STUDENTS: Gretchen Bonilla Caraballo - Jhony Lucia Huallparimachi Garcia

PROPOSAL

TESTING OF MST BENCHMARK WITH DIFFERENT INFRAESTRUCTURES

TECHNICAL APPROACH:

In the field of computation there are certain algorithms that are often used. One such case is the Minimum Spanning Tree algorithm (MST), whose purpose is to find a path towards all the nodes in the shortest path possible. For our project we shall run a MST algorithm in parallel using a variant of Kruskal, this algorithm was obtained from the "Problem Based Benchmark Suite"[1] [2]. We shall execute MST algorithm in different computer architectures.

Our goals for this project is to determine in which computer architecture gives better performance when running the MST algorithm with different data sizes in parallel using OpenMP. To accomplish this we will use three architectures: Ubuntu 16.04, CentOS7, and CentOS7 on Docker. Once done we shall verify the correctness of the parallel execution using a Check file. Then we will calculate the speedup and efficiency obtained from each architecture, afterwards, we compare the performance between them.

PERFORMANCE METRICS:

SPEEDUP:

The parameters considered to calculate speedup are:

- T_s = the sequential time of the Kruskal algorithm
- T_p = the parallel execution time with OpenMP of the Kruskal algorithm

$$Speedup = \frac{T_s}{Tp}$$

EFFICIENCY:

The efficiency for our case is calculated with sequential and parallel time for two processors (p = 2), where each processor has 12 cores and 24 threads.

$$Efficiency = \frac{T_s}{pT_p}$$

CORRECTNESS:

When the MST executable is generated it also automatically generates an executable check file that can be used to verify the correctness of the results obtained by MST. If the results are correct the check returns nothing, otherwise it throws an error message. The format for executing the check file is as follows:

$$>> ./MSTCheck \ < input > \ < output >$$

The <input> refers to the path to the input file that was used for the execution of the MST benchmark and <output> refers to the path where the benchmark wrote its results. The way the check works is that it sequentially generates its own MST results using the same input that the benchmark had used and compares its results with the output received.

EXPERIMENTAL SETTING:

Our data are weighted graphs. These graphs are generated within the benchmark's folders. To create the data we had to go the **/GraphData/data** folders and once inside we executed the "**make**" command. For "make" to work we had to specify which type of graph data to generate. The three options available were: randLocalGraph_WE_5_%, 2Dgrid_WE_% and rMatGraph_WE_%, where '**%**' was replaced with the amount of nodes to create. For our project we used randLocalGraph and started at 10,000,000 nodes and continued generating data in 10,000,000 nodes increments until reaching the final value of 100,000,000.

Here is an example of how the make command was used to create a weighted graph of 10,000,000 nodes:

$$>> make\ randLocalGraph\_WE\_5\_10000000$$

Once the data is created the next step is to go to the **/parallelKrusckal** folder, where all the files necessary for executing the benchmark reside. There we also use the “make” command to generate the MST executable. For our project we want to test the performance for single threaded (sequential) and in multi-threaded (parallel). The way this benchmark is designed, it knows which libraries to use for parallelization by checking on the system variables of the operating system (OS). If no system variable is found it uses the gcc compiler and is single threaded.

Initially when added to each architecture they did not have any of the system variables assigned causing the default for the benchmark to be sequential. When creating the executable for the sequential test only “make” was used to generate the MST executable. In the case of testing with OpenMP we had to add the variable to the system and then use “make”. Once “make” finished the MST executable was available and testing could begin. These were the commands used in the case of OpenMP:

```
>> export OPENMP = True
>> make
```

The final testing environment consisted of 10 test cases with data ranging from 10,000,000 to 100,000,000. The hardware for all systems was the same the difference is in the OS being used and the Centos7 running in the Docker container. For all multi-threaded cases the number of threads was set at 48 threads.

