Prim's Algorithm using Parallel Computation

Akshay Vaje
Department of Computer Engineering and Information Technology
Veermata Jijabai Technological Institute,Mumbai India
amvaje_b17@it.vjti.ac.in

Prathmesh Bendal
Department of Computer Engineering and Information Technology
Veermata Jijabai Technological Institute,Mumbai India
pdbendal_b17@it.vjti.ac.in

Sandip T Shingade
Department of Computer Engineering and Information Technology
Veermata Jijabai Technological Institute,Mumbai India
stshingade@it.vjti.ac.in

1 Introduction

1.1 About Paper

Paper Name:Parallel implementation of prim's algorithm using openmp
Author: Parth Rupala (University of california,Irvine)

1.1.1 Which problem is identified by the author in the paper?

Problems identified is for large and complex data, the sequential implementation of Prims MST algorithm is infeasible and slow.

Prim's algorithm is used to find MST of a graph , the serial implementation of algorithm runs in $o(n2)$, author in this paper provides parallel implementation of algorithm and compares the result with sequential algorithm

1.1.2 Give the appropriate example for the problem mentioned in the paper

MST can be used in the design of distributed computer networks, wiring connections, VLSI layout and routing, transportation networks. It has also indirect applications in the field of image processing, network reliability, and speech recognition, in all the applications mentioned above the number of nodes tend to be of 10^4 so calculating MST with sequential approach becomes computationally expensive

1.1.3 What is the importance/need to find a solution/Analysis of this problem?

The traditional implementation of Prims Algorithm using sequential computation is very slow as the sequential computation uses one processor to run the given task. The other task have to wait for the processor to get free. This increases waiting time for the process and the overall latency of the algorithm increases.

Parallel implementation uses multiple processors or multiple threads to compute the task. Multiple processors helps to run the process parallely, reducing the waiting time and hence reducing the overall latency.

1.2 Related Work (What is the relation of the problem with previous theory and research) ?

Several parallel versions of Prim's algorithm are available. Grama et. al. [1] mentioned that it is very difficult to choose 2 vertices at the same time and thus it is very difficult to parallelize the outer loop. But one can parallelize the method for finding minimum weight edge in cut. The adjacency matrix is distributed in 1-D block fashion. Each processor has (n x n/p) of the adjacency matrix and n/P of Key Vector. Each processor finds locally nearest node, and the global minimum is obtained using all-to-one reduction. The processor containing global minimum then performs one-to-all broadcast, and all the processors then update the value of their respective Key Vector.

Setia et. al.[2] uses cut property of graph to grow minimum spanning tree simultaneously in multiple processes and uses merging mechanism when processes collide. After collision, one process merges with other and continues to grow tree from there and another thread picks another node randomly and grows a new tree. To ensure that all the processes do significant amount of work before terminating, this algorithm also uses load balancing technique.

Loncar et. al. [3] uses parallel algorithm that target message passing parallel machine with distributed memory. Primary characteristic of this architecture is that the cost of interprocess communication is high in comparison to cost of computation. The goal of the algorithm is to develop algorithms which minimize communication, and to measure the impact of communication on the performance of algorithms. the primary interest were graphs which have significantly larger number of vertices than processors involved in computation. Since graphs of this size cannot fit into a memory of single process, The author used simple partitioning scheme to divide the input graph among processes and considered both sparse and dense graphs.

Wang et. at. [4] proposed a minimum spanning tree algorithm using Prim's approach on Nvidia GPU under CUDA architecture. By using new developed GPU-based Min-Reduction data parallel primitive in the key step of the algorithm, higher efficiency is achieved. Data parallel primitives are common fundamental parallel operations when developing parallel algorithms. In the GPU parallelization process of serial algorithm aimed at irregular problem, the use of data parallel primitives can achieve crucial performance improvement. Reduction data parallel primitive is a kind of parallel operation which processes a group of data elements and gets a single value, such as sum, min and max. Reduction primitive is a common process in parallel program design. Many situations of parallel computing involves comparing or summarizing all results of different threads, such as gather computing data or draw a specific value. Reduction primitive is the best choice in these situations. There are many studies about Reduction primitive under traditional parallel computing architectures. Aiming at the goal of paralleling the key step of finding minimum weighted edge in Prim's MST algorithm, the author improve and extend the GSR primitive, and design the GPU Min-Reduction primitive under CUDA architecture.

