Karlsruhe Institute of Technology

Massively Parallel List-Ranking and Tree-Rooting

Master Thesis of

Thomas Alexander Weidmann

at the Department of Informatics
Institute of Theoretical Informatics, Algorithm Engineering

Reviewer:	Prof. Dr. Peter Sanders
Advisor:	M.Sc. Tim Niklas Uhl
Second advisor:	M.Sc. Matthias Schimek

01.08.2023 – 01.03.2024

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself. I have submitted neither parts of nor the complete thesis as an examination elsewhere. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. This also applies to figures, sketches, images and similar depictions, as well as sources from the internet.

Karlsruhe, 01.03.2024

. . . .

(Thomas Alexander Weidmann)

Abstract

The problem of list ranking determines for every vertex in a list the distance to the end of the list. The input can be generalized to a forest. This problem is denoted by tree rooting. Here each vertex aims to determine its distance to the tree's root and identify the root itself. Solutions to both of these problems are fundamental building blocks that are often encountered in graph algorithms. Tree rooting can be used for finding connected components which has a broad spectrum of applications. An important application of list ranking is the Euler tour technique.

Since graphs increase in size due to larger amount of data, leveraging parallelism becomes essential to process them within a reasonable timeframe. For list ranking the sparse ruling set algorithm seems to show the best performance in the literature. However, we are not aware of any implementation that works for tree rooting.

In this thesis, we provide an overview of the related work in the fields of list ranking and tree rooting. Our goal is a practical tree rooting algorithm for distributed memory machines scaling up to several thousands of cores. This leads to our main contribution, a sparse ruling set algorithm for tree rooting. We compare two well known strategies of this algorithm where one initially spawns all rulers and the other approach additionally spawns rulers in the course of this algorithm aiming to reduce communication overhead. Furthermore, we evaluate blocking and non-blocking communication.

We use techniques from the literature to reduce communication volume by omitting implicit known information. We aim to lower communication overhead for global message exchange by utilizing a two-level approach instead of direct communication. We use techniques for contracting edges by removing edges beween vertices on the same processing element (PE), which is a common strategy for other distributed graph problems.

We propose a new algorithm for removing high-degree vertices, which otherwise adversely affect the performance of our algorithms. Our practical evaluation indicates that the effectiveness of this method is not dependent on the initial degrees of the vertices.

Our experiments show that the sparse ruling set algorithm for tree rooting is almost twice as fast for rooting a tree than ranking the Euler tour with the sparse ruling set algorithm optimized for lists and up to 10 times faster than using pointer doubling. This algorithm roots all instances including a list and very shallow forests with a good performance for processor counts up to at least 16384 cores, which is far more than the previous practical contributions we are aware of.

Zusammenfassung

List Ranking bestimmt für jeden Knoten in einer Liste die Distanz zum Ende der Liste. Die Eingabe kann dabei von einer Liste zu einem Wald erweitert werden. Dieses Problem heißt Tree Rooting, wobei jeder Knoten die Distanz zur Wurzel des Baumes bestimmt und außerdem den Wurzelknoten identifiziert. Das Wissen der Knoten über ihre Wurzelknoten kann benutzt werden, um zusammenhängende Komponenten zu identifizieren, ein Problem, welches viele Anwendungen in der Informatik hat. List Ranking findet eine sehr häufige Anwendung in der Euler Tour Technik.

Da in heutigen Zeiten immer mehr Daten anfallen und verarbeitet werden müssen, besteht eine wichtige Aufgabe der Forschung darin, parallele Algorithmen zu entwickeln, die mit solchen Datenmengen in kurzer Zeit umgehen können. Für List Ranking ist uns der Sparse Ruling Set Algorithmus bekannt für seine gute parallele Performance. Bei der Literaturrecherche haben wir eine Anwendung dieses Algorithmus für Tree Rooting jedoch nicht finden können.

In dieser Thesis geben wir einen Überblick über den Stand der Forschung in diesem Bereich. Unser wichtigster Beitrag ist der Sparse Ruling Set Algorithmus, angepasst für Tree Rooting, welcher mit verteiltem Speicher arbeitet. Wir haben unseren Algorithmus mit blockierender und nicht blockierender Kommunikation verglichen. Ein weiterer Freiheitsgrad des Algorithmus ist die Art und Weise, wie Ruler ausgewählt werden. Oft wird einfach eine feste Anzahl an Rulern zu Beginn ausgewählt, im Gegensatz dazu kann man aber auch im Laufe des Algorithmus neue Knoten zu Rulern machen, um den Overhead für die Kommunikation zu verringern.

Unser Algorithmus nutzt bekannte Techniken, um Kommunikationsvolumen und außerdem Kommunikationsoverhead durch den Einsatz von Indirektionen bei der Nachrichtenaggregation erheblich reduzieren. Wir benutzen auch Techniken um lokale Kanten zu kontrahieren, sodass nur noch Kanten zwischen Knoten auf verschiedenen Prozessorkernen existieren. Dies ist ein Ansatz, der bei vielen verteilten Graph Algorithmen benutzt wird.

Ein weiterer Beitrag dieser Thesis ist eine neue Methode, um mit sehr hohen Knotengraden umgehen zu können, denn diese haben einen schlechten Einfluss auf die Laufzeit. In Experimenten konnten wir zeigen, dass die Laufzeit dieser Methode unabhängig von den initialen Knotengraden ist.

In unserer Evaluation zeigte sich die gute Performance unseres Algorithmus, auf Listen sowie auf flachen Wäldern, auf mindestens 16384 Prozessorkernen, was deutlich über den in der Literaturrecherche gefundenen Experimenten liegt,

Contents

List of Figures

List of Tables

1 Introduction

Graphs are a commonly utilized abstraction for describing relationships between objects. In today's world, as these graphs increase in size, processing them within a reasonable timeframe requires the exploitation of parallelism. Given the limited memory in shared-memory machines, a significant challenge lies in developing algorithms that operate on distributed memory machines equipped with several thousand cores. On these highly distributed supercomputers, communication frequently becomes the bottleneck, necessitating the discovery of solutions for scalable algorithms.

We explore the concept of tree rooting, where each vertex in a forest aims to identify its root and its depth. This fundamental issue represents a basic algorithm in the toolbox that can be leveraged for a variety of problems. The root pointers of each vertex designate the specific tree they belong to, a technique that can be utilized to find connected components. This has a broad range of applications, including in VLSI design, machine learning, and image analysis [24].

However, when our input is a list, the problem becomes more restricted, namely list ranking. In this scenario, each vertex in a list seeks to determine its distance to the end of the list. List ranking holds significant importance in computer science as it frequently serves as a subroutine, for instance, in the Euler-tour technique. Ranking the Euler tour enables us to determine the depth of all vertices [20] and can be applied to calculating the minimum spanning tree of a graph [13, 3]. Furthermore, list ranking can be extended to compute the prefix sum of any associative operator, underscoring its extensive study in the field [12, 10, 2, 1, 44].

Despite its widespread application, most algorithms break down when applied to tree rooting [41]. An interesting aspect of list ranking is that, sequentially, the optimal algorithm with linear work is straightforward by just following the vertices' pointers. In contrast, devising a practical parallel algorithm is far from trivial, given all the several different approaches. The high degree of irregularity in list ranking necessitates considerable effort to achieve even a speed-up of one [38].
Our Contribution We provide an overview of the related work in the fields of list ranking and tree rooting. We conduct practical comparisons of various implementations of the sparse ruling set algorithm and pointer doubling, with an emphasis on accelerating these algorithms through diverse techniques. One such technique leverages locality by eliminating edges between vertices on the same processing element (PE). Additionally, we use two distinct routing algorithms to facilitate global message exchange, aiming to minimize communication overhead.
Moreover, we introduce an algorithm designed to reduce a forest into instances with

lower vertex degrees, addressing potential negative effects on our runtime. Our principal contribution is the adaptation and generalization of the sparse ruling set algorithm [41] for tree rooting.
Our experimental results reveal that our algorithms scale well up to at least 16384 cores. This performance contrasts with previous studies, which typically focus on systems with several hundred cores.

Structure of this Thesis In Chapter 2, we lay the groundwork by introducing fundamental definitions, as well as providing an overview of the machine model and problem definition. Following this, Chapter 3 provides an overview of the related work, covering various algorithms and offering a historical perspective on the field.
Chapter 4 details our implementations and the strategies we have employed to enhance performance. In the following Chapter 5, we conduct a comprehensive evaluation of our algorithms. This includes demonstrating their performance across a range of instances and sizes, and discussing their scalability.
Finally, in the concluding chapter, we summarize our findings and propose directions for future research.

2 Preliminaries

In this chapter, we introduce the fundamental concepts utilized throughout this thesis, along with additional information regarding the underlying machine model and problem definitions.

2.1 Basic Definitions

Let $G = (V, E)$ be a directed graph where $V = \{0, ..., n-1\}$ is the set of vertices and $E \subseteq V \times V$ is the set of edges. G has n vertices and m edges.
The *ingoing neighbours* of a vertex v are a set of vertices pointing to v with $N_{in}(v) = \{u|(u, v) \in E\}$ and analogue are the *outgoing neighbours* defined as $N_{out}(v) = \{w|(v, w) \in E\}$. We will call the *indegree* of a vertex v as the number of ingoing neighbours $d_{in}(v) = |N_{in}(v)|$ and the *outdegree* of v as $d_{out}(v) = |N_{out}(v)|$. The *neighbours* of v are the ingoing and outgoing neigbours with $N(v) = N_{in}(v) \cup N_{out}(v)$ and the *degree* of v is $d(v) = |N(v)|$.

A *path* is an ordered set of vertices $(v_1, ..., v_k) \in V^k$ where all the vertices are connected with edges i.e. $\forall i \in \{1, ..., k-1\} : (v_i, v_{i+1}) \in E$. The length of this path $|(v_1, ..., v_k)|$ is the number of edges which is $k-1$. If there exists for every pair of vertices $u, v \in V$ a path $(u, .., v)$ then G is called *connected.*

An *in-tree* is a graph where one vertex is called *root* and for every other vertex there is exactly one path to the root. The vertices with zero ingoing neighbours are called *leaves* and the other vertices apart froom the root are called *inner vertices.* An in-tree can consist of only one vertex.

An *in-forest* is a graph that is a disjoint union of in-trees. A list is a in-tree that just has one leaf. In this case we call the root *final vertex* and the leaf *initial vertex.*

2.2 Machine Model and Input Format

We take a look at two different machine models. The *PRAM* (parallel random access machine) model is a shared-memory model. Here we have p processors where each knows its index $i \in \{0, .., p-1\}$. There are submodels that specify concurrent access on the same memory cell. These are exclusive read exclusive write (EREW) PRAM, concurrent read exclusive write (CREW) PRAM and concurrent read concurrent write (CRCW) PRAM. Real shared-memory machines behave similar to the CREW model since they support something that resembles concurrent read [34]. The CRCW PRAM model is the most powerful since it supports also concurrent write. However we need to define what happens when multiple values are written in the same memory cell at the same time. Here the most widely used are "common" and "arbitrary". The common CRCW PRAM model handles concurrent write by only accepting these concurrent operations when they are equal. Different concurrent write on the same memory cell will result in no write operation. The arbitrary CRCW PRAM model handles these concurrent operations by picking an arbitrary operation of those. This model also assumes synchronized time steps where each PE executes exactly one operation in each time step [34].

We also consider the distributed-memory computing model since our goal is a distributed memory algorithm. It consists of p processing elements (PEs) numbered $0, .., p-1$ that makes single-ported communication possible [34]. Each PE knows its own index. Sending a message of length l between two different PEs takes time of $\alpha + l\beta$ where α is the overhead for sending the first machine word and β is the time for sending every further machine word [34]. Accessing a memory cell located on another PE consists of a *request* and a *reply* that are sent through our interconnection network.

Input Format

We assume that our input graph is an in-forest. It will be stored as a *successor array* $succ[v_0], .., succ[v_{n-1}]$ where the successor of a root is itself and the successor of every other vertex is the unique outgoing neighbour. On the distributed-memory computing model we assume $p|n$ and every PE with index i has an successor array for the vertices $v_{i\frac{n}{p}}, .., v_{(i+1)\frac{n}{p}-1}$. We denote the number of vertices per PE as $k = \frac{n}{p}$.

2.2.1 Problem Definitions

We focus on the general problem of tree rooting, where our input is an in-forest. Each vertex v aims to identify the root of the tree it belongs to, as well as the distance to this root. We denote these as $dist[v]$ and $root[v]$.

Additionally, we explore the concept of list ranking. In this scenario, our input is a list also stored in a successor array, where each vertex v seeks to determine its distance to the final vertex. This distance is similarly denoted as $dist[v]$.

3 Related Work

In this chapter, we delve into the list ranking problem, a fundamental challenge in computer science that involves calculating the distance of every vertex in a given list to the final vertex. This problem is widely recognized and has been the subject of extensive research [12, 10, 2, 1, 44]. Moreover, list ranking can be extended to compute the prefix sum for any associative operator, showcasing its versatility.
Additionally, this problem finds relevance in the context of a forest, where each vertex v seeks to identify the root of the tree it belongs to and the distance to this root. This specific application is referred to as tree rooting [41]. In this chapter, we provide an introduction to both list ranking and tree rooting, accompanied by a review of related work, covering various algorithms and offering a historical perspective on the field.

The first parallel list ranking algorithm was presented by Wyllie named pointer doubling in 1979 [44]. He knew that this algorithm can also be applied to trees however in context of list ranking he conjectured that $\Omega(n)$ processors are required to reach a logarithmic running time. In 1985 Kruskal, Rudolph and Snir presented the first deterministic optimal speed up algorithm in $O(n^{\epsilon})$ using $n^{\frac{1}{\epsilon}}$ processors for fixed epsilon [23]. They just assumed EREW PRAM. In 1984 Vishkin proposed optimal speed up randomized algorithms which are today known as independent set removal and ruling set [43]. Two years later, in 1986, Vishkin and Cole proposed a deterministic ruling set algorithm in $O(\log(n))$ on EREW PRAM using $\frac{n\log(n,k)}{\log(n)}$ processors for any fixed k (note that $\log(n, 1) = \log(n)$ and $\log(n, k) = \log(\log(n, k - 1))$ for $k > 1$) thereby showing that Wyllie's conjecture is incorrect [11]. In 1988 then Cole and Vishkin proposed the first deterministic optimal speed up algorithm in $O(\log(n))$ while assuming just EREW PRAM [12].

3.1 Applications

Tree rooting effectively calculates the root $root[v]$ for every vertex v, enabling the determination of connected components within a graph because when two vertices have $root[v_0] = root[v_1]$, it indicates they are part of the same connected tree. Chung and Condon used distributed pointer doubling for contracting connected components [9]. Another application is for calculating a minimum spanning tree (MST) where Arge et al proposed using small rounds of Euler tour construction and list ranking [3]. The Euler tour technique is a general approach and can be used for a variety of problems [20]. List ranking is also used in computational biology, particularly in the analysis of genome sequences [19], where large data sets of short chains need to be contracted using an adjustion of the sparse ruling set algorithm [38], which we explain in the following.

3.2 Euler Tour Technique

The Euler tour of a directed graph is a path on all edges where each edge is included exactly once. By viewing the edges as a list we can give each edge a specific weight according to the problem it should solve, simplifying a complex graph problem to a list problem.

Calculating the Euler Tour The goal is to order all edges of the tree such that it is a Euler tour. Note that in our case, where our input is an in-forest stored in a successor array *succ*, we interpret every edge $(i, succ[i])$ as two edges in both directions. Additionaly we can ignore the self edge for the root. This also means that for our inputs the Euler tour always exists and calculating the Euler tour can be simply done by computing a successor function that maps every edge $e \in E$ to its successor $s(e) \in E$. When having an adjacency array this function is very simple. Let $N(v) = (u_0, .., u_{d-1})$ be an ordered set then $s(u_i, v) = (v, u_{i+1 \bmod d})$ is a correct Euler tour [20].

Applications In cases where our input is an undirected tree, we can transform each undirected edge into a directed edge pointing towards the root, thereby converting it into an in-tree [20]. To accomplish this, we first convert every undirected edge into two directed edges, one in each direction, to facilitate the construction of the Euler tour. The euler tour ends at one edge pointing to the root by $s(u, r) = (u, r)$ and each edge is given a weight of one. Then for each edge (u, v) set $p(v) := u$ when the rank of (u, v) is smaller than the rank of (v, u).

Another application is determining a postorder numbering for an in-tree [20]. Here we also add reversing edges and remove self edges of the root. The edge weights are $w(v, succ[v]) = 1$ and $w(succ[v], v) = 0$. Then $\text{post}(v)$ is equal to the prefix sum of $(v, succ[v])$ and the post order numbering of the root is just n.

We can also determine for all vertices the distances to their root which we denoted by $dist[v]$. Here the edge weights are set to $w(v, succ[v]) = -1$ and $w(succ[v], v) = 1$ [20]. In Figure 3.1 these edge weights are visualized.

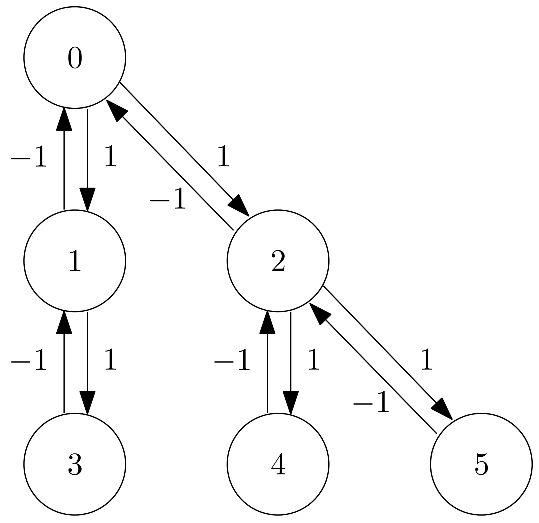


Figure 3.1: Edge weights on Euler tour for tree rooting

3.3 Communication Primitives

Collective operations like broadcast, allreduce or scan are important for synchronization. In a *broadcast* one PE wants to send a message of length l to all other PEs. Broadcast can be implemented with a time complexitiy of $O(\alpha\log(p)+l\beta)$ using pipelining on a binary tree [34]. *Allreduce* computes for any associative operator $\bigotimes$ given a message m_i of length l for every PE i the term $\bigotimes\limits_{j<p} m_j$. The *scan* operation just computes the prefix sum $\bigotimes\limits_{j<=i} m_j$ and the exclusive scan operation $\bigotimes\limits_{j<i} m_j$. Allreduce and both scan operations can be implemented using the hypercube algorithm in $O(\log(p)(\alpha+l\beta))$ [34]. However we often need a more general operation like alltoall where each PE has p personalized messages for each other PE where $l_{i,j}$ is the message from PE i to PE j. We will first consider regular alltoall which we will call *alltoall* where all p^2 messages have the same length l. This operation can be implemented in $O(\alpha p+L\beta)$ with $L=pl$ being the length of all messages that every PE sends or receives. However for small messages the startup overheads dominate the running time. We can add one indirection by viewing the PEs as two dimensional grid and we first exchange messages horizontally and then vertically. Therefore we achieve $O(\sqrt{p}\alpha+L\beta)$. We can generalize this scheme by adding indirections with a computational complexity $O(d(p^{\frac{1}{d}}\alpha+L\beta))$. We can add up to $d=\log_2(p)$ (assuming $p=2^d$) indirections which equals the hypercube algorithm. This algorithm has a computational complexity of $O((\alpha+L\beta)\log(p))$ [22]. In practice an intermediate solution turns out to be optimal on highly distributed memory machines [5].
However in practice we need irregular alltoall which we denote by *alltoallv* where all p^2 messages have arbitrary lengths. With zero indirections we have the same formula $O(\alpha p+\hat{L}\beta)$ but $\hat{L}$ is now the bottleneck volume which is the maximum amount of data sent or received by any PE. By adding one indirection we achieve $O(\sqrt{p}\alpha+\hat{L}\beta)$ which for practical purposes can significantly improve performance for massively parallel algorithms [32]. Note that that the bottleneck volume $\hat{L}$ can differ by adding one indirection even if the messages are all the same, which we further explain in Section 4.1.1.

3.4 Parallel Algorithms

In this section we will cover various algorithms from the literature and offering a historical perspective on the field.

3.4.1 Pointer doubling

Pointer doubling or also called Wyllie's algorithm [44] is a simple algorithm with the basic idea of updating the successor of a vertex according to the successor of its successor.

Algorithm 1: Pointer Doubling

```
   Input: Successor Array $succ \in \mathbb{N}^n$
   Output: Dist Array $dist \in \mathbb{N}^n$
1  $passive \leftarrow \{false, .., false\}$ // $passive[i] = true$ iff $i$ is root
2  $root \leftarrow succ$
3  $dist \leftarrow \{1, .., 1\}$ // $dist[i] = 0$ iff $i$ is root
   // Start pointer doubling
4  for $0 <= iteration < \lceil \log_2(n-1) \rceil$ do
5      for $0 <= i < n$ do
6          if $\neg passive[i]$ then
7              $root[i] \leftarrow root[root[i]]$
8              $dist[i] = dist[i] + dist[root[i]]$
9              $passive[i] = passive[root[i]]$
10 return $dist$
```

The pseudocode of this is Algorithm 1. In the first 3 lines we initialize our variables. The final vertex has a *root* pointer to itself with distance 0 and is passive. Every other vertex has a pointer to its successor with a distance 1 and is active. Afterwards we start pointer doubling where in each iteration in Line 4 we double the distance that our *root* pointer covers. Since a list has a maximum distance of $n - 1$ we need to perform at least $\log_2(n-1)$ iterations. In Figure 3.2 there is a visualization of this algorithm where we have $n = 8$ and we therefore need 3 iterations. The edges visualize the *root* pointers and at the top we can see how the *root* pointers are initialized. Below we see the three iterations chronologically and in the end every vertex has a *root* pointer that actually points to the root.
This algorithm consists of $O(\log(n))$ iterations. Each iteration costs $O(\frac{n}{p})$ resulting in a total runtime of $O(\frac{n}{p}\log(n))$ on EREW PRAM for lists since the *passive* values ensure that there is no contention on any memory cell [38].

3.4.1.1 Variants and Optimizations

This algorithm consists of $\lceil \log_2(n-1) \rceil$ iterations in which every vertex v updates its *root*, *dist* and *passive* value according to the values from $root[v]$. Since we run our algorithms on distributed machines we do not have shared memory and the *root* of a vertex is not neccesarily on the same PE. Thus every read consists of a request and a reply where packets are sent through our interconnection network. In practice there is room for some improvements and therefore we will now discuss different optimizations found in the literature.

