Assignment 3; COL380

Priyanshu Gautam, 2017CS10362 Rachit Kumar, 2017CS10364

Abstract

In this report we present the rationale behind our codes and try to compare the different times each of the non-parallel and parallel programs take in order to compute the page ranks for a system with webpages varying between a few to 1000000.

1 General Algorithm

In this section we shall describe the outline algorithm to implement `Page Rank` using `Map-Reduce` procedure.
We break the calculation

$$M = sA + sD + tE \tag{1}$$

In the above equation, A is the general matrix with links outside, D is the matrix with only the *holes* having entries 1, rest 0. While E is the random matrix, having all values as $\frac{1}{n}$, where n is the size of M. And $s + t = 1$. Throughout the assignment, $s = 0.85$ First step is to have two map-reduce jobs.

1. To calculate Dp. In order to accomplish this, we use map-reduce to calculate inner product. We rewrite

   $$Dp = I(d\dot{p})/n \tag{2}$$

   Where d is a mask vector of holes, and p is the probability distribution. We already have a machine using map-reduce to compute inner product, so we feed this computation to that map-reduce job, and we get vector Dp.

2. Using the value of Dp, calculated in previous step, we now compute Mp. To compute this, calculate an outgoing vector v_i, and for each page k_i, using the mapper; send the pair (k_i, $s\ \frac{p_i}{size(v_i)}$). We also add Dp, and for each key add the random part contribution which is $\frac{t}{n}$. Finally, reducer sums for each key.

2 Part a (Map Reduce Library)

We were supposed to use the library given in the assignment details to implement our version of page rank computation using map reduce. To implement the general algorithm above, we made use of three structs.

1. `datasource`: This is used to convert the given input into key-value pair. We have used this to use the existing input of the form of vector of vector to key val pairs.

2. `mapper`: This is used to create key-value pairs which will be collected and reduced by the reducer. The key-value pairs are created as per the aforementioned steps

3. `reducer`: It simply sums each value for each key that is received. And sends the reduced key-value pair to the callee.

Two different namespaces, one each for the two jobs were created and executed one after the other.

3 Part b(Our own Implementation)

We implemented our own variations of map-reduce functions for these parts. These functions behave in a similar way to the functions in part a. Mapper *emits* one key-value pair. This key-value pair is collected by an intermediate function(or a semi-reducer). The semi reducer sorts the incoming pair with respect to the keys, and sums. The sorting algorithm used is *bucket sort*. This intermediate function then sends the partial sums to the final or master reduce, which provides us with the output. Master reduce collects already reduced sums and gives the final key, val pair as required by the algorithm. Work was also divided equally by division using modulo nprocs.

4 Part c(MPI library)

In this part, we implemented pagerank using the MapReduce-MPI library which is an open-source implementation of MapReduce written for distributed-memory parallel machines on top of standard MPI message passing. This has been implemented by defining a mapper and a reducer function. The mapper function creates the key-value pairs parallely, which are stored in a KeyValue object. These are then gathered and converted into a KeyMultiValue object. Then the reducer function sums the multi-values of each page to get the pageranks for that iteration.

5 Experimentation

The programs were run on our machine of *MacBook Pro*, 2.3 GHz dual core. All the parts were run with the default values. For part a, number of map tasks were est to default of 1, and number of reduced tasks were set to the default of 4(hardware concurrency value). For the other two parts, we used as many slots as MPI allotted to our machines, which were 2 physical slots. The correctness was mapped with the correctness given in tests.

