Parallel Minimum Spanning Tree Algorithm

Xiwen Chen

Department of Computer Science and Engineering, York University
4700 Keele Street, Toronto, Ontario, Canada, M3J 1P3

Abstract. The Minimal Spanning Tree (MST) problem is a classical graph problem which has many applications in various areas. In this paper we discuss a concurrent MST algorithm derived from Prim's algorithm presented by Setia et al. in 2009, targeting symmetric multiprocessing (SMP) with a shared address space. The pseudocode of the algorithm is presented, combined with three interesting heuristics in order to achieve a high level of parallelism. We also analyze the parallel time complexity of this algorithm and the experimental results demonstrate it is a good time bound. In the last part of the paper, we use different methods to test and verify our implementation, including directly testing and model checking using Java PathFinder.

1 Introduction

A spanning tree is a sub-graph of the given undirected connected graph, which is a tree and covers all the nodes of the graph. A Minimum Spanning Tree (MST) is the spanning tree whose weights are less than or equal to any other spanning trees of the graph. MST is one of the well-known classical graph problems which has many critical applications in network organization, VLSI layout and routing, touring problems, partitioning data points into clusters and various other fields. It was in 1926 that Borůvka [1] produced the first fully realized sequential algorithm to find the minimal spanning tree. At a later time, Kruskal and Prim developed the two most commonly used MST algorithm, Kruskal's algorithm [2] and Prim's algorithm [3], respectively.

However, with the proliferation of multicore processors, people are eager to build the MST more quickly with the power of those multicore machines. In this report, we discuss a parallel MST algorithm based on the paper presented by Setia et al. [4], which focuses on symmetric multiprocessing (SMP) with a shared address space. In this implementation multiple threads are used, running in parallel and each of them grows its own MST concurrently by coloring the nodes with a unique color (i.e. its id). When one thread has to add a node which already belongs to another thread, we merge these two trees using a MergeTree operation. Moreover, we force the tree with the larger id to merge into the tree with the smaller id. Eventually, the thread with the smallest id will have the MST of the whole graph.

This parallel algorithm are derived from Prim's algorithm [3] based on the cut property of a MST and will be discussed in Section 3.1 and Section 3.2. We

also present some interesting heuristics which make the algorithm perform more efficiently. In Section 5, we analyze the parallel time complexity of the algorithm and discuss it by using two different graph models. More importantly, since we run the implementation on the 32 cores machine in the Intel® Manycore Testing Lab, we compare the differences in terms of the performance when using different number of threads. Finally, we discuss our work on testing and verifying the correctness of this implementation using both a traditional testing method and a state of the art model checking tool called Java PathFinder. In the Appendix, we also attach a brief proof [4] of the correctness of the algorithm and the MergeTree operation.

2 Related Work

A lot of existing concurrent MST algorithm are based on Borůvka's algorithm, for instance, Chung et al. [5] and Chong et al. [6]. Recently Bader et al. [7] discovered a hybrid method based on Prim's [3] and Borůvka's approach.

Grama et al. [8] pointed out that the main outer loop of Prim's sequential algorithm is very difficult to run in parallel. But we can find the nearest nodes in parallel by the so called Min-Reduction in that paper, and also the update-keys step can be done in parallel [4]. Gonina et al. [9] try to add many nodes to the tree at a time by doing some extra computation. In that paper, the algorithm finds locally the nearest K nodes and does a global Min-Reduction to obtain globally the closest K nodes.

Both parallel versions of Prim's algorithm are growing a single tree, and find the nearest node concurrently. Bader et al. [7] developed a non-deterministic shared memory algorithm which combines both Borůvka and Prim's algorithm. In their approach, each thread chooses a root and grows a tree just as Prim's serial algorithm. The thread will stop and start with a new root when it finds a nearest node which belongs to another thread. Eventually, we will get many different connected components, which are trees with one or more nodes. Now Borůvka's algorithm is used to merge those component into a tree.

The algorithm in the selected paper [4] is similar to the work that Bader et al. [7] did before, which also concurrently grows trees using multiple threads. However, when a collision happens between thread i and thread j $(i<j)$, a subroutine will be called to merge the tree of thread j into i*'s* tree. Then thread i continues to grow the tree from there and thread j randomly picks another node to grow a new tree.