Reducing Communication Volume

Sibeyn et al. proposed reducing the packet size and therefore reducing the communication volume by omitting implicitly known information [38]. In one iteration of the pointer doubling algorithm we want to update the values of every vertex v with the information

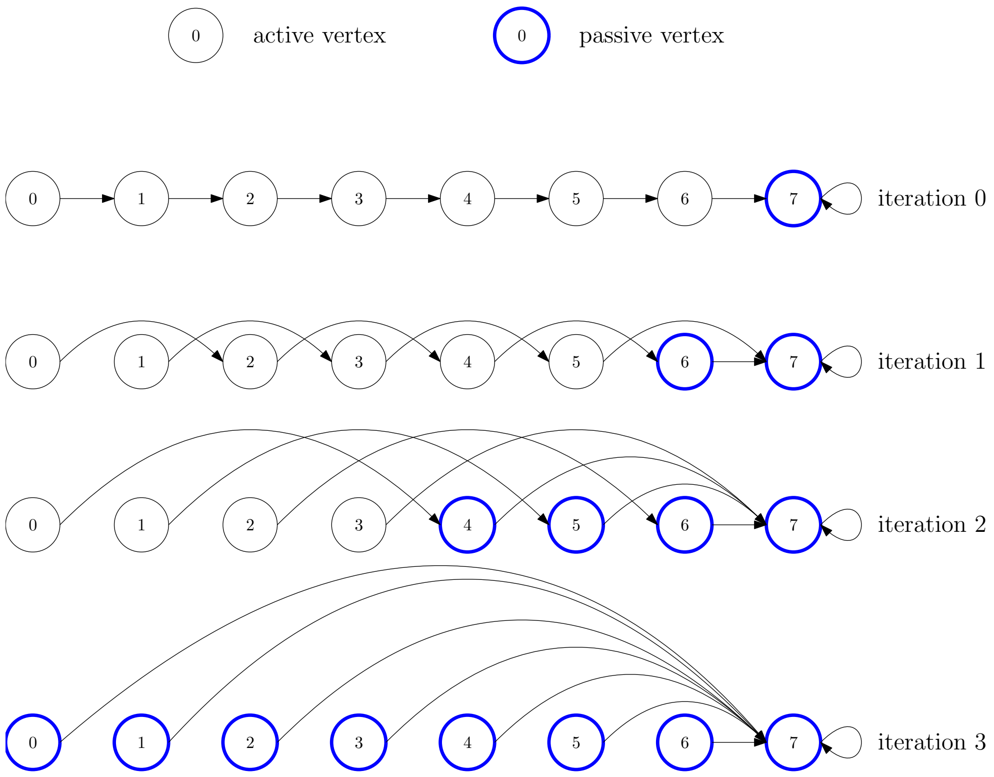


Figure 3.2: Pointer doubling visualization

of $root[v]$. Any PE would write a vector of request packages and route them by a alltoallv. These will be answered and routed back by another alltoallv. Trivially one would write in each packet from which vertex the request is and then the request itself. However one could also just route back all the packages. In this case there is no need to add information about the source vertex.
We can further reduce the communication volume because we do not need to transfer the *passive* status, *dist* and *root* information in our reply. We can omit the *dist* field. Since if $root[v]$ is an active vertex the *dist* just doubles. Only when $root[v]$ is an passive vertex this field has to be requested. When a vertex becomes passive we request in one last extra round $dist[root[v]]$. In total every active vertex v has the request $root[v]$ which is one integer and our reply is $root[root[i]]$ which is also one integer (if we can take one bit for our *passive* status) and we need $\lceil \log_2(n-1) \rceil + 1$ iterations.

Pointer Tripling

We can reduce the number of iterations by tripling the pointers in every iteration leading to only $\lceil \log_3(n-1) \rceil$ iterations [38]. The idea is instead of answering the requests they can be forwarded one more time and then answered. We will therefore need $3 \cdot \lceil \log_3(n) \rceil \approx 1,89 \cdot \log_2(n)$ alltoallv which is less than pointer doubling as explained needing $2 \cdot \lceil \log_2(n) \rceil$ alltoallv. However we cannot ommit the requesting vertex and therefore double the communication volume for reducing the startup overheads to $\frac{1,89}{2} \approx 0.945$ in comparison to the optimization reduce communication volume. More than tripling will never be worth [38].

Double Pointer Jumping

We can further reduce the number of iterations by omitting every request. The idea is to first calculate the predecessor array $pred$. In the first iteration vertex v sends its predecessor $pred[v]$ to its sucessor $succ[v]$ and vice versa. The only restriction is that initial and final vertex do not send and do not receive. Therefore we have just $\lceil \log_2(n-1) \rceil + 1$ global message exchanges. Thus we reduce our commuication overhead to half for doubling the communication in comparison to the optimization reduce communication volume [38].

3.4.1.2 Conclusion

Note that all three optimizations exclude each other. The optimization reduce communication volume has the lowest communication volume while double pointer jumping has the lowest communication overhead. Pointer tripling reduces communication overhead about 5% while doubling the communication volume. Sibeyn et al. implemented reducing communication volume and double pointer jumping. Their conclusion is reducing communication volume has the best performane while double pointer jumping is just good for very small instances where then the seqential algorithm dominated. He prospected that only for very large parallel computers this may help [38].

Even though this algorithm has more than linear work many parallel list ranking algorithms follow the following scheme[20].

- Step 1: Shrink the list to size $O(\frac{n}{\log n})$
- Step 2: Apply pointer doubling to shrinked list
- Step 3: Restore original List

The reason for this is that the work of pointer doubling on this reduced sublist is

$$O(\frac{n}{\log n} \log(\frac{n}{\log n})) = O(n)$$

so we have to just achieve linear work in step one and three to reach linear work in total.

Tree Rooting for Pointer Doubling

Wyllie proposed this algorithm also for forests [44]. The pseudocode in Algorithm 1 works also for tree rooting. However note that not all optimizations do work directly on forests. We implemented double pointer jumping and reduce communication volume for list ranking but as proposed, they do not work for tree rooting. Reduce communication volume returns for the last request only the $dist$ field and therefore has a wrong $root$ pointer [38]. And while calculating the predecessor array for double pointer jumping a vertex might have an arbitrary amount of predecessors which makes this algorithm more

complicated.
On forests we only need to perform $O(\log(h))$ pointer jumping iterations where h denotes the depth of the forest. Here we achieve a work of $O(n\log(h))$ and a runtime $O(\log(h))$ on CREW PRAM [20]. However if we just assume EREW PRAM and $h \leq c\sqrt{n}$ for some constant c this leads to a worst case runtime of $\Omega(h)$ [29]. There exist optimizations to asymptotically improve its performance, however, we observe these approaches more to be categorized as ruling set or independent set removal algorithms, which we explain further in the following sections.

3.4.2 Ruling Set

The first occurance in the literature we are aware of is by Vishkin [43]. The basic idea behind this algorithm is that a subset of size r of the vertices are selected as *rulers*. Then these initiate a wave by sending packets along the edges. These packets are forwarded until they reach a ruler or a final vertex. In Figure 3.3 we see that the algorithm picks 4 rulers. These split our input into sublists. The waves contain information about the initiating ruler such that every vertex has a pointer to the preceeding ruler. When there are no active waves the set of rulers form a sublist that can be ranked independently. Note that the sublist of rulers is directed in reversed order.

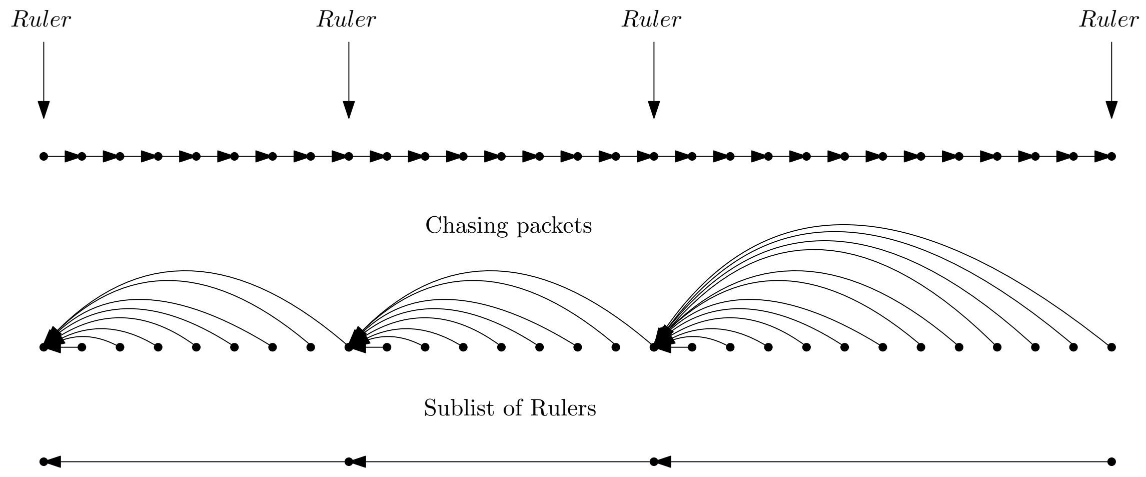


Figure 3.3: Ruling set visualization

The pseudocode of the ruling set algorithm is illustrated in Algorithm 2. First the variables are initialized and we pick rulers by blackbox function choose_rulers$(succ, r)$ which returns a subset of the vertices of size r. Then all rulers start sending packages. These will be forwarded until they reach a ruler or a final vertex. Each vertex v then has a $root[v]$ pointer to the preceeding ruler with its edge weight $dist[v]$. Note the edges in Figure 3.3 represent the $root$ pointers. The sublist of rulers form a reversed weighted sublist of the initial successor array which will be ranked by some arbitrary ranking algorithm. After this we have to calculate the $dist$. First we restore the $dist$ for every ruler in Line 15. When a ruler has rank $rank$ in this sublist then the ruler has rank $n-1-rank$ in our initial successor array because on a list for every vertex the distance to the final vertex plus the distance to the initial vertex is always $n-1$. Afterwards we can rank all the other vertices like in Line 17.

Algorithm 2: Ruling Set

```
   Input: Successor Array $succ \in \mathbb{N}^n$, Number of Rulers $r \in \mathbb{N}$
   Output: Dist Array $dist \in \mathbb{N}^n$
1  $R \leftarrow$ choose_rulers($succ, r$)
2  $root \leftarrow \{0, .., n-1\}$
3  $dist \leftarrow \{0, .., 0\}$
   // Send Packets
4  for $ruler \in R$ do
5      $packet \leftarrow (succ[ruler], ruler, 1)$ // (destination,ruler_source,distance)
6      send($packet$)
   // Chase Packets
7  while $recv(packet)$ do
       // packet = (destination,ruler_source,distance)
8      $root[destination] = ruler\_source$
9      $dist[destination] = distance$
10     if $destination$ is neither final nor ruler then
11         $new\_packet \leftarrow (succ[destination], ruler\_source, distance + 1)$
12         send($new\_packet$)
   // Rank Sublist of rulers
13 $ranks \leftarrow$ algorithm()
   // $(i, rank) \in ranks$ iff vertex $i$ has rank $rank$ in weighted reversed
      sublist
   // Calculate Final Ranks
14 for $[i, rank] \in ranks$ do
15     $dist[i] = n - 1 - rank$
16 for $0 <= i < n \wedge i \notin R$ do
17     $dist[i] = dist[root[i]] - dist[i]$
18 return $dist$
```

3.4.2.1 Variants and Optimizations

The first variant of this algorithm was presented by Vishkin in 1984 [43]. For $p = \frac{n \log(\log(n))}{\log(n)}$ Vishkin chooses p random rulers. Then $\log(n)$ iterations of packet chasing are performed and after that, the list could be contracted to a size $2p+\log(n)$ with probability $1 - \frac{1}{\log(\log(n))}$ in $O(\log(n))$ on CRCW PRAM. The sublist of rulers is ranked with pointer doubling in $O(\log(n))$.
Two years later, in 1986, Vishkin and Cole [11] proposed a deterministic ruling set algorithm in $O(\log(n))$ on EREW PRAM using $\frac{n \log(n,k)}{\log(n)}$ processors for any fixed k (note that $\log(n, 1) = \log(n)$ and $\log(n, k) = \log(\log(n, k-1))$ for $k > 1$) thereby showing that

Wyllie's conjecture is incorrect. He defined the r-ruling set problem, which basically meant finding a subset of V called rulers R where no two rulers are adjacent and for every vertex $v \in V$ there exists a ruler $ruler \in R$ with distance at most r [11]. He then found an iterative approach for finding an $\mathcal{O}(\log(\log(n)))$-ruling set in 1986 [11] that Jaja simplified in his influental book six years later [20]. Each vertex $v \in \{0, .., n-1\}$ is the color $c[v] = v$ assigend. One iteration of his coloring procedure recolored each vertex v to $c'[v] = 2k + c[v]_k$ where k is the least significant bit where $c[v]$ and $c[succ[v]]$ disagree and $c[v]_k$ the kth bit of $c[v]$. Note that if c is a correct coloring then c' is a correct coloring. After two iterations the vertices are properly colored with $\mathcal{O}(\log(\log(n)))$ colors and each vertex whose color is a local minimum becomes a ruler defining a $\mathcal{O}(\log(\log(n)))$-ruling set. The contraced list can be ranked with pointer doubling. We are not aware of any practical evaluations of this algorithm even though it was referenced often [31, 2, 20, 11].

Since then there are various different variants of this algorithm proposed. The main problem that each of the variants tries to solve differently is that two succeeding rulers can have an arbitrary distance. Dehne and Song prove Lemma 3.4.1 and use this lemma to show for $xk = r = \frac{n}{p}$ that $P[m > c3p\log(n)] \leq \frac{1}{n^c}$ for $c > 2$ and conclude that $3p\log(n) = 3\frac{n}{r}\log(n)$ is a good probabilistic upper bound that in practical experiments was never cossed. We generalize this in Lemma 3.4.2 for any r.

Lemma 3.4.1 *Consider $xk \leq n$ randomly chosen rulers. The maximum distance between two succeeding rulers is denoted as m. If $k \geq \log(x) + 2\log(n)$ then P[$m > c\frac{n}{x}$] $\leq \frac{1}{n^c}$ for $c > 2$ [14]*

Lemma 3.4.2 *Consider $r \leq n$ randomly chosen rulers. Then $P[m > c\frac{3n\log(n)}{r}] \leq \frac{1}{n^c}$ for $c > 2$.*

Proof: Set $x = \frac{r}{\mathrm{W}(n^2 r)}$ and $k = \log(x) + 2\log(n)$. Note that W ist the lambert W function

$$\begin{aligned}
n^2 r &= n^2 r \\
n^2 r &= e^{\mathrm{W}(n^2 r)}\mathrm{W}(n^2 r) \\
n^2 \frac{r}{\mathrm{W}(n^2 r)} &= e^{\mathrm{W}(n^2 r)} \\
\log\left(\frac{r}{\mathrm{W}(n^2 r)}\right) + 2\log(n) &= \mathrm{W}(n^2 r) \\
\frac{r}{\mathrm{W}(n^2 r)}\left(\log\left(\frac{r}{\mathrm{W}(n^2 r)}\right) + 2\log(n)\right) &= r \\
xk &= r
\end{aligned}$$

Then $\frac{1}{n^c} \overset{\text{lemma 3.4.1}}{\geq} P[m \geq c\frac{n\mathrm{W}(n^2 r)}{r}] \geq P[m \geq c\frac{n\log(n^3)}{r}] = P[m \geq c\frac{3n\log(n)}{r}]$ for $c > 2$. $\square$

We prove in Lemma 3.4.3 which uses the basic idea of Sibeyn [38] that for $t = 1$ there are after $\frac{n}{r}\log(r)$ iterations an expected number of one ruler unreached. Note that when all rulers are reached then all vertices are reached. In practice there is always the first ruler unreached, which we ignore for simplicity in this lemma.

Lemma 3.4.3 *After* $d = \frac{\log(\frac{t}{r})}{\log(1-\frac{r}{n})} \approx \frac{n}{r}\log(\frac{r}{t})$ *iterations of chasing packets, there are expected* t *rulers unreached given randomly chosen rulers.*

Proof:
For any given ruler the probability that there are no rulers the d positions before in the list are

$$\left(1-\frac{r}{n}\right)^d \approx e^{-d\cdot\frac{r}{n}} \tag{3.1}$$

We want this probability to be $\frac{t}{r}$ so we have an expected number of t rulers unreached

$$\left(1-\frac{r}{n}\right)^d \overset{!}{=} \frac{t}{r} \tag{3.2}$$

$$d = \frac{\log(\frac{t}{r})}{\log(1-\frac{r}{n})} \tag{3.3}$$

$$d \approx \frac{n}{r}\log\left(\frac{r}{t}\right) \tag{3.4}$$

The Equation 3.4 results by replacing the left term of Equation 3.2 with its approximate value from above. □

We measured for one single random list (we explain our experimental setup and generation of a random list in Chapter 5) this maximum distance with different parameters to validate these formulas in Table 3.1. Here we can see that the bound $3\frac{n}{r}\log(n)$ is never crossed but we can also see that for $t = 1$ our formula $\frac{n}{r}\log(r)$ is a good approximation of the maximum distance.

$\frac{n}{r}$	5	10	15	20	25	30	35	40
maximum distance ($p = 2^8, \frac{n}{p} = 10^5$)	75	151	248	305	415	426	452	581
$3\frac{n}{r}\log(n)$	256	512	768	1023	1279	1535	1791	2047
$\frac{n}{r}\log(r)$	77	148	215	281	346	410	473	535
maximum distance ($p = 2^{14}, \frac{n}{p} = 10^7$)	115	216	340	452	594	710	771	813
$3\frac{n}{r}\log(n)$	387	775	1162	1549	1937	2324	2711	3099
$\frac{n}{r}\log(r)$	121	235	347	457	565	673	779	885

Table 3.1: Maximum distance of rulers

Improve maximum distance of vertices

Dehne and Song try to improve the maximum distance by initially picking $\frac{n}{p}$ random rulers [14]. Then they iteratively make every ruler to a non ruler, that was in a certain distance of

another ruler. They stated that the way they picked rulers, the maximum distance between two successive rulers is $6p\log(3p) + 3p(\frac{2}{3}p)^{2k+1}$ where $k = \arg\min_{i>0} \quad \log^{i+1} \leq (\frac{2}{3}p)^{2i+1} \leq$ $\ln^*(n)$ is a very small number which is at most 2 for $n \leq 10^{10^{100}}$ and $p \geq 4$. However this method also uses $2k\log_2(\frac{2}{3}p)$ iterations of pointer jumping, which can be costly.
Dehne et al. proposed another deterministic way to bound the maximum distance of rulers some years later [13]. There in $O(\log(p))$ communication rounds the maximum distance between two successive rulers was improved to $O(p^2)$ with a ruling set of size $O(\frac{n}{p})$. The idea is assigning each vertex the label i if the vertex is located on PE i. A vertex v is selected as ruler when its label is larger than the label of its predecessor and its successor. In $O(\log(p))$ iterations that consist of a pointer doubling contractions, rulers that would be contracted with another ruler become non rulers. However Dehne et al tested this method but they came to the conclusion that just picking random $\frac{n}{p}$ rulers performs better in practice [8].

Leave some unreached vertices

Sibeyn proposed to run packet chasing for a fixed number of iterations and then leave some unreached vertices. Then the unreached vertices are ranked by any list ranking algorithm and afterwards the set of rulers are independently ranked by any list ranking algorithm [37]. Note that the edges for the unreached vertices are in the opposite direction than the edges for the rulers. So ranking them with the same algorithm requires reversing the edges for one of the two subproblems. Sibeyn said this approach limits the recursion deph to at most two because the number of subproblems grow exponentially [41].
One year later Sibeyn published a paper where he implemented this algorithm. He gave some practical infos on implementation details and most importantly how to choose r with $r = 3p\sqrt{n}$. Then the algorithm is running for $d = \frac{n}{r}\log(\frac{n}{r})$ iterations leaving approximately r vertices unreached on which the same terminal algorithm is applied. Sibeyn concluded that two recursions of this algorithm is the best for most inputs [38].

Spawn new Rulers

Sibeyn's latest work on the ruling set algorithm proposed that if a wave hits a ruler then a new wave starts by spawning a new ruler on the same PE. This leads to $\frac{n}{r} + 1$ communication rounds with high probability [41]. However there are in the end $r \cdot \mathrm{H}(\frac{n}{r}) \approx r\log(\frac{n}{r})$ rulers where $\mathrm{H}(i)$ is the ith harmonic number. Note that this means less communication overhead but a bigger subproblem. To balance this out he concludes that $r = \frac{3p\sqrt{n}}{\log(\frac{\sqrt{n}}{3p})}$ in this new algorithm and that this version is on average twice as fast as leaving some unreached vertices for any choice of r [41].

In Table 3.2 we compared three different versions of this algorithm in terms of communication overhead and reduced problem size. Note that we prove in Lemma 3.4.3 for chasing packets until the end a good approximation is $\frac{n}{n}\log(r)$ and for spawning new rulers we can achieve always $\frac{n}{r}$ global communication round with a small modification that we explain in Section 4.2.

Version	#global communications	Reduced problem size
Chase packets until the end [14]	$\leq 3\frac{n}{r}\log(n)$	r
Leave some unreached vertices [38]	$\frac{n}{r}\log(\frac{n}{r})$	$2r$
Spawn new rulers [41]	$\leq \frac{n}{r}+1$	$r\log(\frac{n}{r})$

Table 3.2: Comparison of different versions of ruling set algorithm

Version	#global communications	reduced problem size
Leave some unreached vertices	$\frac{\sqrt{n}}{3p}\log(\frac{\sqrt{n}}{3p})$	$2 \cdot 3p\sqrt{n}$
Spawn new rulers	$\frac{\sqrt{n}}{3p}\log(\frac{\sqrt{n}}{3p})$	$(1+\frac{\log(\log\frac{\sqrt{n}}{3p}))}{\log(\frac{\sqrt{n}}{3p})}) \cdot 3p\sqrt{n}$

Table 3.3: Comparison of different versions of ruling set algorithm with set r according to Sibeyn [38, 41]

Because of different behaviours of different versions, r has to be set individually. In Table 3.3 for leaving some unreached vertices we set $r = 3p\sqrt{n}$ [38] and for spawning new rulers $\frac{3p\sqrt{n}}{\log(\frac{\sqrt{n}}{3p})}$ [41]. As Sibeyn proposed [41] for the same communication overhead we have about half the reduced problem size. Since we can't compare Sibeyn's latest version to simply chasing packets until the end we will do that on our own in Chapter 4 and show that spawning rulers has superior performance the bigger p gets.

