COL730 Parallel Programming
A3 Report

Harsh Sharma
2019MT60628

November 15, 2022

1 Introduction

In this assignment, we implemented PageRank algorithm in MapReduce paradigm of problem solving in three ways :

1. using mapreduce C++ library
2. implementing MapReduce from scratch using MPI
3. using mapreduce MPI library

2 MapReduce strategy in pagerank

In implementing mapreduce solution to pagerank, we took a master-slave approach where master which is the scheduler and co-ordinator directs its slaves (also knows as map or reduce workers) to do the necessary computations and give the result back to master, which then executes the pagerank algorithm.

Map-Worker takes the input as '`page1 page2`' and returns (`key = page2`, `value = page1`), where the input represents a link from `page1` to `page2`.

Input to Reduce-Worker would be of the form (`key = P`, `multival =` {`P1, P2, ...Pn`}) and it simply returns the list {`P1, P2, ...Pn`} back which is exactly all the pages that have a link to page `P`

Rationale for taking above mapreduce decision : pagerank algorithm computes its solution iteratively using the following equation

$$I(P_i) = \alpha \sum_{P_j \in B_i} \frac{I(P_j)}{l_j} + \frac{\alpha}{n} \sum_{P_j \in D} I(P_j) + \frac{1-\alpha}{n} \sum_j I(P_j)$$

where B_i is set of all pages that have a link to page P_i and D is set of all the dangling pages in the web, so the second and third term can be computed easily with a single scan for each page P_i, but computing the first sum each time can be costly as the web-matrix is generally very sparse. So we need to pre-determine B_i for each i to make pagerank run faster which is exactly what reduce-worker computes for `key = Pi` This strategy is used to implement pagerank algorithm using mapreduce in all the three parts

3 Implementation of MapReduce using MPI

This is second part of the assignment, high level strategy is that master-process directs the map workers to read the assigned input file/shard and do the map task to parse intermediate `key`, `value` pairs out of it. It then sends these `key`, `value` pairs to appropriate reduce-workers (according to partition function : `hash(key) % R`) using `MPI_Send`. The reduce workers do `MPI_Recv` to receive key-value pairs one by one and sort them w.r.t `key` and occurrences of same keys are grouped together (to get `(key, multi-value)`) and reduce function is called upon them (to produce `(key, value)`) . The result is then communicated back to the master process from the reduce workers. See `mapreduce` namespace in `mr-pr-mpi.cpp` file for more details

4 Instructions on how to use the library

1. Open the terminal and cd into the source code directory
2. Type make to compile all the source code
3. Step 2. creates 3 object files `mr-pr-cpp.o`, `mr-pr-mpi.o`, `mr-pr-mpi-base.o` and an executable `check` which checks the correctness of the implementation by verifying the outputs against corresponding python outputs (see tests)
4. You can use the shell script `run.sh` to run all the three implementations on a data-set provided in `/tests` directory by just providing the filename inside the `/tests` directory (without .txt extension) and output will be emitted in `/result` directory with appropriate name format

Eg: $ ./run.sh barabasi-20000

This command takes `tests/barabasi-20000.txt` file as input and outputs `result/barabasi-20000-pr-cpp.txt`, `result/barabasi-20000-pr-mpi.txt`, `result/barabasi-20000-pr-mpi-base.txt`. It also outputs some mapreduce/pgrank timing information and correctness check result on `stdout`

5. `/include` directory contains include files for mapreduce C++ (1st part) and `/mapreduce-7Apr14` directory contains include files for mapreduce MPI (3rd part)

5 Benchmarks

Since the page rank implementations are the same except for the map-reduce part, we'll compare the time taken by mapreduce part in the three implementations

All the results were ran on a Zorin OS (Linux Distro) with Intel® Core™ i5-8250U CPU @ 1.60GHz × 8, 15.5 GiB Memory. MPI implementations were spawned with 8 processes in all the benchmarks .

