Georgios Piskas

georgios.piskas@epfl.ch

SCIPER: 244587

21 Dec 2014

Parallelizing the Minimum Spanning Tree Problem

Project Report - Program Parallelization on PC Clusters

1. Introduction

The Problem

A Spanning Tree is a subgraph of the input graph that contains all vertices, but it forms a tree structure meaning that it does not contains cycles. Apparently, one can derive multiple different spanning trees from the same graph that satisfy the above statement.

In a scenario where the input is an undirected and weighted graph, each unique Spanning Tree can be characterized by its length. This length is the sum of the weights of all the edges that form the tree. Thus, the problem is to find the tree that has the minimum length, hence the Minimum Spanning Tree. Assuming that all edges have distinct weights, there exists only one Minimum Spanning Tree.

The Solution

Several different approaches have been proposed to solve the Minimum Spanning Tree problem. However, there are three classical techniques from which the research on the problem was born and on which most modern approaches are built. These three are Borůvka's [1], Prim's [2] and Kruskal's [3] algorithms.

This project is specifically focused on the parallelization of Kruskal's algorithm. The parallel version of Kruskal's algorithm is implemented in C programming language using *MPI* (Message Passing Interface) [4] and *OpenMP* (Open Multi-Processing) [5] for the parallelization of local tasks such as sorting of data.

2. Kruskal's Algorithm

The Algorithm

The basic, simplified idea of Kruskal's algorithm is as follows:

1. Treat each vertex of the graph as a tree (1) in the forest (the set of trees).
2. Sort the edges of the graph in increasing weight order.
3. Scan the edges in increasing order.

   3.1. If this edge connects two distinct (2) trees, combine (3) the trees into one

Before formalizing the above description, let us mention what assumptions and data structures it uses.

Notations, Assumptions and Data Structures

The input graph $G = (V, E)$ is a weighted, undirected and connected graph. It consists of $|V|$ vertices and $|E|$ edges. Let us denote the weight between two connected vertices u and v as $w(u, v)$.

As stated in the introduction, we assume that all edges have distinct weights in order to simplify the description and the implementation of the algorithm, without loss of generality. If we would like to support non-distinct weight values, we could use a combination of the weight and the connected node identifier (denoted *ID*) to distinguish each edge. Then, we could prioritize the selection of the same-weight edge that leads to the lowest node ID. It is a matter of assumption.

The last point that needs to be defined is the Disjoint-set data structure [6], which is vital for the functionality of the algorithm.

- In step 1 of the previous naive description, denoted with (1), we need to create a tree, or equivalently a set, for each vertex of the graph. This operation is called *MakeSet(v)*. *MakeSet* trivially creates a set containing the initial vertex assigned to it, which eventually acts as the "root" of the set.
- The second operation used, denoted with (2), is called *Find(v)*. *Find* determines in which subset is vertex *v* in. This operation is typically implemented by returning the "root" element of the set that *v* belongs to.
- The third operation used, denoted with (3), is called *Union(u,v)*. *Union* merges the given sets under a common "root".

The project implementation uses two optimizations of the Disjoint-set data structure, namely path compression and union by rank. It can been shown that this optimized data structure can perform its three operations in $O(E \cdot a(V))$ time, where $a(V)$ is the inverse of the Ackermann function, given that the edges are already sorted. Since the details of this data structure and its complexity analysis are out of the scope of this project, interested readers can refer to [6].

Formal Definition and Complexity Analysis

The pseudocode of Kruskal's algorithm in its serial version is given below. The *mst* set will finally contain all the edges that form the Minimum Spanning Tree. Clearly, the Disjoint-set data structure is the core of the implementation.

```
Kruskal( G = (V, E) ):
1. mst = ∅;
2. sort E in increasing weight order;
3. for each vertex v in V:
4.     MakeSet(v);
5. for each edge e = (u, v) in E:
6.     if Find(u) ≠ Find(v):
7.         mst = mst ∪ {(u, v)};
8.         Union(u, v);
9. return mst;
```

In a nutshell, the complexity of this algorithm derives from the sorting of the edges at line 2 and the for loop at line 5. A more detailed per-line explanation follows.

1. ***O(1)****:* Initialization of an empty set.
2. ***O(E·log(E))****:* Sorting of all edges in *E*.
3. ***O(V)***: Initialization of a singleton set for each vertex in *V*.
4. ▲ *MakeSe*t is ***O(1)***.
5. ***O(E·log(V))***: *Union* and *Find* operations for each edge in *E*.
6. ▲ *Find* is ***O(1)***.
7. ▲ Appending an element is ***O(1)***.
8. ▲ *Union* is ***O(log(V))***.
9. ***O(1)***: Return.

We now formally prove that the upper bound of the time complexity of Kruskal's algorithm is *O(E·log(V))*. At this point it should be noted that in the worst case (complete graph), the maximum amount of edges is $|E| \leq (|V|\cdot(|V| - 1))/2 \leq |V|^2/2 - |V|/2 \Rightarrow O(E) = O(V^2)$.

- $$T(\ G = (V, E)\ ) = O(1) + O(E\cdot\log(E)) + O(V) + O(E\cdot\log(V)) + O(1)$$
$$= O(E\cdot\log(E)) + O(E\cdot\log(V)) \quad \textbf{\{1\}}$$

- $$\text{Since } O(E) \leq O(V^2)$$
$$\Rightarrow O(\log(E)) \leq O(\log(V^2))$$

$$\Rightarrow O(\log(E)) \leq O(2 \cdot \log(V))$$
$$\Rightarrow O(\log(E)) \leq O(\log(V)) \quad \{2\}.$$

- $\{1\}$ using $\{2\} \Rightarrow T(G) = O(E \cdot \log(V)) + O(E \cdot \log(V))$
$$\Rightarrow \mathbf{T(G) = O(E \cdot \log(V))} \blacksquare$$

It should be noted that *O(E·log(E))* complexity is equivalent to *O(E·log(V))* due to **{2}**, but we prefer the second version since is provides a tighter upper bound. This "linearithmic" time complexity hints that it will be challenging to achieve a significant speedup by parallelizing this algorithm, in contrast to other approaches that are bounded by quadratic time complexity.

