Parallel Breath First Search on Distributed Memory

Aniruddh Adkar, Anuj Sharma
Department of Computer Science and Engineering,
State University of New York at Buffalo,
Buffalo, NY 14260, USA

ABSTRACT

Graphs and their traversal is becoming significant as it is applicable to various areas of mathematics, science and technology. Various problems in fields as varied as biochemistry (genomics), electrical engineering (communication networks), computer science (algorithms and computation) can be modeled as Graph problems. Real world scenarios including communities their interconnections and related properties can be studied using graphs.

So fast, scalable, low-cost execution of parallel graph algorithms is very important. In this implementation of parallel breadth first search of graphs we implemented Parallel BFS algorithm with 1-D partitioning of graph as described in paper written by Aydin Buluc and Kamesh Madduri[1]. We were able to optimize communication for local buffers to reduce execution time.

I. 1. INTRODUCTION

Fast and scalable graph algorithms are critical for a large range of application domains with a vital impact on both national security and national economy which includes social media, logistics, e-commerce etc. Parallel scalable graph algorithms are challenging in scenarios where we have distributed memory architecture, as message passing includes overhead of message passing between multiple nodes.

We got inspired for this work from the fact that large and often scale-free, graphs are ubiquitous in communication networks and social networks like that of Facebook. For traversal on different types of graphs we used start graph, Erdős–Rényi graph [2] and small world graph. Though our results by showing success of this parallel BSF algorithm we hope to show potential of this proposed algorithm to wide range of parallel graph problems. We tried our level best to bring out any limitations of this algorithm on certain types of graphs wherever possible.

Another force of motivation for our research was our Professor Dr. Jaroslaw Zola who demonstrated the power of HPC and parallelism in various algorithms along with guidelines to leverage power of massively parallel compute nodes and apply on different type of existing problems.

II. 2. SHARED MEMORY IMPLEMENATION

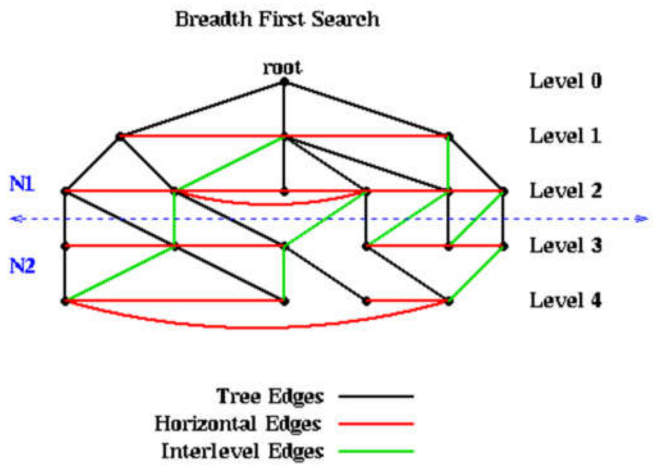
Serial algorithms work on single machine with large enough computing resources including memory and processor power.
Number of such serial implementations are already available. Various challenges included in implementation of parallel algorithm not only involved partitioning and distribution of vertices among processors but also communication among all processors, updating processor local data with others and keeping global data update for next iteration of algorithm. It also involved collecting, merging and distributing local data buffers in efficient way to reduce communication time. Various approaches used in tackling such challenges can be seen in depth further.

2.1 Partitioning

We used 1-d partitioning which partitions graph in following steps:

1. Choose start vertex assign it to first processor.
2. Calculate all neighbors of start vertex.
3. In next step divide all n neighbors of start vertex among p processors (n/p).

In this way after completion of each phase we first gather computed neighbors from all processors and partition them at the beginning of next step.
As demonstrated below:



In step 1 Level 0 only processor 1 will have root vertex. In next step Level 1 there are three vertices and processor 1, 2 and 3 each will have one vertex. In next step Level 2 there are 6 vertices and each processor from 1-6 will have one vertex. In this way partitioning of vertices will keep on happening till all vertices are traversed.

