Karlsruher Institut für Technologie

Kommunikationseffizientes Branch-and-Bound für das Rucksackproblem

Bachelorarbeit von

Collin Lorbeer

03. April 2020

Erstgutachter:	Prof. Dr. Peter Sanders
Zweitgutachter:	Prof. Dr. Dorothea Wagner
Betreuender Mitarbeiter:	M.Sc. Lorenz Hübschle-Schneider

Institut für Theoretische Informatik
Fakultät für Informatik
Karlsruher Institut für Technologie

Collin Loubeau

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

Karlsruhe, den 03.04.2020

Zusammenfassung

Das Rucksackproblem ist eines der bekanntesten NP-vollständigen Probleme der Informatik. In dieser Arbeit soll das Problem mit Hilfe des so genannten Branch-and-Bound-Verfahrens gelöst werden. Dabei wird das Problem so lange in kleinere Teilprobleme aufgeteilt, bis die einzelnen Teilprobleme trivial lösbar sind. Das Verfahren wird in dieser Arbeit parallelisiert. Dies geschieht dadurch, dass jeder Prozessor eine lokale Prioritätswarteschlange erhält, in welcher die einzelnen Teilprobleme verwaltet werden. Bei der Parallelisierung von Algorithmen stößt man unweigerlich auf die Frage, wie viel Kommunikation für eine gute Lastbalancierung notwendig ist. Für verteilte Prioritätswarteschlangen findet man in der Literatur häufig den Ansatz, neue Elemente – in unserem Fall Teilprobleme – an einen zufällig ausgewählten Prozessor zu senden. Allerdings nimmt das Senden und Empfangen der Elemente viel Zeit in Anspruch. Wir verfolgen daher den Ansatz, neu entstandene Elemente zunächst in die lokale Prioritätswarteschlange einzufügen. Nur im Falle einer stark ausgeprägten Lastimbalance nehmen wir eine Umverteilung vor. Dadurch erhalten wir signifikante Laufzeitverbesserungen.

Abstract

The Knapsack problem is one of the most important NP-complete problems in computer science. This thesis will focus on a common solving strategy, called Branch and Bound. Using this strategy, the problem is divided into subproblems until the subproblems can be solved trivially. We will parallelize this strategy in our work. Therefore, every process gets a local priority queue where it manages several subproblems. When parallelizing a program, one might ask how much communication between the processes is necessary to achieve balance load. Many authors recommend to send new elements, which represent subproblems in our work, to a random process. The selected process then adds the new elements to its local priority queue. Unfortunately, sending elements to other processes takes much time and leads to a high communication overhead. Therefore this work pursues the strategy that every process stores new elements in its local priority queue. Sending elements to another process will only occur at high imbalance. Using this strategy, significant speedups in runtime can be achieved.

Danksagungen

Mein Dank richtet sich vor allem an meinen Betreuer Lorenz Hübschle-Schneider für seine stetige Unterstützung während meiner Bachelorarbeit. Des Weiteren danke ich Timo Bingmann für die Bereitstellung wichtiger Implementierungen durch die tlx-Bibliothek. Zuletzt danke ich für die Bereitstellung des Rechenclusters bwUniCluster, durch welches ich meine Implementierungen auf einer großen Anzahl Prozessoren auswerten konnte.

Inhaltsverzeichnis

Abbildungsverzeichnis

Algorithmenverzeichnis

1 Einleitung

Mit dem Problem, die Taktrate von Prozessoren nicht mehr signifikant erhöhen zu können, gewinnen parallele Prozesse in unserer heutigen Zeit mehr und mehr an Bedeutung. Daher liegt es nahe, auch für eines der bekanntesten NP-schweren Probleme, dem Rucksackproblem, parallele Algorithmen zu betrachten. Beim Rucksackproblem geht es darum, einen Rucksack mit begrenzter Kapazität möglichst gewinnbringend zu füllen. Dazu stehen verschiedene Gegenstände zur Verfügung. Jeder dieser Gegenstände besitzt ein Gewicht und einen Profit. Ziel ist es, den Rucksack so zu befüllen, dass der Profit maximal wird und gleichzeitig die Summe der Gewichte der eingepackten Gegenstände die Kapazität des Rucksacks nicht übersteigt. Es gibt verschiedene Ansätze, das Problem zu lösen. In dieser Arbeit soll ein Branch-and-Bound-Ansatz verwendet werden. Dieser basiert darauf, dass man stets die bestmöglich erscheinende Entscheidung trifft. Dafür benötigt man als wichtige Datenstruktur eine Prioritätswarteschlange. Es soll gezeigt werden, dass sich durch die Verwendung einer kommunikationseffizienten Prioritätswarteschlange sehr gute Speedups beim parallelen Lösen von Instanzen des Rucksackproblems erreichen lassen.

1.1 Ergebnisse

Betrachtet man die bisher vorhandene Literatur zu verteilten Prioritätswarteschlangen, so werden neu einzufügende Elemente meist an einen zufälligen Knoten gesendet (vgl. Karp und Zhang [1993a] sowie Karp und Zhang [1993b]). Dadurch entsteht jedoch ein nicht zu vernachlässigender Overhead. Diese Arbeit zeigt, dass es ausreicht, neue Elemente zunächst in die lokale Prioritätswarteschlange einzufügen. Nur im Falle einer hinreichend großen Unausgeglichenheit in der Lastverteilung sollen Elemente an andere Knoten gesendet werden.

1.2 Übersicht

Wir werden uns in Abschnitt 2 zunächst mit dem Rucksackproblem beschäftigen. Hierzu wird zunächst das Problem formal definiert werden. Anschließend gehen wir auf die bisher existierenden Ansätze, Instanzen des Rucksackproblems zu lösen, ein. Dabei werden die jeweiligen Vor- und Nachteile der einzelnen Ansätze beleuchtet.
In Abschnitt 3 wird auf die für diese Arbeit wichtigen Algorithmen und Datenstrukturen eingegangen. Zunächst geht es um eine genauere Betrachtung des Maschinenmodells. Anschließend beschäftigen wir uns mit der verwendeten Prioritätswarteschlange sowie ihrer parallelen Verwaltung.
Abschnitt 4 stellt den in dieser Arbeit für die Experimente verwendeten Algorithmus vor. Die Algorithmen werden mit Hilfe von Pseudocode ausführlich beschrieben. Zudem betrachten wir unterschiedliche Arten von Instanzen. Die darauf basierenden Experimente sowie deren Ergebnisse folgen in Abschnitt 5.

2 Verwandte Arbeiten

Um zu verstehen, wie man das Rucksackproblem lösen kann, ist zunächst eine formale Beschreibung des Problems notwendig. Hierfür verwenden wir folgende Definition:

Definition 2.1. *Instanz: Eine Instanz besteht aus einer Menge $N \in \mathbb{N}$ von Gegenständen sowie einer Kapazität $C \in \mathbb{R}^+$. Jeder Gegenstand r_j mit $1 \leq j \leq N$ besitzt ein Gewicht $w_j \in \mathbb{R}^+$ sowie einen Profit $p_j \in \mathbb{R}^+$.*

Seien weiter $x_j \in \{0, 1\}$ für alle $1 \leq j \leq N$.
Das Rucksackproblem besteht nun darin, eine Belegung der x_j zu finden, sodass der Wert

$$p = \sum_{j=1}^{N} p_j x_j$$

maximiert wird und dabei gleichzeitig die Bedingung

$$\sum_{j=1}^{N} w_j x_j \leq C$$

nicht verletzt wird.

Es gibt einige Varianten des Problems. Beispielsweise kann es sinnvoll sein, einen bestimmten Gegenstand mehrmals mitzunehmen, d.h. einem oder mehreren x_i auch Werte > 1 zuzuweisen. Eine weitere Möglichkeit besteht darin, für Gewichte, Profite und Kapazitäten nur natürliche Zahlen zuzulassen. Weitere Varianten werden durch Kellerer u. a. [2004] beziehungsweise Martello und Toth [1990] beschrieben. In dieser Arbeit beschränken wir uns jedoch auf die obige Definition. Diese wird in der Literatur (zum Beispiel durch Martello und Toth [1990]) auch häufig als 0/1-Rucksackproblem bezeichnet, da die Werte von x_i nur 0 oder 1 sein können.
Es sei angemerkt, dass das Problem (auch, wenn man die oben genannten Modifikationen betrachtet) NP-vollständig ist und somit bisher noch kein Polynomialzeitalgorithmus zur Lösung des Problems gefunden wurde. Allerdings gehört das Problem zu den leichtesten NP-vollständigen Problemen. Dies zeigt sich beispielsweise daran, dass ein Algorithmus zur Lösung des Problems existiert, welcher pseudoplynomielle Laufzeit aufweist. Zudem wurden auch einige Approximationsalgorithmen entwickelt. Dies ist keineswegs für alle NP-vollständigen Probleme möglich. Beispielsweise zeigen Arora und Barak [2009], dass SAT in Polynomialzeit lösbar ist, sofern ein Approximationsalgorithmus mit einer Güte von $7/8 + \epsilon$ mit $\epsilon > 0$ für das Problem existiert. In dieser Arbeit sind jedoch ausschließlich Algorithmen, welche eine exakte Lösung bestimmen, von Interesse. Es gibt zahlreiche exakte Algorithmen, unter anderem auch einen evolutionären von Khuri u. a. [1994]. Die wichtigsten Algorithmen zur exakten Lösung von Rucksackinstanzen sollen im Folgenden kurz vorgestellt werden.