Execution Example

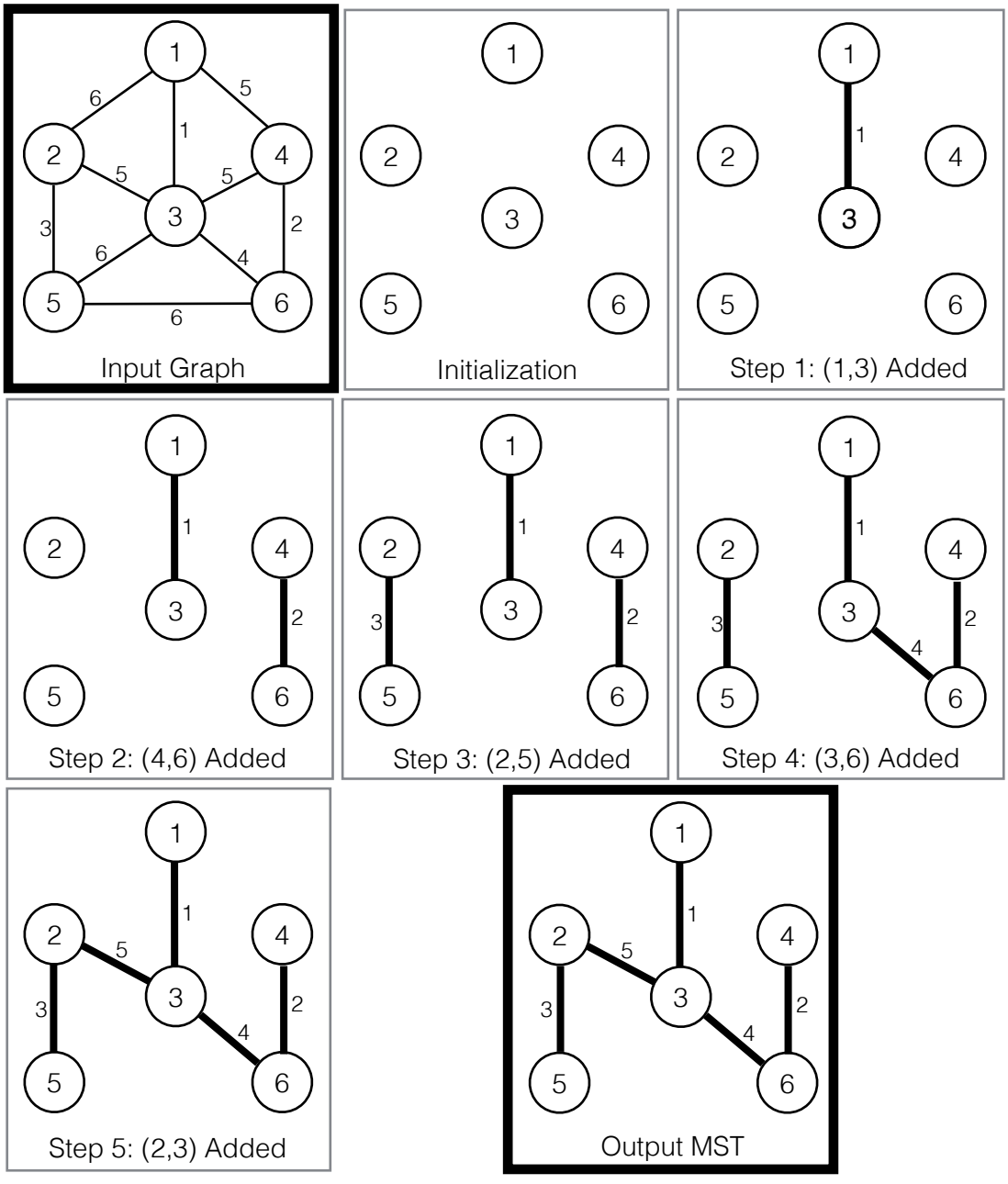


Figure 1: Kruskal's algorithm execution example.

The example's input is a weighted, connected and undirected graph. Each step of the algorithm is shown in each box. The end result is the Minimum Spanning Tree of the graph.

Amdahl's Law

Before diving into parallelization strategies and analysis, we can first try to provide an upper bound of the achievable speedup using Amdahl's law. The maximum speedup that can be achieved based on this metric is given by the following equation, where p is the number of processing nodes and f is the percentage of the algorithm that cannot be parallelized. We now need to informally estimate the parameter f using a sketch of a proof in combination with the actual timing measurements of the serial execution.

$$Sp \leq \frac{1}{f + \frac{1-f}{p}}$$

Suppose we use a simple split - process - merge strategy to parallelize the application. Then, the strictly serial part is when parsing the input and sending parts of it to the processors, and also when merging the partial MSTs that will be created by the parallel processing. The input parsing requires $O(E) = O(V^2)$ time, since we read all edges into memory from the input file, and the input distribution requires an additional $O(V^2)$. The merging of the partial MSTs requires $O(V^2 \cdot log(V))$, since we must execute Kruskal's algorithm once more, in order to eliminate possible circles that the merging will create. However, in this phase, the number of edges will be much closer to $|V|$ rather than $(|V| \cdot (|V|-1))/2$ since the majority of the initial edges has already been eliminated. This is the simplest merging solution, since this step can be implemented in a more efficient way.

The parallel part of the application consists of the sorting, the initialization of the input nodes and the computation of the local MSTs. The sorting of the edges requires $O(E \cdot log(E)) = O(V^2 \cdot log(V^2)) = O(V^2 \cdot log(V))$ time, the initialization of the nodes using *MakeSet* requires $O(V)$ time and the MST processing also requires $O(V^2 \cdot log(V))$ time.

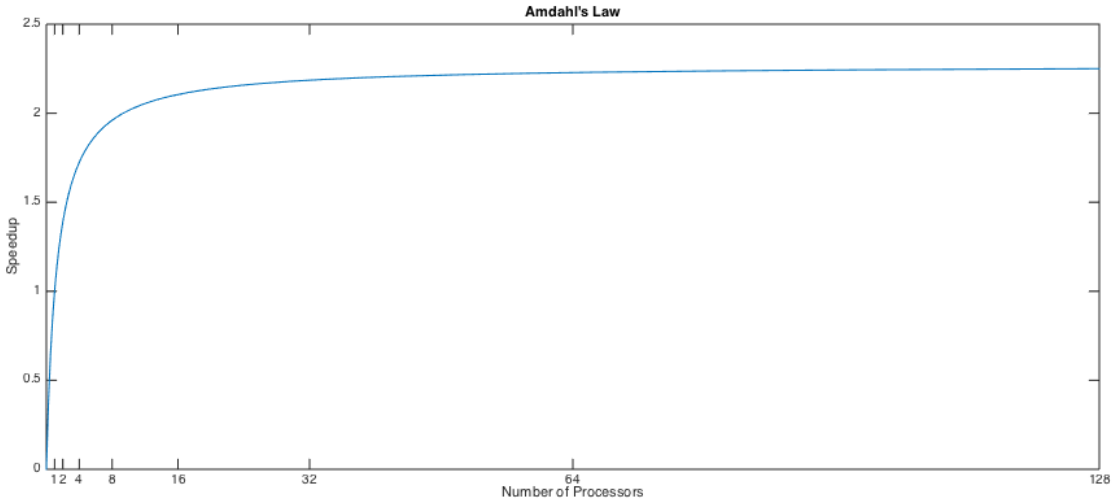
To sum up:

- Serial part: $2 \cdot O(V^2) + O(V^2 \cdot log(V))$.
- Parallel part: $2 \cdot O(V^2 \cdot log(V)) + O(V)$.
- Total: $2 \cdot O(V^2) + 3O(V^2 \cdot log(V)) + O(V)$.

