Karlsruher Institut für Technologie

Bachelorarbeit

Engineering und Evaluation von verteilten MST Algorithmen für dichte Graphen

David Bumm

01.06.2023 bis 30.10.2023

Betreuer: M.Sc. Matthias Schimek
Prüfer: Prof. Dr. Peter Sanders

Institut für Theoretische Informatik, Algorithm Engineering
Fakultät für Informatik
Karlsruher Institut für Technologie

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Ort, Datum, Name

Zusammenfassung

Das **M**inimum **S**panning **T**ree (MST) Problem sucht für einen ungerichteten und (kanten-)gewichteten Eingabegraphen $G = (V, E)$ nach einem Baum $T = (V, E' \subseteq E)$, welcher alle Knoten aus V verbindet und minimales Gewicht besitzt.
Wir beschäftigen uns mit der Frage, wie MSTs auf Supercomputern mit mehreren tausend Prozessoren effizient berechnet werden können. Solche Supercomputer sind verteilte Systeme, in welchen die Prozessoren nicht über den geteilten Speicher (shared-memory), sondern mittels dedizierter Nachrichten über Hochleistungsnetzwerke kommunizieren. Dadurch ergeben sich deutlich andere Anforderungen an effiziente Algorithmen als in herkömmlichen shared-memory Systemen.
Unsere Arbeit beruht auf den Algorithmen von Dehne et al. [6] und Adler et al. [1]. Diese Algorithmen arbeiten auf einem verteilt vorliegenden Eingabegraphen, wobei jeder Prozessor $|E|/p$ Kanten erhält. Zusätzlich muss die Knotenmenge auf allen p Prozessoren repliziert werden können. Dafür muss die Anzahl an Kanten um den Faktor p größer sein als die Knotenmenge ($|V| \leq |E|/p$).
Auf diesen „dichten“ Graphen hab wir vier verteilte MST Algorithmen implementiert, weitere Verbesserungen für die Praxis vorgenommen und auf über zweitausend Prozessoren evaluiert.

Inhaltsverzeichnis

1 Einführung

Das MST Problem zählt zu den fundamentalsten Graphenproblemen überhaupt und bietet Raum für viele algorithmische Ansätze. Neben der Bedeutung in der Algorithmentheorie ist das MST Problem auch in der Praxis relevant und findet in verschiedenen Bereichen breite Anwendungen. Dazu gehören beispielsweise Clustering [3], Bildsegmentierung [21] und Netzwerkplanung [14].

Abbildung 1 zeigt *links* einen ungerichteten und gewichteten Eingabegraphen, *mittig* einen beliebigen dazugehörigen Spannbaum und *rechts* den Minimalen Spannbaum.

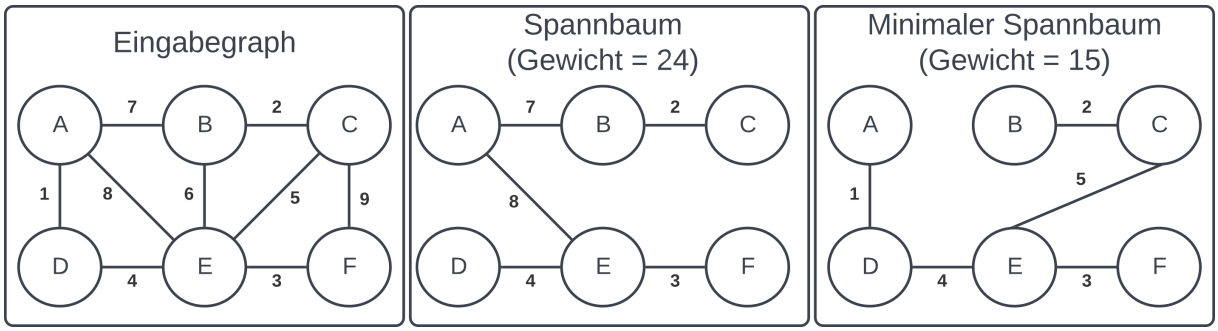

Abbildung 1: Beispiel für einen Graphen mit entsprechendem MST

1.1 Motivation

Graphen geben uns eine universelle und effiziente Möglichkeit verschiedene Sachverhalte für den Computer einfach darzustellen. Für große und komplexe Probleme kann der resultierende Graph enorme Größen annehmen. Um trotzdem noch solche Eingaben schnell genug zu verarbeiten, reichen sequenzielle Algorithmen nicht aus. Durch die parallele Ausführung von mehreren Prozessoren, kann eine wesentliche Beschleunigung erzielt werden. Bei typischen shared-memory Systemen, können alle Prozessoren gemeinsam auf denselben Hauptspeicher lesend oder schreibend zugreifen. Das ermöglicht einerseits eine relativ einfache Implementierung, da jeder Prozessor die gleichen Informationen besitzt. Anderseits bedeutet das für sehr große Eingaben, dass ein einzelner „gigantischer“ Hauptspeicher nötig ist. Dieser muss nicht nur den gesamten Graphen auf einmal speichern können, sondern gegebenenfalls ein Vielfaches davon, da bei der Verarbeitung zusätzliche Datenstrukturen oder Ähnliches notwendig sind.

Weil Hauptspeicher in diesem Ausmaß in der Realität zu teuer und unrealistisch zu realisieren sind, arbeiten wir auf Supercomputern mit verteiltem Speicher. Auf diesen Supercomputern besitzt jeder Prozessor seinen eigenen Speicherbereich, sodass gezielt eine bestimmte Anzahl an Prozessoren für eine Aufgabe ausgewählt werden kann und trotzdem immer gleich viel Speicher für jeden Prozessor zur Verfügung steht. Das ermöglicht also eine parallele Bearbeitung auf sehr großen Eingaben, indem diese auf allen Prozessoren (möglichst) gleichmäßig aufgeteilt werden.

Aber auch für vergleichsweise kleinere Eingaben sind schnelle verteilte Algorithmen sehr relevant. Man kann beispielsweise beobachten, dass viele gut skalierende Simulationen Probleme damit haben, ihre (deutlich kleinere) Datenanalyse effizient auszuwerten. Dies wiederum mindert die eigentlich sehr gute Skalierung. Manche von diesen Analyse Problemen können als Graphenprobleme umformuliert werden, die in wenigen Millisekunden auf den Daten, die von hoch parallelisierten Maschinen entstehen, gelöst werden müssen. Da in diesem Fall die Daten bereits verteilt vorliegen, eignen sich verteilt parallele Algorithmen für diese Auswertung besonders.

1.2 Problemstellung

Im Gegensatz zu Systemen mit geteiltem Speicher, wird bei *distributed-memory* die Eingabe auf alle Prozessoren verteilt. Das hat zur Folge, dass jeder Prozessor nur Sicht auf seine eigenen (lokalen) Daten hat und mit anderen Prozessoren kommunizieren muss, um an die gesamten (globale) Informationen zu gelangen. Diese Kommunikationsvorgänge spielen eine erhebliche Rolle bei verteilt parallelen Algorithmen. Besonders bei Ausführungen mit mehreren tausend Prozessoren kann sowohl die Nachrichtenübertragung als auch notwendige Synchronisationen eine große Auswirkung auf die Laufzeit haben. Die von uns implementierten Algorithmen müssen also diese Aufwände explizit berücksichtigen und ihre Arbeitsweise so gut wie möglich darauf anpassen.
Außerdem ist es nicht immer eindeutig, wie man den Eingabegraph am besten auf alle Prozessoren aufteilt. Zum Beispiel können auf manchen dünn besetzten Graphen ($|V| \gg |E|$), alle Kanten auf jedem Prozessor gespeichert werden, aber nicht unbedingt alle Knoten. Ein verteilter MST Algorithmus muss in diesem Fall unterschiedlich arbeiten als auf anderen Eingaben, weil die lokale Sicht der Daten ein Ausschlaggebender Faktor für das Vorgehen ist.
In dieser Arbeit befassen wir uns nur mit einer bestimmten Familie von Graphen, um dieses Problem einheitlich zu betrachten. Im Gegensatz zum eben genannten Beispiel, arbeiten die für uns relevanten Algorithmen nur auf Graphen, bei denen die Knotenmenge auf allen Prozessoren repliziert werden kann. Das bedeutet jeder Prozessor kennt alle Knoten, aber nur m/p Kanten. Das ist der Fall, wenn die Anzahl an Kanten um den Faktor p größer ist als die Knotenmenge. Da in diesem $|V| \leq |E|/p$ ist, also insbesondere $|V| < |E|$ gilt, nennt man den Graphen dicht. Auf „kleineren“ Eingaben funktionieren die Algorithmen auch mit $|V| \geq |E|$, da auch hier die Knotenmenge in den Speicher von allen Prozessoren passt. Allerdings sind sie in der Praxis auf dichteren Graphen am effizientesten, wie in Kapitel 6 zu sehen ist.

1.3 Übersicht

Sequenzielle MST Berechnungen spielen auch in verteilten Algorithmen eine wichtige Rolle für die (lokale) Ausführung auf einzelnen Prozessoren. In Kapitel 2 klären wir notwendige Grundlagen zu minimalen Spannbäumen und den klassischen sequenziellen MST Algorithmen von Kruskal, Borůvka sowie Jarnik und Prim. Hier stellen wir auch Filter-Kruskal, als eine in der Praxis effizientere Variante von Kruskals Algorithmus vor.
Lokalen Berechnungen sind auch wesentlicher Bestandteil für Merge-Local-MST, einer von vier verteilten parallelen Algorithmen, die wir in Kapitel 4 vorstellen. Merge-Local-MST berechnet schrittweise auf jedem Prozessor einen MST und fügt diese anschließend zusammen. Da sich Borůvkas Algorithmus sehr effektiv parallelisieren lässt, bildet dieser das Herzstück für Borůvka-Allreduce. Dieser Algorithmus nutzt das Vorgehen Borůvkas Algorithmus, wobei in jeder Iteration ein Allreduce Aufruf durchgeführt wird. Ein Allreduce sammelt von allen Prozessoren Daten, führt sie zusammen und sendet sie anschließend wieder zurück.
Um die Vorteile von Borůvka-Allreduce und Merge-Local-MST zu kombinieren haben wir Borůvka-Then-Merge und Borůvka-Mixed-Merge implementiert. Dabei besteht Borůvka-Then-Merge aus einer Reiche an Iterationen von Borůvka-Allreduce gefolgt von Merge-Local-MST. Borůvka-Mixed-Merge hingegen, führt abwechselnd Borůvka-Allreduce und Merge-Local-MST aus.
Theoretische Laufzeitschranken können in der Praxis nicht immer vorhersagen welcher Algorithmus am besten funktioniert, da dies stark von der Implementierung und den auftretenden Eingaben abhängt. Außerdem stimmen die theoretischen Modelle nicht mit den real-existierenden Maschinen überein. Während z.B in theoretischen Analysen die Anzahl an Kanten, Knoten und Prozessoren gegen unendlich gehen, sind die Prozessoren in der Praxis nicht so zahlreich

verfügbar. Auch Effekte in der Praxis wie Cache Effizienz werden für theoretische Algorithmen selten in Betracht gezogen, obwohl diese massive Unterschiede erzielen können. Daher werden in Kapitel 5 die wesentlichen Faktoren unserer Implementierung genannt und besprochen auf welchen Eingaben wir unsere Algorithmen ausführen.
Eine praktische Evaluation zu den verwendeten Algorithmen, mit bis zu 2048 Prozessoren, ist in Kapitel 6 aufgeführt. Hierbei vergleichen wir die jeweiligen Laufzeiten auf drei verschiedenen Graphtypen und prüfen mit welchen Parametern die Algorithmen am effizientesten sind. Zusätzlich zeigen wir die Auswirkung von lokalen MST Berechnungen und inwiefern das Überlagern von Nachrichten die Laufzeit einer Iteration von Borůvka-Allreduce verbessern kann.
In Kapitel 7 besprechen wir schließlich die gesammelten Ergebnisse und erörtern weitere Verbesserungen für mögliche weitere Arbeiten.

2 Theoretische Grundlagen

2.1 Minimaler Spannbaum (MST)

In der Graphentheorie versteht man unter einem Graph $G = (V, E)$ eine Menge von Knoten (**V**ertices) und Kanten (**E**dges), die eine Verbindung zwischen zwei Knoten darstellen. Insbesondere gibt $|V|$ oder auch n die Anzahl an Knoten an und $|E|$ bzw. m die Anzahl an Kanten. Zusätzlich können Kanten noch ein Gewicht (oder Kosten) enthalten. Im Allgemeinen sind Kanten bei dem MST Problem ungerichtet, das bedeutet, dass jede Kante $\{s, t\}$ sowohl von Knoten s nach t, als auch von t nach s durchlaufen werden kann. Wir gehen im Folgenden immer von ungerichteten Graphen als Eingabe aus.

Ein minimaler Spannbaum (engl. **M**inimum **S**panning **T**ree oder kurz MST) ist die Teilmenge eines Graphen, bei dem alle Knoten miteinander verbunden sind und die Summe aller Kantengewichte minimal ist. Besteht der Eingabegraph aus einer einzigen Komponente, so ist das Ergebnis einer MST Berechnung ein einzelner Baum. Also ein Graph mit genau $n - 1$ Kanten, bei dem alle Knoten über einen eindeutigen Pfad aus Kanten verbunden sind. Ist der Graph nicht zusammenhängend, also besteht er aus mehr als einer Komponente, dann kann iterativ auf jeder Komponente eine MST Berechnung durchgeführt werden. Das Ergebnis ist in diesem Fall ein minimaler Spannwald (engl. **M**inimum **S**panning **F**orest). Um also immer einen MST als Ergebnis zu erhalten, können wie ohne Beschränkung der Allgemeinheit annehmen, dass der Eingabegraph aus genau einer Komponente besteht.
Der MST eines Graphen ist nicht immer eindeutig. Hat beispielsweise jede Kante in einem Graph Gewicht 1, so kann es eine Vielzahl an unterschiedlichen MSTs geben, wenn es mehrere Möglichkeiten gibt die Knoten miteinander zu verbinden. Der resultierende MST ist eindeutig, wenn jedes Kantengewicht im Graphen höchstens ein Mal vor kommt [19].

2.2 Schnitt- und Kreiseigenschaft

Die Schnitteigenschaft, zusammen mit der Kreiseigenschaft, bilden wichtige Merkmale für die Auswahl von MST Kanten in einem Graph. Anhand dieser Eigenschaften konnte die Korrektheit von verschiedenen MST Algorithmen bewiesen werden. In Kapitel 4.1.3 wird die Kreiseigenschaft benutzt, um die Korrektheit eines verteilten Algorithmus zu begründen.