INFRASTRUCTURES:

System Specs - Ubuntu				
OS	**CPU's**	**Cores**	**Thread count**	**Processor**
Ubuntu 16.04	2 CPU	24 cores	48 threads	Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz

System Specs - CentOS				
OS	**CPU's**	**Cores**	**Thread count**	**Processor**
CentOS7	2 CPU	24 cores	48 threads	Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz

System Specs - CentOs Docker					
OS	**Host CPU's**	**Host Cores**	**Hosts Thread count**	**Processor**	**Host OS**
CentOS7 on Docker	2 CPU	24 cores	48 threads	Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz	Ubuntu 14.06

PROGRAMMING MODEL

In the case **Ubuntu** and **CentOS7** we used a low-level programing model, since we only relied on the native hardware the OS was running one. They both had the same hardware specifications, as can be seen in the **Infrastructure** section. Both had 2 CPUs with 12 core each and each core had 2 threads. This gives us 48 threads total to work with.

On the other hand, **CentOS7 on Docker** used a high-level programing model. CentOS7 will be running inside a container and not directly on top of the hardware as the other two architectures. The host OS used was Ubuntu 14.06 and the hardware specifications for the host were the same as the previous systems. The way this works the OS in the container shares the user space and kernel with other containers and get access to the hardware thought the host's kernel. Figure 1 presents the hierarchy of the system. Ubuntu as the host is at the base of the system, then docker goes on top of it providing and managing the containers and CentOS7 is a container running on top of docker.

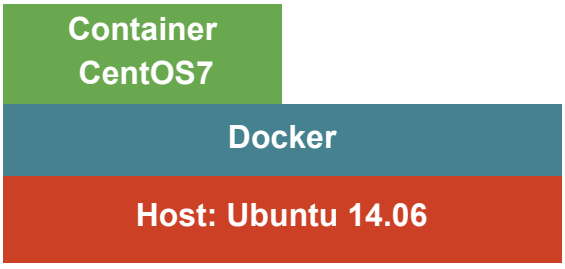

Figure 1: Hierarchy of the architecture with CentOS7 on Docker

For the parallel execution of the code we used the OpenMP programming interface since we can optimize the use of the resources offered by each architecture we select, for our experimentation we used all 48 threads.

EXPERIMENTAL RESULTS:

To execute the MST algorithm we used the following command:

$$>> ./MST\ [-r\ <roundnum>]\ -o\ <output>\ \ <input>$$

The <output> is the name of the output file where the resulting MST will be stored and <input> is the path to one of the previously generated data files. The -r flag indicates rounds and as a parameter receives the number of round that we want to execute the code. By rounds it means number of times it will do the MST to the same data. For example if we set "**-r 2**" then then it will run MST twice and provide us with the run time of each round separately.

In the following figures and tables we present the data we obtained after executing the MST benchmark.

Parallel			
	Time		
Size	Ubuntu	CentOs7	CentOS7 on Docker
10,000,000	3.775	3.1125	3.3325
20,000,000	7.155	6.595	6.6825
30,000,000	12.075	11.8	10.725
40,000,000	14.75	13.475	13.175
50,000,000	19.5	18.35	17.725
60,000,000	24.025	22.625	22.45
70,000,000	27.375	25.8	27.2
80,000,000	30.225	29.3	30.15
90,000,000	35.25	33.975	32.75
100,000,000	40.275	37.775	38.05

Table 1: Results of executing MST in parallel with OpenMP.