In order to estimate the percentage of the program that is strictly serial, we can assign "weights" to each notation. Suppose that "$O(V)$ is 10", "$O(V \cdot log(V))$ is 33", "$O(V^2)$ is 100" and "$O(V^2 \cdot log(V))$ is 332". Then we get:

- Serial part: 532 units or 44%.
- Parallel part: 674 units or 56%.
- Total: 1206 units.

If we apply these figures to the Amdahl's law equation, when the number of processes approaches infinity ($p \to \infty$), we get a maximum achievable speedup of **2.27**. The following graph depicts how the speedup scales corresponding to the number of processing nodes.



It should be noted that Amdahl's Law estimation is usually inaccurate. The reason why is that the timing measurements heavily depend on the input, since the problem is data dependent. Moreover, one can execute the application on several different setups (different network infrastructure, different machine specifications etc.). Hence, the measurements cannot be always accurately predicted by the above model. Either way, the purpose of Amdahl's law is only to provide a rough estimation. A more accurate prediction is made in the following section, where the use of timing diagrams provide us with a better insight into the parallel performance.

3. Parallelizing Kruskal's Algorithm

Parallelization Strategies

Before diving into the implementation of the modifications that are needed to execute Kruskal's algorithm in parallel, we need to find the best strategy for the task. To do so, we use timing diagrams in which we sketch the critical path of the parallel execution. We can then derive a theoretical model for predicting the speedup.

Three parallelization strategies that build upon each other will now be described. In all cases, it should be noted that the input file is an edge list that is loaded in memory before the execution. Also, we assume that all vertices have a unique integer ID from 1 to $|V|$. This ID is used to split and distribute the edges to the processing nodes, simply by sending the edges of $|V|/p$ vertices to each one. More details will be given in the data distribution section.

• **The first strategy** - and the simplest that one can think of - is to split the input in the root node (ID 0) and distribute it (the edges) to the processing nodes. Then, the processing nodes will calculate their local MSTs and send them to the root node, which in turn will merge them into the global Minimum Spanning Tree. This strategy is the one that was used in the "Amdahl's Law" subsection. If we consider a MPI implementation, the messages sent are firstly the sending of the edges from the root to the processors and secondly the sending of the local MSTs from the processors to the root. The following message passing graph depicts the first strategy.

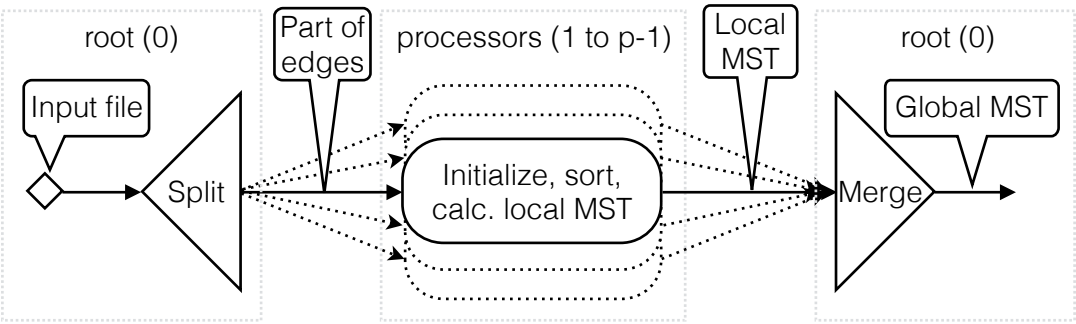


Figure 2: Strategy 1 message passing graph.

• **The second strategy** improves the previous in two ways. First of all, the input file is not split and distributed to the processing nodes. We assume that every processing node now has direct access to the input file. As a result, every node reads and fetches the edges that correspond to its work independently, and from then and on the procedure is the same as in the first strategy. This optimization significantly reduces the communication overhead of the application.

The second improvement is that the root node does not remain idle anymore (waiting to merge results) but instead it participates by computing a local MST as well. However, the local MSTs are still "gathered" and merged in the root node.

It should be noted that the technology used to access the input file is irrelevant, since this abstract strategy can be applied to both shared file systems and distributed file systems scenarios. The practical implementation depends on the network drive provided by EPFL to the students.

The following message passing graph depicts the second strategy. In a MPI implementation, the only communication needed would be the sending of the local MSTs from the processors to the root.

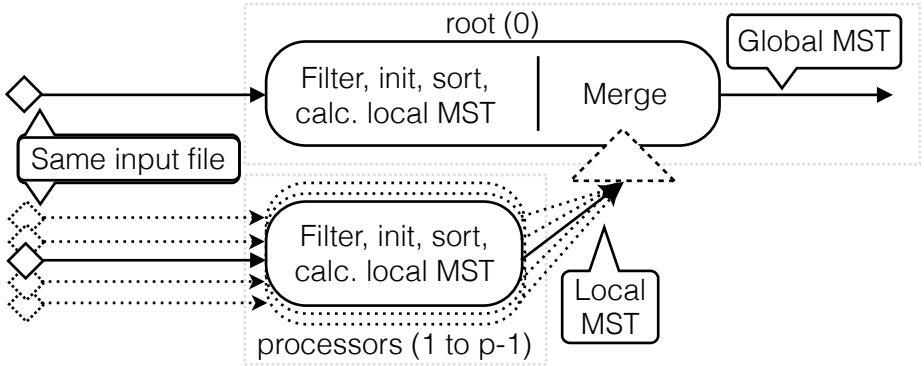


Figure 3: Strategy 2 message passing graph.

- **The third strategy** further improves the previous by merging the local MSTs in a more sophisticated way. In the second strategy, all processing nodes sent their local MSTs to the root node. Then, the partial results are merged in the root node in order to form the global MST, while the processing nodes are not needed anymore and remain idle or terminate.