2.1 BFS Traversal

Traversal begins with single start vertex and keeps on incrementing with each step of finding all neighbors of vertex in current hand for each processor. We have refered this algorithm from paper written by Aydin Buluc and Kamesh

Madduri[1]. Algorithm followed is:

```
Input: $G(V, E)$, source vertex $s$.
Output: $d[1..n]$, where $d[v]$ gives the length of the shortest
    path from $s$ to $v \in V$.
 1: for all $v \in V$ do
 2:     $d[v] \leftarrow \infty$
 3: $level \leftarrow 1$, $FS \leftarrow \phi$, $NS \leftarrow \phi$
 4: $op_s \leftarrow find\_owner(s)$
 5: if $op_s = rank$ then
 6:     push $s \rightarrow FS$
 7:     $d[s] \leftarrow 0$
 8: for $0 \leq j < p$ do
 9:     $SendBuf_j \leftarrow \phi$            ▷ $p$ shared message buffers
10:     $RecvBuf_j \leftarrow \phi$            ▷ for MPI communication
11:     $tBuf_{ij} \leftarrow \phi$        ▷ thread-local stack for thread $i$
12: while $FS \neq \phi$ do
13:     for each $u$ in $FS$ in parallel do
14:         for each neighbor $v$ of $u$ do
15:             $p_v \leftarrow find\_owner(v)$
16:             push $v \rightarrow tBuf_{ip_v}$
17:     Thread Barrier
18:     for $0 \leq j < p$ do
19:         Merge thread-local $tBuf_{ij}$'s in parallel,
    form $SendBuf_j$
20:     Thread Barrier
21:     All-to-all collective step with the master thread:
    Send data in $SendBuf$, aggregate
    newly-visited vertices into $RecvBuf$
22:     Thread Barrier
23:     for each $u$ in $RecvBuf$ in parallel do
24:         if $d[u] = \infty$ then
25:             $d[u] \leftarrow level$
26:             push $u \rightarrow NS_i$
27:     Thread Barrier
28:     $FS \leftarrow \bigcup NS_i$                ▷ thread-parallel
29:     Thread Barrier
```

2.1 Global Computation and Communications

If we generalize whole algorithm it is comprised of two steps. First is computation performed by each processor and 2nd is communication step which is performed by all processors at the end of each iteration.

For explaining computation step say we are in nth iteration step, where total number of vertices are 8000 and we have implemented on 8 processors. So according to 1-D partitioning all these vertices will be partitioned in *(n/p)* small chunks among each processor. Now according to our example each processor will have 1000 vertices. Then each processor will perform local computation on each vertex finding all the neighbors of this vertex. Only owner i.e. processor can decide if vertex is visited or not and can assign level to it. Each processor stores the same number of local buffers as of total processors participating into BFS. In case neighbor found is such that computing processor is the owner of such neighbor then immediately processor will update its distance vector for that vertex.

After completion of computations by all processors, next step involves communication and interchanging of data between them. Every processor sends all of the local buffers except the one it owns to other owner processors. Once communication step is completed, each processor receives all the vertices it owns, such vertices are evaluated again to update their distance vector and then move on to next iteration.

III. 3. EVALUATION

Basic results of scalability for a parallel code running on parallel cluster system are *strong and weak scaling*. As strong scaling has a constant problem size and increasing processor or core count, it measures how well a parallel code can solve a fixed size problem as the size of parallel computer is increased. Whereas weak scaling has a fixed problem size per core per processor and measures the ability of parallel machine with parallel code to solve large versions of the same problem.

To evaluate our approach, we conducted empirical study of the strong and weak scaling on HPC cluster up to 64 nodes involving combinations of processors and cores. We tested on graphs consist of up to 4 Million vertexes. To check for any anomaly, we checked our data for following types of graphs:

- Star Graphs
- Erdős–Rényi Graphs
- Small World Graphs

Environment used in the implementation is described below which precedes results.

3.1 Test Environment

Test environment is CPU cluster hosted at University at Buffalo. For our algorithm execution we used from 1 to 64 nodes of cluster. While relying on CPU memory we faced certain problems like while generation of graph with 4 million vertices with distributed memory architecture we were running out of memory again and again. To tackle such problems, we divided our set of work into small chunks of works. For example, graph is generated for 10,000 vertices and then concatenated locally to get resulting graph of 4 million vertices. Runtime environment used was Open MPI which is implementation of MPI standard. It keeps tracks of all participating processors, their communicator group and provides us with very handy constructs like MPI_Bcast (for broadcasting), MPI_Isend (Non-blocking send), MPI_Recv (for receiving data from any rank), MPI_Gatherv (for uniting data from various ranks/processors). For graph generation we used BOOST library with C++ language. This library provides variety of graph generators which allowed different type of sample data for our testing environment.

