ECE 420

Lab 4 Report

Lab Section: H41

March 31st, 2017

By: Quentin Lautischer and Raghav Vamaraju

Description of Implementation:

Our serial implementation of PageRank was based on the "serialtester.c" code provided on eClass. We then created an MPI program that would parallelize using separate processes. The code for our MPI PageRank Implementation can be found in "main.c". The details are as follows:

1. From a terminal, the command: m*pirun –np <num of procs> ./main*, will spawn several processes each running our MPI program. One process of which is the master process.
2. Each process begins by instantiating the startup components: MPI_INIT, MPI_Comm_size, MPI_Comm_rank.
3. Then each process generates the data structure *nodehead* from the data_input file. As a side note, we could have had only the master process generate this structure and share it with the remaining processes, but we chose not to do this, as we felt that it would be more efficient for each process to create it.
4. Now that the data structure exists, we can use it to solve our PageRank problem. An array R (for Rank) is malloc'd and the processes split the task of initializing the data contained within the structure. This is done by splitting the iterations over the indices of R by the number of processes. MPI_Scatter and MPI_Gather were used.
5. Now the Core Calculation is performed. The master processes determines if the relative error between Rank and the last snapshot of Rank is greater than our EPSILON, and the master process sends a broadcast to all other processes informing them that there is more computations to be done. The computation is split between all processes. The computation consists of the following code block:

```
while(workMore) {
    interation_count++;
    printf("Process %d starting work round: %d\n", rank, interation_count);
    MPI_Scatter(R, nodes_per_proc, MPI_DOUBLE, localR, nodes_per_proc, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(r_pre, nodecount, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    for ( i = (nodes_per_proc*rank); i < (nodes_per_proc*(rank+1)); i++){
        ii = i-(nodes_per_proc*rank);
        localR[ii] = 0;
        for ( j = 0; j < nodehead[i].num_in_links; j++)
            localR[ii] += r_pre[nodehead[i].inlinks[j]] / out_links[nodehead[i].inlinks[j]];
        localR[ii] *= DAMPING_FACTOR;
        localR[ii] += damp_const;
    }
    MPI_Gather(localR, nodes_per_proc, MPI_DOUBLE, R, nodes_per_proc, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    printf("Process %d finished work round: %d\n", rank, interation_count);
```

6. Once the condition for workMore is broken (the computation is finished), a final broadcast is sent to all processes informing them to stop.
7. The master process will then report timing and generate an output of the data.

Testing and Verification:

We tested and verified our program using the ***serialtester*** and ***check.sh*** that was supplied in the Lab4 development kit. We used the data provided in the lab kit. We verified that the solution produced by our code was within the error range as indicated by the *serialtester*. We found that our implementation consistently produced a negligible error amount, as well as a significant speedup. We also used multiple "printf()" statements to ensure that the parallelizations were running as expected and correctly (in an expected sequence). We found that the provided *serialtester* program and our simple "printf()" debug statements were enough to verify correctness of the program.

Performance Discussion:

Performance greatly increased by using 8 processes on a single CPU. Running our program on a cluster computer had a slowdown, and this is most likely due to the trivial size of our input_data in contrast to the overhead required to organize and coordinate several machines together. If the problem size required much more than 8 processes (for example, 100+ processes), a cluster would have been much more efficient than the single machine solution.

The following data was collected by running our program using various MPI settings. We varied the number of processes and number of hosts, and ran 10 iterations per setting. A summary of the data and a graph is at the bottom of this series of charts.

MPI Cluster: 8 Processes (-np 8) over 8 Hosts										
#	1	2	3	4	5	6	7	8	9	10
Time (s)	0.782	0.762	0.855	0.814	0.795	2.251	0.442	0.430	0.876	0.425
Avg (s)	0.843									
Median (s)	0.789									

MPI Cluster: 4 Processes (-np 4) over 4 Hosts										
#	1	2	3	4	5	6	7	8	9	10
Time (s)	0.318	0.331	0.333	0.381	0.322	0.319	0.333	0.332	0.323	0.332
Avg (s)	0.332									
Median (s)	0.332									

