SDS 374C - Parallel Computing for Science & Engineering

Final Project Report

Chahak Mehta (cm59984)

May 11, 2022

Abstract

PageRank is a fundamental link analysis graph algorithm to evaluate the importance of vertices. It forms one of the backbones of search engines like Google, DuckDuckGo etc. to determine the relative importance of webpages to rank search results. As of 11th May 2022, there are at least 4.26 billion indexed webpages[1].For such large graphs calculating the pagerank values can be an expensive task, so to improve the speed of search results, we can perform this calculation in a parallel manner. The key observation that enables parallelization is that at each iteration we can perform the updates for each vertex independently. Moreover, the overall fundamental problem is an error minimization problem that requires us to parallelize a while loop. The experiments were performed on two datasets - a subset of Google Web Graph and the LiveJournal social network provided by the SNAP project[2].

1 Introduction

Graph analytics plays a critical role in many applications such as genome analysis, cybersecurity, fraud detection, social networks, identifying points of weakness in infrastructure networks like power grids, etc. Hence, algorithms that analyze graphs are helpful for a diverse set of problems. However, these graph algorithms often are time intensive and thus can be a prime case study to understand the importance of parallel implementations. One such graph ranking algorithms is the PageRank algorithm which was initially developed at Google for ranking web pages. It can be applied to any graph - directed or undirected. It measures centrality through a recursive definition: important nodes are those that are connected to and from other important nodes. The underlying assumption is that more important websites are likely to receive more links from other websites. The remainder of the report is organized as follows. After reviewing the PageRank algorithm, we introduce the parallel version of the algorithm based on OpenMP in Section 2. Section 3 discusses various scaling experiments on different datasets. Section 4 concludes with discussion of results and further work.

2 Background and Implementation

2.1 PageRank

The PageRank algorithm is a widely used algorithm for ranking the importance of vertices in a graph. It computes a probability value for each vertex that indicates the likelihood that a user will reach that page. A higher value corresponds to more importance. For a static input graph, the

[1] `https://www.worldwidewebsize.com/`

[2] `https://snap.stanford.edu/data/index.html`

PageRank algorithm traverses the entire graph iteratively. In each iteration, a vertex v updates its PageRank value based on Equation 1 where d is a damping factor (set to 0.8), N is the total number of vertices in the graph, $M(v_i)$ is the set of vertices that have an outgoing link to v, and $L(v_i)$ is the number of outgoing edges from v_i.

$$PR(v_i) = \frac{1-d}{N} + d \sum_{v_j \in M(v_i)} \frac{PR(v_j)}{L(v_j)} \tag{1}$$

The formula consists of two terms, the first term represents the random jump component whereas the second term represents the walk component. Intuitively, the jump component represents that a user will select a random page from the total number of pages with probability $(1-d)$. The walk component shows given that the user is at a certain page, they will select a random outgoing link from the current page with uniform probability with probability d. The damping factor d assures that the algorithm will converge to a distribution.

Algorithm 1 PageRank Serial algorithm

Require: $PR(v_i)^{(0)} = \frac{1}{N} \forall i \in V$
  while error $> \epsilon$ **do**
    for each node $v_i \in V$ **do**
      for each node $v_j \in M(v_i)$ **do**
        sum += $\frac{PR(v_j)^{(k-1)}}{L(v_j)}$
      end for
      $PR(v_i)^{(k)} = \frac{1-d}{N} + d \times$sum
    end for
    error = 0
    for each node $v_i \in V$ **do**
      error += $|PR(v_i)^{(k)} - PR(v_i)^{(k-1)}|$
    end for
  end while

2.2 OpenMP Parallelization

We use OpenMP to parallelize the PageRank algorithm on a multi-core system. There is a `while` loop and two `for` loops that we can parallelize in the algorithm. `while` loops terminate on checking a condition and hence cannot be parallelized trivially using OpenMP. On the other hand, the for loops don't have a race condition and hence can be parallelized by using the OpenMP directive `#pragma omp parallel for`. This algorithm is shown in 2. This does not help us in achieving high speedup since creating and destroying threads in each iteration of the while loop is an intensive task with high overhead. We also need to use `reduction` to calculate the error in parallel.

