COL730 Assignment 3 - Parallel PageRank using MapReduce

Pagerank algorithm comprised of a series of matrix product operations between S (which stores all the connections) and a vector q. We keep iterating till the q doesn't converge. The complexity can be formally written as O(n^2) where n is the number of pages. This can be improved by iterating over the connections in S (as it sparse) rather than the rows of S, so the final complexity would be O(l) where l is number of links.

Setup of map-reduce problem

Let the hyperlink matrix be H.
Then we can find the stochastic matrix S (in which all nodes without outgoing connections have their column $= 1/N$ for all other nodes, i.e., equal chance of jumping to any node).
Let the page rank vector (stationary vector) be q.
Since we are using the power method, $q^{(k)}$ represents the stationary vector after k iterations.

Then, we can calculate $q^{(k+1)}$ as:

$$q^{(k+1)} = \alpha * S * q^{(k)} + \frac{(1-\alpha)}{N} * \mathbb{1}^n * q^{(k)}$$

where α is an explorative weighting (usually = 0.85), $\mathbb{1}^m$ is the $m \times m$ matrix of all ones, and the second term is added to model exploration.

We can model this using map-reduce:

With p processors,
Assume nodes $1 \dots \frac{N}{p}$ are being calculated on proc. p_0, (and so on).

$$\mathcal{M}ap: \ key = i, value = q_{i,p}^{(k+1)} \ for \ i \in \{1 \dots n\}, for \ each \ proc. \ p$$

$$\mathcal{R}educe: \ q_i^{(k+1)} = \sum_1^p q_{i,p}^{(k+1)}; \ summing \ up \ q \ from \ each \ proc.$$

Part 1: Using c++ mapreduce library (sequential/shared memory parallelism using pthreads)

I have used this modular library to set up the map reduce problem as shown in the documentation. `MapTask` and `ReduceTask` both need the `operator()` function to be defined, and additionally a "data source" to initiate the values of the keys.

Part 2: Using MPI (distributed memory; own implementation)

I have implemented the map reduce in MPI form as well. This was achieved by making the "map" implicit.
That is, instead of mapping $i : q_i^{(k+1)}$, I directly store the value in $q^{(k+1)}$, implicitly using the index as a key.

This makes the collate and reduce steps much easier, as now we can just call `MPI_Allreduce` on the whole of $q^{(k+1)}$ and nothing else.
Moreover, `MPI` can decide whichever way is the best and most optimal for this all-to-all communication (like butterfly etc.).

A simplified code is as follows:

```
// q_curr is a vector of doubles, representing q^{(k+1)}.
while (convergence)
{
    // Map
    for (int j = start; j < end; j++)
        for (int i = 0; i < total_pages; ++i)
            q_curr[i] += alpha*S[j][i]*q_last[j] + (1.0 -
alpha)/total_pages*q_last[j];

    MPI_Barrier(MPI_COMM_WORLD);
    // Collate and Reduce all in one step!
    MPI_Allreduce(MPI_IN_PLACE, q_curr.data(), total_pages, MPI_DOUBLE,
MPI_SUM, MPI_COMM_WORLD);
}
```

Part 3: Using MPI MapReduce library (distributed memory; library)

The MPI MapReduce library requires the definition of at least 2 functions: a map function and a reduce function, the prototypes of which are given in the documentation. In the map function, we have to add our keys and values to the `KeyValue` object provided by the library. It then finds all duplicate keys (either on-processor using `convert` or over all processors using `collate`) and collects them into a `KeyMultiValue` object. We then perform some operations on this object in the reduce function.

The library automatically assigns keys for mapping and reducing to different processors.

A pseudocode would be as follows:

```
// MAP FUNCTION
map_make_list(rank, *kv, *args) {
    temp_row[mrargs->total_pages] = {};
    for (j:start:end; j++)
        for (i:0:total_pages; ++i)
            temp_row[i] += alpha*S[j][i]*q_last[j] + (1.0 -
alpha)/total_pages*q_last[j];

    for (i:0:total_pages; i++)
        kv->add(i, temp_row[i]);
}
// REDUCE FUNCTION
calc_q(key, *multivalue, nvalues, *kv, *args) {
    sum = 0.0;
    for (i:0:nvalues; ++i)
        row = ((double *) multivalue)+i;
        sum += *row;
    q_curr[key] = sum;
```

