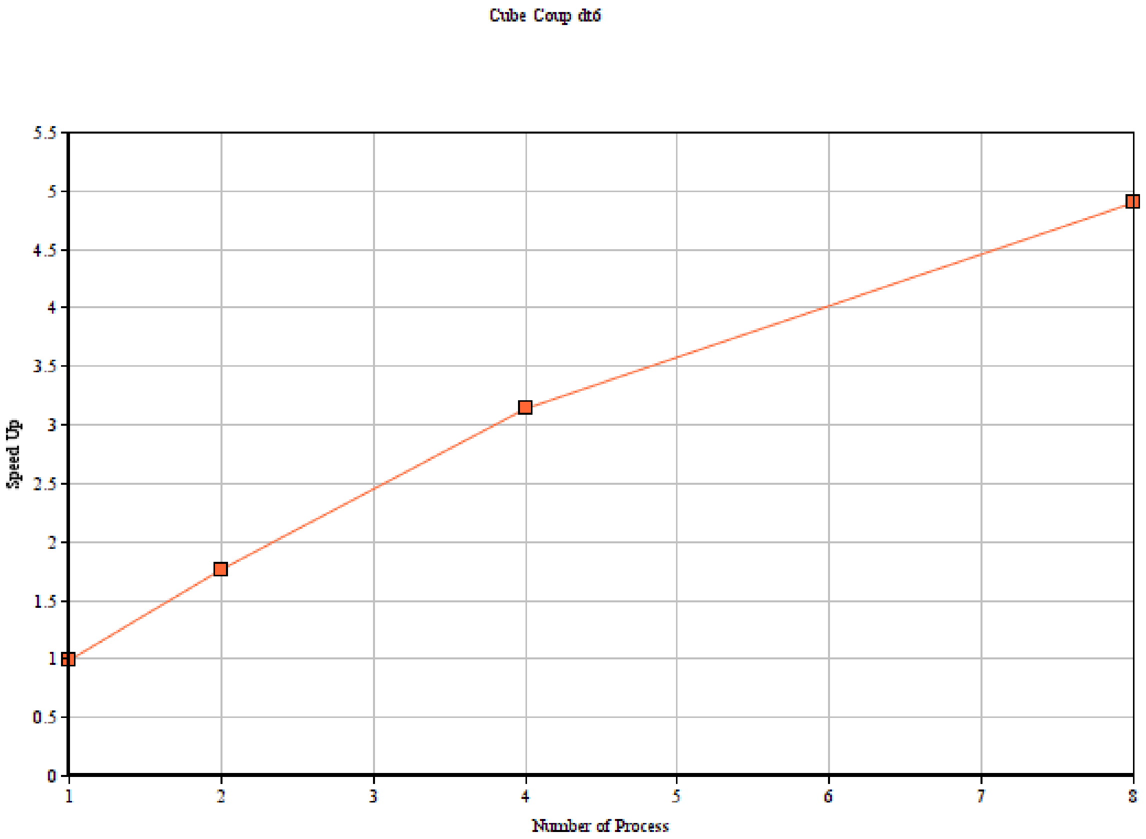
5- sbatch submit_job.sh

6- cat spmv-job.out

Note that number of rows was assumed to be the multiple of number of processes, hence tests does not include runs with 16 processes for Cube Coup matrix. Figure 1 shows the execution time with respect to number of processes used for Cube Coup matrix.



As seen in the Figure 1, the parallelization with processes decreased the time spent on computation in a respectable amount. Figure 2 shows the speed up versus number of processors used for Cube Coup matrix. Speed up is calculated with respect to serial run time of the Cube Coup matrix, which is 4.27 seconds.



Although the tests were not conducted with 16 processes, almost x5 speed up was achieved during the experiments. Since the creation of processes and MPI environment has a bottleneck, the speed up is less than 1 when the number of process is 1.