Tree Rooting for Ruling Set

Chung and Codon proposed a supervertex algorithm which is similar to a ruling set algorithm for tree rooting. During this algorithm vertices with probability of $\frac{1}{2}$ become a supervertex, which is similar to being a ruler. They implemented packet chasing by pointer doubling and recursively called the supervertex algorithm on the set of supervertices. However the idea of the ruling set algorithm is specifying a small set of rulers to achieve smaller reduced instances, which is why this algorithm is also called sparse ruling set [41]. Jackson et al. proposes a method for transcriptome assembly where they encounter graphs with many long chains [19]. They use the sparse ruling set algorithm on the set of these chains to contract these chains into one vertex.
However to our best knowledge we are not aware of an implemention of the sparse ruling set algorithm for tree rooting. As shown in Figure 3.3 the algorithm needs a small modification to work on forests. In Line 15 of Algorithm 2 we use the invariant that on a list for every vertex the distance to the final vertex plus the distance to the initial vertex is always $n-1$. On forest there may be several roots and leaves with arbitrary depths. Just reversing directions of the edges of the initial input *succ* or the sublist of rulers will make it work and we will evaluate this algorithm.

3.4.3 Independent Set Removal

The first occurence in the literature that we found of was by Vishkin [43]. The basic idea of this algorithm is to find independent sets I and exclude these vertices so we have a smaller subproblem. An independent set in this context means that for every vertex in I its predecessor and successor are not in I. This is visualized in Figure 3.4. First we determine the independent set I which are the blue vertices. A vertex $v \in I$ is excluded by informing its predecessor about $succ[v]$. When the remaining vertices are ranked, every vertex $v \in I$ just has to ask its successor's rank to calculate the final ranks.

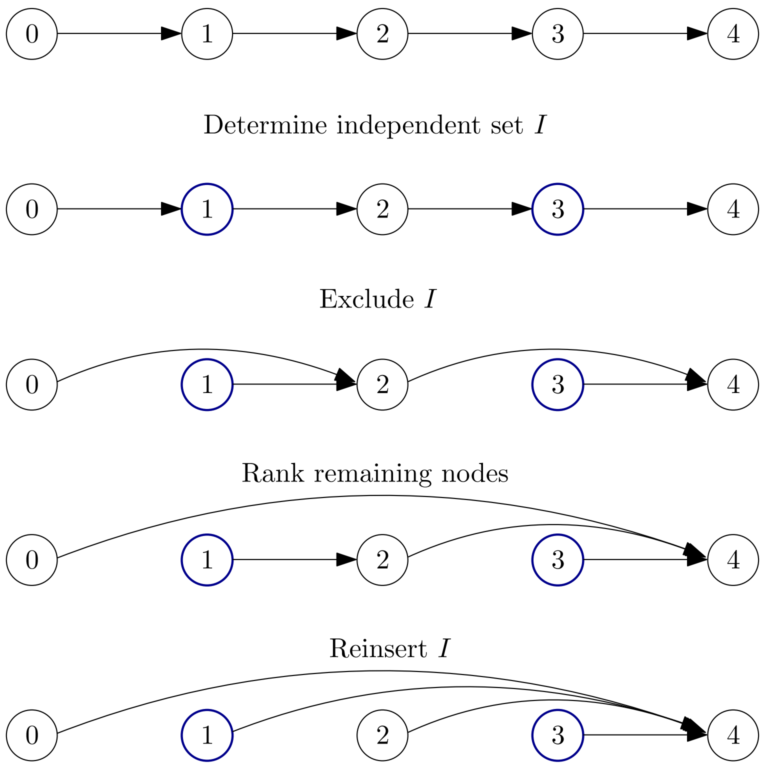


Figure 3.4: Independent set visualization

The pseudocode of this algorithm is shown in Algorithm 3. In every iteration in Line 5 we calculate an independent set I and exclude it. By iteratively reducing the problem size we can finally rank the remaining vertices and then in reverse order insert the excluded independent sets. We exclude a vertex v by letting the predecessor expand its *root* pointer. Reinserting is also straight forward since our excluded vertices have a pointer to a correctly ranked vertex.

3.4.3.1 Variants and Optimizations

The important difference beween different variants is the way I is determined. Note that on lists at most every second vertex can become an independent vertex.

Throwing a Coin

The idea is for every vertex v throwing a binary coin $c[v]$. Then v is in I set when $c[v] = 1$ and $c[pred[v]] = 0$. Its predecessor $pred[v]$ is not in I since $c[pred[v]] = 0$ and also its successor v' because $c[pred[v']] = c[v] = 1$. This can be achieved with one global message exchange where every vertex v where $c[v] = 0$ informs it successor about this. This method achieves independent sets of size $\frac{n}{4}$ [43] which is not optimal however the determination of I is so cheap that it has widespread relevance [6, 31, 41, 38, 39].
This method was proposed in the first independent set algorithm that we are aware of by Vishkin in 1884 [43]. The algorithm iteratively exclude vertices until the number of remaining vertices is less than $\frac{n}{\log(n)}$. Miller devised this algorithm one year later and named it random mating [25]. Here each vertex is assigned a gender male and female. The determination of I stays the same but the algorithm excludes independent sets until there are no more vertices [30, 25]. Newer results on GPUs also use this approach because of its simplicity [6].

Algorithm 3: Independent Set Removal

```
Input: Successor Array $succ \in \mathbb{N}^n$
Output: Dist Array $dist \in \mathbb{N}^n$
1  $root \leftarrow succ$
2  $dist \leftarrow \{1, .., 1\}$ // $dist[i] = 0$ iff $i$ is root
3  $A \leftarrow \{0, .., n-1\}$
   // Reduce problem sufficiently small
4  $phase \leftarrow 0$
5  while $|A| > f(n)$ do
6      $phase \leftarrow phase + 1$
7      $I \leftarrow$ calculate_independent_set()
8      for $i \notin I \wedge succ[i] \in I$ do
9          $root[i] \leftarrow root[root[i]]$
10         $dist[i] \leftarrow dist[i] + dist[root[i]]$
11     $A \leftarrow A \setminus I$
   // Rank all remaining active vertices
12 $root, dist \leftarrow$ some_rank()
   // Reinsert vertices from last phase to first phase
13 while $phase > 0$ do
14     foreach vertex i removed in iteration phase do
15         $root[i] \leftarrow root[root[i]]$
16         $dist[i] \leftarrow dist[i] + dist[root[i]]$
17     $phase \leftarrow phase - 1$
18 return $root$
```

Coloring

When coloring our list with k colors then the set of local minima (or maxima) is an independent set of size $\Omega(\frac{n}{k})$ [20]. Jaja then referenced an optimal 3-coloring algorithm in $O(\log(n))$ time using $O(n)$ operations in $O(\log(n))$ on EREW PRAM [20]. Miller proposed a similar algorithm [31], however we are not aware of any practical evaluations.

Ruler

This iterative method initially assigns every vertex no status. Every vertex with no status becomes with probability q a ruler and informs their predecessor and sucessor that they become a non ruler. This is repeated r times and afterwards every vertex with no status becomes a ruler. All non rulers form an independent set. For $r = 2$ and $q = 0.35$ the indepentent set has expected size $0.449n$ and for $r = 3$ and $q = 0.28$ it has size $0.475n$ [38]. Sibeyn evaluated this method but came to the conclusion that throwing a binary coin is faster in practice [38, 41].

Tree Rooting for Independent Set Removal

Independent set removal can be generalized for forests. However we have to make sure before excluding a vertex that every successor and predecessor is not excluded which makes our independent sets small. Sibeyn stated that this algorithm for tree rooting is inefficient [41].

3.4.4 Cleaning Algorithm

The first occurance that we are aware of was by Sibeyn [39]. The basic idea of this algorithm is that the vertices are divided into two distinct subsets S_0 and S_1. Then we reduce the instance to S_0, rank these vertices and then reconstruct all ranks. This algorithm uses two important subroutines named *autoclean* and *altroclean*. Both subroutines work on a real subset of all vertices and initially $root$ is initalized the same as $succ$ and $dist[v]$ is initialized with 0 for the actual root of a tree and 1 for all the other vertices.
Autoclean Autocleaning S means that all vertices $v \in S$ follow their $root$ pointers until a vertex $w \notin S$ is reached. Then their $root$ field gets updated by $root[v] \leftarrow w$ and $dist[v]$ to the covered distance until w was reached.
Altroclean Altrocleaning S means that all vertices $v \in S$ that have not reached a final vertex and where $root[v] \notin S$ update their $root$ and $dist$ values according to their $root[root[v]]$.
This algorithm is visualized in Figure 3.5. Our input is divided into two distinct sets. Then we call Autoclean(S_1) which ensures there are no edges between two different vertices in S_1. Afterwards we call Altroclean(S_0) such that each vertex $v \in S_0$ points to another vertex in S_0 or to a rootin S_1. Then by first ranking S_0 every vertex $v \in S_1$ just needs one request to compute their final ranks which happens by Altroclean(S_1).

In Algorithm 4 cleaning is explained. After reducing the problem in Line 5 there are no $root$ pointers to vertices in S_1 that are not a root. Then the remaining vertices are ranked by any ranking algorithm and the final values can be calculated.

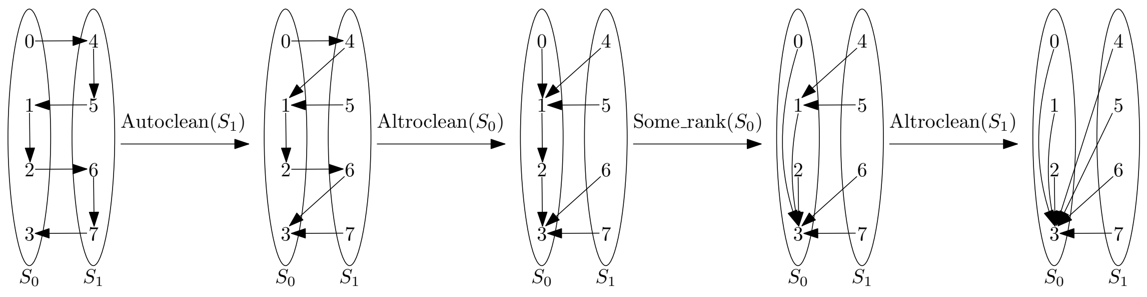


Figure 3.5: Peeling Off Visualization

3.4.4.1 Variants and Optimizations

Sibeyn scheduled the subroutines autoclean and altroclean between sets of vertices on two distinct PEs and called this One-by-One cleaning [39]. The advantage was that there was no global communication but rather direct communication between two PEs used. However the communication volume per PE was $\mathcal{O}(k \log P)$. He optimizes this algorithm by minimizing the global communication volume by a constant factor, but he concludes one-by-one cleaning just performs good on networks on which global communication is strongly penalized [36, 40].

Sibeyn solves this by dividing the PEs into two sets and iteratively halves the number of active PEs until there is just one left that can solve the problem sequentially. The obvious disadvantage is that number of idle PEs doubles each round. He calls this algorithm repeated halving [39].

In a different approach Sibeyn avoids idle PEs by defining S_0 as half of the vertices of every PE and S_1 accordingly. Then he iteratively halves the number of vertices per PE for $\log \log n$ recursions which is then ranked by pointer doubling. He calls this algorithm peeling off [41].

Algorithm 4: Cleaning

```
  Input: Successor Array $succ \in \mathbb{N}^n$
  Output: Dist Array $dist \in \mathbb{N}^n$
1 $root \leftarrow succ$
2 $dist \leftarrow \{1, .., 1\}$ // $dist[i] = 0$ iff $i$ is root
3 $S_0, S_1 \leftarrow$ split() with $S_0 \dot\cup S_1 = V$
  // Reduce problem
4 Autoclean($S_1(t)$)
5 Altoclean($S_0(t)$)
  // Rank all remaining vertices
6 $root, dist \leftarrow$ some_rank($S_0$)
  // Reconstruct all ranks
7 Altroclean($S_1$)
8 return $dist$
```

Tree Rooting for Cleaning

These algorithms works for tree rooting. The only difference is that several vertices may have the same successor. Sibeyn stated for peeling off, that when each vertex is randomly assigned to any PE, then any forest is rooted in $O(\frac{n}{p} + \log(n))$ expected time. When the input is a set of lists then they are rooted in $O(\frac{n}{p} + \log(n))$ w.h.p [41].

3.4.5 Comparison of Algorithms

We want to summarize and compare our findings. After years of intense study of list ranking algorithms Sibeyn stated to have found the ultimate parallel list ranking algorithms where he practically evaluated from each of our presented categories one algorithm and compared them to each other [41]. He implemented independent set removal with determining I by throwing a coin. As a ruling set algorithm he picked the version where rulers are spawned whenever a wave dies and he implemented peeling off as cleaning algorithm. He also theoretically analyzed these algorithms in terms of communication which we now summarize.
Ruling set Chasing packets with spawning new rulers needs $\leq \frac{n}{r}+1$ global communication rounds where in total each vertex sends a packet consisting of 3 integers [41]. The reduced instance size is $r\log(\frac{n}{r})$ [41]. We have to add to his analysis in order to calculate the final ranks, every vertex sends a request to its designated ruler and the ruler answers with its distance to the end of the list. In total we have a communication volume of 5 per vertex by using $\leq \frac{n}{r} + 3$ global communication rounds to reduce our instance of size $r\log(\frac{n}{r})$.
Independent set removal Sibeyn evaluated throwing a coin for determining I. We can efficiently determine I where every vertex v with $c[v] = 1$ sends a packet to its successor. If a vertex v' with $c[v'] = 1$ receives such a packet then it will be excluded and send back $dist[v']$ and $root[v']$ and if $c[v'] = 0$ a placeholder indicating $v' \notin I$. While reinserting every $v \in I$ sends a packet to $root[v]$ which answers with $dist[root[v]]$ and $root[[v]]$. In total we have a communication volume of two integers per vertex for the first phase [41]. Then we have to multiply this with $\sum_{i=0}^{\infty} \frac{3}{4} = 4$ which results in a total communication volume of 8 integers per vertex and 4 global communication rounds per exluding an independent set [41]. We now further analyze this algorithm since we are interested in how many recursive independent set removals are neccessary. This independent set has expected size $\frac{n}{4}$ [38] and we can show that its size is close to the expected value. The size of the independent set changes at most by 2 by changing one single coin throw for v which one can see by brute forcing all 2^3 cases $pred[v], v, succ[v] \in \{0,1\}^3$. We denote with $I(c)$ the size of the independent set for a given coin assignment. The bounded difference inequality [17] gives

$$\mathbb{P}\Big[|I(c) - \mathbb{E}[I(c)]| \geq \sqrt{2n\log(2n)}\Big] \leq 2e^{\frac{-2(\sqrt{2n\log(2n)})^2}{4n}} = \frac{1}{n}$$

thus the size of I will be very close to expectation with high probability. Therefore we can assume that I always has size $\frac{n}{4}$. In every iteration we exclude our independent set of size $\frac{n}{4}$. We need d iterations to reduce our instance to $n(\frac{3}{4})^d \overset{!}{=} \frac{n}{\log(n)}$ which means $d = \frac{\log(\log(n))}{\log(\frac{4}{3})} \approx 3.48\log(\log(n))$. Sibeyn showed every iteration needs 4 global communication rounds [41]

Algorithm	#global communication	routing volume	Reduced problem size
Cleaning	$d(6 + 2\log(\log(n)))$	$6 + \frac{3\log(d)+6\log(p)}{d+1}$	$\frac{n}{2^d}$
Ruling Set	$\frac{n}{r} + 3$	5	$r\log(\frac{n}{r})$
Independent Set Removal	$14\log(\log(n))$	8	$\frac{n}{\log(n)}$

Table 3.4: Comparison of different list ranking algorithms

which makes in total $4 \cdot 3.48\log(\log(n)) \approx 14\log(\log(n))$ global communication rounds with a communication volume per vertex of at most 8 and a reduced problem size $\frac{n}{\log(n)}$.
Peeling off Note that in this version of the cleaning algorithm the sets S_0 and S_1 are distributed over all PEs. Sibeyn implemented autoclean by pointer doubling and altroclean is a request and reply global message exchange. Sibeyn stated that the most important feature to optimize is the number of recursive calls d. In iteration t the problem is reduced by factor $\frac{1+d-t}{2+d-t}$ for a good trade off between communication volume and global communication rounds [41]. Then after d iterations the communication volume is $\leq 6 + \frac{3\log(d)+6\log(p)}{d+1}$ w.h.p per vertex using $\leq d(6 + \log(\log(n))$ w.h.p global communications to reduce our instance to $\leq \frac{n}{2^d}$ [41].
We summarize the analysis of these algorithms in Table 3.4 and compared them in terms of number of global communications, routing volume which we measure by number of integers per vertex and reduced problem size. Note that the comparisons are difficult because all algorithms have parameters that are optimized and some of the values are upper bounds as explained above. Sibeyn practically evaluated all of these algorithms for list ranking and compared them to each other. He came to the conclusion, that ruling set performs best on most instances and on small instances simply pointer doubling is superior. In terms of tree rooting he stated that independent set removal is inefficient and peeling off is a simple and versatile algorithm that can be used for tree rooting. However he did not comment on the possibility of tree rooting for ruling set.

3.5 Tree Contraction

Tree contraction is a systematic way to shrink a tree into a single vertex [20]. Miller and Reif initially motivated this problem for the expression evaluation problem where each leaf is assigned a number and each inner vertex an arithmetic operator [26]. They further found many applications like tree isomorphism, maximal subtree isomorphism of unbounded degrees, computing 3-connected components, planar graph embeddings and list ranking [27]. Since then there has been a massive amount of research for applications like finding planar graph seperators [16], online evaluation of arithmetic circuits [28], the huffman code problem [4] and approximating smallest linear grammar [21]. There exist basically two approaches of parallel tree contraction. A top-down approach starts starts at the root where we have to find a seperator vertex that divides the tree into two subtrees. The main problem is finding seperators where both subtrees have roughly the same size. The bottom-up approach uses the parallel contract subroutine where $O(\log(n))$

contractions reduce the tree into one vertex [31] and newer results show that the bottom up approach is in theory massively parallel [18]. We are aware of an evaluation of list- and tree contraction showing noticable speed-ups using 80 hyper threads [35].

4 Engineering List-Ranking and Tree-Rooting Algorithms for Massively Parallel Scalability

There exists a lot of reseach on list ranking [12, 10, 2, 1, 44] and our goal is to find an versatile algorithm that can also be used for tree rooting. Sibeyn stated that independent set removal breaks down for tree rooting [41]. Peeling off is a simple algorithm that can also be used for tree rooting, however he did not comment on the possibility of the ruling set algorithm for tree rooting, which was the best algorithm for list ranking on most instances [41, 38].
In this chapter we will first explain some of the building blocks that influence our algorithms. Then we explain our ruling set implementation with different optimizations and with the possibility for tree rooting.

4.1 Building Blocks

Massively parallel algorithms are often dominated by communication rather than local computation. Sending a message of length l between two different PEs takes time of $\alpha + l\beta$ where α is the overhead for sending the first machine word and β is the time for sending every further machine word [34]. For highly distributed algorithms the overheads can dominate the running time so we will use indirect communication to reduce overheads which we denoty by *latency*. We can also reduce the communication volume using different techniques where we omit implicitly known data and remove duplicate data. Then we present different techniques to modify our input that would otherwise negatively influence the running time of our algorithms.

4.1.1 Reducing Latency

We often use alltoallv as global communication. However as the number of PEs increases the startup overhead becomes the dominant term. Let $\hat{L}$ be the bottleneck volume, which is the maximum amount of data a PE sends or receives. Then the time complexity is given by the formula 4.5.

$$T_{\text{one-level}}(p) = p\alpha + \hat{L}\beta \tag{4.5}$$

By adding indirections we can reduce latency. Note that this is not a build in MPI routine. We used a similar impementation than by Sanders and Schimek where the PEs are arranged in a virtual two dimensional grid [32]. The grid consists of $c = \lfloor\sqrt{p}\rfloor$ columns and $r = \lceil\frac{p}{c}\rceil$

rows. PE i is assigned colum $\mathrm{col}(i) = i \bmod c$ and row $\mathrm{row}(i) = \lfloor \frac{i}{c} \rfloor$. Global communication is performed in two steps. A message from PE i to PE j is first sent to PE t in row $\mathrm{row}(j)$ and column $\mathrm{col}(i)$ and then to the final destination. Note that during both message exchanges at most $\sqrt{p} + 2$ PEs participate which reduces the startup overheads to $O(\sqrt{p})$ at cost of double the communication volume [32]. This scheme can be generalized for up to $\log(p)$ dimensions which equals the hypercube algorithm [22]. We call this the *two-level* message exchange. The time complexity is given by

$$T_{\text{two-level}}(p) = 2 \cdot (\sqrt{p}\alpha + \hat{L}\beta) \tag{4.6}$$

where $\hat{L}$ is still the maximum amount of data any PE sends or receives. However note that $\hat{L}$ can be worse by factor $O(\sqrt{p})$ for the same p^2 messages when for example just the ith PE in the first collumn wants to send a single message of length one to the ith PE in the first row. Then all $O(\sqrt{p})$ messages will be aggregated in PE in row 1 and column 1. However this usually does not happen in practice. When all p^2 messages have the same length then $\hat{L}$ stays the same.

4.1.2 Reduce Communication Volume

Besides from communication overheads the communication volume dictates the time for a message exchange so we are also interested in reducing the communication volume. During our algorithm often a subset of the vertices in our input graph have requests for other vertices that are aggregated sent with one global message exchange. The requests are then answered and sent back to the requester with one more global message exchange. We call this *request-reply scheme*. Since we can simply route the answers back the same way we do not have to add to our request the source vertex [38]. Note that this is also possible with the two level approach.

During our request-reply scheme we can further reduce the communication volume by *removing duplicate requests*. Before sending our requests, we find duplicates using hash tables and then send only the unique requests. Note that this is also compatible with the omission of the requester just explained, which is always be done. If we use the two-level approach the requests are first sent to an intermediate PE before they reach the final destination. In this case we use the intermediate hop to further remove duplicate requests also using hash tables.

4.1.3 Exploiting Locality

We further reduce the overall instance size by local preprocessing. This is a general scheme that we observed by various parallel graph algorithms [32, 24]. There exist instances with many edges between vertices on the same PE. Note that our input is a successor array so we cannot simply do a breadth-first search. This routine first determines for every vertex v its local root by following its successor pointers. If we reach a already rooted vertex v' while following the successor pointers we can stop and use the local root of v' as local root for v such that we achieve linear work. Then the set of local roots is our modified instance. Note that we need one request-reply scheme for the local roots since they may point now to a removed vertex and additionally we need to update the correct weights. This is visualized in Figure 4.1. We can see on the left there are 5 vertices on the PE that all belong to the same local tree and in total we see 7 vertices distributed over several PEs of this tree. On the edges we can see the updated edge weight so that the distances from our mainpulated instance are actually the correct distances and we can restore the *root* and *dist* values for all vertices from our correcly rooted modified instance without any further communication.