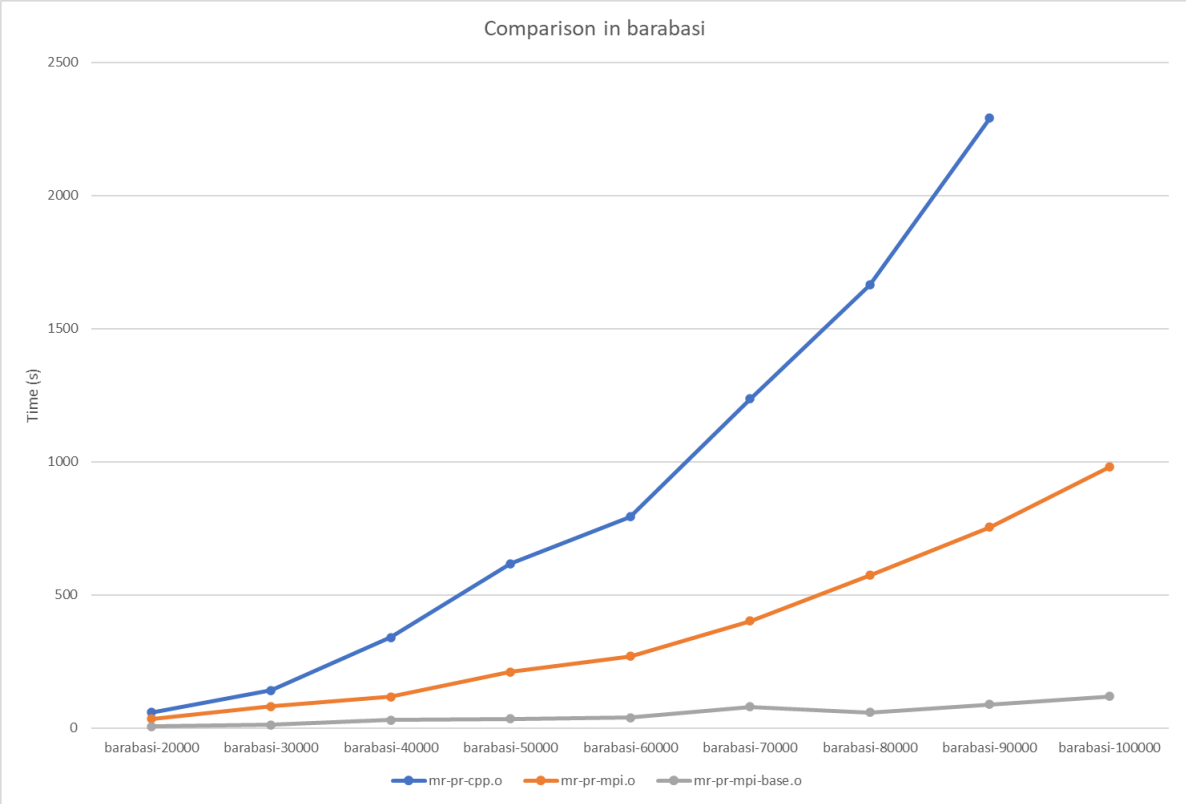
```
}

// MAIN LOOP
while (convergence)
{
    mr->map(n_proc, map_make_list, &args);
    mr->collate(NULL);
    mr->reduce(calc_q, &args);
    MPI_Barrier(MPI_COMM_WORLD);
    // Gather the reduced parts of q_curr from all procs
}
```

Results

Graphs comparing times for the three implementations:

