Universität Paderborn
Fachbereich Mathematik/Informatik

Ein System zur automatischen Konfiguration effizienter paralleler Algorithmen im BSP-Modell

Diplomarbeit von Olaf Bonorden

vorgelegt bei
Prof. Dr. Friedhelm Meyer auf der Heide

Februar 2002

Übersicht

In dieser Arbeit wird ein System vorgestellt, wie verschachtelte Algorithmen im BSP-Modell modelliert werden können. Wenn zu jedem implementierten Algorithmus eine diesem Modell ensprechende Beschreibung seiner Eigenschaften (Laufzeit, Unterprobleme) mitgegeben wird, entsteht eine Sammlung von Algorithmen. Aus dieser Bibliothek kann nun ein Scheduler bei einem gegebenen Problem und für einen im BSP-Modell beschriebenen Computer einen effizienten Algorithmus zusammenstellen.

Neben dem Modell enthält diese Arbeit zahlreiche implementierte Algorithmen, mittels derer einerseits das Modell und die entstehenden Algorithmen auf Effizienz untersucht werden sollen. Andererseits soll dies zeigen, wie man Algorithmen und Beschreibungen dazu für dieses Modell erstellt.

Inhaltsverzeichnis

1. Einleitung **1**

2. Das BSP-Modell und seine Varianten **5**
2.1. Das reine BSP-Modell 5
2.2. Erweiterung um Paketgröße und Lokalität 8
2.2.1. Das BSP*-Modell 8
2.2.2. Das D-BSP-Modell 8
2.3. Das neue BSP-Unteraufruf-Modell 9
2.4. Die Experimentierplattform: Die PUB-Library 14

3. Evaluation des Unteraufruf-Modells **15**
3.1. Eingesetzte Parallelcomputer und Scheduler 15
3.1.1. Der benutzte Computer 15
3.1.2. Der Scheduler 17
3.2. Broadcast 17
3.2.1. Untersuchte Algorithmen 18
3.2.2. Laufzeitvorhersagen 20
3.2.3. Messungen 22
3.3. Berechnung Minimaler Spannbäume 26
3.3.1. Untersuchte Algorithmen 27
3.3.2. Implementierungen 31
3.3.3. Laufzeitvorhersagen 34
3.3.4. Messungen 37
3.4. Bewertung 41

4. Implementierungen **45**
4.1. Das BSP-Unteraufruf-Modell 45
4.1.1. Die Klasse „Problem“ 45
4.1.2. Die Klasse „Algorithm“ 46
4.1.3. Scheduler 48
4.2. Beispiel-Implementation Broadcast 49
4.2.1. Klasse BroadcastProblem 50

4.2.2. Algorithmus TreeBroadcast 52
4.3. Implementation Minimale Spannbäume 56
4.4. Probleme 56
4.4.1. Broadcast 56
4.4.2. ReduceAddInt, ReduceAllAddInt, ScanAddInt und ScanAll-AddInt 57
4.4.3. IntegerSort 57
4.4.4. Sort 58
4.4.5. MST 58
4.4.6. BorůvkaStep 59
4.4.7. MSTMerge 59
4.5. Algorithmen 60
4.6. Der Quelltext 60
4.7. Das Programm CallTreeEdit 63

5. Zusammenfassung und Ausblick **65**
5.1. Die Ergebnisse 65
5.2. Offene Probleme 65

Literaturverzeichnis **67**

Index **71**

Abbildungsverzeichnis

1. BSP-Modell . . . 6
2. Gültiges Schedule für Broadcast, 1024 Bytes, 16 Prozessoren . . . 13
3. Topologien PSC2 . . . 15
4. Veranschaulichung der Broadcast-Algorithmen . . . 18
5. Vorhersagen Broadcast PSC2, 16 Prozessoren . . . 21
6. Messungen Broadcast auf PSC2, 16 Prozessoren . . . 23
7. Genauigkeit der Vorhersagen, Broadcast auf PSC2, 16 Prozessoren . 25
8. Algorithmen und Probleme für minimale Spannbäume . . . 28
9. Verteilung der Kanten bei DenseBoruvkaStep . . . 33
10. Schedule für 4096 Knoten, 32768 Kanten, 16 Prozessoren . . . 35
11. Optimale Algorithmen für MST auf PSC2, TCP/IP (Vorhersage) . . 36
12. Optimale Algorithmen für MST auf PSC2, SCI (Vorhersage) . . . 36
13. Messungen MST PSC2, TCP/IP, 2 Prozessoren, 1024 Knoten . . . 39
14. Messungen MST PSC2, TCP/IP, 16 Prozessoren, 32768 Knoten . . . 40
15. Messungen MST PSC2, SCI, 2 Prozessoren, 1024 Knoten . . . 42
16. Messungen MST PSC2, SCI, 16 Prozessoren, 32768 Knoten . . . 43
17. Eingabe der Problemgröße . . . 63
18. Hauptfenster CallTreeEdit . . . 63

Tabellenverzeichnis

1. gemessene BSP*-Parameter des PSC2 . . . 16
2. Optimaler Broadcastalgorithmus auf PSC2 (Vorhersage) . . . 20
3. C++-Klassen des Modells . . . 46
4. Implementierte Probleme . . . 57
5. Implementierte Algorithmen . . . 60
6. Übersicht über die Quelltexte: BSP-Unteraufruf-Modell . . . 61
7. Übersicht über die Quelltexte: Broadcast- und Prefix-Algorithmen . 61

8. Übersicht über die Quelltexte: MST-Algorithmen 62
9. Übersicht über die Quelltexte: Sortier-Algorithmen 62

1. Einleitung

Betrachtet man die Entwicklung bei Hard- und Software der Computer in den letzten Jahrzehnten, so fällt auf, dass fast alle Computer nach den Prinzipien des Modells von *John von Neumann*[1] aufgebaut sind. Dies führt dazu, dass man all diese Computer gleich programmieren kann, wenn man die genaue Programmiersprache außer Acht lässt. Der gesamte Fortschritt der Hardware führt zu schnelleren und geschickteren Versionen von von-Neumann-Computern. Durch dieses über die Jahre stabile Modell entwickelten sich viele Algorithmen und Implementationen in Software. Viele Bibliotheken für verschiedene Einsatzbereiche erleichtern dem Programmierer heutzutage die Arbeit (z.B. LEDA für effiziente Datentypen und Algorithmen, s. [MN99]).

In letzter Zeit hat die Verbreitung von parallelen Computern mit mehreren Prozessoren stark zugenommen, und so stellt sich die Frage, ob es für diese Computer eine ähnliche Erfolgsgeschichte gibt. Wenn man parallele Rechner untersucht, stellt man fest, dass es mehrere verschiedene Architekturen gibt, die von der Programmierung her sehr unterschiedlich aussehen. Im wesentlichen gibt es zwei Arten *massiv paralleler* (d.h. mit großer Anzahl von Prozessoren) Computer:

- Netzwerke einzelner Knoten mittels eines Nachrichtensystems: Diese gibt es mit ganz verschiedener Hardware von preiswerten Standardtechniken (z.B. Ethernet) bis zu den Hochgeschwindigkeitsnetzwerken (z.B. Myrinet, SCI).
- Computer mit gemeinsamem Speicher: Alle Prozessoren greifen direkt auf einen gemeinsamen Speicher zu, das explizite Senden und Empfangen von Nachrichten ist nicht erforderlich (z.B. SGI Cray T3E).

Für beide Arten existiert eine Vielzahl von Modellen und noch mehr verschiedene inkompatible Softwareschnittstellen (z.B. MPI, PVM für Nachrichten, SHMEM, Yasmin für gemeinsamen Speicher). Um die Entwicklung und Analyse von parallelen Algorithmen zu vereinfachen, stellte Leslie G. Valiant 1990 ein Brückenmodel vor ([Val90]), das eine Verbindung zwischen der Hardware und der Software bilden sollte: Das *Bulk Synchronous Parallel Model*[2], kurz BSP-Modell.

[1] John von Neumann (1903-1957) entwickelte 1945 die Idee, Programme wie Daten im Arbeitsspeicher zu speichern, und revolutionierte damit die Softwareentwicklung, [Neu93].

[2] Beschreibung s. Kapitel 2.1

Programme für dieses Modell lassen sich ohne Änderung auf einer Vielzahl von verschiedenen Parallelcomputern ausführen, wie die BSP-Bibliotheken und Programmierwerkzeuge, z.B. BSPlib ([HMS+98]) und PUB ([pub]) zeigen.

Seit 1990 wurden eine große Anzahl von Problemen im BSP-Modell untersucht und viele BSP-Algorithmen angegeben. Trotzdem sind BSP-Programme in der Praxis nicht weit verbreitet. Woran scheitert der Einsatz? Die Angabe eines schnellen Algorithmus in BSP-Notation und der Einsatz eines solchen in einem realen Programm als kleine Unterroutine sind sehr unterschiedliche Problemstellungen. Neben vielen softwaretechnischen Problemen (sequentiell kann man einfach verschiedene Objekt-Dateien zusammenlinken, parallel muss man sich auf eine BSP-Implementation einigen) gibt es auch die Frage, auf welchen Prozessoren des Computers der Unteralgorithmus laufen soll und wie die Daten verteilt sind. Außerdem existiert nicht der schnellste Algorithmus für alle Computer, da das BSP-Modell einige hardwareabhängige Dinge (z.B. Netzwerkbandbreite) als Parameter enthält. Es stellt sich also auch die Frage, welcher Algorithmus auf einem bestimmten Rechner am schnellsten ist, und auf wie vielen und welchen Prozessoren er ausgeführt werden soll.

Neben der Auswahl der Algorithmen ist oft die Wahl von Algorithmen-Parametern wichtig. Ein Beispiel hierfür ist ein sequentieller Sortieralgorithmus, Quicksort, der für rekursive Aufrufe entweder wieder Quicksort, oder bei Folgen der Länge kleiner als ein Parameter k Bubblesort benutzt. Der Wert des optimalen k ist hier abhängig von den Eigenschaften des Computers, auf dem der Algorithmus zum Einsatz kommt.

Eine ähnliche Fragestellung tritt auch beim Hardware/Software-Codesign auf. Bednara et al. untersuchen in [BBTW00] die Kombination des Algorithmus MergeSort mit einem in Hardware realisierten systolischen InsertSort. Hierbei müssen zusätzlich zu der Laufzeit des zusammengesetzten Algorithmus auch die Kosten für die Hardware berücksichtigt werden; gesucht ist ein Tradeoff zwischen Laufzeit und Kosten.

In der Arbeit [Eil98] von Eilinghoff steht die Wiederverwendung von implementierten Algorithmen mittels Werkzeugsystemen im Vordergrund. So genannte Experten entwerfen hier die Algorithmen und konstruieren ein Werkzeugsystem, mit dem der Anwender für seinen Computer einen optimalen Algorithmus erzeugt. Das Werkzeugsystem generiert hierbei den neuen Algorithmus als Quelltext durch Auswahl von Lösungskomponenten aus einer Anwendungsbibliothek. Diese Werkzeugsysteme sind jeweils für einen bestimmten Anwendungsbereich wie Branch-And-Bound oder Sortieren erstellt und nicht allgemein anwendbar.

Diese Diplomarbeit stellt ein System vor, das die Entscheidung des optimalen Algorithmus trifft und die Algorithmen-Parameter optimal wählt. Dazu wird ein Modell angegeben, welches parametrisierte BSP-Algorithmen mit Unterproblemen formalisiert und ein Kostenmaß für die Laufzeit solcher Algorithmen definiert (Kapitel 2.3). Dieses System nutzt als Basis eine einfach zu erweiternde Sammlung von

BSP-Algorithmen für verschiedene Probleme inklusive einer Laufzeit-Beschreibung im BSP-Modell. Als Anwender kann man nun ein Problem beschreiben und dieses zusammen mit den hardwareabhängigen Parametern eines gewünschten Zielcomputers dem System als Eingabe geben. Daraufhin wird ein Schedule berechnet, das angibt, welcher Algorithmus mit welchen Parametern und Unteralgorithmen dieses Problem auf der spezifizierten Hardware am schnellsten löst.

Neben dem Modell für das System und einer Implementierung liegt der Schwerpunkt auf der Evaluation anhand verschiedener Problemstellungen. Zuerst wird das relativ einfache Problem des Broadcasts (Verteilung von Daten von Prozessor 0 an alle anderen) untersucht, anschließend mit der Berechnung minimaler Spannbäume von ungerichteten Graphen ein komplexeres Beispiel.

Durch Messungen auf verschiedenen Computern soll untersucht werden, wie effizient dieses System ist und wie genau Vorhersagen im BSP-Modell sind. Dazu werden zusätzlich zu dem reinen BSP-Modell auch einige Erweiterungen betrachtet (insbesondere Lokalität-berücksichtigende Modelle), diese sind in Kapitel 2.2 beschrieben.

2. Das BSP-Modell und seine Varianten

Wenn man in der heutigen Zeit von „Computer“ spricht, denken die meisten Menschen an einen PC. Auch die weniger verbreiteten Systeme wie Apple Macintosh oder Unix Workstations sind hinsichtlich des Aufbaus sehr ähnlich. In der Welt der Parallelcomputer ist dies anders: Es gibt eine Vielzahl verschiedener Computer-Systeme, die sich nicht nur in Art und Anzahl der Prozessoren unterscheiden, sondern vor allem in der Art des Verbindungsnetzwerkes. Einige bieten gemeinsamen Speicher, andere tauschen Daten mittels Nachrichten über Verbindungsleitungen aus. Des Weiteren können auch die Topologien der Verbindungen sehr unterschiedlich sein, verbreitet sind Gitter, Torus (z.B. PSC2, s. Abbildung 3), Butterfly-Netzwerk, Clique (z.B. Netzwerke mit zentralem Switch) und andere.

Um nicht jeden Rechner einzeln untersuchen zu müssen, wurden verschiedene Modelle vorgeschlagen, die einen Computer hinreichend genau beschreiben sollen, ohne alle Details zu berücksichtigen. Dieses Kapitel beschreibt eines dieser Modelle, das BSP-Modell sowie die Erweiterungen um blockweise Kommunikation (BSP*) und Lokalität (D-BSP). Anschließend wird das in dieser Arbeit vorgestellte System zur Modellierung von verschachtelten BSP-Algorithmen definiert und beschrieben.

Verwandte Modelle sind *Coarse Grained Multicomputer* (CGM, s. [DFRC93]), *Hierarchical Parallel Random Access Machine* (H-PRAM, s. [HR92]) und LogP (s. [CKP+93]).

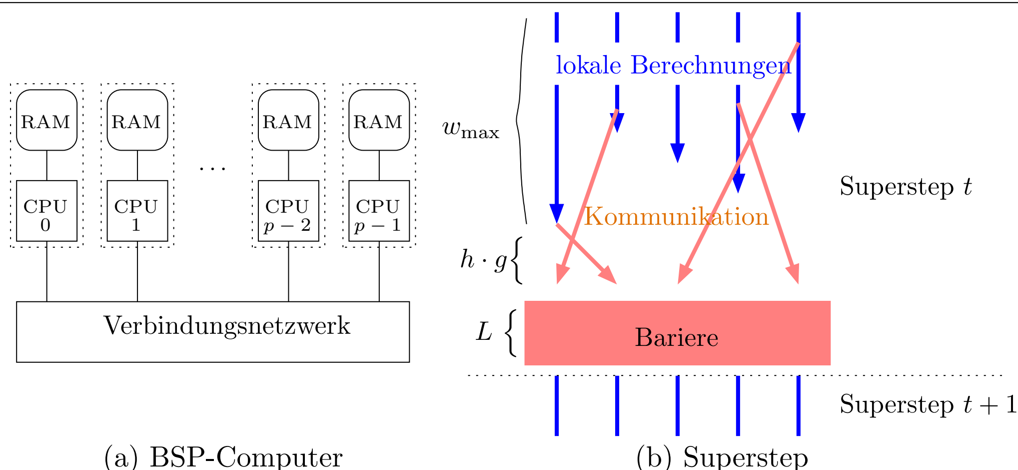
2.1. Das reine BSP-Modell

Das *Bulk-Syncronous Parallel (BSP)-Modell* wurde in [Val90] als Brücke zwischen Hard- und Software vorgestellt. Es soll – ähnlich dem von-Neumann-Modell für sequentielle Computer – eine Verbindung zwischen verschiedenen Soft- und Hardwarelösungen sein.

Im BSP-Modell besteht jeder Computer – von Valiant auch *Bulk-Synchronous Parallel Computer (BSPC)* genannt – aus drei Teilen (s. Abbildung 1 (a)):

- p Komponenten mit Berechnungs- und/oder Speicherfunktion. In dieser Arbeit sind diese Komponenten meist Workstations oder einzelne Knoten von Parallelcomputern mit Prozessor und Speicher.

Abbildung 1 BSP-Modell



(a) BSP-Computer (b) Superstep

- Ein Netzwerk mit einem Routingmechanismus, welcher Punkt-zu-Punkt-Verbindungen erlaubt, d.h. jede Komponente kann jeder beliebigen anderen Komponente Nachrichten schicken. Weitergehende Operationen, wie z.B. das Verschicken von Nachrichten an mehrere Ziele gleichzeitig, sind nicht möglich.

- Ein Mechanismus zur Synchronisation der Komponenten.

In [Val90] sind verschiedene Arten der Synchronisation beschrieben. Diese unterscheiden sich im Wesentlichen nur bei der Sichtweise des BSP-Modells, bei Analysen der Laufzeit liegt zwischen den einzelnen Sichtweisen maximal ein kleiner konstanter Faktor.

In dieser Arbeit wird die Sicht gewählt, die der Implementierung eines BSP-Computers mittels der PUB-Library[1] am Nächsten kommt. Ein Algorithmus im BSP-Modell besteht aus zeitlichen Abschnitten, *Supersteps* genannt, die durch Synchronisationen getrennt sind. Innerhalb eines Supersteps kann jeder Prozessor mit seinen lokalen Daten rechnen oder Nachrichten an andere Prozessoren versenden. Anschließend ruft er eine Synchronisationsfunktion auf. Sobald alle Prozessoren diese Synchronisation gestartet haben, wird gewartet, bis alle Nachrichten ihr Ziel erreicht haben. Anschließend wird der nächste Superstep begonnen. Erst in diesem Superstep stehen die im vorherigen Schritt gesendeten Nachrichten zur Verfügung. Ein Beispiel für einen Superstep zeigt Abbildung 1 (b).

Obwohl dieses Modell lediglich nachrichtenbasierte Parallelcomputer unterstützt, können auch Computer mit gemeinsamen Speicher modelliert werden, da der Austausch von Nachrichten mit gemeinsamen Speicher effizient zu implementieren ist

[1] Paderborn University BSP-Library, s. Kapitel 2.4

(z.B. PUB-Library auf SHMEM, [BJHR00a]). Außerdem unterstützen einige Implementierungen auch Funktionen, um direkt in den Speicher anderer Prozessoren zu schreiben. Um der Semantik des BSP-Modells zu entsprechen, werden diese Schreibzugriffe bis zur nächsten Synchronisation verzögert. Damit BSP-Programme kompatibel bleiben, werden solche Zugriffe auf nachrichtenbasierten Computern durch Nachrichten simuliert.

Das BSP-Modell ist nicht nur ein Modell für Programmierer und Computerhersteller, es dient darüberhinaus der Analyse von Algorithmen. Dazu definiert es ein *Kostenmaß*, um Laufzeiten abhängig von wenigen maschinenabhängigen Parametern vorherzusagen. Diese BSP-Maschinenparamter sind:

- Die *Anzahl der Prozessoren* p.
- Die *Bandbreite* des Netzwerkes, die angibt, wie viele lokale Operation in der Zeit ausgeführt werden können, die das Verschicken eines Wortes benötigt.
- Die Zeit L, die für die Synchronisation benötigt wird.
- Um Laufzeiten auch in Sekunden angeben zu können, wird manchmal noch ein weiterer Parameter s angegeben. s ist ein Maß für die *CPU-Leistung* und gibt die Zeit in Sekunden an, die durchschnittlich eine lokale Operation benötigt.

Hiermit lassen sich sowohl Laufzeiten exakt vorhersagen, als auch Schranken im O-Kalkül angeben.

Die Gesamtlaufzeit eines BSP-Algorithmus ist die Summe der Laufzeiten aller seiner Supersteps. Die Länge eines einzelnen Supersteps ist die Summe von drei Teilen:

- Die maximale *lokale Arbeit* eines Prozessors $w_{\max} := \max\{w_i; i \in \{1, \ldots, p\}\}$, wobei w_i die Anzahl an Rechenschritten in Prozessor i ist.
- *Kommunikationskosten*: Wenn jeder Prozessor maximal h Wörter empfängt und maximal h Wörter sendet, so ist $h \cdot g$ eine obere Schranke für die Kommunikationskosten. g ist hierbei ein Parameter des Computers und beschreibt die Bandbreite des Netzwerkes.
- Kosten für die *Synchronisation* L.

Die Kosten eines Supersteps sind dann die Summe $w_{\max} + hg + L$. Dies ist eine worst-case-Abschätzung, es wird nicht berücksichtigt, dass das Versenden der Nachrichten eventuell zeitgleich mit der lokalen Berechnung geschehen kann, falls die Nachricht nicht am Ende des Supersteps abgeschickt wird und der Computer das asynchrone Versenden von Nachrichten erlaubt.

2.2. Erweiterung um Paketgröße und Lokalität

2.2.1. Das BSP*-Modell

In der Praxis hat sich gezeigt, dass das BSP-Modell nicht immer sehr gute Vorhersagen der Laufzeit ermöglicht. Insbesondere berücksichtig es nicht, dass bei vielen Parallelrechnern und vor allem bei Clustern von Computern die maximale Bandbreite erst erreicht wird, wenn die Nachrichten eine bestimme Größe haben. Bei kleinen Nachrichten ist der Overhead des Systems zu groß. Dieser Overhead entsteht vor allem durch Software (Zusammenstellen von Nachrichten, Funktions- und Betriebssystemaufrufe) sowie durch Protokolle, die Nachrichten mit zusätzlichen Headern versehen (Absender, Prüfsummen). Falls mehrere solche Protokolle zum Einsatz kommen (z.B. PUB auf TCP/IP), kann der Overhead bei kleinen Nachrichten erheblich sein.

Um dieses zu modellieren, wurde in [BDM98] als Erweiterung das BSP*-Modell beschrieben. In diesem Modell gibt es einen zusätzlichen Parameter B, der die minimale Blockgröße angibt, ab der die Kommunikation effizient abläuft. D.h., es wird nicht nur gezählt, wie viele Daten jeder Prozess sendet und empfängt, sondern auch untersucht, wie die einzelnen Nachrichten aussehen. Dabei wird jede Nachricht, die kleiner als B Bytes ist, wie eine Nachricht der Länge B gezählt.

2.2.2. Das D-BSP-Modell

Das BSP-Modell berücksichtigt keine Lokalität. Das heißt, eine Nachricht von Prozessor 0 zu Prozessor 1 benötigt genau so lange, wie eine Nachricht zu Prozessor 1000. Außerdem ist es nicht möglich, nur Teile des Computers zu synchronisieren. So ist das parallele Ausführen zweier Algorithmen auf Teilen des Computers nur sehr schwierig zu realisieren, da beide die gleichen Synchronisationen ausführen müssen.

Das D-BSP-Modell (*decomposible*, [DK96]) behebt diese Probleme. Es erlaubt, den BSP-Computer in zwei gleich große Teilcomputer, auch Partitionen genannt, aufzuteilen. Diese reagieren dann wie zwei autonome BSP-Computer, erlauben es, Nachrichten innerhalb ihrer Prozessormenge zu verschicken und diese zu synchronisieren. Kommunikation der beiden Teile miteinander ist dagegen nicht möglich. Erst nachdem beide die Aufteilung wieder beenden, steht erneut ein gemeinsames Netzwerk zur Verfügung.

