Parallel Programming Report of the 2nd Assignment

Φωτεινή Κοκκώνη

This is the report of the second Assignment, first there will be a guide on how to get set up the files and run them appropriately.

Step1:

We download the files and upload them to our folder/dir on Aristotelis.The zip code will consist of the OpenMPI+OpenMP code ,the .sh file we need to run a batch and our graphs.
The .sh file when you submit a job creates an output file where you can see the progress of your job, it technically works like a console with prints.

Step 2:

The .sh contains all the compiling and graph loading therefore you only need to do:
chmod +x nameoffile.sh
sbatch nameoffile.sh graph.mtx

Mind you this code was mostly crafted specifically for the demanding graphs of our previous Assignment, them being com-Friendster and Mawi_201512020330.

Step 3:

Go to Aristotelis's Active Job and check if the job is queued or running, then go to your folder/dir to see the output file. Click on it {open it} and hit refresh now and then to see the progress being printed

Approach Strategy and Code Analysis

This code does not differ a lot from my original OpenMP code but the use of OpenMpi allowed me to speedup the process reaching almost ½ of the best time I had achieved with open_cilk and ¼ of the time I have achieved solely with the use of OpenMp.
My strategy is simple, since we have multiple nodes with multiple processors and distributed memory then I had to "cut" my problem into chunks/parts and create a communication between the processors that will be proper and in sync to avoid race conditions and also avoid "ghost vertices"[1]. Inside each different process "chunk" there will be OpenMp threads which will execute parallel computations to achieve minimum process duration. In other word each processor will have a piece of the graph to label and cross in parallel with the use of threads, so our CSR is being built "in pieces" which have their own memory and communicate with each other so that they can "unite" in the end to form the CSR and the CC process.

More specifically, my rank 0 ONLY reads the graph ,converts it in CSR format and then Broadcasts it to all of my MPI ranks, which means that all of my MPI process hold a copy of the graph thus there is no graph partitioning in my memory. Each rank(rank number=number of our processors) "own" some vertices and stores only their adjacency lists but with this implementation we faced "ghost vertices".

What are ghost vertices?

Ghost vertices are basically vertices that are neighbours of the "main vertices that belong to the given chunk of our rank. In order to correctly handle said ghost vertices we make sure synchronise their values through communication with the ranks that owns them.

How to handle ghost vertices

We separate them and “save” them in an array and later on each of our ranks “send” an update of their owned vertices thus the neighbours receive them and update their own ghost vertices.

Parallelism inside each Rank Via OpenMp

Just like the first assignment I use the openMp threads with commands like #pragma omp parallel for, to make all of my threads execute the for loop for both the owned and not owned vertices but with a slight change on my implementation on the owned ones. I decided to use #pragma omp parallel for reduction(||:local_changed) to process a subset of rach of my rank’s vertices,then I scan the neighbours before performing a label relaxation

label relaxation

It updates each vertex’s label to the minimum label among itself and its neighbors. This causes the smallest label in each connected component to propagate through the graph. The process repeats in a synchronous manner until no labels change between iterations. At convergence, all vertices in the same connected component share the same label.

Synchronising the Processes Via OpenMPI

MPI_Allreduce performs a logical OR on all of my MPI ranks If any of my ranks keeps changing labels then keep repeating the whole code/process described above. On the int main is where the most OpenMpi commands are being used to start the MPI runtime and create multiple processes, then the rank(id of the process) and size(number of MPI processes) are being set. Later I used simple code to print messages for debugging reasons in order to have a confirmation that my code runs smoothly and properly. After the graph loads the Broadcasting I mentioned earlier follows and the workload is being shared to all the ranks which allocate memory BEFORE receiving their workload for safety reasons. After this point the MPI is responsible mostly for the data sharing/exchanging and synchronisation of the processes.

Check the code!There will be commends next from the commands or even above whole coding blocks to explain every little detail!

Experiment analysis

1.File .sh

The chosen partition was rome since it was the only one capable to serve our resources needed. I ran multiple experiments changing the number or nodes,cpus per task,threads and memory which led me to the conclusion that there is a threshold, especially on the nodes. Depending on which graph I was loading I had to change my parameters to find out the best times possible but also have enough memory to run my code which was a very crucial part of the experiments. To be precise 32Gb were used for Friendster and 64Gb for Mawi.

