MPI Assignment

Yukti Makhija (2019BB10067)

1) PageRank problem statement

The PageRank problem is to rank a set of webpages in a closed network in the order that a random surfer might visit them in order, jumping from page to page via hyperlinks and/or randomly.
Let us assume we are given a hyperlink matrix ***H***, which has $\boldsymbol{1/N_j}$ for all links originating from page ***j***, where N_j is the number of total connections of page ***j***.
So, for example, let's take the *"tetrahderal"* network
(0→1, 0→2, 0→3, 1→2, 1→3, 2→3)
H =

0	0	0	0
1/3	0	0	0
1/3	1/2	0	0
1/3	1/2	1	0

Using the power law method, we would not be able to solve this as the page 3 "drains" all the probability. To fix this, we can simply replace that column with **1/N** (N = 4). This is called the stochastic matrix ***S***.

This is **guaranteed** to have a stationary vector (i.e. solution to the page rank problem).

But, this limits our 'surfer' to only following links in the web. The surfer might also decide to randomly open another webpage, with some probability. This probability is taken conventionally to be $\boldsymbol{\beta}$ = **0.15**. We achieve this by adding a matrix of all ones divided by ***N*** to S and weighting these two terms using $\boldsymbol{\beta}$ and ***(1 - β).*** So,

$$G = (\boldsymbol{\beta}/N)1_{N*N} + (1 - \boldsymbol{\beta}) * S$$

Now, we can apply power law method to get
$I^{k+1} = GI^{k}$
until they converge.

2) Map-Reduce paradigm

Map Reduce allows us to convert problems into parallel ones.

This involves using a map function to solve smaller chunks of that problem and emit intermediate key-value pairs. These pairs are then gathered and combined into the final result using a reduce function.

3) Modelling PageRank as a Map-Reduce problem

I have modelled the map and reduce parts as follows:

Mapper: For each machine/processor (***p***), {key= j $\in$ [***1 to N***], value = jth element of $G_p{*}I_p^k$}

where, G_p is the vertical stripe of ***G*** (columns ***N*p/P*** to ***N*(p+1)/P)*** and I_p is also found the same way

Reducer: Add all values for key j to get the jth element of final vector I^{k+1}.
Repeat for all j and we get I^{k+1}.

Complexity Analysis
Pagerank algorithm provided in the assignment involves multiplying a stochastic matrix (NxN) to a vector(Nx1) continuously till it converges(i.e. it becomes stationary). The resultant complexity is of the form $O(N^2)$. I try to reduce this complexity and make it proportional to the number of connections (or hyperlinks (h)) O(h). This is achieved because the matrix provided is sparse and iterating over all the connections instead of rows would save a lot of time and bring down the runtime.

Different ways of storing G
I tested out the adjacency matrix type approach (using ***G***), and also an adjacency list type approach (by making 2 vectors for each node, one to store the connection and another to store the value at that connection (which would be the value in the matrix)).

When I compared the results for both matrix approach and adjacency list, I found the matrix approach required a lot of memory when the number of pages was large. On the other hand, adjacency lists require lesser memory since the matrices are quite sparse and as explained above, iterating on all hyperlinks brings down the time required also. Hence, for all the final results I use adjacency lists.

Part 1: cdmh MapReduce Library

I implemented this PageRank problem in the given library with the help of the provided templates for a MapTask and a ReduceTask.
I used the null combiner given in the library as this is sequential.

Part 2: MPI, Own Implementation

I also implemented this using MPI. I used the MPICH implementation of the MPI standard. My mapper function stored the value of each new element of $\boldsymbol{I_p}$ in the vector itself, rather than creating key-value pairs. We can see this as the index being the key. Then for the reduce step, I called the Allreduce operation on the whole vector, as that would handle communication between the processors optimally.

Part 3: Map-Reduce MPI Library

Using the MPI Map Reduce library, I had to write my map task and my reduce task in the form of 2 functions, that are arguments to the library's map and reduce function and are called internally on each key, so that the library can handle the parallelism (where to send each key).