2.1 Branch-and-Bound

Der in dieser Arbeit verwendete Algorithmus beruht auf dem Branch-and-Bound-Prinzip. Bei diesem Ansatz wird die Liste der Gegenstände einer Instanz zunächst nach der Profitdichte p_j/w_j sortiert. Anschließend bestimmen wir eine obere sowie untere Schranke für den erreichbaren Gesamtprofit. Der Branch-Teil besteht

nun darin, dass wir nach Berechnung der Schranken das Problem in zwei kleinere Teilprobleme aufteilen. Beide Teilprobleme entsprechen dem ursprünglichen Rucksackproblem mit der zusätzlichen Bedingung, dass wir im ersten Teilproblem uns darauf festgelegt haben, den Gegenstand mit der höchsten Profitdichte zu nehmen. Im zweiten Teilproblem hingegen legen wir fest, diesen Gegenstand keinesfalls mitzunehmen. Dieses Verfahren wird rekursiv fortgesetzt, bis für alle Gegenstände eine Entscheidung getroffen wurde. Dadurch entsteht ein Suchbaum der Größe 2^n. Die oberen bzw. unteren Schranken können genutzt werden, um Teile des Baums zu verwerfen. Liegt beispielsweise die obere Schranke eines Teilproblems unter dem Profitwert einer bereits gefunden Befüllung, so kann sich in diesem Teilbaum keine bessere Lösung mehr befinden. Somit müssen wir diesen Teilbaum nicht weiter betrachten. Ebenso können wir einen Teilbaum verwerfen, dessen zusätzliche Bedingung fordert, dass wir mehr Gegenstände als erlaubt in unseren Rucksack packen. Dieses Vorgehen bezeichnet den Bound-Teil des Algorithmus. Ein Beispielbaum ist in Abb. 3 dargestellt.
Dieses Vorgehen kann in Form einer Tiefensuche in Code umgesetzt werden (siehe Abschnitt 5). Ein alternatives Vorgehen beschreibt Pisinger [2002]. Hier werden zunächst solange Gegenstände mit der höchsten Profitdichte in den Rucksack gepackt, bis dieser voll ist. Ist der Rucksack bereits voll, wird zunächst ein Gegenstand entfernt und anschließend der neue Gegenstand eingefügt.
Balas und Zemel [1980] haben herausgefunden, dass die Sortierung der Gegenstände einen signifikanten Anteil an der Gesamtlaufzeit haben kann. Daher kann es sich lohnen, lediglich den gewichteten Median zu ermitteln, um so herauszufinden, welcher Gegenstand zur Ermittlung der oberen Schranke geteilt werden muss. Ein Algorithmus zur Ermittlung dieses Medians wird beispielsweise durch Korte und Vygen [2012] betrachtet.

2.2 Dynamische Programmierung

Dynamische Programmierung kann man immer dann verwenden, wenn eine optimale Lösung auf optimalen Lösungen von Teilproblemen beruht. Dies ist beim Rucksackproblem der Fall. Betrachtet man die optimale Lösung einer bestimmten Instanz, so bleibt diese auch dann noch optimal, wenn man einen Gegenstand aus dem Rucksack entfernt und gleichzeitig die Instanz inklusive Kapazität um jenen Gegenstand bzw. dessen Gewicht verringert. Die zentrale Idee ist nun, nicht von Anfang an alle Gegenstände zu betrachten, sondern die Anzahl sowie die Kapazität schrittweise zu erhöhen. Nehmen wir dazu an, dass wir bereits die optimale Lösung für ein Teilproblem, bei welchem wir einige Gegenstände entfernt sowie die Kapazität verringert haben, kennen. Formal betrachtet besteht diese Instanz aus Gegenständen $M = N \backslash \{r_0, r_1, ..., r_i\}$ sowie einer Kapazität $d = C - w$, wobei $w \in (0, C]$. Bezeichne $p(I(M, d))$ den maximalen Profit der Instanz $I(M, d)$. Fügen wir nun dieser Instanz $I(M, d)$ den Gegenstand r_j mit $0 \leq j \leq i$ hinzu, ohne die Kapazität zu verändern. Dann berechnet sich die optimale Lösung durch

$$p(I(M \cup j, d)) = \begin{cases} p(I(M, d)), & \text{falls } d < w_j. \\ \max\{p(I(M, d)), p(I(M, d - w_j)) + p_j\} & \text{falls } d \geq w_j. \end{cases} \tag{2.1}$$

Der Fall $d < w_j$ tritt ein, wenn der Rucksack für den Gegenstand zu klein ist. In diesem Fall können wir den Gegenstand nicht mitnehmen. Falls wir den Gegenstand mitnehmen können, tritt der Fall $d \geq w_j$ ein. Der Ausdruck $p(I(M, d))$ entspricht

j	1	2	3	4	5	6
p_j	6	8	3	9	4	3
w_j	2	3	4	4	5	5

(a) Beispielinstanz

$C \backslash j$	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
2	0	6	6	6	6	6	6
3	0	6	8	8	8	8	8
4	0	**6**	8	8	9	9	9
5	0	6	**14**	**14**	14	14	14
6	0	6	14	14	**15**	**15**	**15**

(b) Berechnungstabelle

Abbildung 1: Einfaches Beispiel für das Lösen einer Instanz mit Hilfe von dynamischer Programmierung

dem Profit, den wir ohne Mitnahme des Gegenstands j erhalten. Der Ausdruck $p(I(M, d - w_j)) + p_j$ hingegen entspricht dem Profit, den wir erhalten, falls wir den Gegenstand mitnehmen. Da unsere Lösung bestmöglich sein soll, wählen wir die Entscheidung mit dem höheren Profit, also das Maximum der beiden Ausdrücke.

Das genaue Vorgehen soll anhand eines einfachen Beispiels mit Hilfe einer Tabelle näher erläutert werden. Sei hierzu die Beispielinstanz aus Abb. 1a gegeben. Wir setzen die zulässige Kapazität auf 6. Um die Instanz zu lösen, werden nach und nach die Werte in Abb. 1b ermittelt. Die Berechnung findet spaltenweise von links nach rechts und innerhalb einer Spalte von oben nach unten statt. Möchte man einen neuen, nichttrivialen Eintrag der Tabelle ermitteln, so müssen nur zwei Einträge berücksichtigt werden. Exemplarisch betrachten wir den Eintrag $(C = 6, j = 4)$. Nehmen wir zunächst an, dass wir den Gegenstand nehmen. Dann müssen wir den Eintrag an der Stelle $(C = 6 - w_4, j = 4 - 1)$ betrachten, welcher $(C = 2, j = 3)$ entspricht. Den dortigen Eintrag addieren wir auf p_4 und erhalten als Profit 15. Nehmen wir nun an, dass wir den Gegenstand nicht nehmen. Dann müssen wir den Eintrag $(C = 6, j = 3)$ betrachten und erhalten 14. Den neuen Tabelleneintrag setzen wir im Anschluss auf das Maximum (hier also $\max\{14, 15\} = 15$). Sollte es bei unserer Berechnung vorkommen, dass nach Einträgen (C, j) mit $C < 0 \vee j < 0$ gefragt wird, kann mit dem Wert 0 geantwortet werden. Ist die Tabelle vollständig ermittelt, befindet sich die optimale Lösung im unteren, rechten Eintrag.

Auch für diesen Ansatz gibt es verschiedene Umsetzungsmöglichkeiten. Einige Optionen, wie der Word RAM Algorithmus, werden durch Kellerer u. a. [2004] vorgestellt. Ein weiterer Algorithmus, der so genannte Nemhauser-Ullmann-Algorithmus, wird durch Korte und Vygen [2012] beschrieben.