3 Prim's Parallel Algorithm

3.1 Prim's Algorithm

Definition 1. *A cut is a partition of the nodes of a graph into two disjoint subsets. For any cut C in a graph, an edge with the smallest weight in the cut called a light edge.*

Prim's algorithm grows a node set A_{mst} by adding a light edge that connects A_{mst} to an isolated node, one on which no edge of A_{mst} is incident, to the tree A_{mst}. In the pseudocode below, the connected graph G and the root r of the MST are inputs to the algorithm. During execution of the algorithm, we keep all the nodes that are not in the tree in a priority queue Q based on a *key* attribute. For each node v, the attribute $v.key$ is the minimum weight of any edge connecting v to a node in the tree. We assign $v.key = \infty$ if there is no such edge. The attribute $v.\pi$ indicates the parent of v in the tree [10]. The algorithm implicitly maintains the set A_{mst} as

$$A_{mst} = \{(v, v.\pi) : v \in V - \{r\} - Q\}.$$

When the algorithm terminates, the priority Q is empty and the MST A_{mst} for G is thus

$$A_{mst} = \{(v, v.\pi) : v \in V - \{r\}\}.$$

MST-PRIM(G, w, r) *pseudocode:*
(Here G is the graph, w is the set of weight of all edges, r is root we choose to grow the MST.)

```
1   for each u ∈ G.V
2       u.key ← ∞
3       u.π ← NIL
4   r.key ← 0
5   Q ← G.V
6   while Q ≠ ∅
7       u ← EXTRACT–MIN(Q)
8       for each v ∈ G.Adj[u]
9           if v ∈ Q and w(u,v) < v.key
10              v.π ← u
11              v.key ← w(u,v)
```

The **while** loop of *line 6-11* has the following properties.

1. The edge set of MST $A_{mst} = \{(v, v.\pi) : v \in V - \{r\} - Q\}$.
2. The nodes already selected into the MST are those in $V - Q$.
3. For all the nodes $v \in Q$, if $v.\pi \neq NIL$, then $v.key < \infty$ and $v.key$ is the weight of a light edge $(v, v.\pi)$ connecting v to some nodes already placed into the MST.

Line 7 chooses a node $u \in Q$ incident on a light edge that crosses the cut $(V - Q, Q)$, removing u from the priority queue Q, implicitly adding it to the set A_{mst} (i.e. adding $(u, u.\pi)$ to A_{mst}). The **for** loop of *line 8-11* updates the *key* and π attributes of every node v adjacent to u but not in the tree, thereby maintaining the third loop invariant.

The running time of Prim's algorithm depends on how we implement the min-priority queue Q. One can use the binary min-heap to implement the priority queue, which will take $O(V \log V + E \log V) \approx O(E \log V)$, as the number of edges is usually larger than the number of nodes. We can still improve the asymptotic

running time of the algorithm by using a Fibonacci heap [11], achieving $O(E + V \log V)$. This will be a significant improvement if the number of edges becomes far more than the number of nodes.

3.2 Parallel Algorithm

In this section, we present the parallel approach published by Setia et al. [4]. A fundamental property that underlies this approach is the *Cut property of* MST.

> **Cut property of MST:** For any cut C in the graph, the light edge in the cut belongs to all MSTs of the graph. If there are multiple light edges with the same weight in the cut, at least one of them will be in the MST.

Intuitively, in order to take the full advantage of multiple processors, we use multiple threads to grow their own MST simultaneously based on the *cut property of* MST. Once a collision occurs, we force the tree grown by the thread that has the larger index (j) to merge into the tree who has the smaller index (i). Thread j will randomly choose another new node to grow a new MST, thread i will continue growing. This eventually guarantees that thread *0* will have the MST for the whole graph.

The Main Thread The main thread initializes the *environment variables*, *MSThreads*, and waits for the termination of all the *MSThreads*.

> **Input:** A connected graph $G = (V, E)$ represented by an adjacency matrix. $|V|$ and $|E|$ denote the number of the vertices and edges, respectively. All nodes have unique ids.
> **Output:** Minimum Spanning Tree of the Graph G.

