COL380: Introduction to Parallel and Distributed Computing

MapReduce Using Google's PageRank Algorithm

using MPI

Vasu Jain: 2017CS10387 , **Shreya Sharma: 2017CS50493**

May 4, 2020

Contents

Design Philosophy, Objectives and Workflow

We approached this assignment with the following objectives and design philosophy:

1. Implement mapreduce-pagerank using mapreduce C++ library
2. Implement our own mapreduce library with MPI by implementing the functions needed for pagerank.
3. Implement pagerank using existing mapreduce MPI library. In each of the above cases compare correctness of the pagrank output against the given java/python outputs.
4. Plot graphs with x-axis as benchmark ID, and y-axis as pagerank runtime (three graphs for the three executables).
5. Compare pagerank latencies across the three implementations and comment on the observations

The time measurements were done by using the `std::chrono::high_resolution_clock` These objectives were achieved to a large extent by continuous evolution of the code-base. The design philosophy caused changes across the objectives in tandem. However, the general design cycle was

Optimize Serial $\longrightarrow$ Parallelize algorithm $\longrightarrow$ Refactor Code $\longrightarrow$ Optimize Serial $\longrightarrow$ ...

Our code and scripts can be found in the repository at:
`https://github.com/jainvasu631/MPI-MapReduce-PageRank`

The PageRank Algorithm

We will assign to each web page P a measure of its importance I(P), called the page's PageRank.
Suppose that page Pj has lj links. If one of those links is to page Pi, then Pj will pass on 1/lj of its importance to Pi. The importance ranking of Pi is then the sum of all the contributions made by pages linking to it. That is, if we denote the set of pages linking to Pi by Bi, then $I(P_i) = \sum \frac{I(P_j)}{l_j}, P_j \in B_i$

- **Hyperlink Matrix**
  H := [Hij] = if Pj in Bi then 1/lj else 0
  H := [Hij] = if Pj to Pi is Edge then 1/lj else 0
  H is a **stochastic matrix**. The sum of all entries in a column is 1/0
- **Importance Vector**
  I := [I(Pi)] whose components are the PageRanks or the importance rankings of all the pages.
  I = HI i.e. I is an eigenvector of H with eigenvalue 1. This is the **stationary vector** of H.
- **Computing I**
  H can be a very very large matrix of the order of 10^10. However most entries of H are zero.Therefore its a sparse matrix.
  I(k+1) = HI(k). Sequence of I(k) converges to I.

Problems with Convergence of I

1. Dangling nodes which have no out links. Then H is no more perfectly stochastic. In this case these nodes will act as importance sinks.
2. Circular Reference Problem. S isn't Primitive anymore and the I(k) never converges.
3. Importance Sink. S is reducible, i.e. when we have a sub graph of fully connected nodes with no edge coming out of it.

Solving the Problem

Google Matrix

Replace all 0 columns in matrix H with 1/n to create S. Thus we don't have dangling nodes anymore.
So, S = H + A where A := [Aij] = if Hi==0 then 1/n else 0

Choose a parameter α between 0 and 1. Now, with probability α , the random surfer is guided by S and with probability $1 - \alpha$, he chooses the next page at random. This solves the problem of importance sinks and circular references.
So, G := α x S + $(1 - \alpha)$ x 1/n x 1.

Choosing alpha

The parameter α has an important role. When α is 1 we get G:=S and when α is 0 G:=$(1 - \alpha)$/n x 1. The rate of convergence of I depends on α. Therefore as a compromise we use $\alpha = 0.85$.

Optimizing the Calculation

Simplifying Formula

I(k+1) = α x H x I(k) + α x A x I(k) + $(1 - \alpha)$/n x 1 x I(k).
This can be further simplified. A = J/n where J = [Ji] = 1 if Corresponding Column is 0 else 0.
Let β := (1/α-1) and so M:= (J + β x 1)/n.
Then I(k+1) = α x (H x I(k) + (J + (1/α-1) x 1)/n x I(k)) which simplifies to
I(k+1) = α (H I(k) + M I(k)).

Calculating M x I(k)

As all rows of M are identical, we can write M = 1 x N and M x I(k) = 1 x N x I(k).
factor:= N x I(k) = (1 + beta if Corresponding column is 0 else beta) x I(k)/n
Now the expression simplifies to I(k+1) = α x H x I(k) + 1 x factor.

Data Structures

Graphs

Input files contain two space separated numbers (denoting 2 web-pages) in each line indicating a link from the first webpage to the second. We read and convert the input into a graph with of nodes (N) = maximum node index in the file.
Class Graph contains -

- VertexList (type vector(int))- Protected variable denoting a list of vertices (type int).
- EdgeList (type vector(pair(int,int))) - Protected variable denoting list of edges (type pair(int,int)).
- toList (type vector(VertexList)) - Public variable denoting a list such that toList[i] is the VertexList of all the vertices which have an edge from "i".
- fromList (type vector(VertexList)) - Public variable denoting a list such that fromList[i] is the VertexList of all the vertices which have an edge to "i".

