A New Parallel Algorithm for Minimum Spanning Tree Problem

Rohit Setia
Google, India
setiasgnr@gmail.com

Arun Nedunchezhian
Department of CSE
College of Engg., Anna University
arunn3.14@gmail.com

Shankar Balachandran*
Department of CSE
Indian Institute of Technology Madras
shankar@cse.iitm.ac.in

ABSTRACT

Minimum Spanning Tree (MST) is one of the well known classical graph problems. It has many applications in VLSI layout and routing, wireless communication and various other fields. In this paper, we present a new parallel Prim algorithm targeting SMP with shared address space. We use the cut property of graphs to grow trees simultaneously in multiple threads and a merging mechanism when threads collide. We also present two new heuristics and a simple load balancing scheme to improve the performance of the algorithm.

1. INTRODUCTION

Minimum Spanning Tree(MST) problem is defined as follows: Given an undirected connected graph with weighted edges, find a spanning tree of minimum total weight.

A fundamental property that we use in our algorithm is the **Cut property of MST.** For any cut *C* in the graph, the edge with the smallest weight in the cut belongs to all MSTs of the graph. Such a minimum weight cut edge for a cut is called a *light edge*. If there are multiple edges with the same smallest cost, at least one of them will be in the MST.

MST problem has applications in network organization, touring problems, VLSI routing problem, partitioning data points into clusters and various other fields. There exist many serial and parallel algorithms for the MST problem. The first serial algorithm for finding MST was given by Borůvka [2]. Other two commonly used algorithms are Kruskal's algorithm and Prim's algorithm [1]. Most of the existing Parallel algorithms are based on Borůvka's algorithm. Examples are Chung et. al. [3] and Chong et. al. [4]. Recently a hybrid approach (of Prim and Borůvka) was used by Bader et. al. [5]. There are several parallel formulations of Prim's algorithm [5, 6].

In this paper we design and implement a new Parallel Prim algorithm for the MST problem targeting SMP with shared memory. We use multiple threads to run algorithm in parallel. The algorithm non-deterministically chooses a node and sets it as root. Each thread starts growing a tree in parallel by colouring the nodes with a unique colour (called its id). When a collision occurs(a thread likes to add a node that belongs to another tree), one of the thread sends a signal to other and we merge these trees using a *MergeTree* operation. We force the tree grown by the thread with larger-id to merge with tree grown by a thread with smaller-id. Eventually, thread 0 will have the MST. The threads are assumed to have the capability to send asynchronous signals to each other.

2. RELATED WORK

There are several parallel implementations of Prim's algorithm. Kumar et. al. [6] pointed out that the main outer while loop of serial Prim is very difficult to run in parallel. But one can find nearest outside node in parallel by Min-Reduction and also the update-keys step can be done in parallel. The adjacency matrix is partitioned in a 1-D block fashion. (Each processor has n × n/P of the adjacency matrix and n/p of the Key Array). Each processor finds the locally nearest node, a global min reduction is done and main thread adds the nearest node to the tree and the row entry of this node in adjacency matrix is broadcast to all processors.

Gonina et. al. [7] follows a very similar algorithm but instead of adding one node to the current tree, their algorithm tries to add more nodes to the tree in every pass by doing some extra computation. The algorithm finds locally K nearest outside nodes and global Min-Reduction is done to obtain globally closest K nodes. The algorithm then iterates through the list to find out whether they are valid or not.

The main point to note here is that in both parallel formulations of Prim's algorithm they are growing a single tree. Bader et. al. [5] came up with a non-deterministic shared memory algorithm which uses a hybrid approach of Borůvka and Prim algorithm. Each processor chooses a root node and grows tree in similar fashion of serial Prim approach and when the tree finds a nearest node that doesn't belong to any other tree it can add the node, whereas if the node belongs to another tree then it

*Corresponding author. The work was started when Rohit Setia was a student at IIT Madras. Arun Nedunchezhian is currently a senior student in the Dept. of CSE, College of Engineering, Guindy, Anna University.

must stop growing and start with a new root. In the end, we get different connected components(which are trees) and some isolated vertices. No two trees share a vertex because merging was avoided. Now Find-Min step of Borůvka Algorithm is used to shrink each of the components into a super node.