Main Thread *pseudocode:*

```
1  for each node v
2      v.color ← −1
3      v.parent ← v
4      v.rank ← 0
5  create threads with unique thread−ids
6  for each thread i
7      i.start()
8  for each thread i
9      i.join()
10 return the result of thread 0
```

Line 1-4 firstly initialize all nodes as uncolored ones. Furthermore, because we will use the disjoint set combined with path compression and balanced union by rank [10] for the convenience of maintaining the MST set of each thread and also for the merge operations, we create $|V|$ disjoint sets for those nodes and set

the rank of them as 0. Actually, this data structure guarantees that we can union two MSTs using at most *â(n)* steps, where *â(n)* is the inverse of Ackermann's function [12]. (Basically, *â(n)* is at most 4 for all conceivable values of n.) After the initialization steps, we create several $MSThreads$ and let them start. Finally, the main thread should wait for the termination of all those children threads, and then returns the MST found in the given graph G.

The MSThread Thread Each $MSThread$ individually grows its MST by selecting the light edge it encountered and try to color it with its own thread-id. When a collision occurs, one of the trees should be merged into (colored into) the other tree by calling the MergeTree method.

MSThread *pseudocode:*

```
1   while(true)
2       root r ← randomly choose an uncolored node
3       if no such r exist
4           return
5       create a PriorityQueue for the thread.
6       for each neighbor u of r
7           if PriorityQueue.decreaseKey(u, Edge[r][u])
8               u.π ← r
9       while(!PriorityQueue.empty())
10          minnode ← PriorityQueue.deleteMin()
11          setidx ← Graph.find(minnode)
12          lock setidx
13          if setidx.color = −1
14              block all signals
15              setidx.color ← i
16              add setidx into MST of i
17              for each neighbor v of minnode
18                  if PriorityQueue.decreaseKey(v, Edge[minnode][v])
19                      v.π ← minnode
20              unlock setidx
21              unblock all signals
22          else if setidx.color ≠ i
23              j ← setidx.color
24              if i < j
25                  lock i and j
26                      MergeTree(i,j)
27                  unlock setidx,i,j and restart j
28              else if i > j
29                  lock j and i
30                      MergeTree(j,i)
31                  unlock setidx,j,i and restart i
32              else
33                  unlock setidx
34                  continue
35          else if setidx.color = i
```

```
36                unlock setidx
37                continue
38          end while
39    end while
```

Each thread has to randomly pick an uncolored node as its root to grow a MST as shown in *line 2-4*. When the thread successfully picks a root, it updates the node information in its priority queue. *Line 9-38* are the loop the thread grow its MST. It chooses a node from the next light edge and sees if such node is colored or not. It's easy if the node is uncolored as displayed in *line 13-21*, where it performs similar as the Prim's sequential algorithm. If the node is colored by itself, the thread continues. Otherwise, we perform a MergeTree operation (*line 22-34*) according to the order of the two thread ids. One may also notice that the situation in *line 32-34* seems impossible to happen. However, this case actually is so important since in a concurrent setting, we can not guarantee that once we step into *line 23* the pre-condition still holds (i.e. now $setidx.color = i$).

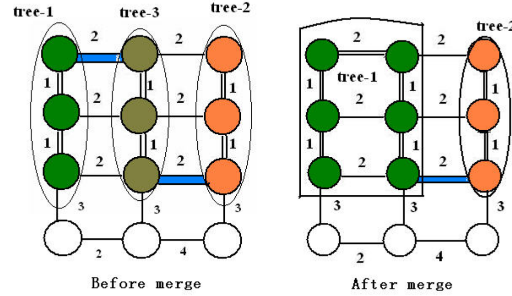

Fig. 1. MergeTree(1,3) Operation

Figure 1 [4] shows an example where we have three threads growing their own tree and each tree was colored with its unique thread-id, and now the collision occurs. Let's assume that thread 1 chooses a *minnode* which belongs to thread 3. So thread 1 performs MergeTree(1,3). The right figure shows the result after the merge operation. Certainly, the same result can also be obtained by calling MergeTree(3,1) from thread 3, and merge its MST into thread 1's. What's more, in the next iteration, MergeTree(1,2) may be invoked, merging the tree2 into tree1.