2.3 Reduktion auf andere NP-vollständige Probleme

Da das Rucksackproblem NP-vollständig ist, kann man es auch mit Hilfe einer polynomiellen Reduktion auf andere NP-vollständige Probleme lösen. Exemplarisch soll hier das Problem Longest Path betrachtet und wie von Scheithauer [2018] beschrie-

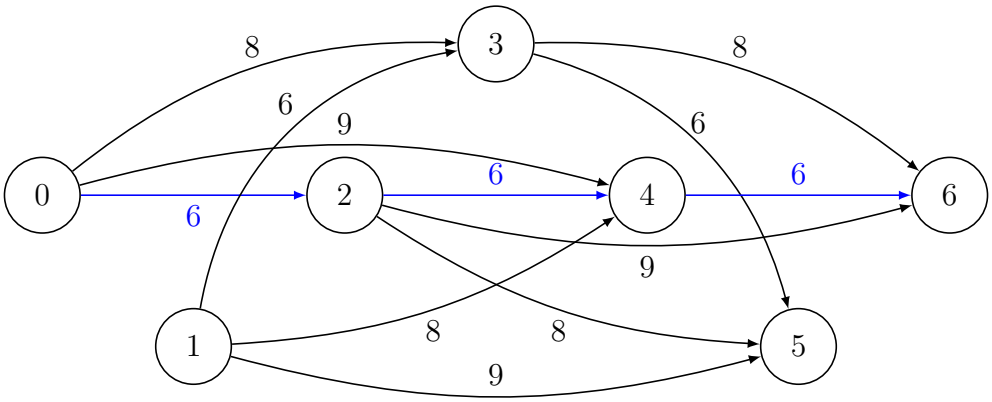


Abbildung 2: Graph einer einfachen Rucksackinstanz. Der resultierende längste Pfad ist blau eingezeichnet.

ben vorgestellt werden. Vereinfachend beschränken wir uns hier auf eine Modifikation des Rucksackproblems: Wir erlauben, dass Gegenstände mehrfach mitgenommen werden dürfen – also x_i Werte > 1 annehmen darf – und zudem Profite, Gewichte sowie die Kapazität natürliche Zahlen sein müssen. Für die Transformation müssen wir eine konkrete Instanz des Rucksackproblems als Graph darstellen und auf diesem Graphen dann den längsten Pfad finden. Dies kann wie folgt geschehen:

(i) Erstelle Knoten $V := \{V_1, V_2, ..., V_C\}$, wobei C der Kapazität des Rucksacks entspricht.

(ii) Erstelle Kanten $E := \{(j,k) \in V \times V : \exists\, i \in [1,N] \text{ mit } j + w_i = k\}$, wobei N die Menge der Gegenstände darstellt. Setze das Gewicht der Kante auf p_i

Als Beispiel sei eine Instanz gegeben, welche aus Gegenständen mit $(p_1 = 6, w_1 = 2)$, $(p_2 = 8, w_2 = 3)$ und $(p_3 = 9, w_3 = 4)$ besteht. Die Kapazität sei auf 6 gesetzt. Der zugehörige Graph ist in Abb. 2 visualisiert.

Wir stellen fest, dass der längste Pfad im Graphen Länge 18 hat und dabei drei Kanten der Länge 6 verwendet. Die Kanten verbinden jeweils Knoten V_i und V_{i+2}. Somit müssen wir in unseren Rucksack dreimal den Gegenstand mit Profit 6 und Gewicht $(i+2) - i = 2$ einpacken. Der optimale Gesamtprofit entspricht der Länge des längsten Pfades.

2.4 Das Kernkonzept

Dieses Konzept ist kein eigenständiges Lösungsverfahren, sondern stellt vielmehr eine Modifikation der zuvor vorgestellten Lösungsmethoden dar. Das Konzept beruht auf folgender Beobachtung: Betrachtet man eine konkrete Instanz, so stellt man in der Regel fest, dass einige Gegenstände eine sehr hohe und andere eine sehr niedrige Profitdichte aufweisen. Es scheint also sinnvoll, diese Elemente gar nicht zu betrachten, sondern das Problem auf diejenigen Elemente, welche eine mittlere Profitdichte aufweisen, den so genannten Kern, zu beschränken. Formal sind im Kern einer nach Profitdichte sortierten Instanz alle Gegenstände im Intervall $[a,b]$, wobei $a := \min\{j \mid x_j = 0\}$ und $\max\{j \mid x_j = 1\}$ und $j \in [1,N]$. Betrachten wir dazu ein konkretes Beispiel von Pisinger [1995]:

Gegenstand	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}	r_{11}	r_{12}	r_{13}	r_{14}
Schranke	1	1	1	1	1	1	1	1	1	0.35	0	0	0	0
optimal	1	1	1	1	1	1	0	0	1	1	0	1	0	0

In diesem Beispiel gilt $a = 7$, da dieser Gegenstand der erste ist, der nicht in der optimalen Lösung eingepackt wird. Weiter ist $b = 12$, da dieser der Gegenstand (von den mitgenommenen Gegenständen) derjenige mit der geringsten Profitdichte ist. Sofern die optimale Lösung bekannt ist, ist der Kern leicht zu ermitteln. Allerdings ist es teilweise gar nicht so einfach, den Kern zu finden (vgl. Pisinger [1995]). Daher gibt es neben Algorithmen, welche von einem festen Kern ausgehen, auch die Möglichkeit, Algorithmen mit dynamischem Kern zu erstellen. Man setzt den Kern initial auf den abgeschnittenen Gegenstand r_b (im obigen Beispiel r_{10}). Anschließend erweitert man den Kern um die Gegenstände r_{b-1}, r_{b+1}, r_{b-2}, usw.
Vorteilhaft bei diesem Verfahren ist, dass beispielsweise der Suchbaum beim Branch-and-Bound-Verfahren signifikant verkleinert werden kann, da der Suchbaum nur noch die Gegenstände im Kern betrachtet. Allerdings kann es passieren, dass sich im noch zu vergrößernden Kern viele Gegenstände mit sehr ähnlicher Profitdichte befinden. Für die optimale Lösung benötigt man in solchen Fällen auch Gegenstände mit niedriger Profitdichte, die allerdings zunächst gar nicht beachtet werden. Insgesamt hat man festgestellt, dass besonders Branch-and-Bound von diesem Ansatz profitiert. Ansätze, die auf dynamischer Programmierung beruhen, werden jedoch nach Kellerer u. a. [2004] kaum von der Wahl des Kerns beeinflusst. Zusätzlich spielt die Wahl der Instanzen eine große Rolle. So merkt Pisinger [1995] an, dass es für Testinstanzen üblich ist, $C = 1/2 \cdot \sum_{i=1}^{N} w_i$ zu setzen (was wir in Algorithmus 6 ebenfalls machen). Dies mache es Algorithmen, die auf dem Kernkonzept basieren, einfacher.

3 Vorbereitungen

3.1 Maschinenmodell

Im Folgenden wird ein einfaches, aber realistisches Maschinenmodell eingeführt. Wir orientieren uns hierbei an der Beschreibung eines parallelen Maschinenmodells von Sanders [1997]. Seien dazu p identische Prozessoren gegeben, die über ein Kommunikationsnetzwerk miteinander verbunden und von eins bis p durchnummeriert sind. Von besonderem Interesse ist die Kommunikation zwischen den einzelnen Prozessoren, da diese im Vergleich zu lokalen Berechnungen sehr langsam ist. Die einzelnen Prozessoren sind durch eine vollständige Verknüpfung miteinander verbunden, d.h. jeder Prozessor kann mit jedem anderen Prozessor direkt kommunizieren. Wir gehen weiter davon aus, dass die Verbindungen voll-duplex sind, d.h. ein Prozessor A kann Daten an Prozessor B senden und gleichzeitig Daten von Prozessor B empfangen. Des Weiteren nehmen wir an, dass ein Prozessor nicht mehrere Nachrichten parallel senden beziehungsweise empfangen kann.

3.2 Kommunikation zwischen einzelnen Prozessoren

Unsere Algorithmen beruhen auf dem Prinzip Single Program with Multiple Data (SPMD), d.h. es wird auf allen Prozessoren der gleiche Code auf unterschiedlichen Daten ausgeführt. Das bedeutet insbesondere, dass eine Variable v zu einer bestimmten Zeit auf verschiedenen Prozessoren unterschiedliche Werte annehmen kann. Um im späteren Pseudocode einfach darstellen zu können, was der Wert einer Variablen auf einem bestimmten Prozessor ist, verwenden wir die Notation $v@i$ (wobei $1 \leq i \leq p$). Diese beschreibt den aktuellen Wert der Variablen v auf Prozessor i. Die eigentliche Kommunikation zwischen den einzelnen Prozessoren erfolgt durch das Senden beziehungsweise Empfangen von Werten einzelner Variablen. Um nun die Werte einzelner Variablen unter den Prozessoren austauschen zu können, verwenden wir folgende Operationen:

(i) $p_i.send(a,\ p_j)$ sendet den Wert a auf Prozessor p_i an Prozessor p_j

(ii) $a \leftarrow receive(p_j)$ empfängt eine Nachricht von Prozessor d und speichert sie in a

(iii) $allreduce(a,\ op(a))$ sendet $op(a)$ an alle Prozessoren. Die Operation op kann hierbei eine beliebige Operation sein, zum Beispiel die Summe $\sum_{i=1}^{p} a@i$ oder das Minimum $\min_{i=1}^{p} a@i$.