Theorem 1. *Schnitteigenschaft*
Sei $S_1, S_2 \subset$ V mit $S_1 \cup S_2 = V$ und $S_1 \cap S_2 = \emptyset$. Die (Schnitt-) Menge an Kanten, die zwischen S_1 und S_2 verlaufen, nennen wir E_s. In diesem Fall besagt die Schnitteigenschaft, dass die Kante $e \in E_s$ mit dem geringsten Gewicht aus E_s immer Teil des MSTs von G ist.

In Abbildung 2 sind die Teilmengen $S_1 = \{A, D, E\}$ und $S_2 = \{B, C, F\}$, sowie die zugehörige Schnittmenge $E_s = \{(A, B, 7), (B, E, 6), (C, E, 5)(E, F, 3)\}$ zu sehen. Nach der Schnitteigenschaft gehört die Kante $e = (E, F, 3)$ auf jeden Fall zum MST von G, da e minimales Gewicht in E_s hat.

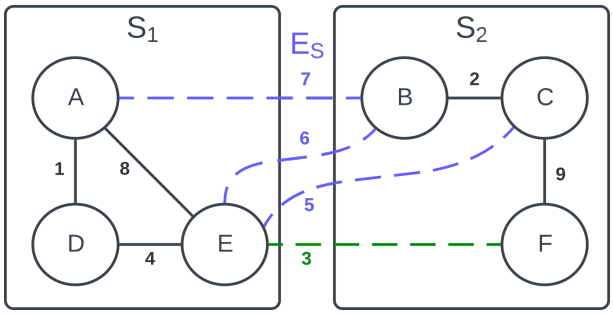

Abbildung 2: Beispiel für die Schnitteigenschaft

Theorem 2. *Kreiseigenschaft*
Sei $K \subseteq E$ ein beliebiger Kreis des Graphen G. Dann besagt die Kreiseigenschaft, dass die Kante $e \in K$, mit dem höchsten Gewicht aus K, nicht im MST von G enthalten ist.

Wir sehen in Abbildung 3 einen Graphen mit eingefärbten Kreis $K = \{(A, B, 7), (B, C, 2), (C, E, 5), (D, E, 4), (A, D, 1)\}$. Wir wissen also nun, dass die Kante $(A, B, 7)$ nicht zum MST gehören kann, da sie die Kante mit dem höchsten Gewicht in K ist.

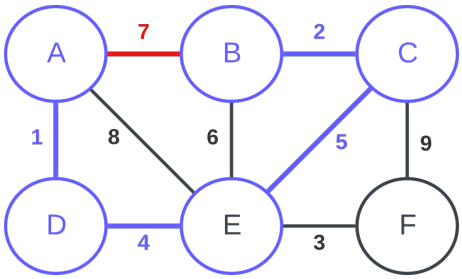

Abbildung 3: Beispiel für die Kreiseigenschaft

Einen Beweis für die Kreis- und Schnitteigenschaft findet man beispielsweise im Buch von Sanders et al. [19].

2.3 Sequenzielle Algorithmen

Auch wenn wir uns im Folgenden nur mit verteilt parallelen Algorithmen auseinandersetzen, spielen auch lokale MST Berechnungen bei diesen eine wichtige Rolle.
Die Algorithmen von Kruskal [13], Jarník und Prim [16] sowie Borůvka [4] bilden die bekanntesten sequenziellen MST Algorithmen.
Borůvkas Algorithmus wurde in 1926 veröffentlicht und ist damit der älteste MST Algorithmus. Dieser fügt in jeder Iteration die leichtesten inzidenten Kanten aller Knoten zum MST hinzu und kontrahiert anschließend den Graph, bis nur noch ein Knoten übrig ist. Die Laufzeit liegt dabei in $O(n \log(m))$. Ein ausschlaggebender Vorteil von Borůvkas Algorithmus ist, dass man ihn sehr einfach und effektiv parallelisieren kann.

Der Jarník-Prim Algorithmus wurde 1930 von Jarník entwickelt und 1957 von Prim wieder entdeckt. Er nutzt die Schnitteigenschaft aus, um den MST zu berechnen. So startet der Algorithmus mit einem beliebigen Knoten von G und fügt in jeder Iteration, die leichteste Schnittkante zwischen den bisher hinzugefügten Knoten und dem restlichen Graphen hinzu. So werden nach der Schnitteigenschaft in jedem Schritt eine MST-Kante und ein Knoten hinzugefügt,

solange bis alle Knoten hinzugefügt wurden. Unter Verwendungen einer geeigneten Prioritätswarteschlange (z.B Fibonacci Heaps) liegt die Laufzeit von Jarník und Prims Algorithmus in $O(m + n \log n)$.

1956 hat Kruskal seinen Algorithmus entwickelt, welcher nach und nach die leichteste Kante des Graphen zum MST hinzufügt, sofern sie keinen Kreis im Graphen schließt. Mit einer Laufzeit von $O(m \log m)$ ist Kruskal auf ausreichend dichten Graphen asymptotisch ineffizienter als Jarník-Prim oder Borůvka. Allerdings zeigen Osipov et. al [15], dass (Filter-)Kruskal in der Praxis für viele Graphinstanzen effizienter ist als Jarník-Prim. Zusätzlich ist Kruskal einfacher zu Implementieren. Aus diesen beiden Gründen nutzen wir im Folgenden ausschließlich Kruskal (bzw. Filter-Kruskal) für lokale MST Berechnungen.

Borůvkas Algorithmus bildet in Kapitel 4 eine wichtige Basis für die verteilten Algorithmen, wobei Kruskal für die lokalen MST Berechnungen am effizientesten ist. Da uns Jarník-Prim keinen ausschlaggebenden Vorteil bietet, betrachten wir diesen nicht weiter und gehen im Folgenden genauer auf das Vorgehen von Borůvkas und Kruskals Algorithmus ein.

2.3.1 Borůvkas Algorithmus

Zu Beginn jeder Iteration von Borůvkas Algorithmus wird für jeden Knoten, die inzidente Kante mit dem geringsten Gewicht zum MST hinzugefügt. Als nächstes werden alle Knoten, die über diese leichtesten inzidenten Kanten verbunden sind, kontrahiert und die noch übrigen Kanten entsprechend umbenannt.
Bei der Umbenennung entstehen dann gegebenenfalls mehrmals die gleichen (parallele) Kanten, von denen man alle, bis auf die leichteste, entfernen kann. Das Entfernen der parallelen Kanten ist hierbei für die Korrektheit des Algorithmus nicht zwangsläufig nötig, kann aber in der Praxis die folgenden Iterationen beschleunigen.
Dieses Vorgehen wiederholen wir so oft, bis nur noch ein Graph mit einem Knoten übrig ist.
Da bei einer Kontraktion des Graphen, immer mindestens zwei Knoten entlang einer Kante kontrahiert werden, verringert sich in jeder Iteration die Anzahl an Knoten um mindestens die Hälfte. Damit sind höchstens $\log(n)$ Iterationen notwendig, um den MST zu berechnen. In jeder Iteration werden alle m Kanten einmal durchlaufen um die leichtesten inzidenten Kanten zu finde und ein weiteres Mal um diese umzubenennen. Die Kontraktion des Graphen ist in $O(n) \subseteq O(m)$ möglich. Die gesamte Laufzeit von Borůvkas sequenziellen Algorithmus liegt damit in $O(m \log(n))$.
Aufgrund der Umbenennung der Kanten muss man allerdings zum Schluss diese wieder in ihre Ursprungsform bringen, sofern man nicht nur am Gewicht interessiert ist.

Der Pseudocode zu einer parallelen Variante von Borůvka findet sich bei Algorithmus 7. In Abbildung 4 ist ein schematischer Durchlauf zu sehen. Hier werden in der ersten Iteration von Borůvka die Kanten $(A, D, 1)$, $(B, C, 2)$ und $(E, F, 3)$ zum MST hinzugefügt und die jeweiligen Knoten zusammengefasst. Damit sind im nächsten Schritt nur noch 3 Knoten übrig, wobei hier die Kanten $(AD, EF, 4)$ und $(BC, EF, 5)$ zum MST gehören. Anschließend können alle Knoten zu einem kontrahiert werden, der Algorithmus terminiert und liefert den MST mit Gewicht 15.

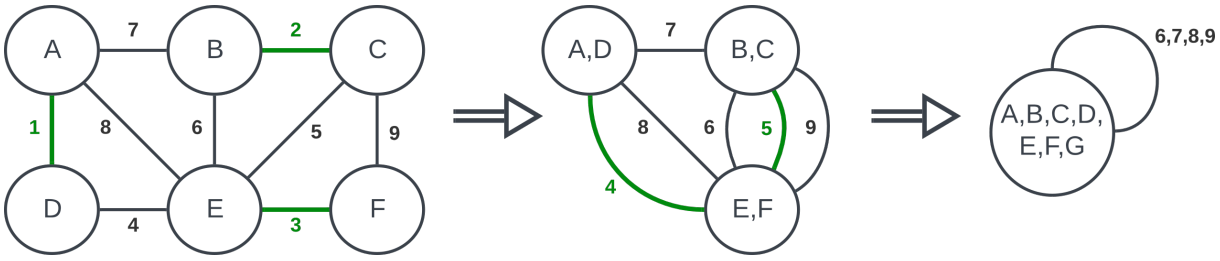

Abbildung 4: Beispieldurchlauf von Borůvkas Algorithmus

2.3.2 Kruskals Algorithmus

Als erstes sortiert Kruskals Algorithmus alle Kanten aufsteigend nach Kantengewicht. Anschließend wird für jede Kante $\{s, t\}$ überprüft, ob die Knoten s und t bereits über andere Kanten verbunden sind. Ist das der Fall, so schließt diese Kante einen Kreis, da s und t bereits über einen anderen Pfad verbunden sind. Wegen der Sortierung ist diese Kante im Kreis diejenige mit dem größten Gewicht und kann nach der Kreiseigenschaft verworfen werden. Andernfalls wird die Kante zum MST hinzugefügt.
Um effizient zu überprüfen ob s und t bereits in derselben Komponente liegen, wird häufig die UnionFind Datenstruktur verwendet. Diese Datenstruktur verfügt über die Operation *union(s,t)* und *find(s)*, welche zwei Knoten zur selben Zusammenhangskomponente hinzufügt (union) oder überprüft (find) in welcher Zusammenhangskomponente ein Knoten bereits enthalten ist. Eine union bzw. find Operation benötigt $O(\log(n))$ Operationen, so dass der Kruskal Algorithmus $O(m(\log(m))$ Operation zum Sortieren, gefolgt von $O(n)$ union und $O(m)$ find Operationen benötigt. Insgesamt hat Kruskals Algorithmus eine amortisierte Laufzeit von $O(m \cdot \alpha(m, n))$, wobei α die inverse Ackermann-Funktion ist. Der folgende Pseudocode von Algorithmus 1 zeigt das Vorgehen von Kruskal.

Algorithmus 1 Kruskal(V, E, UF: UnionFind): Kantenliste

```
1: MST: Kantenliste
2: E_sorted ← sortiere E aufsteigend nach Kantengewicht
3: for each e = (s, t, w) ∈ E_sorted do
4:     if UF.find(e.s) ≠ UF.find(e.t) then
5:         MST ← MST ∪{e}
6:         UF.union(e.s, e.t)
7:     end if
8: end for
9: return MST
```

2.4 Filter-Kruskal

Der Filter-Kruskal Algorithmus [15] ist eine Abwandlung von Kruskals Algorithmus und dient ebenso zur sequenziellen Berechnung von MSTs. Die Idee ist es den Hauptaufwand von Kruskal, dem Sortieren der Kanten, in der Praxis zu verbessern.
Hierfür wird ein Ansatz ähnlich zum *quick-sort Algorithmus* [9] verwendet:
Zunächst wird eine zufällige Kante als Pivot ausgewählt und die zu sortierende Kantenmenge in zwei Mengen $E_{\leq}$ und $E_{>}$ aufgeteilt. Wobei in $E_{>}$ alle Kanten mit einem größeren Gewicht, als die Pivotkante enthalten sind und in $E_{\leq}$ die übrigen Kanten. Ist $E_{\leq}$ *klein genug*, so wird auf dieser Menge der normale Kruskal ausgeführt, ansonsten wird erneut eine $E_{\leq}$ und $E_{>}$ Menge gebildet. Auf $E_{>}$ führen wir anschließend einen Filter-Schritt durch. Dabei wird mithilfe der UnionFind Datenstruktur überprüft, ob Kanten aus dieser Menge bereits nicht mehr benötigt werden. Schließlich wird diese Menge (sofern nötig) erneut in $E_{\leq}$ und $E_{>}$ aufgeteilt. Der Wert für die Grenze, die angibt ab wann Kruskals Algorithmus ausgeführt wird, kann variieren, sollte aber in $O(n)$ liegen [15]. Der Pseudocode von Algorithmus 2 zeigt dieses Vorgehen im Detail.

Algorithmus 2 Filter-Kruskal(V, E, UF: UnionFind, Grenze: int): Kantenliste

```
1: if |E| < Grenze then
2:    return Kruskal(V, E, UF)
3: end if
4: pivotGewicht ← Gewicht einer zufälligen Kante e ∈ E
5: E_≤ ← e ∈ E mit e.w ≤ pivotGewicht
6: E_> ← e ∈ E mit e.w > pivotGewicht
7: E_≤ ← Filter-Kruskal(V, E_≤, UF, Grenze)
8: E_> ← Filter(E_>, UF)
9: E_> ← Filter-Kruskal(V, E_>, UF, Grenze)
10: return E_≤ ∪ E_>
```

Algorithmus 3 Filter(E, UF: UnionFind): Kantenliste

```
1: return e ∈ E mit (UF.find(e.s) != UF.find(e.t))
```

Die erhöhte Effizienz in der Praxis stammt daher, dass in vielen MSTs nur *leichte* Kanten enthalten sind. In diesem Fall werden „schweren“ Kanten nicht mit sortiert, sondern über den Filter Aufruf vorher entfernt. Die asymptotische Laufzeit ist also identisch zu Kruskal, aber auf zufälligen Graphen liegt die erwartete Laufzeit in $O(m + n \log n \log \frac{m}{n})$ [15].

2.5 Parallele Modelle

Es gibt verschiedene theoretische (Berechnungs-) Modelle, um die Operationen (und Komplexität) von parallelen Algorithmen zu beschreiben. Da wir uns mit verteilten Algorithmen beschäftigen, sind Modelle, die für geteilten Speicher entwickelt wurden, wie PRAM (**p**arallel **R**andom **A**ccess **M**achines) für uns nicht ausreichend. Wir wollen die Komplexität von Nachrichtenübertragungen explizit mitberücksichtigen und betrachten daher im Folgenden das BSP und α/β Modell.