This strategy suggest that the processing nodes should not remain idle, but instead they should participate in the merging phase. The goal is to complete the merging in *log(p)* steps, where *p* is the number of processors, including the root.

To achieve this in the most optimal way (fairness), we assume that $p = 2^n$, $n > 1$, meaning that the number of nodes must be a power of two. If we do not want to hold the above assumption but, instead, use all the nodes in the system, then we can merge each local MST with the one that the node with the previous index has calculated. This way, the merging requires *log(p)* steps.

The implementation of this project is based on this strategy and assumes that $p = 2^n$, $n > 1$. In a MPI implementation, the communication needed would be the sending of the local MSTs between nodes until, after log(p) steps, the global MST will be available in the root node. The following example explains the merging phase.

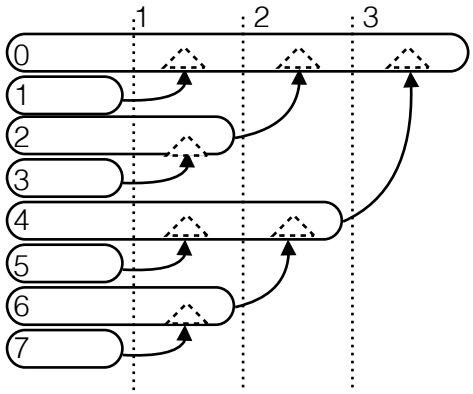


Figure 4: Merging phase example.

In this example $p = 2^3 = 8$ processing nodes are used. The three merging phases are denoted using the vertical dotted lines. In each phase, half of the active processes send their local MST to the previous process, and the other half receives and merges. Finally, the root (ID 0) will hold the global MST. In total, the merging phase is completed in $log_2(8) = 3$ steps.

The following message passing graph depicts the third strategy. The merging of the local MSTs works as described in the previous example. For visual clarity, it is represented by the circle labelled "Merging Phase". The circle overlaps with the processors (including the root) in order to depict that there is communication and cooperation among the participants.

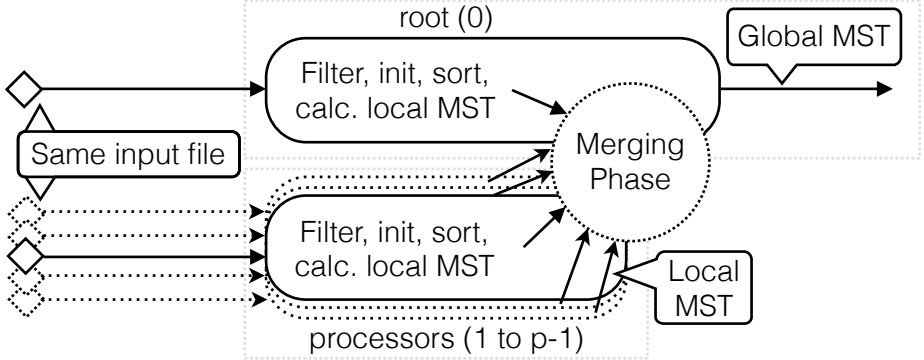


Figure 5: Strategy 3 message passing graph.

It should be noted that the amount of messages sent is equal to those of the second strategy, which means that this optimization improves the performance of the merging phase since the merging workload is now shared between nodes. Once again, the global MST is finally formed in the root node. The third strategy is used for the practical implementation using MPI, and OpenMP is used to speed up local computations by exploiting multithreading.

Input Parsing & Data Partitioning

We assume that the input file contains is an edge list and is in binary form for compression reasons. In its ASCII version, the input file is structured in the following manner:

- The first line of the file contains the number of vertices and the number of edges. The number of edges is in essence the number of the lines that follow.
- Every following line represents an edge of the graph. An edge consists of three elements: The two vertices that are connected by this edge and its weight. Hence, each line contains three integers separated by space for example: "5 3 21" (u v weight). Moreover, we assume that there is no duplication, meaning that if the previous example exists in the file, then the line "3 5 21" will not exist.

Concerning the data distribution to the processing nodes for the preferred strategy (the third), there are two approaches.

The first, naive approach is to distribute $|E|/p$ edges to the nodes, simply by splitting the input file as it is provided. However, this approach yields no performance improvement, since the computation of the local MSTs will eliminate a very small amount of edges. This will happen since most edges will belong to disconnected components, which means that no circles will be easily formed, thus most edges will not be eliminated.

The second and correct approach to the edge distribution problem is based on the intuition that each subproblem must, in the best case, consist of a single disconnected component (a connected subgraph). To achieve this, each processing node processes the interconnecting edges of $|V|/p$ vertices. Stated differently, each node processes a connected subgraph that consists of $|V|/p$ vertices and their edges.

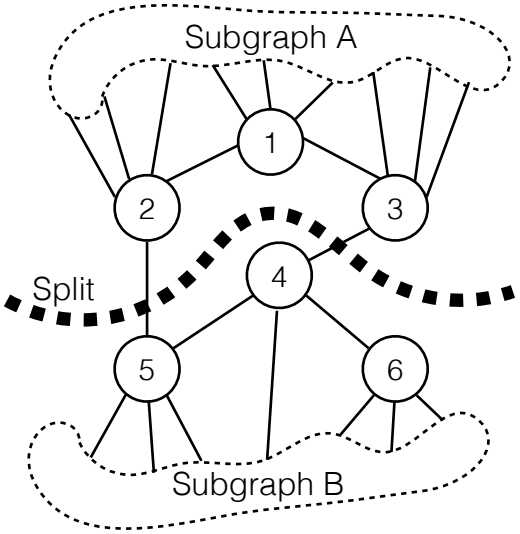


Figure 6: Input splitting example.

For example, let us split the graph on the left and distribute it to two processing nodes. Subgraph *A* and *B* are of equal size, so each node processes one of each, plus the depicted nodes 1, 2, 3 and 4, 5, 6 that belong to the above subgraphs respectively. The overlapping border edges are included in the computation of both processing tasks as they are candidate edges of the global MST. These overlapping edges are the cause of difficulty to be solved when merging two local MSTs.