3.3 Asynchrones Senden von Daten an einen zufälligen Prozessor

Erwähnenswert ist die Realisierung des Sendens von Daten an einen zufälligen Empfänger. Verwendet wird ein Algorithmus von Hoefler u. a. [2010]. Dieser basiert auf kollektiven, asynchronen Operationen. Asynchrone Operationen haben den Vorteil, dass die CPU während der kollektiven Operation weitere Berechnungen durchführen kann. Für unsere Anwendung bedeutet das beispielsweise, dass ein Prozess bereits Daten empfangen kann, während seine zu versendenden Daten noch nicht beim jeweiligen Empfänger angekommen sind. Dies ist ein klarer Vorteil gegenüber einem synchronen Senden. Hinzu kommt, dass man bei einem synchronen Senden stärker

darauf achten muss, keine Deadlocks zu erzeugen. So kann man bei einem synchronen Senden nicht jeden Prozess zuerst seine Daten senden lassen. Ein weiterer Vorteil ist dadurch gegeben, dass einige Prozessoren bereits Daten untereinander austauschen können, während andere noch mit lokalen Berechnungen beschäftigt sind. Es ist also nicht notwendig, dass alle Prozessore zur gleichen Zeit mit der kollektiven Operation beginnen. Der genaue Pseudocode der Operation wird ebenso von Hoefler u. a. [2010] beschrieben. Die Operation benötigt – in Bezug auf die Prozessorenanzahl – lediglich konstanten Platz sowie logarithmische Laufzeit.

3.4 Verteilte Prioritätswarteschlangen

Eine Prioritätswarteschlange P ist eine Datenstruktur, welche eine Menge von Elementen verwaltet. Jedes dieser Elemente *E(key, value)* besitzt einen Schlüssel *key*, welchen man mit anderen Schlüsseln vergleichen kann. Der Wert *value* stellt das eigentliche Element dar. Jede Prioritätswarteschlange unterstützt folgende Operationen:

(i) *P.insert(e)* fügt das Element e in die Prioritätswarteschlange ein

(ii) *P.max()* gibt das Element mit dem größten Schlüssel aus

(iii) *P.deleteMax()* entfernt das Element mit dem größten Schlüssel

Es gibt auch die Möglichkeit, anstatt des Maximums das Minimum zu löschen beziehungsweise auszugeben (vgl. Sanders u. a. [2019]). Wir werden in dieser Arbeit jedoch stets Prioritätswarteschlangen benötigen, welche das Maximum ausgeben und löschen können.

Da Parallelität in dieser Arbeit eine zentrale Rolle spielt, verwenden wir eine Modifikation der Prioritätswarteschlangen. Diese besteht darin, dass wir nicht immer nur ein größtes Element erhalten beziehungsweise löschen möchten, sondern mehrere. Details beschreiben Hübschle-Schneider und Sanders [2016]. Eine solche Prioritätswarteschlange kann man beispielsweise durch einen Heap, bei welchem in einem Knoten mehrere Elemente gespeichert werden, realisieren. Derartige Prioritätswarteschlangen werden auch als ”Bulk Priority Queue” bezeichnet.

Konkret wird die Prioritätswarteschlange in dieser Arbeit durch einen B+-Baum realisiert. Ein B+-Baum besteht in der untersten Ebene aus einer nach dem Schlüssel sortierten doppelt verketteten Liste, in welcher sich alle Elemente des Baums befinden. Darüber befindet sich eine Suchdatenstruktur, mit welcher sich Elemente in logarithmischer Zeit einfügen beziehungsweise löschen lassen. Da man zudem das Maximum sehr leicht findet – es steht an erster Stelle der verketteten Liste – stellt der B+-Baum eine effiziente Realisierung einer Prioritätswarteschlange dar. Details zur genauen Funktionsweise eines B+-Baums werden zum Beispiel durch Comer [1979] beschrieben. Zusätzlich werden wir von der Funktion Gebrauch machen, einen B+-Baum in zwei B+-Bäume aufzuteilen. Dabei wird die verkettete Liste an einer bestimmten Position k mit $0 \leq k \leq n$, wobei n die Anzahl der Elemente im Baum darstellt, geteilt. Diese Operation wird im Folgenden als *split(k)* bezeichnet, wobei *split()* dem Ausdruck $split(\lfloor n/2 \rfloor)$ entspreche. Ein Beispiel ist durch [Sanders u. a., 2019, Kapitel 7] gegeben. Zuletzt fordern wir, dass die Größe eines B+-Baums stets bekannt ist, um die Operation *rank(k)* zu ermöglichen. Mit dieser Operation kann die Anzahl der Elemente, die kleiner k sind, in logarithmischer Zeit ermit-

telt werden. Gleichzeitig kann durch Speichern der Baumgröße das Element eines bestimmten Ranges in logarithmischer Zeit gefunden werden.

3.5 Listen

Die Syntax der Listenoperationen übernehmen wir im Wesentlichen aus Sanders u. a. [2019]. Konkret benötigen wir folgende Listenoperationen für eine Liste L:

(i) *L.first()* gibt das erste Objekt der Liste aus

(ii) *L.popFront()* löscht das erste Objekt der Liste

(iii) *L.pushFront(e)* fügt das Element e am Anfang der Liste ein

(iv) *L.concat(L')* hängt Liste *L'* an Liste L an

Wir werden später häufig alle Elemente einer Liste in eine Prioritätswarteschlange einfügen. Dafür verwenden wir die Operation *P.insert(L)*, welche *P.insert(e)* für jedes Element e der Liste L aufruft. Zudem definieren wir *L.pushFront(P)* als diejenige Operation, welche alle Elemente einer Prioritätswarteschlange P in eine Liste einfügt.

3.6 Parallelisierung von Branch-and-Bound-Algorithmen

Wir werden später den Ansatz verfolgen, die Auswertung der beim Branch-and-Bound entstehenden Teilprobleme zu parallelisieren. Dafür erhält jeder Prozessor eine lokale Prioritätswarteschlange, in welcher er die einzelnen Teilprobleme verwaltet. Neben der Parallelisierung über diese Teilprobleme gibt es auch die Möglichkeit, die Menge der Gegenstände aufzuteilen. Jeder Prozess würde dann alle möglichen Lösungen für seine Menge an Gegenständen bestimmen. Diese Lösungen können im Anschluss zusammengeführt werden. Details beschreiben Horowitz und Sahni [1974]. Man kann ebenfalls den Ansatz verfolgen, den Suchbaum nur einmal aufzuteilen. Beispielsweise könnte man nur so lange umverteilen, bis jeder Prozess mindestens ein Teilproblem zu lösen hat. Allerdings ist es nur in sehr wenigen Fällen sinnvoll, die Gegenstände mit der höchsten Profitdichte nicht in den Rucksack zu packen (vgl. Sanders [1997]). Wir werden daher diesen Ansatz nicht weiter verfolgen.

3.7 Selektion global größter Elemente

Da in unserer Arbeit jeder Prozessor eine lokale Prioritätswarteschlange besitzt, ist es nicht mehr so einfach, die global größten Elemente zu ermitteln. Die Bestimmung der global größten Elemente kann zum Beispiel durch einen Algorithmus von Hübschle-Schneider und Sanders [2019] ermittelt werden. Dieser basiert auf einem Selektionsalgorithmus aus einer Publikation von Hübschle-Schneider und Sanders [2016].

4 Algorithmus

Wir werden im Folgenden den Algorithmus, welchen wir für das parallele Lösen des Rucksackproblems verwenden, vorstellen. Dazu werden wir zunächst wichtige Begriffe einführen sowie grundlegende Algorithmen, welche durch den Löser verwendet werden, vorstellen. In Abschnitt 4.1 wird dann die sequenzielle, in Abschnitt 4.2 die parallele Version unseres Lösers betrachtet.