Diese Aufteilung ist auch rekursiv möglich, so dass aus p Prozessoren Partitionen mit Größen von $\frac{p}{2^i}, i = 1, \ldots, \lfloor \log_2 p \rfloor$ Prozessoren entstehen können. Innerhalb der kleineren BSP-Computer gelten andere BSP-Parameter. So werden aus den Konstanten g und L Funktionen $g : \mathbb{N} \to \mathbb{Q}$ und $L : \mathbb{N} \to \mathbb{Q}$, die zu jeder Größe der Partitionen die entsprechenden BSP-Maschinen-Parameter angeben.

Diese Erweiterung für L ist auf vielen BSP-Computern sinnvoll, da die Synchronisation oft mittels Nachrichten realisiert wird. Die PUB-Library berechnet dazu z.B.

in einer parallelen Prefix-Operation, wie viele Nachrichten jeder Prozessor empfangen soll. Daher ist hier meistens $L(p) = O(\log p)$. Die Abhängigkeit der Funktion g von p ist nicht auf allen Rechnern erkennbar, da z.B. Workstation-Cluster meist auf einem zentralen Switch basieren und daher eine Stern-Topologie besitzen. So ist die Distanz zweier Prozessoren im Netzwerk stets die gleiche. Auch bei dem in Kapitel 3.1.1 beschriebenen SCI-Cluster gibt es nur zwei mögliche Distanzen (gleicher SCI-Ring oder Routing über einen weiteren Ring erforderlich).

Die PUB-Library geht noch einen Schritt weiter und erlaubt nicht nur die Halbierung des BSP-Computers, sondern eine beliebige Aufteilung in beliebig große Partitionen. Allerdings müssen diese Partitionen Intervalle bilden, so ist z.B. eine Aufteilung in eine Partition mit geraden Prozessor-Ids und eine mit ungeraden nicht erlaubt (Funktion bsp_partition der PUB-Library, s. [BJHR00b]). So eine Aufteilung würde auch der Idee der Ausnutzung von Lokalität entgegensprechen, da die Prozessoren einer Partition dann nicht mehr lokal benachbart wären. Eine Aufteilung in Partionen eignet sich sehr gut, um auf Teilen des Parallelrechners einen Unteralgorithmus zu starten, da jede Partition eine eigene Nummerierung der Prozessoren beginnen bei 0 bekommt, so dass jede Partition wie ein eigenen Comuter wirkt.

2.3. Das neue BSP-Unteraufruf-Modell

Das BSP-Unteraufruf-Modell benutzt eine Kombination aus dem BSP*- und dem D-BSP-Modell, um die Hardware zu modellieren. Außerdem definiert das Modell, wie Algorithmen beschrieben werden, d.h., ein Algorithmus in diesem Modell besteht aus dem eigentlichen Programmcode und einer Beschreibung der Laufzeit und der erzeugten Unterprobleme.

Desweiteren spezifiziert das Modell, wie ein Problem gelöst werden kann, in dem der Begriff Schedule definiert wird. Ein Schedule für ein Problem auf einem bestimmten Rechner gibt an, mit welchem Algorithmus es gelöst wird. Dabei wird neben dem Hauptalgorithmus zu jedem erzeugten Unterproblem angegeben, welcher Algorithums es löst und auf wie vielen Prozessoren er ausgeführt werden soll.

Um Algorithmen in diesem Modell untersuchen zu können, muss zu jedem implementierten Algorithmus eine Beschreibung vorliegen. Diese enthält Information über folgende Dinge:

- Problem, welches der Algorithmus löst.
- Anzahl der Prozessoren, auf denen der Algorithmus ausgeführt werden kann. Nicht alle dieser Prozessoren müssen wirklich benötigt werden. Allerdings stehen für alle Unterprobleme maximal so viele Prozessoren zur Verfügung, wie der Hauptalgorithmus an dieser Stelle angibt.

- Anzahl der möglichen Varianten. Viele Algorithmen erlauben mehrere Varianten bzw. sind von Parametern abhängig; z.B. kann bei einer baumförmigen Kommunikationsstruktur der Grad des Baumes gewählt werden. Diese verschiedenen Wahlen werden in diesem Modell durch verschiedene Varianten des Algorithmus unterstützt.
- Laufzeit des Algorithmus. Bei dieser Laufzeitangabe wird die Laufzeit von Unterproblemen nicht berücksichtigt.
- Beschreibung über die erzeugten Unterprobleme

Bis auf den ersten Punkt sind diese Angaben jeweils abhängig von einer Beschreibung der Eingabedaten, beim Sortieren z.B. die Anzahl der zu sortierenden Daten und deren Größe. Formal lässt sich eine Algorithmusbeschreibung wie folgt definieren:

Definition (Algorithmusbeschreibung): Seien $\mathit{Eingabebeschr} := \mathbb{Q}^*$, *Probleme* eine beliebige Menge.

Eine *Algorithmusbeschreibung* ist ein 6-Tupel $A = (p, p_{count}, v_{count}, t, r_{count}, r)$ mit

- $p \in \mathit{Probleme}$
  p ist das Problem, welches der Algorithmus löst.
- $p_{count} : \mathit{Eingabebeschr} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}^* : (n, p, p_{max}) \mapsto p_{count}(n, p, p_{max})$ mit $p_{count}(n, p, p_{max}) \subseteq \{p, p+1, \ldots, p_{max}\}$
  p_{count} gibt an, mit wie vielen Prozessoren die Berechnung möglich ist, falls die Eingabedaten mit Beschreibung n auf p Prozessoren verteilt liegen und p_{max} Prozessoren maximal für die Berechnung zur Verfügung stehen.
- $v_{count} : \mathit{Eingabebeschr} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N} : (n, p, c) \mapsto v_{count}(n, p, c)$
  v_{count} gibt an, wie viele Varianten des Algorithmus es gibt. Dabei sind n die Beschreibung der Eingabe, p die Anzahl der Prozessoren, auf denen die Eingabe verteilt liegt und c die Anzahl der für die Berechnung benutzten Prozessoren.
- $t : \mathit{Eingabebeschr} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{Q} : (n, p, c, v) \mapsto t(n, p, c, v)$
  t beschreibt die Laufzeit des Algorithmus ohne Unteralgorithmen im BSP*-Modell. n, p und c analog zu oben, v gibt an, welche Variante des Algorithmus gewählt werden soll.
- $r_{count} : \mathit{Eingabebeschr} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N} : (n, p, c, v) \mapsto r_{count}(n, p, c, v)$
  Diese Funktion gibt an, wie viele verschiedene Unteralgorithmen aufgerufen werden. Für jeden Aufruf ergibt die Funktion s Informationen über Art und Anzahl der Aufrufe. Die Parameter n, p, c, v beschreiben wie oben, mit welchen Daten und Prozessoren welche Variante des Algorithmus aufgerufen wird.

- $r : \begin{cases} \mathit{Eingabebeschr} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathit{Probleme} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \\ (n, p, c, j, v) \mapsto (s_p, s_n, s_d, c_s, c_p) \end{cases}$

  Die Funktion s beschreibt die einzelnen Unteraufrufe. Als Parameter bekommt sie zusätzlich zu der Beschreibung der Eingabedaten noch den Index j, der angibt, zu welchem Aufruf die Beschreibung gehört. Die Elemente des Bildes haben folgende Bedeutung: s_p Problem, s_n Datengröße des Unterproblems, s_d Anzahl der Prozessoren, auf denen die Daten verteilt sind, c_s Anzahl der sequentiell auszuführenden Aufrufe des Unteralgorithmus, c_p gibt an, wie oft das Problem gleichzeitig entsteht. Dabei muss trivial $s_d \cdot c_s \leq c$ gelten.

  Es reicht aus, falls r nur auf Eingaben

  $$\{(n, p, c, j, v); n, p, c, v, j \in \mathbb{N}, c \geq p, v \leq v_{count}(n, p, c), j \leq r_{count}(n, p, c, v)\}$$

  definiert ist.

Es gilt dann: Algorithmus A *löst* das Problem p.

Beispiel (TreeBroadcast): Bei dem Problem Broadcast ist die Aufgabe, Daten von Prozessor 0 zu allen anderen zu schicken. Eine Veranschaulichung des hier beschriebenen Algorithmus Tree zeigt Kapitel 3.2.1, in Kapitel 4.2 wird eine Implementierung angegeben.

Da die Eingabebeschreibung für dieses Problem nur aus einer natürlichen Zahl (Größe der Daten in Bytes) besteht, ist bei diesem Problem die Funktion $\max : \mathit{Eingabebeschr} \times \mathbb{Q} \to \mathbb{Q}$ wohldefiniert. Eine Algorithmusbeschreibung für das Problem Broadcast sieht wie folgt aus (Broadcast als binärer Baum):
BinTreeBroadcast:= $(\mathit{Broadcast}, p_{count}, v_{count}, t, r_{count}, r)$ mit

- $p_{count}(n, p, p_{max}) := \{p\}$

  Anzahl erlaubter Prozessoren ist immer die Anzahl der Zielprozessoren.
- $v_{count}(n, p, c) := 1$

  Es gibt nur eine Variante des Algorithmus.
- $t(n, p, c, v) := \lceil \log_2 p \rceil (2 \cdot \max(n, B(p)) \cdot g(p) + L(p))$

  Jede Ebene des Baumes benötigt einen Superstep, jeder Prozessor verschickt je n Daten an seine beiden Söhne, der Baum hat Tiefe $\lceil \log_2 p \rceil$.
- $r_{count}(n, p, c, v) := 0$

  Unterprobleme treten in dieser Implementation nicht auf.
- r ist die auf keiner Eingabe definierte Funktion.

Alternativ ist auch eine Definition mit Unteralgorithmen möglich, hier wird das Beispiel für einen Baum mit beliebigem Baumgrad (verschiedene Baumgrade werden als Varianten des Algorithmus definiert) angegeben. Eine Veranschaulichung der Kommunikationsstruktur und der erzeugten Unterprobleme zeigt Abbildung 4 (a), Seite 18.

TreeBroadcast:= $(Broadcast, \bar{p}_{count}, \bar{v}_{count}, \bar{t}, \bar{s}_{count}, \bar{s})$ mit

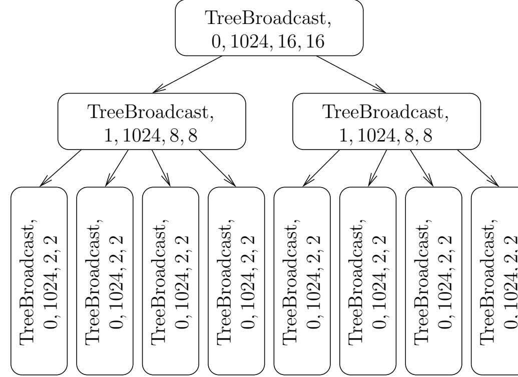
- $\bar{p}_{count}(n, p, p_{max}) := \{p\}$
- $\bar{v}_{count}(n, p, c) := |\{d \in \{2, \ldots, p\}; d|p\}|$
  Varianten sind alle möglichen Baumgrade d. Es gilt:
  $$d \text{ ist erlaubter Baumgrad} \Leftrightarrow d \text{ teilt } p$$
- Sei d die v-te natürliche Zahl mit $d \geq 2$ und $d|p$ (d ist der Baumgrad der Variante v).
  $\bar{t}(n, p, c, v) := \max(n, B(p)) \cdot (d-1) \cdot g(p) + L(p)$
  Nur die BSP-Zeit für die obereste Ebene im Baum, die d Teilbäume werden als Unterproblem erledigt.
- d sei wie oben definiert.
  $\bar{r}_{count}(n, p, c, v) := \begin{cases} 0 & \text{falls } p = d \\ d & \text{sonst} \end{cases}$
  Falls mehr als d Prozessoren beteiligt sind, existiert ein Unterproblem, nämlich Broadcast auf den d Partitionen der Größe p/d.
- d sei wie oben definiert.
  $\bar{r}(n, p, c, j, v) := (Broadcast, n, p/d, 1, d)$
  Das Unterproblem Broadcast hat die gleiche Datenbeschreibung, soll diese Daten auf p/d Prozessoren verteilen und wird einmal nacheinander, aber d mal parallel aufgerufen.

Die genaue Implementierung des Problems Broadcast und des Algorithmus TREE-BROADCAST zeigt Kapitel 4.2.

Definition (Gültiges Schedule): Sei P eine Menge von Problemen, A eine Menge von Algorithmusbeschreibungen. Dann ist ein Baum S ein *gültiges Schedule* für ein Problem $q \in P$ mit Eingabebeschreibung n verteilt auf p Prozessoren auf einem hierarischen BSP-Computer mit c-Prozessoren, falls gilt:

(i) An jedem Knoten u von S steht ein Tupel $(A_u, v_u, n_u, p_u, c_u)$ mit $A_u = (\bar{p}, \bar{p}_{count}, \bar{v}_{count}, \bar{t}, \bar{r}_{count}, \bar{r}) \in A$ Algorithmusbeschreibung, v_u Variante des Algorithmus mit $v_u \leq \bar{v}_{count}(n_u, p_u, c_u)$, n_u Datenbeschreibung, p_u Anzahl der Datenprozessoren und c_u Anzahl der Rechenprozessoren.

Abbildung 2 Gültiges Schedule für Broadcast, 1024 Bytes, 16 Prozessoren



(ii) Für die Wurzel u gilt: A_u löst Problem q, $n_u = n$, $p_u = p$, $c_u \leq c$.

(iii) Für jeden Knoten u mit Beschriftung $(A_u, v_u, n_u, p_u, c_u)$ und $A_u = (p', p_{count}, v_{count}, t, r_{count}, r)$ gilt: $r'_{count}(n', p', p_{count}, v_u)$ ist die Anzahl der Söhne von v, der j–ten Sohn mit $r(n_v, p_v, c_v, j, v_u) = (s_{p_j}, s_{n_j}, s_{d_j}, c_{s_j}, c_{p_j})$ ist mit $(A_j, v_s, s_{n_j}, s_{d_j}, c_j)$ beschriftet, mit $A_j \in A$ löst s_p. Außerdem gilt

$$\forall j \in \{1, \ldots, r_{count}(n_u, p_u, c_u)\} : s_{p_j} c_{p_j} \leq c_u$$

D.h. die Söhne jedes Knotens u sind mit Algorithmenbeschreibungen und Datengrößen beschriftet, die die Unterprobleme des Algorithmus A_u beschreiben und es werden für Unterprobleme niemals mehr Prozessoren benutzt wie für den Algorithmus am Knoten u.

Ein Beispiel für ein gültiges Schedule für das Problem Broadcast von 1024 Bytes auf 16 Prozessoren zeigt Abbildung 2.

Definition (Kosten eines Schedules): Sei S ein gültiges Schedule für ein Problem mit Eingaben der Größe n auf p Prozessoren für einen c-Prozessor-BSP-Computer.

Sei $u = (A_u, v_u, n_u, p_u, c_u)$ ein Knoten von S mit einer Algorithmusbeschreibung $A_u = (p', p'_{count}, v'_{count}, t', r'_{count}, r')$ und Söhnen $w_1 = (A_{w_1}, v_{w_1}, n_{w_1}, p_{w_1}, c_{w_1})$ bis $w_k = (A_{w_k}, v_{w_k}, n_{w_k}, p_{w_k}, c_{w_k})$.

Für $j = 1, \ldots, k$ sei c_{s_j} die vierte Komponente von $r'(n_u, p_v, c_v, j, v_u)$, d.h. c_{s_j} gibt an, wie oft Unterproblem j nacheinander auftritt.

Dann sind die *Kosten eines Teilbaumes* mit Wurzel u definiert als:

- Falls $k = 0$, d.h. u hat keine Söhne, so ist $t(u) := t'(n_u, p_u, c_u, v_u)$.
- Sonst ist $t(u) := t'(n_u, p_u, c_u, v_u) + \max \left\{ t(u_j) \cdot c_{s_j} \middle| j = 1, \ldots, k \right\}$.

Die *Kosten des Schedules S* sind die Kosten der Wurzel des Schedules.

In Kapitel 4.1.3 wird ein Algorithmus angegeben, der für eine gegebene Eingabebeschreibung zu einem Problem und einer Menge von Algorithmenbeschreibungen ein gültiges Schedule mit minimalen Kosten berechnet. Dazu werden mittels einer Tiefensuche alle gültigen Schedules gesucht.

2.4. Die Experimentierplattform: Die PUB-Library

Es existieren verschiedene Software-Bibliotheken, um Programme nach dem BSP-Modell zu implementieren. Am weitesten verbreitet ist die BSPlib (s. [HMS+98]), die auch zur Definition eines Standards für BSP-Bibliotheken führte (s. [bsp]).

In dieser Arbeit wurde die *Paderborn University BSP* (PUB)-Library (s. [pub], [BJvOR99]) benutzt. PUB ist in der Sprache C geschrieben, unterstützt aber auch die Programmierung in C++. In PUB implementierten wir den BSPlib-Standard, unterstützen aber viele weitere Funtionen wie oblivious Synchronisation, parallele Präfix-Operationen, unabhängige BSP-Computer, Threads, virtuelle Prozessoren und Untergruppen. Insbesondere das Aufteilen des BSP-Computers in mehrere, unabhängige Partitionen, die sich wieder wie eigene BSP-Computer verhalten, ist in dieser Arbeit wichtig, da so jeder Unteralgorithmus unabhängig von anderen Unteralgorithmen ausgeführt werden kann. Außerdem unterstützt PUB auch die Abfrage der BSP-Maschinen-Parameter in Abhängigkeit der Partitionsgröße.

Ein weiterer Vorteil von PUB ist die aktive Weiterentwicklung und die einfache Installation, während die aktuelle Version der BSPlib bereits 1999 erschien.

3. Evaluation des Unteraufruf-Modells

3.1. Eingesetzte Parallelcomputer und Scheduler

Dieser Abschnitt beschreibt die für die Messungen benutzten Parallelrechner und Workstationcluster und den Scheduler, der automatisch optimale Algorithmen konfiguriert.

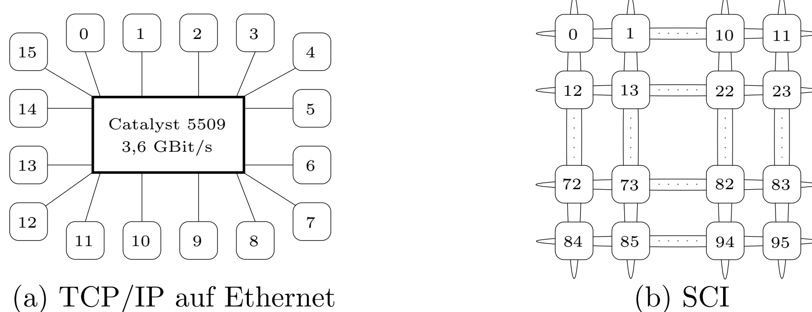
3.1.1. Der benutzte Computer

Der Rechner PSC2 des Paderborner Centers for Parallel Computing (s.a. [Pad]) ist aus der Produktreihe hpcLine der Firma Siemens und besteht aus 96 Knoten. Jeder dieser Knoten ist ein Dualprozessor-System mit zwei Pentium 850 MHz Prozessoren und 512 MB Hauptspeicher. Als Betriebssystemsoftware ist Linux installiert. Alle Knoten sind mittels zweier verschiedener Netzwerke (s. Abbildung 3) verbunden, für die jeweils eine eigene Variante von PUB genutzt werden kann:

Fast Ethernet: Alle Knoten besitzen eine 100-MBit-Ethernetkarte und sind an einem Switch (Cisco Catalyst 5509, Bandbreite 3,6 BBit/s) angeschlossen. PUB benutzt in dieser Konfiguration TCP/IP mittels der Socket-Funktionen des Linux-Betriebssystems.

Da der Switch nur eine begrenzte Bandbreite hat, kann man für Messungen nicht alle Knoten gleichzeitig benutzen, da dann der Ethernet Switch Pake-

Abbildung 3 Topologien PSC2



(a) TCP/IP auf Ethernet (b) SCI

Tabelle 1 gemessene BSP*-Parameter des PSC2

p	$s[1/ns]$	B	L	g
2	3,814	3092	377210	30,71
4	4,024	1467	585486	30,27
8	4,008	264	564087	156,99
16	4,167	239	636177	148,28

(a) TCP/IP

p	$s[1/ns]$	B	L	g
2	3,889	316	8174	8,34
4	3,889	509	17468	9,65
8	3,889	389	24452	11,29
16	3,889	21	35312	50,02

(b) MPI mit SCI

te löscht. TCP erkennt nach einem Timeout, dass Pakete fehlen und sendet diese erneut. Dies führt jedoch zu drastischen Einbrüchen der Bandbreite, so dass der Cluster in diesem Zustand nicht sinnvoll genutzt werden kann. Daher beschränken sich alle Messungen in dieser Konfiguration auf 16 Knoten. Die PUB Library stellt eine künstliche Beschränkung der Sende-Bandbreite zur Verfügung, diese Beschränkung wurde jedoch hier nicht benutzt.

Auf Grund der sternförmigen Topologie gibt es keine unterschiedlichen Distanzen zwischen einzelnen Knoten.

SCI: Jeder Knoten ist mittels einer Dolphin SCI in zwei SCI-Ringen integriert, die einen zweidimensionalen 12×8 Torus bilden. Jeder Ring erreicht eine Bandbreite von ca. 500 MB/s, die sich allerdings alle Knoten des Ringes teilen. PUB benutzt hier die von der Firma Scali gelieferte MPI-Version SCAMPI.

So liefert dieses System Daten für zwei verschiedene Computer-Arten, einmal ein Workstationnetz mit entsprechend hohen Latenzzeiten (daraus resultiert ein großes L im BSP-Modell) oder als ein Parallelcomputer mit niedriger Latenz und sehr hoher Bandbreite. Die von der PUB Library gemessenen BSP*-Werte zeigt Tabelle 1.

Durch die exklusive Nutzung sind störende Einflüsse durch andere Nutzer weitestgehend ausgeschlossen. Allerdings läuft auf jedem Knoten ein komplettes Linux Betriebssystem, so dass einige Prozesse im Hintergrund aktiv sind. Dies führt zu nicht ganz deterministischen Zeiten bei Messungen. Ein weiteres Problem ist, dass sich sowohl den Switch als auch die SCI-Ringe eventuell mehrere Benutzer teilen.

Alle Messungen auf diesem Computer wurden mindestens fünf Sekunden lang mit der gleichen Eingabe wiederholt, um Messungenauigkeiten zu minimieren, die z.B. durch Caches oder Verzögerungen beim erstmaligen Anfordern von Speicher entstehen können. Jeder Prozessor bestimmt die Zeit je Durchlauf, anschließend wird über alle Prozessoren das Maximum dieser Zeiten berechnet. Die im folgenden Kapitel angegebenen Messwerte sind jeweils Mittelwerte über alle Durchläufe dieser Maxima.

3.1.2. Der Scheduler

Zur automatischen Konfiguration des optimalen Algorithmus wurde ein einfacher Scheduler implementiert. Dieser ist in Kapitel 4.1.3 beschrieben. Er führt eine Tiefensuche aus und durchsucht den gesamten Raum gültiger Schedules. Obwohl die maximale Laufzeit des Schedulers exponentiell in der Größe der Algorithmenbeschreibungen und der Anzahl der Prozessoren sein kann, dauerte die Berechnung für die in dieser Arbeit untersuchten Probleme und Eingabegrößen maximal wenige Sekunden. Da hier nur Offline-Verfahren berücksichtigt werden, d.h. das Schedule wird komplett vor der Ausführung des Algorithmus berechnet, ist die Laufzeit des Schedulers selber hier nicht entscheidend.