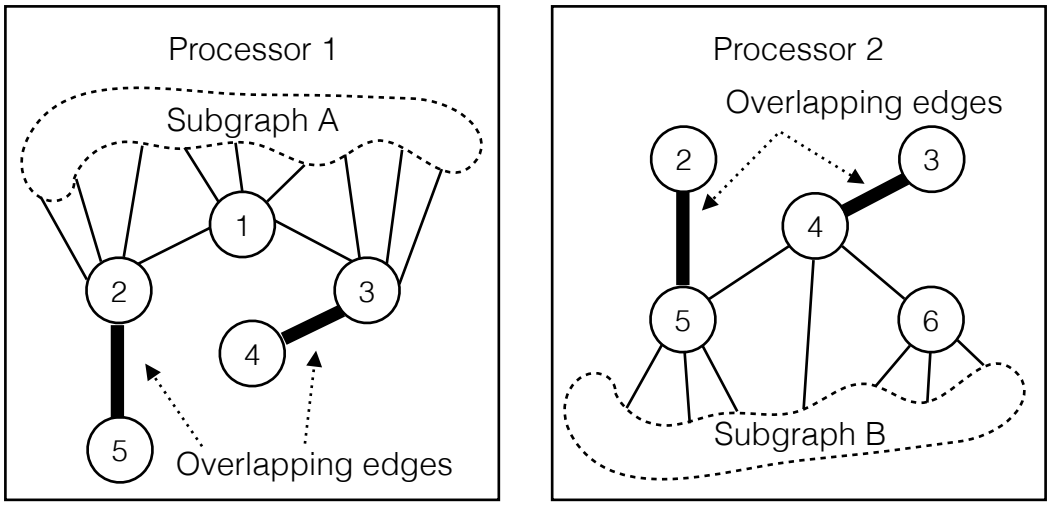


Figure 7: Input splitting example per processor.

Computation to Communication Ratio

We will now calculate the computation to communication ratio of the parallel application for p processing nodes using the first strategy. Let the input graph be $G = (V, E)$.

- **Input: $|E|/p$ —** Each node receives $|V|/p$ edges. This means that every node receives a connected subgraph with, on average, $|E|/p$ edges.
- **Output: $(|V| - 1)/p$ —** Each node outputs a local MST which consists of $(|V| - 1)/p$ edges, since the rest edges do not belong to the local MST and are discarded.
- **Processing: $(|E|/p) \cdot log(|V|/p)$ —** Each node calculates a local MST in $(|E|/p) \cdot log(|V|/p)$ time.

In conclusion, the computation to communication ratio is:

$$\frac{\frac{|E|}{p} log \frac{|V|}{p}}{\frac{|E|}{p} + \frac{|V| - 1}{p}} = \frac{|E| log \frac{|V|}{p}}{|E| + |V| - 1}$$

Since the performance of the algorithm is heavily data dependent, we notice that for a very dense input graph, the numerator (computation) is greater than the denominator (communication). Hence, the denser the input graph (meaning larger problem size), the smaller the impact of communication. The division by the processing nodes hints that we should not use an extremely large number of processing nodes, since this will cause the communication time to exceed computation time.

It should be noted that the same analysis applies for the second and the third strategies, but the $|E|/p$ factor of the communication is now abstracted by a distributed file system mechanism such as a network drive. This means that the communication ratio will eventually be smaller compared to the first strategy.

Timing Diagrams

In this part of the analysis we will design the timing diagrams of all three strategies. Since the execution is heavily data dependent, the timing diagrams are based on the scenario that follows. Moreover, we assume that the merging of the local MSTs is included in the edge retrieval time in the timing diagrams.

- The measurements were made using a 22.9MB input file containing 10'000 vertices and 1'000'000 edges. The graph that they form is random, connected, undirected and weighted.
- The real world throughput of a gigabit ethernet connection is 66MB/s and the latency is 0.3ms (as professor R.D. Hersch noted). This means that the 22.9MB dataset can be sent in about $330 + p \cdot 0.3$ milliseconds, where p is the number of processing nodes.
- Moreover, since the final MST will contain $|V| - 1 = 9999$ edges (0.229MB) that will be transferred over the network, we add to the previous measurement another $3.3 + p \cdot 0.3$ milliseconds. In total, the transfer time will be approximately $333.3 + 2 \cdot p \cdot 0.3$ ms.
- By executing the serial version of the algorithm on a commodity machine, we measure that the execution time is 686ms. We notice that the processing time is greater (>), but not much greater (>>) than the communication overhead and this can be explained by the fact that the algorithm runs in $O(E \cdot log(V))$ and not in quadratic time. As a result, it is difficult to achieve significant speedup with this application.

Let us name the total graph distribution time t_d, the total processing time t_p, the total time required to send back the partial results t_b and the latency for one transmission t_l. We have measured that $t_p > t_d > t_b > t_l$. Also, $t_{serial} = t_p$.

- **The first strategy:**

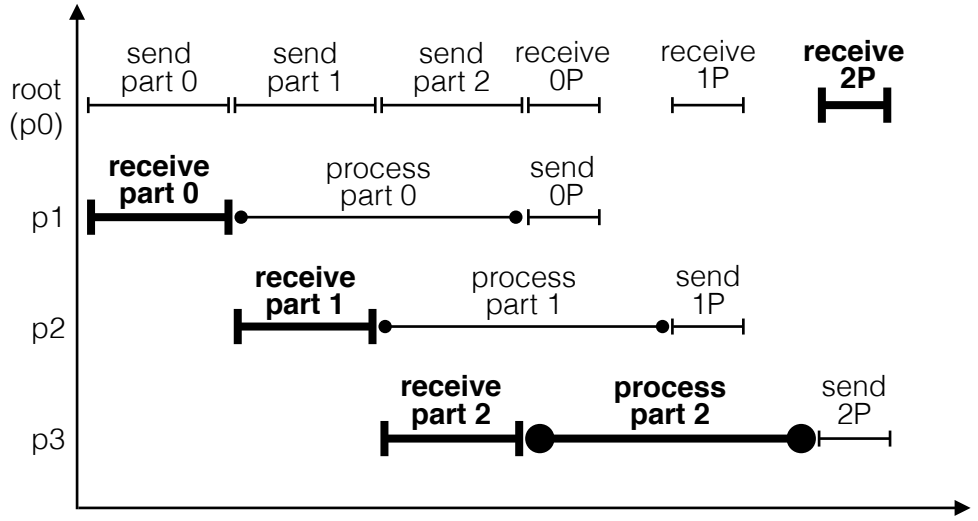


Figure 8: Strategy 1 timing diagram and critical path.