MPI Cluster: 8 Processes (-np 8) over 4 Hosts										
#	1	2	3	4	5	6	7	8	9	10
Time (s)	0.295	0.547	0.750	0.316	0.311	0.292	0.353	0.294	0.291	0.292
Avg (s)	0.374									
Median (s)	0.303									

MPI Cluster: 8 Processes (-np 8) over 2 Hosts										
#	1	2	3	4	5	6	7	8	9	10
Time (s)	0.167	0.170	0.173	0.161	0.168	0.175	0.214	0.211	0.160	0.166
Avg (s)	0.177									
Median (s)	0.169									

MPI Cluster: 4 Processes (-np 4) over 2 Hosts										
#	1	2	3	4	5	6	7	8	9	10
Time (s)	0.174	0.176	0.179	0.174	0.175	0.176	0.176	0.178	0.173	0.175
Avg (s)	0.176									
Median (s)	0.176									

MPI Cluster: 2 Processes (-np 2) over 2 Hosts										
#	1	2	3	4	5	6	7	8	9	10
Time (s)	0.167	0.166	0.169	0.168	0.169	0.167	0.169	0.169	0.170	0.168
Avg (s)	0.168									
Median (s)	0.169									

MPI Cluster: 1 Process (-np 1) over 2 Hosts										
#	1	2	3	4	5	6	7	8	9	10
Time (s)	0.015	0.015	0.016	0.015	0.018	0.017	0.015	0.019	0.015	0.014
Avg (s)	0.016									
Median (s)	0.015									

MPI Cluster: 1 Process (-np 1) over 1 Host										
#	1	2	3	4	5	6	7	8	9	10
Time (s)	0.015	0.016	0.017	0.015	0.015	0.017	0.013	0.014	0.019	0.014
Avg (s)	0.016									
Median (s)	0.015									

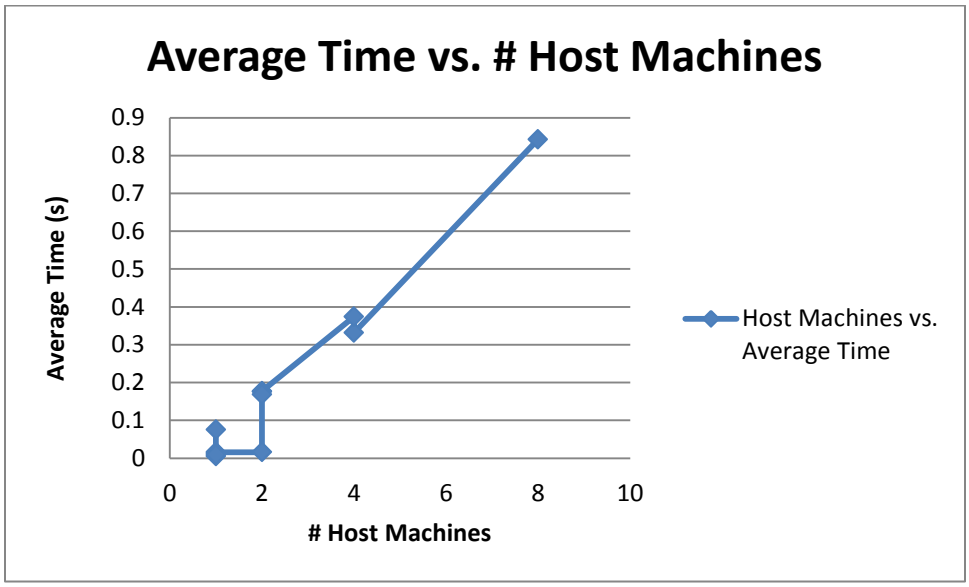
MPI Single Machine: 8 Processes (-np 8)										
#	1	2	3	4	5	6	7	8	9	10
Time (s)	0.0048	0.0048	0.0048	0.0045	0.0048	0.0047	0.0058	0.0047	0.0049	0.0044
Avg (s)	0.00488									
Median (s)	0.0048									

MPI Single Machine: 4 Processes (-np 4)										
#	1	2	3	4	5	6	7	8	9	10
Time (s)	0.0191	0.0048	0.0640	0.0046	0.0050	0.0119	0.0048	0.0046	0.0046	0.0048
Avg (s)	0.01285									
Median (s)	0.0049									