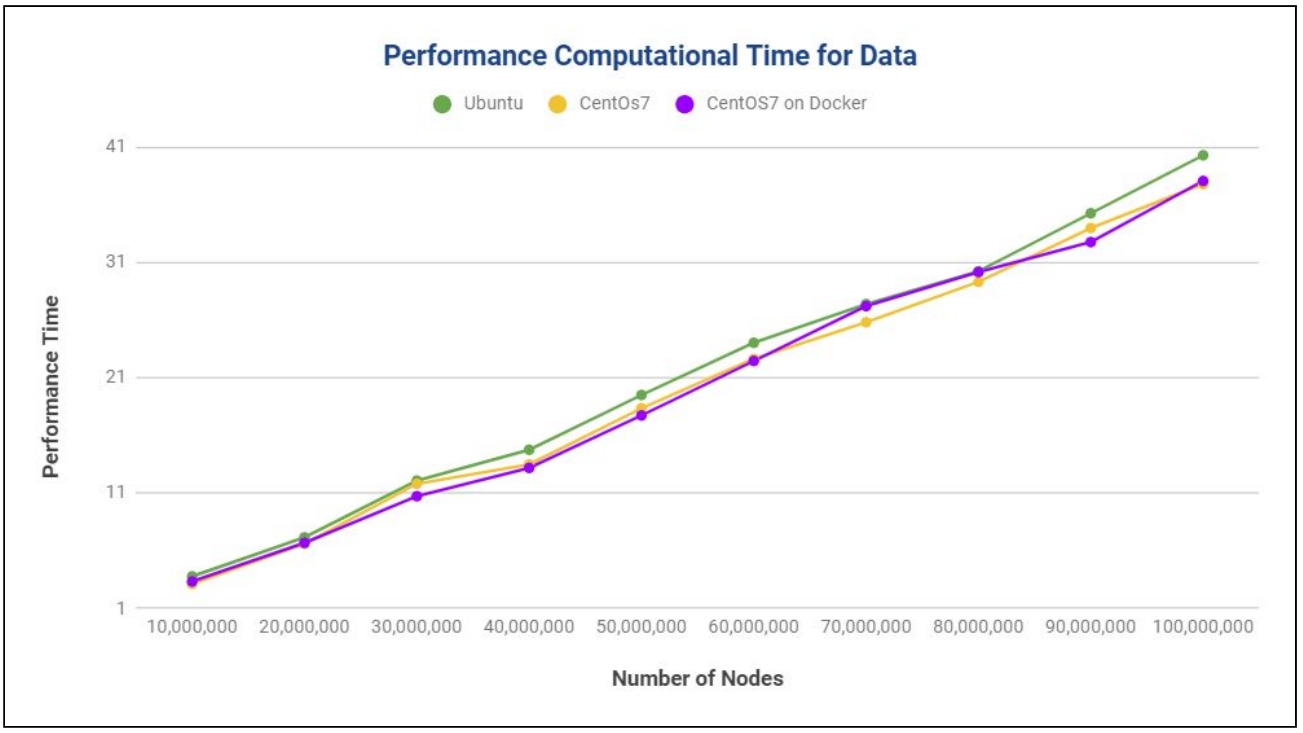

Figure 2: Graph of results obtained after executing MST in parallel with OpenMP.

Figure 2 shows the times obtained for each architecture according to the data size shown in Table 1. As can be seen for most of the data the performance is better with CentOS on Docker, although it has a small dip at 70,000,000 and 80,000,000. The lowest performance was obtained from Ubuntu. CentOS7 always does better than Ubuntu and in certain values (70,000,000 and 80,000,000) does better than CentOS7 on Docker.

Sequential			
	Time		
Size	Ubuntu	CentOs7	CentOS7 on Docker
10,000,000	8.18	7.10	7.0125
20,000,000	17.875	14.90	14.8
30,000,000	29.975	24.03	29.875
40,000,000	39.05	30.83	35.1
50,000,000	52.175	51.70	50.8
60,000,000	72.825	71.67	60.575
70,000,000	81.1	79.13	60.05
80,000,000	93.525	109.67	74.825
90,000,000	108	117.00	93.025
100,000,000	125	126.00	83.45

Table 2: Results of executing MST sequentially.

