Parallel implementation of Prim's Algorithm for computing a Minimum Spanning Tree

Gpu Computing project's report

Antonio Pugliese 987249

Contents

1 Introduction

[1] A spanning tree of a connected undirected graph G is a subgraph of G that is a tree containing all the vertices of G. If the graph is not connected it is called spanning forest.

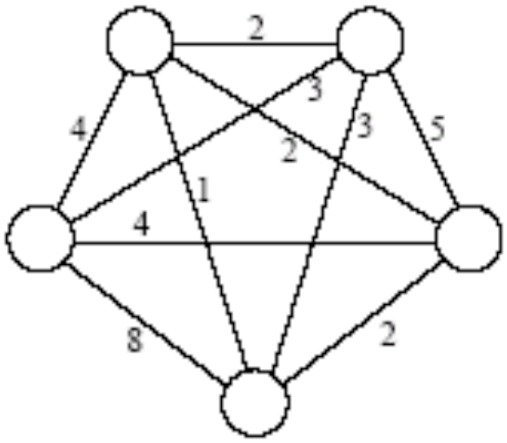
The weight of a subgraph in a weighted graph is the sum of the weights of the edges in the subgraph.

The properties of a spanning tree are:

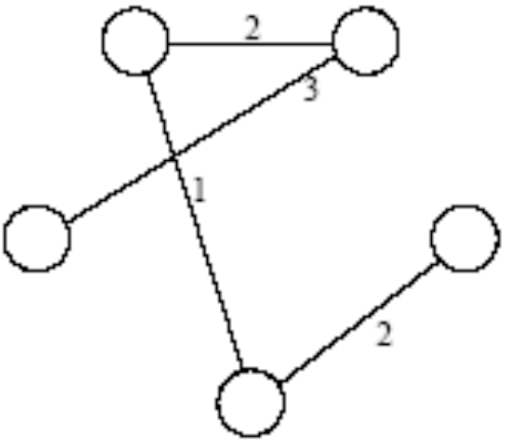
- The number of vertices (V) in the graph and the spanning tree is the same.
- There is a fixed number of edges in the spanning tree which is equal to one less than the total number of vertices (E = V-1).
- The spanning tree should not be disconnected, as in there should only be a single source of component, not more than that.
- The spanning tree should be acyclic, which means there would not be any cycle in the tree.
- The total cost (or weight) of the spanning tree is defined as the sum of the edge weights of all the edges of the spanning tree.
- There can be many possible spanning trees for a graph.

The minimum spanning tree (MST) has all the properties of a spanning tree with an added constraint of having the minimum possible weights among all possible spanning trees. Like a spanning tree, there can also be many possible MSTs for a graph.

An example of a MST can be seen in the following figure:



Undirected graph



Minimum spanning tree

2 Prim's algorithm

There are several algorithms that can be applied to find a minimum spanning tree, one of these being Prim's algorithm. In this report two versions will be presented: a sequential version and a parallel version.

2.1 Sequential version

Prim's algorithm falls under the category of greedy algorithms; this algorithm always starts with a single node and moves through several adjacent nodes, in order to explore all of the connected edges along the way.

The algorithm starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

The main elements used during the computation are:

- set of vertices *V*,
- set of edges *E*,
- weight of the vertices *w*,
- root *r*, the starting vertex of the MST.

We can divide the algorithm in two main passages:

- Initialization
- Main loop

During the initialization phase, the spanning tree vertices V_T are initialized with vertex *r*, the designated root. After this, *d[:]* is computed, which is the weight between *r* and each vertex outside V_T.

After the initialization phase, the main loop begins and ends when there are no more vertices outside V_T. At each iteration of the loop, we use *d[:]* to find *u*, the closest vertex to V_T and we add it to V_T. After this, we recompute *d[:]*, considering the fact that *u* is now in V_T.