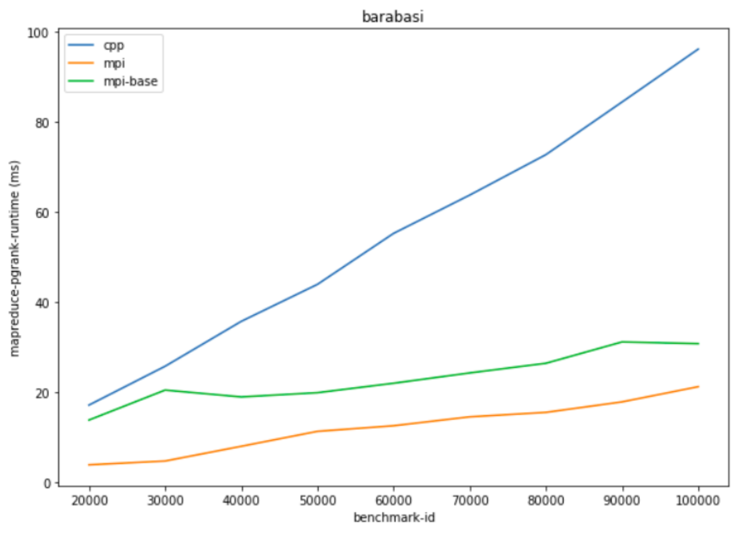

Figure 1: Benchmark result for barabasi-dataset

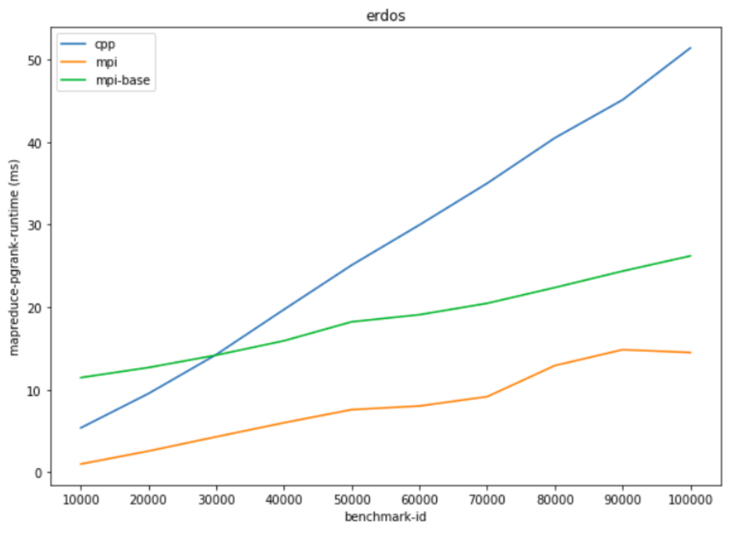

Figure 2: Benchmark result for erdos-dataset

Some more benchmark results (small datasets)			
Benchmark ID	cpp-runtime (μs)	mpi-runtime (μs)	mpi-base-runtime (μs)
bull	792.45	75.74	17622
chvatal	866.38	135.18	13522
coxeter	938.89	131.84	11211
cubical	851.36	320.36	14670
dodecahedral	872.16	167.41	16675

6 Observations

It is clear from the graphs that runtime of `mpi` is less than runtime of `mpi-base` which is less than runtime of `cpp` as you scale the problem size. This is because `mpi` and `mpi-base` are benefiting from the fact that parallel map/reduce computations are going on in different processes that together solve the problem. Also due to relatively small slope in both mpi implementations, it can be said that they scale really well with large problem size. `mpi` implementation seems to be faster than `mpi-base` this is because lower level MPI details were exposed to the programmer here, so I could use it to maximize performance of pagerank algorithm whereas `mpi-base` would have a lot of unnecessary details/data-structures that it had to manage to be as general implementation as possible, that could've slowed it down a bit.

For benchmark results on smaller data sets (websize of about 5 - 50), you can clearly observe `mpi-base` is extremely slow, that is because it is bottle-necked by initialization of its data structures/other parameters and thus you'll have to pay this extra cost initially. To be effective dataset size needs to be large for `mpi-base`.

In all, `mpi` implementation seems to work the best both on small datasets and large datasets, and it due to very small relative slope (or runtime growth with problem size) it seems to scale really well for large datasets

`cpp` implementation is fine for small datasets, but it is not scalable at all and runtime grows very quickly as you increase the problem size

`mpi-base` implementation is fine for large datasets and it also scales well with the problem-size.