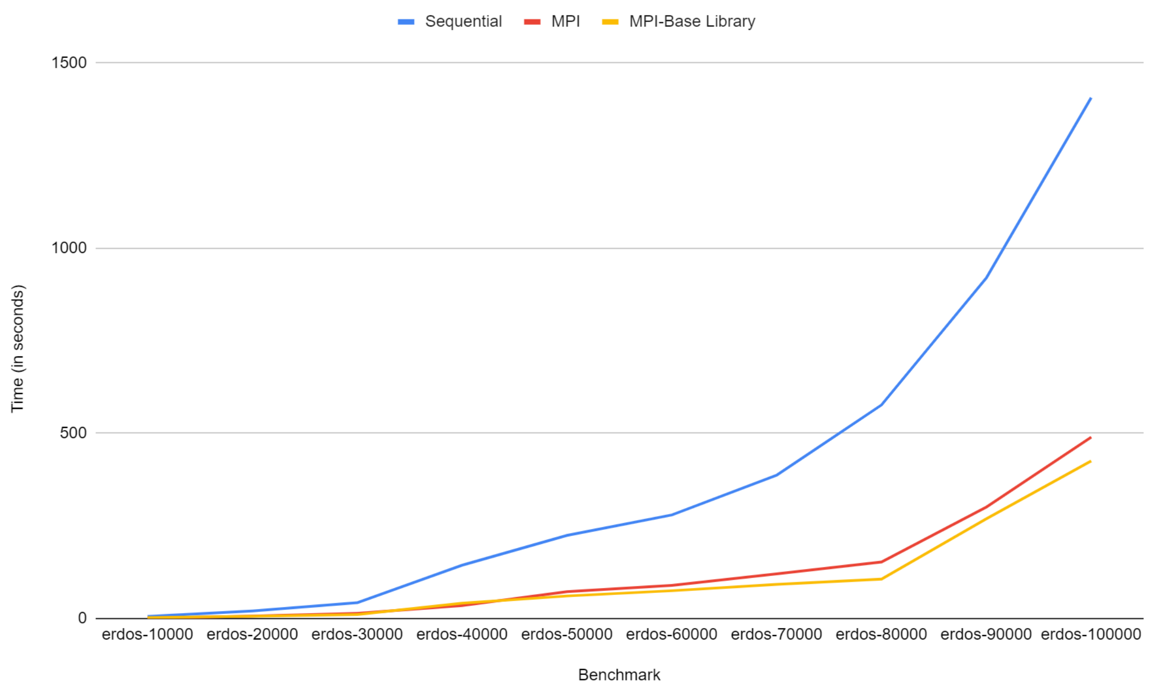
4) Results

The runtime for all the benchmarks (the number of processes is 4) is as follows:

