Scalable Parallel Minimum Spanning Forest Computation Project Report

Pratheek Reddy Amireddy (112073273)

Abstract

The proliferation of data in graph form calls for the development of scalable graph algorithms that exploit parallel processing environments. One such problem is the computation of a graph's minimum spanning forest (MSF). Past research has proposed several parallel algorithms for this problem, yet none of them scales to large, high-density graphs. In this project a novel, scalable, parallel MSF algorithm for undirected weighted graphs has been implemented. The algorithm leverages Prim's algorithm in a parallel fashion, concurrently expanding several subsets of the computed MSF. The project focuses on minimizing the communication among different processors without constraining the local growth of a processor's computed subtree. In effect, the project achieves a scalability that previous approaches lacked. OpenMP has been used to implement the algorithm and is run on the Stampede2 supercomputer using real and synthetic, sparse as well as dense, structured and unstructured graph data.

Introduction

A spanning tree of a connected graph G is an acyclic subgraph of G that connects all vertices of G. The Minimum Spanning Tree (MST) problem calls to find a spanning tree of a weighted connected graph G having the minimum total weight [1]. In case the graph is not connected, i.e. consists of several connected components, the problem is generalized to finding the Minimum Spanning Forest (MSF), i.e. a subgraph containing an MST of each component.

Past research has proposed several sequential MST algorithms, starting out with Boruvka's seminal work [2]. Highlights of this research are Kruskal's [2], Prim's [3], and the Reverse-Delete [4] algorithms. Other MST algorithms may have lower asymptotic complexity, but larger hidden constants [5]. While existing algorithms are reasonably efficient, modern applications call for algorithms that can scale well to very large and high-density graphs, including complete graphs, as in the case of computing the MST in Eucledian space, which finds applications in hierarchical clustering [6]. Parallel processing comes into play to achieve this objective. In a similar spirit, Vineet et al. have already offered a data parallel version of Boruvka's MST algorithm adopted ˙ for a GPU [7]. However, Boruvka's algorithm is ill-chosen if the objective is to solve the MST computation in a scalable manner for dense graphs , as its performance is known to deteriorate in comparison to Prim's algorithm as graph density grows [8].

To date, two parallel adaptations of Prim's algorithm have been proposed [9, 10]; however, out of these, [9] allows for limited parallelism, as it does not allow two growing subtrees to touch each other, while [10] necessitates costly inter-processor communication to merge subtrees when they do get in contact. Thus, there is still a need for a size-scalable and density-scalable GPU-based MST computation algorithm.

In this project the need has been responded: a novel Parallel MSF Algorithm (PMA)) for undirected weighted graphs, which adapts Prim's algorithm while eschewing the drawbacks of [9] and [10]; in contrast to [9], it allows for full-scale flexible parallelism; still, unlike [10] it raises a much lower communication overhead.

Related Work

The MSF problem was first formulated and solved by Boruvka [2]. Subsequently, three altervative algorithms were proposed, namely Kruskal's [2], Prim's [3] and the Reverse-Delete [4] algorithm. All these algorithms exploit two properties of the MSF ; the cycle property, which maintains that the heaviest edge in a cycle does not belong to the MSF; and the cut property, which maintains that, for every subset of the graph's vertex set, $C \subset V$, the lightest edge with one vertex in C and the other vertex in $V \setminus C$ belongs to the MSF.

The Reverse-Delete algorithm, based on the cycle property, iteratively removes the heaviest edge that does not break any graph component's connectivity. Likewise, Kruskal's algorithm iteratively adds the lightest edge that does not introduce a cycle. Prim's algorithm selects an arbitrary vertex and iteratively inserts the lightest edge from the current subtree to an unvisited vertex, based on the Cut property. Boruvka's algorithm differs from Prim's ˙ algorithm in starting from all vertices at once, and expanding all running subtrees at each iteration.