1.3 Contributions from the base paper can be identified from the points given below.

Author in the paper exploit the min cut property of mst, . For any cut in the graph, the edge with the smallest weight in the cut belongs to all MSTs of the graph. Such a minimum weight edge in cut is known as a light edge. If there are multiple edges with same minimum cost, at least one of them will be in the MST, Author used multiple threads to find minimum weight edge in a given cut between the sets of vertices which are already included in MST

and vertices that are not included in MST but present in the graph. author also uses multiple threads to update the key property of vertices. OpenMP API available for C/C++ author used to parallelize Prim's MST algorithm.

1.3.1 Investigate the new proposed algorithm/strategies for our problems.

Min-Reduction data parallel primitive under CUDA architecture on GPU is another such technique that we can use to run our Prims Algorithm parallely. Reduction data parallel primitive is a kind of parallel operation which processes a group of data elements and gets a single value, such as sum, min and max. Reduction primitive is a common process in parallel program design. Many situations of parallel computing involves comparing or summarizing all results of different threads, such as gather computing data or draw a specific value. Reduction primitive is the best choice in these situations. There are many studies about Reduction primitive under traditional parallel computing architectures.

Another such algorithm is a parallelization of Prim's sequential algorithm. Each process is assigned a subset of vertices and in each step of computation, every process finds a candidate minimum-weight edge connecting one of it's vertices to MST. Leader process collects those candidates and selects one with minimum weight which it adds to MST, and broadcasts result to other processes. This step is repeated until every vertex is in MST.

1.3.2 Find out some limitation of this paper / some parameter is not consider, if we add this in our result/Analysis will be improved

Only dense and connected graph are considered for study. Further experimental work would give us information about practical limitations of our algorithms for wider array of input graphs and uncover new areas for improvement.

1.3.3 Any future work (which is mostly given by the Conclusion section in the paper) identified?

The code is parallelized at two steps:

1) finding minimum key for vertices by finding local and global minimum

2) updating key vector after finding minimum

3) Finding a way do that the algorithm runs on wider array of inputs graphs.

for step 1 author have used thread for finding minimum a better approach would be to use heuristic functions

1.3.4 Any new ideas you think to find out the problem?

We can use CUDA programming platform developed by Nvidia for parallelizing the program. We can also parallelize other algorithms like Kruskals Algorithm and Boruvka's algorithms

There are multiple other parallel algorithms that deal the issue of finding an MST With a linear number of processors it is possible to achieve this in O(logn) . Bader und Cong presented an MST-algorithm, that was five times quicker on eight cores than an optimal sequential algorithm.