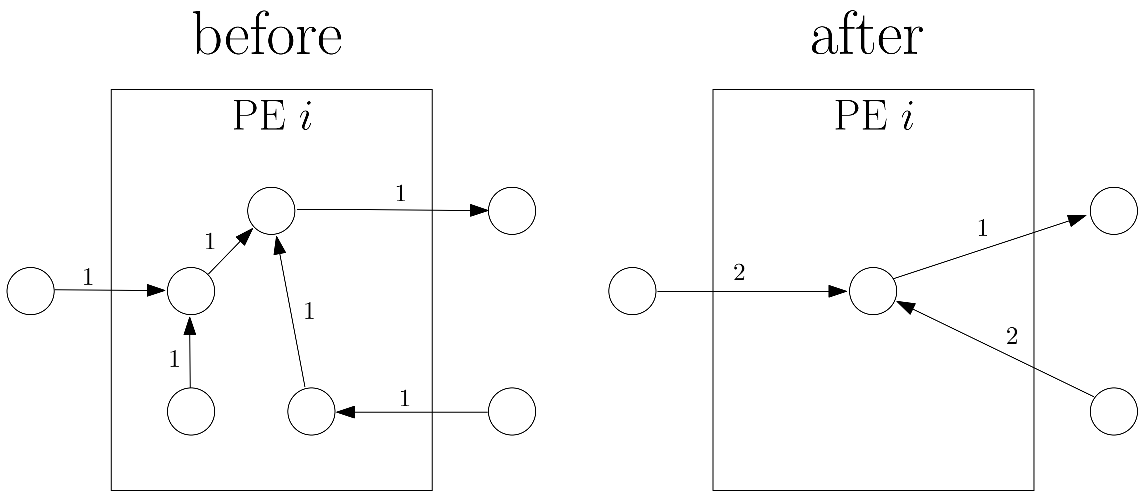


Figure 4.1: Local contraction visualization

4.1.4 Load Balancing

The time and space complexity of our algorithms depends on the number of ingoing and outgoing neighbours. If there are n' vertices on a PE, then the number of outgoing neighbours is also n'. However if *succ* is a forest the number of ingoing neighbours is arbitrary. We can first balance this by randomly permuting our instance further explained in Section 5.1. This method helps to load balance most instances, but there is one problem that is not solved.
Our instance may consist of a few high degree vertices. Just one high degree vertex on a PE can cause it to slow down massively. Our idea is to modify the instance and then recover the real *dist* and *root* array from our modified instance. In Figure 4.2 we visualized four of the five steps of our algorithm. Our initial instance is shown in step 1 where we marked each high degree vertex. Note that we picked our threshold degree for becoming a high degree vertex $d_{\max} = 3$ for our example. In step 2 we modify our instance by determining all vertices v that point to a high degree vertex and transform v into a root. In the next step we rank our modified instance. We can use this modified instance and a linear number of operations to generate our instance of high degree vertices. In the last step we rank our instance of high degree vertices and by basically performing one iteration of pointer doubling we calculated all final *root* and *dist* values. We now explain chronologically each step of the algorithm.

Determine high degree vertices In the first step every vertex v sends a packet to its successor $succ[v]$. Just like our request-reply scheme we remove duplicate requests but additionally count packets to the same successor. Note that we just want the indegrees of all vertices and do not need a reply. When using the two-level global message exchange we use the indirection PE for removing duplicate packets to the same successor as well. Upon receiving the requests we can count the degrees and every vertex whose degree is at least $d_{\max}$ becomes a high degree vertex.
Modify instance In the second step every vertex v asks its successor if it is a high degree vertex and if so we set $succ'[v] \leftarrow v$ such that the high degree vertices have no ingoing neighbours. If the successor is not a high degree vertex we do not change the successor $succ'[v] \leftarrow succ[v]$. This step consists of one request-reply scheme.
Rank modified instance In this step we rank our modified instance $succ'$. Note that this instance has the same size but the maximum vertex degree is bounded by $d_{\max}$ and we denote the resuls as $root$ and $dist$.
Create instance of high degree vertices We first need to restore our pointers to the high degree vertices. Every vertex v where $v \neq succ[v]$ sends a request to $v' = root[v]$. Note that v' is never a high degree vertex itself. In the first case $succ[v']$ is a high degree vertex. Then it sends back $succ[v']$ such that $root[v] \leftarrow succ[v']$ and $dist[v] \leftarrow dist[v] + 1$. In the other case v' sends back a placeholder such that v does not need to update anything. Now each high degree vertex v performs one step of pointer doubling $root[v] \leftarrow root[root[v]]$ and $dist[v] \leftarrow dist[v] + dist[root[v]]$. Now $root[v]$ is the correct root pointer or $root[v]$ is a high degree vertex and $dist[v]$ is also the correct distance to $root[v]$. This means we have our instance of high degree vertices.
Calculate final ranks We first rank our instance of high degree vertices and now each vertex v performs one step of pointer doubling and we have our correct $dist$ and $root$ values for every vertex.

Note that during this algorithm we have two instances to rank. One is our modified instance that can be seen in step 2 of Figure 4.2. This instance is the same size as our initial instance but our maximum degree is bounded by $d_{\max}$. After this we create our instance of high degree vertices whose size is at most $\frac{n}{d_{\max}}$. Note that the degrees of our instance of high degree vertices just have a trivial bound of $\frac{n}{d_{\max}}$, however, if we set $d_{\max} = \sqrt{n}$ we reduce our maximum degree to $\sqrt{n}$ in our modified instance as well as in our instance of high degree vertices.

4.2 Ruling Set Specific Optimizations

In this section, we begin by explaining our ruling set algorithm optimized for lists. Following that, we describe the process through which we generalized this algorithm for tree rooting.
Algorithm 5 is the pseudocode of our algorithm, which is close to an implementation. We start picking $r < n$ vertices as rulers. Each of those initiates a wave by sending a packet to their successor. In every iteration of packet chasing all incoming packets from the $recv\_buffer$ are processed and written into the out_buffer. The algorithm stops when

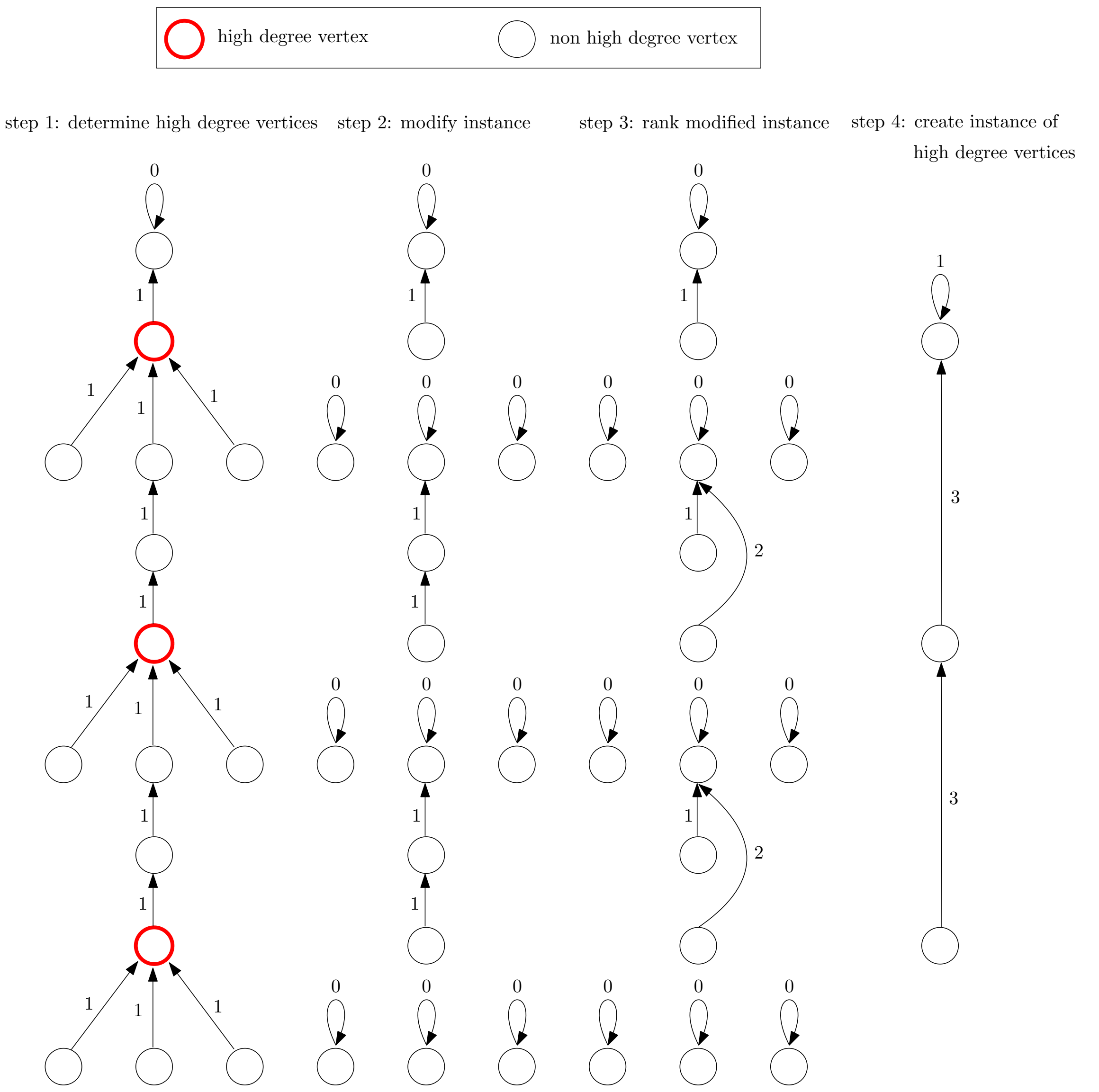

high degree vertex
non high degree vertex
step 1: determine high degree vertices
step 2: modify instance
step 3: rank modified instance
step 4: create instance of high degree vertices
0
1
2
3

Figure 4.2: Removal of high degree vertices

there are no active waves left. Note that in an parallel implementation of Algorithm 5 we can only stop our while loop in Line 9 when every PE receives zero packets in their *out_buffer*. After packet chasing we can rank the sublist of rulers by any list ranking algorithm. The successors of each ruler v are then $root[v]$ with an edge weight of $dist[v]$. In Line 20 we have to restore our final *dist* values for our rulers. Note that in a list the distance of v to the start plus the distance of v to the end is $n - 1$. Since the *rank* from a ruler is the distance to the start of the list, $n - 1 - rank$ is the distance to the end of the list shown in Line 20. In Line 22 we rank all non rulers.
In related work in Chapter 3 we showed that there are many different implementations and compared three of them, as shown in the Table 3.2.
The first version chooses random rulers and follows all packets until they reach the next ruler or the end [8]. The second version chases packets for a fixed number of iterations,

leaving some unreached vertices [38]. The third spawns new rulers each time a packet reaches a ruler, making the number of active waves constant [41]. Sibeyn showed that the third variant is superior to the second, as shown in Table 3.3. However, we cannot conclude from the literature that this algorithm is superior to simply chasing packets to the end. Chan et al. compared two list ranking methods in 2005, where one version picked rulers randomly and chased packets to the end, and the other version picked rulers in a deterministic way to bound the maximum distance between two consecutive rulers, and the randomised algorithm performed better [8]. Since we are not aware of any comparison of spawning rulers to just chasing packets to the end, we compare these techniques. We name the first variant where packets are chased until the end and no rulers are spawned *ruling set chase* and the third variant where new rulers are spawned whenever a wave dies *ruling set spawn.*

Ruling set chase starts by randomly selecting r rulers, all of which send packets. The maximum number of iterations in the while loop in line 9 is equal to the longest distance between two consecutive rulers (or the last ruler and the root vertex). The longest distance is approximately $\frac{n}{r}\log(r)$ with Lemma 3.4.3. We test before each iteration with an allreduce over the number of active waves indicating if there is need for further iterations. Note that in this version our packets as defined in Line 6 omit the distance, since all packets in iteration i also have distance i.

Ruling set spawn picks rulers, sends packets and forwards packets the same way as ruling set chase. However Sibeyn suggested that whenever a wave hits a ruler, a new local ruler is spawned [41]. This limits the number of iterations of the while loop in Line 9 with $\frac{n}{r} + 1$ w.h.p. if $\frac{r}{p} \geq 64\log(p)$ [41], which means that there must be on average at least $64\log(p)$ rulers per PE selected at the beginning. Note that this inequality is not easy to satisfy for large p and also depends on the choice of r. Sibeyn called this probabilistic bound an "obstacle for optimal-time PRAM algorithms" [41] and we will explain why the number is just probabilistically bounded. The number of rulers r tells us the number of active waves. However, in the end it may happen that we cannot spawn a new ruler locally, but we could on another PE which means a wave dies. In Figure 4.3 we can see that we set $r = 3$, but the number of active waves decreases already in the first iteration. Vertices labelled r are rulers. We adapted this algorithm by setting the number of packets leaving in

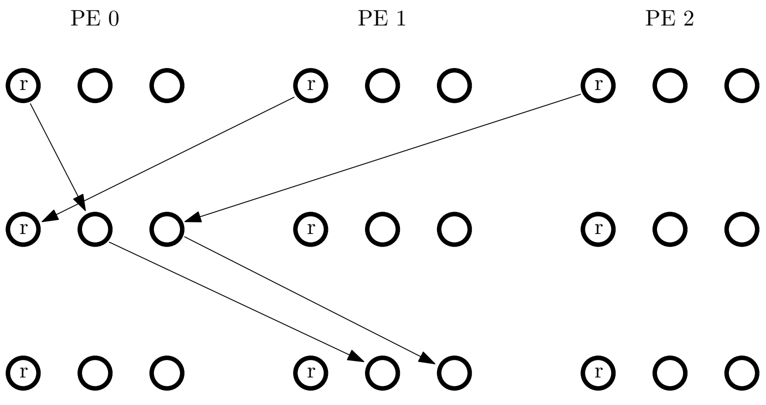


Figure 4.3: Wave by a ruler dies

each iteration to a fixed number for each PE. A PE with n' vertices will pick $n'\frac{r}{n}$ rulers, and in each further iteration $n'\frac{r}{n}$ packets leave our PE. If a PE would forward more than this threshold these packets are queued in a FIFO queue. If a PE would forward less than this threshold, then first packets from our queue are forwarded and if that is still not enough then new rulers are selected. Our small adjustment also ensures that there are always r active waves and always $\frac{n}{r}$ iterations in Line 9. However note that load imbalances can still happen when all ingoing neighbours from PE i forward a wave in the same iteration with or without our adjustion. Sibeyn proved for his algorithm that there will be about $r\log(\frac{n}{r})$ rulers at the end [41], which we also observed for our adapted approach.

Algorithm 5: Ruling Set for Lists

```
Input: Successor Array $succ \in \mathbb{N}^n$, Number of Rulers $r \in \mathbb{N}$
Output: Dist Array $dist \in \mathbb{N}^n$
// Initialize Variables
$R \leftarrow$ choose_rulers($succ, r$)
$root \leftarrow \{0, .., n-1\}$
$dist \leftarrow \{0, .., 0\}$
$out\_buffer \leftarrow$ Vector<$\mathbb{N} \times \mathbb{N} \times \mathbb{N}$>(0)
// Send Packets
for $ruler \in R$ do
    $packet \leftarrow (succ[ruler], ruler, 1)$ // (destination,ruler_source,distance)
    $out\_buffer$.push_back($packet$)
$recv\_buffer \leftarrow$ send($out\_buffer$)
// Chase Packets
while $\neg recv\_buffer.is\_empty()$ do
    $out\_buffer \leftarrow$ Vector<$\mathbb{N} \times \mathbb{N} \times \mathbb{N}$>(0)
    for $(destination, ruler\_source, distance) \in recv\_buffer$ do
        $root[destination] = ruler\_source$
        $dist[destination] = distance$
        if destination is neither final nor ruler then
            $new\_packet \leftarrow (succ[destination], ruler\_source, distance + 1)$
            $out\_buffer$.push_back($new\_packet$))
    $recv\_buffer \leftarrow$ send($out\_buffer$)
// Rank Sublist of rulers
$ranks \leftarrow$ algorithm()
// $(i, rank) \in ranks$ iff vertex $i$ has rank $rank$ in weighted reversed
   sublist
// Calculate Final Ranks
for $[i, rank] \in ranks$ do
    $dist[i] = n - 1 - rank$
for $0 <= i < n \wedge i \notin R$ do
    $dist[i] = dist[root[i]] - dist[i]$
return $dist$
```

Comparison of Versions

Comparing these versions cannot be done by simply running the algorithms for the same r and comparing the running times, because one could argue that r should be set differently for the two versions, since ruling set chase reduces our list to size r and ruling set spawn reduces it to $r\log(\frac{n}{r})$. We compare the running times for the same result, i.e. the same reduced instance size so we can postpone the choice of r until later. In Figure 4.4 on the left plot we can see for $p = 256$ the comparison of both versions for $\frac{n}{p} \in \{10^5, 10^6, 10^7\}$.

reduction factor f	5	10	15	20	25	30	35	40
ruling set spawn	17	44	73	105	137	170	203	241
ruling set chase ($p = 2^8, \frac{n}{p} = 10^5$)	75	151	248	305	415	426	452	581
ruling set chase($p = 2^{14}, \frac{n}{p} = 10^7$)	115	216	340	452	594	710	771	813

Table 4.1: Number of iterations for ruling set spawn versus ruling set chase

Note that the $x-$axis is the factor by which we reduced our instance and the $y-$axis is the running time until we start our recursion in Line 18 since the rest of the algorithm is the same. For smaller n ruling set spawn performas superior, and for larger n ruling set chase. In Figure 4.4 on the right plot we can see the same experiment for $p = 16384$. Here ruling set spawn is far superior for smaller n, and for $\frac{n}{p} = 10^7$ ruling set spawn is superior for most reduction factors. In the Table 4.1 we can see the number of iterations which is at the same time the number of global message exchange while chasing packets. In theory for ruling set chase for given reduction factor f this means $r = \frac{n}{f}$ and the number of iterations is $\frac{n}{r}\log(r)$ with Lemma 3.4.3. Ruling set spawn needs $\frac{n}{r}$ iterations and the formula $\frac{n}{f} = r\log(\frac{n}{r}) \Leftrightarrow f = \frac{\frac{n}{r}}{\log(\frac{n}{r})}$ holds [41]. Note that the number of iterations does not depend on n for ruling set spawn, but it does for ruling set chase. This is the reason why ruling set chase is the worse scaling algorithm considering the communication overheads. Note that ruling set chase has less communication volume because we can omit the distance in packets however for recursive calls we cannot omit the distance because the edge weights are arbitrary, which reduces its few advantages. We conclude that ruling set spawn is the better scaling algorithm and will further engineer this version.

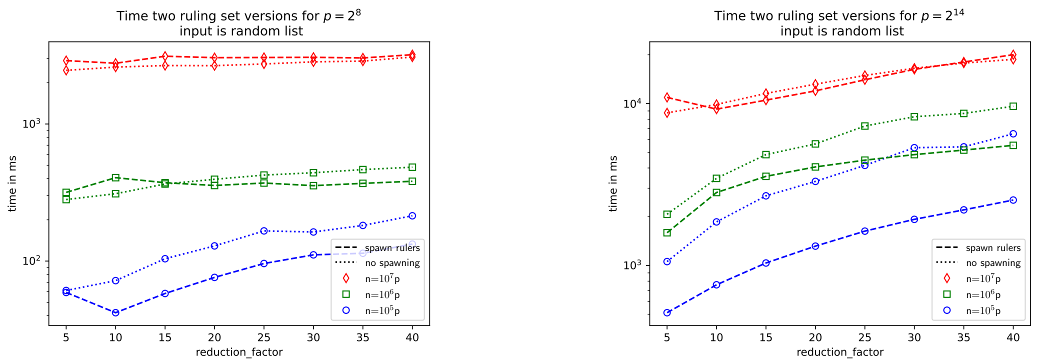


Figure 4.4: Comparison of ruling set chase versus ruling set spawn

We now want to approach the determination of r. Note that the problem becomes more dimensional since this algorithm is called recursively. Chan et al. suggested setting $r = \frac{n}{p}$ [8] when testing their ruling set chase which means they reduced their instance to size $\frac{n}{p}$.

Sibeyn, however, proposed for ruling set spawn $r = \frac{3p\sqrt{n}}{\ln(\frac{\sqrt{n}}{3p})}$, resulting in a reduced instance size of $(1 + \frac{\ln(\ln(\frac{\sqrt{n}}{3p}))}{\ln(\frac{\sqrt{n}}{3p})}) \cdot 3p\sqrt{n} \approx 3p\sqrt{n}$. We want to test Sibeyn's formula for r, since he proposed it for his algorithm. First, we must note that he concludes r as a result of an optimization problem he defined, where he set $\alpha = 2 \cdot 10^{-4}$ seconds and a running time of $5 \cdot 10^{-5} n$ seconds for a sequential list ranking algorithm of size n which are appropriate numbers for the Intel Paragon [38]. Sibeyn models the running time as one reduction phase of ruling set and then sequentially rank the sublist of rulers [38].

4.2.1 Choose Parameters for Ruling Set Spawn

Sibeyn considered a general routing algorithm for global message exchange that has s start ups. Our two-level approach consists of approximately $2\sqrt{p}$ startups wich we explain in Section 4.1.1 and our one level approach of p. His formula is illustrated in the Equation 4.7, where the constant we are now optimizing is c, which Sibeyn set to $\frac{1}{3}$ [38]. Note that W is the Lambert W-function, which is the inverse of the function $x \mapsto x \cdot e^x$.

$$r = \frac{\sqrt{nps}}{c\ln(\frac{c\sqrt{n}}{\sqrt{ps}})} \Leftrightarrow \frac{n}{r} = \frac{c\sqrt{n}}{\sqrt{ps}}\ln(\frac{c\sqrt{n}}{\sqrt{ps}}) \Leftrightarrow c = \frac{e^{W(\frac{n}{r})}}{\frac{\sqrt{n}}{\sqrt{ps}}} \tag{4.7}$$

Our idea is to run our algorithm for different r and then compute back c and use this result to compute r for deeper recursion calls. In figure 4.5 we evaluated our algorithm for

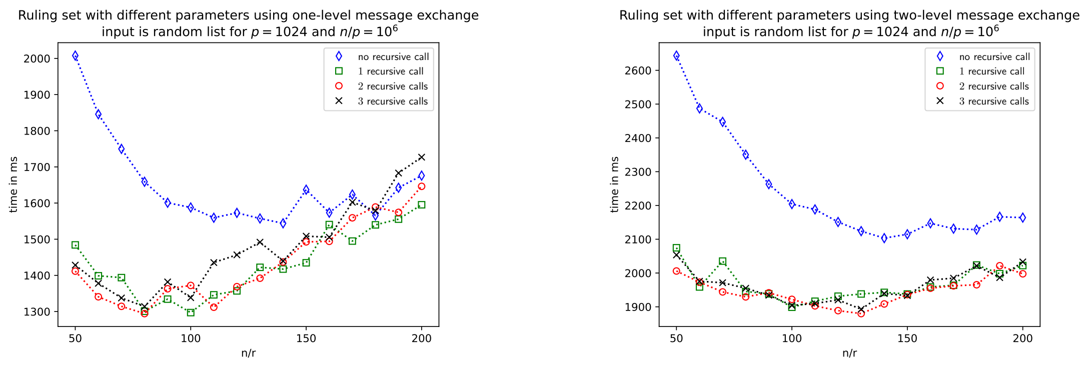


Figure 4.5: Experiments for determining c

different parameters and for one-level message exchange our best data point is for $\frac{n}{r} = 80$ for which we can conclude $c = 0.796$ and for two-level message exchange $\frac{n}{r} = 130$ for which we can conclude $c = 0.290$ and in both cases three recursive calls of this algorithm.