Columns

Columns is a vector of Values where Values has type - "double". Two important uses of Columns are -

- PageRank - This data structure simply contains the calculated page rank for each web-page.
- Hyperlink - ith element of Hyperlink contains the importance ith node will give to the nodes it has an outgoing edge to.

Initialization

After the graph is generated from the input file:

Listing 1: Calculating Hyperlink

```
static Column calculateHyperLinkColumn(const Graph::ToList& toList){
   const Graph::Size N = toList.size();
   Column hyperlink(N);
   for(Graph::Size i=0;i<N;i++)
      hyperlink[i] = (toList[i].size()>0)? 1.0/toList[i].size() : 0;
   return hyperlink;
}
```

PageRank is initialised as a vector of doubles of size N with all values = 1.0/N

Listing 2: Initialising PageRank

```
static inline Column getInitPageRank(const Graph::Size N) {
   return Column(N,1.0/N);
}
```

Mapreduce C++ Library

The MapReduce C++ Library implements a single-machine platform for programming using the the Google MapReduce idiom. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key.
The developer is required to write two classes:

- MapTask - It implements a mapping function to process key/value pairs generate a set of intermediate key/value pairs
- ReduceTask - It implements a reduce function to merges all intermediate values associated with the same intermediate key.

There are three optional template parameters that can be used to modify the default implementation:

- Datasource - It implements a mechanism to feed data to the Map Tasks - on request of the MapReduce library
- Combine - It can be used to partially consolidate results of the Map Task before they are passed to the Reduce Tasks
- IntermediateStore - It handles storage, merging and sorting of intermediate results between the Map and Reduce phases

Our Implementation

We have used the concept of Mapreduce while solving the pagerank finding problem at two steps. First is in finding the factor Column (as defined in the pagerank algorithm section) and second is in finding the pagerank Column.

Calculating Factor using Mapreduce

Listing 3: MapTask for Factor

```
class MapFactor : public mapreduce::map_task<Graph::Vertex, Value>{
   // Identity Map
   public: template<typename Runtime>
      void operator()(Runtime& runtime, const key_type& key, const value_type&
          value) const{runtime.emit_intermediate(COMMON,value);}
   private: static constexpr Graph::Size COMMON = 0;
};
```

Listing 4: ReduceTask for Factor

```
class ReduceFactor : public mapreduce::reduce_task<Graph::Size, Value>{
   // Calculate Factor by Accumulation
   public: template<typename Runtime, typename Iterator>
      void operator()(Runtime& runtime, const key_type& key, Iterator it,
          Iterator end) const{runtime.emit(key,std::accumulate(it,end,0.0));}
};
```

We have defined a class named FactorData used by "Calculation" - an instance of mapreduce::job to compute factor column using mapreduce template of C++.

Listing 5: Mapreduce Job for Factor

```
using Calculation = mapreduce::job<MapFactor,ReduceFactor,
    mapreduce::null_combiner,FactorData>;
```

Calculating PageRank using Mapreduce

Listing 6: MapTask for PageRank

```
class MapPageRank : public mapreduce::map_task<Graph::Vertex, VertexInfo >{
   // Function to generate Intermediate Values
   // Will map for each to Vertex a probability contribution that is summed by
       reduce
   public: template<typename Runtime>
      void operator()(Runtime& runtime, const key_type& key, const value_type&
          value) const{
         runtime.emit_intermediate(key, ZERO); // Emitting Zero Insures that
             each Node has atleast Tuple
         for (const Graph::Vertex& to : value.second)
            runtime.emit_intermediate(to, value.first);
      }
   private: static constexpr Value ZERO = 0.0;

};
```

Listing 7: ReduceTask for PageRank

```
class ReducePageRank : public mapreduce::reduce_task<Graph::Vertex, Value>{
   // Function to add all probabilities in Hyperlink Matrix
   public: template<typename Runtime, typename Iterator>
      void operator()(Runtime& runtime, const key_type& key, Iterator it,
          Iterator end) const{runtime.emit(key,std::accumulate(it,end,0.0));}
};
```

We have defined a class named PageRankData used by "Iteration" - an instance of mapreduce::job to compute pageRank column using mapreduce template of C++.

Listing 8: Mapreduce Job for PageRank

```
using Iteration = mapreduce::job<MapPageRank, ReducePageRank,
    mapreduce::null_combiner,PageRankData>;
```

Mapreduce MPI Library

In terms of Principle

In C++, your program includes two library header files and uses the MapReduce namespace:

include "mapreduce.h"
include "keyvalue.h"
using namespace MAPREDUCE_NS

Arguments to the library's **map()** and **reduce()** methods include function pointers to serial "mymap" and "myreduce" functions in your code (named anything you wish), which will be "called back to" from the library as it performs the parallel map and reduce operations.