While computations all data is stored at CPU memory and all computations for example computing of all neighbors of any vertex has been done on CPUs. On provided HPC cluster we submitted jobs to SLURM job scheduler which optimizes job scheduling and further prioritize jobs based on their resources usage and priority.

IV. 4. RESULTS

Scalability results of our algorithm demonstrated the underlying promise of our approach. They basically contrasted computation costs vs communication costs in some cases very distinctively.

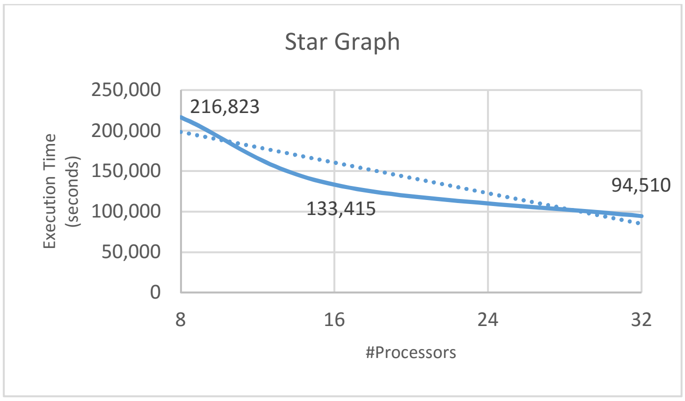
Types of graphs we implemented our algorithm on are:

- Star Graphs
- Erdős–Rényi Graphs
- Small World Graphs

4.1 Scalibility Results with Star Graphs

Parameters defined for star graphs are as follows:

No. of Processors	No. of vertices	Execution Time (seconds)
8	4,000,000	216,823
16	4,000,000	133,415
32	4,000,000	94,510

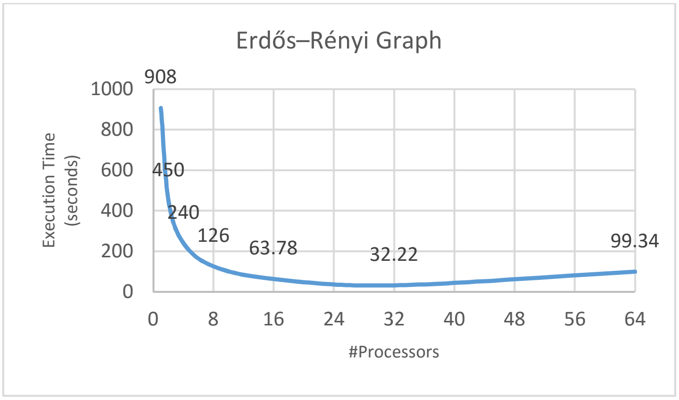


We were able to achieve these three computations for 4,000,000 vertices and processors 8, 16 and 32. For nodes like 1, 2, and 4 as stated in next sub-section CCR discarded jobs due to time limit of 72 hours.

It showed strong scaling as we kept problem size fixed and it showed decrease in execution time as we increased number of processors.

4.2 Scalability Results with Erdős–Rényi Graphs

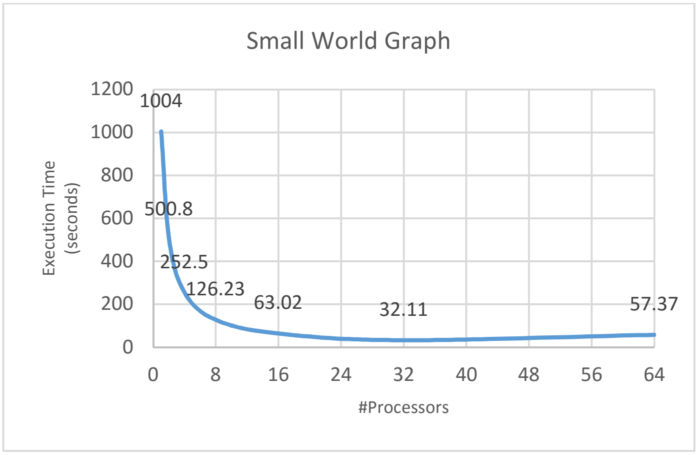
No of Processors	No. of Vertices	Execution Time (seconds)
1	1,00,000	908
2	1,00,000	450
4	1,00,000	240
8	1,00,000	126
16	1,00,000	63.78
32	1,00,000	32.22
64	1,00,000	99.34



Results for Erdős–Rényi graphs shows very good scalability till 32 processors. After 32 processors when we checked it for 64 processors it increased which is caused by domination of message passing overhead between processors over computation time. So it proved communication overhead limit for such type of graphs where simply increasing number of processors won't result into execution time reduction and communication performance affects computation.

