PageRank Parallel Implementation

Kanak Agrawal

E-mail: `kanak@cse.iitb.ac.in`

Shraddheya Shendre

E-mail: `shraddheya@cse.iitb.ac.in`

Harshith Goka

E-mail: `harshith@cse.iitb.ac.in`

Abstract. We have done parallel graph analysis using matrix algebra, specifically ranked web pages using Google's PageRank algorithm [1]. The graph of the model web was analyzed in parallel by distributing the work done on different graph nodes to different workers. We have implemented the shared memory model ourselves in CUDA and have subsequently profiled it. For the distributed memory model, we have profiled the implementation given here [2]. As the web graph is very large, we would want to speed the process of calculating the PageRank.

1. Introduction

PageRank is an algorithm used by Google Search to rank websites in their search engine results. The PageRank algorithm outputs a probability distribution used to represent the likelihood that a person randomly clicking on links will arrive at any particular page. PageRank can be calculated for collections of documents of any size. It is assumed that the distribution is evenly divided among all documents in the collection at the beginning of the computational process. The PageRank computations require several passes, through the collection to adjust approximate PageRank values to more closely reflect the theoretical true value. Viewed mathematically, PageRank is nothing but the principal Eigen vector of a sparse matrix which can be computed using any of the standard iterative methods. In this work, we have parallelized Power method for computing PageRank using CUDA. This has been done previously by Praveen et al [3].

We have also profiled an existing MPJ (a Java implementation of a message passing library similar to MPI) [4] application for analyzing the timing of the algorithm.

2. Details of the work done

2.1. CUDA implementation

We started out with implementing the CUDA shared memory model on a GPU using **Algorithm 1**. It involves

(i) computing the product of a sparse matrix and a dense vector,

(ii) finding the norm of a vector, and

(iii) adding two vectors.

The reason to use sparse matrix representation is to save memory. In real life, number of nodes in a web graph are in billions and hence it is impossible to use a normal dense matrix. We used CUSPARSE and CUBLAS libraries provided by the CUDA SDK for doing the above mentioned calculations since efficient sparse matrix algorithms are difficult to implement. The list of operations involved in Power method for computing PageRank and their equivalent CUSPARSE/CUBLAS library calls used in our implementation are as follows

(i) $\boldsymbol{y} = \alpha P^T \boldsymbol{x} \rightarrow$

`cusparseDcsrmv()`

to perform the product of a sparse matrix P^T and a dense vector $\boldsymbol{x}$.

(ii) $\| \boldsymbol{x}^k \| \rightarrow$

`cublasDasum()`

to calculate the 1-norm of the dense vector $\boldsymbol{x}^k$.

(iii) $\boldsymbol{x}^{k+1} = \boldsymbol{x}^{k+1} + \omega \boldsymbol{v} \rightarrow$

`cublasDaxpy()`

to update the vector $\boldsymbol{x}^{k+1}$.

Algorithm 1 Find the final PageRank values using the Power method

```
Matrix P, the adjacency matrix representing the web graph
Initial vector $\boldsymbol{x_0}$, taken with all nodes with uniform weights
Damping factor $\alpha$, the probability, at any step, that the person will continue
Personalization vector $\boldsymbol{v}$, which is equal to $\frac{1}{n}\boldsymbol{e}$, where $\boldsymbol{e}$ is a vector with all ones
repeat
    $\boldsymbol{x}^{k+1} = \alpha P^T \boldsymbol{x}^{k+1}$
    $\omega = \| \boldsymbol{x}^{k+1} \| - \| \boldsymbol{x}^k \|$
    $\boldsymbol{x}^{k+1} = \boldsymbol{x}^{k+1} + \omega \boldsymbol{v}$
until $\delta < \epsilon$
return $\boldsymbol{x}^{k+1}$
```

2.2. Profiling of the MPJ implementation and the serial code implementation

We profiled the implementation given in [2] for analyzing the timing of the application, the results of which are reported below in the **Table 1**.