A typical simple MapReduce program involves these steps:

Listing 9: MapReduce Program Layout

```
MapReduce *mr = new MapReduce(MPI_COMM_WORLD); // instantiate an MR object
mr->map(nfiles,&mymap); // parallel map
mr->collate() // collate keys
mr->reduce(&myreduce); // parallel reduce
delete mr; // delete the MR object
```

All the library methods operate on two basic data structures stored within the MapReduce object -

- **KeyValue object (KV)** - A KV is a collection of key/value pairs. The same key may appear many times in the collection, associated with values which may or may not be the same.
- **KeyMultiValue object (KMV)** - A KMV is also a collection of key/value pairs. But each key in the KMV is unique, meaning it appears exactly once.The value associated with a KMV key is a concatenated list (a multi-value) of all the values associated with the same key in the original KV.

Various library methods operated on KV and KMV objects in our implementation are -

- **MapReduce add() method** - *uint64_t MapReduce::add(MapReduce *mr2)*
  Adds the KeyValue pairs contained in a second MapReduce object mr2, to the KeyValue object of the first MapReduce object, which is created if one does not exist.
- **MapReduce broadcast() method** - *uint64_t MapReduce::broadcast(int root)*
  Deletes the key/value pairs of a KeyValue object on all processors except root, and then broadcasts the key/value pairs owned by the root processor to all the other processors.
- **MapReduce collate() method** - *uint64_t MapReduce::collate(int (*myhash)(char *, int))*
  Aggregates a KeyValue object across processors and converts it into a KeyMultiValue object.
- **MapReduce gather() method** - *uint64_t MapReduce::gather(int nprocs)*
  Collects the key/value pairs of a KeyValue object spread across all processors to form a new KeyValue object on a subset (nprocs) of processors.
- **MapReduce sort_keys() method** - *uint64_t MapReduce::sort_keys(int flag)*
  Sorts a KeyValue object by its keys to produce a new KeyValue object.

In terms of Implementation

The concept of approach is completely similar to the one used in the first part with MapReduce C++ Library. We calculate the factor column and then the pagerank column using MapReduce API of MPI.

Calculating Factor using Mapreduce of MPI

In **"MPI Base/Calculator.cpp"** We have defined function -

- MapFactor - *static void MapFactor(Graph::Vertex key, KeyValue* keyvalue, void* calculator)* as "mymap" function
- ReduceFactor - *static void ReduceFactor(char* key, int keybytes, char* multivalue, int nvalues, int* valuebytes, KeyValue* keyvalue, void* calculator)* as "myreduce" function.
- GatherFactor - *static void GatherFactor(uint64_t index, char* key, int keybytes, char* value, int valuebytes, KeyValue* keyvalue, void* calculator)* as "mygather" function.

Listing 10: Calculating Factor

```
auto kvPairs = mapreduce.map(N,MapFactor,void_this);// Map Part
mapreduce.collate(NULL);
auto kvmPairs = mapreduce.reduce(ReduceFactor,void_this);// Reduce Part
mapreduce.gather(HOME);
mapreduce.broadcast(ROOT);
kvPairs = mapreduce.map(&mapreduce,GatherFactor,void_this);// Gather Part
```

Calculating Pagerank using Mapreduce of MPI

In **"MPI Base/Calculator.cpp"** We have defined function -

- MapPageRank - *static void MapPageRank(Graph::Vertex key, KeyValue* keyvalue, void* calculator)* as "mymap" function
- ReducePageRank - *static void ReducePageRank(char* key, int keybytes, char* multivalue, int nvalues, int* valuebytes, KeyValue* keyvalue, void* calculator)* as "myreduce" function.
- GatherPageRank - *static void GatherPageRank(uint64_t index,char* key, int keybytes, char* value, int valuebytes, KeyValue* keyvalue, void* calculator)* as "mygather" function.

Listing 11: Calculating PageRank

```
auto kvPairs = mapreduce.map(N,MapPageRank,void_this);// Map Part
mapreduce.collate(NULL);
auto kvmPairs = mapreduce.reduce(ReducePageRank,void_this);// Reduce Part
mapreduce.gather(HOME);
mapreduce.sort_keys(INT_SORT);
mapreduce.broadcast(ROOT);
kvPairs = mapreduce.map(&mapreduce,GatherPageRank,void_this);// Gather Part
```

Self-implemented Mapreduce Library using MPI

Here we have implemented the methods of the MPI Library for MapReduce like "map", "reduce" in the template form similar to C++ MapReduce Library.