The MergeTree Operation

pre-condition: $i < j$
post-condition: The MST of thread j is merged into i's MST.

MergeTree(i, j) *pseudocode:*

```
1  merge the j's edge list into i's
2  update the size info of current MST of thread i
3  update the weight info of current MST of thread i
4  union MST set of j with MST set of i
5  color the new set with i's id
6  update all the key values in the priority queue of thread i
```

Line 1-5 can be done using $O(1)$ time easily. And *line 6* will take $O(n)$ time if we use a Fibonacci Heap [11]. One may think that the thread with the smallest id will always hold the final MST of the graph at the end. But we notice that sometimes if the first thread doesn't even get really started, the thread with the second smallest id may have the MST of the whole graph.

4 Heuristics

We modify and implement some interesting strategies and heuristics similar to the original paper [4], with the aim of achieving a higher degree of parallelism for this concurrent algorithm.

4.1 Base Problem Size

> **Base problem size [4]**: We set a threshold value for the number of uncolored nodes. If such number falls below that threshold, we will terminate the thread instead of restarting it to pick a new random root to grow a new MST.

In our implementation, we use a similar idea which actually keeps track of the number of the times that a thread randomly picks a node as root, but such node was already colored. We then set a threshold for that number and if a thread fails too many times to pick a root, it will be terminated. It is straightforward that as long as the random number generator can uniformly generate numbers from 0 to $|V| - 1$, the number of failures to pick an uncolored node are probabilistically related with the number of uncolored nodes. Furthermore, this simple change does improve the parallelism of the algorithm a lot, since when we keep track of the exact number of uncolored nodes, we have to lock and unlock that number frequently. This actually makes all the threads behave sequentially, competing the mutual exclusive access to that object with each other.

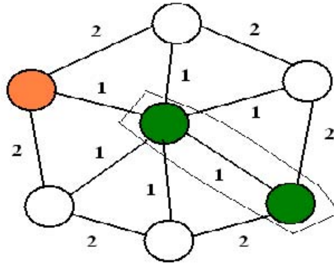

Fig. 2. A graph with underperforming thread

4.2 Threshold of MST Size

Figure 2 [4] displays a situation that no matter what the orange colored thread tries, it will grow a MST of size one and get collisions immediately with the green thread. Thus, if a thread repeatedly grows small trees, there is no incentive for us to let the thread continue. It's better to terminate the thread and thereby avoiding the overhead of the MergeTree operation.

Therefore, we check the size of the MST of a thread after it merges its tree into another thread before it restarts. If that size is consecutively under a threshold for K times, we force such underperforming thread to be killed. In our implementation, K is set to 2.

4.3 Wrap-around find-Min

The most interesting heuristic presented by the authors is the Wrap-around find-Min [4] strategy. The basic idea is shown in Figure 3.

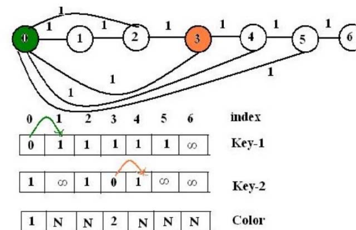

Fig. 3. Problems with symmetric KeyArray search [4]

As we can infer from the Figure 3, if all the threads start from beginning to the end to find the minimum key systematically, it will result in a high probability of collisions. Setia et al. [4] suggest that instead of always starting from the lowest index, we start from the root and wrap around to the beginning of the array when it reaches the end. For example, in Figure 3, thread 2 starts from node 3 instead of node 1 and finally wraps around when it reaches node 6.

In order to test the performance of such a strategy, we implemented three different methods: the naïve approach without warp around, start from root and start from a random node. The left part of Figure 4 illustrates the average size of the minimum spanning tree the thread can grow when a collision happens (20 trials), and the right diagram displays the total size of MST that children can grow in each corresponding trial.