2 Model Formulation / Mathematical model / Architecture model

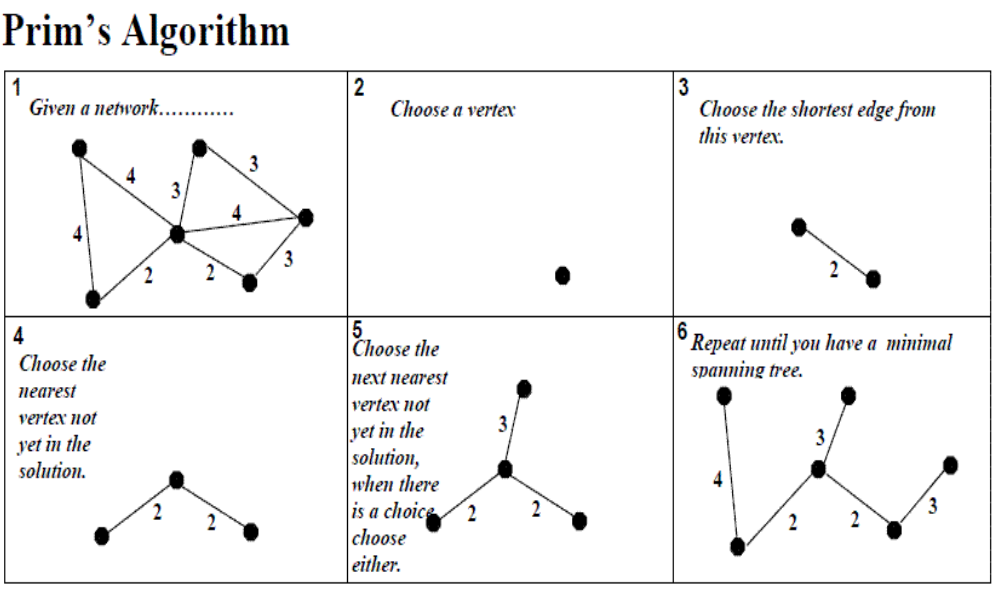

Figure 1 Prims Algorithm

Prims algorithm is parallelized at two steps 1)-finding minimum 2)-updating key vector

	1	2	3	4	5	
1	0	3	2	6	4	p1
2	3	0	1	9	4	p2
3	2	1	0	7	5	p3
4	6	9	7	0	10	p4
5	4	4	5	10	0	p5

Figure 2 adjacency matrix

In serial for finding minimum would require travelling the matrix which would require $O(n^2)$, if we use p threads here p=n then n/p rows would belong to each thread and we apply 1d-row reduction to find local minimum of each processor

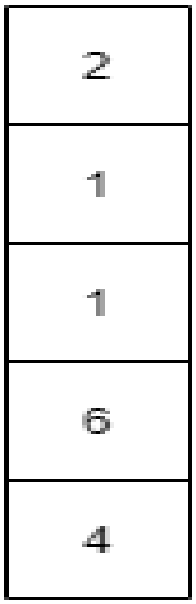

Figure 3 row wise reduced array

Now, by using all to one reduction we will get the minimum value of the array as 1.So the time complexity for finding minimum value will be $O(n/p + logp + p)$.

In step 2, the sequential computing will use an for loop over all the vertices which will result in linear time complexity $O(N)$ where N is number of Vertices. In parallel computation we have used p processors n rows, hence the time complexity will be $O(n/p + logp)$ for updating key vector.

Thus, combining the two steps, a parallel implementation of Prim's algorithm is obtained. The time complexity in each iteration is O(n/p + log p + p). Since we are iterating over the loop n times, the overall time complexity of this algorithm turns out to be O(n2 + n*log p + np).

3 Methodology

3.1 Sequential approach

Let G=(V,E,w) be the weighted undirected graph v_t- contains vertices that are already in MST The array key[1...n] where n is the number of vertices, contains the weight of the edge with the least weight from any vertex in v_t to vertex v in the graph G, initially v_t is empty and Key array is initialized with infinity

At the start of algorithm an arbitrary vertex r is selected as root and included in v_t during each iteration a new vertex u is added to v_t such that the following condition is satisfied key[u] = minkey[v] | v

After the above step all value of Key[v] such that v does not to belong to v_t are updated because there is may be a new edge with smaller weight between vertex v and newly added vertex u , the algorithm terminates when all vertices are included in v_t , the overall complexity of algorithm is O(n2)

```
PRIM_MST (V, w, r)
begin
    V_T := {};
    for all v ∈ V do
        set key[v] := ∞;
    V_T := V_T ∪ {r};
    key[r] = 0;
    while V_T ≠ V do
    begin
        find vertex u such that key[u] :=
                min{key[v] | ∈(V − V_T)};
        V_T := V_T ∪ {u};
        for all v ∈ (V − V_T) do
            key[v] := min{key[v], w(u, v)};
    endwhile
end PRIM_MST
```

Figure 4 pseudocode

3.2 Parallel approach

The parallel implementation can be divided into two steps:
1-Parallelizing the code for finding minimum key for the vertices by finding local minimum and then global minimum.
2-Parallelizing the logic for updating the values of the key vector after finding the minimum key