In more detail, it may be implemented following the pseudocode below [1]:

```
1.   procedure PRIM_MST(V, E, w, r)
2.   begin
3.       V_T := {r};
4.       d[r] := 0;
5.       for all v ∈ (V − V_T) do
6.           if edge (r, v) exists set d[v] := w(r, v);
7.           else set d[v] := ∞;
8.       while V_T ≠ V do
9.       begin
10.          find a vertex u such that d[u] := min{d[v]|v ∈ (V − V_T)};
11.          V_T := V_T ∪ {u};
12.          for all v ∈ (V − V_T) do
13.              d[v] := min{d[v], w(u, v)};
14.      endwhile
15.  end PRIM_MST
```

2.1.1 Complexity

In the implementation considered in this report, the worst case scenario happens when all the elements are considered for searching and marked as suitable edges. In this situation the complexity will be $O(|V|^2)$. There exist, though, data structures which can be used in order to reduce the total time complexity of the algorithm, for example binary heap or priority queues.

2.2 Parallel version

The main loop of Prim's algorithm is inherently sequential and thus not parallelizable since adding two vertices to MST concurrently is problematic. However, the inner loop, which determines the next edge of minimum weight that does not form a cycle, can be parallelized by dividing the vertices and edges between the available processors. The following pseudocode demonstrates this.

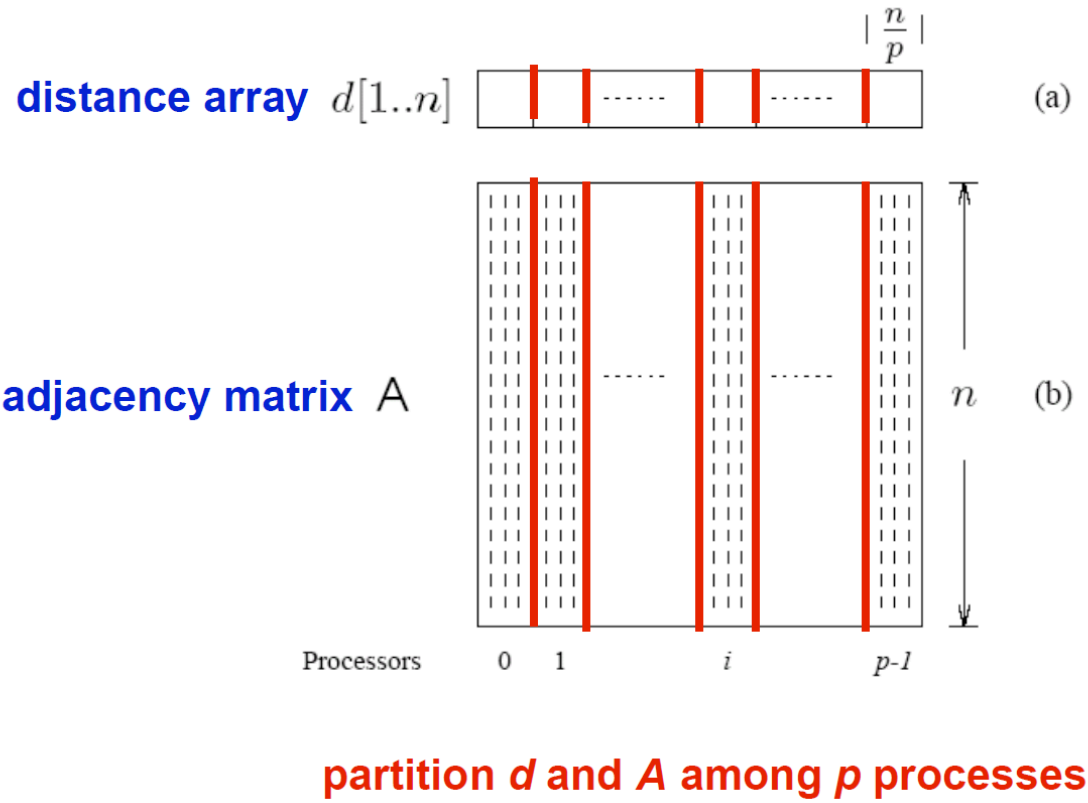
1. Partition adjacency matrix in a 1-D block fashion (blocks of columns). Assign each processor $\boldsymbol{P_i}$ a set $\boldsymbol{V_i}$ of consecutive vertices of length $\boldsymbol{|V|/|P|}$.
2. As in the sequential algorithm, create an empty forest ***F*** and a set ***Q*** of vertices that have not yet been included in ***F***. Then, divide ***C*** (vector of the minimum costs from ***F*** to each node) and ***E*** (vector of the edges providing that cheapest connection) as well as the graph between all processors such that each processor holds the incoming edges to its set of vertices. Let $\boldsymbol{C_i}$, $\boldsymbol{E_i}$ denote the parts of ***C***, ***E*** stored on processor $\boldsymbol{P_i}$.
3. Repeat the following steps until ***Q*** is empty:
    a. On every processor: find the vertex $\boldsymbol{v_i}$ having the minimum value in $\boldsymbol{C_i[v_i]}$ (local solution.
    b. Min-reduce the local solutions to find the vertex ***v*** having the minimum possible value of ***C[v]*** (global solution).
    c. Broadcast the selected node to every processor.
    d. Add *v* and ***E[v]*** to ***F***.
    e. On every processor: update $\boldsymbol{C_i}$ and $\boldsymbol{E_i}$ as in the sequential algorithm.
4. Return ***F***