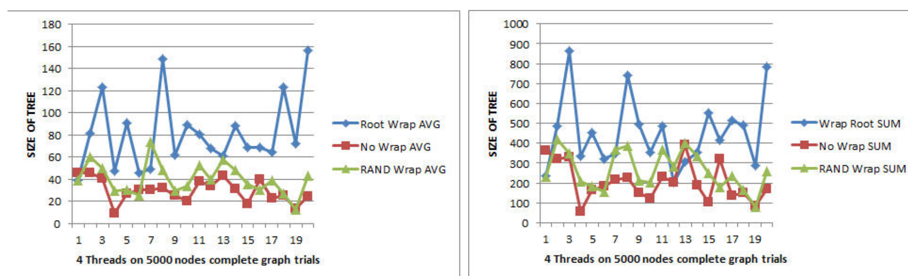

Fig. 4. The average size of MST when collision occurs and the corresponding total size of MST generated by children threads, tested 20 times.

The two diagrams show the same fact that with such a heuristic, threads will have a better chance to grow a bigger MST and the probability of collision will be reduced as well.

5 Time Complexity and Performances

By using the Fibonacci heap [11], which has a constant amortized time for the DecreaseKey operation, combined with the disjoint set using path compression and balanced union by rank, we can achieve an $O(|E| + |V| \log |V|)$ time for the sequential algorithm. For the concurrent algorithm, it's usually hard to figure out the exact time bound for it, since there are too many interleavings while running the concurrent program. However, we can still roughly compute the time if we don't consider the overhead of locks, and assume that each thread performs ideally and the graph is perfect for multiple threads to grow their MST (i.e. only a few collisions). Thus, the parallel running time for this algorithm will

be

$$O(P \cdot |V| \log |V| + \frac{|E|}{P})$$

where P is the number of threads. According to the above formula, it's easy to conclude that the concurrent algorithm will gain some speed up if the right term dominates. Thus, we will use two types of graph to check the performance of our implementation, one has a huge amount of edges, the other are costly for edge access.

5.1 Graph in Adjacency-Matrix

If we store the graph in the form of an adjacency matrix, where edges are represented by an int (4 bytes), it costs 4 Gigabyte to store a complete graph with 30,000 nodes. Thus, we run our algorithm on a randomly generated complete graph with 1,000 - 4,000 nodes, 10,000 nodes and 30,000 nodes to see how the left term and the right term affect the running time.

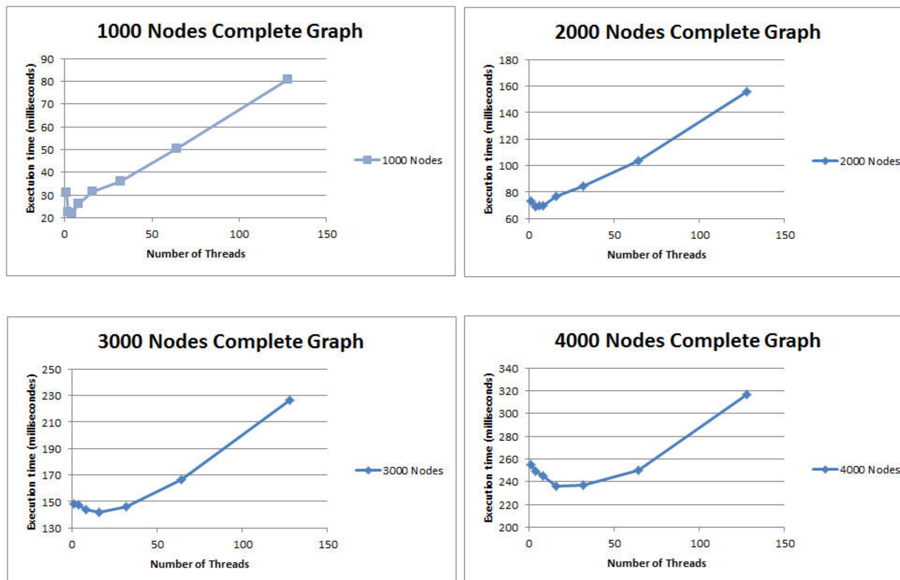

Fig. 5. Execution time on different size of graph using different number of threads stored in int matrix.