Own Experiments

In our own experiments, we observed a much simpler pattern. Sibeyn's formula led to very strange choices of r in some cases. For very large p we had very small reduction factors.

We notice that $r = \frac{n}{100}$ is always a good pick. This would result in a reduced instance of size $r \log(\frac{n}{r}) = \frac{n}{100} \log(\frac{n}{\frac{n}{100}}) \approx 0.0461n$. Then we recursively call our ruling set algorithm until we have reduced our instance to where $\frac{n}{p} \leq 10^4$, and break the recursion with pointer doubling. We compared Sibeyn's method and our simple approach for a random list with $\frac{n}{p} \in \{10^5, 10^6, 10^7\}$ for different p and for one-level and two-level message exchange. Note that in some cases Sibeyn's formula resulted in $r > n$, which is impossible, and therefore called directly pointer doubling, but in all other cases we made three recursive calls. In Figure 4.6 we can see that both approaches lead to similar results. However for $p \geq 2^{12}$ our apprach had a better performance for all instance sizes and also for one-level and two-level message exchange. We conclude that our simplified approach scales better, but we note that choosing r is far from trivial and there is room for improvement.

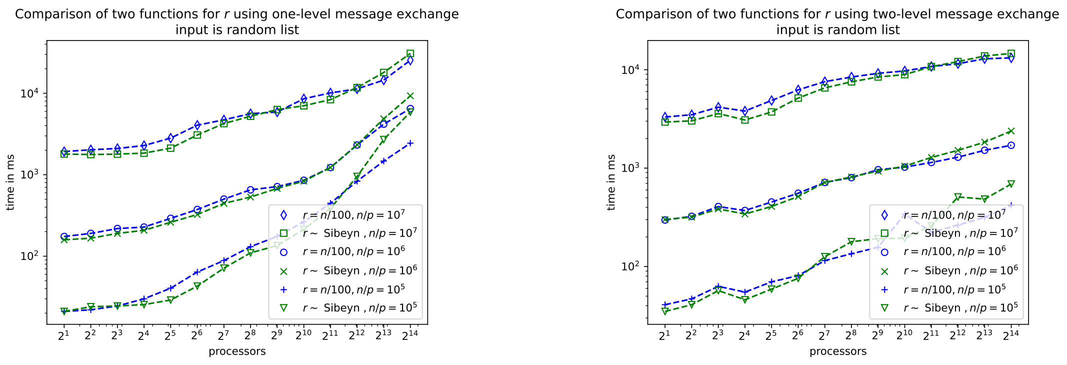


Figure 4.6: Comparison of two functions for determining r

4.2.2 Ruling Set for Tree Rooting

The first difference with forest as input is that in addition to a *dist* array, we also want a correct *root* array, which tells us for each vertex its corresponding root. Note that in Algorithm 5 our sublist of rulers is reversed. In order for each non-ruler to update its *dist* pointer according to the *dist* pointer of our preceding ruler, we must either reverse our sublist of rulers before recursion or reverse our initial list of all vertices. Even though reversing the edges is more expensive at the beginning, it has advantages. The most important advantage is that we have a noticeable reduction in size for all *succ* instances. To understand why, we first need to explain the problem of not reversing edges at the beginning. By implementing ruling set spawn all leaves become rulers, since they will never be reached by a packet. For example, if *succ* is a binary tree, then about half of our vertices are leaves resulting in a reduced instance size of at least half. By reversing the edges, we get noticeable reductions in size regardless of our instance. In Figure 4.7 we visualize the problem. Our input is a binary tree and some inner vertices but all leaves are rulers. When we first reverse all edges we can exclude also all leaves due to our ruler picking policy explained next.
Note that ruling set chase has a similar problem. Leaves are either picked as rulers or they are unreached vertices at the end of packet chasing. Both have to be ranked independently.

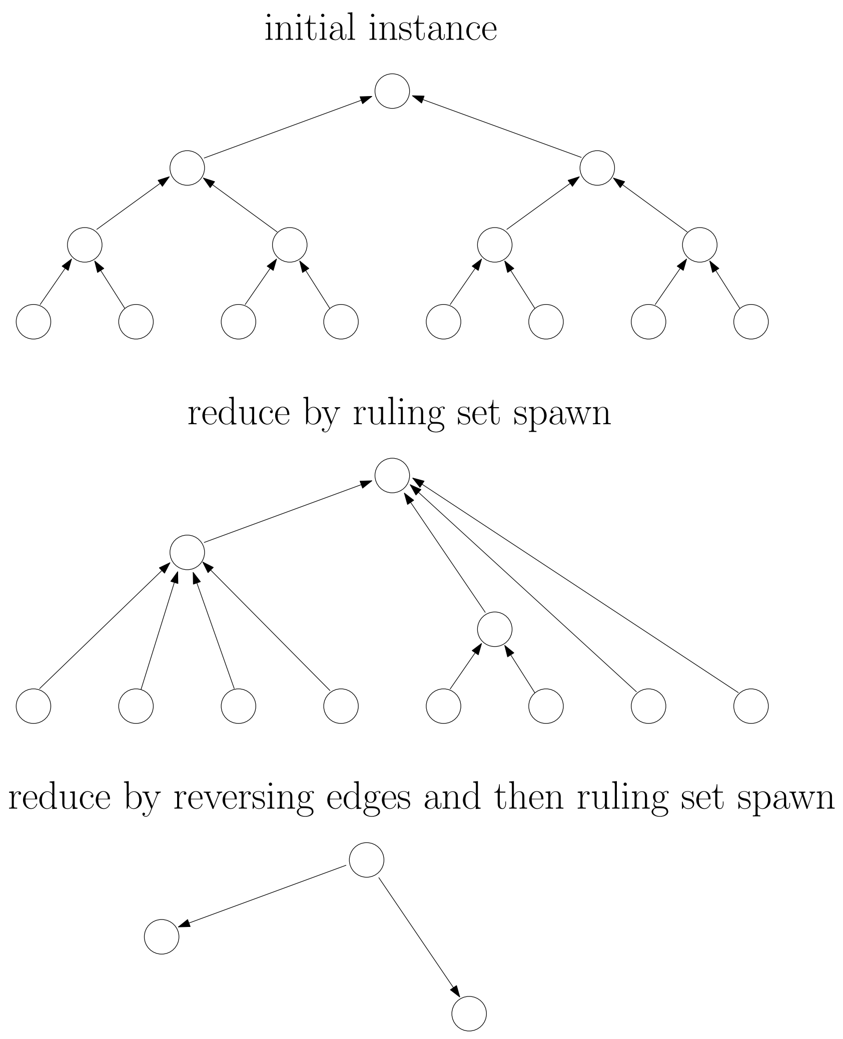

initial instance
reduce by ruling set spawn
reduce by reversing edges and then ruling set spawn

Figure 4.7: Sibeyn's algorithm on binary in tree

We now explain all differences when generalizing ruling set spawn for forests. Our input is still our successor array *succ* and r. In context of list ranking r is the number of rulers initially picked and also the number of active waves during each iteration. In this context r displays just the number of active waves.

Ruler Picking Policy In our implementation we know all the roots because they satisfy $succ[i] = i$. Since they become rulers anyway we first pick all the roots as rulers and then uniform at random from the set of vertices that are not Leaves. Note our ruler picking policy views a tree of size 1 as a leaf so these never become a ruler. Now we explain all differences in each part as proposed in our pseudocode in Algorithm 5.

Initialize Variables As just explained, we need to reverse all edges, which can be done by a single global message exchange. Now every vertex knows all ingoing neighbours. Note that every vertex has one outgoing neighbour but can have an arbitrary number of ingoing neighbours. Every ingoing neighbour receives one packet during the course of this algorithm so PE i with e ingoing neighbours iteratively picks vertices as rulers that send a packet to each ingoing neighbour until $t = e/\frac{n}{r}$ packets in total are sent. Note

that for a list this means in total r rulers are initially picked but for forests $\leq r$ rulers are initially picked however in any case there are r active waves.
Chase Packets In each iteration in Line 9 we process our incoming packets. Each packet is targeted to a single vertex v. We forward this packet by sending it to any ingoing neighbour of v. We process as many packets as long as we do not exceed our threshold t. If we were to exceed t, we would put the packets in a FIFO queue q to be processed in later iterations. If we processed all incoming packages and do not reach t, we process packets from q first, and if we still do not reach t, we would pick new rulers that send packets to all incoming neighbours until we hit our threshold t. Since every PE with e ingoing neighbours send $t = e/\frac{n}{r}$ packets per iteration we always need $\frac{n}{r}$ iterations in Line 9 until every vertex is reached.

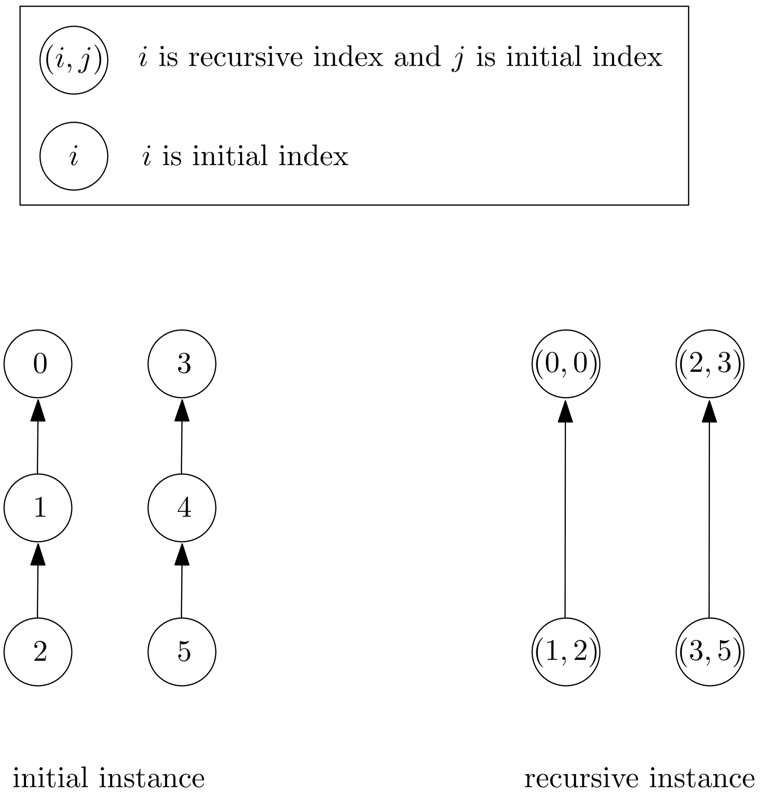


Figure 4.8: Representation of recursive instances

Rank Sublist of Rulers Since we also want to compute correct *root* pointers, we obviously also need to call a forest rooting algorithm for the sublist of rulers. Besides arbitrary edge weights, recursive algorithms also have an adaptation. The problem is illustrated in figure 4.8. Our instances are defined as successor arrays *succ* with consecutive indices. So in the recursive instance, the vertex with recursive index 3 would have a *root* pointer to the vertex with recursive index 2. However, we want the result to be the initial index 3. So in addition to our new successor array with edge weights recursive instances also consist of an array indicating for each vertex its initial index.

Calculate Final Ranks Since we added also initial indices to our recursive instances our rulers already have correct *root* and *dist* values. Thus all non rulers v send a request to the ruler that initiated the wave that hit v. This ruler is stored in $r = root[v]$ and ruler r answers by $root[r]$ and $dist[r]$ which consists of one request-reply scheme.

4.2.3 Non Blocking Packet Chasing

Sibeyn proposed an approch for his ruling set spawn called interlacing computation and communication [41]. The basic idea is that global message exchange is divided into $p - 1$ point-to-point message exchanges such that we can directly process packets after each point-to-point exchange and do not wait until we receive all packets from all PEs. This method effectively doubles the efficiency of the algorithm on average [41]. This approach however still performs $p - 1$ point-to-point messages that hinder scalability. We adjusted this a approach by using a message queue. Uhl is developing a generalized message queue framwork that buffers messages locally until a certain threshold is exceeded [42], which he first introduced in a previous distributed memory triangle counting algorithm [33]. This message queue can perform a nonblocking poll that returns all received messages to immediatly process them, which is at the core a similar idea than interlacing computaiton and communication. Additionally this framework allows message indirection, aiming to reduce latency to not perform $O(p)$ point-to-point exchanges. We use indirections similar to the two-level message exchange 4.1.1.
This message queue allows custom aggregation and deaggregation. In our usecase we just append a packet to the existing packets. Messages are buffered in the message queue locally, until a certain global threshold exceeds.
Our implementation starts sending the same number of waves as our blocking ruling set, by posting a message for each wave. Note in our implementation the threshold is ∞. In a while loop our algorithm processes all local messages and posts the forwarded and new spawned waves. The loop ends with a termination detection that basically returns all buffered messages.

5 Evaluation

In this chapter we evaluate the performance of our algorithms across various test scenarios with processor counts up to 16384. Initially, we detail the selection of test instances and the methodology used for their generation.
Experimental Setup We implemented our algorithms in C++ and our implementations are available at `https://github.com/ThomasWeidmann/MasterarbeitCode`. For experiments we use the thin compute nodes of the SuperMUC-NG supercomputer at the Leibniz Supercomputing Center. The thin compute nodes consist of 8 islands with 792 nodes per island which makes up a total of 6336 nodes. Each node consists of an Intel Skylake Xeon Platinum 8174 processor with 48 cores and in total 96GB of memory per node using the SUSE Linux Enterprise Server (SLES) operating system.
We compile our code with the g++ 12.2.0 and Intel MPI 2021 using optimization level -O3. All of our datapoints are the average of 3 runs.

5.1 Methologies for Test Instances

KaGen is a communication-free graph generator [15]. We utilize KaGen for generating distributed lists at scale. KaGen computes a pseudo-permutation for a specified global size n using a Feistel cipher. Additionally, we employ this list as a test instance to manipulate our instances. A *random* instance of our input instance *succ* is created by first generating a list with KaGen, where we link the final vertex to the initial vertex, which we denote by $succ'$. Assuming our input successor array is *succ*, for every edge $(i, succ[i])$, our random instance includes the edge $(succ'[i], succ'[succ[i]])$. We also explore the use of Boruvka trees. Boruvka's algorithm [7] calculates the minimum spanning trees for undirected weighted graphs. The initial step involves identifying and picking one adjacent edge with the lowest weight for each vertex v, resulting in a pseudotree. These pseudotrees are subsequently converted into a tree. Our implementation picks from the set of adjacent vertices with lowest edge weight, the one with lowest index. When transforming the pseudotrees into a rooted tree, the vertex with the lowest index becomes a root. Note that our input is an undirected graph to which we first assign edge weights $w \in \{0, .., k\}$ uniformly at random. We use Boruvka trees from RGG or GNM (Erdős Rényi) graphs which are also generated by KaGen. For scaling random geometric graphs (RGG) KaGen places vertices uniformly at random within a unit square (RGG2D) or unit cube (RGG3D). Each MPI process is assigned a subset of these vertices. Edges are formed between vertices if the Euclidean distance between them falls below a certain threshold. The $G(n, m)$ model chooses a graph uniformly at random from the set of all possible graphs with n vertices and m edges.

5.2 Algorithms

We have developed a variety of versions for pointer doubling and ruling set algorithms. Here is a brief overview of the foundational components we employed:
Message exchange For one-level message exchange, we utilize the built-in MPI routine alltoallv. Our approach to two-level message exchange involves using this routine twice, engaging at most $\sqrt{p} + 2$ PEs, as as explained in Section 4.1.2.
Request-reply Our algorithms frequently require a double round of global message exchange to respond to a collection of requests. In some experiments we use the optimization *remove duplicate requests*, where we refine our request-reply process by eliminating duplicate requests with the aid of hash tables, a method thoroughly explained in Section 4.1.2.
Now we introduce our algorithms for list ranking and tree rooting. It is important to note that our input consists of a successor array *succ* with each PE being allocated an equally sized segment. However our algorithms can also handle irregular instances which occur in recursive calls of our algorithms.

Algorithms for List Ranking

Although tree rooting algorithms can be applied to list ranking, we first focus on our implementations specifically tailored for list ranking.
List pointer doubling This algorithm utilizes techniques aiming to minimize the communication volume to one integer per request and one integer per reply [38]. It requires $1 + \lceil \log_2(n-1) \rceil$ iterations, each involving a request-reply cycle.
Double pointer jumping This method starts by computing the predecessor array from the given successor array *succ*. It then proceeds through $\lceil \log_2(n-1) \rceil$ iterations, each featuring one global message exchange [38].
List ruling set This approach initially picks $r = \frac{n}{100}$ rulers, leading to 100 iterations, as detailed in Section 4.2. The algorithm recursively calls itself until $\frac{n}{p} \leq 10^4$ at which point it switches to list pointer doubling.

Algorithms for Tree Rooting

Now we turn our attention to our algorithms for tree rooting. Importantly, in addition to determining the distance of every vertex from its root, each vertex also aims to identify the root itself.
Forest pointer doubling This algorithm operates similarly to list pointer doubling. Additionally it assesses, after each iteration, whether there are active vertices by performing an allreduce operation with the addition operator over the number of active vertices per PE. The algorithm stops when there are no active vertices remaining.
Euler tour This algorithm calculates the weighted Euler tour [20] and then applies the list ruling set algorithm to this tour. Note that this approach is specifically utilized when the input *succ* represents a tree, as in such cases the Euler tour forms a single list, allowing for an efficient application of our list ruling set algorithm.
Forest ruling set This technique is a generalization of the list ruling set, as detailed in Section 4.2.2. It employs $r = \frac{n}{100}$ rulers, resulting in 100 iterations each consisting of one

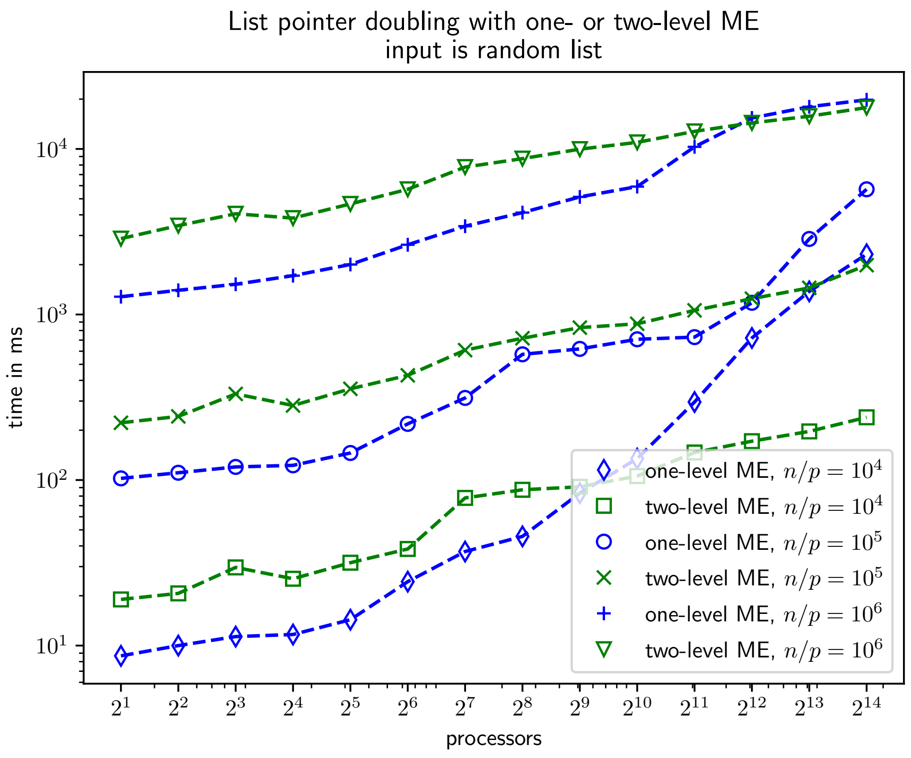
global message exchange. Note that after the set of rulers is ranked, we need one request-reply scheme to calculate the final values for all vertices. Similar to the list ruling set, this algorithm recurses until $\frac{n}{p} \leq 10^4$, at which point forest pointer doubling is employed. We have also developed a version using non blocking communication, referred to as *non blocking forest ruling set* detailed further in Section 4.2.2. Our non blocking forest ruling set performs one reduction. If $\frac{n}{p} \leq 10^4$, forest pointer doubling continues and otherwise the blocking version, which we just call forest ruling set, is used.
Local Contraction This algorithm initiates with a local rooting process aimed at eliminating all edges between distinct vertices located on the same PE, as detailed in Section 4.1.3. It employs a single request-reply for all remaining vertices. The modified instance is then suitable for ranking by any tree rooting algorithm. These results are used for calculating the final *root* and *dist* values for all vertices without the need for further communication.
Remove high degree vertices This method modifies our instance by identifying all vertices with a degree higher than $d_{\max}$ further explained in Section 4.1.4 by removing all connections to these vertices. The resulting modified instance can be ranked by any tree rooting algorithm. Afterwards we use forest pointer doubling for the set of high degree vertices due to the typically small size of these instances, for which pointer doubling is optimally suited. This algorithm involves a fixed number of request-reply cycles.

5.3 Preliminary Experiments

In our preliminary experiments, we evaluate the impact of two-level message exchange. Theoretically, for any given communication volume, there exists a threshold beyond which higher processor counts benefit from two-level message exchange which is further explained in Section 4.1.1. To investigate this, we test our algorithms using both one-level and two-level message exchange. The results, illustrated in Figure 5.1 and detailed further in the appendix in Section 7.1, present the running times for various algorithms — including double pointer jumping, list pointer doubling, synchronous and asynchronous forest ruling set, and list ruling set — for $\frac{n}{p} \in \{10^4, 10^5, 10^6\}$, comparing one- and two-level message exchanges.