Our new parallel algorithm also grows multiple trees in parallel and when a collision occurs between two threads and *j* (*i* < *j*), thread *j* merges with *i*. Thread *i* continues to grow the tree from there and thread *j* picks another node randomly and grows a new tree.

The rest of the paper is organized as follows. In Section 3, we describe the new parallel algorithm and prove its correctness. We also describe the load balancing schemes we developed. In Section 4, we describe two new heuristics to reduce the number of collisions. In Section 5, we present our results.

3 PARALLEL ALGORITHM

Algorithm 3.1 **Main Thread**

Input: A connected graph G = (V,E) represented by adjacency matrix. |V| = number of vertices; |E| = number of edges. Vertices have unique ids.
Output: Minimum Spanning Tree for Graph G

begin

1. For each vertex, set the colour attribute to -1.
2. Create threads with unique thread-ids.
3. For each thread i and node v, set status[i][v] = WHITE and KeyArray[i][v] = ∞
4. Child threads will run MST Algo in parallel.
4. Wait for termination of all threads.
5. Combine result of all threads.

end

Algorithm 3.2
MST Algorithm executed by Child Thread with threadId **i** (Tree (i))

```
begin
while(1)
1.    Choose root node non-deterministically
      If all nodes are visited return
      flag= true;
      while(flag==true)
2.          find the nearest node 'minnode' with
               status[i][minnode]=GRAY
            if no node can be found  return
            lock minnode
3. 1        if color[minnode] = -1 then
                block all signals
                color[minnode] = i
                status[i][minnode] = BLACK
                unlock minnode
                append minnode to treelist of 'i'
3.1.1           for all neighbours 'v' of minnode
                   if status[i][v] = WHITE
                      status[i][v] = GRAY
                      append v to treelist of 'i'
                   else if status[i][v] = GRAY
                     KeyArray[i][v] = min(value[i][v],
                              AdjMat[minnode][v] );
                end for
                unblock all signals
3.2         else if color[minnode] != i then
               j = color[minnode];
3.2.1           if i < j then
                      send signal -1 to j
                      wait till thread 'j' accepts signal
                      and executes signal handler
                      ( Mergetree(i,j) )
                      unlock minnode and kill j
3.2.2           else if i > j then
                      send signal -2 to j
                      wait till thread 'j' accepts signal
                      and executes signal handler
                      ( Mergetree(j,i) )
                      unlock minnode and kill j
3.2.3           else
                      unlock minnode
                      continue;
3.3         else if color[minnode] == i then
                      continue;
      end while
end while
end
```

Initially, color[] of all nodes are set to -1. Each thread begins with unique thread-id *i* which is set as its color. Now each thread grows a tree from a randomly chosen root node and starts coloring each node with its color. Initially status[i][v] is set to WHITE for each node v and KeyArray[i][v] is the minimum distance from a node v to the tree i. In step 3.1.1, the status of nodes that are adjacent to current node are marked as GRAY. Among the nodes that have status[i] as GRAY, the node which has minimum KeyArray[i][] value is chosen as 'minnode' and it is added to tree and status[i] [minnode] is set as BLACK.

When a collision occurs, to resolve the collision, one of the trees should be merged with (or colored with) by the other tree. To resolve this, let us assume that the tree grown by the thread *i* collides with thread *j* (*i* < *j*). We force the nodes in tree-*j* to be colored by *i* and thus tree-*j* is merged with tree-i. This step is called **MergeTree(*i,j*)** and it is done in the critical section to avoid race conditions.

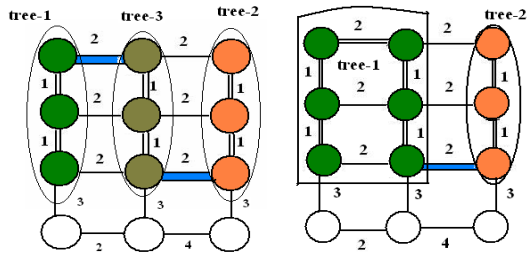
Merging of one tree with another in this way is a novel idea and it is initiated by sending signals between threads. The two types of signals are

Signal -1

It is sent by thread with smaller-id(i) to thread with larger-id(j) asking it to surrender tree-j. After the surrender, thread *j* is killed.