Several theoretical results highlight potential parallelism in MST computation, most of them do not lead to efficient practical algorithms as they incur large constant factors [7]. Thus, most practical parallel MST algorithms are merely parallelized versions of classical sequential algorithms, adapted for specific hardware architectures or programming models. As Boruvka's algorithm is ˙ naturally prone to parallelization, most of these works adapt that algorithm, sometimes in combination with Kruskal's or Prim's algorithm [7]. Chung and Condon [11] propose a parallel version of Boruvka's algorithm for asynchronous, distributed-memory ma- ˙ chines. Boruvka's algorithm is divided into five steps and parallelize each step. Dehne and Gotz propose the Boruvka Mixed Merge (BMM) algorithm, which lets each processing unit find a local MST for its stored edges sequentially, and then prunes and merges the resulting partial MSTs at a single unit using a balanced D-ary tree.

The work that most related to this project, [9], stands between Prim's and Boruvka's algorithms. This MST-BC algorithm lets each processing unit run Prim's algorithm starting from different vertices simultaneously, marking the vertices in its own MST and coloring all neighbors of marked vertices. These local MSTs grow until a conflict occurs, i.e. one unit reaches a vertex marked or colored by another unit. Then, the conflicting unit starts building a new MST from another unvisited vertex. The algorithm terminates when all vertices are either colored or marked and merges the resulting local MSTs. MST-BC degenerates to Boruvka's algorithm for ˙ P processing units equal to the number of vertices n, and to Prim's algorithm for one only processing unit. The algorithm in [10] is a relaxed version of MST-BC for the transactional memory model, differing therefrom in the way it treats conflicts. When a conflict occurs, the two parties involved switch to a Merge state to resolve the conflict (see state transition diagram in Figure 1), with one unit appending the other's MST to its own, while the other starts building a new tree from another unvisited vertex; on a connected graph, this algorithm ends up having only one thread working on the final MST, hence reduces parallelism.

Vineet et al. recently adapted Boruvka's algorithm for the GPU, using parallel primitives. This · algorithm provides the current state of the art for GPU-based MSF computation in terms of efficiency; however, it cannot scale to large numbers of edges.

Motivation

Most existing approaches to parallel MSF computation share a similar intuition: : They build different trees in parallel, and, when conflicts occur (i.e. different trees run into each other), they merge the components and start over. However, this strategy is not equally efficient on all types of graphs; MST-BC [8] fares well with sparse graphs, its relaxation [10] manages well graphs with large diameter (thanks to a heuristic that may result in less conflicts), [12] does well only with sufficiently dense graphs, and , being an adaptation of Boruvka's algorithm, is challenged by high graph density; besides, the question of finding the MSF of a non connected graph is mostly ignored. Most significantly, these approaches tend to be cautiously conservative when expanding their trees, as they have to be alert to potential conflicts, invoking too many redundant iterations that deteriorate their performance. Unfortunately, the attempt to alleviate this conservatism in , by merging the trees when a conflict occurs, ends up substituting that problem with another, as it raises the inter-processor communication cost and results into an unbalanced load as progressively fewer processors are left building fewer MSTs, forsaking the benefits of parallelism.

Goals

- To implement all the three algorithms using openMP: minPrim Parallel MSF Algorithm, sortPrim Parallel MSF Algorithm and hybridPrim Parallel MSF Algorithm discussed in [13]
- Compare the Running times of all the algorithms and also the speedup of Parallel MSF Algorithm with respect to the serial implementation of Prim's algorithm

Implementation

The input graph is represented as an adjacency list. This data structure consists of a vertex list (an array of vertices that stores for each vertex, its index and a pointer to its adjacent vertices in the edge list) and an edge list (that stores for each edge, its weight and the index of the endpoint vertex of the edge).

Partial Prim Implementation

Each processor running Partial Prim needs to pick one unvisited vertex and grow a tree. When a growing tree is terminated, another unvisited vertex is picked. To reduce the conflict among P processors in picking vertices we divide the set of vertices into P partitions of equal size. Each processor only picks vertices in its own partition to start growing a tree. This operation is still done atomically since a processor growing its tree might visit a vertex in another's partition. The most important step in PP is to pick the lightest outgoing edge to grow from those in the list Q containing the vertices in a processor's current tree.

MinPMA Algorithm

The approach is to go through the list of vertices in Q, and, for each vertex u, through its adjacent edges, and pick the lightest one with a destination v ∈ Q. To check if a vertex v is in Q or not, we check whether successor[v] is the same as the root of the current tree or not. By this approach, we have to go through the list of adjacent edges of each vertex in Q up to |Q| times, thus its complexity is O(γ|E|).