2.2.1 Complexity

The costs of each part of the parallel version of Prim's algorithm are:

- Cost to select the minimum entry:
    - O(n/p): scan n/p local part of d vector on each processor.
    - O(log p): all-to-one reduction across processors.
- Broadcast next node selected for membership:
    - O(log p).
- Cost of locally updating d vector:
    - O(n/p): replace d vector with the minimum of d vector and matrix row.
- Parallel time per iteration:
    - O(n/p + log p).

So in the end, we find that the total parallel time is $O(n^2/p + n \log p)$.



3 Implementation

First of all, we need to develop a random graph generator. This is used for both the sequential and the parallel versions. The generator takes as input the number of nodes and returns a vector of vectors of pairs. In this way, each node is associated with a vector of pair *<node, weight>*, representing each edge of the considerate vertex and its weight. The code also guarantees that the generated graph is connected and acyclic.

In order to make the code regarding the graph more readable, the two following alias has been defined:

```
typedef pair<int, int> Edge;
typedef vector<vector<Edge>> Graph;
```

3.1 Sequential version

The sequential version of Prim's algorithm is implemented by the function

```
Graph sequential_prim_MST(const Graph& graph, int n)
```

which takes as parameters the random generated graph and the number of node sand returns the minimum spanning tree.

The data structures used for the computation are

`vector<int> key(n, INT_MAX)`: key values used to pick minimum weight edge, initialized as the maximum possible int value.

`vector<int> mst(n, -1)`: the nodes in the minimum spanning tree.

`vector<bool> notInMst(n, true)`: false if the vertex is in the MST, true otherwise.

Prim's algorithm requires choosing a root from which to start forming the minimum spanning tree. In this implementation, the node 0 will be the root and this is done by the operations

```
key[0] = 0;
mst[0] = -1;
```

The main loop iterates as much as the number of nodes of the generated graph. The first step is to identify the minimum weight vertex from the set of vertices not yet included in MST. This is done by the function

```
int minWeightVertex(int n, vector<int>& key,
                    vector<bool>& notInMst) {
    int minWeight = INT_MAX;
    int minWeightVertex;
    for (int v = 0; v < n; v++) {
        if (notInMst[v] && key[v] < minWeight) {
            minWeight = key[v];
            minWeightVertex = v;
        }
    }
    return minWeightVertex;
}
```

After this, the picked vertex is added to the MST set, by setting to false the value in the `notInMst` at the node's position.

In the end, the inner loop takes care of the update of the key values and MST indexes of the adjacent vertices of the picked vertex, considering only those vertices which are not yet included in MST.

```
for (const Edge& e: graph[u]) {
      int v = e.first;
      int weight = e.second;
      if (notInMst[v] && weight < key[v]) {
          mst[v] = u;
          key[v] = weight;
      }
}
```

3.1 Parallel version

As said before, the main loop is not parallelizable, but the inner one is. At each step, we call two kernels. The first one is