Listing 12: MapReduce Template

```
template<typename MapKey, typename MapValue, typename ReduceKey, typename
    ReduceValue, typename ResultKey, typename ResultValue>
```

Listing 13: Job Template

```
template <typename MapTask, typename ReduceTask, typename Generator, typename
    Combiner, typename Distributor, typename Result>
```

General Execution Flow:

The processPipe (in Job.hpp) runs the generator to produce (key, value) pairs. It then pipes the output to MapTask to initialise it with generator's output known as MapTuple

⇓

The MapTask operator does - (k1,v1)->list(k2,v2) where (k2,v2) are created as pairs and added to results, defined as CombinerTuple

⇓

Now a combiner combines the output from the MapTask's of each processor into ReduceTuple

⇓

A distributor equally distributes the ReduceTuples to each processor for ReduceTask

⇓

The ReduceTask does - (k2,list(v2))->list(k3,v3) where (k3,v3) are created as pairs and added to results, giving ResultTuples

We have also defined an OutputTask that combines output from the ReduceTask of different processors and returns the final output. It also does any additional processing of the output before returning. In our implementation it simply divides the produced output by "N" - Number of nodes in the input graph, before returning.

After implementing the library methods we can define "mymap" and "myreduce" functions which are implemented similar to part-a and part-b and define data structures accordingly. We also define generator functions for each job.

Listing 14: MapReduce class instances for Factor and PageRank Calculation

```
using FactorJob = MapReduce<Graph::Vertex, Value, Graph::Size, Value,
    Graph::Size, Value>;
using PageRankJob = MapReduce<Graph::Vertex, VertexInfo, Graph::Vertex, Value,
    Graph::Vertex, Value>;
```

Listing 15: Job class instances for Factor and PageRank Calculation

```
using Calculation = Job<MapFactor, ReduceFactor, FactorData,
    FactorJob::Combiner, FactorJob::Distributor, OutputFactor>;
using Iteration = Job<MapPageRank, ReducePageRank, PageRankData,
    PageRankJob::Combiner, PageRankJob::Distributor, OutputPageRank>;
```

Observations and Conclusions

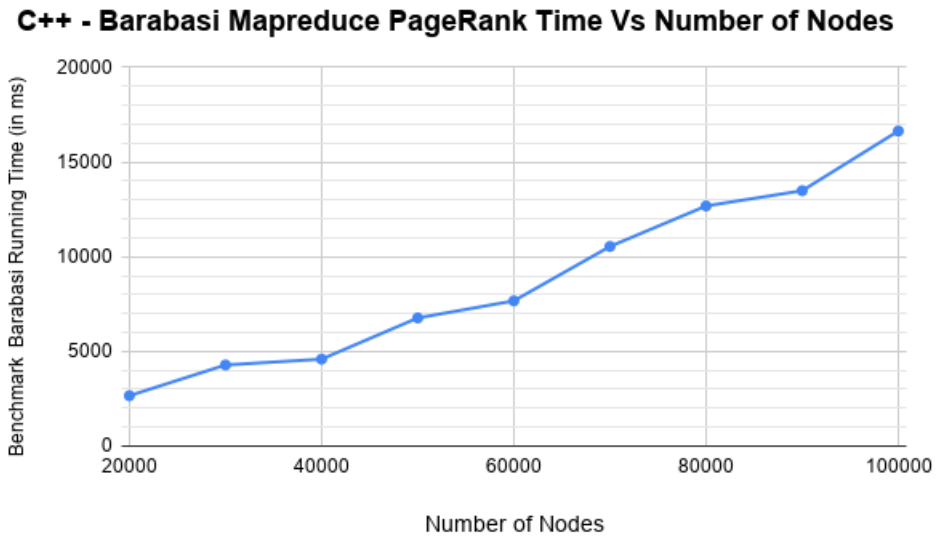
Execution Time of Different Implementations

Benchmark Test Examples	Number of Nodes N	Number of Edges E	MapReduce C++ Library (in ms)	MapReduce Self with MPI (in ms)	MapReduce Library with MPI (in ms)
barabasi-20000	20000	19999	2662	2052	7566
barabasi-30000	30000	29999	4286	4045	9280
barabasi-40000	40000	39999	4593	4170	9030
barabasi-50000	50000	49999	6769	6030	9433
barabasi-60000	60000	59999	7672	6773	9741
barabasi-70000	70000	69999	10544	9366	12394
barabasi-80000	80000	79999	12684	11441	14034
barabasi-90000	90000	89999	13489	11768	14285
barabasi-100000	100000	99999	16642	17754	16627
erdos-10000	10000	5111	297	170	940
erdos-20000	19999	9930	525	381	1660
erdos-30000	30000	14894	947	839	2034
erdos-40000	40000	20077	1226	1274	2282
erdos-50000	49999	25140	1548	1517	2586
erdos-60000	59999	29970	2108	1794	3023
erdos-70000	69999	34860	2284	2815	3245
erdos-80000	80000	39919	2395	3133	3712
erdos-90000	89999	45094	3010	3679	4107
erdos-100000	100000	50101	3443	3780	4197
bull	5	5	3	0	2250
chvatal	12	24	4	0	2639
coxeter	28	42	4	0	2406
cubical	8	12	5	0	2622
diamond	4	5	3	0	2943
dodecahedral	20	30	5	1	3465
folkman	20	40	4	0	2589
franklin	12	18	5	0	4023
frucht	12	18	5	0	2331
grotzsch	11	20	2	0	1407
heawood	14	21	5	0	4019
herschel	11	18	3	0	2055
house	5	6	4	0	3230
housex	5	8	3	0	2653
icosahedral	12	30	4	0	2734
krackhardt_kite	10	18	15	0	5543
levi	30	45	9	1	6611
mcgee	24	36	10	1	4975
meredith	70	140	6	1	2364
noperfectmatching	16	27	7	1	4357
nonline	50	72	5	1	2501
octahedral	6	12	3	0	2347
petersen	10	15	2	0	2418
robertson	19	38	18	1	3158
smallestcyclicgroup	9	15	2	0	2120
tetrahedral	4	6	2	0	1960
thomassen	34	52	10	1	5205
tutte	46	69	13	2	3332
uniquely3colorable	12	22	4	0	1890
walther	25	31	8	1	3082