Für unseren Algorithmus verwenden wir folgenden Begriff:

Definition 4.1. *Kandidat: Ein Kandidat einer Instanz $I(M, C)$ ist ein Tupel $(a_1, a_2, ..., a_n)$, wobei $a_i \in \{0, 1, \perp\}$ und $n = |M|$ die Anzahl der Gegenstände bezeichnet.*

Kandidaten werden verwendet, um Teilprobleme beim Branch-and-Bound-Verfahren einfach darstellen zu können. Interpretiert wird ein Kandidat wie folgt: Falls $a_i = 1$ gilt, so bedeutet dies, dass wir den Gegenstand an Position i in der Liste von Gegenständen einpacken. Gilt $a_i = 0$, so nehmen wir jenen Gegenstand nicht mit. Sofern $a_i = \perp$ gilt, haben wir uns noch nicht entschieden, ob wir jenen Gegenstand mitnehmen. Beispielsweise beschreibt das Tupel $(0, 1, \perp)$, dass wir den ersten Gegenstand nicht in unseren Rucksack packen, aber den zweiten. Bezüglich des dritten Gegenstands haben wir uns noch nicht entschieden.

In unserer Arbeit verfolgen wir den Ansatz, die Gegenstände zunächst nach Profitdichte zu sortieren. Bezogen auf Gegenstand r_i ist die Profitdichte durch p_i/w_i definiert, d.h. man muss den Profit des Gegenstands durch dessen Gewicht teilen. Anschließend wird ein initialer Kandidat $c_0 = (\perp, \perp, \perp, ...)$ erstellt und die bisher beste Lösung s, welche dem maximalen Gesamtprofit entspricht, auf 0 gesetzt. Wir erstellen außerdem eine leere Prioritätswarteschlange. Daraufhin berechnen wir für c_0 eine obere und untere Schranke bezüglich des Profits.
Für einen Kandidaten $c_i = (a_1, a_2, ..., a_h, \perp, \perp, ..., \perp)$ berechnet sich die obere Schranke $ub(c_i)$ beziehungsweise die untere Schranke $lb(c_i)$ wie folgt: Die untere Schranke ergibt sich daraus, dass wir solange, wie im Rucksack Platz ist, die Gegenstände mit der größten Profitdichte einpacken. Sobald wir auf einen Gegenstand r_b stoßen, welcher nicht mehr in den Rucksack hineinpasst, packen wir keine weiteren Gegenstände mehr ein. Damit erhalten wir als untere Schranke

$$lb(c_i) = \left(\sum_{l=1}^{h} a_l \cdot p_l\right) + p_{h+1} + p_{h+2} + ... + p_{b-1}.$$

Somit stellt die untere Schranke eine zulässige Belegung dar. Daher wird die bisher beste Belegung s auf diese untere Schranke gesetzt, sofern der Wert von s kleiner ist als die gefundene untere Schranke. Die obere Schranke ergibt sich daraus, dass wir auch hier zunächst die Gegenstände mit der größten Profitdichte einpacken. Sobald wir auf einen Gegenstand r_b stoßen, welcher nicht mehr in den Rucksack hineinpasst, schneiden wir ihn ab und packen denjenigen Teil $w_b \cdot t$ (wobei $t \in [0, 1)$), der noch hineinpasst, ein. Wir erhalten also

$$ub(c_i) = lb(c_i) + p_b \cdot t, \text{ wobei } t = (C - \sum_{l=1}^{h} a_l \cdot w_l - \sum_{l=h+1}^{b-1} w_l)/w_b$$

Algorithm 1: evaluate

Input: Ein Kandidat $c_i = (a_1, a_2, ..., a_h, \perp, \perp, ..., \perp)$ sowie s und $|M|$

$L \leftarrow \emptyset$ *// erstelle leere Liste von Kandidaten*

if $h = |M|$ then *// falls der Kandidat keine $\perp$-Symbole mehr beinhaltet*

  return L

$c_{i+1} \leftarrow (a_1, a_2, ..., a_h, 0, \perp, ..., \perp)$

$lb(c_{i+1}) \leftarrow \left(\sum_{l=1}^{h} a_l \cdot p_l\right) + p_{h+2} + p_{h+3} + ... + p_{b-1}$

$ub(c_{i+1}) \leftarrow lb(c_{i+1}) + p_b \cdot a$

$s \leftarrow \max(s, lb(c_{i+1}))$

if $ub(c_{i+1}) > s$ then

  $L.pushFront(c_{i+1})$

$c_{i+2} \leftarrow (a_1, a_2, ..., a_h, 1, \perp, ..., \perp)$

$lb(c_{i+2}) \leftarrow \left(\sum_{l=1}^{h} a_l \cdot p_l\right) + p_{h+1} + p_{h+2} + ... + p_{d-1}$

$ub(c_{i+2}) \leftarrow lb(c_{i+2}) + p_d \cdot a$

if $h + 1 \neq d \wedge ub(c_{i+2}) > s$ then

  s $\leftarrow \max(s, lb(c_{i+2}))$

  $L.pushFront(c_{i+2})$

Output: L sowie s

Man beachte, dass diese obere Schranke scharf ist. Um dies einzusehen, nehmen wir die Existenz einer Befüllung mit einem Profit $p(c_i) > ub(c_i)$ an. Dann muss sich diese Befüllung von der Befüllung, mit welcher man die obere Schranke erhält, unterscheiden. Damit wird also mindestens ein Gegenstand mitgenommen, dessen Profitdichte schlechter ist als die Profitdichte des Gegenstands r_b. Dadurch ist dann aber auch der Gesamtprofit schlechter, d.h. es gilt $p(c_i) < ub(c_i)$.

Der Kandidat c_0 wird daraufhin evaluiert. Die Evaluation eines Kandidaten $c_i = (a_1, a_2, ..., a_h, \perp, \perp, ..., \perp)$ wird durch den Pseudocode von Algorithmus 1 beschrieben. Die bei der Evaluation entstandenen Kandidaten werden in die Prioritätswarteschlange eingefügt. Als Schlüssel (key) verwenden wir die obere Schranke ub der Kandidaten. Im Anschluss entnehmen wir der Prioritätswarteschlange den Kandidaten mit der höchsten oberen Schranke und evaluieren diesen. Dieser Prozess wird wiederholt, bis die Prioritätswarteschlange leer ist.

Um nicht immer wieder für jeden Kandidaten die Profite einzelner Gegenstände aufsummieren zu müssen, wird eine Präfixsumme verwendet. Dazu wird auf der (nach Profitdichte sortierten) Liste der Gegenstände eine Präfixsumme sowohl für die Profite als auch die Gewichte erstellt. Nehmen wir an, dass wir den Kandidaten $c_i = (a_1, a_2, ..., a_h, \perp, \perp, ..., \perp)$ evaluieren möchten. Für diesen Kandidaten haben wir bereits das bisherige Gewicht $w_c = \sum_{j=1}^{h} a_j \cdot w_j$ sowie den bisherigen Profit $p_c = \sum_{j=1}^{h} a_j \cdot p_j$ ermittelt. Um nun zu ermitteln, bis zu welcher Stelle weitere Gegenstände in den Rucksack gepackt werden können, geht man wie folgt vor:

(i) Ermittle $w_t = C - w_c + q(w_h)$, wobei $q(w_h)$ die Präfixsumme der Gewichte an Position h sei und C die Gesamtkapazität bezeichne

(ii) Bestimme nun die kleinste Position j, an welcher $q(w_{j+1}) > w_t$.

(iii) Als untere Schranke ergibt sich $p_{lb} = p_c + q(p_j) - q(p_h)$

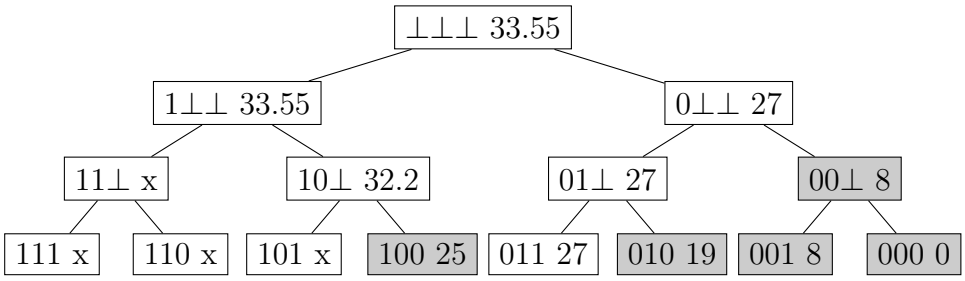


Abbildung 3: Kandidaten einer Instanz mit den Gegenständen $(w_1 = 21, p_1 = 25)$, $(w_2 = 20, p_2 = 19)$ und $(w_3 = 10, p_3 = 8)$ sowie einer Kapazität von 30. Mit einem x markierte Kandidaten werden aus Kapazitätsgründen nicht erstellt. Grau markierte Kandidaten werden aufgrund unterer Schranken nicht erstellt. Die obere Schranke der Kandidaten ist – falls vorhanden – ebenfalls angegeben.