We also developed a serial version of the **Algorithm 1** to compare the results (for correctness and timing comparison), the results of which are reported below in the **Table 1**.

We also computed the Karp-Flatt measure for the data-set with $500,000$ vertices on the shader machine, the results of which are tabulated in the **Table 2**. We can see that the measure is steadily increasing with the number of processors. Since e is steadily increasing, the inherently serial portion is not the bottleneck. The possible sources of inefficiency can be

(i) Process start-up time,

(ii) Process synchronization time,

(iii) Idling (Imbalanced workload), or

(iv) Architectural overhead.

Some shortcomings of the application code are as follows

(i) `communicator.Allreduce(danglingArray,0, sumDangling ,0, 1, MPI.DOUBLE, MPI.SUM);`
The information related to the number of dangling nodes in the graph is used only by the process 0, so there is no need of using Allreduce, simple Reduce would have sufficed. Allreduce is basically Reduce followed by a Bcast, which might lead to inefficiency.

(ii) `communicator.Allreduce(localRV,0, FinalRVT,0,totalNumUrls, MPI.DOUBLE, MPI.SUM);`
Using Allreduce is not required here as Allgather could have been used and localRV need not have

Table 1. Comparison of timings(in ms) on different data-sets

Number of vertices	Number of edges	Serial	CUDA	MPJ
10	19	0.1	469.83	116
10000	38598	28.6	460.48	168
10000	53711	25.4	433.80	182
50000	216015	95.5	537.35	369
150000	796966	323.7	721.48	835
200000	2134806	800.2	1561.08	1269
250000	2755675	1025.5	1968.82	1599
300000	3396307	1266.4	2338.25	1851
350000	4039301	1494.1	2769.11	2188
400000	4690576	1737.1	3174.50	2471
450000	5351877	1971.4	3589.49	2762
500000	6007221	2224.4	4139.48	3026
20000000	79999989	68459.9	45215.38	
100000000	199999998	314330.8	158517.0	

Table 2. Karp-Flatt measure for 500,000 vertices

p	T_p	T_s	Ψ	e
1	4389	2224.4	0.5068124858	0
2	3186	2224.4	0.6981795355	0.4661481748
3	2653	2224.4	0.8384470411	0.5728985594
4	2517	2224.4	0.8837504966	0.6611558173
5	2386	2224.4	0.9322715842	0.6981190433
6	2649	2224.4	0.8397130993	0.8535135567
7	2518	2224.4	0.8833995234	0.8478287197
8	2689	2224.4	0.8272220156	0.948382148
9	2723	2224.4	0.8168931326	0.9893681969
10	2896	2224.4	0.7680939227	1.0817317029
11	3002	2224.4	0.7409726849	1.1442439302
12	3136	2224.4	0.7093112245	1.2159446243
13	3117	2224.4	0.7136349054	1.2224803125
14	3404	2224.4	0.65346651	1.3546666104
15	4020	2224.4	0.5533333333	1.6245247657
16	6138	2224.4	0.3623981753	2.5283398051
17	6287	2224.4	0.3538094481	2.6047591699
18	6246	2224.4	0.3561319244	2.5994819521
19	6869	2224.4	0.3238317077	2.8756346758
20	6758	2224.4	0.3291506363	2.8387165078

References

[1] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd, The pagerank citation ranking: Bringing order to the web., Technical report, Stanford InfoLab, 1999.

[2] `https://github.com/jayesh15111988/PageRankParallelAlgorithm`

[3] Praveen, K., Vamshi Krishna, S. Balasubramanian, and P. K. Baruah. "Cost efficient pagerank computation using GPU." (2012).

[4] Shafi, Aamir, Bryan Carpenter, and Mark Baker. "Nested parallelism for multi-core HPC systems using Java." Journal of Parallel and Distributed Computing 69, no. 6 (2009): 532-545.