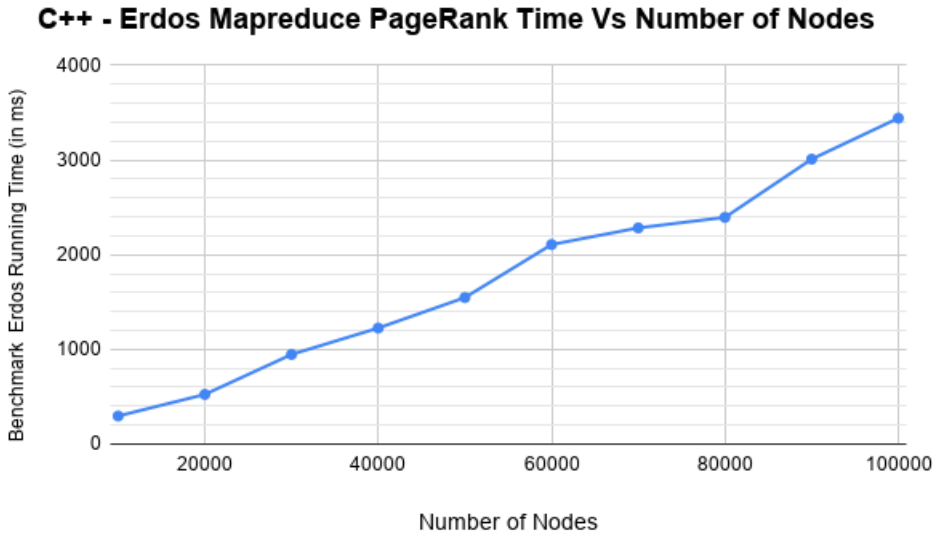
Graphs and Analysis

MapReduce C++ Library

1. Benchmark - Barabasi

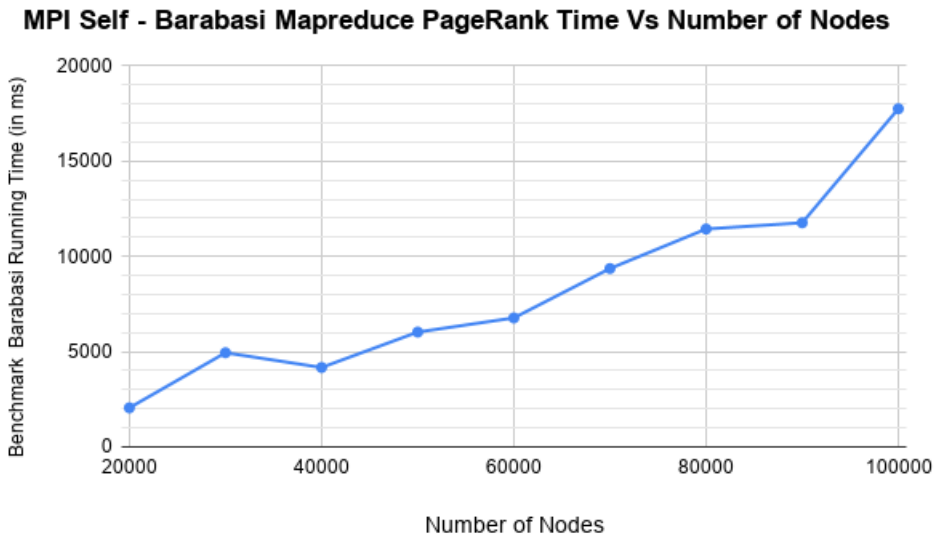


2. Benchmark - Erdos

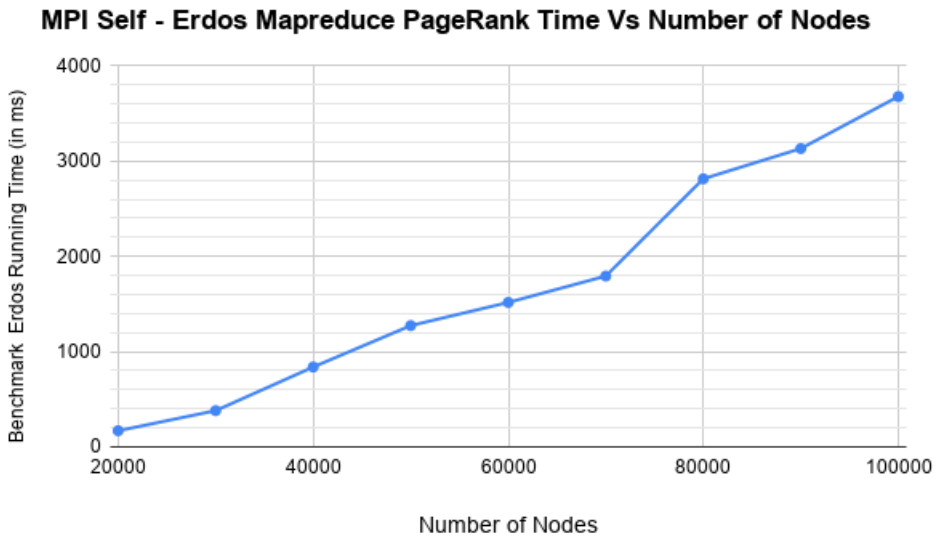


MapReduce Self with MPI

1. Benchmark - Barabasi

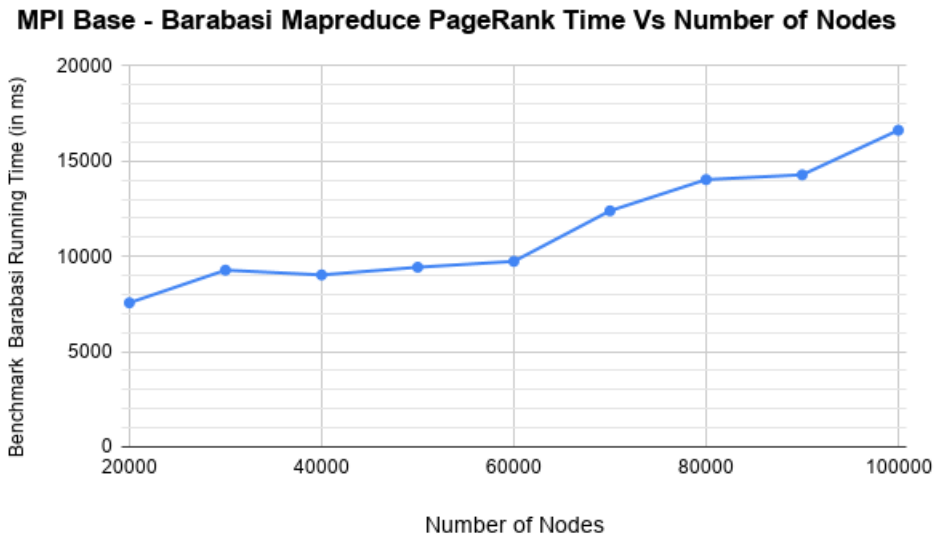


2. Benchmark - Erdos

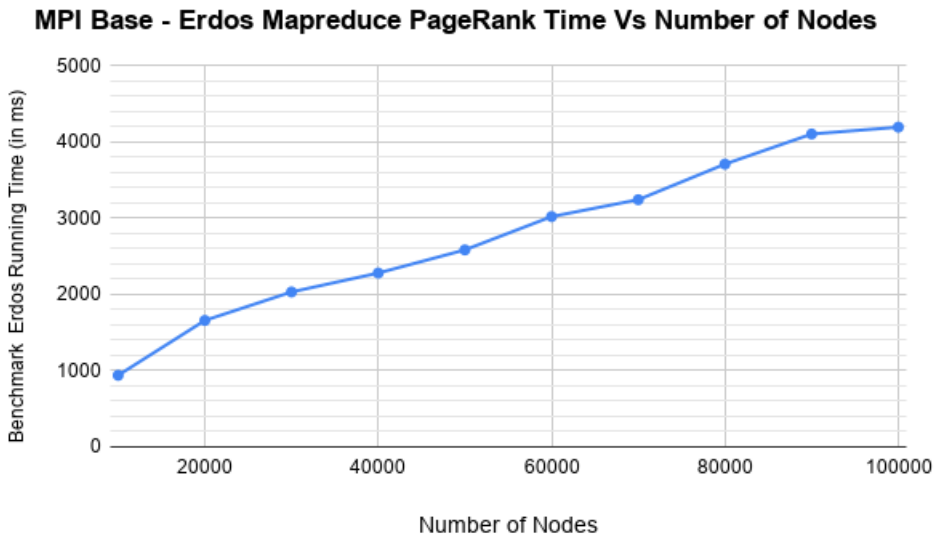


MapReduce with MPI Library

1. Benchmark - Barabasi

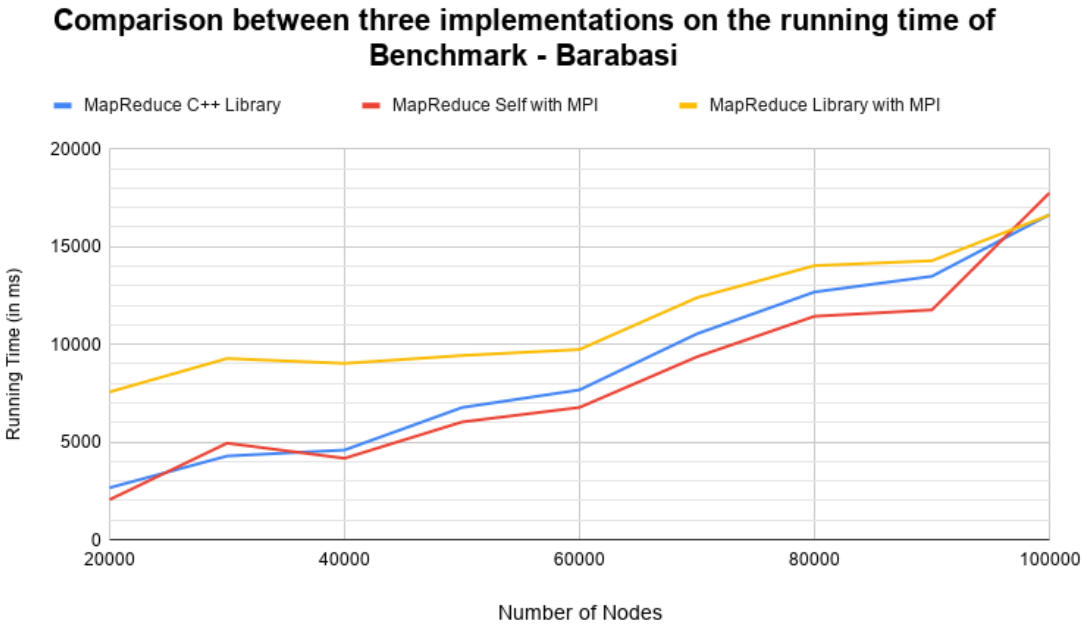


2. Benchmark - Erdos

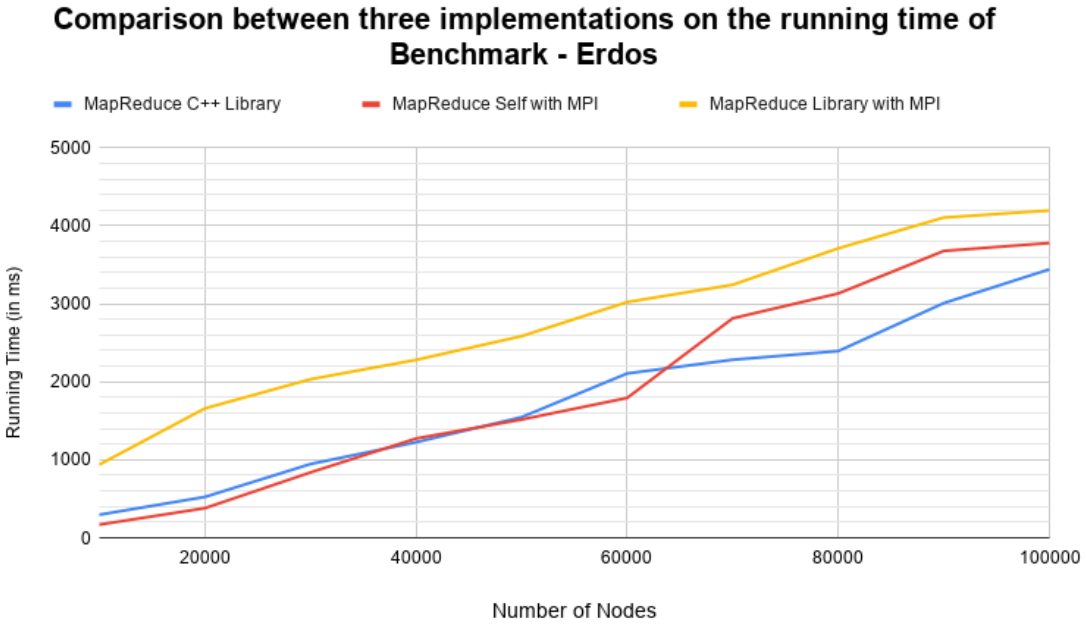


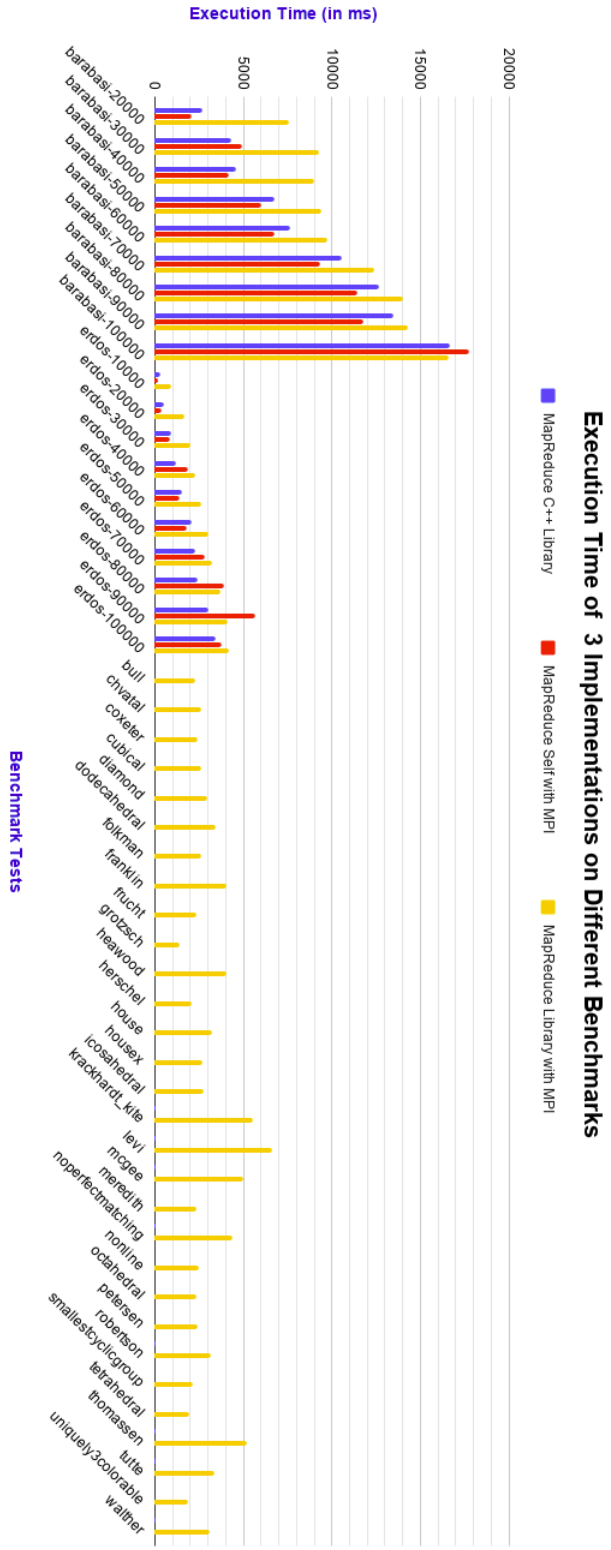
Comparative Analysis of Different Implementations