Signal -2

It is sent by thread with larger-id(j) to thread with smaller-id(i) to merge tree-j with tree-i. Thread *j* is killed after *i* has merged the nodes.



Before merge After merge

Figure 1. Merge Tree (1,3) Operation

Figure 1 shows the case where trees 1..3 have started growing in parallel and let us assume that thread 1 finds a node that is coloured by thread 3 as its minnode. Thread 1 sends signal-1 to thread 3 to initiate MergeTree(1,3). The right side of the figure shows the colours after the operation. MergeTree( ) could have been initiated by thread 3, if it had found the collision first. In that case, signal-2 is sent by thread 3 to thread 1. The nodes that belong to tree-3 are colored by thread 1 and are made as a part of tree-1, without any change in the results. In the next iteration, tree-1 will merge with tree-2. MergeTree() is done in critical section to avoid formation of cycles and race conditions.

If threads *i* and *j* both identify a collision simultaneously, and if *j* enters the critical section, thread *i* will notice later that the node on which collision occured is already coloured with its color. Step 3.3 takes care of this by not changing nodes that got colored in the recent MergeTree operation.

We show the correctness of the algorithm below.

Lemma 1.1 *No cycles are formed during MergeTree operation.*

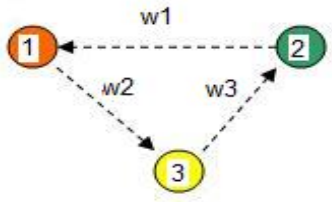


Figure 2: Illustration to Prove Lemma 1.1

Proof :

The smallest cylce that can be formed is of length 3. In Figure 2, let us assume to the contrary that a cycle is created during MergeTree(). Then,

$w_3 < w_2$ (as tree-3 selects lightest edge leaving it)

$w_1 < w_3$ (as tree-2 selects lightest edge leaving it)

$w_2 < w_1$ (as tree-1 selects lightest edge leaving it)

$\Rightarrow w_3<w_2<w_1<w_3$ clearly a contradiction

So such a case is not possible. But, if $w_1=w_2=w_3$ then we get merge requests MergeTree(1,2), MergeTree(2,3), and MergeTree(1,3). As we do MergeTree in the critical section, only 2 out of 3 requests are executed and the last request is not granted in step 3.3 in Algorithm 3.2. The proof can be extended to cycles of any length greater than 3. Hence no cycles are formed.

Lemma 1.2 *The edges added by the Algorithm 3.1 belongs to MST.*

Proof:

Parallel Prim algorithm always converges. In the end, the tree (corresponding to the smallest thread-Id) keeps growing till all the nodes are part of its tree. Tree obtained has all n nodes and (n-1) edges and no cycles.

Consider threads growing tree i and tree j with t_1-1 nodes and t_2-1 nodes respectively. When they merge there will be only one additional edge joining the two trees. So the resulting tree will contain t_1+t_2 nodes and t_1+t_2-1 edges. Hence the graph is connected and it is a tree.

If $E(t_1,t_2)$ is the lowest cost edge joining a node in t_1 with a node in t_2, then

$$MST(t_1) + MST(t_2)+ E(t_1,t_2) = MST(t_1 \cup t_2)$$

if and only if $E(t_1,t_2)$ has its weight larger than all the edges in $MST(t_1)$ or $MST(t_2)$. In our algorithm, the edges that are added during MergeTree always satisfy this property. To prove that, consider that an edge *e* exists whose weight is less than the weight of some edge in either of the trees. Then, *e* would have been added already to that tree in Step 2 of the algorithm 3.2. Hence no such edge exists and thereby $E(t_1,t_2)$ is the next lowest cost edge.

Hence all edges chosen by the algorithm belongs to MST.

3.1 Load Balancing Scheme

As the algorithm randomly picks root nodes for new trees, there is no guarantee in the algorithm that each thread will do significant amount of work before terminating. So we need to balance the load among threads to ensure that each thread does a fair amount of work.

Termination of Threads:

One simple load-balancing scheme is that instead of terminating the thread *j* in step 3.2 of Algorithm 3.2, we can let *j* choose a new random root node and then grow a new tree from this root node.