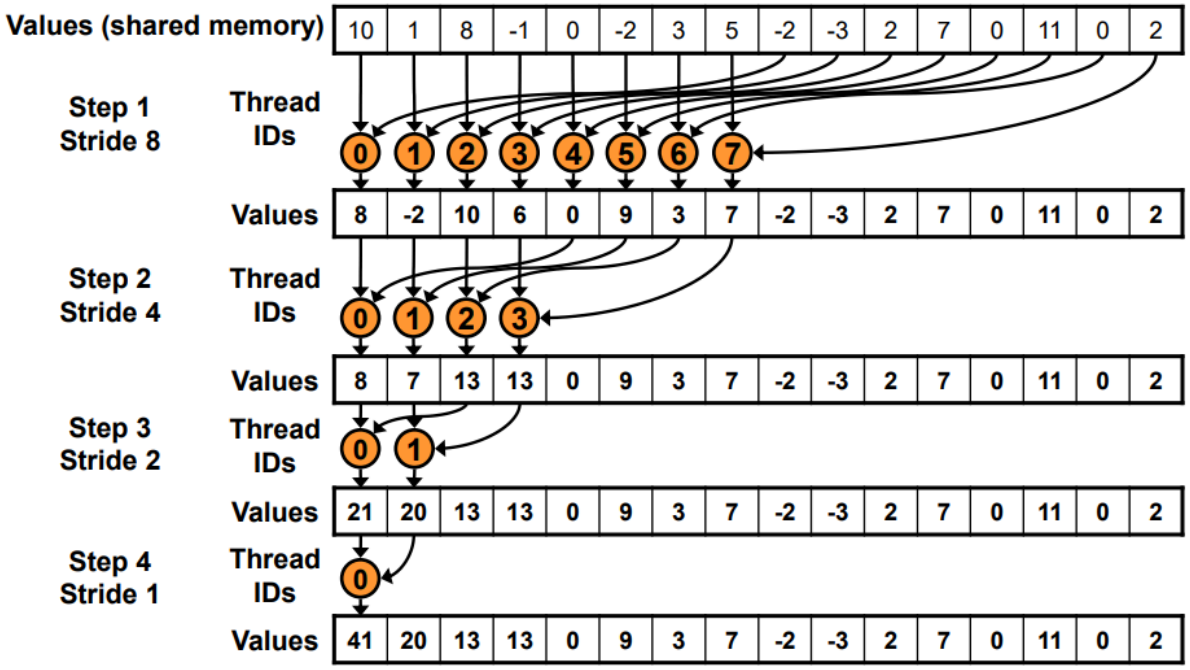
```
__global__ void local_closest_node(const int *d_distance_vector,
                   int *d_min_weights, int *d_min_nodes,
                   const bool *d_present_in_mst)
```

which is in charge of finding the local nodes closest to the MST. The arrays `d_min_nodes` and `d_min_weights` contain respectively the local closest nodes and the weight of its connection to the MST. These arrays are then returned to the host who will find the global closest node.

```
__shared__ int min_weight[NODES];
__shared__ int closest_node[NODES];

int tid = threadIdx.x;
int index = blockIdx.x * blockDim.x + threadIdx.x;

// Load data into shared memory
min_weight[tid] = d_distance_vector[index];
closest_node[tid] = index;
__syncthreads();

// Ignore the elements that are already present in MST
if (d_present_in_mst[index]) {
   // Set to maximum value to avoid selection
   min_weight[tid] = INF; // INF = INT_MAX
}
__syncthreads();
```

The shared memory initialization contains a check in order to not consider the nodes already in the MST by setting their distance to infinite.

To find the local closest node and its index in an optimized way, the Parallel Reduction technique has been used. Parallel Reduction is a strategy to perform quickly some operations that output a single value from the computing of an array of values. The idea is to maximize the usage of a parallel processor, by reducing the original array in smaller and smaller parts, with each thread evaluating two values per cycle. At the end of the operation the result will lay in the first cell of the array. In particular, the Sequential Addressing approach has been implemented. A scheme of this strategy can be seen in the following figure:



Scheme for the sum of the elements of an array using Parallel Reduction: Sequential Addressing [2].