2.5.1 Das Bulk Synchronous Parallel Modell

1990 wurde das BSP Modell von Valiant et al. [20] entwickelt, um die Kommunikation und Synchronisation von verteilten Algorithmen berücksichtigen zu können. In diesem Modell gibt es eine Maschine mit p Prozessoren, wobei jeder Prozessor seinen eigenen Speicherbereich besitzt. Zusätzlich gibt es einen Router mit Kommunikationsdurchatz g. Eine Synchronisation kann alle L Zeitschritte stattfinden. Ein BSP Algorithmus besteht aus einer Reihe von sogenannten *Supersteps*, welche von Barrieren-Synchronisationen getrennt sind. Insofern können Nachrichten, die in Superstep i gesendet werden, erst im folgenden Superstep i+1 für die Berechnung verwendet werden. Jeder Superstep i hat einen Aufwand von $w_i + gh_i + L$. Dabei ist im i-ten Superstep w_i die größte Anzahl an lokalen Operationen, und h_i an gesendeten oder empfangenen Nachrichten, aller p Prozessoren. Die gesamten Kosten eines Algorithmus sind als $W + gH + LT$ angegeben, wobei $W = \sum_i w_i$, $H = \sum_i h_i$ und T die Anzahl an Supersteps ist.
In diesem Modell haben Dehne et al. [6] und Adler et al. [1] die Komplexität der uns zugrunde legenden Algorithmen angegeben. Wir werden die Algorithmen zusätzlich im fein-körnigeren Alpha-Beta Modell evaluieren, um einen zusätzlichen Einblick in die jeweiligen Laufzeiten zu erhalten.

2.5.2 Das α/β Modell

Im Gegensatz zum BSP Modell werden im α/β (Alpha/Beta) Modell alle Nachrichtenübertragung zwischen *processing elements* (PEs) einzeln Berücksichtigt [19]. Das ermöglicht eine detailliertere Analyse für die Laufzeit und Komplexität verteilter Algorithmen. Die Übertragung einer Nachricht der Länge ℓ zwischen zwei PEs benötigt $\alpha + \beta\ell$ Zeit. Wobei α die Zeit für die Initialisierung (Startup) der Übertragung ist und β die Zeit zum Senden einer Dateneinheit angibt.

2.6 Kollektive Operationen

Kollektive Operationen sind bestimme Aufrufe, die für die Kommunikation zwischen PEs verwendet werden. Sie ermöglichen einen einfachen und effizienten Umgang für den Nachrichtenaustausch von mehreren PEs. Da kollektive Operationen für die Algorithmen ein wichtiger Bestandteil sind, geben wir nun eine kurze Übersicht über deren Funktionsweise und Komplexität.

2.6.1 Boradcast

Der *Boradcast* Aufruf wird verwendet, wenn ein PE eine Nachricht der Länge ℓ mit alle anderen PEs teilen will. Die untere Schranke für die Laufzeit ist dabei $\alpha + \log(p) + \beta\ell$ [18]. Die Laufzeit ergibt sich logarithmisch in der Anzahl p an PEs, da sich die Anzahl an PEs, die die Nachricht weitersenden können, in jedem Schritt verdoppelt.

2.6.2 (All-)Reduce

Angenommen jeder PE i besitzt einen Vektor M_i der Länge ℓ von Typ T. Sei zusätzlich $\bigoplus$ eine assoziative binäre Operation auf dem Datentyp T. Der *Reduce* Aufruf führt alle Daten auf einem PE zusammen und berechnet $M := \bigoplus_{i=0}^{p-1} M_i$. Ein **All**reduce hingegen funktioniert wie ein Reduce gefoltg von einem Boradcast Aufruf. Das bedeutet nach dem Allreduce Aufruf liegt M anschließend nicht nur auf einem PE, sondern auf allen PE's vor. Dabei liegt die Laufzeit von einer Reduce als auch Allreduce Operation in $O(\alpha \log(p) + \beta\ell)$ [18].

3 Verwandte Arbeiten

Da das MST Problem eines der grundlegendsten Graphenprobleme ist, wird sich schon lange damit beschäftigt, effiziente Algorithmen für dieses Problem zu finden. Seitdem die bereits erwähnten Algorithmen von Kruskal [13], Jarnik und Prim [16] sowie Borůvka [4] entwickelt wurden, gibt es viele Arbeiten, die darauf aufbauen.

So hat zum Beispiel in 1995 Karger et al. [11] den KKT-Algorithmus entworfen. Dieser kombiniert Borůvka zusammen mit randomisierten Stichproben und effektivem Kanten filtern, sodass dieser eine erwartete lineare Laufzeit erreicht hat. Ansonsten hat Osipov et al. [15] in 2009 den Filter-Kruskal Algorithmus entwickelt. Dieser Algorithmus verbessert in der Praxis den hohen Aufwand des Kanten Sortierens bei Kruskal, weshalb wir ihn auch für unsere lokale Berechnungen weiter verwenden.

Natürlich gibt es mittlerweile auch eine Vielzahl an parallelen MST Algorithmen. In 2021 hat Dhulipala [7] eine shared-memory Varainte von Borůvkas Algorithmus entwickelt. Wobei erst letztes Jahr Esfahani et al. [12] einen *structure aware* Algorithmus veröffentlicht hat, der diesen noch übertrifft.

Neben diesen shared-memory Varianten gibt es noch weitere parallele Algorithmen, die wie wir auf geteiltem Speicher arbeiten. So haben Chung et al. [5] in 1996 erstmalige eine verteilte Versions von Borůvkas Algorithmus entwickelt. Nur zwei Jahre später veröffentlichten Dehne und Götz [6] ihre Algorithmen, auf denen wir mit unserer Arbeit aufbauen. Im selben Jahr veröffentlichten auch Adler et al. [1] eine ähnliche Reihe an Kommunikationsoptimalen Algorithmen.

Im Gegensatz zu diesen Algorithmen, die für dichte Graphen konstruiert, wurden haben Sanders und Schimek [17] kürzlich einen *generel purpose* Algorithmus entwickelt. Dieser beruht auch auf Borůvkas Algorithmus und wurde auf über 65.000 Prozessoren evaluiert.

4 Algorithmen

In diesem Kapitel betrachten wir die Funktionsweise, Eigenschaften und Komplexität der von uns implementierten verteilt parallelen Algorithmen [1, 6]. Wir gehen ab sofort davon aus, dass der Graph bereits verteilt auf den PEs vorliegt, welche von 0 bis $p-1$ durchnummeriert sind. Damit ist die lokale Kantenmenge $E_{\ell i}$ je nach PE i unterschiedlich ($\bigcap_{i=0}^{p-1} E_{\ell i} = \emptyset$) und es gilt für die globale Kantenmenge $E = \bigcup_{i=0}^{p-1} E_{\ell i}$. Am Ende jedes Algorithmus liegt der globale MST vollständig auf dem PE mit Identifikator (ID) 0 vor.

4.1 Merge-Local-MST

Die Idee des Merge-Local-MST Algorithmus ist es schrittweise auf jedem PE einen lokalen MST zu berechnen und diesen anschließend mit dem MST von anderen PEs zusammen zu fassen. Dieser Vorgang wird so lange wiederholt, bis noch genau ein PE mit dem globalen MST übrig ist.

4.1.1 Funktionsweise

In jeder Iteration i des Merge-Local-MST Algorithmus berechnen alle noch aktiven PEs auf ihren lokalen Kanten einen MST (z.b mittels Kruskal). Anschließend empfängt jeder D-te aktive PE die MSTs von den folgenden aktiven $D-1$ PEs. Nachdem ein PE seinen MST versendet hat, ist dieser anschließend nicht mehr aktiv und wird nicht weiter benötigt. Die restlichen PEs können aus den empfangenen Daten nun erneut einen lokalen MST berechnen. Dieses Vorgehen wird so lange wiederholt bis nur noch ein Prozessor aktiv ist. Dieser berechnet nun ein letztes Mal einen MST und gibt diesen zurück.

Wie viele MSTs in jedem Schritt zusammengefasst werden bezeichnen wir als *treefacor* oder auch D. Hierbei kann D (kleiner als p) beliebig gewählt werden. So kann es vorkommen, dass in einer Iteration ein PE weniger als $D-1$ MSTs empfängt, falls $\log_D(p) \notin \mathbb{N}$ ist.
Insgesamt sind also $\lfloor \log_D(\mathrm{p}) \rfloor$ Iterationen nötig. Eine Iteration dieses Verfahrens ist in Algorithmus 4 geschildert und das vollständige Vorgehen in Algorithmus 5.

In Kapitel 6.3 arbeiten wir heraus, welche Werte sich für D in der Praxis eignen.
Abbildung 5 zeigt, wie das Zusammenfügen der MSTs für 8 PEs mit einem treefactor von 2 funktioniert. Zu Beginn sind alle acht PEs aktiv und in jeder Iteration sendet ein PE seinen MST weiter, sodass immer $D = 2$ MSTs verschmolzen werden. So sind z.B nach der ersten Iteration nur noch die PEs 0, 2, 4 und 6 aktiv und insgesamt nach $\log_2(8) = 3$ Iterationen liegt der globale MST auf PE 0 vor.

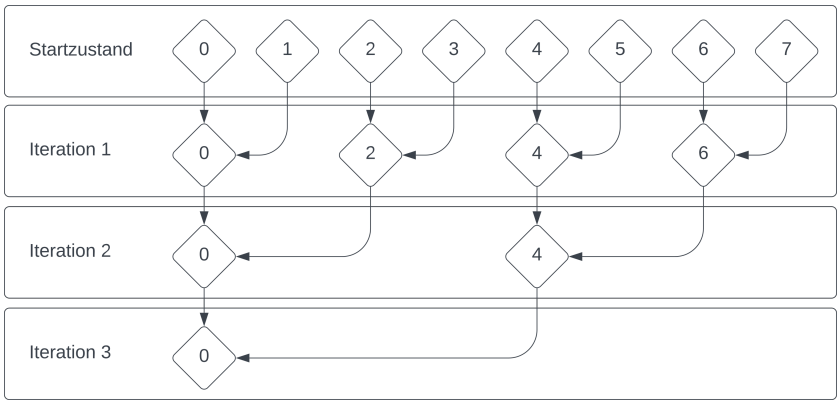

Abbildung 5: Übersicht zur Verschmelzung von MSTs für $D = 2$

Algorithmus 4 MergeStep(V, MST: Kantenliste, D: int)

```
1: andereMSTs: Liste an Kantenlinsten
2: if (PE ist empfänger) then
3:     andereMSTs ← empfange (biszu) D MST
4: else
5:     sende MST zu Empfänger PE
6:     return  //Dieser PE ist nun inaktiv
7: end if
8: for Kanten ∈ andereMSTs do
9:     MST ← MST ∪ Kanten
10: end for
11: MST ← localMST(MST, V)
```

Algorithmus 5 Merge-Local-MST(V, E, D: int): Kantenliste

```
1: MST ← localMST(E, V)
2: for i ← 0 bis ⌊log_D(p)⌋ do
3:     MergeStep(V, MST, D)
4: end for
5: return  MST
```

4.1.2 Eigenschaften und Komplexität

Sei m' die maximale Anzahl an lokalen Kanten eines PEs und T_{seq} die Laufzeit für eine lokale MST Berechnung. Dann liegen im α/β Modell die Kosten für das Senden eines MSTs $\alpha + \beta\ell$, wobei ℓ die Länge des zu sendenden MSTs ist. Angenommen $m \geq n$, so ergibt sich die Laufzeit einer Iteration durch eine lokale MST Berechnung, gefolgt von D Nachrichtenübertragungen für das Empfangen der MSTs mit einer jeweiligen maximalen Länge von $n - 1$. In diesem Fall liegt die Laufzeit für den ersten MergeStep in $O(T_{seq}(n, m') + D(\alpha + \beta n))$. Im Anschluss hält jeder aktive PE höchstens n-1 Kanten. Daraus ergibt sich die Gesamte Laufzeit von Merge-Local-Mst in

$$O(T_{seq}(n, m') + \log_D(p)(T_{seq}(n, Dn) + D(\alpha + \beta n))).$$

Sei nach wie vor $m \geq n$, dann können wir ohne Beschränkung der Allgemeinheit annehmen, dass jeder PE nach Iteration i einen MST mit höchstens n-1 Kanten hält. Dann reduziert sich, in einem weiteren MergeStep, die Anzahl der übrigen Kanten um den Faktor D.

Ein Nachteil dieses Algorithmus ist natürlich, dass nur im ersten Schritt wirklich alle PEs genutzt werden und bereits in der nächsten Iteration mindestens die Hälfte inaktiv sind. Außerdem ist Merge-Local-MST auf dünn besetzten Graphen deutlich weniger effektiv. In diesem Fall werden bei einer lokalen MST Berechnung fast alle Kanten beibehalten, da bei $m \ll n$ beinahe alle Kanten zu einem MST dazugehören.

4.1.3 Korrektheit von lokalen MST Berechnungen

Merge-Local-MST liefert nur einen korrekten MST, wenn bei lokalen MST Berechnungen in keinem Fall eine MST Kanten aus dem globalen Graph entfernt wird. Tatsächlich ist diese Aussage korrekt und wir können lokale MST Berechnungen auch bei den folgenden Algorithmen verwenden.
Angenommen jeder PE i hat auf seinen lokalen Kanten $E_{\ell i}$ einen MST berechnet und dabei die Kanten E_{Ri} entfernt. Das bedeutet, dass für eine Kante $\{s,t\} \in E_{Ri}$, die Knoten s und t bereits lokal über einen günstigeren Pfad verbunden sind. Global betrachtet, kann höchstens noch besserer Pfad von s nach t gefunden werden, aber die Kante $\{s,t\}$ wird nie eine Verbesserung dafür sein. Angenommen wir fügen die Kante $\{s,t\}$ dennoch zum MST hinzu, so schließt sich ein Kreis K und die Kante $e \in K$ mit dem größten Gewicht kann nach der Kreiseigenschaft verworfen werden. Das größte Gewicht in K muss aber die Kante $\{s,t\}$ haben, da sie sonst bereits in der lokalen MST Berechnung zum MST gehört hätte.

4.2 Borůvka-Allreduce

Der Borůvka-Allreduce Algorithmus ist eine verteilt parallele Variante von Borůvkas Algorithmus [4].
Jeder PE rechnet die lokalen leichtesten inzidenten Kanten aus und verständigt sich mittels eines Allreduce Aufrufs, um die global leichtesten inzidenten Kanten zu erhalten. Anschließend kontrahiert jeder PE den Graphen wie im sequenziellen Borůvka Algorithmus und führt die Berechnung fort. Der Pseudocode von Algorithmus 7 veranschaulicht dieses Vorgehen zusammen mit der folgenden Beschreibung der Funktionsweise.