Limitations faced

Number of nodes=5
Number of tasks=24

Friendster Observations

One of the 2 graphs that will be studied is com-Friendster which I had managed to run in the previous assignment but now, for the first time, I received huge improvements on the run time. First a few words about the experiments. After running multiple tests I have come to the conclusion that com-Friendster runs the fastest on a small number of nodes, a somewhat small number of mpi tasks but a high level of cpus-per-task, in other words threads. For the fixed number of 2 nodes we have:

Running Time	Resources
4.45s	(tasks per node=2,cpus-per-task=20)
5.10s	(tasks per node=2,cpus-per-task=8)
4.45s	(tasks per node=2,cpus-per-task=16)
4.43s	(tasks per node=2,cpus-per-task=22)
3.41s	(task per node=1,cpus per task=22)
7.20s	(tasks per node=4,cpus-per-task=20)
7.75s	(tasks per node=4,cpus per task =16)
8.07s	(tasks per node=4,cpus per task =8)

Observations and comparisons

With this implementation I managed to get the best running time of 3.41seconds for com-Friendster which compared to my sequel program of the last assignment has a 47.91% improvement and a 1.92 speedup! Compared to the OpenMp best run we get a 54.05% improvement(due to the lack of OpenMpi the com-Friendster was actually running slower that its sequential implementation) and a speedup 2.17 speedup!

On the best run

2 nodes * 1 task per node = 2 tasks
2 tasks * 22 Cpus = 44 Cpu cores!

Mawi Observations

Another huge improvement can be seen on the mawi implementation since the code's architecture changed now that I have the proper memory and computing power to run it efficiently with the use of OpenMp. When it comes to mawi we notice that for a large number of nodes the run time is too slow (100+ seconds) therefor we will focus on the run times for the fixed number of 2 nodes.

Running Time	Resources
75.70s	(tasks per node=2,cpus-per-task=30)
74.60s	(tasks per node=2,cpus-per-task=20)
73.93s	(tasks per node=2,cpus-per-task=10)
68.40s	(task per node=1,cpus per task=20)
65.32s	(node=1,task per node=2,cpus per task=20)
92.95s	(tasks per node=4,cpus-per-task=30)
92.04s	(tasks per node=4,cpus per task =20)
90.10s	(tasks per node=4,cpus per task =10)

Observations and comparisons

We can obviously notice that the code runs significantly slower for a big number of tasks per node and a big number of cpus per task. Actually we can clearly see that for the combination of the minimum number of nodes and tasks per node but a big number of cpus per task makes the best run time.In fact, compared to the sequential run of mawi I have a 77.23% improvement and a 4.43 speedup! Now when it comes to the OpenMp implementation we get a 73.55% improvement and a 3.78 speedup!

On the best run

1 node * 2 task per node = 2 tasks
2 tasks * 20 cpus per tasks = 40 Cpu cores!

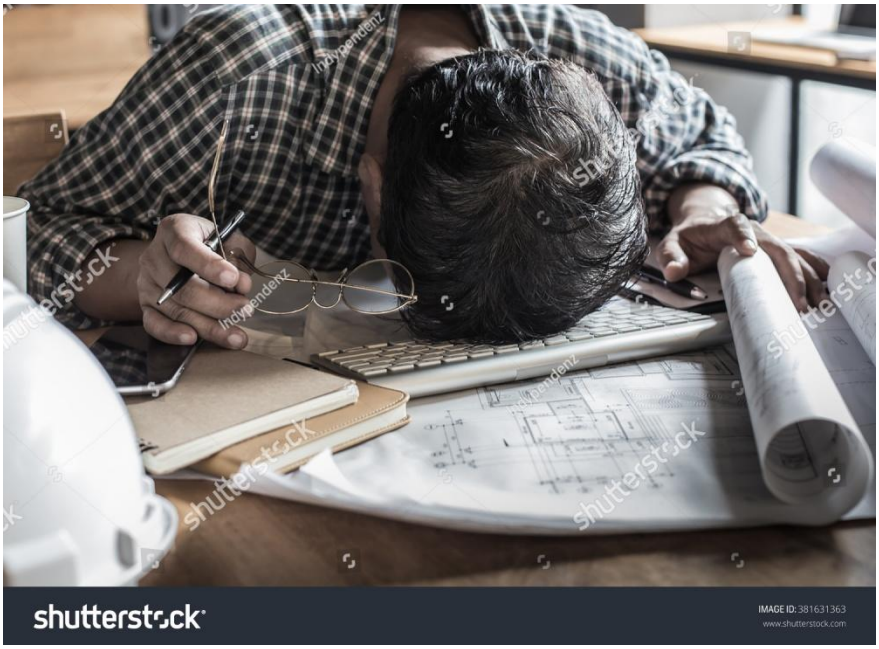
Bonus

Actually if we keep the nodes=1 and tasks=1 or 2 and increase the number of cpus per tasks we get an even better running time!

Since I have a whole page left….
To whoever is reading that…

Merry Christmas and Happy holidays<3

Jane&Jo



Us with all these assignments during Christmas^