Base Problem Size

Parallelization can be prohibitive for problem sizes that are small. As our algorithm proceeds, fewer and fewer nodes remain uncolored and thereby the trees that are grown are quite small before they merge into some other tree. When the number of uncolored nodes are below a certain threshold, serial version of Prim's algorithm may consume lesser time than any parallel version. We call this threshold the Base Problem Size.When we restart thread *j* after a MergeTree(i,j) operation, we check if the number of uncolored nodes is less than the Base Problem Size. If it is so, we terminate the thread *j*. If not, it is allowed to proceed to pick a new random root to start its next round of work. We modify the outermost while loop to check for the number of uncolored nodes before starting to grow a new tree. Any thread *k* which is not currently merging with any other thread is allowed to continue even when the uncolored nodes go below the Base Problem Size.

In practice, this must be set based on the hardware platform the algorithms run on. We have empirically set the value of Base Problem Size to be $|V| / p$ if there are *p* threads created.

4. PROPOSED HEURISTICS

Each mergeTree() call requires a signal to be delivered. The number of signals generated is a good indication of communication cost of the algorithm. More number of collisions result in more communication cost. As the algorithm has randomization, it is not easy to determine the number of collisions. One can reduce the number of collisions by dynamically terminating some of the threads (not the smallest thread-id thread). We have designed two new heuristics to get a bound on number of collisions.

Heuristic-1:Wrap-around find-min

When we find the next lightest edge in thread *i*, we have to search through the list of nodes to find out which edge is appropriate for addition in the next iteration. In our implementation, we always started the search from the lowest node index. This resulted in some systematic collisions.

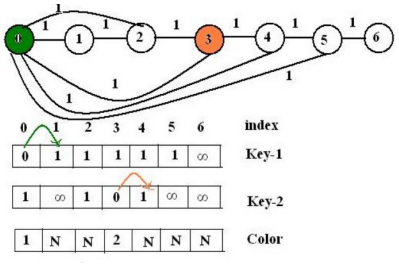


Fig 3. Problems With Systematic KeyArray Search

Consider Figure 3 where there are 3 threads in operation. In KeyArray of thread 1 we find that there are many candidate nodes for nearest outside node who has the same least key (distance from the growing tree). Any one of them can be added to the tree-1 in current pass. Nodes 1, 2, 3, 4, 5 are at distance 1 from tree-1 and they are outside nodes for tree-1. Similarly nodes 0, 2, 4 are at outside nodes at distance 1 from tree-2. If we always systematically started the minimum search from node 1, then both threads will likely collide very soon. We can avoid this by choosing other candidate nodes having least key. In Figure 3, instead of choosing node 0 for tree-2 if we choose node 4 then tree-2 will not merge into tree-1 in current pass. The algorithm is still correct as node 0 and node 4 are nearest outside nodes for tree-2, so we can choose node 4 for tree-2. Our idea is that, instead of searching from the beginning of KeyArray, we start from the node that we added in the last iteration and wrap around to beginning of array when it reaches the end. Thus in figure 3, for thread 2 we start from node 3 instead of node 1 and finally wrap around when it reaches 6.

Heuristic-2: Threshold Nodes

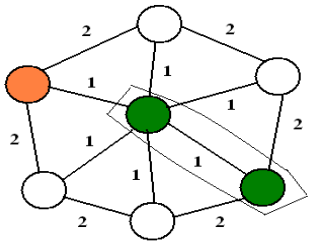


Figure 4. A Graph With Underperforming Thread

It is also possible that a thread picks up root nodes in such a way that the collisions happen very soon and the tree that it grows is small. Such a case will happen for the orange colored thread in Figure 4. No matter what it tries, it will grow only trees of size one node before they merge with the green thread. If a thread repeatedly grows small trees, there is no incentive for us to let the thread continue. It is better to kill the thread and thereby avoid the communication overheads

To achieve this, we check in thread *i* if it added at least N_{TH} nodes before it merged. If not, we count the number of times such a scenario arises. Once the thread *i* has consistently underperformed in more than *k* iterations, we force the thread to be killed. We perform this even if the criteria of Base Problem Size allows growing the tree. We used k=3 in our experiments.