4.3 Funktionsweise

Am Anfang einer Iteration kann jeder PE p_j ($j \in \{0, ..., p-1\}$) auf seiner lokalen Kantenmenge einen MST berechnen. Dieser Schritt ist optional, kann sich aber positiv auf die Laufzeit auswirken (siehe Kapitel 6.5). Als nächstes berechnet p_j für jeden Knoten diejenige inzidente Kante mit dem geringsten Gewicht und schreibt diese in ein Array I_L der Länge n (Zeile 4 - 11). Dabei steht im i-ten Eintrag von I_L, die leichteste inzidente Kante zu Knoten i. Um das globalen inzidente Kanten Array E_I zu erhalten, nutzen wir die Allreduce Operation, welche das Array I_L von jedem Prozessor als Eingabe bekommt (Zeile 12). Hier werden alle Arrays I_L miteinander verglichen, sodass jeder Prozessor ein Array E_I mit den global kleinsten inzidenten Kanten als Ausgabe erhält.
Diese Kanten fügt nur p_0 zum MST hinzu (Zeile 13 - 15), damit der MST einheitlich vorliegt. Anschließen kontrahiert p_j den Graphen (Zeile 20 - 21), genau wie im sequenziellen Fall. Ein Beispiel zur Kontraktion eines Graphen ist in Abbildung 4 zu sehen. Um herauszufinden welche Knoten kontrahiert werden können, nutzen wir ein weiteres Array P der Länge n. Hier schreibt

p_j in den i-ten Eintrag den Vorgängerknoten (**parent vertex**) des i-ten Knoten (Zeile 17 - 19). Der Vorgänger ist hier der kleinste Knoten, der über die leichtesten inzidenten Kanten erreichbar ist. Damit können in $O(n)$ alle Knoten mit demselben Vorgänger kontrahiert werden.
Am Ende dieser Iteration kann optional p_j noch die übrigen parallelen Kanten zwischen zwei Knoten entfernt werden (Zeile 22).
Insgesamt wird das Vorgehen so lange wiederholt, bis nur noch ein Graph mit genau einem Knoten übrig bleibt.
Anschließend muss der berechnete MST noch in die ursprüngliche Form zurück umgewandelt werden, da die MST Kanten durch das Vorgehen umbenannt wurden. Wir haben diesen Schritt in $O(n)$ implementiert, indem unsere Kanten die Form $(s, t, w, s_{origin}, t_{origin})$ hatten. Wir haben den originalen Start- und Endknoten zusätzlich gespeichert, sodass wir eine Kanten in konstanter Zeit in ihren Ausganszustand bringen können.

Algorithmus 6 BorůvkaStep (V, E, MST: Kantenliste)

```
1: E ← localMST(E,V) //optional
2: I_L[|V|]
3: I_L ← [∞, ..., ∞]
4: for each e ∈ E do
5:    if e.w < I_L[e.s].w then
6:       I_L[e.s] ← e
7:    end if
8:    if e.w < I_L[e.t].w then
9:       I_L[e.t] ← e
10:   end if
11: end for
12: E_I ← AllreduceMinIncident(N, I_L)
13: if PE hat ID = 0 then
14:    MST ← MST ∪ E_I
15: end if
16: P[|V|]
17: for i ← 0 bis |V|-1 do
18:    P[i] ← min_V{V ist über E_I zu Knoten i verbunden}
19: end for
20: V ← relabelVertices(E_I, P)
21: E ← relabelEdges(E_I, P)
22: E ← removeParallelEdges(E) //optional
```

Algorithmus 7 Borůvka-Allreduce(V, E): Kantenliste

```
1: MST: Kantenliste
2: while |V| > 1 do
3:    BorůvkaStep(V, E, MST)
4: end while
5: return  getOriginEdges(MST) //Umbenennung der Kanten rückgängig machen
```

4.3.1 Eigenschaften und Komplexität

Borůvka-Allreduce reduziert die Anzahl an Knoten in jeder Iteration um mindestens die Hälfte. Jeder Knoten wird anhand einer Kante mit einem oder mehreren Knoten kontrahiert.

Umso dichter der Graph ist, desto wahrscheinlicher ist es, dass mehr Knoten in einer Iteration kontrahiert werden. Deswegen sind insgesamt maximal $\log(n)$ Iterationen für Borůvka-Allreduce nötig.

Die Berechnung der leichtesten inzidenten Kanten ist in $O(m)$ möglich, der Allreduce liegt in $O(\alpha \log(p) + \beta\ell)$. Wobei ℓ die Größe des Arrays mit den leichtesten inzidenten Kanten ist, also $\ell \in O(n)$. Anhand der inzidenten Kanten, kann man den Graph in $O(n + m)$ kontrahieren.

Sei m' die maximale Anzahl an lokalen Kanten eines PEs. Dann liegt der initiale BorůvkaStep in $O(m' + \alpha \log(p) + \beta n)$. Für jeden folgenden BorůvkaStep wird n mindestens um die Hälfte kleiner. Hierbei liegt die Summe aus $\sum_{i=i}^{\log n} \beta \frac{n}{2^i}$ in $O(\beta n)$. Damit ergibt sich die gesamte Laufzeit von Borůvka-Allreduce in $O(\log n[m' + \alpha \log p] + \beta n)$.

4.4 Borůvka-Mixed-Merge

Jede Iteration von Borůvka-Mixed-Merge besteht aus einer abwechselnden Ausführung von einem BorůvkaStep gefolgt von einem MergeStep. Damit verbindet dieser Algorithmus die Vorteile von Merge-Local-MST und Borůvka-Allreduce. Der Aufwand von einem BorůvkaStep hängt von m und p ab und ein MergeStep ist auf möglichst dichten Graphen am effizientesten. Diese Eigenschaften werden durch Borůvka-Mixed-Merge zum Vorteil genutzt. Denn durch das Vorgehen von Borůvka-Mixed-Merge , wird vor jedem MergeStep die Knotenmenge des Graphen mindestens halbiert und die Kantenanzahl sowie die aktiven PEs vor jedem (außer dem ersten) BorůvkaStep um den Faktor D reduziert. Daher verspricht Borůvka-Mixed-Merge eine effiziente Ausführung in der Praxis. Der Pseudocode von Algorithmus 8 zeigt die genaue Vorgehensweise von Borůvka-Mixed-Merge.

Insgesamt benötigt dieser Algorithmus $\lfloor \log_D(\mathrm{p}) \rfloor$ Iterationen, so wie Merge-Local-MST , wegen dem Zusammenfügen der MST. Zum Schluss müssen die MST Kanten wieder zu ihrer Ausgangsform umbenannt werden, da sich der Graph wie in Borůvka-Allreduce verändert hat.

Algorithmus 8 Borůvka-Mixed-Merge(V, E, D: int): Kantenliste

```
1: MST: Kantenliste
2: lokalerMST ← localMST(E, V)
3: for i ← 0 bis ⌊log_D(p)⌋ do
4:     BorůvkaStep(V, lokalerMST, MST)
5:     MergeStep(V, MST, D)
6: end for
7: return getOriginEdges(MST) //Umbenennung der Kanten rückgängig machen
```

Die Laufzeit setzt sich insgesamt aus $\lfloor \log_D(p) \rfloor$ BorůvkaSteps und einem MergeSteps zusammen. Sei $m \geq n$ und m' die maximale Anzahl an lokalen Kanten eines PEs, dann liegt die gesamte Laufzeit in

$$O(\alpha \log(p) + D\beta n + \log_D(p) \cdot D\alpha \cdot T_{seq}(n, m')).$$

Hierbei ist zu beachten, dass man den $\log(p/D^i)$ Term, der bei jedem MergeStep auftritt, asymptotisch mit $\log(p)$ abschätzen kann.

4.5 Borůvka-Then-Merge

Borůvka-Then-Merge führt mehrmals den BorůvkaStep durch und anschließend den Merge-Local-MST Algorithmus, um einen MST zu berechnen. Da die Kanten durch mehrere Iterationen von Borůvka-Allreduce angepasst wurden, müssen diese für die Ausgabe wieder in den Ausgangszustand gebracht werden.
Wie bei Borůvka-Mixed-Merge werden hier die Vorteile von Borůvka-Allreduce und Merge-Local-MST hervorgehoben. Da dieser Algorithmus damit anfängt mehrmals BorůvkaStep auszuführen, wird der Graph verkleinert ohne das bereits PEs inaktiv werden. Erst wenn nur noch höchstens *Border* Knoten übrig sind, wird der restliche MST mittels Merge-Local-MST berechnet. In diesem Fall hat jeder PE deutlich weniger Knoten, als in der Eingabe und ein MergeStep kann *schneller* abgearbeitet werden.

Wie oft man zu Beginn einen BorůvkaStep durchführt, hängt vom Anwendungsfall ab. Zu viele Iterationen von Borůvka-Allreduce führen zu keiner Verbesserung, da sich der Algorithmus im kaum von Borůvka-Allreduce unterscheidet. Zu wenige Iterationen führen dazu, dass sich Borůvka-Then-Merge sehr ähnlich zu Merge-Local-MST verhält.
Dehne et al. [6] führen Borůvka so lange durch bis weniger als $n/\log_D^2(p)$ Knoten übrig sind und daher haben wir das auch in unserer Implementierung übernommen.

Algorithmus 9 Borůvka-Then-Merge(V, E, D: int, Border: int): Kantenliste

```
1: MST: Kantenliste
2: lokalerMST ← localMST(E, V)
3: while |V| ≥ Border do
4:    BorůvkaStep(V, lokalerMST, MST)
5: end while
6: MST ← Merge-Local-MST(V, MST, D)
7: return getOriginEdges(MST) //Umbenennung der Kanten rückgängig machen
```

Die gesamte Laufzeit ergibt sich durch B mal einen BorůvkaStep und $\lfloor \log_D(\mathrm{p}) \rfloor$ mal einen MergeStep. Sei m' die maximale Anzahl an lokalen Kanten eines PEs, T_{seq} die Laufzeit für eine lokale MST Berechnung und es gelte $m \geq n$. Dann liegt Borůvka-Then-Merge in

$$O(B \cdot (m' + \alpha \log p) + \beta n + \log_D(p)(T_{seq}(\tfrac{n}{2^B}, \tfrac{n}{2^B}) + D(\alpha + \beta \tfrac{n}{2^B})) + T_{seq}(\tfrac{n}{2^B}, m'))$$

5 Implementierung

Auch wenn man mittels asymptotischer Laufzeitanalyse Algorithmen sinnvoll miteinander vergleichen kann, geben sie nur einen Einblick in das theoretische Verhalten. Insbesondere stimmen die theoretischen Modelle nicht mit den real-existierenden Maschinen überein. Damit sind diese oft sehr pessimistisch und die tatsächlichen Ergebnisse können in der Praxis stark schwanken. Aus diesem Grund klären wir im Folgenden Details, die unsere Implementierung betroffen haben sowie die Art und Weise wie wir unsere Eingabegraphen konstruiert haben.
Die Programmierung unserer Algorithmen [1] ist in der Programmiersprache *C++* einstanden unter Verwendung verschiedener Bibliotheken. So haben wir für die Kommunikation zwischen Prozessoren das **M**essage **P**assing **I**nterface (MPI) verwendet. MPI ermöglicht einen einfachen und gezielten Umgang und bietet auch eine Vielzahl an kollektiven Operationen an. Für die Generierung der Eingabegraphen haben wir *KaGen*[2] [8, 10] verwendet und zum Sortieren (z.B bei Kruskals Algorithmus) *IpsoSort*[3] [2]. Dieser Sortieralgorithmus kann sowohl parallel als auch sequenziell verwendet werden. Da wir in den Algorithmen nur sequenziell Sortieren, haben wir den parallelen Algorithmus auch nicht verwendet.

5.1 Zeitmessung

Damit die Laufzeitmessung unserer Algorithmen auf mehreren Prozessoren funktioniert, wird vor dem Start-Aufruf des Timers immer eine Barriere aufgerufen, die so lange das Programm anhält bis alle Prozessoren an dieser Stelle im Code angekommen sind (und auch die Barriere aufrufen). Erst ab diesem Punkt fängt die Zeit an zu laufen. Mit demselben Vorgehen wird vor dem Stoppen des Timers auch über eine Barriere sichergestellt, dass alle Prozessoren am Ende der Ausführung angekommen sind. Erst dann wird die Zeit angehalten.

Das bedeutet im Wesentlichen, dass ein Timer Aufruf auch immer zwangsläufig ein Barrieren Aufruf beinhaltet. Somit wird zusätzliche Zeit benötigt, um darauf zu warten, dass alle Prozessoren dieselben Codebereiche ausgeführt haben.
Da wir für alle Algorithmen nicht nur die gesamte Laufzeit, sondern auch einzelne Phasen der Algorithmen messen wollen, sind wir wie folgt vorgegangen: Für die Evaluation jeglicher Weakscaling Ergebnisse, haben wir nur die Gesamtlaufzeit der Algorithmen gemessen. Das bedeutet wir haben nur einen Timeraufruf vor und nach dem Algorithmus ausgeführt, damit zusätzliche Synchronisatione des Timers keine Auswirkung auf die Algorithmen haben.
Bei den Messungen der einzelnen Phasen eines Algorithmus, können wir dieses Problem allerdings nicht umgehen. Daher sollte man beachten, dass einerseits die Gesamtlaufzeit ggf. höher ist als bei den Weakscaling Ergebnissen. Anderseits können Phasen wie das Allreduce bei dem Borůvka-Allreduce Algorithmus geringer erscheinen als sie im Normalfall sind. Das liegt daran, dass für ein Allreduce eine Synchronisation aller Prozessoren für den Nachrichtenaustausch stattfindet. Sollte also ein einzelner Prozessor länger für eine vorherige Aufgabe benötigen als die übrigen, so gehört zum Allreduce das Warten auf diesem Prozessor dazu. Da wir in unserem Fall aber über den Timer Aufruf alle Prozessoren vor dem Allreduce Aufruf synchronisieren, dauert das Warten auf einen einzelnen Prozessor im Anschluss nicht so lange wir üblich.