Algorithm:

Input: G(V,E) : An undirected weighted graph; Q : the list of vertices in the current sub-tree;
Output: The lightest edge e(u, v) such that u ∈ Q and v ∈ Q

```
1. minW = ∞ ;
2. for Each u in Q do
3.     for Each edge e(u, v) of u do
4.         if successor[u] != successor[v] and e.weight < minW then
5.             minE = e;
6.             minW = e.weight;
7. Return minE;
```

SortPMA Algorithm

The Second Approach tries to alleviate the disadvantage of MinPMA. We first sort the list of adjacent edges of each vertex by increasing weight3 . Then, whenever we look for the lightest edge going out from vertex u, we pick the first edge that does not have a destination in Q. By recording the previously chosen edges for each vertex u in Q, we only have to go through the list of adjacent edges of each vertex at most once. Thus, the cost of this step becomes O(|E|), at the cost of sorting the edges, which takes O(|E| log(|E|)). The latter cost can render SortPMA more costly than MinPMA for small |Q| or large |E|. Thus, for very sparse graphs, MinPMA fares better than SortPMA. On the other hand, for dense graphs, PP has a low level of parallelism, and, as a result, SortPMA becomes faster as most of the work is done in the sorting step, which runs efficiently on the whole edge list.

Algorithm

Input: G(V,E) : An undirected weighted graph, with the list of adjacent edges of each vertex sorted by weight; Last : Last minimum outgoing edge found for each vertex. Q : the list of vertices in the current sub-tree;
Output: The lightest edge e(u, v) such that u ∈ Q and v ∈ Q

```
1. minW = ∞ ;
2. for Each u in Q do
3.     for Each edge e(u, v) of u starting from Last[u] do
```

```
4.          if successor[u] = successor[v] then
5.              if e.weight < minW then
6.                  minE = e;
7.                  minW = e.weight;
8.              Last[u] = e ;
9.              Break;
10. Return minE;
```

HybridPMA Algorithm

As MinPMA is efficient for certain graphs and SortPMA for others, we combine the strengths of both in a hybrid approach, HybridPMA, where we use MinPMA for the first iteration and SortPMA for the rest. The rationale for this choice is that, in the later iterations, the graph gets a lot denser as multiple vertices get unified into one.

Results

All the results have been tabulated on Stampede2. The number of cores used are 68. OpenMP has been used for the parallel implementation.

A Maximum subtree size $\gamma = 2$, is used in each of the implementation.

All the codes are present in src/

All the input graphs are present in input/
All the output graphs and the plot of run times are present in output/

The code to generate the synthetic graph is present in src/graph_generation.cpp

To run the algorithms on randomly generated synthetic graphs run the following script run.sh.
Run as follows: bash run.sh <num_of_test_cases>
example: bash run.sh 20. To run on 20 synthetically generated graphs.

Following is the table of running times for the Graphs:

Graph	Vertices	Edges	MinPMA	SortPMA	HybridPMA
Arxiv Astro Physics	19K	7.9M	2.87s	0.567s	0.124s
Penn Road Network	1M	6.17M	18.88s	10.344s	5.45s
Internet Topology	1.7M	22.2M	56.23s	22.56s	19.123s

Graph	Vertices	Edges	MinPMA	SortPMA	HybridPMA
California Road Network	1.8M	4.6M	34.788s	23.77s	18.790s

Also Synthetic graphs have been generated and their running times are compared and plotted as follows:
The following plot compares the serial prims algorithm, minPMA Algorithm, sortPMA algorithm and hybrid PMA algorithm. The performances (run times) of sortPMA and hybridPMA are close as the number of edges are less, as the number of edges are increasing the run times differ. This is due to the fact that sortPMA and hybridPMA implementations are similar as the number of edges are less i.e., the graphs are less dense.