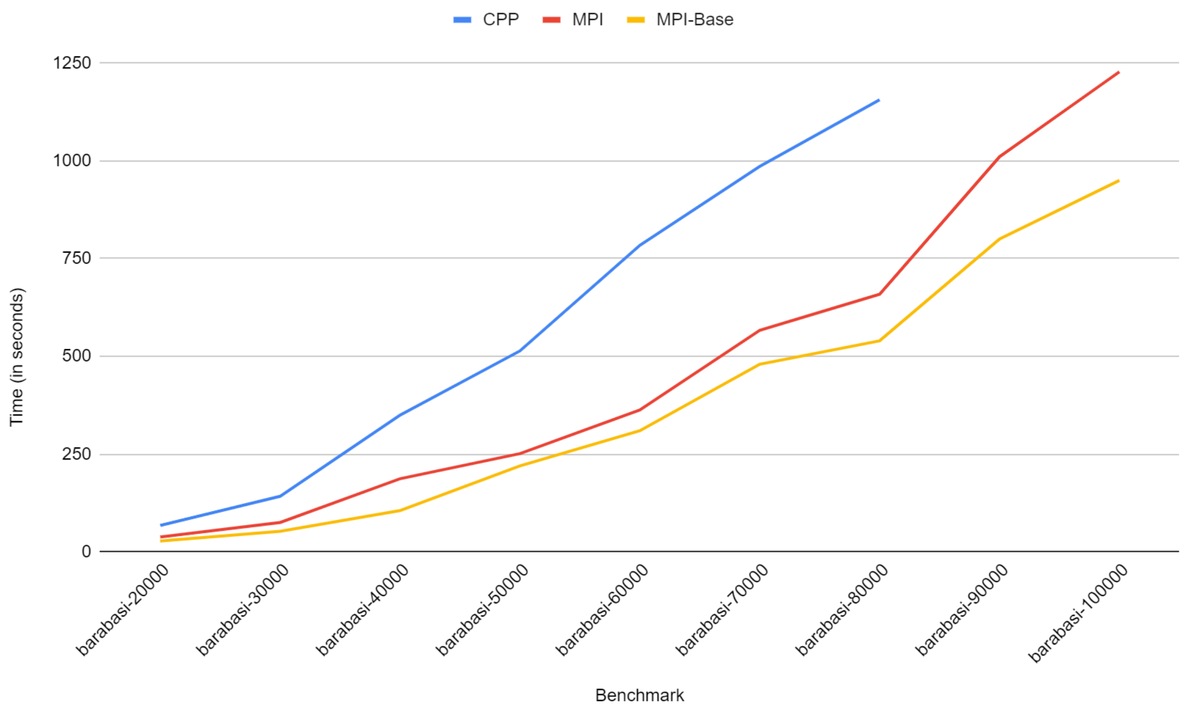
File	Part 1 (cdmh MapReduce C++)	Part 2 (My implementation on MPI)	Part 3 (Map-Reduce MPI Library)
bull	0.034972	0.0000240	0.086072
chvatal	0.041713	0.000031	0.05545
coxeter	0.039839	0.000054	0.079237
cubical	0.042707	0.000034	0.335809
diamond	0.032889	0.0000209	0.095209
dodecahedral	0.054111	0.000083	0.111773
folkman	0.040676	0.000059	0.077201
franklin	0.057441	0.000068	0.087296

frucht	0.034747	0.000065	0.101292
grotzsch	0.020818	0.000020	0.059456
heawood	0.055783	0.000032	0.094905
herschel	0.025156	0.000024	0.091039
house	0.048078	0.0000407	0.107965
housex	0.029866	0.0000159	0.084868
icosahedral	0.044613	0.000048	0.085007
krackhardt_kite	0.075041	0.000097	0.444971
levi	0.07118	0.000067	0.078714
mcgee	0.062135	0.000052	0.202762
meredith	0.0288	0.000177	0.09165
noperfectmatching	0.062573	0.000086	0.062407
nonline	0.033171	0.000145	0.061511
octahedral	0.030021	0.0000219	0.097254
petersen	0.028992	0.000025	0.231033
robertson	0.054266	0.000075	0.108017
smallestcyclicgroup	0.024862	0.000028	0.290899
tetrahedral	0.026654	0.0000224	0.095015
thomassen	0.081409	0.000281	0.095935
tutte	0.05122	0.000139	0.067451
uniquely3colorable	0.028102	0.000036	0.068733
walther	0.050191	0.000083	0.120173

Runtime analysis for Erdos:



Runtime analysis for Barabasi:



From the graphs and table, it is clear that the sequential approach in part 1 (using mapreduce c++) requires maximum time for all the benchmarks. The difference between the sequential and MPI runtime increases with the number of nodes (or pages). On comparing the runtime of our implementation with that of mapreduce MPI library we found that when the number of pages is smaller the runtimes are comparable and have similar order of magnitude (shown in table) but with large pages the runtime of MPI library is significantly lesser. This is also visible from the graphs.

To conclude, we have successfully completed our MPI implementation which performs similar to the baseline provided and differences are observed only at very large values of pages. I have also explained above why the time complexity of sequential is higher and now we improved it with adjacency lists.