[1] https://github.com/u-Texon/parallel-dense-mst
[2] https://github.com/KarlsruheGraphGeneration/KaGen
[3] https://github.com/SaschaWitt/ips4o

5.2 Generierung der Graphen

Die von uns verwendete KaGen Bibliothek generiert abhängig von der Anzahl an n Knoten, m Kanten, p Prozessoren und Graphtyp t einen bestimmten Graphen. Für diese Generierung muss $p, m, n \in \{2^k | k \in \mathbb{N}\}$ gelten. Daher haben wir unsere Implementierung nur mit einer Zweierpotenz an Prozessoren, Kanten und Knoten evaluiert und getestet. Auch wenn die Algorithmen mit beliebigen Eingaben und Konfigurationen funktionieren.
Nach der Generierung liegt der Graph global lexikographisch sortiert auf den einzelnen Prozessoren vor. Zusätzlich wird zu jeder Kante (s, t, w) auch die Rückkante (t, s, w) generiert. Allerdings können die verwendeten Algorithmen auch auf zufällig permutierten Kantenlisten arbeiten. Weil wir eventuelle Auswirkungen einer sortierten Eingabe ausschließen möchten, wurde die global sortierte Kantenliste zufällig auf alle PEs verteilt. Zu den wesentlichen Graph Typen, die wir verwendet haben gehören GNM, RHG und Pair. Bei der Generierung von GNM Graphen mit n Knoten wird m mal zwischen zwei zufälligen Knoten eine Kante gezogen. Bei RHG Graphen hingegen werden zunächst n Punkte auf einem Kreis mit Radius R gesetzt. Mit weiteren Parametern kann gesteuert werden, wie nah diese Punkte an dem Kreisradius liegen sollen. Zwischen zwei Knoten verläuft eine Kante, wenn der *hyperbolische Abstand* kleiner als R ist.

5.2.1 Pair Graph

Der Pair Graph ist ein von uns erstellter Graph beruhend auf der Idee von Dehne und Götz [6] mit dem Ziel möglichst viele Iterationen in Borůvkas Algorithmus zu erzwingen.
In jeder Iteration von Borůvkas Algorithmus verringert sich die Knotenanzahl um mindestens die Hälfte. Besonders auf dichten Graphen, können in einem Schritt eine Vielzahl von Knoten kontrahiert werden, sodass Borůvkas Algorithmus deutlich weniger als $\log(n)$ Iterationen benötigt. Um aber explizit dieses Szenario betrachten zu können, haben wir mit der Beschreibung einen eigenen Pair Graphen generiert.

Das Vorgehen zum Generieren eines solchen Graphen ist relativ simpel, besonders für uns, da wir nur Graphen mit $n = 2^k$ betrachten:
Wir erstellen zunächst zwischen jedem Zweierpaar an Knoten eine Kante mit Gewicht 1, anschließend zwischen jedem Vierernpaaren mit Gewicht 2, dann zwischen Achterpaaren mit Gewicht 3 und so weiter. Somit generieren wir insgesamt n-1 Kanten und können weitere zufällige Kanten (mit minimalem Gewicht von n) hinzufügen. Eine schematische Darstellung eines Pair Graphen mit 8 Knoten ist in Abbildung Abbildung 6 dargestellt.

Durch die geringen Kantengewichte stellen wir sicher, dass es sich bei all diese Kanten genau um die MST-Kanten des Graphen handelt. Im ersten Borvkaschritt werden also alle Knotenpaare die über eine Kante mit Gewicht eins verbunden sind kontrahiert, im nächsten Schritt diejenigen Paare, die über Gewicht zwei verbunden sind und so weiter. Damit ergibt sich für einen Pair Graphen immer die maximal mögliche Anzahl von $\log(n)$ Iterationen für Borůvkas Algorithmus.

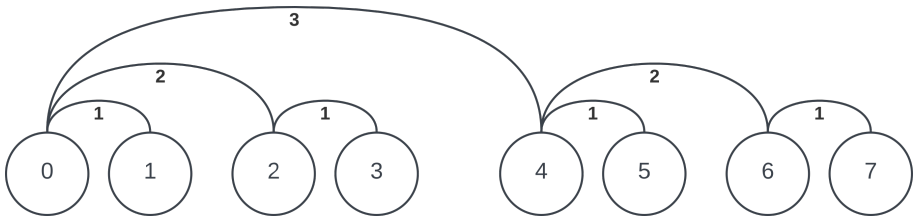

Abbildung 6: Pair Graph mit $n = 8$

6 Ergebnisse und Evaluierung

Die Algorithmen aus Kapitel 4 haben wir auf dem SuperMUC-NG Supercomputer ausgeführt. Dieser Supercomputer ist ein Cluster bestehend aus 6336 *Compute-Nodes*. Ein Compute-Node besteht aus 48 Prozessoren und besitzt 96 GByte Hauptspeicher. Untereinander sind die Compute-Nodes über das OmniPath-Network mit einer Bandbreite von 100GBit/s verbunden. Als Compiler haben wir GCC 11.2 und die MPI Version 4.0.7 verwendet.
Die folgenden Angaben zur Kantenmenge bezieht sich immer auf den **ungerichteten** Graphen. Allerdings ist die Kantenmenge für die Ausführung genau doppelt so Groß, da wir die jeweiligen Rückkanten mit generiert haben. Insgesamt haben wir jeweils 5 durchläufe von jedem Algorithmus ausgeführt und den jeweils ersten verworfen, um *Warmup-Effekte* auszuschließen. Aus den restlichen vier Laufzeiten haben wir schließlich den Durchschnitt berechnet.

6.1 Lokalen MST Berechnung

Weil lokale Berechnung von MSTs einen wesentlichen Teil von den verteilt parallelen Algorithmen ausmachen, muss dieser so effizient wie möglich sein. Daher benutzen wir für die lokale MST Berechnung Filter-Kruskal als Alternative zum klassischen Kruskal. Dieser ermöglicht in der Praxis eine deutlich bessere Laufzeit auf einer breiten Masse an Eingaben. In Abbildung 7 ist ein direkter Vergleich zwischen Kruskals Algorithmus und Filter-Kruskal zu sehen. Hierbei ist auf der X-Achse der Knotengrad (m/n) und auf der Y-Achse die benötigte Zeit pro Kante (m/Laufzeit) abgebildet. Die Ausführung hat auf einem (GNM) Graph mit 2^{18} Knoten und 2^{18} bis 2^{24} Kanten stattgefunden. Es ist zu eindeutig zu erkennen, dass Filter-Kruskal effizienter ist als Kruskals Algorithmus. Dass liegt daran, dass Filter-Kruskal im Gegensatz zu Kruskals Algorithmus nicht alle Kanten sortiert werden. Aufgrund der besseren Laufzeit verwenden wir für die folgenden Experimente immer Filter-Kruskal als Basis Algorithmus für die Bestimmung von lokalen MSTs.

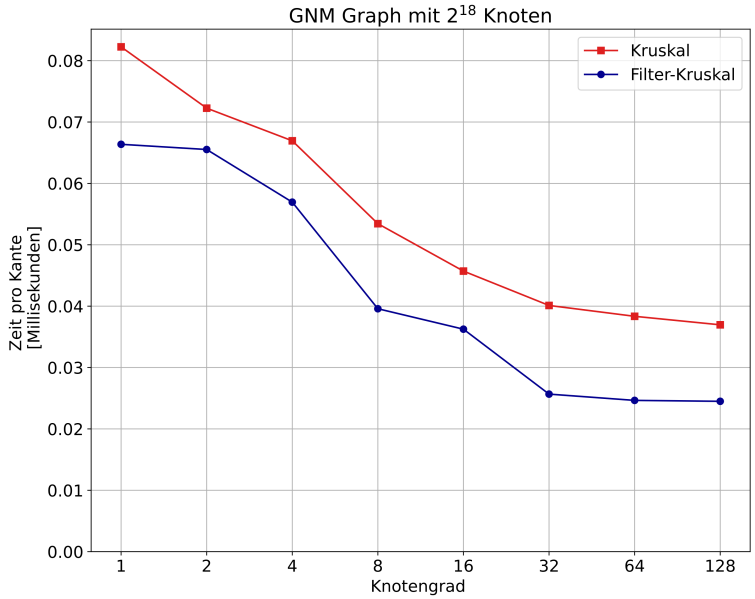

Abbildung 7: Vergleich zwischen Kruskals Algorithmus und Filter-Kruskal

6.2 Sehr Dichte Graphen

Die behandelten MST Algorithmen sind für sehr dichten Graphen ausgelegt. Einerseits muss die Knotenmenge auf jedem PE repliziert werden können und andererseits sind jegliche Iterationen von Borůvka-Allreduce und Merge-Local-MST auf dichten Graphen besonders effektiv. Insbesondere werden auf dichten Graphen in einem BorůvkaStep mehr Knoten kontrahiert und bei einem MergeStep mehr Kanten durch lokale MST Berechnungen entfernt. In Abbildung 8 ist die Laufzeiten von Borůvka-Allreduce, Merge-Local-MST, Borůvka-Mixed-Merge und Borůvka-Then-Merge abgebildet. Hierbei handelt es sich um „weak scaling“, d.h die Anzahl an Kanten im Graphen skaliert mit der Anzahl an PEs. Eine optimale Effizienz der Algorithmen würde sich also als eine konstante Linie widerspiegeln. In diesem Fall haben wir die Algorithmen auf einem RHG und einem GNM Graph mit 2^{18} Knoten und 2^{22} Kanten pro PE ausgeführt. Auf der X-Achse ist die Anzahl an PEs von 1 bis 2048 angegeben und auf der Y-Achse sehen wir die durchschnittliche Laufzeit in Millisekunden. Borůvka-Allreduce berechnet nur zu Beginn einen lokalen MST und entfernt keine parallelen Kanten. Außerdem ist der verwendete Treefactor für alle Algorithmen $D = 2$.

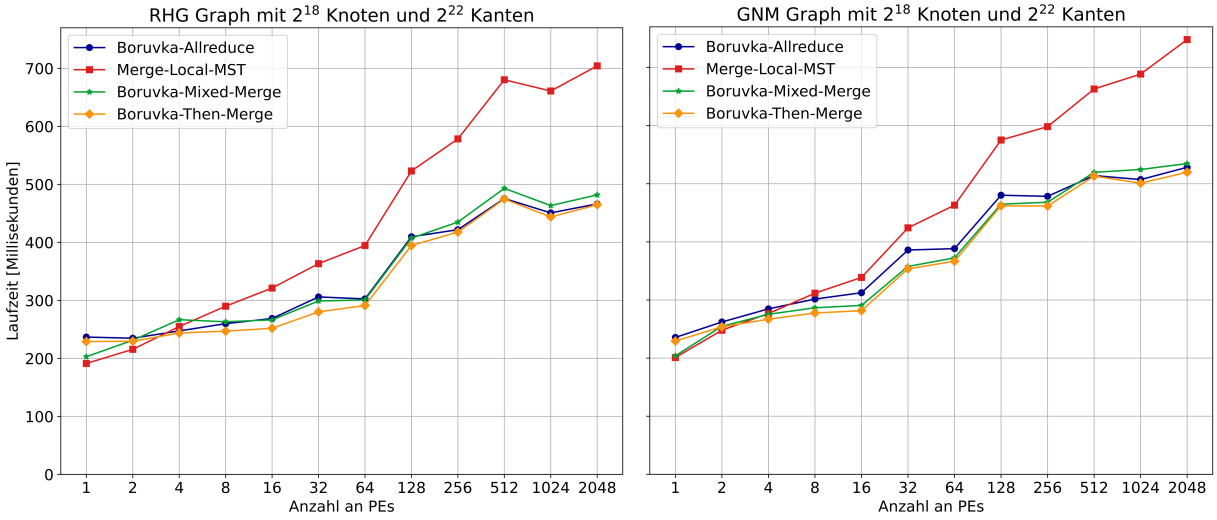

Abbildung 8: Weakscaling auf dichten Graphen

Wir erkennen, dass auf dem GNM Graphen Merge-Local-MST mit ca. 0,75 Sekunden bei 2048 PEs am schlechtesten abschneidet, wobei die übrigen Algorithmen alle mit ca. 0,5 Sekunden fast identisch sind. Auf dem RHG Graphen ist das Verhalten ebenso zu erkennen. Es ist erwartbar, dass Merge-Local-MST am langsamsten ist. Immerhin werden nur in der ersten Iteration alle PEs gänzlich genutzt. Es überrascht allerdings, dass es kaum einen Unterschied zwischen den restlichen Algorithmen gibt. Hierfür betrachten wir in Abbildung 9 explizit die Laufzeiten der einzelnen Algorithmen. Diese sind in die jeweiligen Iterationen und Phasen aufgeteilt.

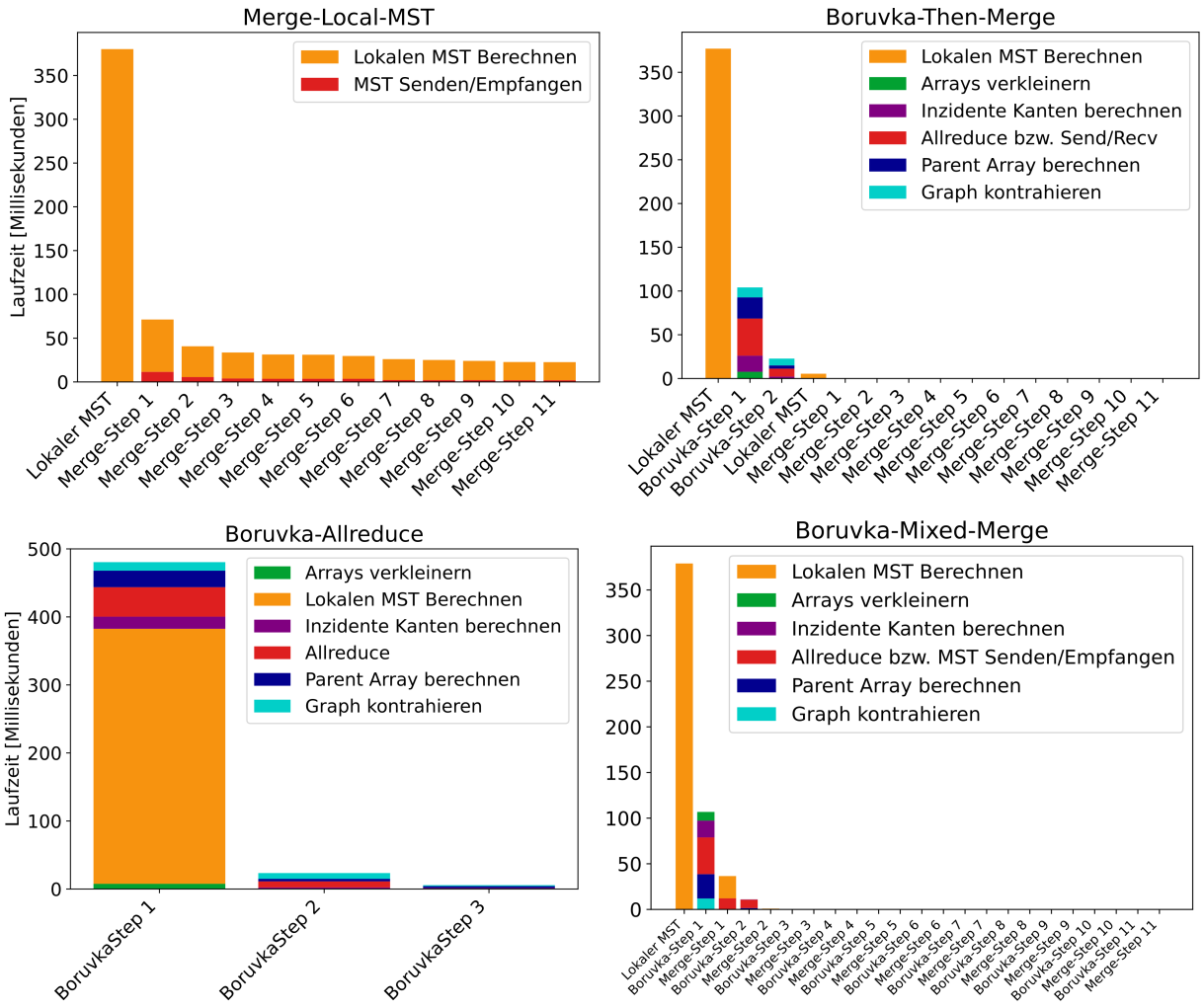

Abbildung 9: Ansicht für einzelne Algorithmen

Hier ist eindeutig, dass die Laufzeit fast ausschließlich von lokalen MST Berechnung stammt. Bei Borůvka-Allreduce dominiert einerseits die Laufzeit der ersten Iteration und andererseits haben folgenden Iterationen kaum eine Auswirkung auf die Laufzeit. So benötigt der lokale MST mit ca. 0.38 Sekunden etwa 80% der gesamten Laufzeit. Da sowohl Borůvka-Allreduce als auch Borůvka-Then-Merge und Borůvka-Mixed-Merge mit einem BorůvkaStep beginnen und dieser Schritt fast die gesamte Laufzeit ausmacht, ist es logisch, dass diese Algorithmen fast gleich schnell sind.