Figure 5 shows the performance of the concurrent algorithm on complete graphs of small size. We can clearly see a linear increment in time when we

put more threads at work together. Another obvious fact is that once the graph becomes larger, the algorithm is more tolerable for more threads (i.e. the linear increment point moves to the right, gradually). These facts are more significant when the algorithm runs on the complete graph with 10,000 nodes and 30,000 nodes as illustrated in Figure 6.

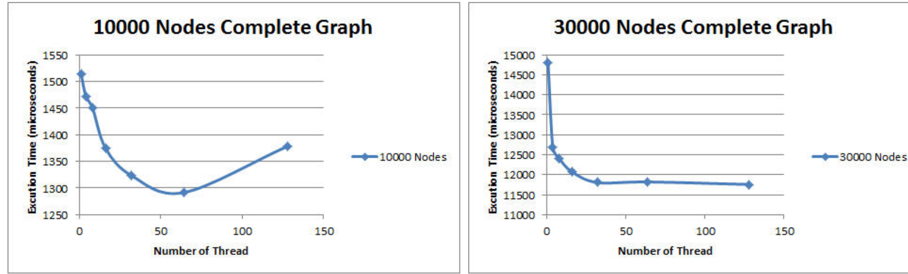

Fig. 6. Execution time on complete graph stored in int matrix with 10,000 nodes and 30,000 nodes using different threads.

As shown in Figure 6, when the algorithm is running on the graph with 30,000 nodes, the linear increment point may move to the very right part, and can't be seen in this diagram. What's more, we can't really see more improvements in the execution time in the same graph after we use more than 32 threads. This is because the algorithm will gain speed up if and only if those threads run in parallel. Furthermore, the machines in the Intel® Manycore Testing Lab only have 32 cores. Thus, even if we create more and more threads to run the algorithm, there are at most 32 threads who are really computing at the same time.

5.2 Graph in Adjacency-List

If we store the graph using an adjacency list, it costs linear time to access an edge in the graph. More realistically, we use the TreeMap<Integer,Integer> which is implemented by a red-black tree in Java, to store the edge information. Thus, it costs $O(\log|V|)$ time to visit an element in the red-black tree, which makes the right term in the complexity formula dominate. However, in this case, we can't store too many edges by using TreeMap<Integer,Integer>, since even with a 8,000 nodes complete graph, it needs 8 Gigabytes to store the edge information.

As we can see from Figure 7, even though the size of the graph is not that large, the algorithm still gets a speed up with a complete graph of 1,000 nodes. What's more, as the size of the graph increases, we can save more time by using more threads.

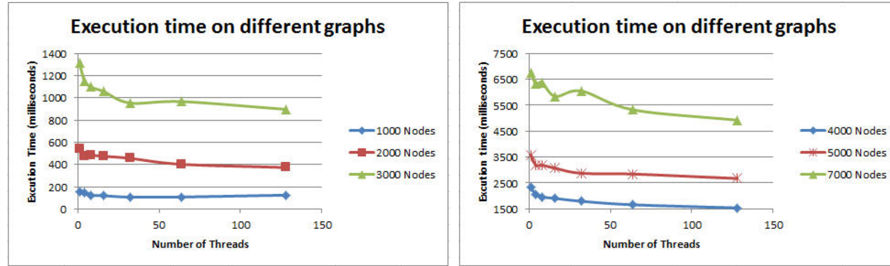

Fig. 7. Execution time on complete graph with 1,000-7,000 nodes using different threads on TreeMap.

6 Testing and Verification

We try both the testing method and the model checking method to verify the correctness of our implementation. Firstly, we directly check if the program obtains the correct result based on the MST properties. A sequential version of Kruskal's algorithm [2] is used to compute the standard weight of the MST in a graph. Then, we run our concurrent implementation on the same graph, testing if the final result matches to the one computed by Kruskal's algorithm. If the two values are the same, we then check the edge list obtained by the concurrent program whether it satisfies the spanning tree properties or not. The properties of spanning tree are already defined in Section 1. Since we have an edge list stored in the final thread, the way to verify it is simple. We firstly make $|V|$ sets corresponding to $|V|$ nodes, and union $|V|-1$ times according to the two end points of an edge in the edge list. Finally if every time the sets we union are disjoint, then the edges in the list form a spanning tree.