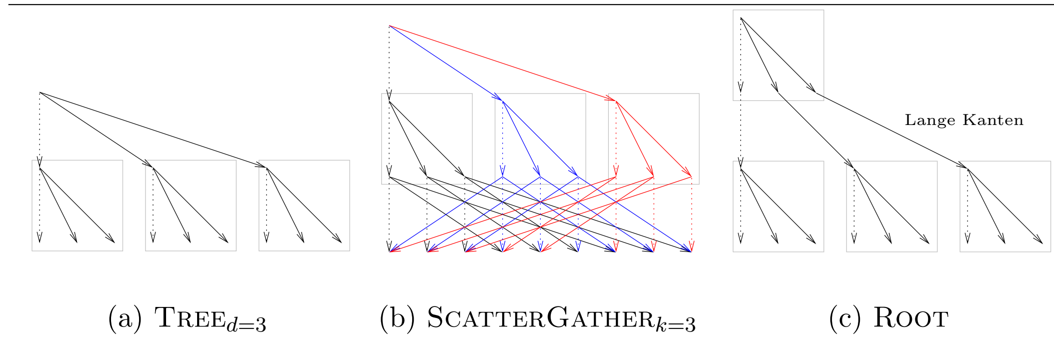
3.2. Broadcast

Das Problem *Broadcast* ist eine der grundlegenden Kommunikationsaufgaben, die in sehr vielen Algorithmen benötigt wird. Ein Prozessor besitzt Daten, die an alle andere Prozessoren geschickt werden sollen. Diese Operation ist in verschiedenen Parallelrechner-Modellen (z.B. BSP-Modell [JW96a], H-PRAM [JKMR00]) theoretisch umfassend analysiert. Außerdem ist sie in fast allen Kommunikationsbibliotheken (z.B. MPI, PUB) direkt verfügbar. In dieser Arbeit wird nur der Fall betrachtet, dass Prozessor 0 die Daten an alle anderen schickt.

Die Evaluation mittels Broadcast soll zeigen, wie genau das Modell bei einer einfachen Anwendung ist und ob es in der Praxis einen Vorteil bringt, verschiedene Broadcast-Algorithmen zu kombinieren. In der BSP-Bibliothek PUB sind alle hier vorgestellten Broadcastalgorithmen implementiert. In der Initialisierungsphase werden für jeden Broadcast-Algorithmus anhand einiger gemessener Maschinenparameter die optimale Konfiguration der Algorithmen berechnet. Hierfür wird ein etwas erweitertes BSP-Modell benutzt. Anschließend werden diese Algorithmen mit unterschiedlichen Nachrichtengrößen auf verschiedenen Prozessoranzahlen ausgemessen, um später im laufenden Betrieb für jede Nachrichtengröße und für jede Prozessoranzahl den optimalen Broadcast Algorithmus zu benutzen. Details und Messwerte hierzu finden sich in [Rie00].

In PUB werden ausschließlich reine Algorithmen benutzt, d.h. es wird keine Kombination verschiedener Algorithmen betrachtet. Allerdings können die Messwerte nicht direkt mit den Versuchen dieser Arbeit verglichen werden, da PUB intern für Broadcasts keine Synchronisationen benutzt und daher effizienter als jede BSP-Lösung implementiert ist.

Abbildung 4 Veranschaulichung der Broadcast-Algorithmen



(a) $\text{Tree}_{d=3}$ (b) $\text{ScatterGather}_{k=3}$ (c) Root

3.2.1. Untersuchte Algorithmen

Eine Veranschaulichung der hier untersuchten Algorithmen zeigt Abbildung 4. Die einzelnen Prozessoren sind horizontal angeordnet, die Zeit verläuft vertikal von oben nach unten. Jeder Pfeil steht für eine Nachricht. Die farbigen Kanten beim Scatter-Gather-Broadcast sollen die verschiedenen Teile der Nachricht verdeutlichen. Die gestrichelt gezeichneten Pfeile gehören zum Broadcast-Baum, werden aber natürlich nicht als Nachricht geschickt, da sie auf den selben Prozessor zeigen.

Für jeden der folgenden Algorithmen kann man eine Algorithmusbeschreibung nach Kapitel 2.3 angeben. Für den Algorithmus Tree steht diese Beschreibung formal als Beispiel in Kapitel 2.3, die Beschreibung der anderen Algorithmen ist den Quelltexten zu den Algorithmen zu entnehmen (s.a. Tabellen 7 bis 9).

Pure Algorihthmen

TREE. Der Algorithmus Tree ist die einfachste Art zur Lösung des Broadcast-Problems. Alle Prozessoren sind als Baum angeordnet, Prozessor 0 bildet die Wurzel. In jedem Superstep schickt jeder Prozessor, der die Daten bereits erhalten hat, diese an seine Kinder weiter. Der einzige Parameter ist der Baumgrad d. Eine Beschreibung im BSP-Modell ist in [GV92] zu finden.

In dem hier benutzten Modell stellt sich dies wie folgt dar: Der Algorithmus benutzt genau die Prozessoren, zu denen die Daten geschickt werden sollen, und führt nur eine Runde aus: Schicke Daten an $d-1$ Kinder. Anschließend teile die Maschine in d Teile und löse auf jedem Teil das Unterproblem Broadcast. Bei p Prozessoren sind die Teiler von p die möglichen Baumgrade. Also gibt es $|\{i \in \{2, ..., p\}; i \text{ teilt } p\}|$ Varianten dieses Algorithmus. Die Laufzeit ohne die Unterprobleme beträgt $n \cdot g(1) + (d-1) \cdot \max(n, B(p)) \cdot g(p) + L(p)$ bei n Bytes Daten und p Prozessoren.

Falls der Grad gleich der Anzahl der Prozessoren ist, ergibt dies einen Baum der Tiefe eins. Dann wird dieser Algorithmus auch Liste genannt, da Prozessor 0 die

Daten der Reihe nach an alle anderen schickt.

Eine Algorithmusbeschreibung nach dem BSP-Unteraufruf-Modell zeigt das Beispiel auf Seite 12.

SCATTERGATHER. Beim TREE wird das Netzwerk relativ schwach ausgelastet, da immer nur die Prozessoren Daten senden, die schon alle Informationen haben, dies sind vor allem in den ersten Runden nur sehr wenige. In den BSP-Modellen spielt es aber für die Kosten keine Rolle, wie viele Prozessoren gleichzeitig kommunizieren. Der SCATTERGATHER-Algorithmus (in [JW96a] 2D-BCAST genannt) zerteilt Nachrichten in k kleinere Blöcke und sendet diese an jeweils p/k Prozessoren. Anschließend verteilen diese in einer Art All-To-All Operation ihre Daten an alle anderen Prozessoren.

Zuerst entsteht folglich das Teilproblem Broadcast mit n/k Daten und p/k Prozessoren (k-mal parallel), anschließend sendet jeder Prozessor seinen Block an k andere Prozessoren. Die Varianten des Algorithmus ergeben sich aus den möglichen Wahlen von k, dies hängt sowohl von Prozessoranzahl als auch von der Datengröße ab.

Die Kosten sind $2\lceil n/l \rceil \cdot g(1) + 2k \cdot \max(n, B(p)) \cdot g(p) + L(p)$ ohne Berücksichtigung der Unteralgorithmen. Falls k kein Teiler von p ist, erhöhen sich die Kosten um $(p - p \bmod k) \cdot \max(n, B(p)) \cdot g(p) + L(p)$.

ROOT. Der Algorithmus ROOT berücksichtigt Lokalität, indem er versucht, alle langen Kanten des Broadcastbaumes in einem Superstep von verschiedenen Prozessoren auszuführen. Er ist in [JKMR00] beschrieben und arbeitet wie folgt: Zuerst werden die Daten an $\sqrt{p}$ Prozessoren geschickt, anschließend sendet jeder dieser Prozessoren i sie an Prozessor $i\sqrt{p}$, so dass in jedem $\sqrt{p}$ Intervall der Prozessoren der jeweils erste die Daten hat. Anschließend wird in jedem dieser Intervalle ein Broadcast gestartet.

Im Modell bedeutet dies, dass es zwei Unterprobleme gibt, zuerst einen Broadcast zu den $\sqrt{p}$ Prozessoren und am Ende in jedem Intervall einen weiteren Broadcast. Die Laufzeit ohne diese Unterprobleme beträgt $n \cdot g(1) + \max(n, B(p)) \cdot g(p) + L(p)$.

Optimaler Algorithmus

Der Begriff optimaler Algorithmus ist von der Eingabegröße und den BSP-Maschinen-Parametern abhängig. Für jede Wahl der Parameter liefert ein Schedule mit minimalen Kosten einen eventuell aus mehreren puren Algorithmen zusammengesetzten, im BSP-Unteraufruf-Modell optimalen Algorithmus. In dieser Arbeit wurde das Schedule mit dem in Kapitel 4.1.3 beschriebenen Algorithmus bestimmt.

Tabelle 2 Optimaler Broadcastalgorithmus auf PSC2 (Vorhersage)

Datengröße[a]	Algorithmus
1 – 128	LISTE
256 – 2048	rekursive Anwendung von $\text{TREE}_{d=4}$
4096 – 32768	$\text{SCATTERGATHER}_{k=16}$
65536	ROOT mit ROOT und $\text{TREE}_{d=3}$ bzw. $\text{TREE}_{d=2}$
128 kByte – 1 MB	ROOT mit $\text{SCATTERGATHER}_{k=4}$ und $\text{TREE}_{d=3}$

(a) TCP/IP

Datengröße	Algorithmus
1 – 64	TREE_{d_8}, als Unteralgorithmus TREE_{d_2}
128	rekursiv $\text{TREE}_{d=4}$
256 – 1024	TREE mit Baumgraden 2, 4 und wieder 2
2048 – 1 MB	$\text{TREE}_{d=2}$ mit $\text{SCATTERGATHER}_{k=8}$

(b) SCI

[a] Die Intervalle geben nicht die genauen Grenzen an, da hier nur Zweierpotenzen untersucht wurden

3.2.2. Laufzeitvorhersagen

In diesem Abschnitt werden die vorhergesagten Laufzeiten folgender Algorithmen verglichen:

TREE: Dies enspricht der rekursiven Anwendung von TREE, untersucht wurden hier die Baumgrade $d = 2$ und $d = 4$.

LISTE: Prozessor 0 sendet die Daten in einem Superstep nacheinander an alle anderen Prozessoren, dies entspricht dem TREE-Algorithmus mit einem Grad $d = p$ bei p Prozessoren.

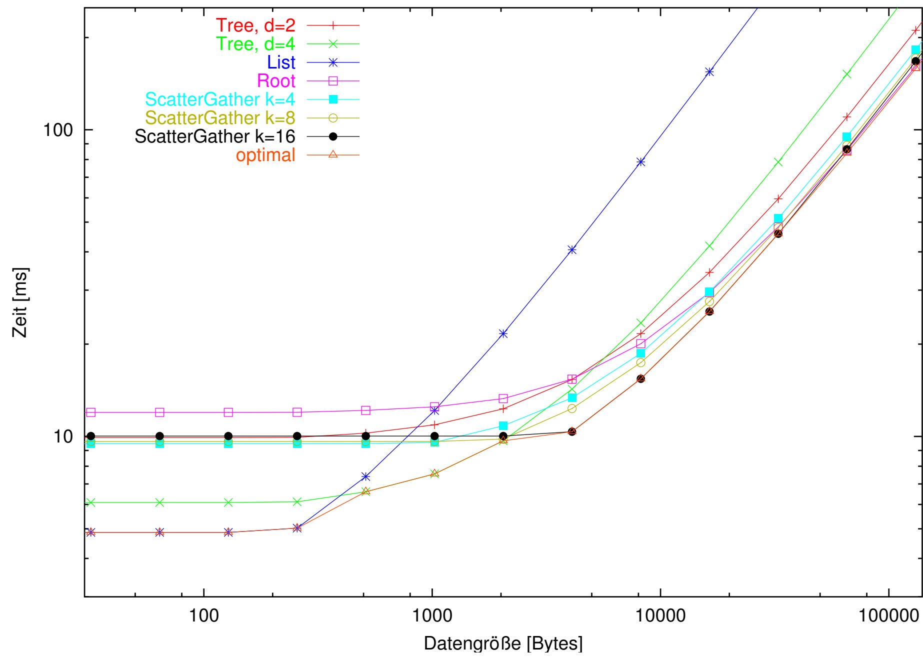
ROOT: Rekursive Anwendung von ROOT für $p > 2$, TREE für $p = 2$.

SCATTERGATHER: Mit verschiedenen Werten für k, als Unteralgorithmus für den Teilbroadcast wird TREE mit Grad $d = p/k$ benutzt.

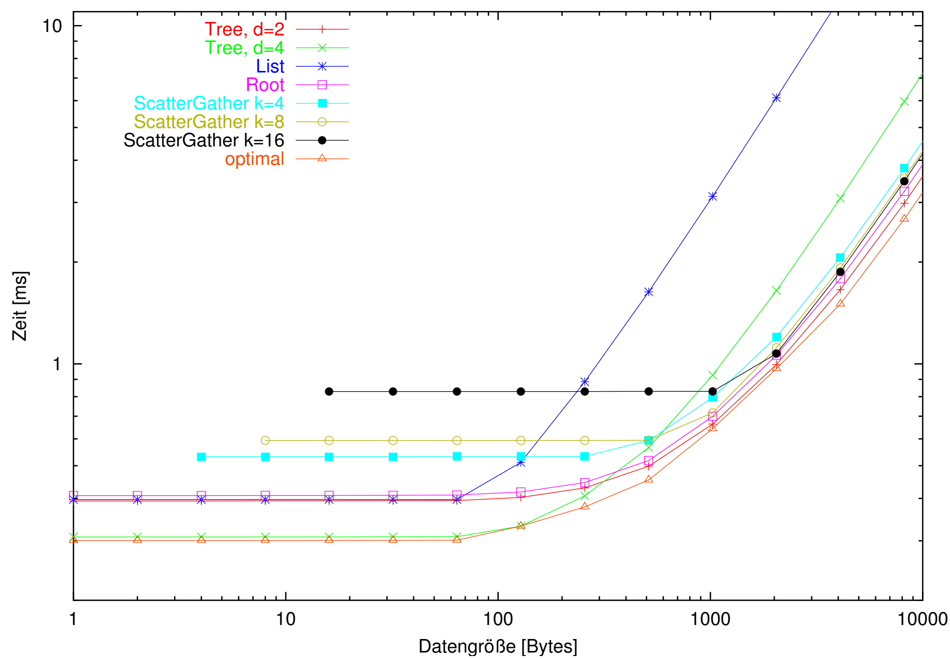
Optimal: Der im BSP-Unteraufruf-Modell optimal zusammengesetzte Algorithmus. Dieser ist das Ergebnis der automatischen Konfiguration und ist in Tabelle 2 in Abhängigkeit von der Datengröße beschrieben.

Einen Überblick über die mit den BSP*-Parametern des PSC2 mit 16 Prozessoren vorhergesagten Laufzeiten zeigt die Abbildung 5.

Abbildung 5 Vorhersagen Broadcast PSC2, 16 Prozessoren



(a) TCP/IP



(b) SCI

TCP/IP. Im Bereich kleiner Nachrichten unter 1kB Größe dominiert die Synchronisation die Laufzeit, da L sehr groß ist ($L(16) = 636177$, auch für kleinere Prozessorzahlen ist L immer noch sehr groß, $L(2) = 377210$). Daher ist der schnellste Algorithmus List (Tree, $d = p = 16$), da dieser nur aus einem Superstep besteht. Bei großen Nachrichten ist Root am schnellsten, da hier ein Teil der Arbeit auf $\sqrt{p}$ Prozessoren ausgeführt wird. Bei den für diese Vorhersagen benutzten BSP*-Parametern ist $g(16) = 148,28$ deutlich größer als $g(\sqrt{16}) = g(4) = 30,27$. Da der Unterschied zwischen $g(4)$ und $g(2)$ nur minimal ist, wird als Unteralgorithmus $\text{ScatterGather}_{k=4}$ benutzt, eine weitere Anwendung von Root bringt keinen Vorteil mehr.

Diese Untersuchungen zeigen, dass der optimale Algorithmus nicht unbedingt eine Reinform eines der Standardalgorithmen ist, sondern dass eine Kombination zweier Algorithmen bei bestimmten Nachrichtengrößen schneller sein kann. Allerdings ist der Vorteil hier nur maximal $1,28\%$ (Root/ScatterGather-Kombination gegenüber rekursivem Root bei 1 MB Daten).

SCI. Mit SCI sind g und L deutlich kleiner als bei TCP/IP. Vor allem das Verhältnis von L/g ist geringer, so dass hier nicht so oft die Synchronisation die Laufzeit bestimmt. Schon bei kleinen Nachrichten sind zwei Synchronisationen deutlich schneller, da dadurch Prozessor 0 nicht so viele Daten verschicken muss. Der optimale Algorithmus benutzt hier einen Baum mit Grad 8 bzw. Grad 2. Dies ist etwas schneller, als beide Male Grad 4 zu verwenden, da PUB bei den Parametermessungen für vier Prozessoren eine etwas höheren Wert erhalten hat ($B(4) = 509$ und $B(8) = 389$).

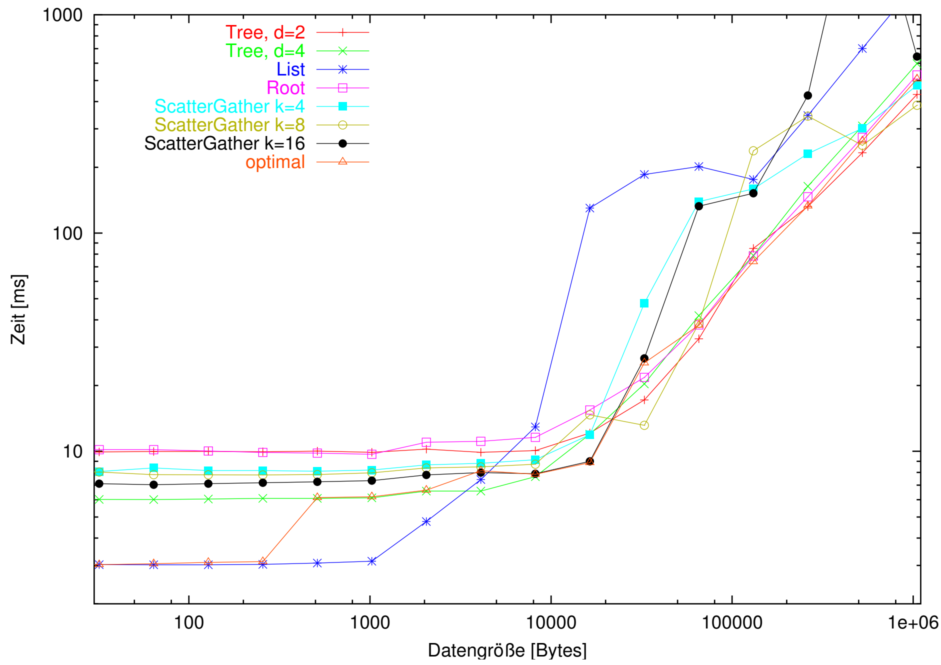
Bei großen Nachrichten zeigt sich auch hier, dass eine Kombination verschiedener Algorithmen im Modell etwas schneller ist als die Verwendung eines reinen Algorithmusses. Allerdings ist hier die Verbindung von $\text{Tree}_{d=2}$ und $\text{ScatterGather}_{k=8}$ am effizientesten.

Performance-Untersuchungen bei der Entwicklung der PUB-Library haben die schon in Kapitel 3.1.1 erwähnten Probleme des Netzwerkes bei dem Computer PSC2 mit TCP/IP gezeigt. Insbesondere durch die verschiedenen Belastungen des Netzwerkes, die die unterschiedlichen Algorithmen erzeugen, kann bei einem größeren Kommunikationsvolumen keine genaue Vorhersage der Laufzeiten erwartet werden.

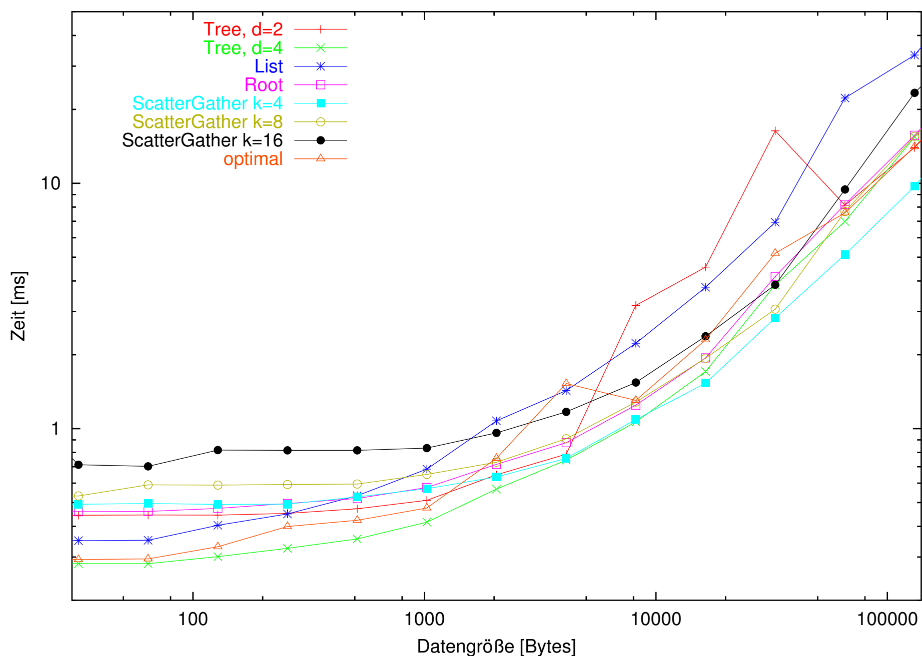
3.2.3. Messungen

Wenn man die Laufzeit der einzelnen Algorithmen ausmisst (s. Abbildung 6, zur Messmethode s. Kapitel 3.1.1), fällt auf, das der im Modell optimale Algorithmus keinesfalls immer der schnellste ist. Die Gründe für diese in der Realität nicht optimalen Konfigurationen beschreiben die beiden folgenden Abschnitte.

Abbildung 6 Messungen Broadcast auf PSC2, 16 Prozessoren



(a) TCP/IP



(b) SCI

TCP/IP. Bei kleinen Nachrichten bis 256 Bytes ist in der Vorhersage wie in der Realisierung der Algorithmus LIST der schnellste, da er nur eine Runde benötigt und die Laufzeit von der Synchronisation dominiert wird. Der Parameter L, der angibt, wie lange eine Synchronisation benötigt, entspricht sehr gut der Praxis, da er in PUB durch Ausmessen wiederholter Synchronisationen erhalten wird.

Bei größeren Nachrichten wird die Laufzeit insbesondere von LIST, aber auch der anderen Algorithmen überschätzt. Hier zeigt sich die Ungenauigkeit des BSP*-Modells für Rechner wie den PSC2. BSP* modelliert die Bandbreite des Systems in einem Parameter g. Dieses g ist im Modell eine Konstante und nicht abhängig von der Auslastung des Netzwerkes. Der Cluster PSC2 ist mittels eines einigen Switches verbunden. Dieser Switch hat eine Backplane mit einer bestimmten Bandbreite, die sich alle Knoten im Netzwerk teilen müssen (Details s. Kapitel 3). Dies bewirkt, dass g auf diesem Rechner sehr stark von der aktuellen Netzwerkauslastung abhängt. Die PUB-Library versucht, ein worst-case-g zu liefern, indem sie einen *Total-Exchange*[1] ausführt und die Zeit dabei misst, d.h. das von PUB gelieferte g ist bei Vollauslastung des Netzwerkes gemessen.

Bei den hier benutzten Algorithmen senden die meiste Zeit nur wenige Prozessoren gleichzeitig Daten (bei LIST sogar nur ein Prozessor), so dass hier das Netzwerk nur minimal belastet ist und es nicht zu Engpässen beim Switch kommt. Um diese Einflüsse besser berücksichtigen zu können, könnte man die Erweiterung E-BSP ([JW96b]) des BSP-Modells untersuchen. Weitere mögliche Erweiterungen des Modells und andere offene Fragen zeigt Kapitel 5.2 auf.