Wenn aber nur die lokale MST Berechnung relevant für den globalen MST ist, warum ist dann nicht auch Merge-Local-MST gleich schnell? Immerhin besteht dieser Algorithmus fast nur aus diesen Schritten.
Um diese Frage zu beantworten, betrachten wir die Boxplots in Abbildung 10. Diese Graphik zeigt die Anzahl an Knoten und Kanten für jeden PE nach den jeweiligen Iterationen.

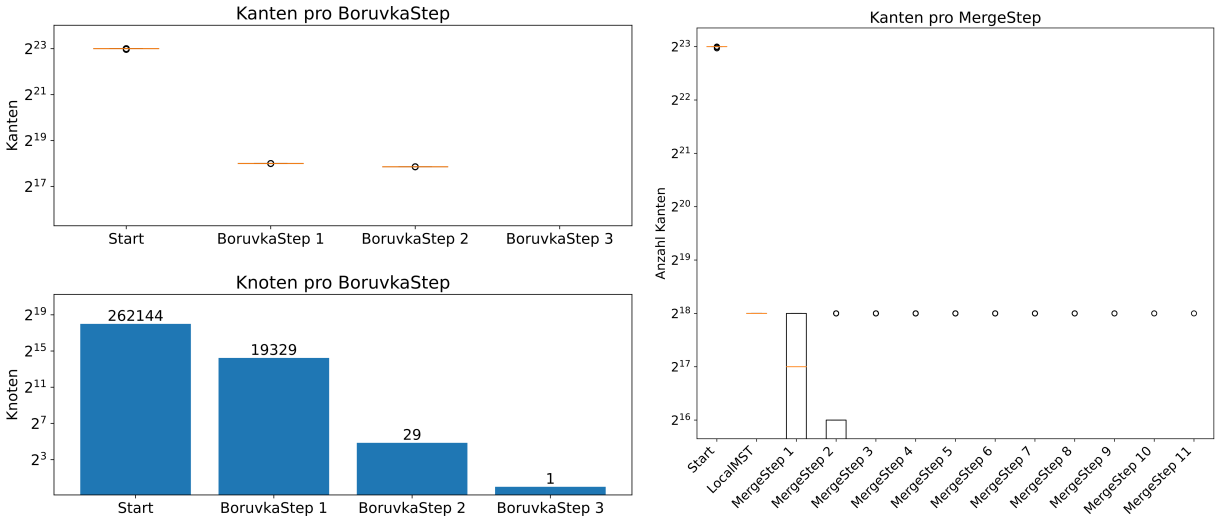

Abbildung 10: Boxplots von Merge-Local-MST und Borůvka-Allreduce

Da der Graph besonders dicht war, konnte Merge-Local-MST in der ersten Iteration besonders viele Kanten bereits aussortieren. Allerdings ist das bei den folgenden Iterationen nicht mehr möglich. Nach der ersten Iteration hält jeder PE nur noch ca. $n = 2^{18}$ Kanten. In den nächsten Iterationen werden also nur zwei MSTs mit 2^{18} Kanten verschmolzen bis nur noch ein PE aktiv ist. Während also die anderen Algorithmen mit einem BorůvkaStep den Graph schrumpfen und die restlichen Kanten aussortieren, benötigt Merge-Local-MST viel Zeit mit weiteren lokalen Berechnungen, die kaum zielführend sind.

6.3 Verschmelzen von lokalen MSTs

Der Treefactor (oder D) gibt an wie viele lokale MSTs in einer Iteration von Merge-Local-MST verschmolzen werden. Ein kleinerer Treefactor sorgt somit für mehr Iterationen, aber dafür sind weniger Prozessoren im Anschluss inaktiv.
In Abbildung 11 sehen wir Merge-Local-MST mit fünf verschiedenen werten für D. Dabei unterscheiden wir für die Treefactor Werte zwischen 2, 3, 4, 8 und 16 auf einem GNM Graphen mit 2^{18} Knoten und 2^{20} Kanten pro PE unter Verwendung von Filter-Kruskal.

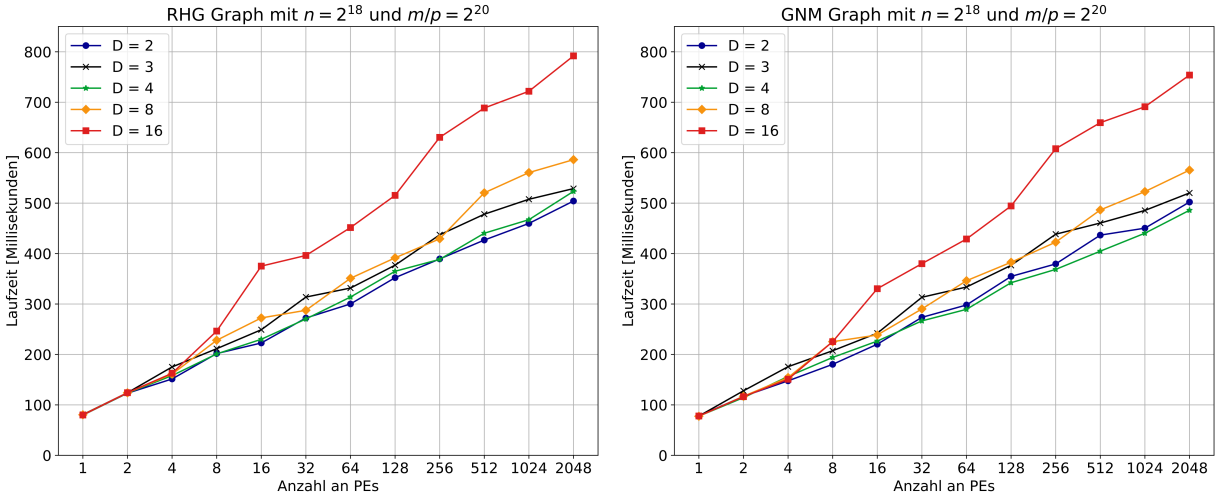

Abbildung 11: Laufzeit von Merge-Local-MST mit verschiedenen Treefactor Werten

Dehne und Götz [6] haben in ihren Experimenten mit D=3 die besten Ergebnisse erzielt. Wie man in Abbildung 11 aber sieht, war das bei uns nicht der Fall. Hier hat Merge-Local-MST mit $D = 4$ bzw. $D = 2$ die beste Laufzeit. Die Experimente haben gezeigt, dass sich bereits ein größerer Treefactor als 4 negativ auf die Laufzeit auswirkt. Dies wird durch diese Graphik auch nochmal bestätigt. Bereits mit D=8 ist die Laufzeit ca. 10% und bei D=16 schon ca.50% höher als bei D=2.
Für die meisten Eingaben war $D = 2$ am effizientesten, weshalb wir diesen Wert die restlichen Experimente verwendet haben.

6.4 Ergebnisse auf unterschiedlichen Graphen

Wir möchten nun beobachten, wie sich die Algorithmen auf den unterschiedlichen Graphen verhalten. Dafür betrachten wir die von uns implementierten Algorithmen auf drei unterschiedlichen Graphtypen. Wir nutzen als Eingabegraphen GNM, RHG und Pair mit 2^{18} Knoten und 2^{17} bis 2^{20} Kanten pro PE.

6.4.1 GNM

Abbildung 12 zeigt die Ergebnisse unserer Experimente mit dem GNM Graphen als Eingabe. Die gute Skalierung der Algorithmen wird dadurch deutlich, dass, obwohl die Eingaben bis zu vier Mal mehr Kanten besitzen, die Laufzeiten dennoch in der selben Größenordnung liegen. So ist Merge-Local-MST bei 2^{20} Kanten pro PE nur 10% langsamer als bei 2^{20} Kanten auf 2048 Prozessoren. Weiterhin zeigen die Experimente zu unserer Erwartung, dass Merge-Local-MST die größte Laufzeit von allen Algorithmen benötigt. Das liegt daran, Merge-Local-MST nach der ersten Iteration nur noch deutlich langsamer Fortschritte macht, wie es auch z.B in Kapitel 6.2 zu beobachten ist.
Im Gegensatz zu Kapitel 6.2 ist der Graph deutlich weniger dicht und die lokale MST Berechnung domioniert die Laufzeit weniger stark. Nun sind Unterschiede zwischen Borůvka-Allreduce , Borůvka-Then-Merge und Borůvka-Mixed-Merge besser erkennbar. So sieht man bei 2^{20} Kanten pro PE, dass Borůvka-Allreduce über 10% langsamer ist, als Borůvka-Then-Merge und Borůvka-Mixed-Merge. Dennoch liegen die Laufzeiten dieser drei Algorithmen nah bei einander, da nach wie vor wenige Iterationen von Borůvka-Allreduce einen großen Teil der Laufzeit ausmachen.

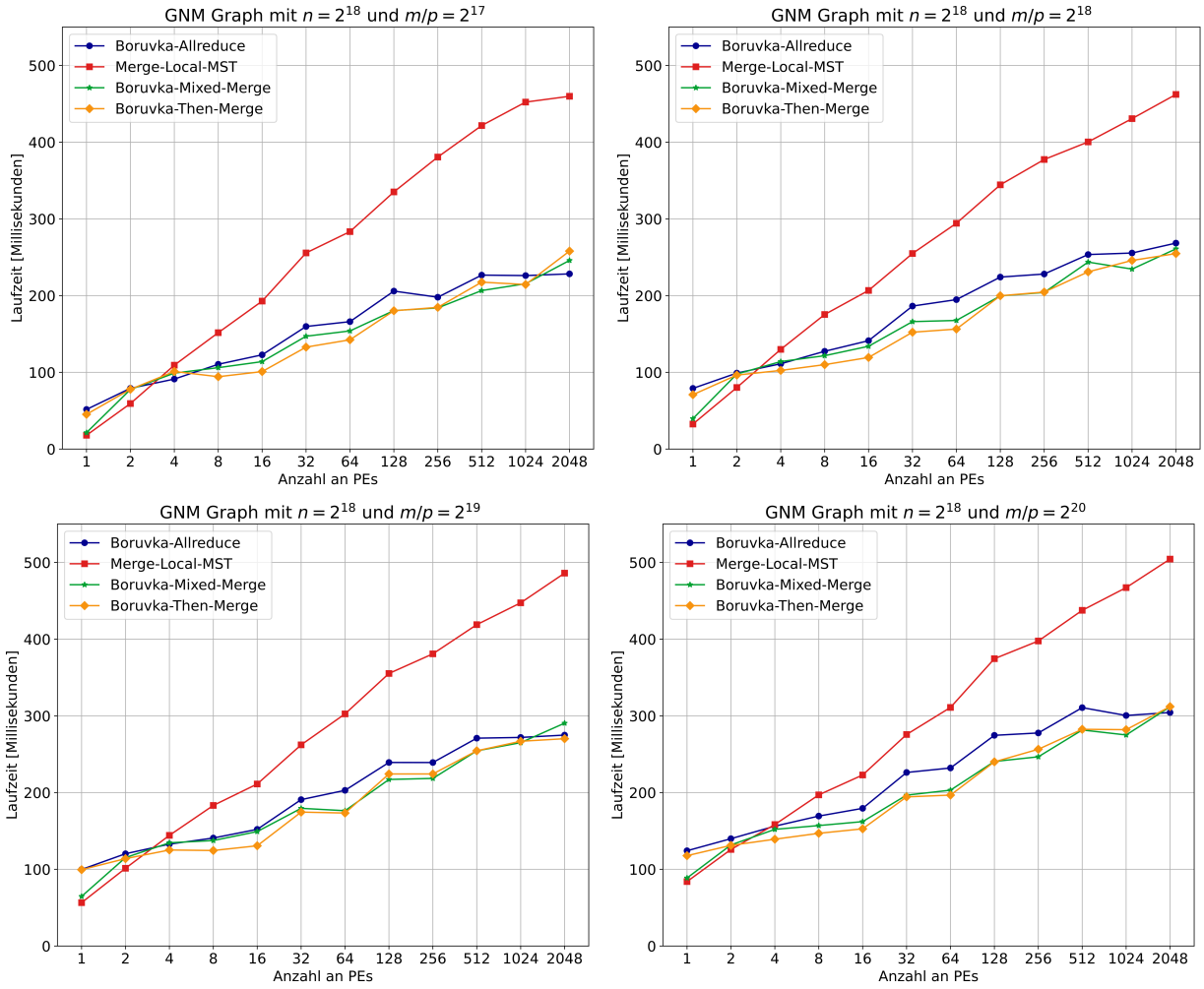

Abbildung 12: GNM Graphen mit 2^{18} Knoten und 2^{17} bis 2^{20} Kanten pro PE

6.4.2 RHG

In Abbildung 13 sieht man die Laufzeiten der Algorithmen mit den selben Konfigurationen, aber diesmal auf dem RHG Graph. Wie bei den GNM Graphen bleibt die gute Skalierung erhalten und auch der Verlauf der Laufzeiten sieht identisch aus. Hierbei liegen die Laufzeiten von Borůvka-Allreduce, Borůvka-Then-Merge und Borůvka-Mixed-Merge noch näher bei einander als auf GNM Graphen. Vermutlich sorgt die Struktur des Graphen für eine erhöhte Effizienz bei Iterationen von Borůvka-Allreduce. Damit sind die Aufwände der drei Algorithmen größtenteils nur von Borůvka-Allreduce abhängig.
Die Unterschiede zwischen den drei Algorithmen werden erst deutlich, wenn mehr Iterationen von Borůvka-Allreduce nötig sind, um den MST zu berechnen. Um diesen Fall zu betrachten bietet sich der Pair Graph an.