We tested our implementation using 80 threads on the machine in the Intel® Manycore Testing Lab, which has 32 cores, on two graphs. The first one is a complete graph with 1,000 nodes and 499,500 edges and the second graph has 2,000 nodes and 1,999,000 edges. We run the test 100,000 times on each graph and eventually find a small bug which causes a null pointer exception. That happens at *line 10* in $MSThread$, where even we checked the priority queue is not empty in *line 9*, some interleavings can make it become empty while executing *line 10*. Once we put *line 10* in a critical section and check again if the priority queue is not empty, the program pass all the tests.

Secondly, we also model check our implementation to detect if there are any uncaught exceptions, null pointer exceptions, data races or deadlocks. Since our implementation is written in Java, we choose a model checking tool called Java PathFinder (JPF) [13] to directly check our Java code.

JPF was developed at the NASA Ames Research Center and was open sourced in 2005. The core of JPF is a particular Java Virtual Machine (JVM)[1] that is also implemented in Java. That JVM tries to identify the points where the execution of the program becomes differently. And it stores, matches and restores such program states in order to explore all of its execution paths. Moreover, JPF is useful to check concurrent programs, because it produces all kinds of interleavings of the code. Thus, by adding some useful listeners, we can detect a lot of defects such as data races and deadlocks automatically.

JPF does have a lot of advantages. However, as a model checking software, it also suffers from the state space explosion problem, which is a bottleneck for checking complicated concurrent programs. In our experiment, we use the PreciseRaceDetector[2] listener combined with different search strategies to explore all the states of the program and detects data races and deadlocks. Even though we try some methods to reduce the state space of our program and achieve a relative good result, JPF still get an out of memory error when it searches all the execution paths of our program for which only two threads are used on a four nodes graph. The following information shows the result.

```
===========================================   results
error #1: gov.nasa.jpf.jvm.NoOutOfMemoryErrorProperty
===========================================   statistics
elapsed time:       9:56:57
states:             new=38636247, visited=53467377,
                    backtracked=92103147, end=1803
search:             maxDepth=813, constraints hit=1
choice generators:  thread=38635883 (signal=0, lock=59258,
                    shared ref=32751065), data=0
heap:               new=5430939, released=13942116,
                    max live=826, gc-cycles=86662523
instructions:       -940434119
max memory:         1351MB
loaded code:        classes=125, methods=1916
```

As we can see, about 38 million execution states have been visited using around 10 hours and then JPF gets an out of memory error. However, since JPF will terminate as long as it detects any defects, our implementation seems safe (i.e. without uncaught exception, deadlocks nor data races) at least for those checked 38 million states.

[1] A JVM running on top of standard JVM.
[2] http://www.cse.yorku.ca/∼franck/research/drafts/race.pdf

7 Conclusion

We have presented the implementation of a parallel algorithm derived from Prim's algorithm based on the paper published by Setia et al. [4]. Also, three interesting heuristics which can make the algorithm perform better are discussed here. It takes $O(P \cdot |V| \log |V| + \frac{|E|}{P})$ time for this parallel implementation to find the MST in a graph. Thus, it will become useful only when the right term dominates. Therefore, we can use this algorithm if the graph is relatively dense or the cost for accessing edges is expensive to achieve some speed up. Perhaps if people want to deal with the Internet, which has a huge number of edges (URLs) and accessing edges is expensive, this concurrent algorithm will be useful.

The verification experiments of our implementation are crippled by the notorious state space explosion problem, when we use a model checking tool, JPF, to verify some properties of using multiple threads. Although, we can't fully explore all the execution paths of the program, we roughly believe there is no deadlocks nor data races in our implementation, considering the states that we have verified and the testing experiments we did. Also in the original paper presented by Setia et al. [4], they have already shown the proof of the correctness of the parallel algorithm based on the cut property of the MST. The key part of that proof can be found in the Appendix.