Since in this case the index of the minimum weight is also needed, the cost of the operations is higher since the number of comparisons that a thread must perform for every cycle is now two instead of one. After the reduction, the minimum value and its index are stored in global memory.

```
for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1) {
    if (tid < s) {
        if (min_weight[tid + s] < min_weight[tid]) {
            min_weight[tid] = min_weight[tid + s];
            closest_node[tid] = closest_node[tid + s];
        }
    }
    __syncthreads();
}

if (tid == 0) {
    d_min_weights[blockIdx.x] = min_weight[0];
    d_min_nodes[blockIdx.x] = closest_node[0];
}
```

After finding the local solutions, we copy the arrays containing the minimum nodes and the weights to the host

```
int *h_min_weights = new int[num_blocks];
int *h_min_nodes = new int[num_blocks];
cudaMemcpy(h_min_weights, d_min_weights, num_blocks *
     sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(h_min_nodes,     d_min_nodes,     num_blocks     *
sizeof(int),
```

```
            cudaMemcpyDeviceToHost);
```

The size of `h_min_weights` and `h_min_nodes` are the same as the number of local solutions, that is the number of blocks launching the kernel. Since this number is relatively small (e.g if the graph has 4096 nodes and we utilize 1024 threads per block, we will only have 4 blocks), we can perform the search of the global closest node host side; avoiding to allocate new resources on the device and not launching a new kernel should improve the performances. This is simply done by iterating the two arrays

```
      int global_min_weight = h_min_weights[0];
      int global_min_node = h_min_nodes[0];

       for (int k = 1; k < num_blocks; ++k) {
             if (h_min_weights[k] < global_min_weight) {
                 global_min_weight = h_min_weights[k];
                 global_min_node = h_min_nodes[k];
             }
       }
```

In the end, we finally call the second kernel,

```
update_distances<<<num_blocks, BLOCK_SIZE>>>(
                                d_matrix,  d_mst,  d_distance_vector,
final_min_node,

thrust::raw_pointer_cast(d_present_in_mst.data()));
```

The logic behind this kernel is the same as the update of the costs in the sequential version. In the parallel version we use the adjacency matrix, partitioning it in a 1-d block fashion and assigning each of them to each block of threads in order to accelerate the process. After updating the distance vector, we insert the considered node in the MST.

```
__global__ void update_distances(
           const int *d_matrix, int *d_mst, int *d_distance_vector,
           int final_min_node, const bool *d_present_in_mst) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < NODES && tid != final_min_node &&
           !d_present_in_mst[tid]) {
        int index = final_min_node * NODES + tid;
        if (d_matrix[index] < d_distance_vector[tid]) {
            d_distance_vector[tid] = d_matrix[index];
            d_mst[tid] = final_min_node;
            d_present_in_mst[final_min_node] = true;
        }
    }
}
```