The number of collisions in our implementation is then bounded as shown below :

$$(p - 1) \leq numofCollisions \leq (C_1 + C_2)$$

where

$No.of\ Threads = p$

$C_1 = k(p - 1)$

$C_2 = (|V'| - 1 - k(p - 1)) / (N_{TH} + 1)$

5. EXPERIMENTS AND RESULTS

We implemented our parallel Prim algorithm using POSIX threads and C++. The implementation is tested on multi-core IBM p690 SMPS machine with PowerPC_POWER4 with 32 cores running at 1.7 GHz running AIX 5 OS (xlC compiler). The implementation of Parallel Prim was done with arrays for dense graphs and with Fibonacci heaps [1] for sparse graphs. We tested our code for various types of graphs.

1. Dense graphs - 1000, 3000, 5000 nodes complete graphs
2. Sparse Graphs – 2d or 3d grids, random connected sparse graphs

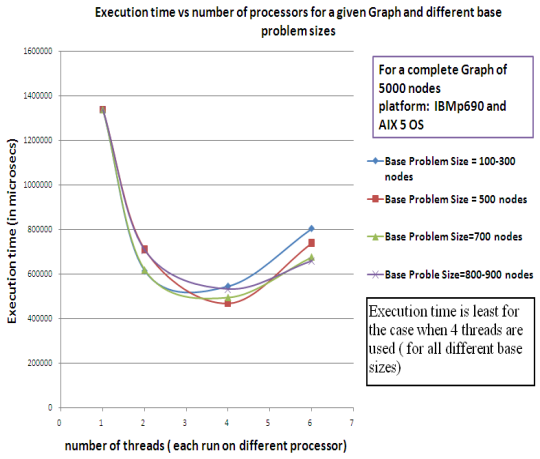


Figure 5 Experimental results for a complete graph with 5000 nodes

Figure 5 shows a representative plot of the perofrmance of our algorithm as we increase the number of thereads. Execution times for a dense graph with 5000 nodes is shown. Different curves show the execution times for different Base Problem Sizes. We changed the Base Problem Size from 100 to 900 nodes and varied the thread count from 1 to 6. We noticed that we get the best performance when the thread count is 4. For a graph with *N* nodes and running *p* threads, we found in our experiments that the best performance is achieved when Base Problem Size is set to *N/p.*

We noticed that as we increase the thread count, MergeTree operation starts becoming expensive and thereby the communication cost and sycnhronization becomes more expensive.

The speedup achieved for the dense graph with 5000 nodes is **2.64** for 4 threads with the base size of 500 or wutg the base size of 800 for 6 threads For Sparse graphs the speedup achieved is 1.3 to 1.8 for 4 to 6 threads. We notice similar trends for larger graphs and larger thread counts.

6 CONCLUSION AND FUTURE WORK

We presented a new parallel Prim algorithm that grows multiple trees in parallel. We made simple observations based on the cut property of the graph to grow MSTs in parallel. We noticed some of the bottlenecks in the implementation and proposed heuristics to make the algorithm scalable. We presented some implementation issues of the algorithm. In particular, we presented some effective load balancing schemes.

Our algorithm achieves reasonable speedup when it is compared with Serial Prim algorithm for dense graphs and sparse graphs. It will be interesting to address some of the scalability issues in our algorithm and see how to effectively use the 32 processors. We are currently working on a few ideas to merge trees based on their sizes instead of their ids. We are also looking at techniques to both speed up the MergeTreeOperation and to reduce the collisions.

7 REFERENCES

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

[2] Otakar Borůvka. O jistém problému minimálním (About a certain minimal problem). Práce mor. přírodověd. spol. v Brně III 3: 37–58.1926

[3] Sun Chung and Anne Condon. Parallel implementation of Bor°uvka's minimum spanning tree algorithm.(IPPS'96)

[4] Ka Wong Chong, Yijie Han, Yoshihide Igarashi, and Tak Wah Lam. 1999. Improving the Efficiency of Parallel Minimum Spanning Tree Algorithms. Discrete Applied Mathematics. 2003

[5] David A. Bader, and Guojing Cong.Fast. Shared-Memory Algorithms for Computing the Minimum Spanning Forest of Sparse Graphs. JPDC, 2006

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

[7] Ekaterina Gonina and Laxmikant V. Kalé. 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.