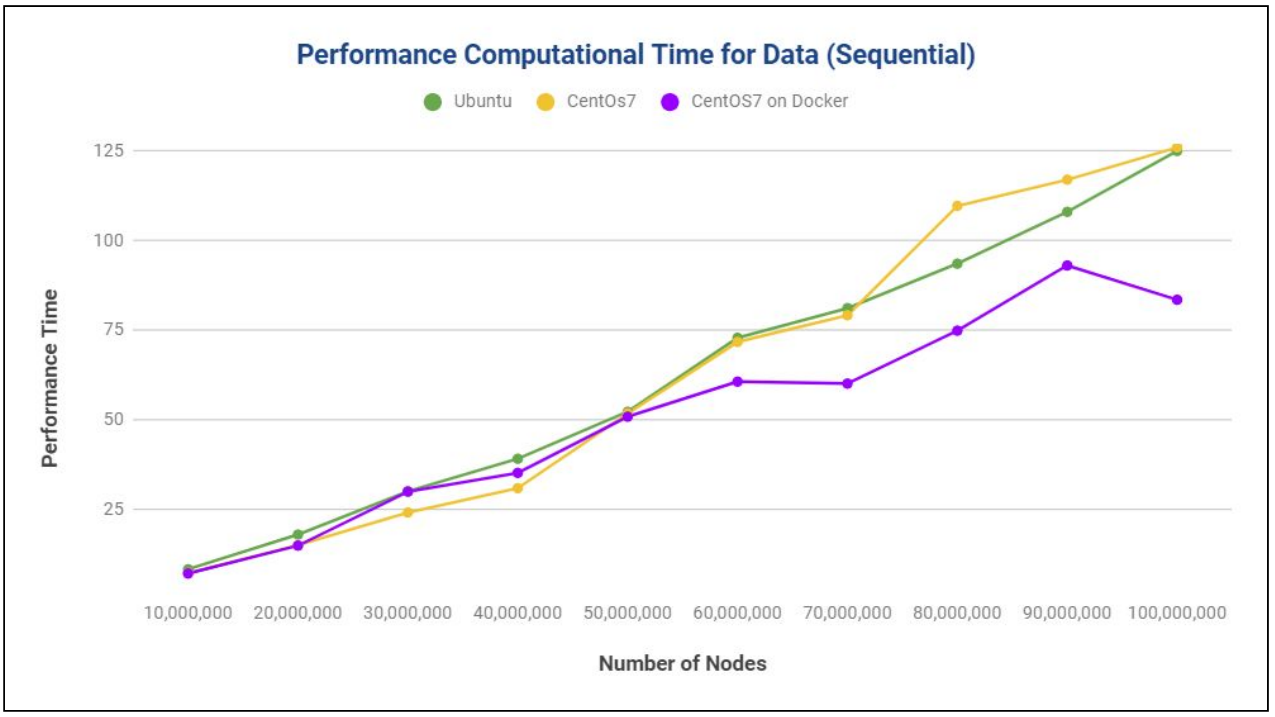

Figure 3: Graph obtained after executing MST sequentially.

Figure 3 shows the times obtained for each architecture according to the data sizes shown in Table 2. It can be seen that for data less than 50,000,000 the best performance is done by CentOS, and for more than 50,000,000 the best performance is obtained by CentOS Docker. Ubuntu had the poorest performance until 70,000,000 nodes, afterwards CentOS performs worse.