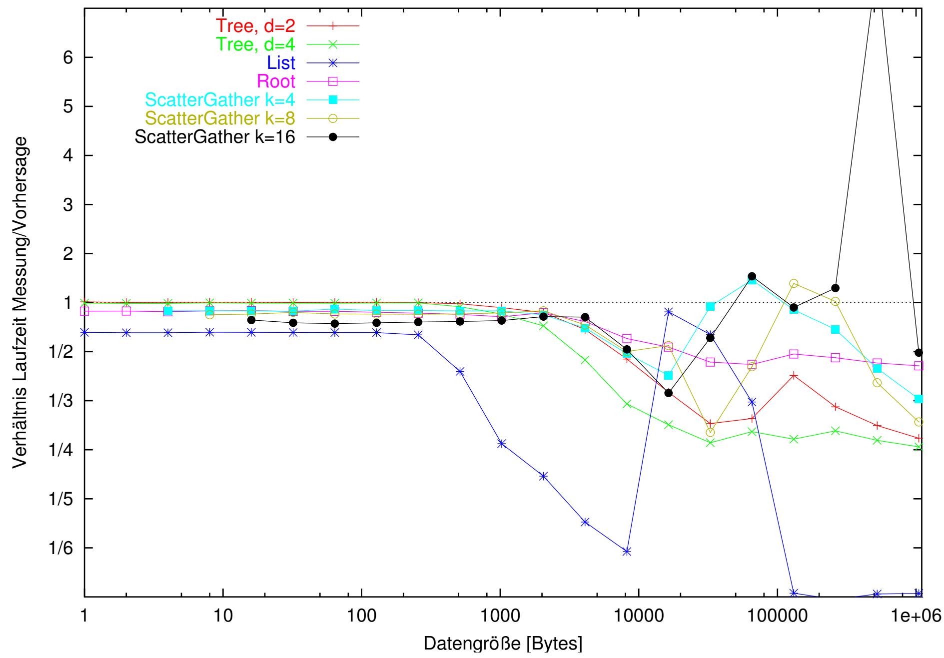
Abbildung 7 gibt eine Übersicht über die Genauigkeit des Modells bei den hier untersuchten Broadcast-Algorithmen. Sie zeigt das Verhältnis aus gemessener und vorhergesagter Laufzeit. Bei Werten größer als eins benötigt der Algorithmus auf der realen Maschine länger als das Modell angibt. Um sowohl Über- als auch Unterschätzungen der Laufzeit gleich darzustellen, wurde im Bereich unterhalb von eins eine logarithmische Skala gewählt. So bedeutet ein Datum eine Skaleneinheit oberhalb der eins, dass der Algorithmus doppelt so lange benötigt wie vorhergesagt, eine Skaleneinheit unterhalb der eins bedeutet, dass genau das Gegenteil, die Vorhersage gibt die doppelte Zeit relativ zur Ausführungszeit.

SCI. Analog zu TCP/IP werden auch die Laufzeiten bei der Kommunikation mittels SCI bei großen Nachrichten überschätzt. Außerdem zeigen sich gerade bei SCI starke Schwankungen in der Laufzeit, insbesondere bei den Messungen zu $\text{TREE}_{d=2}$ im Bereich von 8192 Bytes bis 32 kBytes. Solche nichtdeterministischen Laufzeitschwankungen machen eine genau Vorhersage mit PUB auf SCI fast unmöglich.

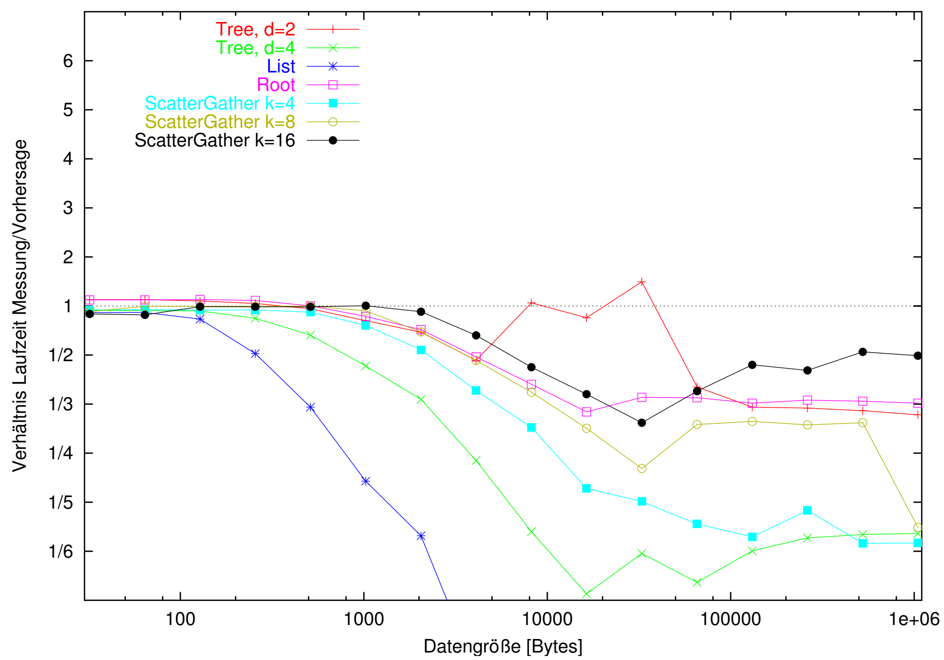
Außerdem kam es während der Messungen ab und zu zu Instabilitäten auf dem System, vom Hängenbleiben des Programmes bis zum Totalabsturz einzelner Kno-

[1] jeder Knoten sendet Daten an jeden anderen Knoten

Abbildung 7 Genauigkeit des Modells, Broadcast auf PSC2, 16 Prozessoren, Erläuterungen zu den Kurven und der Skalierung der Achse s. Text Seite 24



(a) TCP/IP



(b) SCI

ten. Wie bereits in [Rie00] festgestellt wurde, muss PUB auf PSC2 mit SCI wohl auch zum Zeitpunkt dieser Messungen noch als *experimental* bezeichnet werden.

3.3. Berechnung Minimaler Spannbäume

In diesem Kapitel wird die Berechnung minimaler Spannbäume untersucht. Ziel ist es, aus vorhandenen Algorithmen für den entsprechenden BSP-Computer und für jede Eingabegröße optimale Algorithmen zur parallelen Berechnung solcher Spannbäume zu konfigurieren. Diese Anwendung besitzt im Gegensatz zu den bisher untersuchten Problemen keine so regelmäßige Struktur. Für das Modell bedeutet dies, dass ohne Kenntnis der genauen Eingabedaten für die Unterprobleme meist nur eine Abschätzung der Eingabegrößen möglich ist, da diese Größen nicht nur von der Eingabebeschreibung (z.B. Anzahl der Kanten eines Graphen), sondern auch von den Daten selber abhängen. Sogar die Anzahl benötigter Unterprobleme kann eingabesensitiv sein. Dies ergibt neben eventuell ungenauer Vorhersagen auch das Problem, dass es vielleicht nicht sinnvoll ist, ein Schedule genau auszuführen, da einige Unterprobleme – anders als vorhergesagt – in der Realität bei bestimmten Eingaben nicht auftreten.

Für die Vorab-Aussagen ist es erforderlich, immer die maximale Anzahl an Unterproblemen (mit maximalen Eingabegrößen) abzuschätzen. Falls nicht alle Unterprobleme bei der Ausführung mit bestimmten Eingaben auftreten, können die Aufrufe der Unteralgorithmen übersprungen werden. Es dürfen aber niemals mehr Aufrufe gebraucht werden, als das Schedule vorgibt, da dies zu einem Programmabbruch führt, da dann zur Lösung einiger Unterprobleme keine Algorithmen angegeben sind.

Definition: Gegeben sei ein ungerichteter Graph $G = (V, E)$ mit $n = |V|, m = |E|$ und Kantengewichten $w : E \to \mathbb{Q} : e \mapsto w(e)$. Ein Baum $T = (V, F)$ mit $F \subseteq E$ und $|F| = |V| - 1$ heißt *Spannbaum* von G. Das Gewicht $w(T)$ eines Spannbaumes ist definiert als

$$w(T) := \sum_{e \in F} w(e).$$

T heißt genau dann *Minimaler Spannbaum*, wenn für alle Spannbäume T' gilt: $w(T) \leq w(T')$.

Die hier untersuchten Algorithmen bauen den minimalen Spannbaum Schritt für Schritt auf, indem sie Kanten des Spannbaumes auswählen und die verbleibenden Knoten zu neuen Superknoten zusammenfassen. Für die Laufzeitanalyse wesentlich ist dabei, wie viele Kanten in jedem Schritt gewählt werden. Dies hängt allerdings von dem Eingabegraphen ab, so dass hier nur die Abschätzung für den ungünstigsten Fall betrachtet werden kann.

Letztgenanntes ist eine Schwäche des Modells, da alle Abschätzungen der Laufzeiten offline im Vorfeld passieren und nur auf einer Beschreibung der Eingabedaten basieren. Genauer wäre es, die optimale Strategie online bei jedem Unterprogrammaufruf basierend auf den wirklichen Daten neu anzupassen. Allerdings müssen dann auch die Kosten für die Optimierung bei jeder Ausführung berücksichtigt werden.

3.3.1. Untersuchte Algorithmen

Alle hier betrachteten Algorithmen bauen den minimalen Spannbaum auf, indem sie minimale Kanten suchen, die noch nicht zusammenhängende Knoten verbinden. Sie unterscheiden sich in der Art, wie diese Kanten gefunden werden. Außerdem müssen die parallelen Algorithmen, die mehrere Kanten gleichzeitig auswählen, sicherstellen, dass der konstruierte Spannbaum keine Kreise enthält, die entstehen können, wenn eine Kante von zwei Konten gleichzeitig gewählt wird.

Die im Folgenden beschriebenen Algorithmen bauen im Wesentlichen auf drei Grundalgorithmen auf:

(i) Sequentielle Berechnung eines minimalen Spannbaumes. Hierzu wurde der Algorithmus von Kruskal implementiert.

(ii) Reduktion der Anzahl der zu betrachtenden Knoten, indem Kanten des minimalen Spannbaumes gewählt und die dadurch gebildeten Zusammenhangskomponenten als Superknoten zusammengefasst werden. Dies ist hier das so genannte Problem BorůvkaStep, hierfür wurden zwei verschiedene Algorithmen (BoruvkaStep und DenseBoruvkaStep) implementiert.

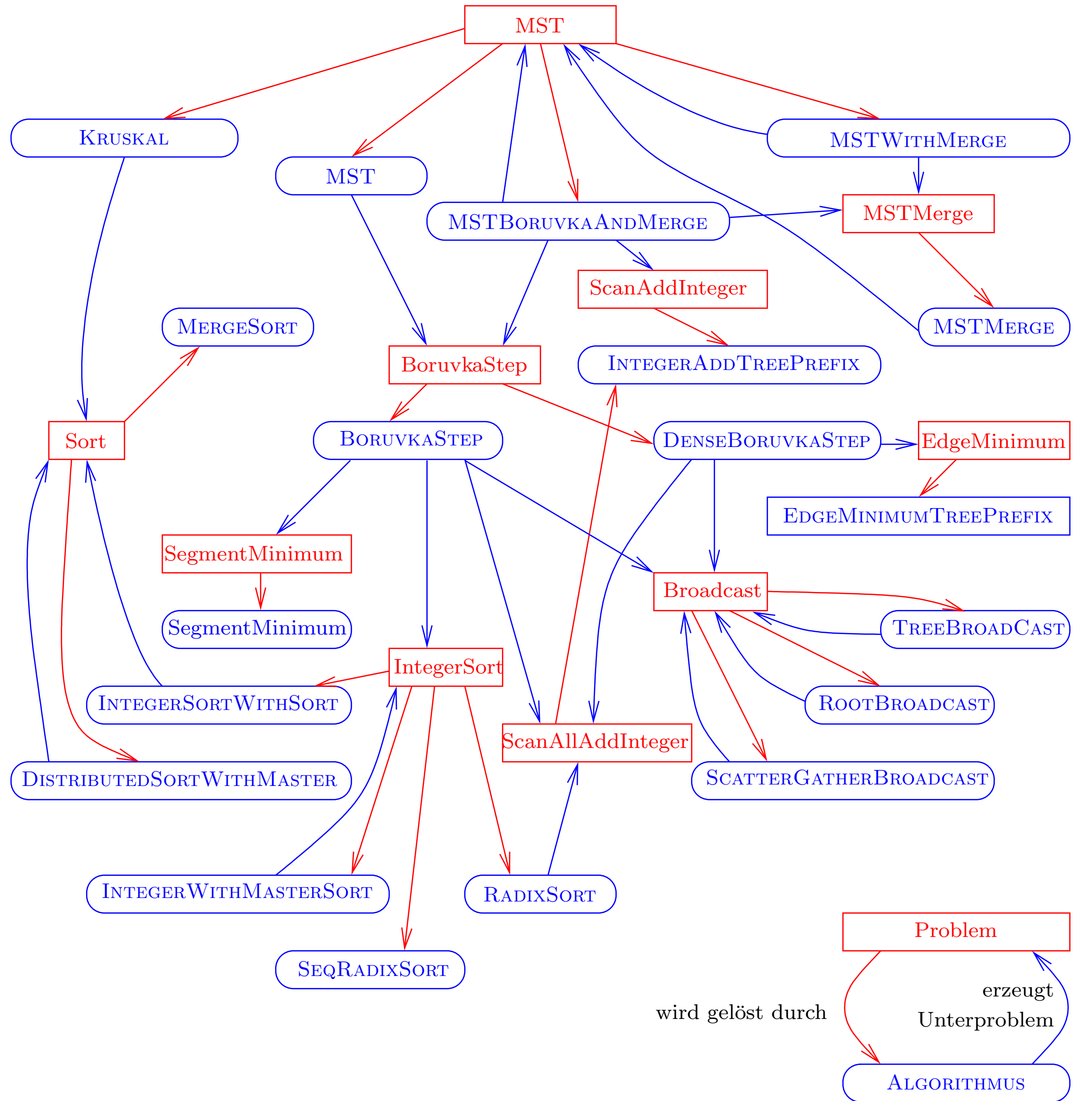
(iii) Reduktion der Anzahl der Kanten. Dazu werden lokal berechnete minimale Spannbäume gesammelt und vereinigt, dieses Problem heißt MSTMerge und ist durch einen Algorithmus implementiert.

Eine Übersicht der implementierten Algorithmen inklusive aller Unterprobleme zeigt Abbildung 8.

Kruskal

Dieser sequentielle Algorithmus von Kruskal ([Kru56], [CLR89]) berechnet einen minimalen Spannbaum, indem für jede Kante in nach Gewichten aufsteigender Reihenfolge geprüft wird, ob diese Kante zwei verschiedene Zusammenhangskomponenten bereits gewählter Kanten verbindet. Dazu werden die Kanten zuerst nach Gewichten sortiert. Dann wird jedem Knoten eine Menge zugeordnet. Nun wird für jede Kante überprüft, ob sie zwei Knoten verbindet, die in verschiedenen Mengen sind. Falls ja, wird sie als Kante des minimalen Spannbaumes gewählt und die beide Mengen vereinigt.

Abbildung 8 Algorithmen und Probleme für minimale Spannbäume
Um einen Überblick über die benötigten Algorithmen eines gültigen Schedules für MST zu erhalten, führt man eine Suche in diesem Graph durch, wobei man an Problemknoten einen beliebigen Nachfolger wählt, an Algorithmusknoten jedoch alle Nachfolger (Unterprobleme) betrachten muß.



Die Laufzeit dieses Algorithmus wird durch das Sortieren der Kanten dominiert und ist sequentiell $O(m \log m)$ bei m Kanten. In der Implementierung in dieser Arbeit ist es natürlich möglich, dass das Sortieren parallel auf mehreren Prozessoren ausgeführt wird, da es als Unterproblem spezifiziert ist.

BorůvkaStep

Im Problem BorůvkaStep wird für jeden (Super-) Knoten die billigste Kante gesucht, die diesen Knoten mit einem anderen verbindet. Es gibt immer einen minimalen Spannbaum, der diese Kanten enthält. Durch jede Ausführung eines Borůvka-Schrittes verkleinert sich so die Zahl der (Super-) Knoten um mindestens den Faktor zwei.

Wie oben erwähnt, sind für dieses Problem zwei Algorithmen implementiert. Sie unterscheiden sich darin, wie die Kanten ausgewählt werden. DENSEBORUVKASTEP (s. Algorithmus 1, Seite 31) entspricht Schritt (2) des Algorithmus MST-DENSE aus [ADJ+98], wählt die minimalen Kanten zuerst lokal aus und berechnet dann mittels einer parallelen Prefixoperation (EDGEMINIMUM) die globalen Minima. Anschließend werden die durch diese Kanten gebildeten Zusammenhangskomponenten berechnet und die neuen Superknoten bestimmt. Wie der Name nahelegt, eignet sich dieser Algorithmus eher für dichte Graphen, da er für jeden Knoten ein Minimum berechnet und dies bei n Kanten mindestens $\Theta(n)$ Zeit benötigt.

Der zweite Algorithmus, BORŮVKASTEP ([Göt98]), berechnet die Adjazenzlisten zu den (Super-) Knoten. Dazu werden alle Kanten in beide Richtungen genommen und nach dem ersten Knoten sortiert. Anschließend bilden sie Segmente von Kanten. Jedes Segment enthält die zu einem Knoten adjazenzen Kanten. Nun werden mittels SEGMENTMINIMUM die Minima dieser Segmente berechnet. Zum Schluß werden auch bei diesem Algorithmus die Zusammenhangskomponenten berechnet. In der in dieser Arbeit gewählten Implementation erfolgt dies sequentiell auf Prozessor 0 mit Kosten $(n-1) \cdot \text{sizeof}(Edge) \cdot g(p) + L(p) + 6 \cdot n \cdot \text{sizeof(int)}$ bei n Superknoten und p Prozessoren. $\text{sizeof}(Edge)$ ist die Speichergröße einer Kante in Bytes. Bei dem hier verwendeten Compiler benötigen die Daten einer Kante (drei Integerzahlen, ein Double-Wert und Füllbytes) zusammen $b = 24$ Bytes. Götz schlägt in [Göt98] (s.a. [DG98]) vor, einen parallelen Zusammenhangskomponenten-Algorithmus aus [CDF+97] mit Laufzeit $O((\frac{n}{p} + \frac{n}{p}g + L) \cdot \log p)$ im BSP-Modell zu verwenden.

MSTMERGE

Dieser Algorithmus entspricht dem zweiten Schritt des Algorithmus MST-MERGE von Adler et al. ([ADJ+98]) und vereint lokal berechnete minimale Spannbäume. Dazu wird ein Kommunikationsbaum mit beliebigem Baumgrad d über die Prozessoren gelegt. In jeder Runde senden alle noch aktiven Prozessoren ihre Bäume zu

ihrem Vaterprozessor. Dieser berechnet dann lokal aus allen empfangenen Kanten einen minimalen Spannbaum, alle anderen Prozessoren werden passiv. Am Ende des Algorithmus besitzt der Vater von allen, Prozessor 0, den globalen minimalen Spannbaum.

Die Kosten betragen für n Knoten und e Kanten betragen

$$\begin{array}{ll} (n-1)\cdot(d-1)\cdot \mathrm{sizeof}(\mathit{Edge})\cdot(g(p)+2g(1))+L(p) & \text{falls } d=p \\ \lceil e/p\rceil\cdot(d-1)\cdot \mathrm{sizeof}(\mathit{Edge})\cdot(g(p)+2g(1))+L(p) & \text{sonst,} \end{array}$$

jeweils zuzügl. der Kosten für die Unterprobleme.

MSTWITHMERGE

Dieser Algorithmus benutzt ausschließlich MSTMERGE zur Berechnung des Spannbaumes. Er berechnet zuerst auf jedem Prozessor den minimalen Spannbaum aus den lokal gespeicherten Daten, anschließend werden diese Spannbäume mit einer baumförmigen Kommunikationsstruktur mittels MSTMERGE zusammengemischt.

Zum Schluss werden die Kanten des minimalen Spannbaumes noch gleichmäßig über alle Prozessoren verteilt wird, da das Problem MST per Definition die Ausgabe auf allen Prozessoren verlangt.

Die Kosten hierfür setzen sich aus dem lokalen Kopieren der Daten für die Unterprobleme $(2\lceil e/p\rceil + n - 1)\cdot \mathrm{sizeof}(\mathit{Edge})\cdot g(1)$ und dem Verteilen des Ergebnisses von Prozessor 0 an alle anderen zusammen: $(e - \lceil e/p\rceil)\cdot \mathrm{sizeof}(\mathit{Edge})\cdot g(p) + L(p)$.

MST

Dieser Algorithmus berechnet den minimalen Spannbaum, indem solange Borůvka-Schritte ausgeführt werden, bis nur noch ein Superknoten übrigbleibt. Durch jeden Schritt reduziert sich die Anzahl der Knoten um mindestens den Faktor zwei, es sind also maximal $\lceil \log_2 n\rceil$ Borůvka-Schritte nötig. Diese obere Schranke wird erreicht, wenn in jedem Schritt je zwei Superknoten die gleiche Kante auswählen.

Neben den Unteraufrufen treten hier nur die Kosten zum Kopieren der Ein- und Ausgabe auf: $(\lceil e/p\rceil \cdot \mathrm{sizeof}(\mathit{BoruvkaStepEdge}) + \lceil n/p\rceil \cdot \mathrm{sizeof}(\mathit{Edge}))\cdot g(1)$. Die Datenstruktur *BoruvkaStepEdge* speichert zusätzlich zu einer normalen Kante zwei Integerzahlen, die angeben, zu welchem Superknoten Start- und Zieleknoten gehören.

MSTBORUVKAANDMERGE

Dieser Algorithmus kombiniert die beiden Lösungsmöglichkeiten und führt zuerst Borůvka-Schritte aus, anschließend einmal MSTMERGE. Die Anzahl der Borůvka-Schritte kann zwischen 1 und $\lceil \log_2 n\rceil - 1$ liegen und wird durch die Variante des

Algorithmus 1 DenseBorůvkaStep

(i) Berechne lokal zu jedem Superknoten die billigste Kante, die diesen Knoten verlässt.

(ii) Berechne als Unterproblem `ReduceAllEdgeMinimum` global die Minima über diese Kanten.

(iii) Berechne in einem Prozessor die Zusammenhangskomponenten, die durch diese Kanten entstehen.

(iv) Schicke mit Unterproblem `BroadCast` die Nummerierung der Superknoten an alle Prozessoren.

(v) Aktuallisiere die Kanten gemäß der neuen Superknoten, lösche Kanten innerhalb eines Superknotens.

(vi) Lösche eine Spannbaum-Kante aus die Kreisen der Länge zwei.

(vii) Jeder Prozessor wählt $1/p$ der minimalen Kanten aus und fügt sie in den Spannbaum ein.

Algorithmus festgelegt. Kosten bei diesem Algorithmus entstehen durch:
Umkopieren der Ein-/Ausgaben und Zwischenergebnissen:

$$(\lceil e/p \rceil \cdot (\text{sizeof}(\mathit{BoruvkaStepEdge}) + \text{sizeof}(\mathit{Edge})) + (\lfloor \frac{n}{2^k} \rfloor - 1) \cdot \text{sizeof}(\mathit{Edge})) \cdot g(1)$$

Zählen der lokalen Spannbaumkanten: $(2p + 3(n-1)) \cdot \text{sizeof}(\mathit{int}) \cdot g(1)$
Senden der gefundenen Spannbaumkanten-Ids zu den Prozessoren, auf denen die Ausgangskante liegt: $(s-1) \cdot \text{sizeof}(\mathit{int}) \cdot g(p) + L(p)$
Zählen und Verteilen der Spannbaumkanten:

$$\left(\sum_{i=\frac{n}{2}}^{\frac{n}{2^k}} i + \min\left(\left\lfloor \frac{n}{2^k} \right\rfloor - 1, \left\lceil \frac{e}{p} \right\rceil \right) \right) \cdot (\text{sizeof}(\mathit{int}) g(1) + \text{sizeof}(\mathit{Edge}) \cdot (g(1) + g(p))) + L(p)$$

Falls die Anzahl auf $2 \cdot \log\log p$ festgelegt und als Unteralgorithmus DenseBoruvkaStep gewählt wird, entspricht dies dem Algorithmus MST-Dense von Adler et al. ([ADJ+98]).

3.3.2. Implementierungen

Die Implementierung der Algorithmen für minimale Spannbäume zeigt einige Nachteile, die durch das Modell und die genau definierten Schnittstellen entstehen. Ins-

Algorithmus 2 BorůvkaStep

(i) Füge zu jeder Kante (v, u) die Kante (u, v) hinzu.