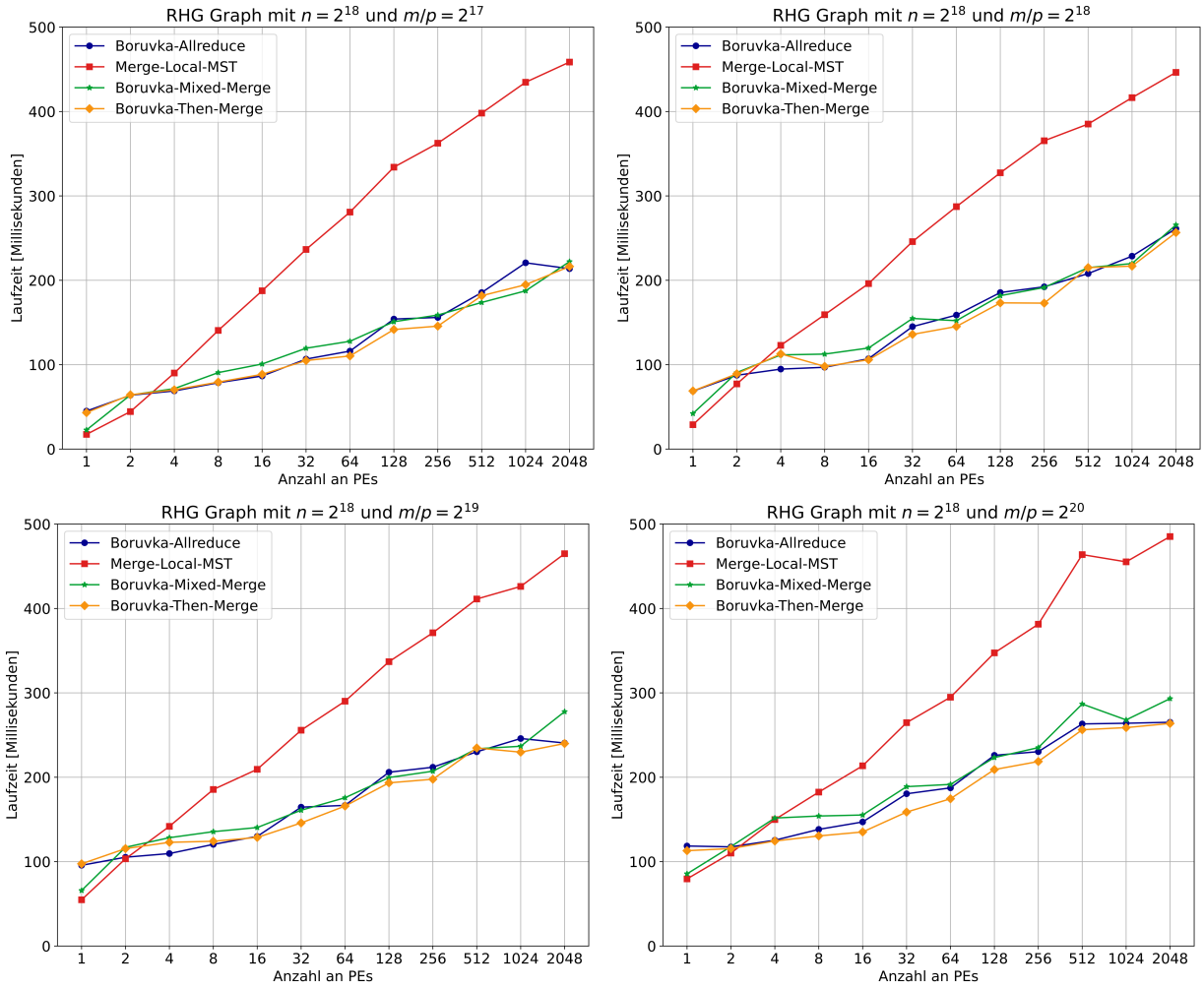

Abbildung 13: RHG Graphen mit 2^{18} Knoten und 2^{17} bis 2^{20} Kanten pro PE

6.4.3 Pair

Schließlich betrachten wir die Laufzeiten der Algorithmen auf dem Pair Graphen in Abbildung 14. Im Gegensatz zu GNM und RHG Graphen, sind hier deutliche Unterschiede in der Effizienz der Algorithmen zu erkennen.
Borůvka-Allreduce ist nun der langsamste Algorithmus. Das ist in diesem Fall sinnvoll, da der Pair Graph den schlechtesten Eingabegraphen für Borůvkas Algorithmus darstellt, indem eine maximale Anzahl an Iterationen erzwungen wird. Der effizienteste Algorithmus ist hingegen Borůvka-Mixed-Merge. Durch die Abwecheslnde Ausführung von einem BorůvkaStep und einem MergeStep wird die Kantenanzahl vor jedem BorůvkaStep verringert, sodass dieser deutlich weniger Zeit benötigt.
Borůvka-Then-Merge ist hier etwas langsamer, da die ersten Iterationen von Borůvkas Algorithmus immer noch starke Auswirkungen haben. Allerdings lohnen sie sich dennoch insofern, dass Borůvka-Then-Merge noch effizienter als Merge-Local-MST ist.

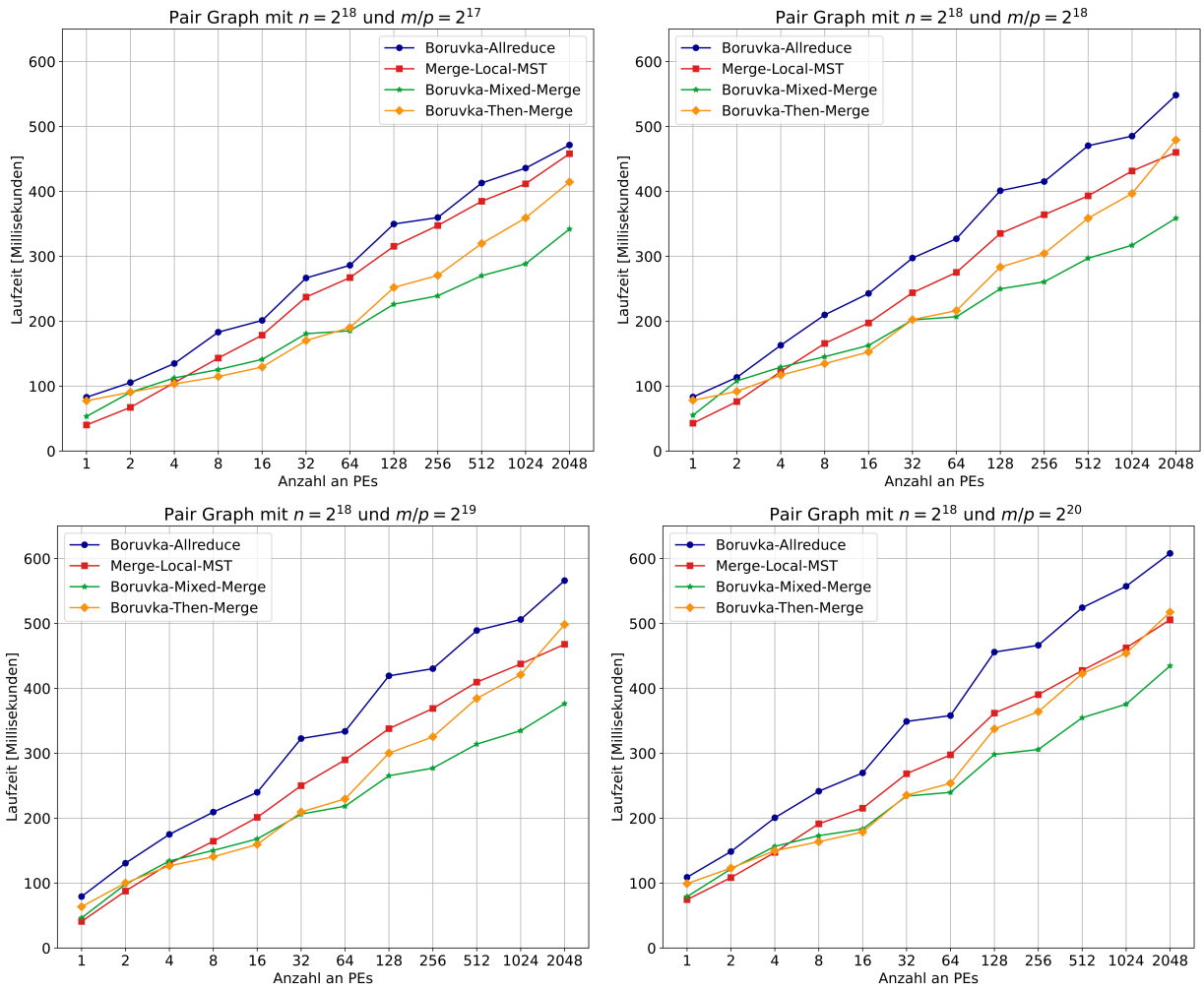

Abbildung 14: Pair Graphen mit 2^{18} Knoten und 2^{17} bis 2^{20} Kanten pro PE

6.5 Entfernen von parallelen Kanten

Nachdem wir in einem BorůvkaStep den Graph kontrahiert und die Kanten umbenannt haben, bleiben viele Kanten übrig die man verwerfen kann. Einerseits gibt es Kanten der Form (s, t, w), bei denen die Knoten s und t in der gleichen Komponente lagen und diese im Anschluss die Form (s', s', w) haben. Diese Schlingen sind offensichtlich niemals Teil eines MSTs und werden in unserer Implementierung bei der Umbenennung direkt entfernt.
Andererseits können auch viele *parallele* Kante auftreten, also mehrere Kanten die die gleichen Knoten Verbinden wie z.B die Kante $(0, 1, 7)$ und $(0, 1, 12)$. Hierbei muss nur diejenige Kante beibehalten werden, welche das geringste Gewicht besitzt. Da dieser Schritt allerdings nur Kanten entfernt, die niemals zum MST hinzugefügt werden, ist dieses Vorgehen bei einem BorůvkaStep optional.
Nun fragen wir uns, ob bzw. wie sehr sich dieser Schritt lohnt und auf welche Weise man diesen am effizientesten implementieren kann.
Für unsere Implementierung zum Entfernen der parallelen Kanten haben wir zunächst alle Kanten nach Startknoten, Endknoten und dann nach Gewicht sortiert. Anschließend iterieren wir ein weiteres mal über die Kanten und behalten nur das erste Vorkommen einer Kante. Damit benötigt removeParallelEdges $O(m \log m)$ Operationen zum Sortieren plus $O(m)$ Operation für das Entfernen.
Alternativ dazu können wir auch zu Beginn von jedem BorůvkaStep mittels Filter-Kruskal

einen lokalen MST berechnen.

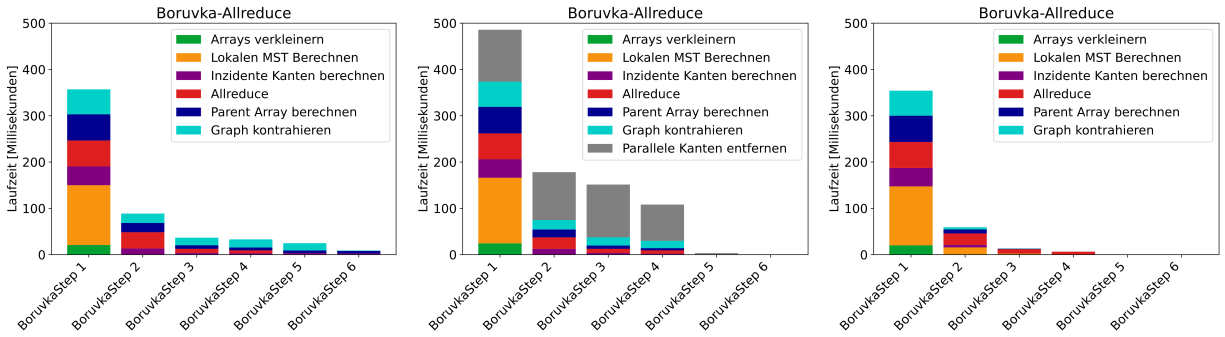

Abbildung 15: Borůvka-Allreduce mit und ohne optionaler Kantenreduktionen

Wir haben in Abbildung 15 die Laufzeiten von Borůvka-Allreduce auf einem GNM Graphen mit 2^{19} Knoten und 2^{19} Kanten pro PE verglichen. Der linke Graph zeigt die Laufzeit von Borůvka-Allreduce ohne das Entfernen von parallelen Kanten und in der Mitte mit dem Entfernen. Rechts sieht man die Alternative mittels Filter-Kruskal. Beachte, dass wir bei allen drei Durchläufen immer zu Beginn einen lokalen MST berechnen, da dieses Vorgehen generell sehr effizient ist.
Wie man sieht, bildet das Entfernen der Parallelen kanten einen großen Nachteil im Vergleich zu den andern beiden Alternativen. Scheinbar werden verhältnismäßig wenige Kanten aussortiert und das benötigte Sortieren aller umbenannten Kanten hat deutliche negative Auswirkungen auf die Laufzeit.
Weiterhin fällt auf, dass Filter-Kruskal hingegen sehr effizient ist. Bei genauerer Überlegung wird klar, dass Filter-Kruskal mit einer erwarteten Laufzeit von $O(m)$ nicht nur schneller als unsere Implementierung in $O(m \log m)$ ist, sondern potenziell noch mehr Kanten als nur die parallelen entfernt.
Damit hat sich herausgestellt, dass sich Filter-Kruskal in der Praxis als eine sehr gute Methode entpuppt um parallele Kanten zu entfernen.
Um unsere Implementierung in der Praxis zu verbessern ist es weiterhin möglich Hash Tabellen zu verwenden. Dafür haben wir einen Teil (ca. 5%) der leichtesten Kanten in einer Hash Tabelle gespeichert und anschließend für die übrigen Kanten überprüft, ob diese bereits in der Tabelle vorkommen. Wenn sie bereits enthalten sind, kann man sie entfernen, da sie ein höheres Gewicht haben. Im Anschluss haben von allen übrigen Kanten mit unserer $O(m \log m)$ Implementierung die parallelen Kanten entfernt.
In unseren Experimenten hat sich dieses Vorgehen nicht als vorteilhaft herausgestellt, weswegen wir dieses wieder verworfen haben.