Our findings indicate that the effectiveness of two-level message exchange depends on the algorithm and the size of the instance. Notably, for small instances where $\frac{n}{p} = 10^4$ and $p > 2^4$, two-level message exchange significantly reduces running times for the list ruling set. For larger instances, where $\frac{n}{p} = 10^6$, an improvement in performance by two-level message exchange requires $p > 2^{10}$. For all other evaluated algorithms we observe similar behavior and for $p > 2^{12}$ every algorithm benefits from two-level message exchange.

Based on our analysis, two-level message exchange proves to be advantageous for the range of evaluation sizes we consider. Therefore we focus exclusively on utilizing two-level message exchange in our further evaluations.



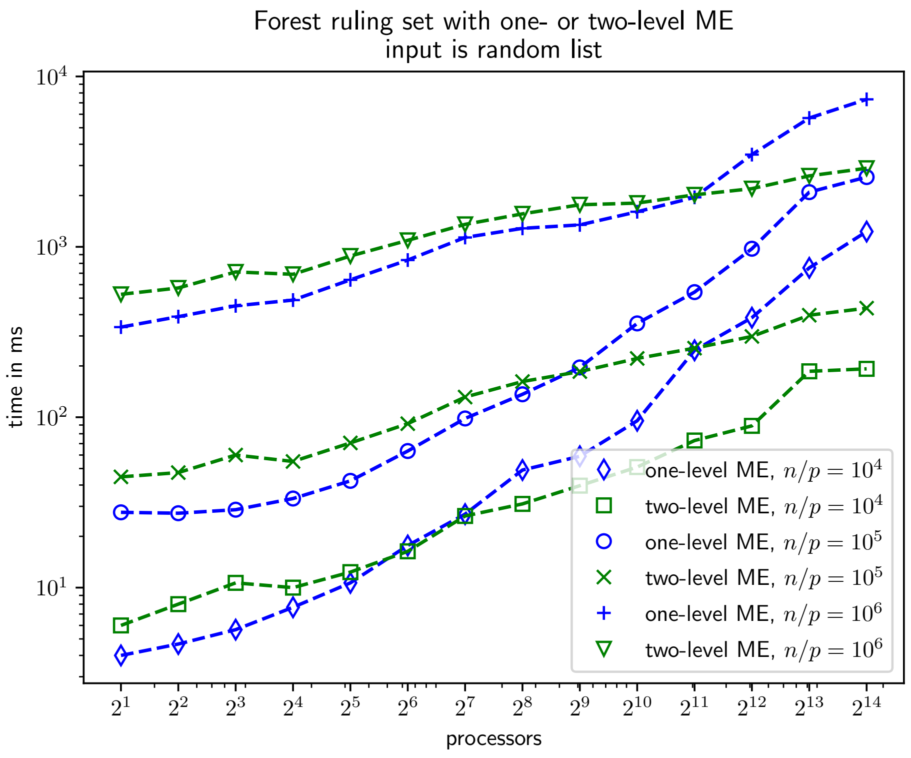


Figure 5.1: Impact of one-level message exchange versus two-level message exchange

5.4 Test instances

We demonstrate the influence of graph structure on our algorithms' runtimes using a variety of synthetic test instances. In the following we introduce all test instances we use.
List For generated distributed random list at scale we use KaGen [15]. KaGen computes a pseudo-permutation for a specified global size n using a Feistel cipher. We call this instance a *random list*.
Tree Tree instances are generated using the formula in Equation 5.8, where $\text{rand}(0, i)$ uniformly returns a random number between 0 and $i-1$. The resulting tree is characterized by logarithmic depth which is visualized in Figure 5.2, where we generated trees for various values of p with $\frac{n}{p} = 10^5$ and determined their depth. Following this, we randomly permute the vertices, as explained in Section 5.1 to load balance this tree. We call this instance a *random tree*.

$$s[i] = \text{rand}(0, i) \in \{0, .., i-1\} \tag{5.8}$$

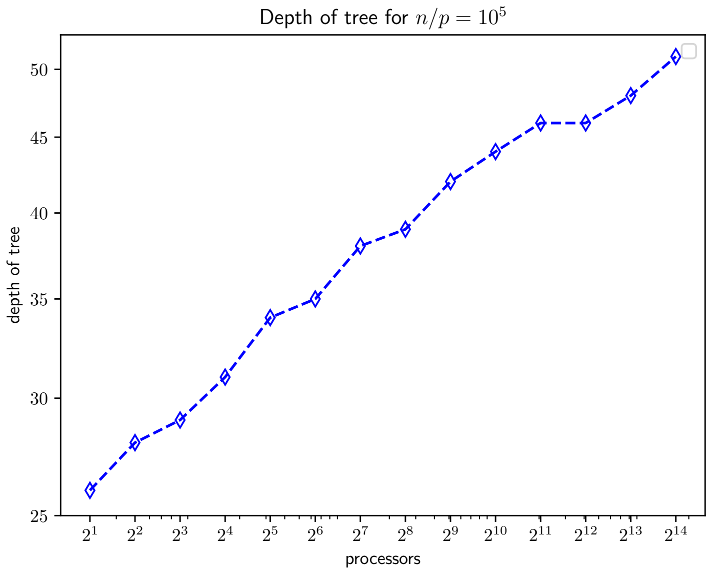


Figure 5.2: Depth of our tree instances

GNM Boruvka forest We use KaGen for generating GNM and RGG2D undirected graphs, assign uniform random edge weights from $\{0, .., k\}$ and then determine a Boruvka forest as explained in Section 5.1. Figure 5.3 displays the depths of different Boruvka forests. The trees in Boruvka forests of $\text{GNM}(n, 10n)$, $k = 0$ do not vary in depth for $\frac{n}{p} \in \{10^4, 10^5, 10^6\}$ as well as for the RGG2D Boruvka forests, which is why we plotted in this figure the depths only for one fixed $\frac{n}{p}$. The Boruvka forests for GNM graphs with $k = 0$ are very shallow with depths of at most 15 in contrary to the depths for RGG2D Boruvka forest for $k = 0$ with some trees reaching depths of several thousand. We can use the parameter k to make these trees more shallow, which makes the RGG2D Boruvka forest with $k = 10$ to a depth of approximately 50.

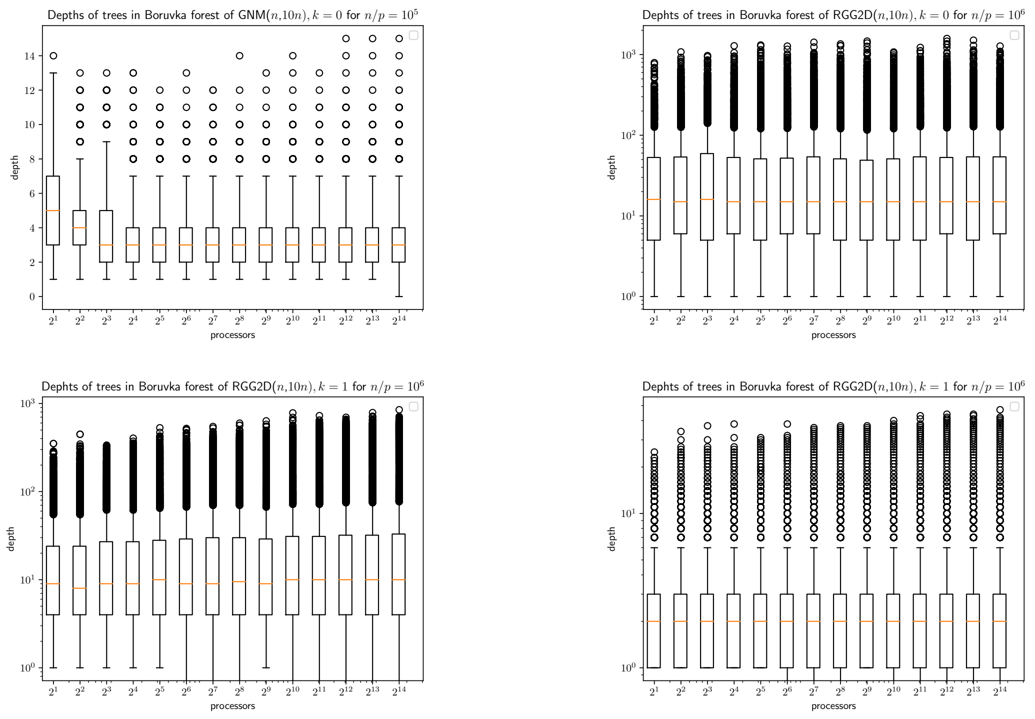


Figure 5.3: Depths of Boruvka forest for different graphs

Caterpillar Here we address instances with vertices that have very high degrees. We generated caterpillar graphs as visualized in Figure 5.4. Our caterpillar is basically a list where every *d*th vertex becomes a vertex of degree *d*.

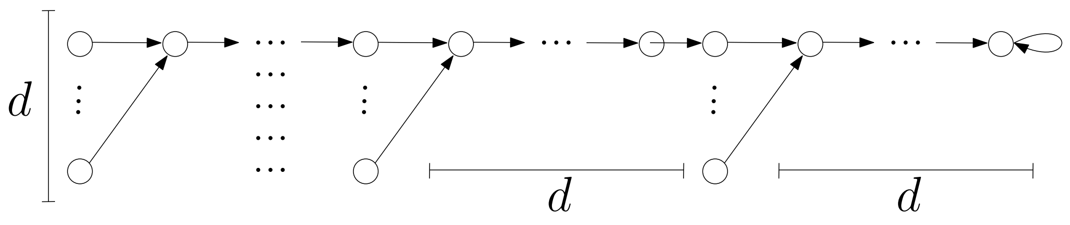


Figure 5.4: Caterpillar

5.5 Scaling Experiments

We conduct a comprehensive analysis of our algorithms across a diverse range of inputs as previously described. Our primary objective is to identify the strengths and weaknesses of each algorithm. We conducted *weak scaling* experiments, keeping the problem size per PE fixed, with processor counts up to 16384. Note that all experiments use two-level message exchange.

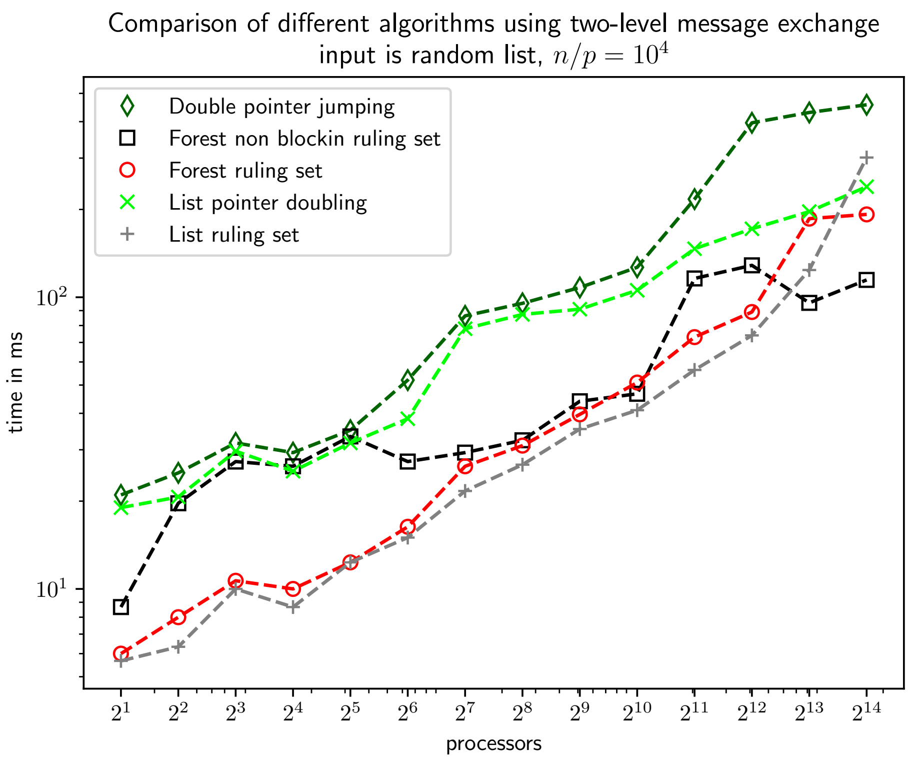
In Figure 5.5, we assessed the performance of various algorithms—including list pointer doubling, the forest ruling set, the non-blocking forest ruling set, double pointer jumping, and the list ruling set—on a random list where the ratio $\frac{n}{p}$ ranges between 10^4 and 10^6. It is important to note that an Euler tour of a list effectively doubles the list's length, and our implementation of forest pointer doubling is inherently slower than list pointer doubling, as detailed in the algorithm descriptions. Consequently, we did not apply either algorithm to list data in our tests.

Results from applying algorithms such as Euler tour ranking, forest ruling set, non-blocking forest ruling set, and forest pointer doubling on a random tree are presented in Figure 5.6, where $\frac{n}{p}$ also ranges between 10^4 and 10^6. The lower plot specifically illustrates the impact of eliminating duplicate requests on the forest ruling set's performance. Additionally, Figure 7.4 in the appendix showcases how removing duplicate requests affects forest pointer doubling on a random tree. The implications of applying forest pointer doubling and forest ruling set algorithms on RGG2D$(n, 10n)$ Boruvka forests (with $k = 0$, given that some trees exceed depths of 1000 as shown in Figure 5.3) are depicted in the appendix in Figure 7.5.

Figure 5.7 displays the outcomes for forest pointer doubling, the non-blocking forest ruling set, and the forest ruling set applied to both GNM$(n, 10n)$ and RGG2D$(n, 10n)$ Boruvka forests at $k = 0$. These instances are random and were chosen to examine the effects of varying tree depths without considering locality.

Our analysis of tree rooting algorithms on RGG2D$(n, 10n)$ Boruvka forests, for both $k = 1$ and $k = 10$, is shown in Figure 5.8. Unlike the previously mentioned instances, these are characterized by significant locality, prompting us to test our local contraction method. We then ranked the contracted instances using both forest pointer doubling and the forest ruling set. Note that the reported runtimes include the time taken to contract the instance and to restore the *dist* and *root* values for all vertices.

In Figure 5.9, we examine how the presence of high-degree vertices affects the performance of our forest ruling set algorithm on random caterpillar graphs with degrees d ranging from 10^5 to 10^8. We applied the algorithm both directly and with removing high degree vertices, setting $d_{\max} = 10^5$ because our caterpillars have degrees of only 1 and d. In all scenarios, it was necessary to remove duplicate requests. We also conducted pointer doubling tests, that are not displayed here, on some data points with runtimes averaging around a minute.



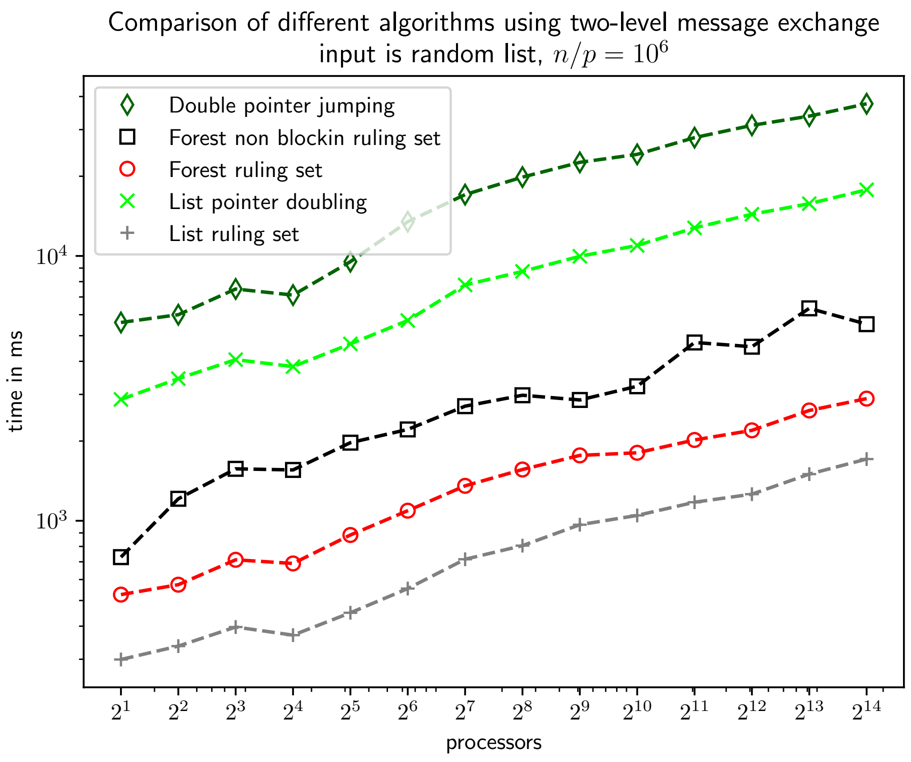
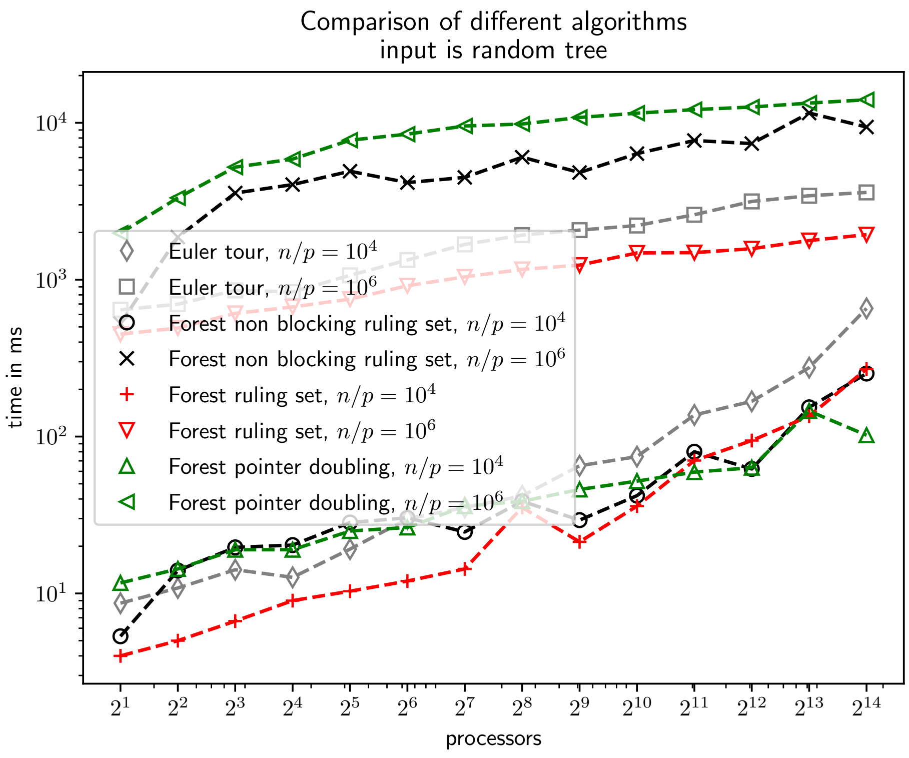


Figure 5.5: Comparison of algorithms on random list



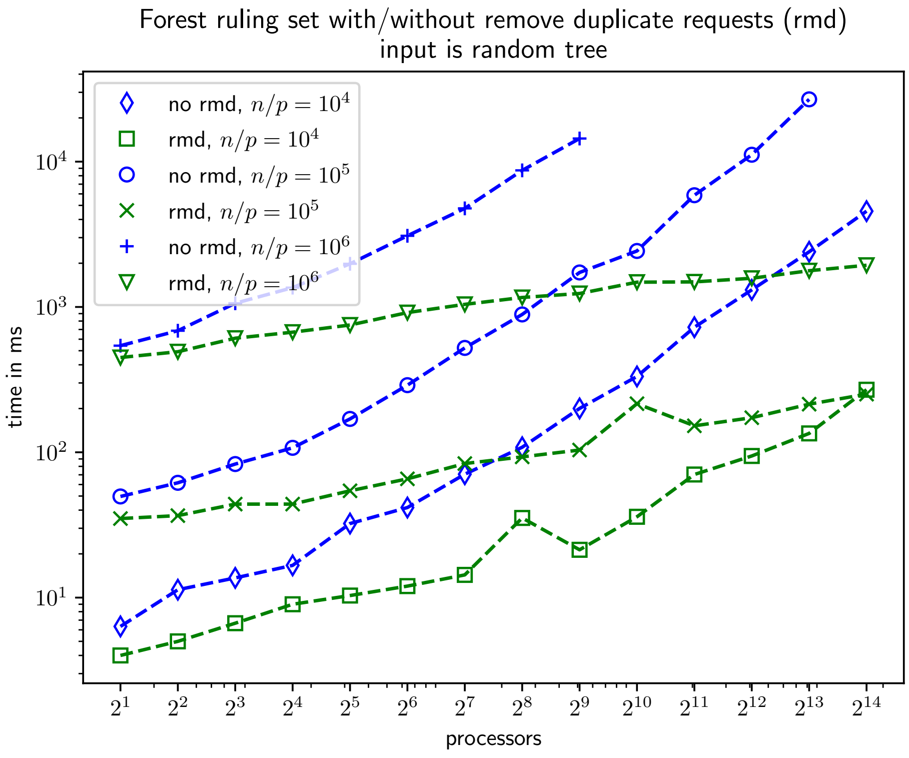
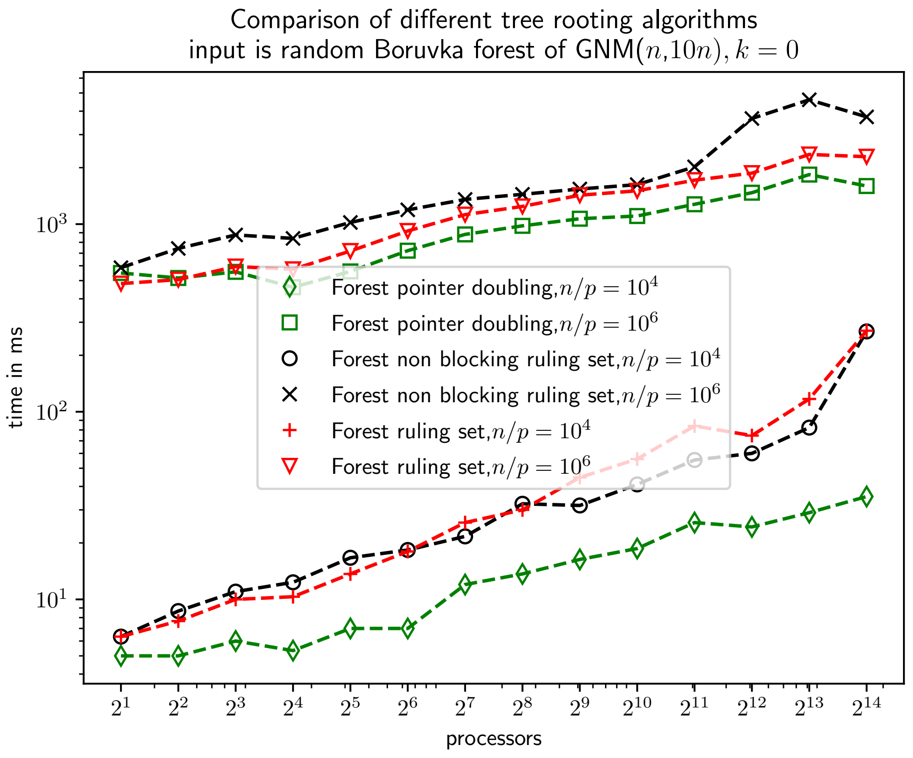