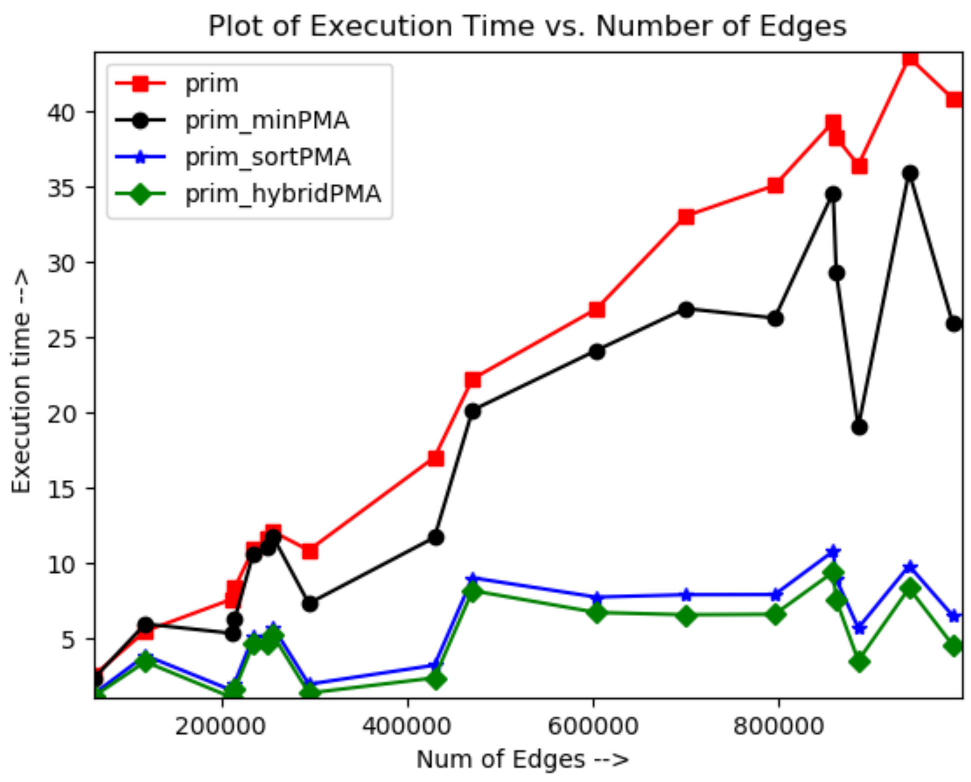

Figure1: Plot of Execution Time vs Number of edges

Future Work

- Since, CUDA has been used to implement the algorithms on GPU in the research paper. Therefore, I would like to implement the algorithms on GPU in future and compare the run times,
- The maximum subtree size parameter used in this implementation is only 2, In future I would like to compare the runtimes as I go on increase the maximum subtree size parameter.

- Speedup is an essential metric in parallel computation. I would like to increase the speedup with respect to a serial implementation of prim's algorithm.

References

[1] E. Horowitz and S. Sahni. Fundamentals of computer algorithms. Potomac, Md., Computer Science Press, 1978.

[2] J. Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. Proc. Amer. Math. Soc., 7:48–50, 1956.

[3] R. C. Prim. Shortest connection networks and some generalizations. Bell System Technical Journal, 36:1389–1401, 1957.

[4] J. Kleinberg and Eva Tardos. The minimum spanning tree problem. ´ In Algorithm Design. Pearson/Addison-Wesley, Boston, 2006.

[5] B. M. E. Moret and H. D. Shapiro. An empirical assessment of algorithms for constructing a minimal spanning tree. In DIMACS Monographs, 1994.

[6] W. B. March, P. Ram, and A. G. Gray. Fast euclidean minimum spanning tree: algorithm, analysis, and applications. In KDD, 2010.

[7] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the gpu. In HPG, 2009.

[8] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. Introduction to algorithms. MIT Press, third edition, 2009. ISBN 0070131511.

[9] D. A. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. JPDC, 66(11):1366–1378, 2006.

[10] S. Kang and D. A. Bader. An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. In PPoPP, 2009.

[11] R. E. Tarjan. Data structures and network algorithms. Society for Industrial and Applied Mathematics, Philadelphia, 1983.

[12] F. Dehne and S. Gotz. Practical parallel algorithms for minimum ¨ spanning trees. In SRDS, 1998

[13] S. Nobari, T.-T. Cao, S. Bressan, and P. Karras. Scalable Parallel Minimum Spanning Forest Computation. In Proc. of International Symposium on Principles and Practice of Parallel Programming, pages 205–214, August 2012.