6 Results

6.1 Part a

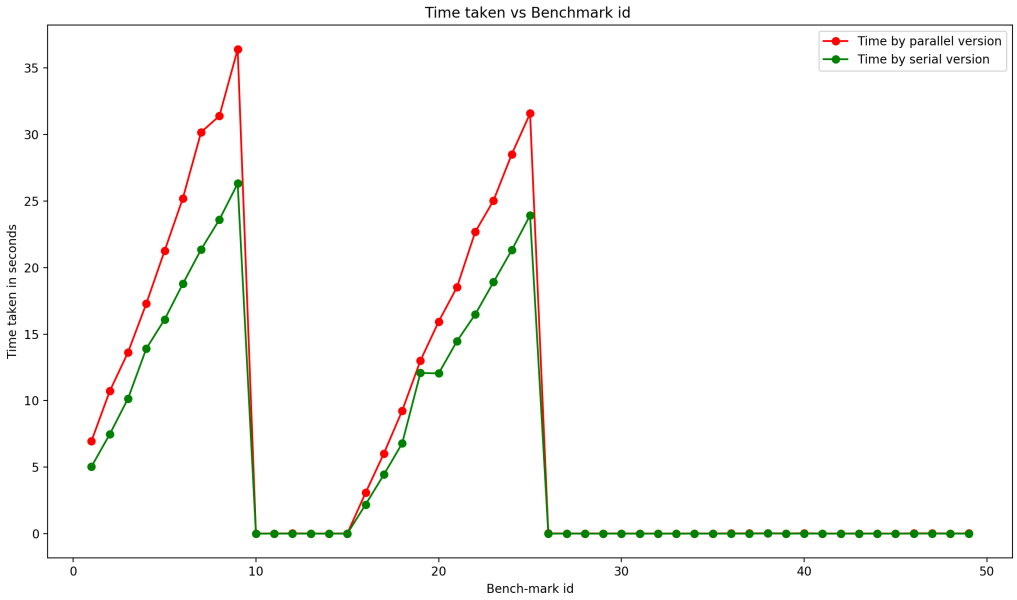

Figure 1: partA-time

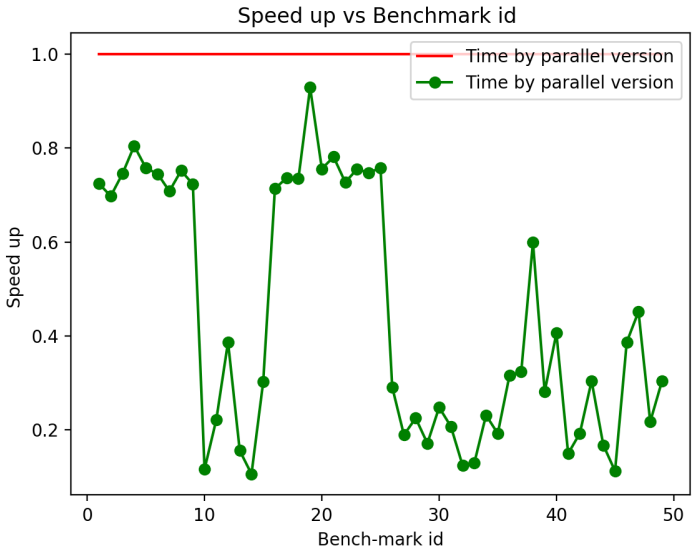

Figure 2: partA-speedup

6.2 Part b

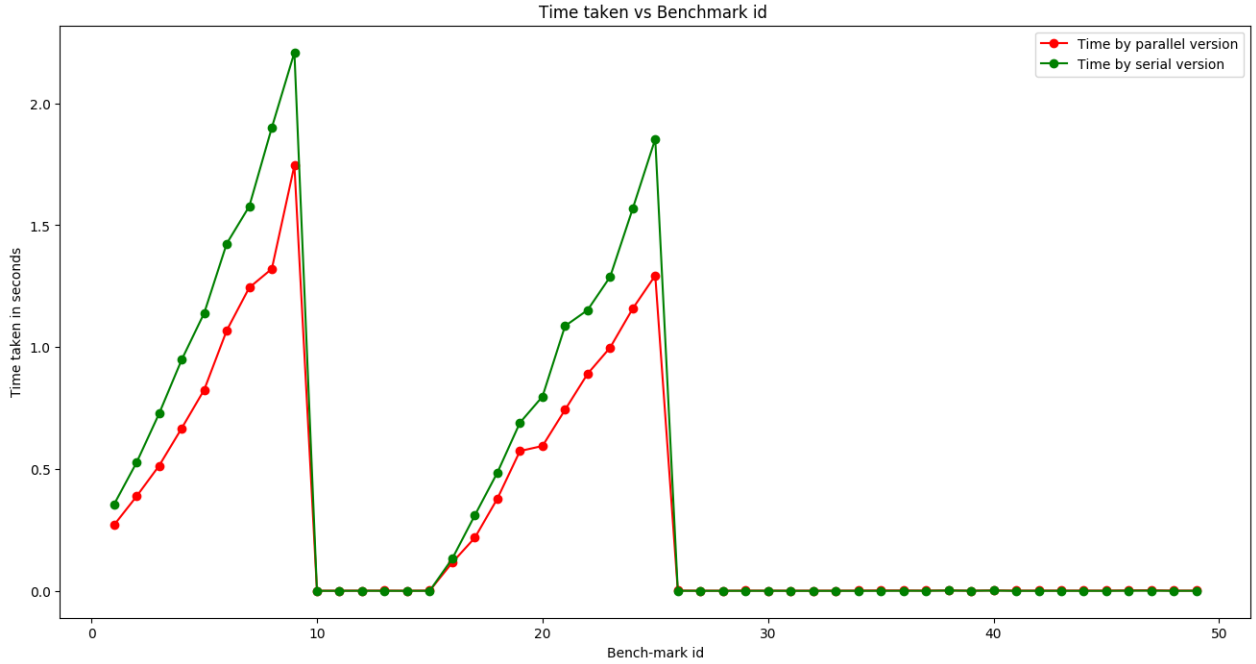

Figure 3: partB-time

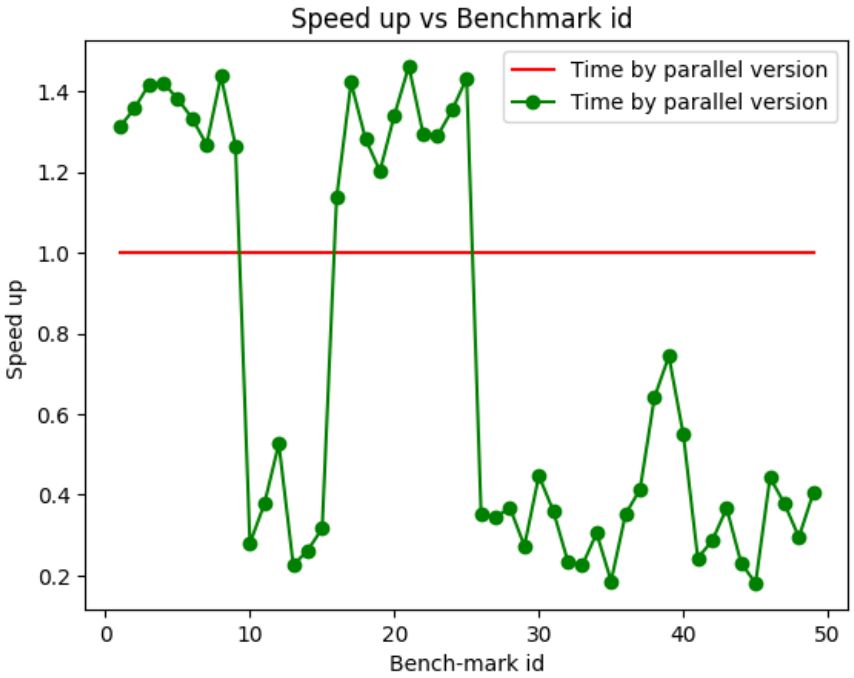

Figure 4: partB-speedup

6.3 Part c

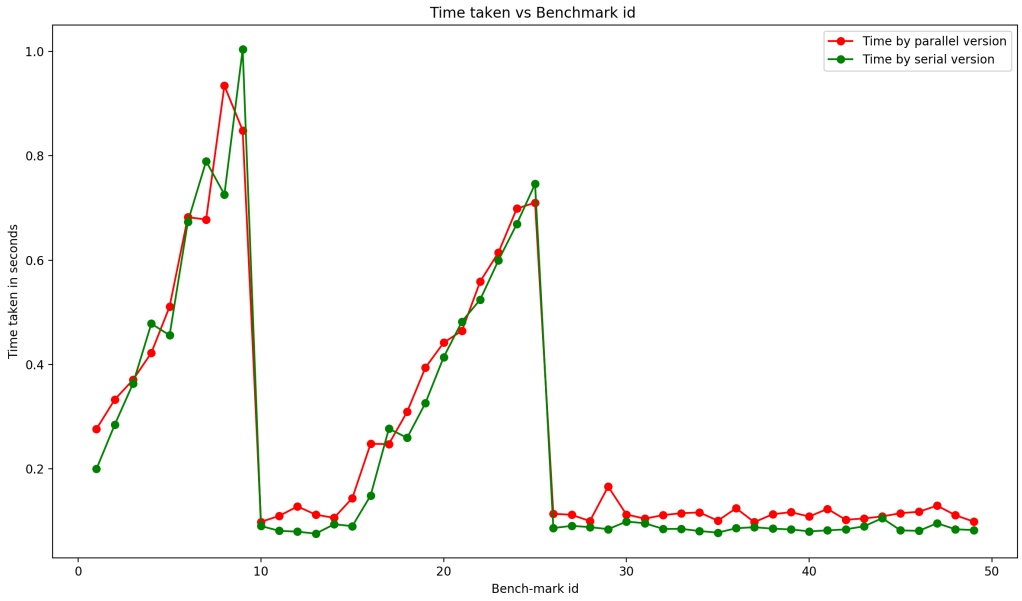

Figure 5: partC-time

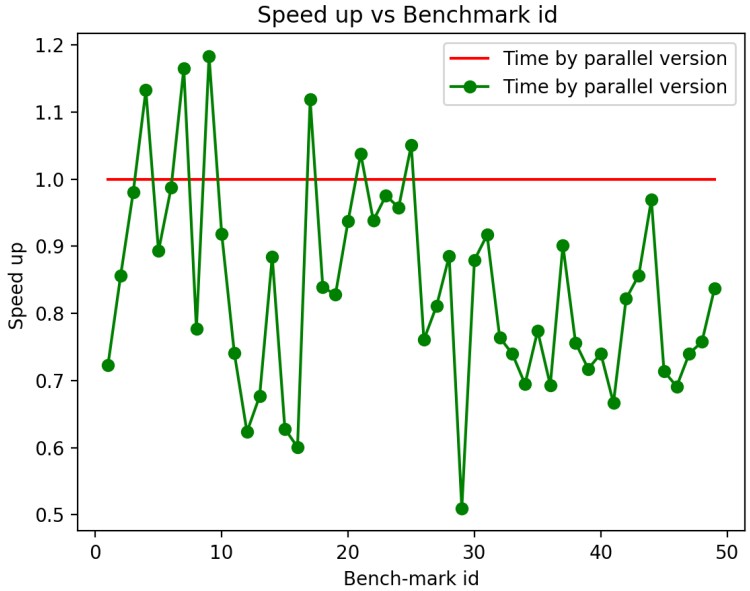

Figure 6: partC-speedup

7 Inferences

We will try to summarise the major takeaways and the tables.

7.1 Part a

We observe that the speed up produced is less than 1 throughout. That is, our parallel code is always slower than the sequential code. This may come as a surprise but is explainable if we look deeper. If we run part a by specifying number of cores as 1, we observe that the time taken for 1 core is more than that of time taken when using 2 or more cores. Which means, our parallelisation is indeed providing speed-up. But, the overhead with process creation is so large that compared to no parallelisation, more time is consumed. This is also in part because we are using 4 reduce tasks which implies overhead of procuring outputs at each rank or thread.

Changing number of map and reduce tasks : By default, the function only takes in default values for number of map-tasks and number of reduce tasks. The default value is 0 for number of map-tasks and (number of cores) for reduce-tasks. The graphs using these values has been shown above and analysed. But further analysis done by us shows that the ideal value