To overcome respawning of threads, we can instead use work replication to then spawn the threads outside the `while` loop such that each thread runs the `while` loop. This allows us to run all the `for` loops in parallel without having to spawn or destroy them. In this parallelization technique, we need to ensure that the error is reset only by a single thread. This can be done using the `#pragma omp single` directive in OpenMP. This parallelization technique can be implemented using the algorithm described in 3.

Moreover, instead of using `single` and having `error` as a shared variable, we can also use `error` as a private variable for each thread and perform the iterations in the `while` loop in each thread,

Algorithm 2 Parallel PageRank v1

Require: $PR(v_i)^{(0)} = \frac{1}{N} \forall i \in V$
  while error $> \epsilon$ **do**
    # pragma omp parallel for
    for each node $v_i \in V$ **do**
      for each node $v_j \in M(v_i)$ **do**
        sum += $\frac{PR(v_j)^{(k-1)}}{L(v_j)}$
      end for
      $PR(v_i)^{(k)} = \frac{1-d}{N} + d \times$sum
    end for
    error = 0
    # pragma omp parallel for reduction(+:error)
    for each node $v_i \in V$ **do**
      error += $|PR(v_i)^{(k)} - PR(v_i)^{(k-1)}|$
    end for
  end while

Algorithm 3 Parallel PageRank v2

Require: $PR(v_i)^{(0)} = \frac{1}{N} \forall i \in V$
  # pragma omp parallel
  while error $> \epsilon$ **do**
    # pragma omp for
    for each node $v_i \in V$ **do**
      for each node $v_j \in M(v_i)$ **do**
        sum += $\frac{PR(v_j)^{(k-1)}}{L(v_j)}$
      end for
      $PR(v_i)^{(k)} = \frac{1-d}{N} + d \times$sum
    end for
    # pragma omp single
    error = 0
    # pragma omp for reduction(+:error)
    for each node $v_i \in V$ **do**
      error += $|PR(v_i)^{(k)} - PR(v_i)^{(k-1)}|$
    end for
  end while

with worksharing for the `for` loops. This can be done using the `#pragma omp flush` directive in OpenMP. It allows us to make sure that the value of a variable is the same in all the threads. Hence, for each thread we check if the error is less than a threshold and once we reach that state, we flush the value of the error so that all threads can exit the while loop. This algorithm is described in 4.

Algorithm 4 Parallel PageRank v3

Require: $PR(v_i)^{(0)} = \frac{1}{N} \forall i \in V$
  # pragma omp parallel private(error)
  while error $> \epsilon$ **do**
    # pragma omp for
    for each node $v_i \in V$ **do**
      for each node $v_j \in M(v_i)$ **do**
        sum += $\frac{PR(v_j)^{(k-1)}}{L(v_j)}$
      end for
      $PR(v_i)^{(k)} = \frac{1-d}{N} + d \times$sum
    end for
    # pragma omp for reduction(+:error)
    for each node $v_i \in V$ **do**
      error += $|PR(v_i)^{(k)} - PR(v_i)^{(k-1)}|$
    end for
    if error $< \epsilon$ **then**
      # pragma omp flush(error)
    end if
  end while

3 Results

The above mentioned strategies were implemented on the LiveJournal graph and the Google Web Graph provided by the SNAP dataset[2]. For the sake of brevity, only the results for Algorithm 4 are discussed below. The code for all the strategies can be found at `https:github.com/chahak13/pagerank`. Table 1 shows the speedup achieved on one node on Frontera. As can be seen, we achieve 15x speedup on the LiveJournal dataset. Furthermore, results in Table 1 show that the web graph is not big enough for us to attain considerable speedup. For reference, the Google web graph has 875,713 nodes and 5,105,039 edges whereas the LiveJournal graph has 4,847,571 nodes and 68,993,773 edges. Figure 1 shows the speedup for the LiveJournal graph with the default settings in OpenMP.

The serial version of the algorithm has time complexity $\mathcal{O}(EN)$ per iteration for a graph with E edges and N vertices. On parallelizing with OpenMP, we can reduce this time complexity to $\mathcal{O}(EN/p)$ where p is the number of processors that we use. This provides us with good scaling upto $p = N$. Furthermore, as long as we give the right number of processors, each chunk is of the same size and hence the load is balanced throughout all the processors. This leads us to believe that various scheduling strategies would also affect the speedup that we can achieve. This can be seen in 2 which looks at the effect of different scheduling strategies on the LiveJournal graph.