(iv) Für die Ermittlung der oberen Schranke wird der Gegenstand an Stelle $j + 1$ partiell berücksichtigt.

Abschließend muss noch geklärt werden, wie man das bisherige Gewicht w_c sowie den bisherigen Profit p_c für die einzelnen Kandidaten ermittelt. Man betrachtet hierzu denjenigen Kandidaten, welcher die Erstellung der neuen Kandidaten ausgelöst hat. Für den neuen Kandidaten $c_n = (a_1, a_2, ..., a_h, 0, \bot, \bot, ..., \bot)$ sind die Werte identisch, da wir keine neuen Gegenstände hinzunehmen. Der neue Kandidat $c_{n+1} = (a_1, a_2, ..., a_h, 1, \bot, \bot, ..., \bot)$ benutzt lediglich einen zusätzlichen Gegenstand. Somit können Gewicht und Profit des neuen Gegenstands auf die bisherigen Werte addiert werden. Dieses Vorgehen setzt voraus, dass w_c und p_c für alle Kandidaten, welche eine Erstellung neuer Kandidaten bei ihrer Evaluation auslösen, bekannt sind, weswegen diese Werte für jeden Kandidaten abgespeichert werden. Lediglich der initiale Kandidat c_0 wird nicht durch die Evaluation eines anderen Kandidaten erstellt. Jedoch sind dessen Werte $w_c = 0$ und $p_c = 0$ trivial. Insgesamt müssen für einen konkreten Kandidaten nur die Werte w_c, p_c und h gespeichert werden. Der eigentliche Entscheidungsvektor $(a_1, a_2, ..., a_h, \bot, \bot, ..., \bot)$ muss nicht gespeichert werden, da wir nur am maximalen Profit interessiert sind. Die Belegung, welche diesen maximalen Profit ermöglicht, ist für uns nicht von Bedeutung.

Das oben beschriebene Vorgehen soll im Folgenden anhand eines Beispiels veranschaulicht werden. Betrachten wir dafür die Beispielinstanz aus Abb. 3. Zunächst müssen wir die Profitdichte der einzelnen Gegenstände ermitteln. Es ergibt sich (sortiert nach Profitdichte): $d_1 \approx 1.19$, $d_2 = 0.95$ und $d_3 = 0.8$. Der Kandidat an der Wurzel des Baums wird initial erzeugt und sofort evaluiert. Bei der Berechnung der Schranken stellen wir fest, dass sich als untere Schranke ein Profit von 25 ergibt. Dies liegt daran, dass wir zunächst r_1 aufgrund seiner höchsten Profitdichte in den Rucksack packen. Danach passt kein Gegenstand mehr vollständig in den Rucksack hinein. Somit setzen wir s auf 25. Für die Berechnung der oberen Schranke stellen wir fest, dass noch $t = 9/20$ von r_2 in den Rucksack passen. Also ergibt sich als obere Schranke ein Wert von $25 + 9/20 \cdot 19 = 33.55$. Anschließend werden die Kandidaten $c_1 = (0, \bot, \bot)$ und $c_2 = (1, \bot, \bot)$ erzeugt. Für diese berechnen wir ebenfalls die obere und untere Schranke. Danach fügen wir c_1 und c_2 in die Prioritätswarteschlange ein. Als Schlüssel dient hierbei die obere Schranke der Profitdichte. Durch c_1 wird

angegeben, dass wir r_1 nicht in unseren Rucksack packen und c_2 gibt an, dass wir r_1 einpacken.
Damit ist ein Evaluationsschritt abgeschlossen und wir entnehmen der Prioritätswarteschlange das Element mit dem größten Schlüsselwert (im Beispiel c_2). Danach evaluieren wir diesen Kandidaten, d.h. wir nehmen die erste Stelle, an welcher sich ein $\perp$-Symbol befindet, und erstellen zwei neue Kandidaten $c_3 = (1, 1, \perp)$ und $c_4 = (1, 0, \perp)$. Für diese berechnen wir ebenso eine obere sowie eine untere Schranke. Bei c_3 haben wir allerdings das Problem, dass wir nicht sowohl r_1 als auch r_2 mitnehmen können, da die Kapazität des Rucksacks nicht ausreichend ist. Daher wird der Kandidat c_3 verworfen. Im obigen Beispiel sind Kandidaten, welche die Kapazitätsbedingung verletzen, mit einem x markiert. Für c_4 errechnen wir als obere Schranke den Wert $32,2 > s = 27$. Wir fügen also nur c_4 in unsere Prioritätswarteschlange ein. Daraufhin entnehmen wir der Prioritätswarteschlange wieder den Kandidaten mit der höchsten oberen Schranke, evaluieren diesen usw. Falls nun die obere Schranke des neu generierten Kandidaten unter dem Wert der bisher besten Belegung liegt, wird der Kandidat verworfen. Er kann nicht mehr auf eine bessere Lösung hinführen. Im Beispiel sind aufgrund dieser Bedingung nicht erzeugte Kandidaten grau markiert. Ebenso kann ein Kandidat, welcher keine $\perp$-Symbole mehr beinhalt, keine neuen Kandidaten erzeugen. Vielmehr stellt er – sofern er die Kapazitätsbedingung nicht verletzt – eine Befüllung des Rucksacks dar. Wir prüfen also lediglich, ob sein Gesamtprofit höher als der Profit der bisher besten Belegung ist und passen diese gegebenenfalls an. Diese Überprüfung findet nur statt, falls der Kandidat nicht aufgrund der Kapazitätsbedingung verworfen wurde. Kandidaten, welche keine neuen Kandidaten mehr erzeugen, befinden sich stets auf der untersten Ebene des Suchbaums.

4.1 Sequenzieller Algorithmus

Von Interesse ist einerseits der sequenzielle Algorithmus. Er dient als Vergleichsimplementierung und zur Ermittlung des Speedups. Hier wird lediglich stets der Kandidat mit der höchsten oberen Schranke aus der Prioritätswarteschlange (in den Kommentaren des Pseudocodes PQ genannt) entfernt und evaluiert. Der Pseudocode für den sequenziellen Algorithmus ist in Algorithmus 2 dargestellt.

Alle Algorithmen verwenden die oben beschriebene Evaluation. Der Pseudocode für die Evaluation ist durch Algorithmus 1 beschrieben.

4.2 Parallele Algorithmen

Bei den parallelen Algorithmen wird die Evaluation der einzelnen Kandidaten parallelisiert. Das bedeutet, dass in einer Schleifeniteration mehr als nur ein Kandidat evaluiert wird. In Bezug auf die parallelen Algorithmen betrachten wir einerseits einen Algorithmus, welcher neu erzeugte Kandidaten an einen zufälligen Prozessor schickt (Ansatz aus der Literatur; vgl. Karp und Zhang [1993a] sowie Karp und Zhang [1993b]). Des Weiteren wird ein Algorithmus vorgestellt, welcher nur bei hoher Imbalance umverteilt.
Sei im Folgenden $p \in \mathbb{N}$ die Anzahl der Prozessoren. Bei den parallelen Algorithmen kann es aufgrund hoher Imbalance notwendig sein, Kandidaten an einen anderen Prozess zu senden. Dies übernimmt die Methode *distribute*, welche in Algorithmus 3

Algorithm 2: sequenzielle Version

```
Input: Eine Rucksackinstanz I(M, C)
1  sort(M)                                  // sortiere die Gegenstände nach Profitdichte
2  P ← ∅                                    // erstelle leere PQ
3  s ← 0                                    // setze bisher beste Lösung auf 0
4  P.insert(c_0)                            // füge initialen Kandidaten hinzu
5  while |P| > 0 do                         // solange noch Kandidaten vorhanden sind
6      c ← P.max()                          // nehme den Kandidaten mit größter oberer Schranke
7      P.deleteMax()
8      (L, s) ← evaluate(c, s, |M|)         // werte den Kandidat aus
9      P.insert(L)                          // füge neue Kandidaten in die PQ ein
Output: s
```