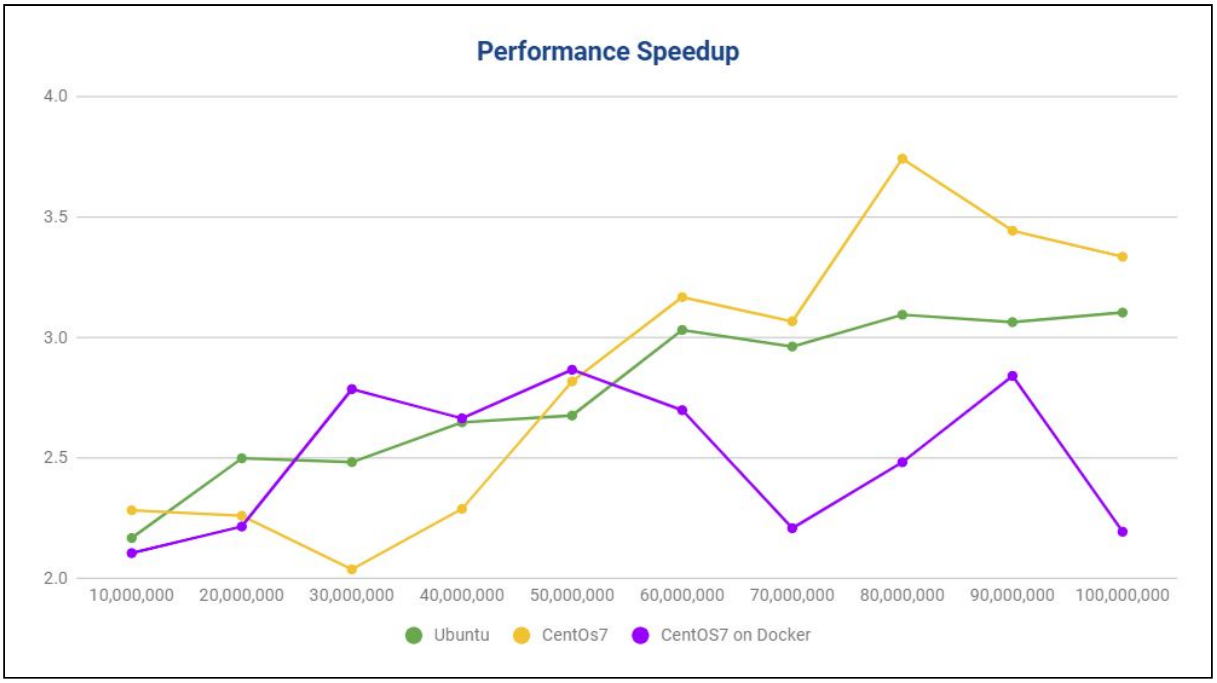

Figure 4: Graph of speedup obtained from each architecture .

Figure 4 demonstrates the speedups obtained. This was calculated using the formula presented in the Performance Metrics section. In the case of Ubuntu the speedup is mostly steadily increasing. Whereas CentOS7 and CentOS7 on Docker are a bit more volatile in their results. CentOS7 on Docker initially (for the first two step) has the worst speedup compared to the others, then its performance is above the rest until 50,000,000 nodes. After that its speedup is the worst of the three. For CentOS7 its speedup is the lowest from 30,000,000-50,000,000 and afterwards it does better than the rest. Ubuntu mostly just stay consistently between the other two.

This graph shows us that in the case of Ubuntu as the amount of data increases so does the performance gain for using parallel. In the case of CentOS7 on Docker we observe that its performance gain before 50,000,000 nodes stays below 2.5 and after that it stays above 3, but is full of up and downs. It does not stay constantly increasing or decreasing. On CentOS7 on Docker its speedup never reaches 2.5 and is volatile in its results since is it constantly increasing and decreasing.

This graph, opposed to the previous ones, shows that CentOS7 on Docker does worse. This is do to how this architecture did so much better in terms of sequential execution that when compared to the other systems its gain in performance is not as great. Ubuntu's performance in both Figure 2 and 3 are almost linear and this reflects in the speedup. CentOS7 had a few small dips and spikes in its performance in Figure 3 whereas in Figure 2 its behaviour is more linear. That would then translate to the dips and spikes seen in the speedup.

Figure 5: Graph of Efficiency obtained from each architecture.

Figure 5 presents the efficiency calculated for each system. The formula used was presented in the **Performance Metrics** section and the number of processors used was 2. The results obtained are similar to the ones in Figure 4. This is due to the efficiency formula resulting in being the speedup divided by 2.

CONCLUSIONS:

After calculating the minimum path with the Kruskal algorithm in parallel and sequential, we take the execution times for these architectures: Ubuntu, CentOS7 and CentOS7 in Docker. We observed the following behaviors: for the number of nodes greater than 50,000,000, the CentOS7 on Docker architecture obtains better performance, but its speedup and efficiency does not show a notable growth, however according to the results it is approximately 2 times better. In the case of CentOS7 it shows better performance and its speedup is approximately 3 times better for data greater than and equal to 50,000,000. In the case of Ubuntu its performance is not good, its speedup is about 3 times better from 70,000,000 of data.

We conclude that for our evaluation of Speedup and Efficiency the best architecture is CentOS7. In terms of performance CentOS7 on Docker does better.

REFERENCES:

[1]J. Shunt, G. Blelloch, J. Fineman, P. Gibbons, A. Kyrola, H. Vardhan Simhadri and K. Tangwongsan, "Brief Announcement: The Problem Based Benchmark Suite", Cs.cmu.edu, 2014. [Online]. Available: http://www.cs.cmu.edu/~guyb/papers/SBFG12.pdf. [Accessed: 19- Sep- 2018].

[2]G. Blelloch, J. Fineman, P. Gibbons, J. Shunt, "Internally Deterministic Parallel Algorithms Can Be Fats", Cs.cmu.edu, 2014. [Online]. Available: http://www.cs.cmu.edu/~guyb/papers/BFGS12.pdf. [Accessed: 19- Sep- 2018].