As we can see, using an optimal scheduling strategy really improves the speedup. For example, the `static` and `dynamic` scheduling with the default options give 4x and 2x speedup respectively, but if we provide chunk size of the order of 3000, which almost equally divides the number of nodes,

Table 1: Running time (s) and speedup to calculate the Pagerank values on LiveJournal graph and Google Web graph with the default OpenMP settings.

Number of cores	LiveJournal		Google Web	
	Time	Speedup	Time	Speedup
1	69.61	1.	9.45	1.
2	56.34	1.236	12.33	0.77
4	41.60	1.673	10.69	0.884
8	26.21	2.656	8.90	1.061
16	15.52	4.485	7.99	1.181
32	8.86	7.857	7.69	1.229
56	4.45	15.654	7.81	1.210

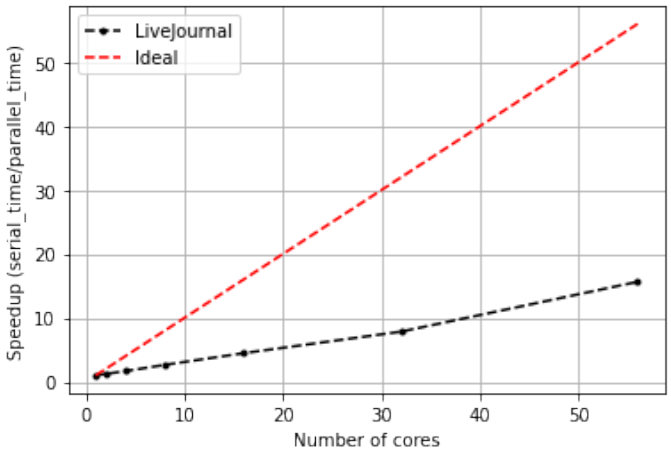

Figure 1: Speedup curve for LiveJournal graph for varying number of cores.

Table 2: Running time (s) and speedup on the LiveJournal graph for various scheduling techniques for 16 cores

Scheduling	Time	Speedup
`static`	15.6576	4.4457644
`static (3000)`	5.4237	12.834412
`static (30000)`	7.33855	9.4855251
`dynamic`	25.3556	2.7453501
`dynamic (3000)`	5.18849	13.416235
`dynamic (30000)`	5.2179	13.340616
`guided`	15.8108	4.4026868
`guided (300)`	16.1834	4.3013211
`guided (3000)`	15.6584	4.4455372
`dynamic (3000), 56 thread`	1.56316	44.531590

we see that we achieve an almost ideal speedup of 12x and 13x respectively. Figure 2 shows the speedup achieved for `dynamic` scheduling with chunk size 3000. As we can see, this is much closer to ideal speedup that we would like to achieve and we get good scaling.

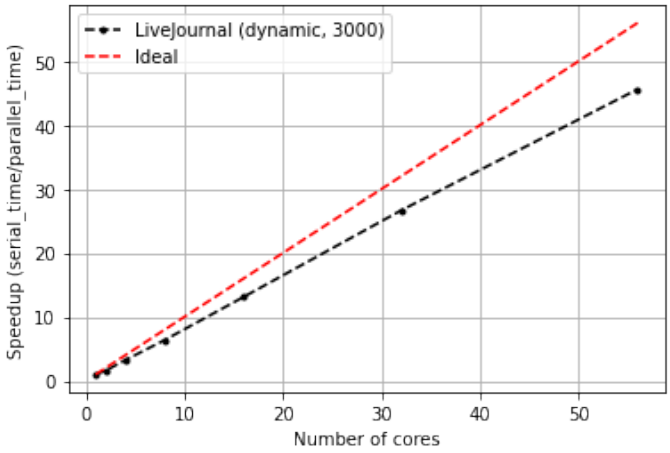

Figure 2: Speedup curve for LiveJournal graph for varying number of cores with dynamic scheduling and chunksize 3000

4 Conclusion

We saw that for a large enough graph, we can improve the runtime of the PageRank algorithm considerably using OpenMP. We performed various experiments based on number of processors and scheduling to test the load balancing and achieve high speedup. As can be seen from the speedup, the PageRank algorithm is a good candidate to implement in parallel, and this in conjunction with it's importance and usability makes it a very useful algorithm in everyday life.