Figure 5.6: Comparison of algorithms on random tree



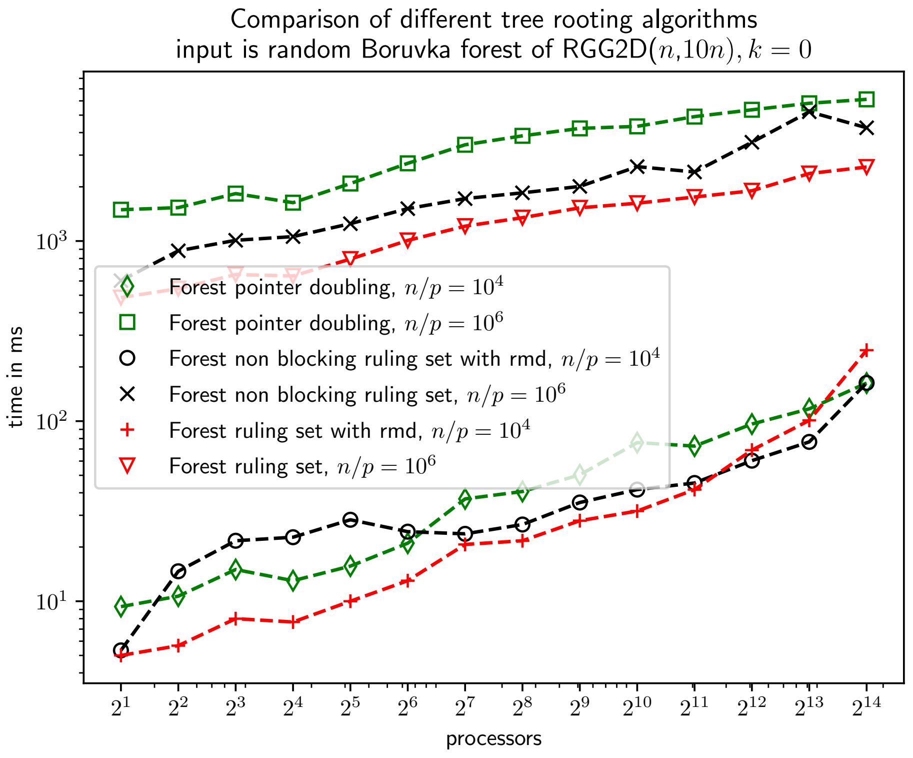
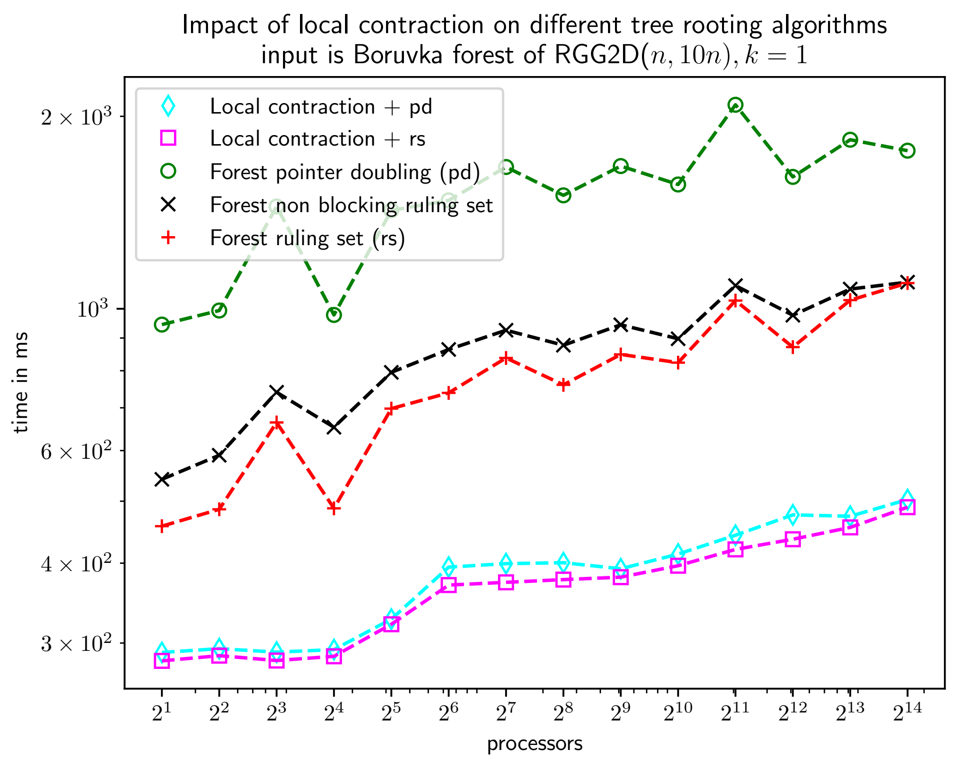


Figure 5.7: Comparison of algorithms on different random Boruvka forest



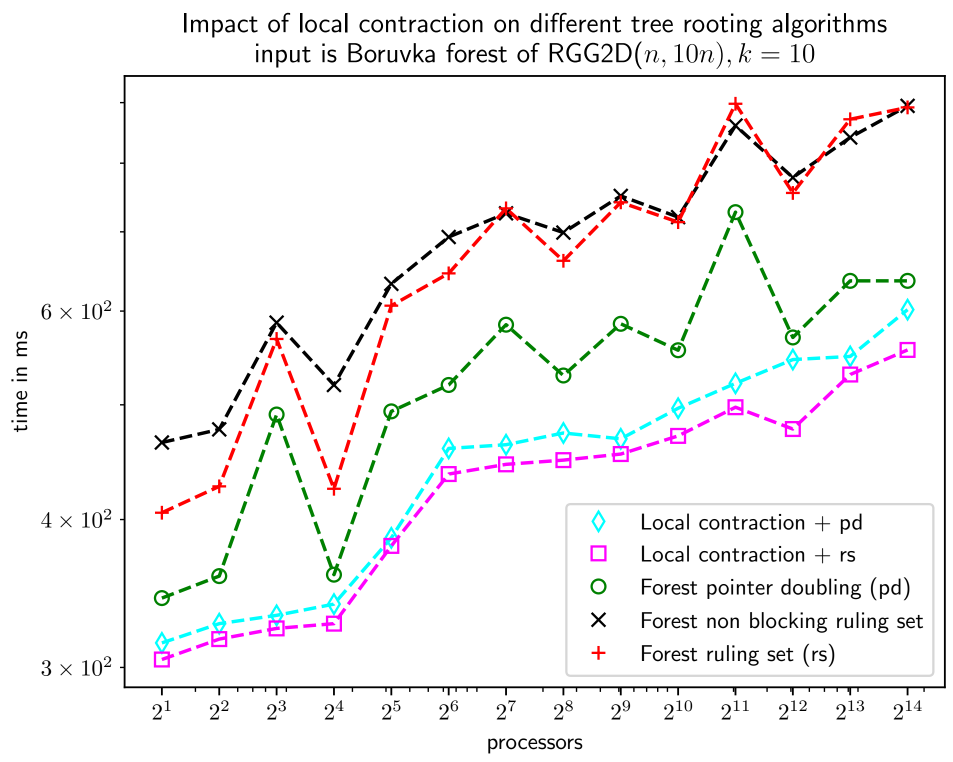


Figure 5.8: Comparison of algorithms on different Boruvka forest

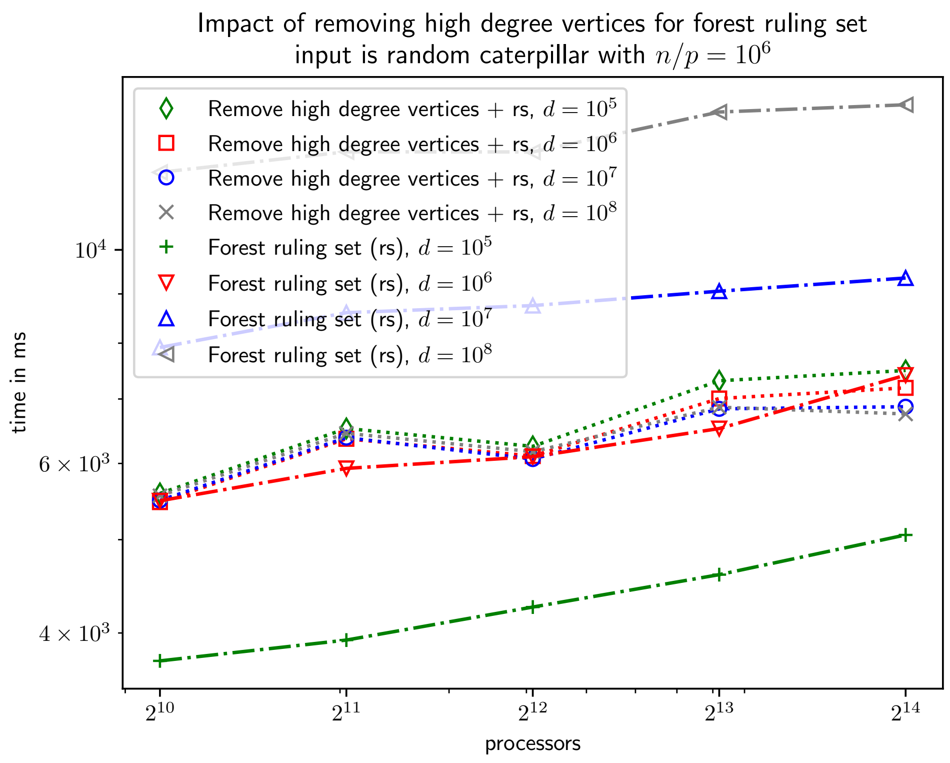


Figure 5.9: Forest synchron ruling set on randomized RGG2D Boruvka tree

Our experiments have demonstrated the runtime performance of our algorithms across a variety of instances. We now proceed to analyze these algorithms in detail.

Pointer Doubling A key advantage of the pointer doubling algorithm is its minimal communication overhead. Our pointer doubling algorithms run for $\log_2(h) \leq \log_2(n)$ iterations. The double pointer doubling variant conducts one global message exchange, while both list and forest pointer doubling require two global message exchanges. In contrast, our ruling set algorithms, with a chosen $r = \frac{n}{100}$, necessitate at least 100 global message exchanges, not including the additional overhead required for ranking the reduced instances. This explains why pointer doubling showcases strong performance for smaller sizes, where $\frac{n}{p} = 10^4$. However, as the ratio $\frac{n}{p}$ increases, the impact of communication volume and local computations on runtime becomes more pronounced. This effect increases for forests with greater depths and especially for lists. At $\frac{n}{p} = 10^6$, pointer doubling only demonstrates marginally better performance than our forest ruling set algorithm on very shallow forests (such as random GNM(n, 10n) Boruvka forests at $k = 0$) and in shallow RGG2D(n, 10n) Boruvka forests at $k = 10$. Double pointer jumping showed inferior performance in the experiments presented, mainly due to its significantly higher communication volume. It is crucial to recognize that double pointer jumping yields the best results on very small instances, with several hundred vertices per PE, for both one-level and two-level message exchange systems. This efficiency is attributed to the reduction of communication overhead by approximately half.

List ruling set and Euler Tour Our list ruling set performed, as expected, on big lists where $\frac{n}{p} = 10^6$ the best. On smaller instances $\frac{n}{p} = 10^4$ the performance was similar to pointer doubling. Ranking the Euler tour needs for every processor count and instance size almost double the runtime than our forest synchron ruling set on our random tree. This increased runtime can be attributed to the increased instance size of the Euler tour. The Euler tour consists of double the vertices than our initial tree with assigned edge weights.

Forest ruling set In our experiments we observed that this algorithm is versatile and has on all instances a good performance. In Figure 5.10 we summarized all running times from our blocking ruling set algorithm and our best pointer doubling algorithm for $\frac{n}{p} = 10^6$ (on not random graphs we performed local contraction, on trees we use remove duplicate requests, list pointer doubling for lists and forest pointer doubling for all other instances). We can see that forest ruling set had on all instances, except the contracted instances, a pretty similar behaviour. Acually the worst performance was on lists. The reason is, that on lists, this algorithm had the lowest reduction facters. While on lists our instance of rulers should be smaller by factor 20 due to our pick of r, the reduced instances were on other inputs far smaller, particularly for trees, where it exceeded a factor of 100. This discrepancy arises because we need $\leq r$ rulers to start r waves, only on lists we need guaranteed r rulers to start r waves. This makes a wave on a dense tree much more impactful than on a list.

Forest non blocking ruling set Given that the internal threshold is set to ∞, we anticipated that our non-blocking variant would perform comparably to the blocking version. Surprisingly on random lists and on our random trees the non blocking variant, forest non blocking ruling set, performed inferior.

Load Imbalances As illustrated in Figure 5.10, the performance of the forest ruling set algorithm is largely unaffected by the graph's structure. However, instances that exhibit load imbalances—where PEs have significantly varying numbers of incoming neighbours—pose challenges for this algorithm. Notably, caterpillar graphs, with their degrees reaching up to 10^8, represent highly imbalanced instances. The effect of these high degree vertices on the forest ruling set's performance is depicted in Figure 5.9. This figure also demonstrates that our method for removing high-degree vertices remains effective regardless of the vertex degree. In additional experiments not presented here, tree instances that have not undergone random permutation also show substantial imbalances. This is especially true for PE 0, handling vertices 0 to $\frac{n}{p} - 1$, which on average have indegrees significantly higher than 1, even though they are far away of being a high degree vertex. While theory suggests load imbalances could occur in recursive instances, our experiments did not reveal such issues.

Exploiting locality In Figure 5.8, we display the runtime results for our forest ruling set applied to Boruvka trees generated from RGG2D$(n, 10n)$ with $k \in 1, 10$. For $k = 10$, the trees exhibit shallowness and fewer edges between distinct vertices within the same PE. This shallowness likely contributes to the superior performance of pointer doubling over the forest ruling set for $k = 10$, a contrast to the results observed for $k = 1$. Our local contraction method for $k = 1$ reduces the graph size to approximately $\frac{n}{p} = 7 \cdot 10^3$, and for $k = 10$, to $\frac{n}{p} = 170 \cdot 10^3$. Surprisingly, even with the significantly contracted instance for $k = 1$, the forest ruling set still outperforms pointer doubling, however, these differences are marginal.

Summary

Pointer doubling consistently performs well for small values of $\frac{n}{p}$ due to its minimal requirement for global message exchanges. However, for a vertex at depth h, $\log_2(h)$ pointer doublings are necessary, which leads to a communication volume of $O(\log(h))$ for each such vertex, in contrast to the forest ruling set where the communication volume per vertex remains constant. This distinction was evident in our preliminary experiments, where pointer doubling applied to lists required significantly larger processor counts to justify the use of two-level message exchange compared to the forest ruling set. This suggests that our forest ruling set could potentially benefit from employing a three-level message exchange—or higher—in our evaluated scenarios.
For extremely shallow forests, pointer doubling invariably shows strong performance. Yet, the gap between it and our forest ruling set remains narrow, as evidenced in Figure 5.10, where we compare the best outcomes from both our pointer doubling algorithm and our forest ruling set algorithm (applying local contraction to non-random graphs, removing duplicate requests for trees, and using list pointer doubling for lists and forest pointer doubling for all other cases). In random instances, the forest ruling set's performance is largely unaffected by the graph's structure. Nonetheless, load imbalances—previously discussed—do impact our runtimes, and the removal of high-degree vertices becomes essential when present. For instances featuring numerous edges between distinct vertices within the same processing element, our local contraction method proved its worth.

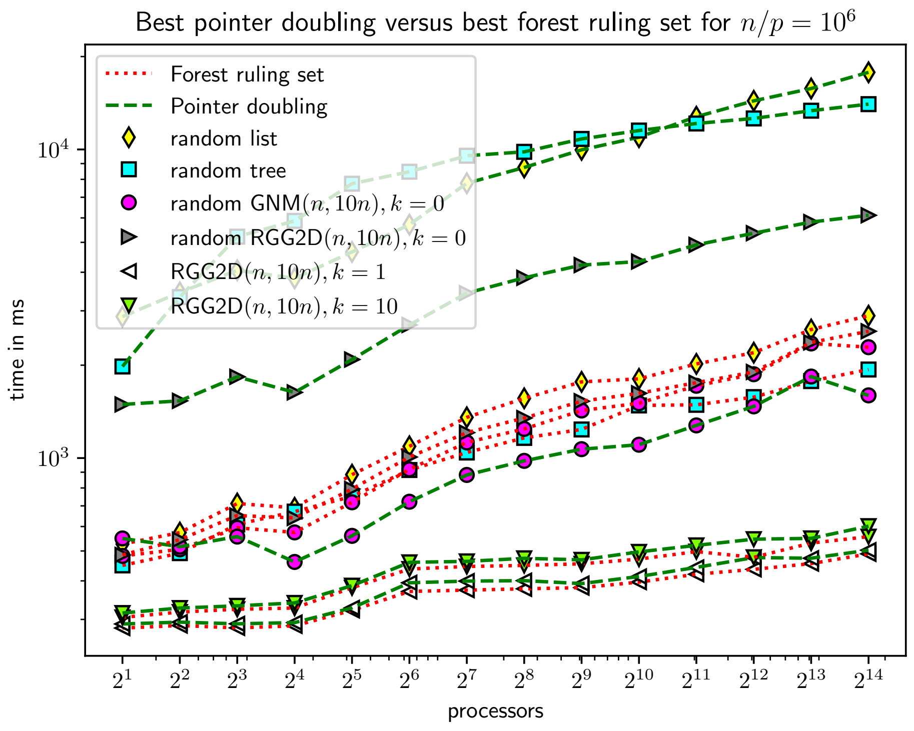


Figure 5.10: Forest ruling set versus pointer doubling, $G(n, m), k$ are Boruvka forests

6 Conclusion

In this thesis we introduced a new ruling set algorithm for tree rooting using the ideas of the sparse ruling set algorithm for list ranking [41]. Our experiments show that the sparse ruling set algorithm for tree rooting is almost twice as fast on trees than ranking the euler tour with the sparse ruling set algorithm optimized for lists and up to 10 times faster than using pointer doubling. This algorithm roots all instances including a list and very shallow forests with a good performance for processor counts up to at least 16384 cores, surpassing the scale of any previously reported experiments in both tree rooting and list ranking to our knowledge. Various techniques were employed to furtherenhance the speed of our algorithms.

One such technique involves exploiting locality by contracting all edges between vertices on the same PE, a strategy also applied in solving other algorithmic graph problems [24, 32]. For instances with a lot of such edges, this method effectively reduces the size of our instance and allows for the reconstruction of all final values from this reduced instance without additional communication. Additionally we implemented strategies from the literature to decrease both the volume and overhead of communication.

We introduced a new approach for removing high degree vertices, which can otherwise poorly influence performance. Our experiments, testing up to degrees of 10^8, showed that vertices did not impact our runtime unlike direct rooting of these instances without the removal of high degree vertices. We refined the sparse ruling set algorithm by Sibeyn [41] with a slight adjustment, ensuring the number of iterations during packet chasing, which is the dominating part of the algorithm, to a fixed number rather than being probabilistically bound, while keeping all important properties of the algorithm.

Our findings reveal that our forest ruling set algorithm can be applied to list ranking with only a slight decrease in performance compared to the sparse ruling set algorithm optimized for lists. We confirmed that pointer doubling performs well for very shallow forests which is a well known fact. However, for pointer doubling to outperform our advanced forest ruling set algorithm, the forests must not only be very shallow but also have a low number of vertices per PE.

Future Work We demonstrated the effectiveness of our forest ruling set algorithm. However, our comparison included different forest ruling set algorithms and the simple pointer doubling method. The field of tree contraction also offers algorithms applicable for tree rooting [18].

We implemented a non blocking version of our algorithm. Using thresholds of ∞ and many terminations result in similar performance than our blocking version. Experiments with different thresholds, differing scheduling of polls and less terminations potentially increase performance.

Furthermore, determining the optimal number of initial rulers is a complex challenge. While Sibeyn proposed a methodology, it did not scale as effectively as our simpler strategy

of consistently selecting 1% of the vertices as rulers. Our experiments indicated that for smaller processor counts, fewer rulers should be chosen, since communication is not strongly penalized in comparison to larger processor counts, increasing the number of rulers showed slightly better results and lists do need more rulers than any other instance. Additionally our results indicate especially instances with small $\frac{n}{p}$ and large p benefit from an three-level message exchange or more even on our evaluation sizes. Optimizing both parameters may enhance runtime.
Another methology that potentially positively affects performance is hybrid parallelization which uses both threads and MPI tasks. Since some threads use a shared memory there is no need for communication between them and on this shared memory the instance sizes increases, which makes exploiting locality more impactful.

Bibliography

[1] Richard J. Anderson and Gary L. Miller. “A Simple Randomized Parallel Algorithm for List-Ranking”. In: *Information Processing Letters* 33.5 (Jan. 1990), pp. 269–273.

[2] Richard J. Anderson and Gary L. Miller. “Deterministic Parallel List Ranking”. In: *Algorithmica* 6 (1991), pp. 859–869.

[3] Lars Arge, Michael Goodrich, and Nodari Sitchinava. “Parallel external memory graph algorithms”. In: May 2010, pp. 1–11. DOI: 10.1109/IPDPS.2010.5470440.

[4] Mikhail Atallah et al. “Constructing Trees in Parallel”. In: ACM. Santa Fe, New Mexico, June 1989, pp. 421–431.

[5] Michael Axtmann and Peter Sanders. *Robust Massively Parallel Sorting*. 2020. arXiv: 1606.08766 [cs.DC].

[6] Dip Sankar Banerjee and Kishore Kothapalli. “Hybrid algorithms for list ranking and graph connected components”. In: *2011 18th International Conference on High Performance Computing*. 2011, pp. 1–10. DOI: 10.1109/HiPC.2011.6152655.

[7] Otakar Borůvka. “O jistém problému minimálním”. In: (1926).

[8] Albert Chan, Frank Dehne, and Ryan Taylor. “CGMGRAPH/CGMLIB: Implementing and Testing CGM Graph Algorithms on PC Clusters and Shared Memory Machines”. In: *The International Journal of High Performance Computing Applications* 19.1 (2005), pp. 81–97. DOI: 10.1177/1094342005051196. eprint: https://doi.org/10.1177/1094342005051196. URL: https://doi.org/10.1177/1094342005051196.