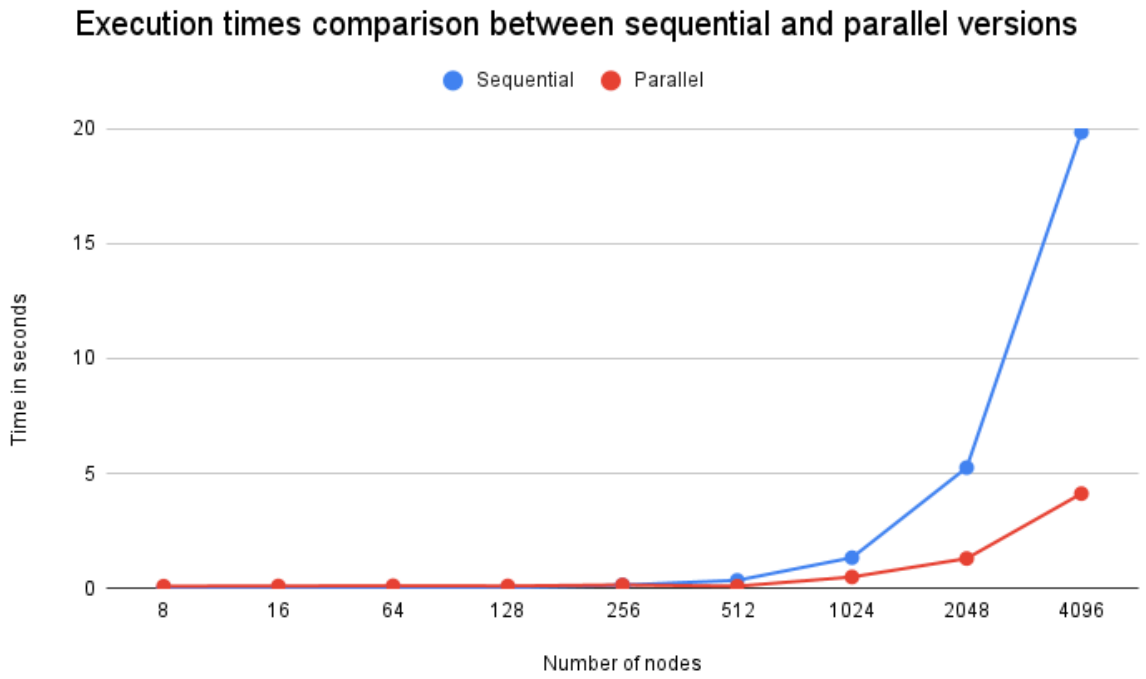
4 Speedup and Profiling

The tests have been performed on a machine equipped with a processor AMD Ryzen 7 3750H with core frequency of 2.3 GHz and a GPU NVIDIA GeForce GTX 1650. The GPU has the following characteristics:

- Compute Capability: 7.5
- Total CUDA Cores: 896
- Clock: 1560 MHz
- Memory: 4096 MB GDDR6
- Max Threads per Block: 1024
- Max Blocks per Multiprocessor: 16
- Max Shared Memory size per Block: 49152 bytes

4.1 Speedup

Since the algorithm, both the sequential and the parallel versions, does not work well with sparse graphs, for the tests it is guaranteed that the random generated graph is connected at least for 75%. To measure the execution time, for the sequential algorithm, we used the C++ Chrono library, while for the parallel version we took advantage of the CUDA Events. The performances in terms of execution time have been summarized into the following chart:



The algorithms could not be tested on a higher number of nodes due to the testing machine limitations. The resulting speedups are shown below:

Number of nodes	Number of edges	Speedup
64	3.280	0,07
128	14.130	0,25

256	55.246	0,96
512	237.948	3,86
1024	894.500	2,70
2048	3.354.644	4,05
4096	14.832.296	4,82

As expected, increasing the number of nodes, the speedup improves as well. In particular, the parallel version outperforms the sequential version starting from about 256 nodes.

4.2 Profiling

Profiling has been performed using the NVIDIA Visual Profiler and NVIDIA Nsight Compute tools. The following data has been collected executing the algorithm on a graph composed of 1024 nodes.

Kernel name	local_closest_node	update_distances
Invocations	1024	1024
Average duration (µs)	3,382	1,637
Static Shared Memory	8192	0
Average Dynamic Shared Memory	8192	0
Compute Throughput [%]	1.13	1.02
Memory Throughput [%]	1.34	4.05
Theoretical Occupancy [%]	100	100
Theoretical Active Warps per SM [warp]	32	32
Achieved Occupancy [%]	97,82	91,37
Achieved Active Warps Per SM [warp]	31,32	29,24

5 Conclusion and future work

In this report a parallel implementation of Prim's algorithm for finding a minimum spanning tree in a graph has been presented. A comparison in terms of execution times with graphs of different numbers of nodes has also been presented, with positive results in terms of speedup and a positive trend as the number of nodes and edges increases. Unfortunately, because of the limitation of the testing machine, the algorithms could not be tested on graphs with more than 4096 nodes.

The parallel algorithm presented in this report does not work particularly well on sparse graphs. Some improvements can be implemented, for example using a heap to maintain

costs. Moreover, it follows the naive approach of choosing a node as root of the MST. This sometimes can be not optimal, so it is worth noticing that there exists more sophisticated variants of Prim's algorithm for shared memory machines, e.g. [3] in which Prim's sequential algorithm is being run in parallel, starting from different vertices, using the cut property of graphs to grow trees simultaneously in multiple threads and a merging mechanism when threads collide.

6 References

[1] Vivek Sarkar. Parallel Graph Algorithms. 2008, https://www.cs.rice.edu/~vs3/comp422/lecture-notes/comp422-lec24-s08-v2.pdf

[2] Optimizing Parallel Reduction in CUDA, Mark Harris, NVIDIA Developer Technology, https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

[3] Setia, Rohit (2009), "A new parallel algorithm for minimum spanning tree problem", Proc. International Conference on High Performance Computing (HiPC).