PARALLEL IMPLEMENTATION OF PRIM'S ALGORITHM USING OPENMP

Parth Rupala

Department of Electrical Engineering and Computer Science
University of California, Irvine

ABSTRACT

This paper describes the parallel implementation of Prim's algorithm for finding minimum spanning tree of dense graph using OpenMP. The parallel implementation uses the cut property of graph to find the minimum weight edge using multiple threads. Several experimental results are described on different dense graphs illustrating the advantages of my approach over the sequential approach.

1. INTRODUCTION

This section gives a general idea about the importance of Minimum Spanning Tree (MST), the sequential approach and how this approach has been parallelized by some. It also includes a comparison of my proposed method with that of the related work.

Minimum Spanning Tree (MST) is defined as a subset of edges of a connected edge-weighted undirected graph that connects all vertices together, without any cycles and with the minimum possible total edge weight.

Motivation: The problem of finding a minimum spanning tree of a graph is very important for many graph algorithms. 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. R.C. Prim provided an algorithm to solve the problem of MST. Prim's algorithm [1] is a greedy algorithm, and it starts by selecting a random vertex as the root of the tree. It then grows the tree by adding a vertex that is closest to current tree. The algorithm ends when all vertices have been added to the tree. The sum of all added edges is the cost of MST. The computational complexity of this serial algorithm is $O(N^2)$, where N is number of vertices which can be reduced using the parallel implementation discussed in this paper. Several parallel versions of MST algorithms are discussed in related work section.

Related work: Several parallel versions of Prim's algorithm are available. Grama et. al. [2] 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. [3] 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.

Setia et. al. [4] 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.

My proposed method of parallelization adds one node at a time to MST unlike Kalé et. al. where they are adding more than one node to the tree in every iteration. My algorithm uses multiple threads to find minimum weight edge in cut by finding local minimum and threads and then global minimum which I have discussed in the later section.

2. BACKGROUND

This section contains a brief overview of how the parallel implementation has been achieved.

In this paper, I have implemented parallel Prim's algorithm for MST which exploits the Cut property of graph. 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. I use 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. I also use multiple threads to update the key property of vertices. I have used OpenMP API available for C/C++ to parallelize Prim's MST algorithm.

3. PROPOSED METHOD

This section first covers the baseline algorithm i.e. the sequential implementation of Prim's algorithm. Next, it explains how this approach is parallelized by giving detailed technical description using pseudocodes and actual code snippets.

Sequential Approach: Let G = (V, E, w) be the weighted undirected graph for which the minimum spanning tree is to be found using Prim's algorithm. The set V_T contains the vertices that are already included in the 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 an empty set and each value in key[1…n] is infinity. To start the algorithm, an arbitrary vertex r is included in VT which becomes the root of the MST and the key of the vertex r i.e. key[r] = 0. During each iteration of the algorithm, a new vertex u is added to V_T such that key[u] = min{key[v] | v $\in$ (V – V_T)}. After this vertex is added, all values of key[v] such that v $\in$ (V – V_T) are updated because there may now be an edge with a smaller weight between vertex v and the newly added vertex u. The algorithm terminates when all the vertices in V are included in V_T. The running time complexity of this algorithm is $O(n^2)$ where n is number of vertices. The pseudocode for this is as follows:

```
PRIM_MST (V, w, r)
begin
    $V_T$ := {};
    for all v $\in$ V do
        set key[v] := $\infty$;
    $V_T$ := $V_T \cup \{r\}$;
    key[r] = 0;
    while $V_T \neq V$ do
    begin
        find vertex u such that key[u] :=
                min{key[v] | $\in (V - V_T)$};
        $V_T$ := $V_T \cup \{u\}$;
        for all v $\in (V - V_T)$ do
            key[v] := min{key[v], w(u, v)};
    endwhile
end PRIM_MST
```

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.

Prim's algorithm is iterative. Each iteration adds a new vertex to the minimum spanning tree. Since the value of key[v] for a vertex v may change every time a new vertex u is added in V_T, it is hard to select more than one vertex to include in the minimum spanning tree. However, the method of finding a vertex u such that key[u] := min{key[v] | v $\in$ (V – V_T)} can be parallelized.