(ii) Sortiere Kanten (u, v) nach Superknoten von v (Berechnung der Adjazenzlisten der Superknoten) mit Unterproblem `IntegerSort`.

(iii) Berechne als Unterproblem `SegmentMinimum` die Minima der durch die Superknoten gebildeten Segmente der Kanten.

(iv) Schicke die diese minimalen Kanten an den Herkunftsprozessor und füge sie dem Spannbaum hinzu.

(v) Berechne die Zusammenhangskomponenten und damit die neuen Superknoten.

(vi) Nummeriere alle Kanten gemäß den neuen Superknoten durch.

(vii) Entferne Kanten innerhalb gleicher Superknoten.

---

besondere die Verteilung der Daten auf die Prozessoren führt hier zu Mehraufwand, z.B. bei DenseBoruvkaStep:

- Die Kanten des Graphen sind gleichmässig über alle Prozessoren verteilt, falls die Anzahl $|E|$ allerdings nicht durch die Zahl der Prozessoren p teilbar ist, erhalten die ersten $|E| \bmod p$ Prozessoren $\lceil |E|/p \rceil$ Kanten und die restlichen $\lceil |E|/p \rceil - 1$. Vor dem Sortieren werden die Kanten verdoppelt (einmal in jede Richtung). Dadurch sind sie eventuell nicht mehr gleichmäßig verteilt, da die Differenz der Anzahlen nun zwei beträgt. Dies ist in der theoretischen Betrachtung im O-Kalkül kein Unterschied, die Spezifikation des Sortier-Problems erfordert aber eine genau gleichmässige Verteilung, da sonst einzelne Elemente nicht mitsortiert werden.

  Also wird ein weiterer Superstep eingeschoben, der die Kanten bei Bedarf wieder gleichmäßig verteilt. Auch wenn dabei jeder Prozessor nur maximal eine Kante verschickt, ist doch eine eventuell teure Synchronisation erforderlich.

- Das Ergebnis der Segment-Minimum-Operation (die minimalen Kanten der Adjazenzlisten der Superknoten) sind gleichmäßig über alle Knoten verteilt. Um aber Kreise erkennen zu können (diese können nach Konstruktion nur mit der Länge zwei auftreten), muss zu jeder Kante $e_1 = (v, u)$ zwischen Superknoten v und u überprüft werden, ob die Kante (u, v) ebenso minimal für Superknoten u ist. Da diese Kanten eventl. auf verschiedenen Prozessoren liegen, werden alle minimalen Kanten zu ihrem Heimatprozessor geschickt. Die-

Abbildung 9 Verteilung der Kanten bei DenseBoruvkaStep

Startzustand

P_0	P_1	P_2	P_3
e_0	e_2	e_4	e_5
e_1	e_3		

Nach Verdopplung

P_0	P_1	P_2	P_3
e_0	e_2	e_4	e_5
e_1	e_3	e_4	e_5
e_0	e_2		
e_1	e_3		

Nach Ausgleichsschritt

P_0	P_1	P_2	P_3
e_0	e_2	e_4	e_5
e_1	e_3	e_4	e_5
e_0	e_2	e_1	e_3

ses erfordert, dass zu jeder Kante auch die Information gespeichert wird, auf welchem Prozessor sie ursprünglich gespeichert war.

Nach der Entfernung doppelter Kanten müssen die gefundenen MST-Kanten wieder gleichmäßig verteilt werden. Hierzu wird in einer parallelen Präfix-operation für jeden Prozessor i berechnet, wie viele Kanten die Prozessoren $0, \ldots, i-1$ speichern. Im nächsten Superstep werden die MST-Kanten dann entsprechend verteilt. Abbildung 9 zeigt dies für vier Prozessoren und sechs Kanten. Diese Berechnungen sind in [Göt98] nicht notwendig, da dort keine Anforderungen an die Verteilung der gefundenen Kanten gemacht wird. Für die Laufzeitabschätzung reicht es dort, dass jeder Prozessor nur Heimat von maximal $|E|/p$ Kanten ist.

Durch die strenge Kapselung der einzelnen Algorithmen ergibt bei der Implementierung des Algorithmus MSTBoruvkaAndMerge ein weiteres Problem: Nachdem einige Knoten durch Borůvka-Schritte zusammengefasst wurden, wird ein Spannbaum auf diesen Superknoten berechnet. In dem Gesamtergebnis sollen nun die in beiden Teilen gewählten Kanten zusammengefasst werden. Allerdings ergibt sich das Problem, dass Kanten, die MSTMerge berechnet hat, Kanten zwischen Superknoten sind. Hierzu müssen die Kanten des Eingabegraphens gefunden werden, aus denen diese Superknoten-Kanten entstanden sind.

Um das Finden der Graphkanten zu diesen Superknoten effizient zu ermöglichen, wurde einer Kante ein weiteres Feld id hinzugefügt. Diese Feld wird von den MST-Algorithmen nicht betrachtet und dient lediglich dazu, die Kante im Ausgangsgraph wiederzufinden. Dazu wird vor Ausführung der Borůvka-Schritte die Id jeder Kante auf ihre Position innerhalb der Kanten gesetzt. Nun kann jede MST-Kante zu dem Prozessor gesendet werden, auf dem die Original-Kante liegt. Anschließend werden die Kanten nach einem Ausgleichsschritt, der sie gleichmässig auf alle Prozessoren verteilt, in den minimalen Spannbaum eingefügt.

Diese Konstruktion hat die zwei folgenden Nachteile:

- Die Eingabedaten enthalten ein „merkwürdiges“ Feld id, welches nur innerhalb

der Bibliothek benötigt wird.

- Durch dieses Feld vergrößert sich die Datenstruktur für Kanten. Dies führt zu einem höheren Speicherbedarf und Kommunikationsvolumen.

3.3.3. Laufzeitvorhersagen

Ein gültiges Schedule für 4096 Knoten und 32768 Kanten mit 16 Prozessoren zeigt Abbildung 10. Horiziontal sind die Prozessoren P_0 bis P_{15} dargestellt, vertikal die ausgeführten Algorithmen. Unteralgorithmen sind innerhalb des aufrufenden Algorithmus in zeitlicher Abfolge von oben nach unten dargstellt, so ist z.B. DenseBoruvkaStep der erste Unteralgorithmus von MSTBoruvkaAndMerge. Die Höhe der einzelnen Algorithmen-Boxen ist nicht maßstabsgetreu gemäß der Laufzeit.

Bei diesem Schedule wurden also folgende Entscheidungen getroffen:

- Auswahl eines Algorithmus für MST: MSTBoruvkaAndMerge, mögliche Alternativen MST, MSTWithMerge.
- Variante mit nur einem Borůvka-Schritt, erlaubt ist 1 bis $\lceil \log_2 4096 \rceil - 1$.
  - Algorithmus DenseBoruvkaStep für den Borůvka-Schritt, alternativ ist auch BoruvkaStep möglich.
    * Broadcast innerhalb des Borůvka-Schrittes zuerst als Baum mit Grad zwei, anschließend mittels ScatterGatherBroadcast. Weitere Algorithmen für Broadcast sind in Kapitel 3.2.1 beschrieben.
  - Baumgrad beim Mischen der lokalen Spannbäumen zwei.

TCP/IP. In Abbildung 11 ist dargestellt, welcher der drei Algorithmen für das Problem der Berechnung minimaler Spannbäume innerhalb des Modells der schnellste ist. Bei kleinen Datengrößen wird der Algorithmus MSTWithMerge ausgewählt, da dieser nur wenige Supersteps benutzt und in der hier gewählten Variante mit Baumgrad zwei bei kleiner Anzahl von Kanten die lokale Rechenarbeit unterhalb den Kommunikationskosten für Borůvka-Schritte liegt.

Je mehr Prozessoren beteiligt sind, desto ungünstiger wird dieser Algorithmus, da er entweder mehrere Runden benötigt oder einen höheren Baumgrad d benutzen muss. Dies führt jedoch zu einer größeren lokalen Rechenarbeit, da in jedem Schritt bei n Knoten $(n-1) \cdot d$ Kanten gemischt werden.

Bei den hier untersuchten Parametern erscheint nur bei vier Proezssoren ein Schedule mit dem Algorithmus MSTBoruvkaAndMerge mit mehr als einem Borůvka-Schritt (8192 bis 16384 Knoten und 16384 bis 32768 Kanten). Jeder weitere Borůvka-Schritt führt zu einer Halbierung der Knoten, die Anzahl der Kanten bleibt

Abbildung 10 Schedule für 4096 Knoten, 32768 Kanten, 16 Prozessoren

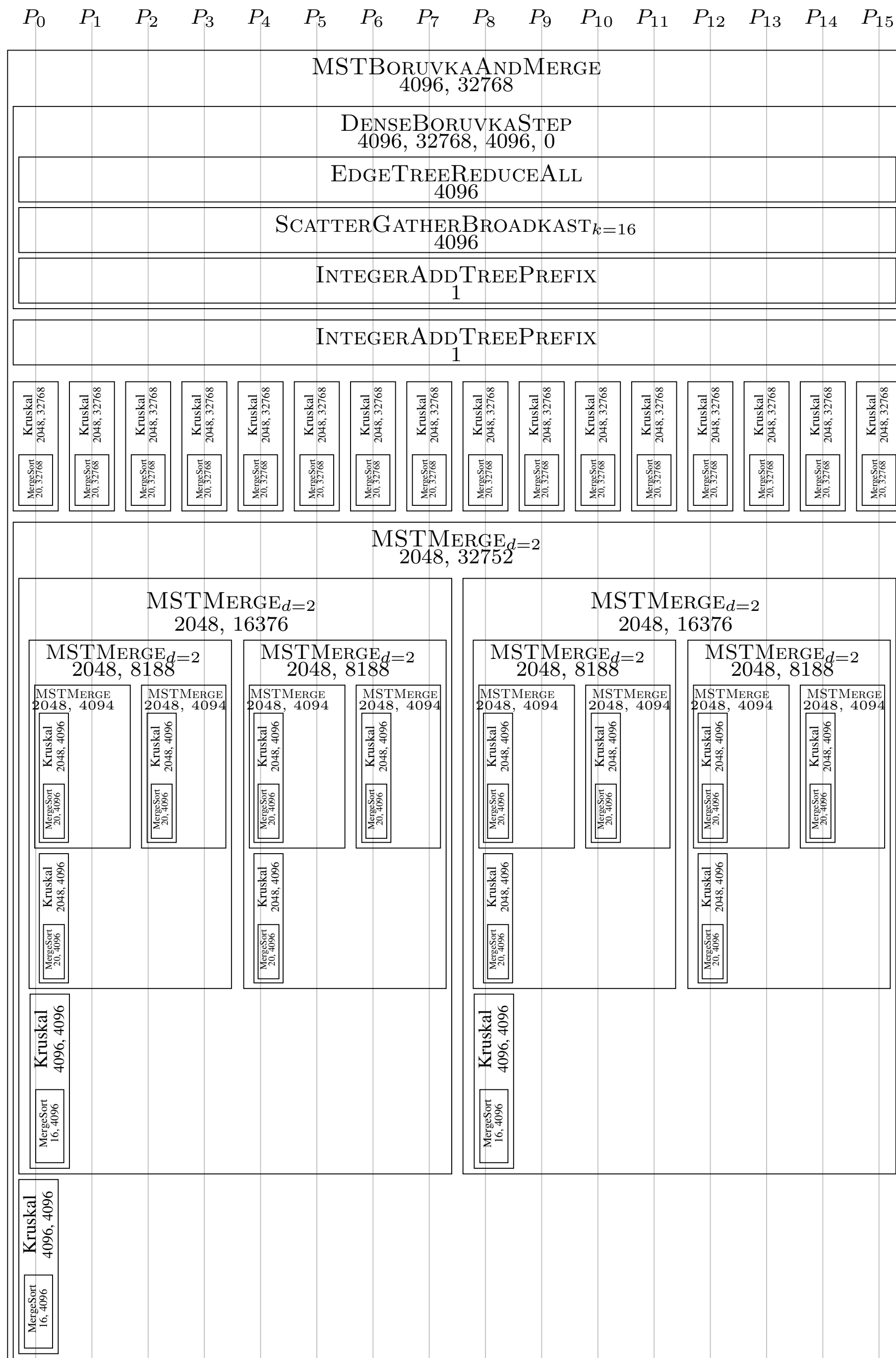
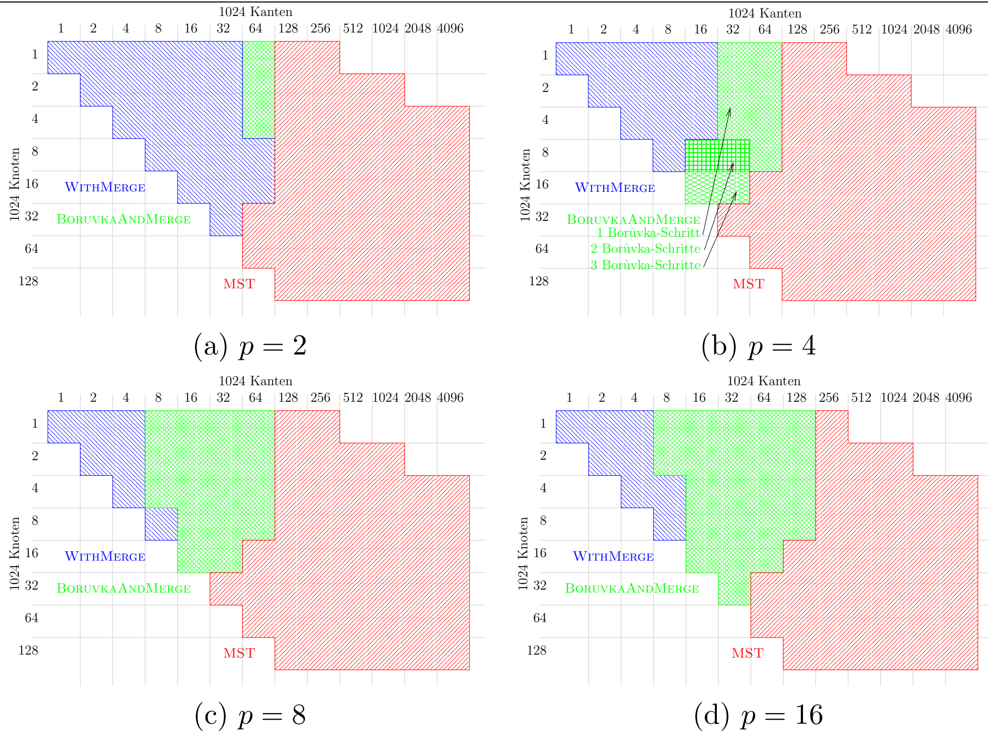
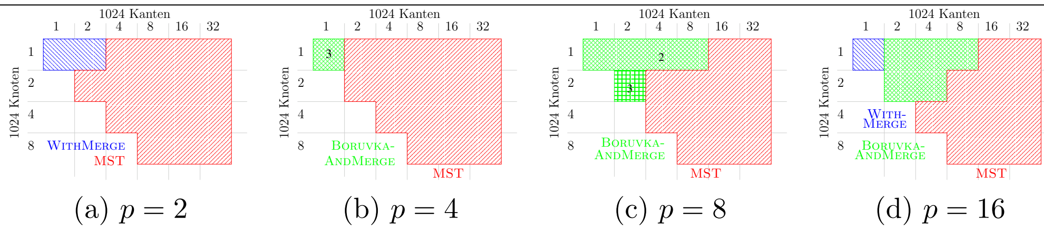


Abbildung 11 Optimale Algorithmen für MST auf PSC2, TCP/IP (Vorhersage)



(a) $p = 2$ (b) $p = 4$ (c) $p = 8$ (d) $p = 16$

Abbildung 12 Optimale Algorithmen für MST auf PSC2, SCI (Vorhersage)



(a) $p = 2$ (b) $p = 4$ (c) $p = 8$ (d) $p = 16$

jedoch im worst-case fast gleich, da lediglich Kanten innerhalb der neuen Superknoten gelöscht werden. Vor der Ausführung des Mischens der Kanten werden lokal Spannbäume berechnet. Die Laufzeit dieser Berechnung wird jedoch hauptsächlich von der Anzahl der Kanten dominiert.

SCI. Betrachtet man die optimalen Algorithmen für einen Computer mit schnellerer Kommunikation (s. Abbildung 12), so fällt auf, dass sich die Grenzen zwischen den Algorithmen in Richtung kleinerer Eingaben verschieben. Dies liegt daran, dass für MSTMerge zwar im Vergleich zu den Borůvka-Schritten weniger Kommunikation erforderlich ist, dieses Mischen jedoch mit einem sequentiellen MST-Algorithmus erfolgt. Des Weiteren sind in diesem Algorithmus in Runde i immer nur $\frac{p}{d^i}$ Prozessoren aktiv. Durch das geringere Verhältnis von Kommunikationskosten zu Kosten lokaler Berechnung lohnt sich schon bei kleinen Datengrößen die Anwendung von MSTBoruvkaAndMerge bzw. MST.

3.3.4. Messungen

Alle Messungen wurden mittels zufällig erzeugten Graphen durchgeführt. Dazu wurden der Reihe nach zu allen Kanten zufällig die beiden Knoten ausgewürfelt. Doppelte Kanten und Kanten der Form $(u-u)$ wurden verworfen. Der folgende Algorithmus zeigt das Erzeugen von Zufallsgraphen mit n Knoten und e Kanten, $\text{rand}(x)$ sei dabei eine Funktion, die zufällige gleichverteilte natürliche Zahlen aus dem Interval $[0, x-1]$ liefert:

$E \leftarrow \emptyset$
while $|E| < e$ **do**
  $u \leftarrow \text{rand}(n)$
  repeat
    $v \leftarrow \text{rand}(n)$
  until $u \neq v$
  if $(u, v) \notin E$ **and** $(v, u) \notin E$ **then**
    $E \leftarrow E \cup \{(u, v)\}$
    $w[(u, v)] \leftarrow \text{rand}(10000)$

Als Kantengewichte werden hier nur natürliche Zahlen aus $[0, 9999]$ erzeugt, obwohl die Implementierung mit dem Datentyp double arbeitet und beliebige Fließkommazahlen größer oder gleich Null erlaubt. Diese besondere Wahl hat hat allerdings keine Auswirkungen auf die Laufzeit der benutzen Algorithmen. Zur schnellen binären Suche nach Kanten wird E während der Erzeugung zusätzlich in sortierter Form gespeichert.

Durch die Mindestmesszeit von fünf Sekunden (s. Kapitel 3.1.1) wurden die einzelnen Messungen unterschiedlich oft wiederholt (nur eine Messung bei großen bis

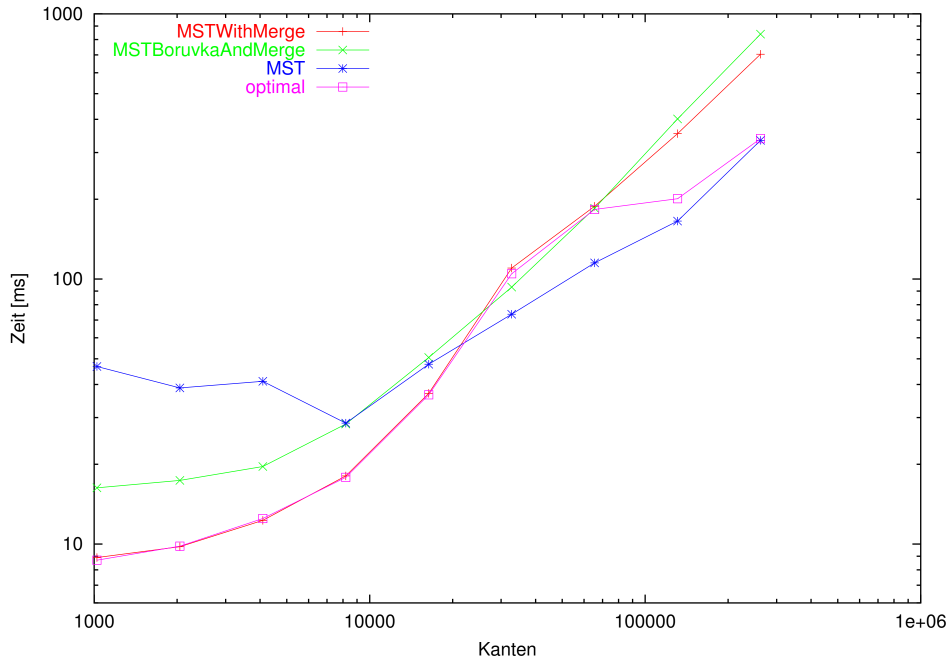
max. 930 Messungen bei kleinen Graphen). Da das Auswürfeln zufälliger Graphen nach diesem Algorithmus sehr lange (bis zu mehreren Stunden) dauert, wurden die Messungen nur für je einen zufälligen Graphen pro Eingabegröße durchgeführt. Durch die große Anzahl an Kanten ist die Wahrscheinlichkeit, zufällig einen Graphen mit bestimmten Eigenschaften zu erhalten, die die Laufzeit im Vergleich zu durchschnittlichen Graphen beeinflussen, vermutlich sehr gering.

TCP/IP. Betrachtet man die Laufzeiten auf zwei Prozessoren des Rechners PSC2 mit TCP/IP und Graphen mit nur 1024 Knoten, so zeigt sich (s. Abbildung 13), dass das Modell bis auf zwei Ausnahmen bei 32768 und 65535 Kanten immer den schnellsten Algoprithmus auswählt. Das Verhältnis zwischen gemessener und verhergesagter Laufzeit liegt bei den Algorithmen MSTWithMerge und MSTBoruvkaStep dicht beieinander. Bei kleinen Daten wird eine zu geringe, bei großen Daten eine etwas zu große Laufzeit vorhergesagt. Lediglich der Algorithmus MST ist bei einer kleinen Anzahl von Kanten deutlich schneller als erwartet. Dies führt dazu, dass der Wechsel zu diesem Algorithmus schon bei einer kleineren Anzahl von Kanten (32768) sinnvoll wäre. Der maximale Laufzeitverlust durch diese Fehleinschätzung tritt bei 65536 Kanten auf und beträgt 37%.

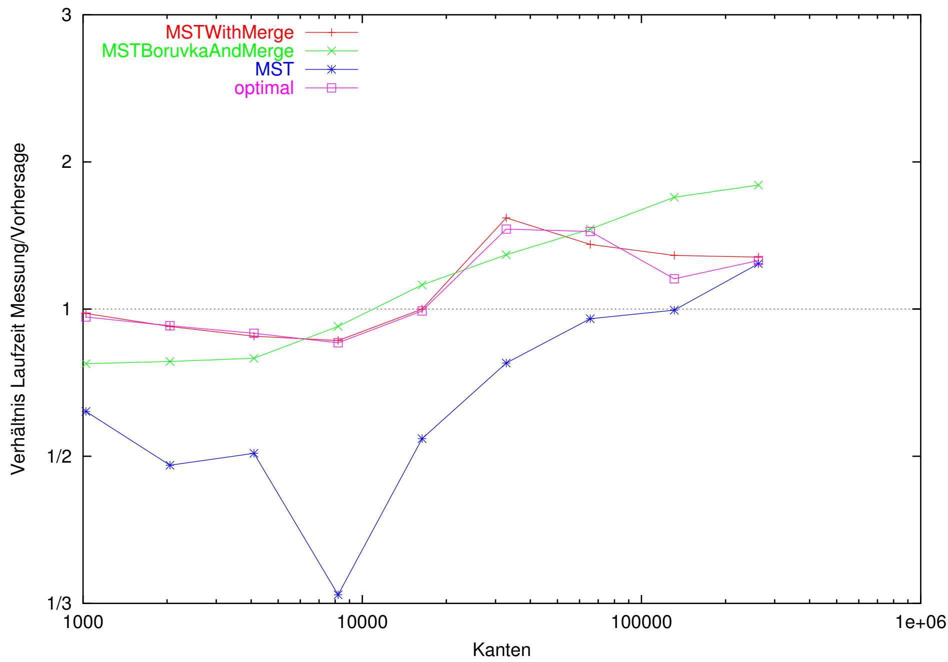
Bei 16 Prozessoren und Graphen mit 32768 Knoten sind die Vorhersagen nicht so genau (s. Abbildung 14). In der Realität ist hier immer der Algorithmus MSTMerge der schnellste. Dieser wird aber vom Scheduler bei keiner Eingabegröße ausgewählt. In Abbildung 14 (b) sieht man, dass dieser Algorithmus im Gegensatz zu den anderen sehr stark unterschätzt wird. Der größte Fehler hierbei kommt analog zum Algorithmus List beim Broadcast dadurch zustande, dass das BSP-Modell keine Netzwerk-Auslastung berücksichtigt. Bei einer großen Anzahl von Kanten benötigt innerhalb des Algorithmus MSTWithMerge das gleichmäßige Verteilen der Spannbaumkanten von Prozessor 0 auf alle anderen die meiste Zeit: $e - \lceil e/p \rceil \cdot g(p) + L(p)$ bei e Kanten auf p Prozessoren. Während dieser Zeit sendet nur Prozessor 0, alle anderen empfangen lediglich. Daher ist das Netzwerk fast unbelastet und das von der PUB-Library unter Volllast gemessene $g(32)$ ist viel zu hoch.