1. Benchmark - Barabasi



2. Benchmark - Erdos




Execution Time of 3 Implementations on Different Benchmarks
MapReduce C++ Library
MapReduce Self with MPI
MapReduce Library with MPI
Execution Time (in ms)
0
5000
10000
15000
20000
Benchmark Tests
barabasi-20000
barabasi-30000
barabasi-40000
barabasi-50000
barabasi-60000
barabasi-70000
barabasi-80000
barabasi-90000
barabasi-100000
erdos-10000
erdos-20000
erdos-30000
erdos-40000
erdos-50000
erdos-60000
erdos-70000
erdos-80000
erdos-90000
erdos-100000
bull
chvatal
coxeter
cubical
diamond
dodecahedral
folkman
franklin
frucht
grotzsch
heawood
herschel
house
housex
icosahedral
krackhardt_kite
levi
mcgee
meredith
noperfectmatching
nonline
octahedral
petersen
robertson
smallestcyclicgroup
tetrahedral
thomassen
tutte
uniquely3colorable
walther

Observations and Explanation of Graph Trends

1. One common observation is that the execution time increases strictly with increasing size of network i.e increasing number of nodes.

2. We can see that the execution time of sparse networks (ex: Erdos) is much less than comparatively denser networks (ex: Barabasi) for all the three implementations.

3. From the previous point we can also conclude that our implementations take more time to calculate pagerank of networks with more edges.

4. From the comparative analysis graph we can see that the MapReduce Library with MPI (part c) has a considerable parallel communication overhead and thus it takes significant time to run on very small graphs compared to the other implementations (ex: bull).

5. In terms of running time the performance of first two implementations - "MapReduce C++ Library" and "Self MapReduce with MPI", is fairly similar and better than the third implemnetation - "MapReduce MPI Library". A possible reason could be the significant parallel communication overhead in the third.