4.3 Scalability Results with Small World Graphs

No of Processors	No of Vertices	Execution Time (seconds)
1	1,00,000	1004
2	1,00,000	500.8
4	1,00,000	252.5
8	1,00,000	126.23
16	1,00,000	63.02
32	1,00,000	32.11
64	1,00,000	57.37



Similar results we got for small world graphs as it showed very good scalability till 32 processors. After 32 processors when we checked it for 64 processors it again increased which is caused by domination of message passing overhead between processors over computation time as was in the case of Erdős–Rényi graphs.

4.4 Issues faced with testing environment:

1. Initially unable to generate 4 million nodes on CCR (our testing environment) due to out of memory error stated as bad_alloc on our test runs.

2. CCR took 3-4 days to bring our jobs from pending state to running state and if any memory error occurred which did in our case as mentioned above, we got to know about it after these many days which resulted in testing of only star graphs for 4 million vertices and other graphs for less number of vertices due to time contraint.

3. Time limit on job execution imposed by CCR is 72 hours but in case of our execution of 4 million nodes on 1, 2 and 4 processors took more than that it resulted in job discarded by CCR.

V. 5. OPTIMIZATIONS

For this project we have taken paper [1]. In their original algorithm, we have made two optimizations, which helped further reduce execution time. The optimizations were as follows:

Algorithm optimizations:

1. Algorithm in described paper involves adding current vertex to local buffer which will sent to the owner processor in the communication step. However, we added conditional check to see if current processor is owner of neighbor vertex and updated distance vector in the very same step. This optimization resulted into relatively lower buffer size
and will prove effective over large data communication.

2.Our optimization lies in the very step of all to all communication step where algorithm in paper [1], all the local buffers are aggregated into one local buffer each processor which are then scattered among different processors. We were able to send local buffers to other processors directly and saved communication overhead of sending, merging and again scattering. Processors are able to update distance vectors after above communication optimization step. Such optimization will enhance the working time of algorithm and communication overhead which increases linearly with the number of processors can be saved. So these were two major optimizations which reduces execution time furthermore and help us achieving better results.

Programming optimizations:

1. When all processors after traversing through neighbors of vertices send populated local buffers to other processors so that each processor should get vertices of which it is the owner.

2. At end of algorithm when we were merging all NS vectors to form new FS vector at master for next iteration.

Firstly we were using MPI_Send and MPI_Recv which were taking 32 seconds when we ran for 10,000 vertices then we replaced it with MPI_Gatherv at both steps, same run with same input and number of processor resulted in 11 seconds of execution time.

VI. 6. CONCLUSIONS AND FUTURE WORK

With the results achieved in graphs in previous section the fundamental understanding that we now have of scalability properties of algorithms is perhaps as important as the software implementation and results obtained.

As our results has shown that it is possible to achieve a scalable solution for challenging BFS graph problems on distributed memory architecture, the next big challenge would be in developing softwares which are portable across different architectures. But still after the results we fell somewhere we can apply number of tweaks to algorithm to make it much more efficient in terms of memory usage, execution time and many other factors. For example, by using some other method of partitioning vertices it might be possible to get much more efficient vertices distribution among processors. Also by inventing some other methods of graph production like by reading it from file it might be possible to make processors from from graph production and in memory graph storage. Which could further free CPU for more computations and might give further reduces execution times.

In this limited period of time we tried our level best to produce as optimal algorithm as possible but surely in future will apply other tricks and tweaks to algorithm which could help reduce its execution time further.

VII. ACKNOWLEDGEMENTS

We wish to express our sincere gratitude to our Professor Dr. Jaroslaw Zola who has given us such wonderful opportunity of working with project related to parallel BFS implementation on distributed memory architecture. Professor's valuable knowledge was always with us at each step in our development.

VIII. REFERENCES

[1] Parallel Breadth-First Search on Distributed Memory Systems – Aydin Bulic, Kamesh Madduri.
[2] On Random Graphs I - Erdős, P.; Rényi, A. (1959) http://www.renyi.hu/~p_erdos/1959-11.pdf
[3] http://www.open-mpi.org/doc/
[4] https://computing.llnl.gov/tutorials/mpi/
[5] http://mpitutorial.com/tutorials/.