SCI. Im Vergleich zu den Messungen der Broadcast-Algorithmen gab es bei den MST-Algorithmen weniger Probleme des Systems PSC2 mit SCI. Die geringere Varianz der Messwerte könnte an der neueren Software-Version des Computer liegen. Ein weiterer möglicher Grund ist das andere Kommunikationsverhalten der Spannbaum-Algorithmen. Während Broadcast-Algorithmen keine lokalen Berechnungen vornehmen, führen MST-Algorithmen neben lokalen Speicherkopien auch Berechnungen wie Minimumsbildung und Sortieren aus. Dadurch und durch die nicht immer genau gleich verteilten Datenmengen wird die Wahrscheinlichkeit gleichzeitiger Zugriffe auf das Netzwerk reduziert.

Abbildung 13 Messungen MST PSC2, TCP/IP, 2 Prozessoren, 1024 Knoten

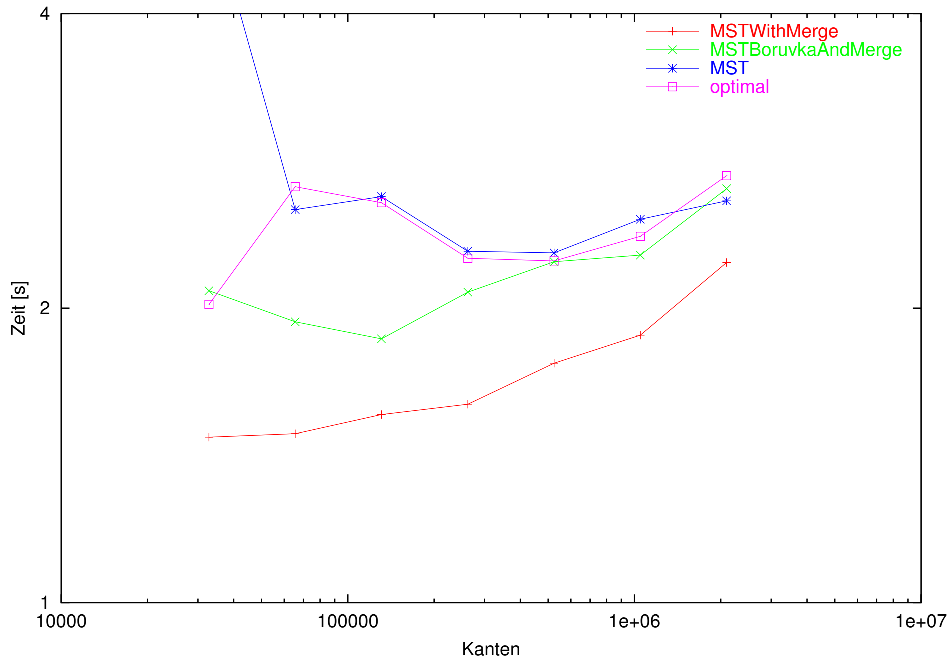


(a) Laufzeiten

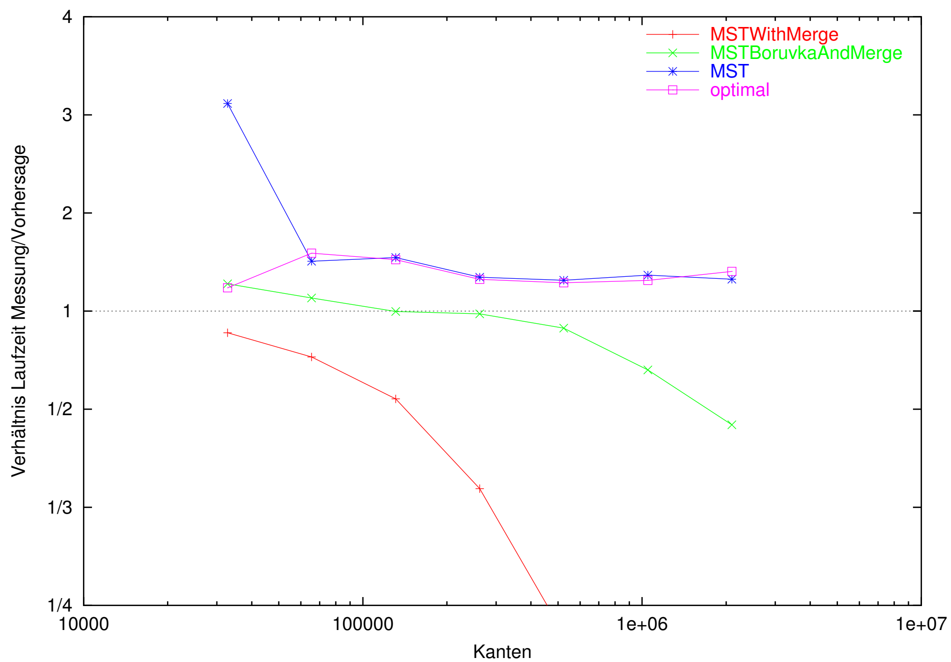


(b) Genauigkeit der Vorhersagen

Abbildung 14 Messungen MST PSC2, TCP/IP, 16 Prozessoren, 32768 Knoten



(a) Laufzeiten



(b) Genauigkeit der Vorhersagen

Bei zwei Prozessoren wählt das System immer den optimalen Algorithmus aus. Alle Algorithmen werden schneller eingeschätzt, als sie in der Realität sind. Allerdings sind die Fehler bei allen Algorithmen in der gleichen Größenordnung, so dass dies nicht zu einer falschen Auswahl führt. Lediglich der reine Misch-Algorithmus ist bei Graphen mit 16384 und mehr Kanten deutlich langsamer als in der Theorie, aber dieser Algorithmus findet nur für kleine Graphen Anwendung.

Bei 16 Prozessoren fallen die größeren Schwankungen der Messwerte auf. So unterscheiden sich die Messwerte zwischen optimal und MST um bis zu 15%, obwohl optimal in dieser Konfiguration immer MST benutzt, also in beiden Versuchsreihen die gleichen Algorithmen ausgeführt werden. Ausserdem gibt es bei der Ausführung des Algorithms MSTWithMerge zwei Aussreißer bei 128k und 512k Kanten.

3.4. Bewertung

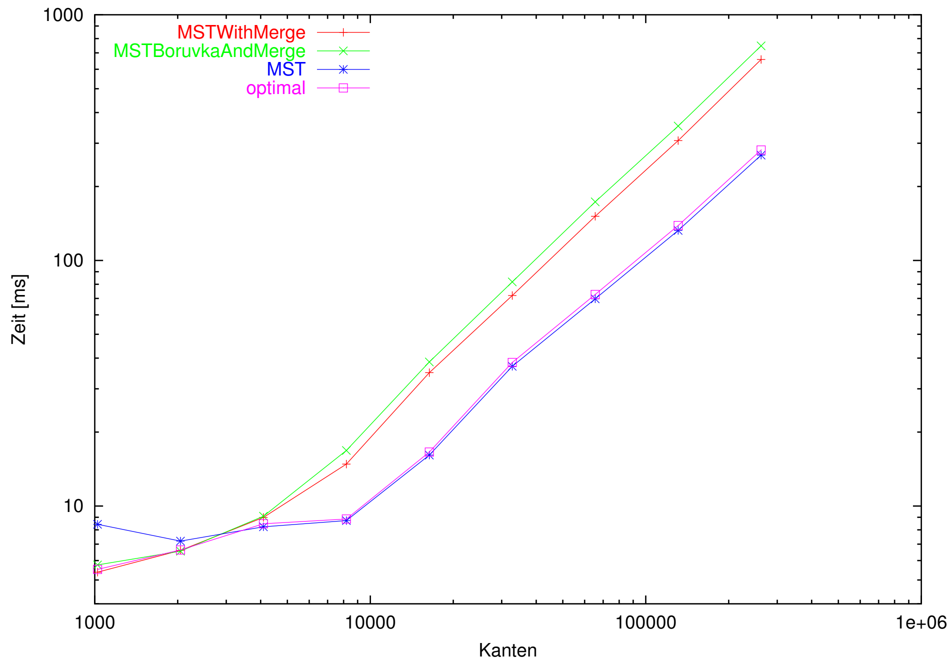
Bei den Untersuchungen in diesem Kapitel haben sich einige Probleme bei der Implementierung und dem Gebrauch dieses Systemes gezeigt. Wie bei fast allen Bibliotheken, die versuchen, bestimmte Probleme möglichst allgemein zu lösen, kommt es zu Performanceverlusten, z.B. durch viele Funktionsaufrufe. Dieses Problem tritt hier vor allem bei Vergleichsfunktionen auf, beispielsweise beim Sortieren und bei der Minimumsbildung. Ein Funktionsaufruf kostet auf modernen Computern viel mehr Zeit als ein einfacher Vergleich.

Dieses Problem erscheint aber nicht nur in diesem System, sondern tritt sogar in einer der Grundbibliotheken auf: Die C-Bibliothek des C++-Compilers enthält auch eine Funktion qsort, die mittels Quicksort beliebige Datentypen sortieren kann. Da man auch hier eine allgemeine Vergleichsfunktion angeben kann, ist hier ein Vergleich bei einfachen Datentypen verhältnismässig teuer. Beim Sortieren von 10^7 zufälligen Integerzahlen benötigt qsort etwa dreimal so lange wie eine einfache eigene Implementierung von Quicksort[2]. An dieser Stelle ergibt sich dann in der Praxis immer ein Trade-Off zwischen Effizienz des Produktes und Verringerung des Implementieraufwandes. Bei einem so einfachen Algorithmus wie Quicksort wird die Entscheidung sicher oft auf eine eigene Implementation fallen, bei komplexeren Algorithmen wie z.B. minimalen Spannbaumalgorithmen halte ich den Laufzeitverlust für tragbar, da es in dieser Arbeit auch um eine allgemein nutzbare Bibliothek für Anwender gehen soll, die nicht unbedingt ihre Zeit mit parallelen MST-Algorithmen verbringen.

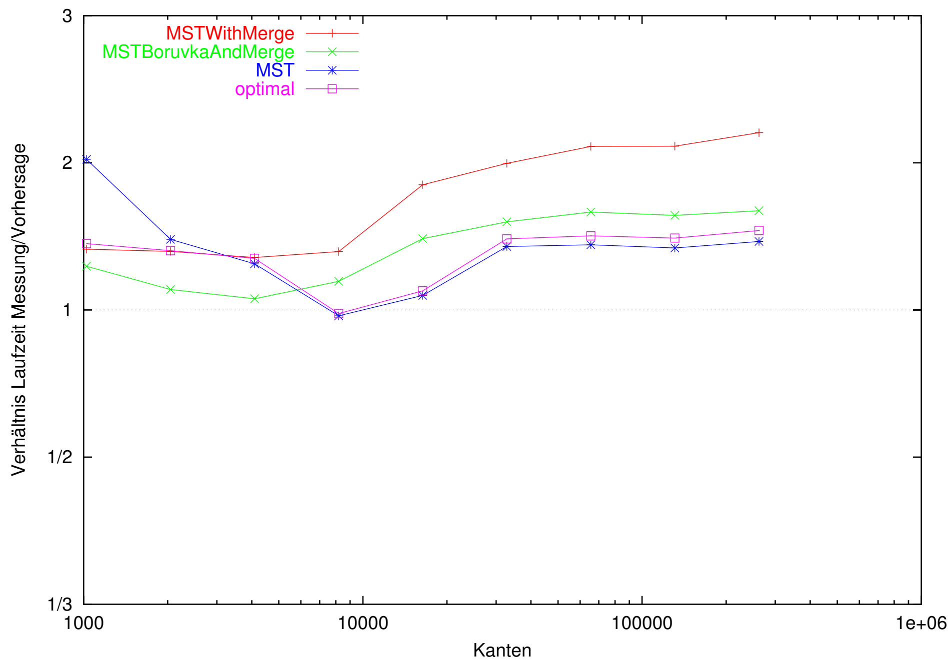
Das nächste aufgetretene Problem betrifft die Datenverteilung bei der Ein- und Ausgabe von Algorithmen. Das Modell verlangt zu jeder Problemspezifikation eine genaue Angabe, wo welche Daten bei Ein- und Ausgabe liegen und in welchem Format diese gespeichert sind. So verlangt z.B. das hier benutzte Problem IntegerSort,

[2] GNU C Library 2.2.4 mit gcc 2.95.3, übersetzt mit -O2, gemessen auf Pentium III, 933 MHz, C Library qsort 15.2 Sekunden, eigene Implementierung 5.1 Sekunden

Abbildung 15 Messungen MST PSC2, SCI, 2 Prozessoren, 1024 Knoten

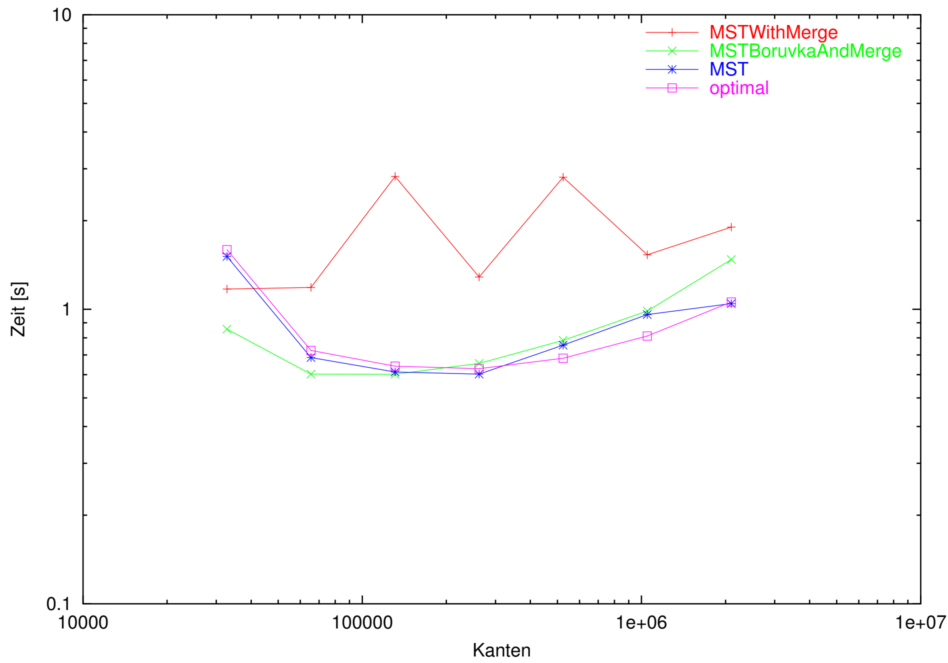


(a) Laufzeiten

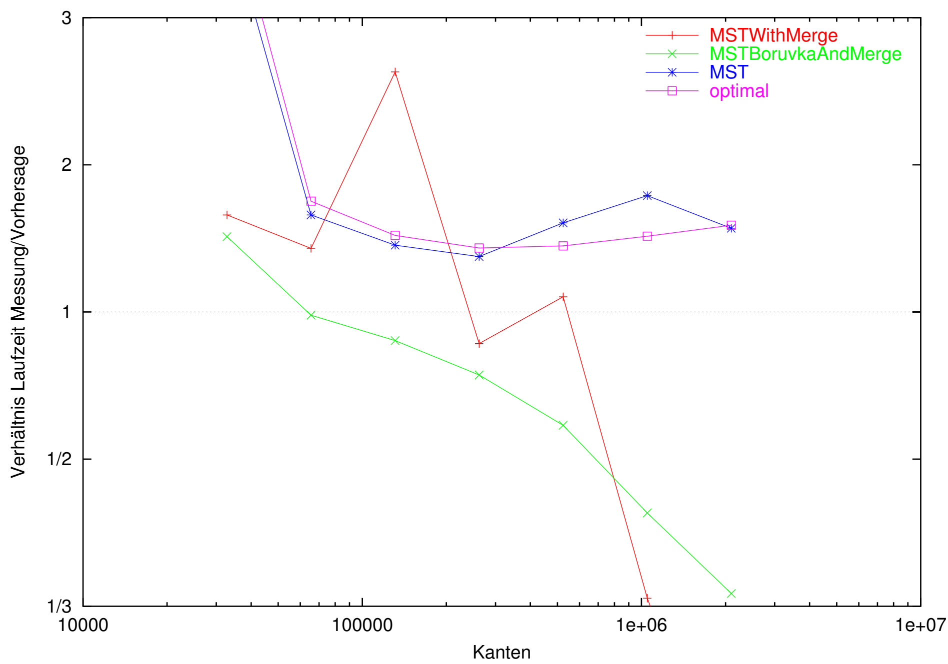


(b) Genauigkeit der Vorhersagen

Abbildung 16 Messungen MST PSC2, SCI, 16 Prozessoren, 32768 Knoten



(a) Laufzeiten



(b) Genauigkeit der Vorhersagen

dass der Integer-Schlüssel das erste Element in den Daten ist. Solche Anforderungen an Daten erzeugen teure Umkopieraktionen, falls die Daten nicht das gewünschte Format haben. Dazu wird nicht nur eine große Menge zusätzlicher Speicher benötigt (im Extremfall $r \cdot n$ bei n Eingabedaten und r verschachtelten Unterproblemen), auch steigt die Laufzeit an. Insbesondere bei schnellen Algorithmen mit annähernd linearer Laufzeit darf dieses Kopieren nicht vernachlässigt werden.

Ein weiteres Problem ist die Verteilung der Daten über die Prozessoren. Dieses tritt z.B. beim Algorithmus BorůvkaStep auf. Die e Kanten sind gleichmäßig über alle p Prozessoren verteilt, d.h. jeder hat maximal $\lceil \frac{e}{p} \rceil$. Für Betrachtungen in O-Kalkül reicht diese Anschätzung. Genauer sind die Daten aber wie folgt verteilt: Die ersten $e \mod p$ Prozessoren erhalten $\lfloor \frac{e}{p} \rfloor + 1$, die anderen $\lfloor \frac{e}{p} \rfloor$ Kanten. Zum Aufbau einer Adjazenzliste müssen diese Kanten verdoppelt werden, da eine Kante von v nach u sowohl bei Knoten v als auch bei Knoten u erscheinen soll. Nach der Verdopplung besitzen die Prozessoren aber $2\lfloor \frac{e}{p} \rfloor + 2$ bzw. $2\lfloor \frac{e}{p} \rfloor$ Kanten. Dies ist aber keine gleichmäßige Verteilung für das Unterproblem IntegerSort, also müssen einige Kanten anders verteilt werden, dies erzeugt zusätzliche Kosten $O(g(p)+L(p))$. Dieses Problem liegt vor allem an der Beschreibung der Eingabe. Das System benutzt eine Beschreibung der Eingabe, die nicht abhängig von einzelnen Prozessoren ist, d.h. die Eingabe für Unterprobleme muss auf jedem Prozessor gleich sein.

4. Implementierungen

Alle Implementierungen dieser Arbeit sind in der Programmiersprache C++ entstanden. Neben üblichen Tools zur Entwicklung (Compiler, Editor) kommen dabei auch zwei Bibliotheken zum Einsatz:

PUB: PUB ist eine C-Bibliothek, die es erlaubt, plattformunabhängige parallele Programme im BSP-Modell zu entwickeln, s. Kapitel 2.4, [pub] und [BJHR00b].

QT: Klassenbibliothek der Firma Trolltech ([Tro01]), enthält neben Klassen für Oberflächenelemente auch einige Datentypen wie z.B. Listen und unterstützt das plattformübergreifende Speichern von Daten in Dateien. Dies wird z.B. zum Speichern der Schedules benutzt.

4.1. Das BSP-Unteraufruf-Modell

Das Modell ist direkt in die in Tabelle 3 aufgeführten C++-Klassen abgebildet. Außerdem existieren einige Hilfsklassen (Eps zum Export von *Encapsulated PostScript* Dateien, ConfigFile zum Lesen von Konfigurationsdateien) und natürlich die implementierten Algorithmen und die Probleme dazu. Eine Übersicht über alle Algorithmen liefert Tabelle 5, über die Probleme Tabelle 4, die genauen Spezifikationen zu den Problemen zeigt Kapitel 4.4.

4.1.1. Die Klasse „Problem“

Diese Klasse ist die Basisklasse aller Probleme. Sie speichert den Namen des Problems und eine Liste aller Algorithmen, die für dieses Problem bekannt sind. Außerdem enthält sie einige abstrakte Methoden, die jedes Problem überschreiben muss:

QValueList<const char*>* getInputDescription()

Diese Methode dient der Beschreibung der Parameter, die eine Eingabe für dieses Problem beschreiben. Sie wird zum Beispiel in CallTreeEdit (s. Abbildung 17, Seite 63) benutzt und liefert eine Liste der Beschreibungen zu allen Parametern zurück.

Tabelle 3 C++-Klassen des Modells

Klasse	Beschreibung
Problem	Basisklasse für alle Probleme
Algorithm	Basisklasse für alle Algorithmen
InputDescription	Beschreibung von Eingabedaten, erlaubt eine beliebige Anzahl von `int` oder `double` Parametern
Data	Kapselt beliebige Eingabedaten
ProblemList	Liste aller Probleme und der dazugehörigen Algorithmen
CallTree	Schedule, speichert zu einem Problem eine Beschreibung der Eingabedaten, den gewählten Algorithmus sowie rekursiv zu allen Unterproblemen einen CallTree, wie diese zu lösen sind.
Search	Basisklasse für alle Scheduler
DepthSearch	Scheduler, der alle möglichen Schedules durch eine Tiefensuche mach dem Optimum durchsucht.

Data* createRandomInput(t_bsp* bsp, const InputDescription& descr)

Diese Methode erzeugt eine zufällige gültige Eingabeinstanz für das Problem und wird innerhalb dieser Arbeit für Messungen benutzt. Im praktischen Einsatz macht es natürlich meist wenig Sinn, zufällige Eingaben zu generieren.

unsigned int checkResult(t_bsp* bsp, Data* result);

Auch diese Methode ist nur für die Entwicklung und das Testen von Algorithmen gedacht. Sie überprüft das Ergebnis und liefert die Anzahl der Prozessoren zurück, auf denen das Ergebnis falsch ist. Natürlich kann sie nicht immer alle Fehler melden, beim Sortieren z.B. überprüft diese Methode, ob die Daten sortiert sind, nicht aber, ob es überhaupt noch die richtigen Daten sind.

4.1.2. Die Klasse „Algorithm“

Diese Klasse ist die Basis der Algorithmen-Klassen. Neben dem Namen und dem zugehörigen Problem gibt es eine Vielzahl virtueller Methoden, die von den Erben überschrieben werden müssen:

QValueList<int>* validNProcs(const InputDescription& n, int dataProcs, int maxCalcProcs)

Diese Methode gibt eine Liste aller möglichen Prozessorzahlen für diesen Algorithmus zurück. Die Elemente der Liste können in beliebiger Reihenfolge vorliegen und

müssen aus dem Intervall [dataProcs:maxCalcProcs] stammen. Auch eine leere Liste ist erlaubt, falls der Algorithmus das Problem bei dieser Eingabe oder Datenverteilung nicht lösen kann.

unsigned int variantCount(const InputDescription& n, int dataProcs, int calcProcs)

Das Modell erlaubt mehrere Varianten der Ausführung, dies wird z.B. beim Baum-Broadcast für die Realisierung verschiedener Baumgrade benutzt. Diese Funktion gibt zu jeder Eingabe an, wie viele verschiedene Varianten es gibt. Mit der folgenden Funktion können diese Varianten auch benannt werden, dies dient allerdings nur dem Benutzer und hat für die Korrektheit keine Bedeutung.