Algorithm 3: distribute

```
Input: Liste L von Kandidaten
1  E = (E_1, E_2, ...E_p)                   // erstelle p leere Listen
2  numReceiver ← max(⌈log_2|L|⌉, 1)
3  messageSize ← |L|/numReceiver
4  while |L| > 0 do                         // solange noch Kandidaten vorhanden sind
5      i ← rand()                           // ziehe Zufallszahl zwischen 1 und p
6      for a = 0 to messageSize do
7          if |L| = 0 then
8              break
9          E_i.pushFront(L.first())
10         L.popFront()
11 R ← NBX–NonblockingConsensus(E)          // sende wie in Hoefler u. a. [2010])
Output: R
```

kurz vorgestellt wird.
Zudem ist es nun nicht mehr so einfach, die global größten Kandidaten zu ermitteln, da jeder Prozessor eine lokale Prioritätswarteschlange verwaltet. Wir verwenden daher einen Selektor von Hübschle-Schneider und Sanders [2016]. Dieser Selektor findet die global größten Kandidaten. Da es wenig Sinn ergibt, in jeder Iteration global nur einen Kandidaten zu evaluieren, evaluieren wir in jeder Iteration zwischen $l \in \mathbb{N}$ und $u \in \mathbb{N}$ Kandidaten. Wir legen nicht genau fest, wie viele Kandidaten in einer Runde zu evaluieren sind, da dies einerseits für die Korrektheit nicht wichtig ist und andererseits der Selektionsalgorithmus von dieser Freiheit profitiert. Die Methode *select(l, u, P)* bekommt somit von jedem Prozessor die lokale Prioritätswarteschlange P sowie zwei natürliche Zahlen l und u übergeben. Sind für einen bestimmten Prozess einige der global größten Kandidaten in dessen lokaler Prioritätswarteschlange, so werden diese Kandidaten durch die Methode aus der lokalen Prioritätswarteschlange entfernt und ausgegeben. Der parallele Algorithmus, welcher neue Kandidaten an einen zufälligen Prozessor schickt, kann wie in Algorithmus 4 umgesetzt werden.

Abschließend soll der parallele Algorithmus, der nur im Falle einer starken Imbalance eine Umverteilung vornimmt, vorgestellt werden. Dessen Struktur unterscheidet sich von Algorithmus 4 lediglich darin, dass neue Kandidaten nicht an einen

Algorithm 4: parallele Version

```
Input: Eine Rucksackinstanz I(M, C) sowie Parameter l und u
1  P ← ∅                                                    // erstelle leere PQ
2  s ← 0                                                    // setze bisher beste Lösung auf 0
3  while true do
4      Q ← ∅                                                // erstelle leere PQ
5      if allreduce(|P|, Σ_{j=0}^{p} |P@j|) = 0 then       // keine Kandidaten?
6          break
7      else if allreduce(|P|, Σ_{j=0}^{p} |P@j|) < u then  // wenige Kandidaten?
8          Q ← P                                            // wähle alle Kandidaten aus
9      else
10         Q ← select(l, u, P)                              // wähle global größte Kandidaten aus
11     K ← ∅
12     for a = 0 to Q.size() do                             // für alle ausgewählten Kandidaten
13         (L, s) ← evaluate(Q.max(), s, |M|)
14         Q.deleteMax()
15         K.concat(L)
16     K ← distribute(K)                                    // sende neue Kandidaten an zufälligen Prozess
17     P.insert(K)                                          // füge empfangene Kandidaten in lokale PQ ein
18     s ← allreduce(s, max_{i=1}^{p} s@i)                  // passe bisher beste Lösung an
Output: s
```

zufälligen Prozessor versendet, sondern in die lokale Prioritätswarteschlange eingefügt werden. Dafür kommt es jedoch zu einer expliziten Umverteilung, sobald die Imbalance-Schranke, im Folgenden durch den Parameter b gegeben, überschritten wird. Der Pseudocode zu diesem Algorithmus findet sich in Algorithmus 5.

4.3 Instanzgeneratoren

Für diese Arbeit wurden im Wesentlichen zwei verschiedene Instanzgeneratoren verwendet. Einerseits kam ein Generator von Martello und Toth [1990] zum Einsatz, dessen Pseudocode in Algorithmus 6 dargestellt ist.
Neben diesem einfachen Algorithmus wurde zusätzlich ein von Pisinger [1999] entwickelter Algorithmus verwendet. Bei diesen Generatoren wurde unter anderem versucht, es auf Branch-and-Bound beruhenden Verfahren möglichst schwer zu machen. Bei diesem Generator sind die Gewichte w_j über $[1, R]$ mit $R \in \mathbb{R}^+$ gleichverteilt. Die Profite werden als $p_j = w_j + R/10$ gewählt. Der Code für diesen Generator ist unter `http://hjemmesider.diku.dk/~pisinger/codes.html` zu finden. Ein Sortieren nach Profitdichte führt hier im Wesentlichen zu einer Sortierung nach Gewichten. Dies führt dazu, dass benachbarte Gegenstände in der nach Profitdichte sortierten Liste ein sehr ähnliches Gewicht haben. Dadurch wird eine exakte Befüllung (bei welcher die Summe der eingepackten Gegenstände der Kapazität entspricht) erschwert. Man muss nun auch Gegenstände in Betracht ziehen, welche eine vergleichsweise schlechte Profitdichte aufweisen. Dadurch werden deutlich mehr Kandidaten in Betracht gezogen. Genauer gesagt können bei diesen Instanzen die oberen und unteren Schranken nicht mehr so häufig dazu beitragen, dass ein Kandidat verworfen wird.

Algorithm 5: parallele Version mit optimierter Kommunikation

Input: Eine Rucksackinstanz $I(M, C)$ sowie Parameter l, u und b

```
1  P ← ∅                                                    // erstelle leere PQ
2  s ← 0                                                    // setze bisher beste Lösung auf 0
3  while true do
4      Q ← ∅                                                // erstelle leere PQ
5      if allreduce(|P|, Σ_{j=0}^{p} |P@j|) = 0 then       // keine Kandidaten?
6          break
7      else if allreduce(|P|, Σ_{j=0}^{p} |P@j|) < u then  // wenige Kandidaten?
8          Q ← P                                            // wähle alle Kandidaten aus
9      else
10         Q ← select(l, u, P)                              // wähle global größte Kandidaten aus
11     m ← allreduce(|Q|, max_{i=1}^{p} |Q@i|)
12     g ← allreduce(|Q|, Σ_{k=1}^{p} |Q@k|)
13     if m > g · b/p then                                  // hohe Imbalance?
14         if |Q| > g · b/p then                            // lokal zu viele Kandidaten?
15             (R, Q) ← Q.split()                           // teile PQ in zwei gleich große Hälften
16         L ← ∅
17         L.insert(R)
18         L ← distribute(L)
19         Q.insert(L)
20     while |Q| > 0 do
21         (K, s) ← evaluate(Q.max(), s, |M|)
22         Q.deleteMax()
23         P.insert(K)
24     s ← allreduce(s, max_{i=1}^{p} s@i)                  // passe bisher beste Lösung an
```

Output: s

Algorithm 6: Instanzgenerator

Input: Anzahl n der zu erstellenden Gegenstände

```
1  capacity ← 0
2  items ← ∅                                  // Liste, in welche Gegenstände eingefügt werden
3  repeat
4      weight ← 1.0 + z_0                     // sei z_0 eine Zufallszahl
5      profit ← weight + weight · z_0 · 0.1
6      items.pushFront(r(profit, weight))     // erstelle Gegenstand r
7      capacity ← (weight · 0.5) + capacity
8  until items.size() = n
```

Output: *items* mit zugehöriger *capacity*

5 Experimente

5.1 Implementierungsdetails

Im Folgenden soll etwas genauer auf die Umsetzung der Algorithmen eingegangen werden. Die im letzten Abschnitt vorgestellten Algorithmen wurden in der Programmiersprache C++ implementiert. Der Code wurde durch den GNU C++ Compiler g++ 9.2 kompiliert. Für die Kommunikation zwischen den einzelnen Knoten wurde MPI 3.1 verwendet. Zudem mussten wir uns auf eine konkrete Prioritätswarteschlange festlegen. Die Wahl fiel hierbei auf einen B+-Baum. Die Implementierung des B+-Baums wurde durch Bingmann [2018] realisiert. Die Listen wurden mit Hilfe der Standardimplementierung *std::vector* implementiert. Für die Realisierung des Zufalls fiel die Wahl auf einen Mersenne-Twister, konkret auf *std::mt19937*.