for our machines is 3 for map-tasks and 1 for reduced tasks. The general pattern to observe is that the ideal values would be n map-tasks and 1 reduce tasks. Where n is the number of cores. Since our machine has 2 physical but 4 logical cores, value of 3 is best in terms of speed up. Note: Even this value gives speed up less than 1.

Using O flags We observe that usage of O1, O2 and O3 flags make the programs faster by 60-70 per cent but there is no positive contribution to speed up on using O flags

7.2 Part b

This is the only part which shows speed up. We have implemented the map-reduce functions as stated above. The speed up(in the range 1.2 to 1.5) exists for large systems with more than thousands of pages but for smaller examples the speed up is infact less than 1. This can be attributed to the overhead of communication, for smaller cases the communication acts as bottleneck, while computation is a small part when the system only has a few pages.

Changing number of processes On changing the number of processes, we see that for and around 3, speed up increases to values of 1.5 to 1.7. This goes well with the fact that our machine has 2 physical and 4 logical cores, therefore a maximum around 3 cores is expected.

Optimisation flags On using O1, the times decrease but there is a sharper decrease in sequential time, and speed up falls below 1 for all values. On using O2, and O3, there is a decrease in time and speed up, but speed up remains above 1. This is a result of the optimisations also affecting map-reduce sub-routines. The max speed up achieved is 1.3.

7.3 Part c

Here the speedup produced is close to but less than 1 for smaller cases, but on very large cases, we notice a small speedup, which decreases when we increase the number of cores beyond 2, indicating a large communication overhead.

Optimisation flags O1 flag scales very well on serial and we see that speed up is non existent for large values in case of O1, but a speed up exists for smaller systems. While the opposite is true for O2 and O3, which show speed up for large systems.

7.4 Comparison with each other

We see that either for large, or small values part a performs the worst. This is a somewhat an expected result as the library is quite mutlti-purpose and gives the user a lot of control over the map and reduce tasks. Therefore a large overhead is observed in the algorithm.

Amongst part b and c, for smaller systems: part b is much much faster than part a. Almost 20 times speed for serial version, and almost 4 times as much speed is observed for part b in comparison to part c. While for larger systems part b was slow by a factor of 2 to 4. This is explainable as in part b we produce a key value pair individually, not as an array and therefore the number of keys sent becomes extremely large in case of part b. Which makes the execution slower, and is also visible in 1 core process as it still is communicating amongst its only core. While in part c we are sending arrays, therefore number of communications are less. And even for large systems it scales.