QString getVariantDescription(const InputDescription& n, int dataProcs, int calcProcs, unsigned int index)

Diese Funktion kann einzelnen Varianten einen Namen geben. Wird sie nicht überschrieben, gibt sie „Variante <index>“ zurück.

double time(const InputDescription& input, int dataProcs, int calcProcs, TimeFunction g, TimeFunction L, SizeFunction B, int variant)

Diese Funktion gibt die Laufzeit zurück, die der Algorithmus zur Ausführung benötigt. Neben der Beschreibung über die Eingabe beschreiben die weiteren Parameter die BSP-Parameter der Maschine. In der in Sekunden zurückgegebenen Zeit ist nur die Laufzeit dieses Algorithmus anzugeben, die Zeiten für alle Unterprobleme werden nicht berücksichtigt, da sie ja erst durch eine Auswahl von Algorithmen für diese Probleme bestimmt sind.

int subcallCount(const InputDescription& input, int dataProcs, int calcProcs, int variant)

Diese Funktion gibt an, wie viele Unterprobleme erzeugt werden. Diese Unterprobleme werden durch die nächste Funktion näher beschrieben.

SubcallDescription* subcall(const InputDescription& input, int dataProcs, int calcProcs, int index, int variant)

Diese Funktion beschreibt jedes erzeugte Unterproblem. Die zurückgegebene Klasse SubcallDescription enthält dabei folgende Felder:

Feld	Beschreibung
name	Name des Unterproblems
subDataProcs	Anzahl der Prozessoren, auf denen die Daten verteilt sind
subInput	Beschreibung der Eingaben für das Unterproblem
count	Anzahl der benötigten Ausführungen nacheinander
parallelCount	Anzahl der parallel benötigten Ausführungen

void execute(t_bsp* bsp, Data* input, CallTree* callTree)

Diese Methode führt den Algorithmus aus. Neben dem für die PUB-Library benötigten BSP-Objekt werden die Eingabe für den Algorithmus und ein Schedule angegeben. Dieses Schedule enthält Informationen darüber, auf wie vielen Prozessoren die Eingabe verteilt liegt (getDataProcs) und welche Variante des Algorithmus gewählt wurde (getVariant). Außerdem ist ihm zu entnehmen, welche Unteralgorithmen für Unterprobleme zu verwenden sind (getSubcall).

4.1.3. Scheduler

Alle Scheduler erben von der Basisklasse Search.

```
1  class Search {
2    public:
3       Search( ProblemList& aList );
4       virtual CallTree* optimize( const char* problemName,
   InputDescription input, int dataProcs, int calcProcs, TimeFunction
   g, TimeFunction L, SizeFunction B, double maxTime=1e100 ) = 0;
5       virtual  Search();
6    protected:
7       ProblemList& list;
8    };
```

Als Eingabe stehen neben der Liste aller Probleme Informationen über das zu lösende Problem zur Verfügung:

problemName: Der Name des Problems.

input: Eine Beschreibung über die Eingabedaten, z.B. Größe der Daten

dataProcs: Anzahl der Prozessoren, auf denen Ein- und Ausgabe liegen.

calcProcs: Maximale Anzahl von Prozessoren, die für die Berechnung verwendet werden können, natürlich immer größer oder gleich dataProcs.

Algorithmus 3 Scheduler mittels Tiefensuche

$t \leftarrow maxTime$
suche gewünschtes Problem *problem* in der Problemliste
for all *algorithmus* $\in$ *problem*->algorithmen **do**
  for all *procs* $\in$ *algorithmus*->validNProcs(*input*,*dataProcs*,*calcProcs*) **do**
    for all $v \in \{1, \ldots,$ *algorithm*->variantCount(*input*, *dataProcs*, *procs*)$\}$ **do**
      $t' \leftarrow$ *algorithm*->time()
      for all Unterprobleme u von Variante v von *algorithmus* **do**
        Berechne rekursiv optimales Schedule *sub* von u
        $t' \leftarrow t' +$ *sub*-> seqCount $\cdot$ *sub*->totalTime
      if $t' < t$ **then**
        speichere Variante und alle Unterschedules
        $t \leftarrow t'$
gib gespeichertes Schedule zurück

g,L,B: Funktionen, die die BSP-Parameter der gewünschten Zielarchitektur in Abhängigkeit von der Anzahl der benutzten Prozessoren beschreiben.

maxTime: Maximal für das Problem zur Verfügung stehende Zeit in Sekunden. Dieser Eintrag wird für rekursive Aufrufe bei der Suche benutzt, falls schon ein gültiges Schedule gefunden wurde.

Der in dieser Arbeit implementierte und verwendete Scheduler *DepthSearch* führt eine einfache Tiefensuche nach Algorithmus 3 aus. Im dort beschriebenen Algorithmus fehlt allerdings ein Test, ob eine Wiederholung benutzt wird. Dies kann z.B. passieren, wenn zwei Brückenalgorithmen sich abwechseln, z.B. kann man das Problem Sortieren lösen, in dem man die Daten zu einem Prozessor schickt und dort das Unterproblem Sortieren löst, hierfür könnte wiederum ein Algorithmus gewählt werden, der die Daten erst wieder verteilt und parallel sortiert. Solche Schleifen in der Schedule-Suche müssen natürlich erkannt und vermieden werden, weil es immer ein optimales Schedule ohne Schleifen gibt. Daher testet die Tiefensuche vor jedem rekursiven Aufruf, ob dieser Algorithmus mit den gleichen Aufrufparametern in der Suche vorher schon einmal vorkam.

4.2. Beispiel-Implementation Broadcast

Am Beispiel Broadcast soll dieser Abschnitt zeigen, wie man ein Problem und einen Algorithmus dafür in diesem System implementiert.

4.2.1. Klasse BroadcastProblem

Zuerst die Klasse für das Problem:

```
1  class BroadcastProblem : public Problem {
2     public:
3       BroadcastProblem();
4       QValueList<const char*>* getInputDescription();
5  #ifdef PUB_MACHINE
6       Data* createRandomInput( t_bsp* bsp, const InputDescription&
   descr );
7       unsigned int checkResult( t_bsp* bsp, Data* result );
8  #endif
9  };
```

Neben dem Konstruktor, der einfach `Problem( "Broadcast" )` aufruft, existiert eine Methode `getInputDescription`, die die Parameter beschreibt. Diese gibt einfach eine Liste mit dem Wort *size* zurück:

```
1  QValueList<const char*>* BroadcastProblem::getInputDescription()
2  {
3     QValueList<const char*>* list = new QValueList<const char*>;
4     list -> append( "size" );
5     return list;
6  }
```

Das Token `PUB_MACHINE` ist immer gesetzt, wenn ein Quelltext mit der PUB Library übersetzt wird. Nur in diesem Fall stehen Datenstrukturen wie `t_bsp` zur Verfügung. Daher werden die nun folgenden Methoden nur deklariert, falls wirlich ein paralleles Programm erzeugt wird.

Die Methode `createRandomInput` erzeugt für Testzwecke eine zufällige Eingabe für das Problem Broadcast:

```
1  Data* BroadcastProblem::createRandomInput( t_bsp* bsp, const
   InputDescription& descr ) {
2     Data* input = new Data( descr.toInt(), descr );
3     if( bsp_pid(bsp)==0 ) {
4       char* data = input->getData();
5       for( int i=0; i<descr.toInt(); i++ )
6         data[i] = rand() & 0xff;
7     }
8     return input;
9  }
```

Auch die letzte Methode des Problems dient nur zum Auffinden von Fehlern in Broadcast-Algorithmen:

```
1  unsigned int BroadcastProblem::checkResult( t_bsp* bsp, Data* result )
   {
2    int error = 0, res;
3    if( bsp_pid(bsp)==0 ) {
4      int i;
5      bsp_sync( bsp );
6      for( i=0; i<bsp_nmsgs(bsp); i++ ) {
7        t_bspmsg* msg = bsp_getmsg( bsp, i );
8        if( bspmsg_size(msg)!=(int) result->getSize() )
9          error++;
10       else
11         if( memcmp( result->getData(), bspmsg_data(msg),
   bspmsg_size(msg)) )
12           error++;
13     }
14   } else {
15     bsp_send( bsp, 0, result->getData(), result->getSize() );
16     bsp_sync( bsp );
17   }
18   bsp_allreduce( bsp, 0, 0, bsp_nprocs(bsp)-1, &error, &res, 1,
   &bspop_addint, NULL );
19   bsp_sync( bsp );
20   return (unsigned int) res;
21 }
```

Zuerst schickt jeder Prozessor seine Daten an Prozessor 0. Dieser vergleicht dann die Länge und den Inhalt der empfangenen Nachrichten mit seinen Daten und zählt die Fehler. Diese werden anschließend an alle Prozessoren geschickt. Statt alle Fehler der Prozessoren mittels `bsp_allreduce` zu addieren, könnte man hier natürlich auch einfach die Fehler von Prozessor 0 an alle schicken. Dieses Beispiel zeigt jedoch eine einfache Möglichkeit, die Gesamtfehleranzahl auf allen Prozessoren zu erhalten, wenn auch auf anderen Prozessoren die Möglichkeit von Fehlern besteht.

Die beiden Methoden `createRandomInput` und `checkResult` werden nur zum Testen und Überprüfen von Fehlern benutzt, daher kommt es hierbei nicht auf eine möglichst effiziente Implementierung an.

4.2.2. Algorithmus TreeBroadcast

```
1  class TreeBroadcast : public Algorithm
2  {
3    public:
4       TreeBroadcast( );
5       virtual QValueList<int>* validNProcs( const InputDescription&
   input, int dataProcs, int maxCalcProcs );
6       virtual unsigned int variantCount( const InputDescription& n, int
   dataProcs, int calcProcs );
7       virtual QString getVariantDescription( const InputDescription& n,
   int dataProcs, int calcProcs, unsigned int index );
8       virtual double time( const InputDescription& input, int
   dataProcs, int calcProcs, TimeFunction g, TimeFunction L, SizeFunction
   B, int variant=0 );
9       virtual int subcallCount( const InputDescription& input, int
   dataProcs, int calcProcs, int variant=0 );
10      virtual SubcallDescription* subcall( const InputDescription&
   input, int dataProcs, int calcProcs, int index, int variant=0 );
11 #ifdef PUB_MACHINE
12      virtual void execute( t_bsp* bsp, Data* input, CallTree* callTree
   );
13 #endif
14 };
```

Die einzelnen Methoden sehen wie folgt aus:

TreeBroadcast::validNProcs

```
1  QValueList<int>* TreeBroadcast::validNProcs( const InputDescription&,
   int dataProcs, int maxCalcProcs ) {
2     QValueList<int>* list = new QValueList<int>;
3     list->append( dataProcs );
4     return list;
5  }
```

Der Algorithmus funktioniert nur auf genau `dataProcs` vielen Prozessoren, es macht keinen Sinn, für einen Broadcast noch mehr Prozessoren zu benutzen. Da auch die Unterprobleme von TreeBroadcast nur wieder Broadcast sind, werden diese Unterprobleme auch nicht mehr Prozessoren benötigen.

TreeBroadcast::variantCount

```
1  unsigned int TreeBroadcast::variantCount( const InputDescription& n,
   int dataProcs, int calcProcs ) {
2     unsigned int i, count = 0;
3     for( i=2; i<=(unsigned int) dataProcs; i++ )
4       if( dataProcs % i == 0 )
5         count++;
6     return count;
7  }
```

Es werden genau so viele Varianten unterstützt, wie es mögliche Baumgrade gibt. Es gilt:

$$d \text{ ist erlaubter Grad} \Leftrightarrow d \big| \texttt{dataProcs}$$

TreeBroadcast::getVariantDescription

```
1  QString TreeBroadcast::getVariantDescription( const InputDescription&,
   int dataProcs, int, unsigned int index ) {
2     unsigned int degree=2;
3     for( ; degree<(unsigned int) dataProcs; degree++ )
4       if( dataProcs % degree == 0 )
5         if( !index-- )
6           break;
7     return "degree " + QString::number(degree);
8  }
```

Diese Methode beschreibt die einzelnen Varianten mit einem Namen. Dies ist übersichtlicher, wenn man sich ein Schedule anschauen will.

TreeBroadcast::time

```
1  double TreeBroadcast::time( const InputDescription& input, int
   dataProcs, int calcProcs, TimeFunction g, TimeFunction L, SizeFunction
   B, int variant ) {
2     double n = input.toDouble();
3     unsigned int degree=2;
4     for( degree=2; degree<(unsigned int) dataProcs; degree++ )
5       if( dataProcs % degree == 0 )
6         if( !variant-- )
```

```
7           break;
8       double msgSize = n;
9       if( msgSize<B(dataProcs) )
10        msgSize = B(dataProcs );
11      return n*g(1)+(degree-1)*msgSize*g(dataProcs)+L(dataProcs);
12  }
```

Dies ist die wichtigste Methode, um die Laufzeit abzuschätzen. Sie gibt im BSP*-Modell an, wie lange dieser Algorithmus benötigt. Zuerst wird aus der Variante der Baumgrad berechnet (Zeilen 2-7), dann wird überprüft, ob die Nachricht kleiner als die Blockgröße B ist (Zeilen 9-10, s. BSP* Modell, Kapitel 2.2.1). Anschließend werden die Kosten zurückgegeben. TreeBroadcast teilt die Prozessoren in d gleich große Gruppen ein und schickt jedem ersten Prozessor in jeder Gruppe die Daten. Dies erzeugt bei Grad d und p Prozessoren Kosten von $(d-1) \cdot \max\{n, B\} \cdot g(p)$ zum Senden und $L(p)$ für die Synchronisation. Die zusätzlichen Kosten von $n \cdot g(1)$ kommen daher, dass alle Prozessoren, die die Daten empfangen, diese aus den Empfangspuffern von PUB in die richtige Position der Daten kopieren müssen.

TreeBroadcast::subcallCount

```
1  int TreeBroadcast::subcallCount( const InputDescription&, int
   dataProcs, int calcProcs, int variant ) {
2     unsigned int degree=2;
3     for( degree=2; degree<(unsigned int) dataProcs; degree++ )
4       if( dataProcs % degree == 0 )
5         if( !variant-- )
6           break;
7     return (dataProcs/degree>1)?1:0;
8  }
```

Die Anzahl der Unterprobleme ist nur von der Größe der gebildeten Untergruppen abhängig. Also wird zuerst wieder der Grad bestimmt, anschließend ausgerechnet, wie groß die Untergruppen werden. Falls sie größer als ein Prozessor sind, gibt es ein Unterproblem, sonst nicht.

TreeBroadcast::subcall

```
1  Algorithm::SubcallDescription* TreeBroadcast::subcall( const
   InputDescription& input, int dataProcs, int calcProcs, int index,
   int variant ) {
```

```
2     unsigned int degree=2;
3     for( degree=2; degree<(unsigned int) dataProcs; degree++ )
4       if( dataProcs % degree == 0 )
5         if( !variant-- )
6           break;
7     SubcallDescription* subcall = new SubcallDescription( "Broadcast",
   dataProcs/degree, input, 1, degree);
8     return subcall;
9  }
```

Das einzige Unterproblem ist wieder ein Broadcast. Dieses Unterproblem hat die Daten auf `dataProcs/degree` Prozessoren und muss einmal nacheinander, aber `degree`-mal parallel ausgeführt werden.

TreeBroadcast::execute

```
1   void TreeBroadcast::execute( t_bsp* bsp, Data* input, CallTree*
    callTree ) {
2     unsigned int dataProcs = callTree->getDataProcs();
3     unsigned int degree = 2;
4     unsigned int variant = callTree->getVariant();
5     for( degree=2; degree<(unsigned int) dataProcs; degree++ )
6       if( dataProcs % degree == 0 )
7         if( !variant-- )
8           break;
9     unsigned int partSize = dataProcs/degree;
10    if( bsp_pid(bsp)==0 ) {
11      unsigned int dest = partSize;
12      while( dest<dataProcs ) {
13        bsp_hpsend( bsp, dest, input->getData(), input->getSize() );
14        dest += partSize;
15      }
16    }
17    bsp_sync( bsp );
18    if( bsp_nmsgs(bsp)>0 ) {
19      t_bspmsg* msg = bsp_getmsg( bsp, 0 );
20      input->setData( bspmsg_data(msg), bspmsg_size(msg) );
21    }
22    if( partSize>1 ) {
23      t_bsp subBsp;
24      split_bsp( bsp, &subBsp, degree );
```

```
25        callTree->getSubcall(0)->getAlgorithm()->execute( &subBsp,
     input, callTree->getSubcall(0) );
26        bsp_done( &subBsp );
27      }
28   }
```

Nachdem alle anderen Methoden nur das Verhalten des Algorithmus beschreiben, führt ihn diese aus. Sie bestimmt zuerst den Grad d und schickt dann von Prozessor 0 die Daten an alle ersten Prozessoren der d Partitionen.

Anschließend teilt sie das BSP-Objekt in d Unterobjekte auf (Zeile 24) und führt auf jeder Partition den im gegebenen `callTree` spezifizierten Algorithmus für das Unterproblem aus (Zeile 25).

Eingabedaten müssen in diesem Beispiel nicht kopiert/konvertiert werden, da die Eingabe des Unterproblems genau der des Hauptalgorithmus entspricht. Dies ist nicht immer so und erzeugt bei anderen Algorithmen zusätzliche Kosten.

4.3. Implementation Minimale Spannbäume

Die in Kapitel 3.3.1 beschriebenen Algorithmen sind in gleichnamigen Klassen implementiert. Zusätzlich zu den dort beschriebenen Schritten werden bei einigen Algorithmen weitere Zwischenschritte benötigt, um z.B. Daten in eine für Unterprobleme passende Form zu kopieren oder gleichmäßig über die Prozessoren zu verteilen.

Diese Schritte sind erforderlich, da die Spezifikationen zu den Problemen eine genaue Verteilung von Ein- und Ausgaben festlegen. So verlangen alle Probleme zu minimalen Spannbäumen, dass die Kanten der Eingabegraphen gleichmäßig über alle Prozessoren verteilt sind (s. Kapitel 3.3.2).

Weitere Details zu der Implementation sind den Quelltexten zu entnehmen, eine Übersicht hierzu bieten Tabellen 7 bis 9.

4.4. Probleme

Eine Übersicht über alle implementierten Probleme zeigt Tabelle 4.

4.4.1. Broadcast

Verschicken von Daten von Prozessor 0 an alle anderen Prozessoren. Einziger Parameter `size` gibt an, wie viele Bytes Prozessor 0 hat.

Eingabe: Jeder Prozessor stellt `size` Bytes Daten zur Verfügung, der Inhalt ist nur auf Prozessor 0 relevant

Ausgabe: Jeder Prozessor hat die gleichen Daten wie Prozessor 0

Tabelle 4 Implementierte Probleme

Klasse	Beschreibung
BoruvkaStepProblem	Berechnung eines Borůvkaschrittes
BroadcastProblem	verschicken von Daten von Prozessor 0 zu allen anderen
EdgeMinimumProblem	Reduce, Scan, AllReduce und AllScan als parallele Prefixoperationen mit Minimum von Graphkanten
IntegerAddPrefixProblem	Reduce, Scan, AllReduce und AllScan als parallele Prefixoperationen mit Integeraddition
IntegerSortProblem	sortieren beliebiger Daten, Ganzzahl-Schlüssel
MSTMergeProblem	Zusammenführen von lokal gespeicherten minimalen Spannbäumen zu einem globalen auf Prozessor 0
MSTProblem	Berechnung eines minimalen Spannbaumes für einen ungerichteten Graph
SegmentMinimumProblem	berechnen der Minima von segmentierten Daten
SortProlem	sortieren beliebiger Daten, beliebige Schlüssel

4.4.2. ReduceAddInt, ReduceAllAddInt, ScanAddInt und ScanAllAddInt

Berechnen der Summe von Integervektoren. Dabei gibt es verschiedene Varianten:

	normal	**all**
Reduce	nur Prozessor $p-1$ erhält die Summe	alle Prozessoren erhalten die Gesamtsumme
Scan	Prozessor i erhält Summe der Prozessoren $0, \ldots, i$	Prozessor i erhält Summe der Prozessoren $0, \ldots, i$ und zusätzlich die Gesamtsumme

Der einzige Parameter der Problembeschreibung gibt die Anzahl der Integerzahlen pro Prozessor an, die Datengrößen sind $j \cdot \text{sizeof(int)} \cdot \text{size}$ mit $j = 3$ für ScanAllAddInt und $j = 2$ sonst. Im ersten Bereich steht die Eingabe. Die anderen Bereiche werden überschrieben, bei ReduceAddInt ist der Inhalt des zweiten Teils nach der Berechnung nicht definiert.

4.4.3. IntegerSort

Sortieren von Zahlen mittels einen Integerschlüssels. Die Integerschlüssel sind jeweils vor jedem Datensatz gespeichert, so dass im Speicher immer einem Schlüssel ein

Datensatz folgt.

Parameter	Erklärung
Anzahl	Anzahl der Datenelement
[Datengröße]	Größe pro Element, default sizeof(int)

4.4.4. Sort

Allgemeines Sortieren von Daten beliebiger Größe mit einer benutzerdefinierten Sortierfunktion.

Parameter	Erklärung
Elementgröße	Größe eines Datenelementes in Bytes
Anzahl	Anzahl der Datenelemente
Ordnung	Reihenfolge, 0 entspricht aufsteigend, $\neq 0$ absteigend
[Vergleich]	Zeiger auf eine Vergleichsfunktion, default Integer-Vergleich, dieser nimmt die ersten sizeof(int) Bytes der Daten als Integer-Schlüssel

4.4.5. MST

Berechnung minimaler Spannbäume (engl. minimal spanning trees, MST) von ungerichteten Graphen. Die Daten sind gleichmäßig über alle Prozessoren verteilt.

Parameter	Erklärung
Knoten	Anzahl der Knoten des Graphens
Kanten	Gesamtanzahl der Kanten

Die Daten selber sind als Array von Kanten gespeichert, jeder Eintrag besteht aus den drei Elementen:

Eintrag	Typ	Erklärung
Startknoten	int	Startknoten
Zielknoten	int	Zielknoten, Ziel und Start sind austauschbar, da es sich um ungerichtete Graphen handelt
Gewicht	double	Kosten der Kante

Nach den Graph-Kanten muss Platz für Knoten-1 Einträge sein (gleichmäßig verteilt über alle Prozessoren). Hier wird das Ergebnis gespeichert.

4.4.6. BorůvkaStep

Ausführung eines Borůvka-Schrittes, dieser wählt aus einem Graphen Kanten aus, die zu einem minimalen Spannbaum gehören und fasst Knoten des Graphens zu Superknoten zusammen.

Parameter	Erklärung
Knoten	Anzahl der Knoten des Graphens
Kanten	Gesamtanzahl der Kanten
Superknoten	Anzahl der Superknoten
Baumkanten	Anzahl der schon gewählten Kanten in diesem Knoten

Die Daten bestehen aus folgenden drei Teilen:

(i) der Ausgangsgraph als Array von Startknoten, Zielknoten und Gewicht, s. MST

(ii) Platz für $\lceil \text{Knoten}/p \rceil$ Kanten, davon sind `Baumkanten` viele schon mit den vorher gewählten MST-Kanten belegt, der Rest ist für das Ergebnis

(iii) Array von Integerzahlen, die zu jedem Knoten angeben, zu welchem Superknoten er gehört

4.4.7. MSTMerge

Dieses Problem führt lokale minimale Spannbäume zusammen und berechnet daraus den globalen minimalen Spannbaum. Dieser wird anschließend in Prozessor 0 gespeichert.