[9] Sun Chung and A. Condon. “Parallel implementation of Bouvka’s minimum spanning tree algorithm”. In: *Proceedings of International Conference on Parallel Processing*. 1996, pp. 302–308. DOI: 10.1109/IPPS.1996.508073.

[10] R Cole and U Vishkin. “Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms”. In: *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*. STOC ’86. Berkeley, California, USA: Association for Computing Machinery, 1986, pp. 206–219. ISBN: 0897911938. DOI: 10.1145/12130.12151. URL: https://doi.org/10.1145/12130.12151.

[11] Richard Cole and Uzi Vishkin. “Deterministic coin tossing with applications to optimal parallel list ranking”. In: *Information and Control* 70.1 (1986), pp. 32–53. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(86)80023-7. URL: https://www.sciencedirect.com/science/article/pii/S0019995886800237.

[12] Richard J. Cole and Uzi Vishkin. "Approximate Parallel Scheduling. Part I: The Basic Technique with Applications to Optimal Parallel List Ranking in Logarithmic Time". In: *SIAM J. Comput.* 17 (1988), pp. 128–142. URL: `https://api.semanticscholar.org/CorpusID:36386698`.

[13] F. Dehne et al. "Efficient Parallel Graph Algorithms For Coarse Grained Multicomputers and BSP". In: vol. 33. Apr. 2006, pp. 390–400. ISBN: 978-3-540-63165-1. DOI: `10.1007/3-540-63165-8_195`.

[14] Frank Dehne and Siang W. Song. "Randomized parallel list ranking for distributed memory multiprocesors". In: *Concurrency and Parallelism, Programming, Networking, and Security.* Ed. by Joxan Jaffar and Roland H. C. Yap. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 1–10. ISBN: 978-3-540-49626-7.

[15] Daniel Funke et al. "Communication-free Massively Distributed Graph Generation". In: *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21 – May 25, 2018.* 2018.

[16] Hillel Gazit and Gary L. Miller. "A parallel algorithm for finding a separator in planar graphs". In: *28th Annual Symposium on Foundations of Computer Science (sfcs 1987).* 1987, pp. 238–248. DOI: `10.1109/SFCS.1987.3`.

[17] M. Habib et al. *Probabilistic Methods for Algorithmic Discrete Mathematics.* Algorithms and Combinatorics. Springer Berlin Heidelberg, 2010. ISBN: 9783642084263. URL: `https://books.google.de/books?id=maMHkgAACAAJ`.

[18] MohammadTaghi Hajiaghayi et al. "Adaptive Massively Parallel Constant-round Tree Contraction". In: *CoRR* abs/2111.01904 (2021). arXiv: `2111.01904`. URL: `https://arxiv.org/abs/2111.01904`.

[19] Benjamin G Jackson, Patrick S Schnable, and Srinivas Aluru. "Parallel short sequence assembly of transcriptomes". In: *BMC bioinformatics* 10 (2009), pp. 1–12.

[20] Joseph JáJá. *An introduction to parallel algorithms.* Addison Wesley Longman Publishing Co., Inc., 1992.

[21] Artur Jeż and Markus Lohrey. "Approximation of smallest linear tree grammar". In: *Information and Computation* 251 (2016), pp. 215–251. ISSN: 0890-5401. DOI: `https://doi.org/10.1016/j.ic.2016.09.007`. URL: `https://www.sciencedirect.com/science/article/pii/S0890540116300785`.

[22] S. Lennart Johnsson and Ching-Tien Ho. "Optimum Broadcasting and Personalized Communication in Hypercubes". In: *IEEE Trans. Comput.* 38.9 (Sept. 1989), pp. 1249–1268. ISSN: 0018-9340. DOI: `10.1109/12.29465`. URL: `https://doi.org/10.1109/12.29465`.

[23] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. "The power of parallel prefix". In: *IEEE Transactions on Computers* C-34.10 (1985), pp. 965–968. DOI: `10.1109/TC.1985.6312202`.

[24] Sebastian Lamm and Peter Sanders. "Communication-efficient Massively Distributed Connected Components". In: *2022 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2022, Lyon, France, May 30 - June 3, 2022*. IEEE, 2022, pp. 302–312. DOI: `10.1109/IPDPS53621.2022.00037`. URL: `https://doi.org/10.1109/IPDPS53621.2022.00037`.

[25] Gary L Miller and John H Reif. "Parallel tree contraction and its application". In: *FOCS*. Vol. 26. 1985, pp. 478–489.

[26] Gary L Miller and John H Reif. "Parallel Tree Contraction Part 1: Fundamentals." In: *Adv. Comput. Res.* 5 (1989), pp. 47–72.

[27] Gary L Miller and John H Reif. "Parallel tree contraction part 2: Further applications". In: *SIAM Journal on Computing* 20.6 (1991), pp. 1128–1147.

[28] Gary L. Miller, Vijaya Ramachandran, and Erich Kaltofen. "Efficient Parallel Evaluation of Straight-Line Code and Arithmetic Circuits". In: *SIAM Journal on Computing* 17.4 (1988), pp. 687–695. DOI: `10.1137/0217044`. eprint: `https://doi.org/10.1137/0217044`. URL: `https://doi.org/10.1137/0217044`.

[29] Noam Nisan and Ziv Bar-Yossef. "Pointer jumping requires concurrent read". In: *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*. STOC '97. El Paso, Texas, USA: Association for Computing Machinery, 1997, pp. 549–558. ISBN: 0897918886. DOI: `10.1145/258533.258648`. URL: `https://doi.org/10.1145/258533.258648`.

[30] Margaret Reid-Miller. "List Ranking and List Scan on the CRAYC90". In: *Journal of Computer and System Sciences* 53.3 (1996), pp. 344–356. ISSN: 0022-0000. DOI: `https://doi.org/10.1006/jcss.1996.0074`. URL: `https://www.sciencedirect.com/science/article/pii/S0022000096900744`.

[31] Margaret Reid-Miller, Gary L Miller, and Francesmary Modugno. "List ranking and parallel tree contraction". In: *Synthesis of Parallel Algorithms* (1993), pp. 115–194.

[32] Peter Sanders and Matthias Schimek. *Engineering Massively Parallel MST Algorithms*. 2023. arXiv: `2302.12199 [cs.DC]`.

[33] Peter Sanders and Tim Niklas Uhl. "Engineering a Distributed-Memory Triangle Counting Algorithm". In: *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, May 2023. DOI: `10.1109/ipdps54959.2023.00076`. URL: `http://dx.doi.org/10.1109/IPDPS54959.2023.00076`.

[34] Peter Sanders et al. *Sequential and Parallel Algorithms and Data Structures*. Springer, 2019.

[35] Julian Shun et al. "Sequential random permutation, list contraction and tree contraction are highly parallel". In: *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '15. San Diego, California: Society for Industrial and Applied Mathematics, 2015, pp. 431–448.

[36] Sibeyn. "One-by-one cleaning for practical parallel list ranking". In: *Algorithmica* 32 (2002), pp. 345–363.

[37] Jop F Sibeyn. “List ranking on interconnection networks”. In: *Euro-Par’96 Parallel Processing: Second International Euro-Par Conference Lyon, France, August 26–29 1996 Proceedings, Volume I 2*. Springer. 1996, pp. 799–808.

[38] Jop F Sibeyn, Frank Guillaume, and Tillmann Seidel. “Practical Parallel List Ranking”. In: *Journal of Parallel and Distributed Computing* 56.2 (1999), pp. 156–180. ISSN: 0743-7315. DOI: https://doi.org/10.1006/jpdc.1998.1508. URL: https://www.sciencedirect.com/science/article/pii/S0743731598915088.

[39] Jop F. Sibeyn. “Better Trade-Offs for Parallel List Ranking”. In: *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA ’97. Newport, Rhode Island, USA: Association for Computing Machinery, 1997, pp. 221–230. ISBN: 0897918908. DOI: 10.1145/258492.258514. URL: https://doi.org/10.1145/258492.258514.

[40] Jop F. Sibeyn. “Minimizing Global Communication in Parallel List Ranking”. In: *Euro-Par 2003 Parallel Processing*. Ed. by Harald Kosch, László Böszörményi, and Hermann Hellwagner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 894–902. ISBN: 978-3-540-45209-6.

[41] Jop F. Sibeyn. “Ultimate Parallel List Ranking?” In: *High Performance Computing – HiPC’99*. Ed. by Prith Banerjee, Viktor K. Prasanna, and Bhabani P. Sinha. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 197–201. ISBN: 978-3-540-46642-0.

[42] Tim Niklas Uhl. “Enabling Scalability Through Asynchronous Messaging and Aggregation”. 2024.

[43] Uzi Vishkin. “Randomized speed-ups in parallel computation”. In: *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*. STOC ’84. New York, NY, USA: Association for Computing Machinery, 1984, pp. 230–239. ISBN: 0897911334. DOI: 10.1145/800057.808686. URL: https://doi.org/10.1145/800057.808686.

[44] James C Wyllie. *The complexity of parallel computations*. Tech. rep. Cornell University, 1979.

7 Appendix

7.1 Algorithms on Random List using one- and two-level Message Exchange

In this section we show the impact of one-level message exchange versus two level message exchange for all tested algorithms on lists, that were not shown in our preliminary experiments. In Figure 7.1 we can see the effects for double pointer jumping, in Figure 7.2 the effects for forest non blocking ruling set and in Figure 7.3 for list ruling set with $\frac{n}{p} \in \{10^4, 10^5, 10^6\}$ and processor counts up to 16384.

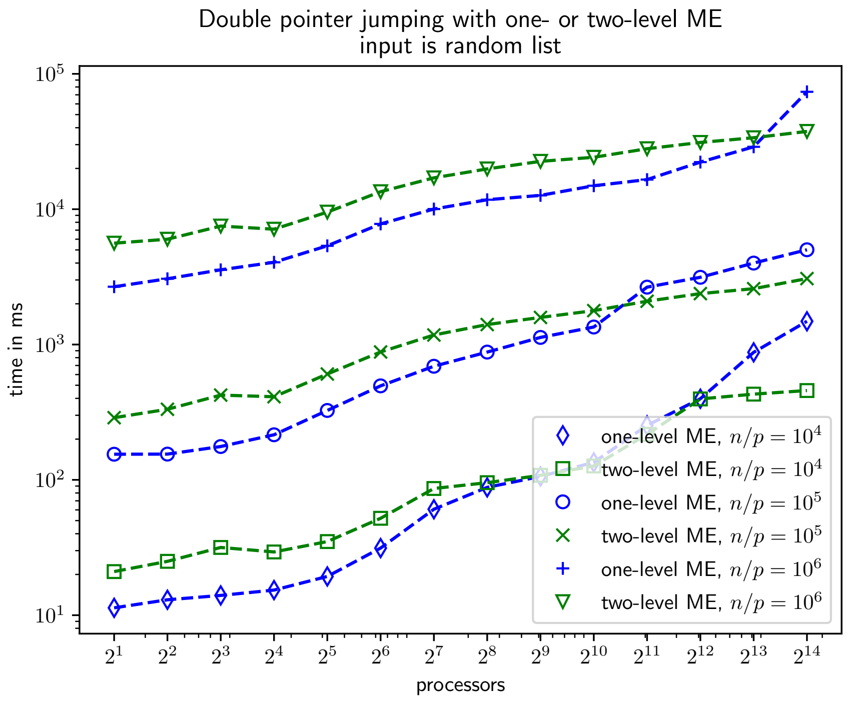


Figure 7.1: Double pointer jumping on random list

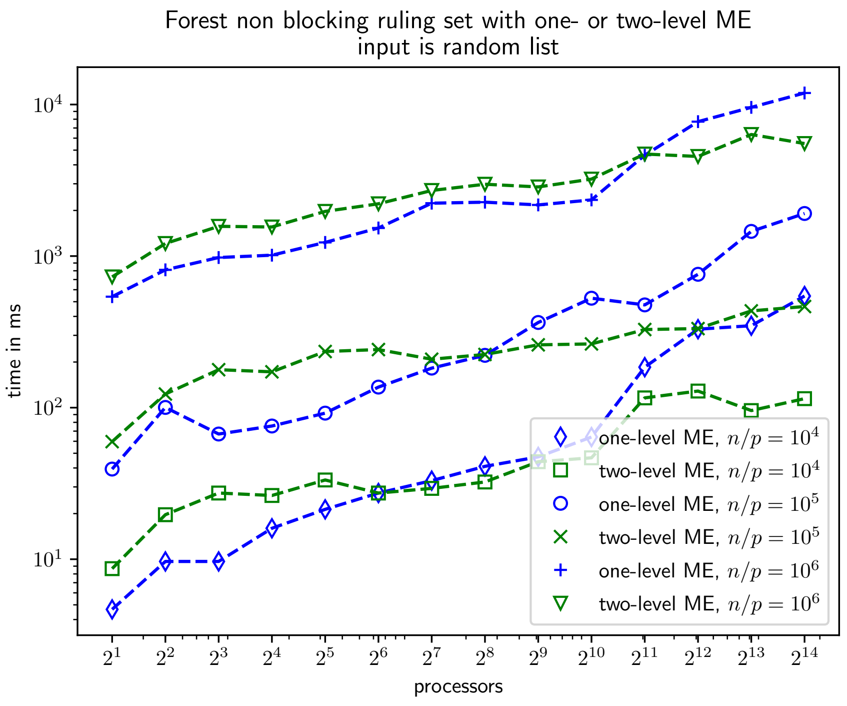


Figure 7.2: Forest non blocking ruling set on random list

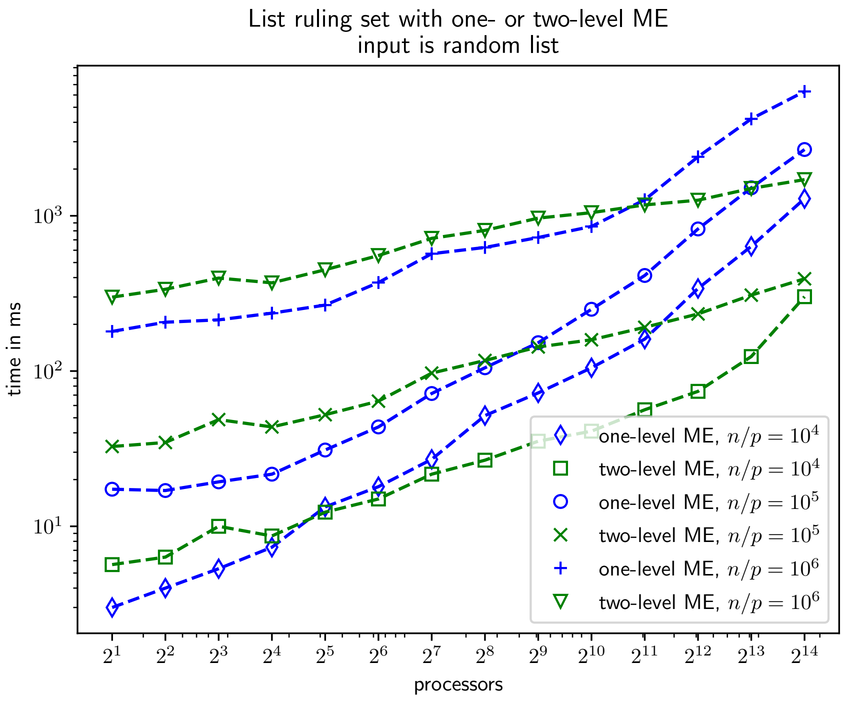


Figure 7.3: List ruling set on random list

7.2 Algorithms on Random Instances with/without remove duplicate requests

In this section we show the effect of remove duplicate requests on different algorithms and different instances. In Figure 7.4 forest pointer doubling is used on a random tree, in Figure 7.5 the effects for random GNM(n, $10n$) Boruvka trees for $k = 0$.

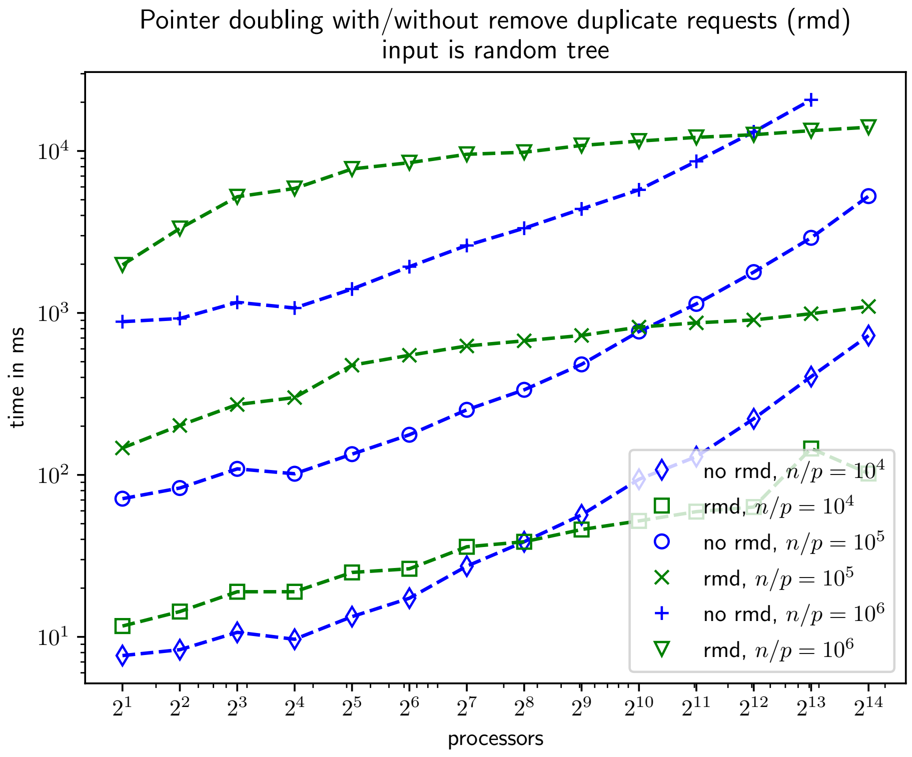
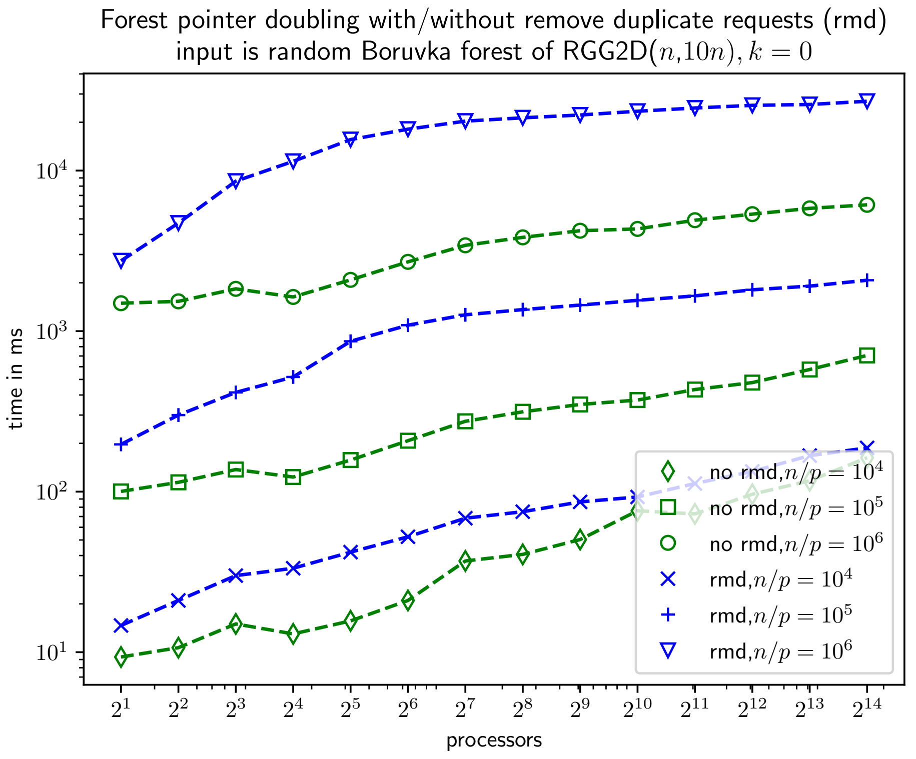


Figure 7.4: Forest pointer doubling on random tree

...



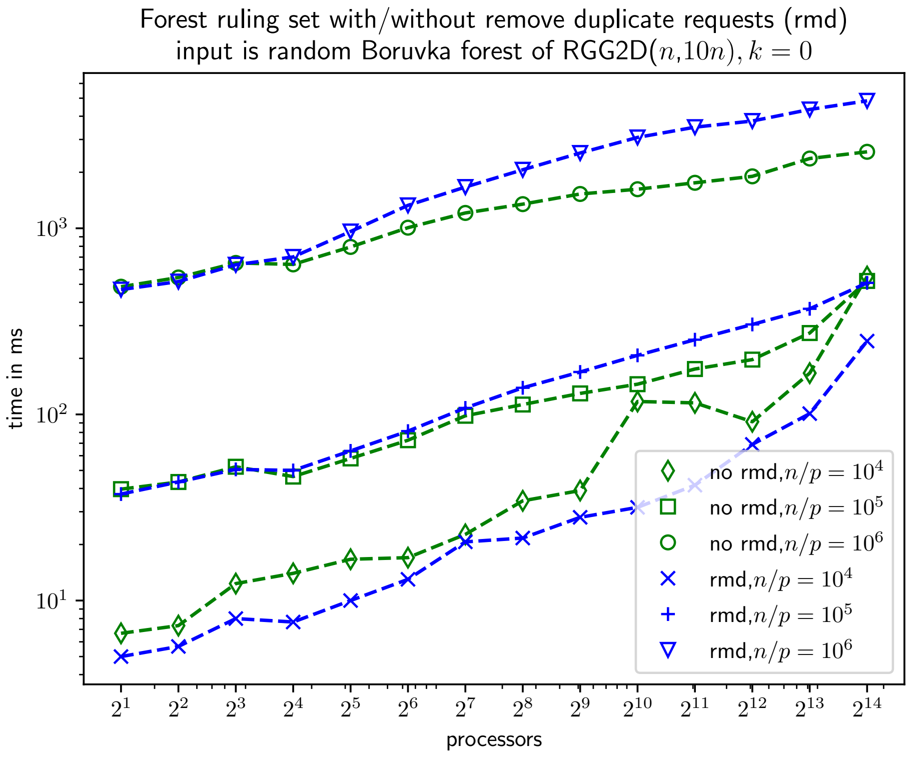


Figure 7.5: Forest pointer doubling and forest ruling set on random RGG2D(n, $10n$) Boruvka forest