Let p be the number of processes and n be the number of vertices. For parallel implementation, the key array is partitioned into p subsets with each subset having n/p consecutive vertices. Each process Pi for i = 0, 1, . . . , p-1 computes keyi[u] = minkeyi[v] | v

thus, obtaining the local minimum value for the vertices in the subset. The global minimum is then obtained over all keyi[u] using critical section.

```
int minKey(int key[], int visited[]) {
    int min = INT_MAX, index, i;
    #pragma omp parallel
    {
        int index_local = index;
        int min_local = min;
        #pragma omp for nowait
        for (i = 0; i < n; i++) {
            if (visited[i] == false && key[i] < min_local) {
                min_local = key[i];
                index_local = i;
            }
        }
        #pragma omp critical
        {
            if (min_local < min) {
                min = min_local;
                index = index_local;
            }
        }
    }
    return index;
}
```

Figure 5 pseudocode

Here, visited is the Boolean array representing all vertices with the status of visited and unvisited represented by true and false respectively. Thus, this method will return index of the vertex with minimum key value. The running time for parallel for loop would be O(n/p + log p) and because of the critical section it has to do extra O(p) work. So, total running time complexity of minKey function would be O(n/p + log p + p) where n is number of vertices and p is number of threads.

4 Result and Analysis

number of vertices	sequential (sec)	parallel (sec)
10	0.0000	0.0027
100	0.0002	0.0042
1000	0.0150	0.0217
5000	0.2183	0.3341
10000	0.9143	0.3600
20000	3.4167	0.9412
30000	7.7250	1.9237
40000	13.6200	3.428