6.6 Nachrichten Überlagern

Eine Möglichkeit, um einen BorůvkaStep zu beschleunigen, ist es den Allreduce Aufruf mit der lokalen MST Berechnung zu überlagern. Da sowohl Synchronisation und Nachrichtenübertragung sowie das lokale Berechnen von MSTs einen Großteil von einem BorůvkaStep ausmachen, möchten wir dies zu unserem Vorteil nutzen.
Unsere Idee war es zunächst mittels *Hyper-Threading* zwei Threads pro PE zu starten, sodass ein Thread den Allreduce Aufruf durchführt während der andere den lokalen MST berechnet. Bei Hyper-Threading laufen mehrere Threads auf einem Prozessor und können den Prozessor

effizienter ausnutzen, indem z.B ein Thread auf dem Prozessor läuft während der andere auf eine Antwort oder Ähnliches warten muss. Eine naive Umsetzung von diesem Vorgehen konnte in der Praxis keinen Vorteil erzielen, da unsere Implementierung nicht ohne weiteres auf Hyper-Threading ausgelegt war. Außerdem benötigt dieses Vorgehen, dass in jedem BorůvkaStep die leichtesten inzidenten Kanten berechnet werden **bevor** die lokale MST Berechnung durchgeführt wird. Damit benötigt die Berechnung der leichtesten inzidenten Kanten mehr Zeit als gewöhnlich, da über mehr Kanten iteriert werden muss. Weil die Zeit für diese Abschlussarbeit knapp wurde, konnten wir das Vorgehen eines BorůvkaStep mit Hyperthreading nicht genauer analysieren und die Implementierung darauf anpassen. Um dennoch untersuchen zu können, ob das Überlagern von Nachrichten in der Praxis effizient ist, haben wir stattdessen zusätzliche PEs für den Allreduce Aufruf angefordert. Hierbei haben wir einem MPI Aufruf zwei PEs zugeordnet, damit in jedem BorůvkaStep ein PE den lokalen MST berechnet, während der andere den Allreduce Aufruf durchführt. Insofern wartet die Hälfte der PEs außerhalb der Nachrichtenüberlagerung. Insgesamt haben wir die Experimente mit der doppelten Anzahl an PEs ausgeführt.

Der Ansatz der Nachrichtenüberlagerung lohnt sich am meisten, wenn der Allreduce Aufruf und die lokale MST Berechnung einen großen Bestandteil der Laufzeit ausmachen. Es hat sich in Abschnitt 6.5 ergeben, dass es sich lohnt in jedem BorůvkaStep einen lokalen MST mit Filter-Kruskal zu berechnen, anstatt parallele Kanten zu entfernen. Das bedeutet im Wesentlichen, dass es einen Vorteil verschafft in jedem BorůvkaStep einen lokalen MST zu berechnen, so wie es bei der Nachrichtenüberlagerung der Fall ist. Ansonsten würde das Überlagern von Nachrichten einen Nachteil bilden.
Zusätzlich muss der lokale MST einen ähnlich hohen Aufwand bilden wie der Allreduce Aufruf. Das ist aber nicht immer der Fall. Unsere Experimente haben ergeben, dass für $m/p \approx n$ der Allreduce Aufruf ungefähr so lange wie Filter-Kruskal benötigt. Um dies zu verdeutlichen, zeigt Abbildung 16 die Laufzeit von drei Borůvka-Allreduce Durchläufen mit jeweils 1000 PEs. Die Ausführung hat auf einen GNM Graphen mit 2^{18} Knoten und 2^{17}, 2^{18} und 2^{19} Kanten pro PE stattgefunden.

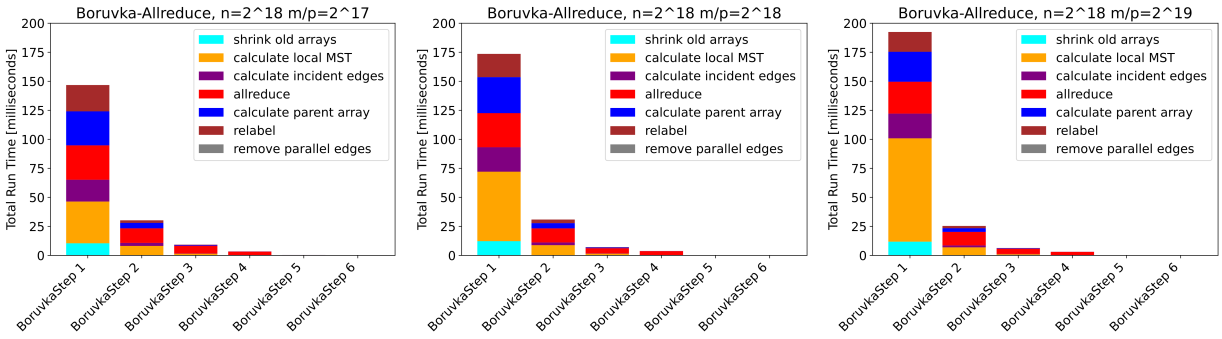

Abbildung 16: Vergleich zwischen $m/p = 2^{17}, 2^{18}$ und 2^{19} mit $n = 2^{18}$

Hierbei erkennt man, dass bei $m/p = 2^{17}$ und $m/p = 2^{18}$ die lokale MST Berechnung ähnlich viel Zeit wie der Allreduce Aufruf gebraucht hat. Daher betrachten wir auf diesen beiden Eingaben in Abbildung 17 die Laufzeiten Borůvka-Allreduce mit und ohne Nachrichtenüberlagerung. Wir führen Borůvka-Allreduce ohne Nachrichtenüberlagerung auch auf doppelt so vielen PE's aus, nutzen aber die extra PEs nicht. Damit sorgen wir für gleiche Bedingungen, da in diesem Fall ggf. mehr Speicher für die einzelnen PEs zur Verfügung steht. Dadurch sind die Laufzeiten in Abbildung 17 außerdem noch ein Stück schneller als in Abbildung 16.

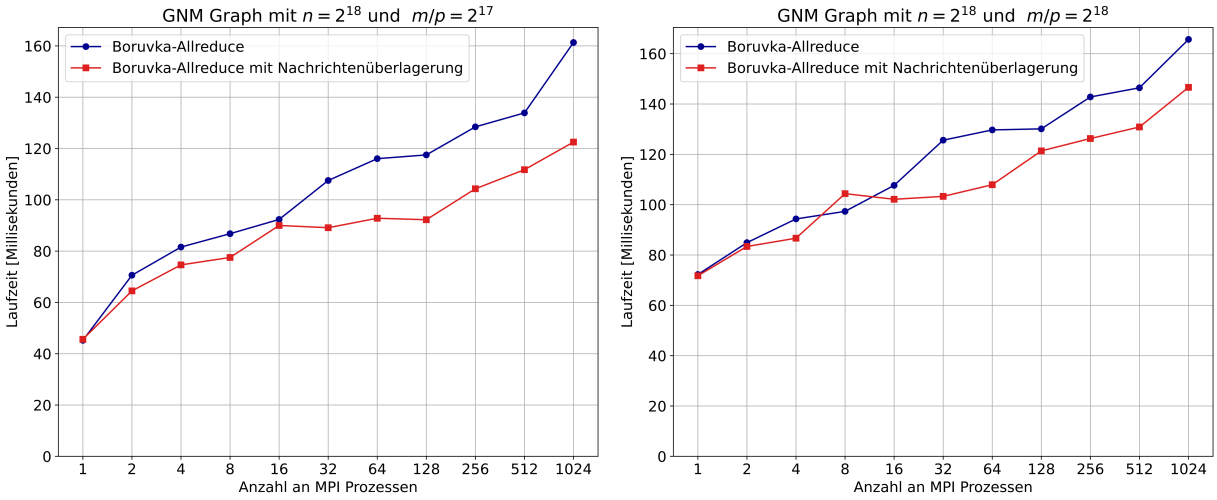

Abbildung 17: Vergleich von Borůvka-Allreduce mit und ohne Nachrichtenüberlagerung

Man erkennt, dass auf diesen beiden Eingaben die Überlagerung von Nachrichten einen Vorteil gegenüber zu einem üblichen Borůvka-Allreduce hat. Dabei ist eine Beschleunigung von biszu 30% auf 1024 Prozessoren zu beobachten. Auch wenn die Bedingung and die Eingabe strikt ist, ist dieser Ansatz dennoch vielversprechend und effizient.

7 Fazit

Die Experimente haben gezeigt, dass die von Dehne et al. [6] entwickelten Algorithmen in der Praxis auch auf mehreren tausend Prozessoren effizient skalieren. Während Merge-Local-MST selten kürzere Laufzeiten als Borůvka-Allreduce besitzt, so waren Borůvka-Mixed-Merge und Borůvka-Then-Merge auf vielen verschiedenen Eingaben die effizientesten Algorithmen. Besonders Borůvka-Mixed-Merge hat auf dem Pair Graphen, der viele Iterationen von Borůvka-Allreduce benötigt, die höchste Effizienz gezeigt.
Zusätzlich konnten wir einen deutlichen Einfluss auf die verteilt parallelen Algorithmen durch lokalen MST Berechnungen beobachten. Nicht nur bildet ein effizienter sequenzieller MST Algorithmus eine wichtige Basis für verteilte MST Algorithmen. Sondern machen diese auch bei sehr dichten Graphen einen Großteil der Laufzeit aus. Dabei war die Verwendung von Filter-Kruskal gegen über Kruskals Algorithmus ein wesentlicher Faktor für eine bessere Effizienz aller Algorithmen. Zusätzlich hat sich Filter-Kruskal als eine sehr effiziente Methode gezeigt, um in einem BorůvkaStep parallele Kanten zu entfernen.
Zusätzlich konnten wir beobachten, dass die Überlagerung von Nachrichten ein effizientes Vorgehen sein kann, um einen BorůvkaStep weiter zu beschleunigen. Allerdings konnten wir aus Zeitgründen dieses Vorgehen nur Anhand einer beispielhaften Implementierung betrachten, die in der Praxis nicht sinnvoll ist. Außerdem ist eine Nachrichtenüberlagerung nur sinnvoll, wenn die lokalen MST Berechnungen einen ähnlich Anteil an der Laufzeit ausmachen wie der Allreduce Aufruf. Dies war bei unseren Experimenten nur auf weniger dichten Graphen möglich, was wiederum die generelle Effizienz der Algorithmen mindert.

Da dieser Ansatz dennoch vielversprechend erscheint, ist eine weiterführende Forschung in diesem Bereich notwendig. Beispielsweise könnte eine Implementierung mittels Hyper-Threading zeigen, wie effizient dieser Ansatz in der Praxis wirklich ist. Zusätzlich sollten weitere Experimente folgen, die zeigen welche Eingaben von der Nachrichtenüberlagerung am meisten profitieren.

Abbildungsverzeichnis

Algorithmenverzeichnis

8 Literatur

[1] Micah Adler u. a. „Communication-optimal parallel minimum spanning tree algorithms“. In: *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*. 1998, S. 27–36.

[2] Michael Axtmann u. a. „In-Place Parallel Super Scalar Samplesort (IPSSSSo)“. In: *25th Annual European Symposium on Algorithms (ESA 2017)*. Bd. 87. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 9:1–9:14. DOI: 10.4230/LIPIcs.ESA.2017.9.

[3] MohammadHossein Bateni u. a. „Affinity clustering: Hierarchical clustering at scale“. In: *Advances in Neural Information Processing Systems* 30 (2017).

[4] Otakar Borůvka. „O jistém problému minimálním“. In: (1926).

[5] Sun Chung und Anne Condon. „Parallel implementation of Bouvka's minimum spanning tree algorithm“. In: *Proceedings of International Conference on Parallel Processing*. IEEE. 1996, S. 302–308.

[6] Frank Dehne und Silvia Gotz. „Practical parallel algorithms for minimum spanning trees“. In: *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems (Cat. No. 98CB36281)*. IEEE. 1998, S. 366–371.

[7] Laxman Dhulipala, Guy E Blelloch und Julian Shun. „Theoretically efficient parallel graph algorithms can be fast and scalable“. In: *ACM Transactions on Parallel Computing (TOPC)* 8.1 (2021), S. 1–70.

[8] Daniel Funke u. a. „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.

[9] Charles AR Hoare. „Quicksort“. In: *The computer journal* 5.1 (1962), S. 10–16.

[10] Lorenz Hübschle-Schneider und Peter Sanders. „Linear Work Generation of R-MAT Graphs“. In: *Network Science* 8.4 (2020), S. 543–550.

[11] David R Karger, Philip N Klein und Robert E Tarjan. „A randomized linear-time algorithm to find minimum spanning trees“. In: *Journal of the ACM (JACM)* 42.2 (1995), S. 321–328.

[12] Mohsen Koohi Esfahani, Peter Kilpatrick und Hans Vandierendonck. „MASTIFF: structure-aware minimum spanning tree/forest“. In: *Proceedings of the 36th ACM International Conference on Supercomputing*. 2022, S. 1–13.

[13] Joseph B Kruskal. „On the shortest spanning subtree of a graph and the traveling salesman problem“. In: *Proceedings of the American Mathematical society* 7.1 (1956), S. 48–50.

[14] You Li und Xianrong Chang. „A MST-based and new GA supported distribution network planning“. In: *2011 International Conference on Mechatronic Science, Electric Engineering and Computer (MEC)*. IEEE. 2011, S. 2534–2538.

[15] Vitaly Osipov, Peter Sanders und Johannes Singler. „The filter-kruskal minimum spanning tree algorithm“. In: *2009 Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2009, S. 52–61.

[16] Robert Clay Prim. „Shortest connection networks and some generalizations“. In: *The Bell System Technical Journal* 36.6 (1957), S. 1389–1401.

[17] Peter Sanders und Matthias Schimek. „Engineering Massively Parallel MST Algorithms“. In: *arXiv preprint arXiv:2302.12199* (2023).

[18] Peter Sanders, Jochen Speck und Jesper Larsson Träff. „Two-tree algorithms for full bandwidth broadcast, reduction and scan“. In: *Parallel Computing* 35.12 (2009), S. 581–594.

[19] Peter Sanders u. a. *Sequential and Parallel Algorithms and Data Structures*. Springer, 2019.

[20] Leslie G Valiant. „A bridging model for parallel computation“. In: *Communications of the ACM* 33.8 (1990), S. 103–111.

[21] Jan Wassenberg, Wolfgang Middelmann und Peter Sanders. „An efficient parallel algorithm for graph-based image segmentation“. In: *International Conference on Computer Analysis of Images and Patterns*. Springer. 2009, S. 1003–1010.