The previous timing diagram represents the first strategy. The critical path is depicted in bold and it consists of the distribution of all parts of the graph, the processing of the last part and the reception of the last part by the root process. This description can otherwise be stated by the following equation:

$$t_{parallel} = 3t_l + 3\frac{t_d}{3} + \frac{t_p}{3} + t_l + \frac{t_b}{3} = \frac{t_p}{3} + 4t_l + t_d + \frac{t_b}{3}$$

Having calculate the parallel execution time, we can estimate the speedup using the following equation:

$$speedup = \frac{t_{serial}}{t_{parallel}} = \frac{t_p}{\frac{t_p}{3} + 4t_l + t_d + \frac{t_b}{3}} = \frac{3}{1 + \frac{12t_l + 3t_d + t_b}{t_p}}$$

We can now generalize $t_{parallel}$ and speedup for n processes (excluding the root node):

$$t_{parallel} = n\,t_l + n\frac{t_d}{n} + \frac{t_p}{n} + t_l + \frac{t_b}{n} = \frac{t_p}{n} + (n+1)t_l + t_d + \frac{t_b}{n}$$

$$speedup = \frac{t_{serial}}{t_{parallel}} = \frac{t_p}{\frac{t_p}{n} + (n+1)t_l + t_d + \frac{t_b}{n}} = \frac{n}{1 + \frac{n(n+1)t_l + n\,t_d + t_b}{t_p}}$$

- **The second strategy:**

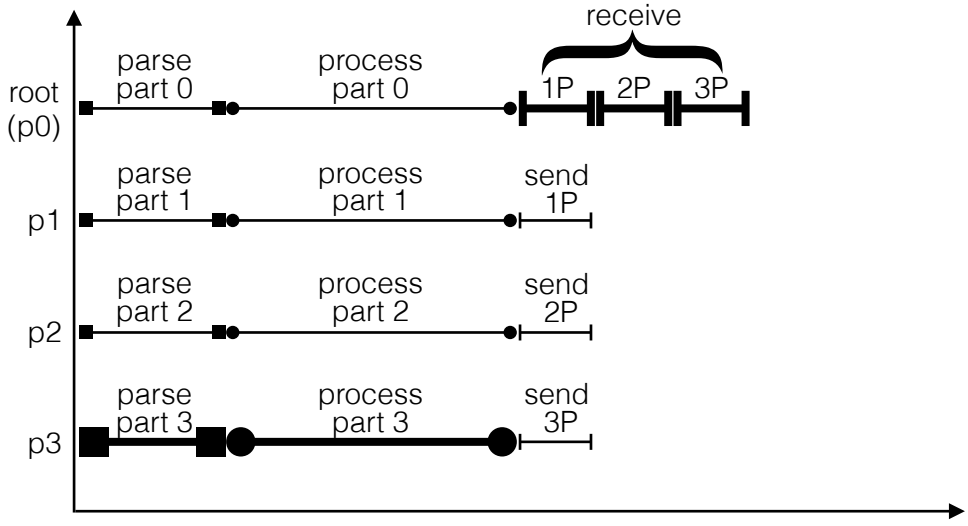


Figure 9: Strategy 2 timing diagram and critical path.

As previously described, the second and third strategy assume that all processes have direct access to the input file through an abstract mechanism (distributed file system). Instead of the graph distribution cost there is a new initial cost for the parsing phase, denoted $t_i < t_d$, where

each processing node filters the input file for the edges that it should process. Moreover, the root node now also participates in the processing, and finally is responsible for merging all local MSTs.

In this strategy, the critical path consists of the parsing and processing of part 3 and the reception of all parts by the root node, except part 0 which is already there. The reception of the parts is executed in a serial manner, since the communication resource of the root node must be released before processing the next incoming part (blocking).

$$t_{parallel} = t_i + \frac{t_p}{4} + t_l + 3\frac{t_b}{4}$$

$$speedup = \frac{t_{serial}}{t_{parallel}} = \frac{t_p}{t_i + \frac{t_p}{4} + t_l + 3\frac{t_b}{4}} = \frac{4}{1 + \frac{4t_i + 4t_l + 3t_b}{t_p}}$$

Generalization for n processing nodes, including the root (p_0):

$$t_{parallel} = t_i + \frac{t_p}{n} + t_l + (n - 1)\frac{t_b}{n}$$

$$speedup = \frac{t_{serial}}{t_{parallel}} = \frac{t_p}{t_i + \frac{t_p}{n} + t_l + (n - 1)\frac{t_b}{n}} = \frac{n}{1 + \frac{n\,t_i + n\,t_l + (n - 1)t_b}{t_p}}$$

- **The third strategy:**

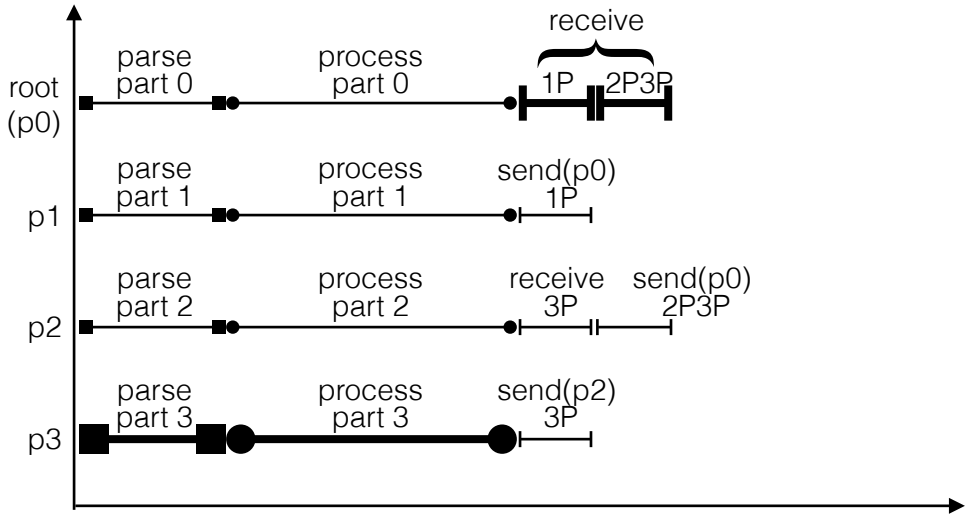


Figure 10: Strategy 3 timing diagram and critical path.

The third strategy is an optimization over the second strategy. Previously, the merging phase worked in a blocking manner, since all parts were sent to the root node. In this strategy, the merging workload is shared between the processing nodes as described previously. What we achieve is a non blocking model that is also more efficient than the previous strategy, since we now require only $log_2(n)$ steps to complete the merging.