1. Barabasi



2. Erdos

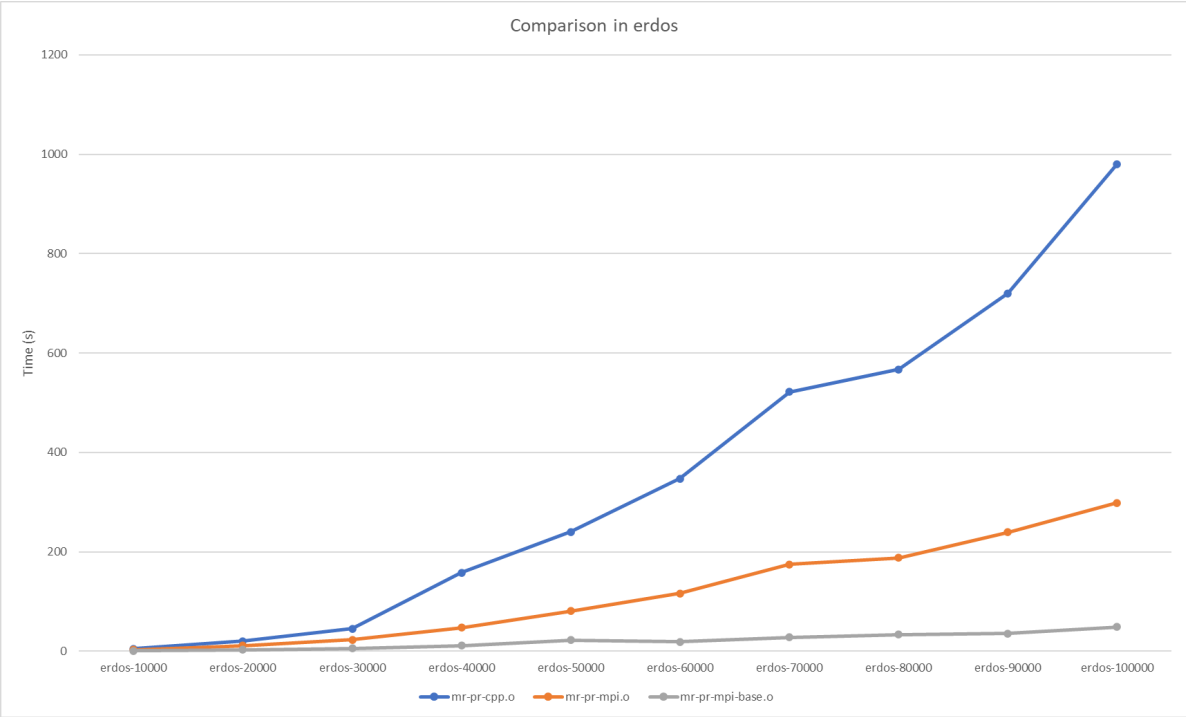


Table comparing all the timings:

file	n_procs (MPI)	./mr-pr-cpp.o	./mr-pr-mpi.o	./mr-pr-mpi-base.o
bull	2	0.012264	0.000101	0.052918
chvatal	4	0.016669	0.000207	0.264667
coxeter	4	0.014950	0.000174	0.163464
cubical	4	0.014281	0.000128	0.255751
diamond	2	0.013148	0.000142	0.069884
dodecahedral	4	0.022000	0.000437	0.657886
folkman	4	0.016072	0.000176	0.476189
franklin	4	0.018318	0.000262	0.316360
frucht	4	0.015015	0.000188	0.518150
grotzsch	4	0.008539	0.000086	0.343903
heawood	4	0.019619	0.000207	0.430348
herschel	4	0.014141	0.000119	0.337505
house	2	0.015226	0.000093	0.069628
housex	2	0.012398	0.000122	0.069702
icosahedral	4	0.017330	0.000181	0.304893
krackhardt_kite	4	0.023202	0.000201	0.230388
levi	4	0.021180	0.000342	0.182505

file	n_procs (MPI)	./mr-pr-cpp.o	./mr-pr-mpi.o	./mr-pr-mpi-base.o
mcgee	4	0.026180	0.000313	0.831685
meredith	4	0.011847	0.000226	0.254503
noperfectmatching	4	0.024758	0.000388	0.225912
nonline	4	0.014673	0.000292	0.157094
octahedral	2	0.011666	0.000107	0.063216
petersen	4	0.010246	0.000161	0.158343
robertson	4	0.016554	0.000245	0.325044
smallestcyclicgroup	4	0.011363	0.000126	0.187598
tetrahedral	2	0.010251	0.000079	0.048021
thomassen	4	0.025999	0.000535	0.206829
tutte	4	0.018884	0.000392	0.103342
uniquely3colorable	4	0.009821	0.000149	0.344742
walther	4	0.018815	0.000314	0.725735

Conclusion

I finished the MPI implementations of PageRank and have shown that it performs better to the library provided. Both MPI implementation ($\mathcal{O}(l)$, l=links) always perform better than C++ map reduce library in part one because that code is serial ($\mathcal{O}(n^2)$)).

Additionally, I have also tested a sparse representation of the matrix, but have shown results for the adj matrix representation.