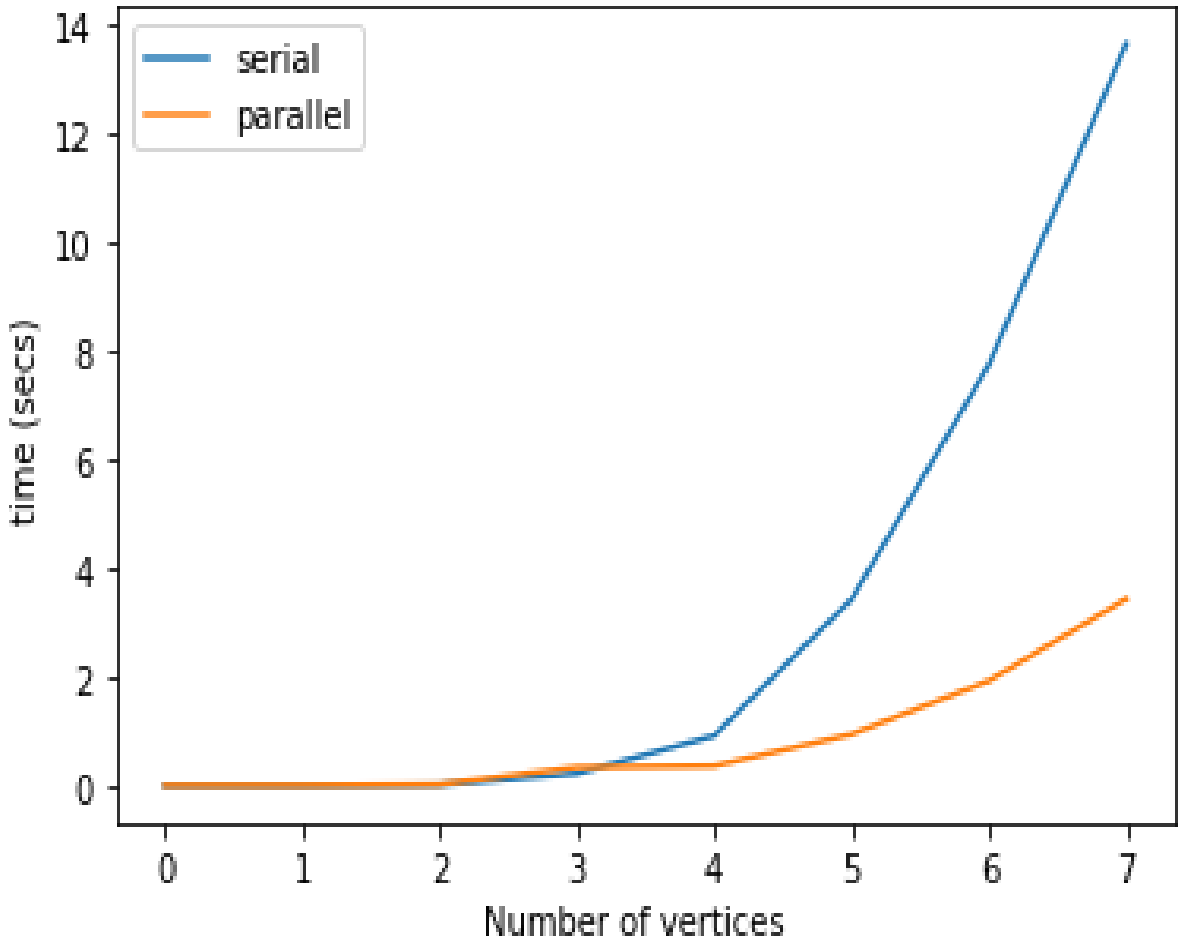

Figure 6 time comparision

We ran code locally and got the following results , the serial algorithm perform better than parallel for small inputs because in parallel algorithm there is overhead of maintaining and scheduling of threads

The overall prim's algorithm cannot be parallelized , but the part of finding minimum

can be improved using multiple threads in serial implementation finding minimum used to take $O(n^2)$ where as in parallel it takes $O(n/p + logp)$

5 Conclusion

5.1 Why did result manifest themselves in a particular way

In the sequential implementation of finding minimum key work in O(n) using a single thread , author in this paper reduced the following complexity to O(n/p + logp +p) where p is the number of threads

5.2 What did the result signify

The parallel approach mainly covers parallelizing the two main logics of the sequential counterpart for finding the local minimum and then updating the key vector with the found local minimum. This algorithm gives a 3x speedup for dense graphs with vertices around 20,000 and 4x to 5x speed up for graphs with vertices around 40,000 for small graphs with less than 5000 vertices sequential algorithm performes better as overhead involved in updating minimum vector is more

5.3 What was relationship between this research and previous research upon it was based

Grama et. al. mentioned that it is very difficult to choose 2 vertices at the same time and thus it is very difficult to parallelize the outer loop. But one can parallelize the method for finding minimum weight edge in cut. The adjacency matrix is distributed in 1-D block fashion. Each processor has (n x n/p) of the adjacency matrix and n/P of Key Vector. Each processor finds locally nearest node, and the global minimum is obtained using all-to-one reduction. The processor containing global minimum then performs one-to-all broadcast, and all the processors then update the value of their respective Key Vector.

Kalé et. al. did something similar to Grama et. al. Instead of adding one node at time to MST, their algorithm adds more nodes to tree in every iteration by doing extra calculations. The algorithm finds K locally nearest nodes, and globally K nearest nodes are obtained using reduction operation. The it iterates through the K nearest nodes to check whether they are valid or not.

author combines the above two approaches with little modification , author uses similar approach to the first method for finding minimum using multiple threads, but rather than adding more than one nodes for growing tree just like method 2 ,instead add one node for growing tree

References

1 G. Karypis A. Grama, A. Gupta and V. Kumar, Introduction to Parallel Computing, Addison Wesley, second edition, 2003.

2 Rohit Setia, Arun Nedunchezhian, Shankar Balachandran, A New Parallel Algorithm for Minimum Spanning Tree Problem, HIPC, 2009

3 Loncar, Vladimir Skrbic, Srdjan. (2012). Parallel Implementation of Minimum Spanning Tree Algorithms Using MPI. CINTI 2012 - 13th IEEE International Symposium on Computational Intelligence and Informatics, Proceedings. 35-38. 10.1109/CINTI.2012.6496797.

4 Wang, Wei Huang, Yongzhong Guo, Shaozhong. (2011). Design and Implementation of GPU-Based Prim's Algorithm. International Journal of Modern Education and Computer Science. 3. 10.5815/ijmecs.2011.04.08.