The critical path is similar to the one of the second strategy, but the difference lies in the merging phase where the number of merging steps is greatly reduced. Even though the size of the intermediate messages grows due to the merging of local MSTs, the communication impact growth is negligible compared to the performance gains and thus we can assume than the cost of transmitting an intermediate message is still t_b.

$$t_{parallel} = t_i + \frac{t_p}{4} + t_l + 2\frac{t_b}{4}$$

$$speedup = \frac{t_{serial}}{t_{parallel}} = \frac{t_p}{t_i + \frac{t_p}{4} + t_l + 2\frac{t_b}{4}} = \frac{4}{1 + \frac{4t_i + 4t_l + 2t_b}{t_p}}$$

Generalization for n processing nodes, including the root (p_0):

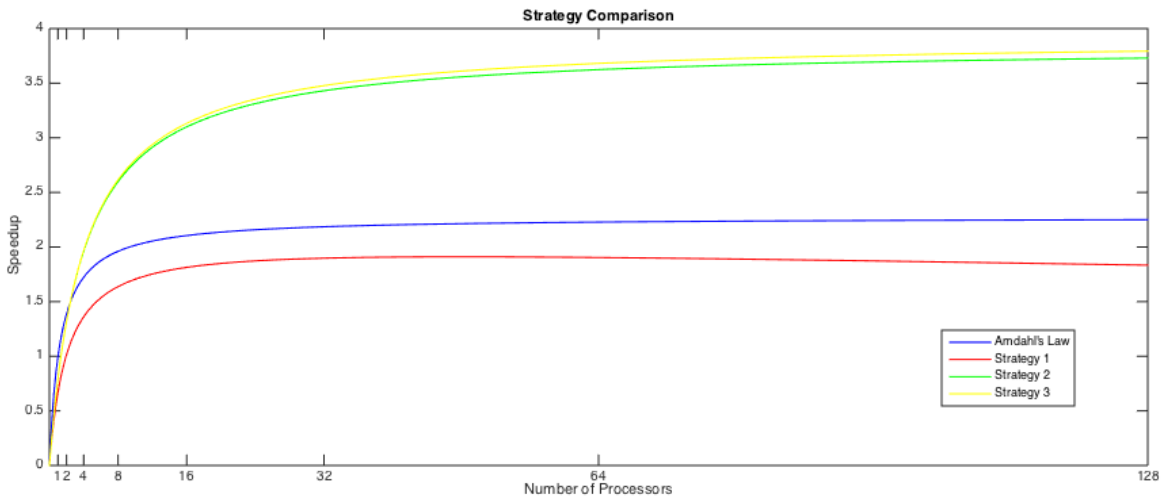
$$t_{parallel} = t_i + \frac{t_p}{n} + t_l + log_2(n)\frac{t_b}{4}$$

$$speedup = \frac{t_{serial}}{t_{parallel}} = \frac{t_p}{t_i + \frac{t_p}{n} + t_l + \log_2(n)\frac{t_b}{n}} = \frac{n}{1 + \frac{n\,t_i + n\,t_l + \log_2(n)t_b}{t_p}}$$

Comparison

In the following graph we can see the comparison between the three strategies and Amdahl's law theoretical prediction.

Amdahl's law speedup estimation was based on the first strategy. The estimation based on the timing diagram of the first strategy is very close to ***Amdahl's law*** prediction with two slight differences. Firstly, the estimation of ***Strategy 1*** shows that by increasing the number of processing nodes above an optimal threshold, the speedup begins to slowly decrease. Also, the peak performance of ***Strategy 1*** and ***Amdahl's law*** differ by about 0.4 in favor of the second.



Strategy 2 and ***Strategy 3*** provide about the same speedup, with ***Strategy 3*** being slightly more efficient due to the merging phase optimization. Clearly, both of these strategies provide a more optimistic estimation compared to ***Amdahl's law***. However, in all cases it is apparent that the achievable speedup is significantly small and bounded due to the nature of the algorithm.

4. Implementation & Evaluation Results

Implementation

Strategy 3 was implemented and tested for both for *Linux* and *Windows* environments, using *Open MPI* and *MPICH2* respectively. For the compilation, *GCC* and *Visual Studio* were used respectively, and the language used is C. Since implementation details are out of the scope of this report, interested readers can refer to the comments within the source code.

The algorithm was implemented with both high performance and minimal memory footprint in mind. Hence, several programming tricks were used to achieve that. At this point, it is important to mention that a parallel quicksort was implemented using *OpenMP* in order to speed up local MST computations. As a result, the parallel implementation of Kruskal's algorithm is a "*MPI - OpenMP*" hybrid. Finally, as Strategy 3 assumes, it is required that the number of processors is a power of two.

Evaluation

The speedup evaluation of the implementation took place in INF3 lab, using 1, 2, 4 and 8 processing nodes. Four datasets of 10'000 vertices and varying graph density were used for the purpose, since Kruskal's algorithm behaves differently based on that characteristic. The speedup results, including comparisons to the theoretical speedup are shown below. The following table summarizes the execution times without taking I/O into account, as instructed by professor R.D. Hersch.

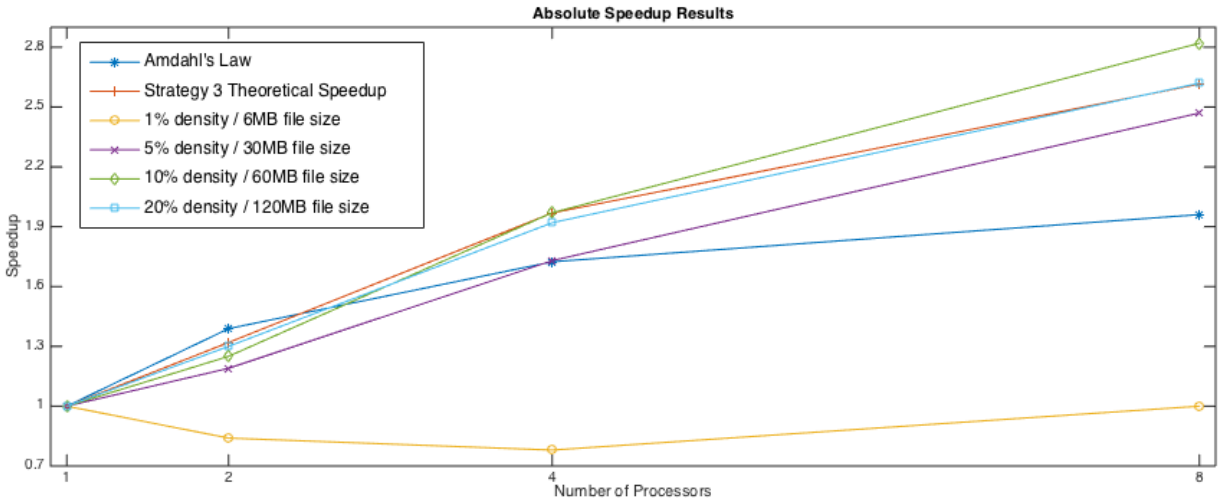
	1% Graph Density ***6MB file size***		5% Graph Density ***30MB file size***		10% Graph Density ***60MB file size***		20% Graph Density ***120MB file size***		
Node Count	**Execution Time (s)**	**Speedup**	**Execution Time (s)**	**Speedup**	**Execution Time (s)**	**Speedup**	**Execution Time (s)**	**Speedup**	**Expected Theoretical Speedup**
1	0.032	1	0.250	1	0.528	1	0.878	1	**1**
2	0.038	0.84	0.210	1.19	0.422	1.25	0.674	1.30	**1.32**
4	0.041	0.78	0.144	1.73	0.268	1.97	0.456	1.92	**1.96**
8	0.032	1	0.101	2.47	0.187	2.82	0.335	2.62	**2.61**