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 P_i for i = 0, 1, …, p-1 computes $key_i[u]$ = min{$key_i[v]$ | v $\in$ (V – V_T) $\cap$ V_i} thus, obtaining the local minimum value for the vertices in the subset. The global minimum is then obtained over all $key_i[u]$ using critical section. The code snippet for this parallel part of finding the minimum key is as follows:

```
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;
}
```

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.

For further optimization, the logic of updating the values of key vector after finding the minimum key value can also be parallelized using the following:

```
#pragma omp parallel for schedule(static)
        for (v = 0; v < V; v++)
            if (graph[u][v] && visited[v] == false &&
graph[u][v] < key[v])
                key[v] = graph[u][v];
```

I have used static scheduling for parallel for loop as shown above for updating the values of key vector after we find the global minimum from the previous parallel for loop. I have also tried dynamic scheduling for the same. But, because of the overhead associated with dynamic scheduling, the performance was poor as compared to static scheduling. The running time complexity of this parallel for loop is $O(n/p + \log p)$.

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 $O(n^2 + n*\log p + np)$.

4. EXPERIMENTAL RESULTS

This section covers the experimental results in terms of running time obtained for both the sequential and parallel implementation of Prim's algorithm using varying number of vertices in dense graphs.

Experimental Setup: I have used OpenMP API available for C/C++ to parallelize Prim's MST algorithm. I use class8-intel queue available on hpc cluster which has 8 cores. The number of vertices taken into consideration vary from 10 to 40,000 for both sequential and parallel implementation.

Results: Using the experimental setup, the running time in seconds for sequential and parallel implementation is shown in the following table:

No. of vertices	Sequential (in sec)	Parallel (in sec)
10	0.0000	0.0037
100	0.0000	0.0010
1000	0.0100	0.0160
5000	0.2100	0.3110
10000	0.8100	0.3673
20000	3.2400	0.8940
30000	7.3100	1.7740
40000	12.9900	2.9407

Table 1. Running time of sequential and parallel implementation of Prim's algorithm in seconds.

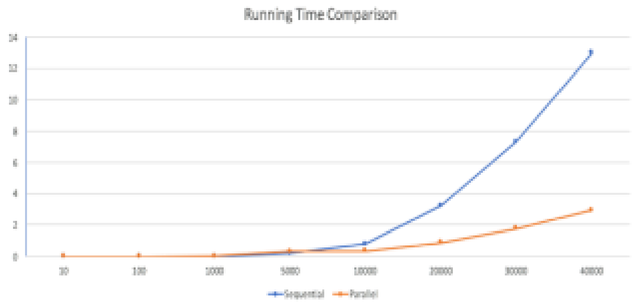

Fig. 1. Performance of sequential and parallel approach in terms of running time in seconds on varying number of vertices.

It is seen from the Fig. 1 that for a dense graph with vertices till the range of 5000, the sequential approach gives a better speedup while a reasonable speedup is obtained when the parallel implementation is run on a dense graph consisting of more than 20,000 vertices.

The reason for this is using multiple threads has overhead which one thread doesn't have. The overhead can be orders of magnitude higher than the work actually done, making it much slower. If the work done is much larger than the overhead you can get very good scalability. Thus, for dense graphs with vertices of the range less than or equal to 5000, the thread overhead is more hence giving a greater running time for parallel than sequential.

5. CONCLUSIONS

In this paper, I have implemented the parallel version of Prim's algorithm and shown experimental results for both sequential and parallel approaches for the running times. 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.

6. REFERENCES

[1] Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA.1990.

[2] G. Karypis A. Grama, A. Gupta and V. Kumar, *Introduction to Parallel Computing*, Addison Wesley, second edition, 2003.

[3] Ekaterina Gonina and Laxmikant V. Kal. *Parallel Prim's algorithm on dense graphs with a novel extension*, *Technical Report*, Department of Computer Science, University of Illinois at Urbana-Champaign, November 2007.

[4] Rohit Setia, Arun Nedunchezhian, Shankar Balachandran, *A New Parallel Algorithm for Minimum Spanning Tree Problem*, HIPC, 2009.