MPI Single Machine: 2 Processes (-np 2)										
#	1	2	3	4	5	6	7	8	9	10
Time (s)	0.0082	0.0083	0.0081	0.0084	0.0083	0.0080	0.0082	0.0080	0.0082	0.0083
Avg (s)	0.008243									
Median (s)	0.0083									

Single Machine: Serial Implementation (1 Process), No MPI										
#	1	2	3	4	5	6	7	8	9	10
Time (s)	0.0757	0.0752	0.0755	0.0759	0.0753	0.0755	0.0759	0.0753	0.0757	0.0756
Avg (s)	0.07556									
Median (s)	0.0756									

Summary of Data		
# Processes	**# Host Machines**	**Average Time (s)**
8	8	0.843
4	4	0.332
8	4	0.374
8	2	0.177
4	2	0.176
2	2	0.168
1	2	0.016
1	1	0.016
8	1	0.00488
4	1	0.01285
2	1	0.008243
1	1	0.07556



From this data, the performance of the program was significantly better on a single machine, and the running time increased as cluster size increased. On a cluster, the performance was best when using 1-2 host machines and 1-2 processes, which is virtually equivalent to running a single machine. Adding more host machines consistently slowed down the program, due to overhead from communicating over the network. On a single machine, we found that increasing the number of processes up to 8 consistently increased the program speed, but above 8 processes, slowdown occurred. This is because the machine had 8 cores, and using 8 processes was optimal; anything above this resulted in too much context switching and interprocess communication.

Please note that a more detailed discussion of the program performance, the impact of certain design decisions, and the speedup from the number of processes, can be found in the Questions section of this report.

Questions:

1. Our MPI program was clearly better on a single machine than on multiple machines. In fact, the speed increased as the number of host machines decreased, and adding host machines almost linearly slowed down the program. This slowdown occurred because the overhead from machine-machine communication (via networking) was quite high in this particular problem, and increased with the number of host machines. Communication involves synchronization between processes/machines, which is even slower over a network, as compared to synchronization between processes on a single machine. Also, the dataset for this problem was relatively small, so the overhead from running on multiple machines was felt more strongly.

Very large problems with large datasets and repetitive computations would benefit from distributed programming. In general, problems that have a computation time significantly larger than the machine-machine communication overhead will benefit using multiple machines. This way, the communication overhead will not be felt as strongly, and will play an insignificant role in the overall timings. This is assuming the problem can be divided into smaller tasks.

2. We partitioned our data (nodes) equally between each process, and limited the program to 8 processes, to match the number of cores on the machine. Each process handled multiple nodes (1/8th of the total number of nodes). On a single machine, we found that 8 processes was the most efficient/optimal number of processes. Anything above this was resulting in slowdown due to communication overhead and context switching on the machine, since the number of processes was greater than the number of cores. In general, assuming the problem is divided properly, granularity will decrease the running time of the

program (make it run faster) until it reaches an optimal point, and then increase the running time (make it run slower) after this. This is because increasing the number of processes also increases the communication overhead, and after a certain point, this overhead dominates the running time of the program. Typically, the optimal number of processes is the number of cores on the machine, because this results in the least amount of context switching on individual cores, and process-process communication will be relatively minimal.

3. We used the MPI Gather/Scatter communication pattern, in conjunction MPI Broadcast (Bcast). The advantage of using Gather/Scatter is that the data can be evenly (and conveniently) split between processes, and the same calculations can be run over the smaller datasets. Since the calculations take approximately the same time for each process, they finish around the same time, and no process is blocking (waiting) for an extended period of time. The Broadcast pattern allows the "main process" to communicate a message to the other processes; in our code, we used it to signal to other processes to either continue computations, or exit. The advantage of this is that all processes will wait for the broadcast before continuing, and will not perform extra work. We also utilized broadcast for sharing a reference data structure from master process to the others.

Conclusion and Experience:

This lab demonstrated that parallelizing with multiple processes is much the same as parallelizing with multiple threads. The added convenience with parallel processes is the ability to scale far beyond the number of CPUs on a single machine and use the number of CPUs found within a cluster of computers. However, running on a cluster of computers has a large overhead and thus the problem size must fittingly large.