Die Eingabedaten sind gleichmäßig über alle Prozessoren verteilt und als Kanten analog zu MST gespeichert.

Parameter	Erklärung
Knoten	Anzahl der Knoten des Graphens
Kanten	Gesamtanzahl der Kanten

Nach den Eingabedaten muss auf jedem Prozessor Platz für Knoten-1 Kanten sein, der Inhalt dieses Speichers ist anschließend nur für Prozessor 0 definiert.

Tabelle 5 Implementierte Algorithmen

Klasse	Problem	Literatur
BoruvkaStep	BoruvkaStep	[Göt98], BSP-B1
DenseBoruvkaStep	BoruvkaStep	[ADJ+98], Schritt (2) des Algorithmus MST-DENSE
DistributedWithMasterSort	Sort	Brückenalgorithmus
EdgeMinimumTreePrefix	EdgeMinimum	par. Prefix mit Minimum von Kanten
IntegerSortWithSort	IntegerSort	Brückenalgorithmus
Kruskal	MST (seq.)	[CLR89]
MergeSort	Sort (seq.)	
MST	MST	
MSTBoruvkaAndMerge	MST	[ADJ+98], MST-DENSE mit var. Anzahl von Borůvka-Schritten
MSTMerge	MSTMerge	[ADJ+98], Schritt 2 von MST-MERGE
MSTWithMerge	MST	[ADJ+98], MST-MERGE
RadixSort	IntegerSort	[GV96]
RootBroadcast	Broadcast	[JKMR00], Schritt von HPRAM-BCAST
ScatterGatherBroadcast	Broadcast	[JW96a], 2D-BCAST
SegmentMinimum	SegmentMinimum	[Göt98], Kapitel 4.1.4
SeqRadixSort	IntegerSort (seq.)	[Knu73]
TreeBroadcast	Broadcast	

4.5. Algorithmen

Eine Übersicht über die implementierten Algorithmen liefert Tabelle 5.

4.6. Der Quelltext

Auf der beigefügten CD sind alle Quelltexte zu den in dieser Arbeit implementierten Algorithmen enthalten. Die Tabellen 6 bis 9 ab Seite 61 zeigen eine Übersicht über die einzelnen Dateien des Modells und der Algorithmen. Das Programm CallTree-Edit befindet sich im gleichnamigen Unterverzeichnis. Weitere Informationen zur Übersetzung und zum Installieren liefert die Datei liesmich.

Tabelle 6 Übersicht über die Quelltexte: BSP-Unteraufruf-Modell

Datei	Zeilen	Beschreibung
algorithm.h	173	Basisklasse für Algorithmen
algorithm.cpp	137	
calltree.h	113	Klasse für Schedules
calltree.cpp	216	
configfile.h	19	Behandlung von Konf.-Dateien
configfile.cpp	61	
data.h	148	Kapselung von Ein-/Ausgabedaten
data.cpp	158	
eps.h	58	EPS-Export-Klasse
eps.cpp	202	
inputdescription.h	132	Kapslung von Ein-/Ausgabebeschreibungen
inputdescription.cpp	461	
problem.h	121	Basisklasse für Probleme
problem.cpp	123	
problemlist.h	61	Liste von Probleme
problemlist.cpp	60	
addalgorithm.h	20	Hinzufügen aller Algorithmen
addalgorithm.cpp	127	zu einer ProblemList

Tabelle 7 Übersicht über die Quelltexte: Broadcast- und Prefix-Algorithmen

Datei	Zeilen	Beschreibung
broadcastproblem.h	30	Problem Broadcast
broadcastproblem.cpp	60	
rootbroadcast.h	27	Algorithmus RootBroadcast
rootbroadcast.cpp	120	
scattergatherbroadcast.h	47	Algorithmus ScatterGatherBroadcast
scattergatherbroadcast.cpp	203	
treebroadcast.h	29	Algorithmus TreeBroadcast
treebroadcast.cpp	130	
integeraddprefixproblem.h	28	Problem Paraller Prefix mit
integeraddprefixproblem.cpp	122	mit Integeraddition
integeraddtreeprefix.h	24	Intergeraddition als Prefixbaum
integeraddtreeprefix.cpp	35	
segmentminimum.h	26	Algorithmus SegmentMinimum
segmentminimum.cpp	358	
segmentminimumproblem.h	54	Problem SegmentMinimum
segmentminimumproblem.cpp	54	
treeprefix.h	33	Basisklasse für Parallel-Prefix-Algor.
treeprefix.cpp	250	

Tabelle 8 Übersicht über die Quelltexte: MST-Algorithmen

Datei	Zeilen	Beschreibung
boruvkastep.h	30	Algorithmus BoruvkaStep
boruvkastep.cpp	357	
boruvkastepproblem.h	37	Problem BoruvkaStep
boruvkastepproblem.cpp	39	
denseboruvkastep.h	27	Algorithmus DenseBoruvkaStep
denseboruvkastep.cpp	267	
edgeminimumproblem.h	28	Problem EdgeMinumum
edgeminimumproblem.cpp	135	
edgeminimumtreeprefix.h	24	Algorithmus EdgeMinimumTreePrefix
edgeminimumtreeprefix.cpp	40	
kruskal.h	27	Algorithmus Kruskal
kruskal.cpp	126	
mst.h	26	Algorithmus MST
mst.cpp	126	
mstboruvkaandmerge.h	30	Algorithmus MSTBoruvkaAndMerge
mstboruvkaandmerge.cpp	328	
mstmerge.h	32	Algorithmus MSTMerge
mstmerge.cpp	92	
mstproblem.h	38	Problem MST
mstproblem.cpp	299	
mstwithmerge.h	27	Algorithmus MSTWithMerge
mstwithmerge.cpp	127	
sets.h	93	Hilfsklasse Mengen

Tabelle 9 Übersicht über die Quelltexte: Sortier-Algorithmen

Datei	Zeilen	Beschreibung
distributedwithmastersort.h	28	Algorithmus Sortieren durch
distributedwithmastersort.cpp	112	seq. Sortieren
integersortproblem.h	30	Problem Sortieren mit Ganzzahlschlüssel
integersortproblem.cpp	78	
integersortwithsort.h	28	Alg. Intergersortieren mit
integersortwithsort.cpp	66	normalem Sortieren
integerwithmastersort.h	28	Algorithmus Integer-Sortieren durch
integerwithmastersort.cpp	110	seq. Integersortieren
mergesort.h	28	Algorithmus sequ. Mergesort
mergesort.cpp	113	
radixsort.h	28	Algorithmus RadixSort
radixsort.cpp	198	
seqradixsort.h	27	Algorithmus SeqRadixSort
seqradixsort.cpp	96	
sortproblem.h	33	Problem Sortieren beliebiger Daten
sortproblem.cpp	98	

Abbildung 17 Eingabe der Problemgröße

calltreeedit

nodes:	4096
edges:	32768
data procs:	16
calc procs:	16

ok

Abbildung 18 Hauptfenster CallTreeEdit

CalltreeEdit 0.1

File Edit View Help

algorithm	variant	input	index	data procs	calc procs	count	parallel	bsp time
⊟ MSTBoruvkaAndMerge	1 BoruvkaSteps	4096, 32768	1	16	16	1	1	7.70992e+07
⊟ DenseBoruvkaStep		4096, 32768, 4096, 0	1	16	16	1	1	2.55913e+07
EdgeTreeReduceAll		4096	1	16	16	1	1	1.87865e+07
ScatterGatherBroadcast	16 blocks	4096	2	16	16	1	1	2.48755e+06
TreeScanAll		1	3	16	16	1	1	3.82062e+06
TreeScan		1	2	16	16	1	1	3.82062e+06
⊞ Kruskal		2048, 32768	3	1	1	1	16	1.0879e+07
⊟ MSTMerge	degree 2	2048, 32752	4	16	16	1	1	2.19407e+07
⊟ MSTMerge	degree 2	2048, 16376	1	8	8	1	2	1.40381e+07
⊟ MSTMerge	degree 2	2048, 8188	1	4	4	1	2	5.85092e+06
⊞ MSTMerge	degree 2	2048, 4094	1	2	2	1	2	2.83032e+06
⊞ Kruskal		2048, 4096	2	1	1	1	1	1.11411e+06
⊞ Kruskal		2048, 4096	2	1	1	1	1	1.11411e+06
⊞ Kruskal		2048, 4096	2	1	1	1	1	1.11411e+06

Ready.

4.7. Das Programm CallTreeEdit

Das Programm CallTreeEdit dient dazu, Schedules manuell zu erzeugen. Es wurde hier vor allem für die Messungen benutzt, um bestimmte vorgegebene Algorithmen mit den optimalen zu vergleichen.

5. Zusammenfassung und Ausblick

5.1. Die Ergebnisse

In dieser Arbeit wurde ein System entwickelt, um aus einer Sammlung von Algorithmen inklusive deren Beschreibung einen für einen gegebenen BSP-Computer optimal konfigurierten Algorithmus anzugeben. Dazu wurde zuerst ein Modell entworfen und der Begriff des Schedules zur Lösung von Problemen definiert. Anschließend wurde ein Kostenmaß eingeführt und das Modell und die erzeugten optimalen Algorithmen an Hand der Beispiele Broadcast und Berechnung von minimalen Spannbäumen untersucht.

Es hat sich gezeigt, dass das Modell zwar nicht immer zu Algorithmen führt, die auch in der Realität eine minimale Laufzeit haben, dieses ist aber ein allgemeines Problem bei universell einsetzbaren Bibliotheken. Hier muß fast immer ein Kompromiss zwischen Komfort und Benutzbarkeit auf der einen Seite und der Performance andererseits gefunden werden.

5.2. Offene Probleme

Im Umfeld dieser Arbeit gibt es eine Vielzahl weiterer Fragestellungen, von denen ich hier nur einige aufzählen möchte:

Experimente auf weiteren BSP-Computern: Die Messungen in Kapitel 3 haben gezeigt, dass sich der hierbei benutzte Computer nicht immer verhält, wie das BSP-Modell vorhersagt. Ergebnisse anderer Arbeiten im Umfeld der PUB-Library (Benchmark-Tests der Kommunikationsfunktionen, s. [BJHR00a]) haben gezeigt, dass andere Computer weniger Annormalitäten zeigen. So weisen z.B. auf einer Cray T3E gemessene Laufzeiten eine deutlich geringe Varianz im Vergleich zum PSC2 auf.

Erweiterungen am BSP-Modell: Da das BSP-Modell auf einigen Computern ungenaue Vorhersagen liefert, könnte man weitere Erweiterungen untersuchen. Insbesondere das E-BSP-Modell ([JW96b]), welches die unbalancierte Kommunikationsmuster berücksichtigt, könnte zu genaueren Zeiten führen.

Ein weiteres zu untersuchendes Modell ist Oblivious-BSP ([GLP+00]), in dem eine auch von der PUB-Library unterstützte schnellere Art der Synchronisation betrachtet wird.

Online-Scheduler: Da die Eingabegrößen für Unterprobleme nicht immer genau abgeschätzt werden können, kann es sinnvoll sein, das Schedule während der Ausführung des Algorithmus anzupassen. So könnte man vor Unterprogrammaufrufen für dieses Unterproblem mit den genauen Eingabedaten erneut ein optimales Schedule berechnen. Hierfür wird dann ein schnellerer Scheduler benötigt. Außerdem ist zu klären, bei welchen Unterproblemen neu optimiert wird, z.B. wenn die Abweichung zwischen Vorhersage der Eingaben von Realität zu groß ist.

Potenzierung in endlichen Körpern: Eine interessante Fragestellung ergibt sich aus dem Problem der Berechnung von Potenzen in endlichen Körpern. Nöcker untersucht in [Nöc01] verschiedene Datenstrukturen. Bei diesen Berechnungen sind jeweils zwei Algorithmen beteiligt: Der Potenzierungsalgorithmus und ein Algorithmus für die Multiplikationen. Durch Wahl der Datenstruktur kann man das Kostenverhältnis zwischen Multiplizieren und Quadrieren in endlichen Körpern beeinflussen. Eine offene Frage ist es, wie man die Prozessoren auf die beiden Algorithmen verteilt, d.h. die viele Prozessoren man für die Multiplikationen benutzt.

Dieses Problem sollte sich mit dem in dieser Arbeit vorgestellten System untersuchen lassen, erste Analysen zeigen allerdings, dass sich durch die vielen Varianten bei diesem Problem der Suchraum für das optimale Schedule so sehr vergrößert, dass der in Kapitel 4.1.3 vorgestellte Scheduler zu langsam ist.

Betrachtung von rekonfigurierbarer Hardware: Durch die Verwendung von *Field-Programmable Gate Arrays* (FPGAs) ist es möglich, Unteralgorithmen direkt in Hardware auf einem konfigurierbaren Co-Prozessor auszuführen. Das in dieser Arbeit vorgestellte BSP-Unteraufruf-Modell könnte erweitert werden, um auch Entscheidungen, welche Algorithmen man auf einem FPGA löst, treffen zu können. Für zwei spezielle Sortieralgorithmen wurde dieses Problem schon von Bednara et al. in [BBTW00] untersucht.

Literaturverzeichnis

[ADJ+98] Adler, Micah, Wolfgang Dittrich, Ben H.H. Juurlink, Miroslaw Kutylowski und Ingo Rieping: *Communication-optimal parallel minimum spanning tree algorithms.* Technischer Bericht tr-rsfb-98-059, Universität Paderborn, Sonderforschungsbereich 376, August 1998.

[BBTW00] Bednara, Marcus, Oliver Beyer, Jürgen Teich und Rolf Wanka: *Tradeoff Analysis and Architecture Design of a Hybrid Hardware/Software Sorter.* In: *Proc. Int. Conf. on Application Specific Systems, Architectures, and Processors (ASAP)*, Seiten 299–308, 2000.

[BDM98] Bäumker, Armin, Wolfgang Dittrich und Friedhelm Meyer auf der Heide: *Truly Efficient Parallel Algorithms: 1-optimal Multisearch for an Extension of the BSP Model.* Theoretical Computer Science, 203(2):175–203, 1998.

[BJHR00a] Bonorden, Olaf, Ben H.H. Juurlink, Nicolas Hüppelshäuser und Ingo Rieping: *The Paderborn University BSP (PUB) Library on the Cray T3E.* Projektbericht, Juni 2000.

[BJHR00b] Bonorden, Olaf, Ben H.H. Juurlink, Nicolas Hüppelshäuser und Ingo Rieping: *PUB-Library - User Guide and Function Reference*, Oktober 2000.

[BJvOR99] Bonorden, Olaf, Ben H.H. Juurlink, Ingo von Otte und Ingo Rieping: *The Paderborn University BSP (PUB) Library - Design, Implementation and Performance.* In: *International Parallel Processing Symposium.* IEEE Computer Society Press, 1999. Full version as Technical report tr-rsfb-98-063, 1998, University Paderborn.

[bsp] *BSP Worldwide.* http://www.bsp-worldwide.org.

[CDF+97] Cáceres, Edson, Frank Dehne, Afonso Ferreira, Paola Flocchini, Ingo Rieping, Alessandro Roncato, Nicola Santoro und Siang W. Song: *Efficient parallel graph algorithms for coarse grained*

multicomputers and BSP. In: *Proceedings of the 24th International Colloquium on Automata, Languages and Programming (ICALP)*, Seiten 390–400, 1997.

[CKP+93] CULLER, DAVID E., RICHARD M. KARP, DAVID A. PATTERSON, ABHIJIT SAHAY, KLAUS E. SCHAUSER, EUNICE SANTOS, RAMESH SUBRAMONIAN und THORSTEN VON EICKEN: *LogP: Towards a Realistic Model of Parallel Computation.* In: *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seiten 1–12, 1993.

[CLR89] CORMEN, THOMAS H., CHARLES E. LEISERSON und RONALD L. RIVEST: *Introduction to Algorithms.* The MIT Press and McGraw-Hill Book Company, 1989.

[DFRC93] DEHNE, FRANK, ANDREAS FABRI und ANDREW RAU-CHAPLIN: *Scaleable Parallel Geometric Algorithms for Coarse Grained Multicomputers.* In: *Proc. ACM Symposium on Computational Geometry*, Seiten 298–307, 1993.

[DG98] DEHNE, FRANK und SILVIA GÖTZ: *Practical Parallel Algorithms for Minimum Spanning Trees.* In: *Proceedings 17th IEEE Symposium on Reliable Distributed Systems, Workshop on Advances in Parallel and Distributed Systems, West Lafayette, IN, USA*, Seiten 366–371, 1998.

[DK96] DE LA TORRE, P. und C. P. KRUSKAL: *Submachine Locality in the Bulk Synchronous Setting.* In: *EUROPAR: Parallel Processing, 2nd International EURO-PAR Conference*, Seiten 352–358. LNCS, 1996.

[Eil98] EILINGHOFF, CHRISTOPH: *Systematische Konstruktion anwendungsspezifischer Werkzeugsysteme zur Entwicklung paralleler Programme.* Shaker Verlag, 1998.

[GLP+00] GONZALEZ, JESUS A., COROMOTO LEON, FABIANA PICCOLI, MARCELA PRINTISTA, JOSÉ L. RODA, CASIANO RODRIGUEZ und FRANCISCO SANDE: *Oblivious BSP.* In: *EUROPAR: Parallel Processing, 6nd International EURO-PAR Conference*, Band 1900, Seiten 682–685. Springer, 2000.

[Göt98] GÖTZ, SILVIA: *Communication-efficient parallel algorithms for minimal spanning tree computations.* Diplomarbeit, Universität Paderborn, http://www.scs.carleton.ca/∼sylvie/work.html, 1998.

[GV92] GERBESSIOTIS, ALEXANDROS V. und LESLIE G. VALIANT: *Direct Bulk-Synchronous Parallel Algorithms.* Technischer Bericht TR-10-92, Harvard University, Computer Science Department, 1992.

[GV96] GERBESSIOTIS, ALEXANDROS V. und LESLIE G. VALIANT: *Primitive Operations on the BSP Modell.* Technischer Bericht TR-23-96, Harvard University, Computer Science Department, 1996.

[HMS+98] HILL, JONATHAN M. D., BILL MCCOLL, DAN C. STEFANESCU, MARK W. GOUDREAU, KEVIN LANG, SATISH B. RAO, TORSTEN SUEL, THANASIS TSANTILAS und ROB H. BISSELING: *BSPlib: The BSP programming library.* Parallel Computing, 24(14):1947–1980, 1998.

[HR92] HEYWOOD, TODD und SANJAY RANKA: *A Practical Hierarchical Model of Parallel Computation I: The Model.* Journal of Parallel and Distributed Computing, 16:212–232, 1992.

[JKMR00] JURLINK, BEN H.H., PETR KOLMAN, FRIEDHELM MEYER AUF DER HEIDE und INGO RIEPING: *Optimal Broadcast on Parallel Locality Models.* In: *Proceedings of 7th International Colloquium on Structural Information and Communication Complexity - Sirocco 2000*, Proceedings in Informatics, Seiten 211–226. Carleton Scientific, June 2000.

[JW96a] JUURLINK, BEN H.H. und HARRY A.G. WIJSHOFF: *Communication primitives for BSP computers.* Information Processing Letters, 58:303–310, 1996.

[JW96b] JUURLINK, BEN H.H. und HARRY A.G. WIJSHOFF: *The E-BSP Model: Incorporating General Locality and Unbalanced Communication into the BSP Model.* In: *EUROPAR: Parallel Processing, 2nd International EURO-PAR Conference*, Seiten 339–347. Springer, 1996.

[Knu73] KNUTH, DONALD E.: *Searching and Sorting*, Band 3 der Reihe *The Art Of Computer Programming.* Addision-Wesley, Reading, Massachusetts, 1973.

[Kru56] KRUSKAL, JOSEPH B.: *On the Shortest Spanning Subtree of a Graph and the Travelling Salesman Problem.* In: *Proceedings of the American Mathematical Society*, 7, Seiten 48–50, Februar 1956.

[MN99] MEHLHORN, KURT und STEFAN NÄHER: *The LEDA Platform of Combinatorial and Geometric Computing.* Cambridge University Press, November 1999.

[Neu93] NEUMANN, JOHN VON: *First Draft of a Report on the EDVAC.* IEEE Annals of the History of Computing, 15(4):27–75, 1993. Edited and corrected by Michael D. Godfrey.

[Nöc01] NÖCKER, MICHAEL: *Data structures for parallel exponentiation in finite fields.* Doktorarbeit, University Paderborn, Juni 2001.

[Pad] PADERBORN CENTER FOR PARALLEL COMPUTING: *Beschreibung des Computers hpcLine PSC2.* http://www.upb.de/pc2/services/systems/psc/.

[pub] *Paderborn University BSP Library.* http://www.upb.de/∼pub.

[Rie00] RIEPING, INGO: *Communication in Parallel Systems - Models, Alogrithms and Implementations*, Band 81 der Reihe *HNI-Verlagsschriftenreihe.* 2000.

[Tro01] TROLLTECH AS: *QT 2.3 online reference documentation.* http://doc.trolltech.com/2.3/index.html, 2001.

[Val90] VALIANT, LESLIE: *A Bridging Model for Parallel Computation.* Communications of the ACM, 33(8):103–111, August 1990.

Index

A

Algorithmus
- Übersicht ... 60
- BoruvkaStep ... 29
- DenseBoruvkaStep ... 29
- Klasse ... 46
- Kruskal ... 27
- MST ... 30
- MSTBoruvkaAndMerge ... 30
- MSTMerge ... 29
- MSTWithMerge ... 30
- Root ... 19
- ScatterGather ... 19
- Tree ... 18

Algorithmusbeschreibung ... 10

B

Bandbreite ... 7
BorůvkaStep ... 29
Broadcast ... 17
- Übersicht Algorithmen ... 18
- Laufzeitvorhersagen ... 20
- Messungen ... 22

BSP
- Computer ... 5

BSP*-Modell ... 8
BSP-Modell ... 5
BSP-Untergruppen ... 9

C

CPU-Leistung ... 7

D

D-BSP-Modell ... 8
DenseBoruvkaStep ... 29

E

Encapsulated PostScript ... 45

G

Gültiges Schedule ... 12
- Beispiel Broadcast ... 13
- Beispiel MST ... 35

K

Kantengewichte ... 37
Kommunikationskosten ... 7
Kosten
- BSP-Modell ... 7
- Schedule ... 13
- Superstep ... 7

Kruskal-Algorithmus ... 27

L

Laufzeitvorhersagen
- Broadcast ... 20
- MST ... 34

Liste ... 18
lokale Arbeit ... 7
Lokalität ... 8

M

massiv parallel ... 1
Minimaler Spannbaum ... 26
- Kruskal-Algorithmus ... 27

Modell
- BSP ... 5
- BSP* ... 8
- BSP-Unteraufruf ... 9
- D-BSP ... 8

MST-Algorithmus ... 30

MSTBoruvkaAndMerge 30
MSTMerge 29
MSTWithMerge 30

N

Neuman, John von 1

P

Paketgröße 8
Problem
    Übersicht 57
    Klasse 45
PUB-Library 14

R

Root 19

S

ScatterGather 19
Schedule
    gültiges 12
    Kosten 13
Spannbaum 26
Superstep
    Kosten 7
Supersteps 6
Synchronisation 7

T

Topologie 15
Tree 18
TreeBroadcast
    Algorithmusbeschreibung 11

U

Unteraufruf-Modell 9

Danksagung

Zu danken ist Prof. Dr. Friedhelm Meyer auf der Heide und Dr. Rolf Wanka für die Betreuung dieser Arbeit. Mein Dank gilt weiterhin allen Dozenten und Übungsleitern des Fachbereiches, die mich in den Jahren meines Studiums begleitet und so die Grundlagen für diese Abschlussarbeit geschaffen haben. Besonders danke ich Dr. Ingo Rieping, der mein Interesse am BSP-Modell geweckt und mir durch zahlreiche Diskussionen bei der Arbeit an der PUB-Library viele Impulse gegeben hat.

Erklärung

Hiermit erkläre ich, dass ich die hier vorliegende Diplomarbeit selbständig verfasst und keine weiteren als die angegebenen Quellen und Hilfsmittel verwendet habe.

Paderborn, den 19. Februar 2002

____