8 Acknowledgements

Thanks to Professor Franck van Breugel who provided us with a marvelous opportunity to expose ourselves to the concurrent computing world. Thanks to the management, staff, and facilities of the Intel® Manycore Testing Lab[3]. Thanks to Trevor Brown who gave a nice talk about how to use MTL and who discussed the memory management of Java with me. Thanks to all of my classmates.

References

1. Borůvka, O.: O jistému problému minimálním. Prúce Mor. Přírodovědecké. Spol. v Brně (in Czech.) **3** (1926) 37–58
2. Kruskal, J.B.: On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. Proceedings of the American Mathematical Society **7**(1) (February 1956) 48–50
3. Prim, R.C.: Shortest connection networks and some generalizations. Bell System Technology Journal **36** (1957) 1389–1401
4. Setia, R., Nedunchezhian, A., Balachandran, S.: A new parallel algorithm for minimum spanning tree problem. In: Proceedings of International Conference on High Performance Computing. (2009) 1–5
5. Chung, S., Condon, A.: Parallel implementation of Borůvka's minimum spanning tree algorithm. In: Proceedings of the 10th International Parallel Processing Symposium, Washington, DC, USA, IEEE Computer Society (1996) 302–308

[3] `www.intel.com/software/manycoretestinglab`

6. Chong, K.W., Han, Y., Igarashi, Y., Lam, T.W.: Improving the efficiency of parallel minimum spanning tree algorithms. Discrete Applied Mathematics **126**(1) (2003) 33–54
7. Bader, D.A., Cong, G.: Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. J. Parallel Distributed Computing **66** (November 2006) 1366–1378
8. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing. Addison Wesley (2003)
9. Gonina, E., Kalé, L.V.: Parallel Prim's algorithm on dense graphs with a novel extension, Department of Computer Science, University of Illionois at Urbanan-Champaign, USA, Academic Press, Inc. (November 2007)
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. The MIT Press, Cambridge, MA, USA (1990)
11. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM **34**(3) (1987) 596–615
12. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. Journal of the ACM **22**(2) (1975) 215–225
13. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering **10**(2) (April 2003) 203–232

A PROOF OF THE CORRECTNESS[4]

A.1 Lemma 1

Lemma 1. *No cycles are formed during MergeTree operation.*

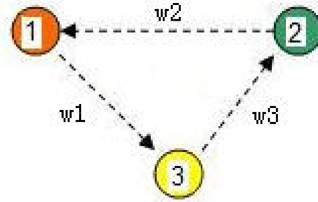

Fig. 8. Illustration to prove Lemma 1

Proof: The simple and typical cycle that can be formed is of length 3. In Figure 8, let's assume the cycle is created during MergeTree() operation. Then,

> $w_3 < w_1$ (as for tree 3, it selects w_3 as its light edge) $w_2 < w_3$ (as for tree 2, it selects w_2 as its light edge) $w_1 < w_2$ (as for tree 1, it selects w_1 as its light edge) $\Rightarrow w_1 < w_2 < w_3 < w_1$

It is a clear contradiction. So such case is not possible. However, if $w_1 = w_2 = w_3$ then we get merge requests MergeTree(1,2), MergeTree(2,3), MegerTree(1,3). As we do MergeTree in the critical section, only 2 out of 3 requests are actually executed and the remaining request is not granted in *lines 38-39* in the algorithm shown in section 3.2. The proof can be extended to cycles of any length greater than 3. Hence, no cycles are formed.

A.2 Lemma 2

Lemma 2. *The edges added by the algorithm belong 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 $E(v_1, v_2)$ is the lowest cost edge joining a node in tree i with a node in tree j, then

$$MST(t_1) + MST(t_2) + E(v_1, v_2) = MST(t_1 \cup t_2)$$

if and only if $E(v_1, v_2)$ has its weight larger than all the edges in $MST(t_1)$ or $MST(t_2)$. In the algorithm we presented here, the edges that are added during MergeTree operation always satisfy this property. To prove that, consider that an edge e exists whose weight is less than the weight of some edges in either of the trees. Then e would have been added already to that tree in *line 7* in the algorithm shown in section 3.2. Hence no such edge exists and thereby $E(v_1, v_2)$ is the next lowest cost edge. Therefore, all the edges chosen by the algorithm belongs to MST.