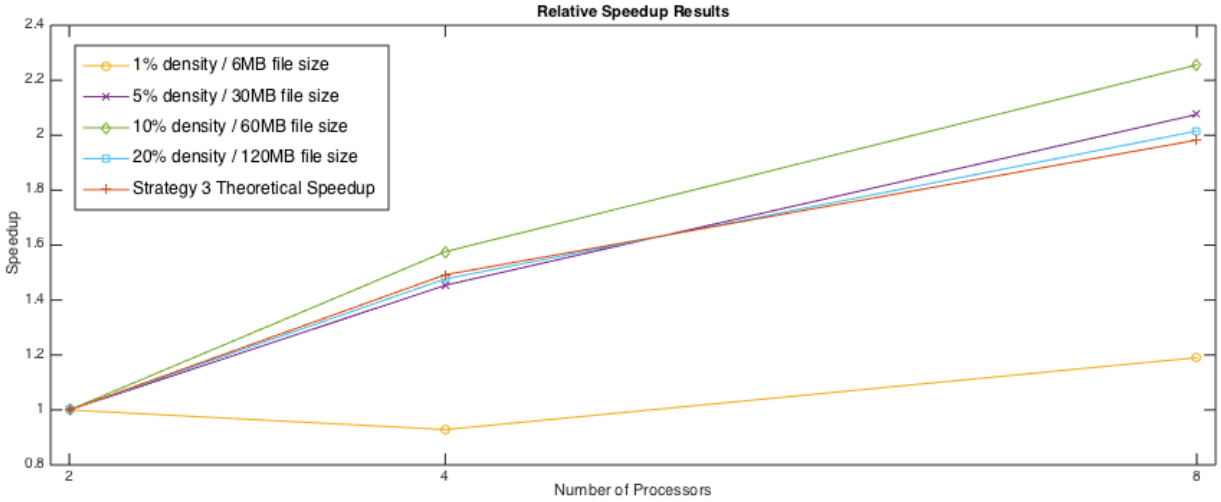
Table 1: Performance evaluation on 1, 2, 4 and 8 nodes.

As we can see, there is a variety of speedup results, based on both the graph density and especially the input file size. We notice that when processing the graph with 1% density, the speedup is significantly low and sometimes below 1. This can be explained due to the fact that the input file is small enough (6MB) to outweigh the potential speedup gain. When we process larger input graphs (30MB, 60MB and 120MB) we notice that we achieve the expected theoretical speedup (last row of *Table 1*). More specifically, the case of the 20% density graph and 120MB input file is extremely close to the expected theoretical speedup, which hints that the theoretical prediction was accurate and successful.

At this point, it should be noted that Kruskal's algorithm works best for sparse graphs. To test this, we ran an experiment with large input graphs of equal file size, but different densities and the result was a faster execution using the less dense graph, as expected. It is apparent that the experiment of *Table 1*, was not designed to demonstrate this characteristic of the algorithm. However, if we compare the speedup of the two last experiments (10% density and 60MB file size versus 20% density and 120MB file size), we notice that the speedup of the sparse graph using 8 nodes (2.82) is greater than the respective one for the denser graph (2.62).

In the following graphs, we can see the absolute and relative speedup of each experiment, as well as the comparison with the theoretical speedup and Amdahl's Law speedup.





5. Conclusion

Trying to parallelize algorithms with "linearithmic" complexity is a challenging task, since in many cases the communication overhead exceeds the parallelization benefits. Parallelizing Kruskal's algorithm belongs to that complexity category, but its parallelization is difficult for several other reasons.

In this theoretical analysis we described how to face issues such as correct data distribution, parallel processing and calculation of local Minimum Spanning Trees, merging of partial results etc. We provided three strategies that build upon each other and finally, based on an execution scenario, we estimated that the maximum achievable speedup using the last strategy and an infinite number of processing nodes is 3.9.

The last strategy was implemented using a hybrid combination of OpenMP and MPI in order to achieve the predicted speedup. The experiments and the evaluation of the implementation have shown that, for a large enough file size (greater than 100MB), the theoretical prediction is extremely accurate. However, the problem is heavily data dependent, since it behaves in a different manner based on the nature of the given input graph. The most important factor of the input graph is its density, which may lead to the worst case execution. Fortunately, in most real life applications, graphs may contain a great number of vertices, but they are sparse, meaning that there are only a few connections from one vertex to another.

References

[1] Borůvka, Otakar. "O jistém problému minimálním." (1926): 36-58.

[2] Prim, Robert Clay. "Shortest connection networks and some generalizations." Bell system technical journal 36.6 (1957): 1389-1401.

[3] Kruskal, Joseph B. "On the shortest spanning subtree of a graph and the traveling salesman problem." Proceedings of the American Mathematical society 7.1 (1956): 48-50.

[4] Gabriel, Edgar, et al. "Open MPI: Goals, concept, and design of a next generation MPI implementation." Recent Advances in Parallel Virtual Machine and Message Passing Interface. Springer Berlin Heidelberg, 2004. 97-104.

[5] Dagum, Leonardo, and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming." Computational Science & Engineering, IEEE 5.1 (1998): 46-55.

[6] Pan, Da-Zhi, et al. "The application of union-find sets in kruskal algorithm." Artificial Intelligence and Computational Intelligence, 2009. AICI'09. International Conference on. Vol. 2. IEEE, 2009.