Die Wahl der Parameter für die untere beziehungsweise obere Schranke l und u unterscheidet sich stark zwischen den beiden Generatoren. So führt beim Generator von Martello eine Wahl von $l = 1250 \cdot p$ und $u = 1875 \cdot p$ zu bestmöglichen Speedups. Beim Generator von Pisinger wurde der Parameter l auf den Wert $9 \cdot p$, der Parameter u auf $12 \cdot p$ gesetzt. Es sei daran erinnert, dass p die Anzahl der Prozessoren darstellt. Der Parameter b – welcher der Imbalance-Schranke entspricht – wurde für den Generator von Pisinger auf den Wert 1.3 gesetzt. Beim Generator von Martello stellte sich heraus, dass ein Wert von $b = 1.075$ deutlich besser geeignet ist. Auswirkungen durch Änderungen der Parameterwerte auf die Laufzeit werden in den jeweiligen Abschnitten beschrieben.

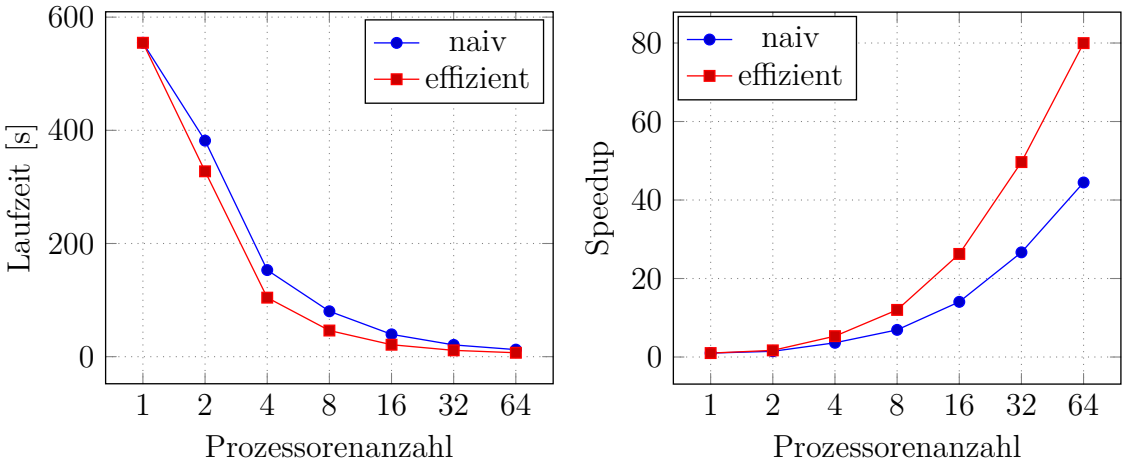
In unseren Experimenten wurden die Laufzeiten von Algorithmus 4 (im Folgenden als ”naiv” bezeichnet) und Algorithmus 5 (im Folgenden als ”effizient” bezeichnet) erfasst. Die gemessene Laufzeit für verschiedene Instanzen beziehungsweise Prozessoren wird auf den nächsten Seiten genauer vorgestellt.

5.2 Shared Memory

Die Messungen wurden auf einem Rechner mit einem AMD EPYC Rome 7702P Prozessor durchgeführt. Dieser besitzt 1024GB DDR4 ECC RAM und einen L3-Cache der Größe 256MB. Als Betriebssystem ist Ubuntu 19.10 installiert.

5.2.1 Ergebnisse des Generators von Martello

Mit dem Generator von Martello lassen sich sehr gute Beschleunigungen durch die Verwendung von mehr Prozessoren erreichen. Details zu Laufzeit und Speedup sind in Abb. 4 dargestellt. Gezeigt wird der Durchschnitt der Laufzeiten (beziehungsweise Speedups) von insgesamt 50 Instanzen mit jeweils 100 Gegenständen. Man erkennt, dass das ständige Umverteilen des naiven Algorithmus für eine deutliche Erhöhung der Laufzeit sorgt. Wirft man einen Blick auf die Speedups, so sind sogar superlineare Speedups zu beobachten. Dies liegt daran, dass die lokalen Prioritätswarteschlangen kleiner werden, falls mehr Prozessoren verwendet werden. Dadurch benötigt das Einfügen von Kandidaten weniger Zeit. Zudem muss seltener geprüft werden, ob noch Kandidaten zu evaluieren sind, da die parallelen Algorithmen nach einer Überprüfung stets mehr als einen Kandidaten evaluieren.



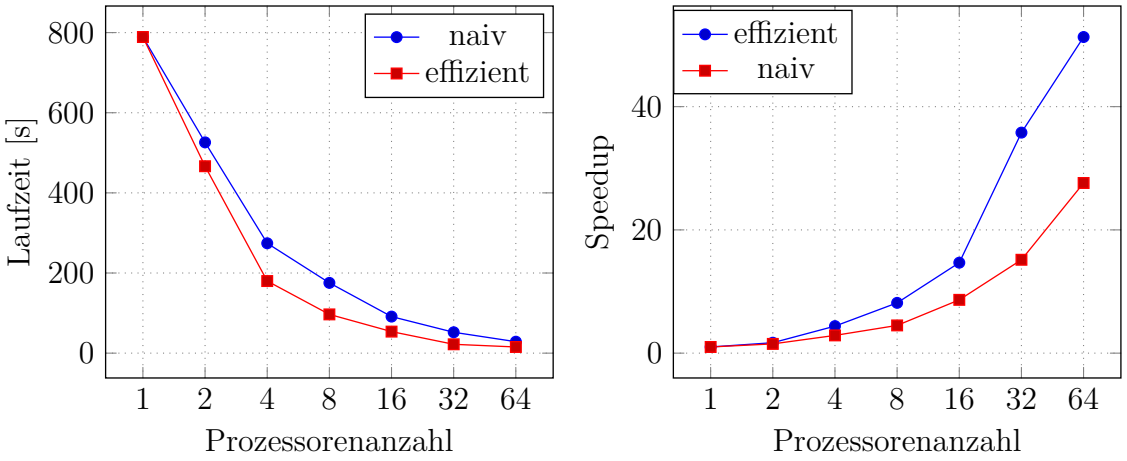
(a) Laufzeiten der Instanzen von Martello (b) Speedup der Instanzen von Martello

Abbildung 4: Shared-Memory-Ergebnisse des Generators von Martello

Änderungen der Parameter haben einen entscheidenden Einfluss auf die Laufzeit. So führt beispielsweise eine Erhöhung der Parameter l und u auch zu einer deutlichen Erhöhung der Laufzeit. Betrachtet man die Anzahl der evaluierten Kandidaten, erkennt man, wodurch dieser Effekt zustande kommt. Es werden deutlich mehr Kandidaten evaluiert, da die unteren Schranken der einzelnen Prozessoren seltener ausgetauscht werden. So kann es passieren, dass Kandidaten von einem bestimmten Prozessor noch in dessen lokale Prioritätswarteschlange eingefügt werden, obwohl ein anderer Prozessor diese bereits verwerfen würde. Die zusätzliche Evaluation ist somit für die Bestimmung einer Lösung nicht notwendig, benötigt aber zusätzliche Zeit. Man stellt daher auch fest, dass sich – sofern man die Parameter wie oben beschrieben wählt – die Anzahl der evaluierten Kandidaten zwischen dem sequenziellen und den parallelen Algorithmen kaum unterscheidet. Auch eine Verringerung der Parameter l und u sorgt für eine Verlangsamung. Dieser Effekt lässt sich dadurch erklären, dass mehr Iterationen benötigt werden, um eine Lösung zu erhalten. Dadurch steigt wiederum der Kommunikationsaufwand, was sich negativ auf die Laufzeit auswirkt. Auch Änderungen der Imbalance-Schranke sorgen für eine Erhöhung der Laufzeit. Erhöht man die Imbalance-Schranke, so führt dies zu einer höheren Imbalance. Einige Prozessoren haben deutlich weniger Kandidaten als andere zu evaluieren. Dies führt dazu, dass einige Prozessoren auf andere Prozessoren warten müssen. Verringert man im effizienten Algorithmus die Imbalance-Schranke, so nähert man sich der Laufzeit des naiven Algorithmus immer mehr an. Dies kann durch die zusätzliche Kommunikation begründet werden. Es müssen nun deutlich mehr Kandidaten umverteilt werden. Es ist sogar möglich, dass die Laufzeit schlechter wird als beim naiven Algorithmus. Das lässt sich dadurch begründen, dass Kandidaten mehrmals den Prozessor wechseln, bevor sie evaluiert werden.

5.2.2 Ergebnisse für Instanzen des Generators von Pisinger

Beim Generator von Pisinger müssen für die gleiche Anzahl an Gegenständen gegenüber dem Generator von Martello im Schnitt etwas weniger Kandidaten betrachtet werden, um die optimale Lösung zu ermitteln. Es lassen sich aber auch mit diesem Generator hohe Beschleunigungen durch die Verwendung von mehr Prozessoren erzielen. Betrachtet man die Anzahl der evaluierten Kandidaten, so merkt man



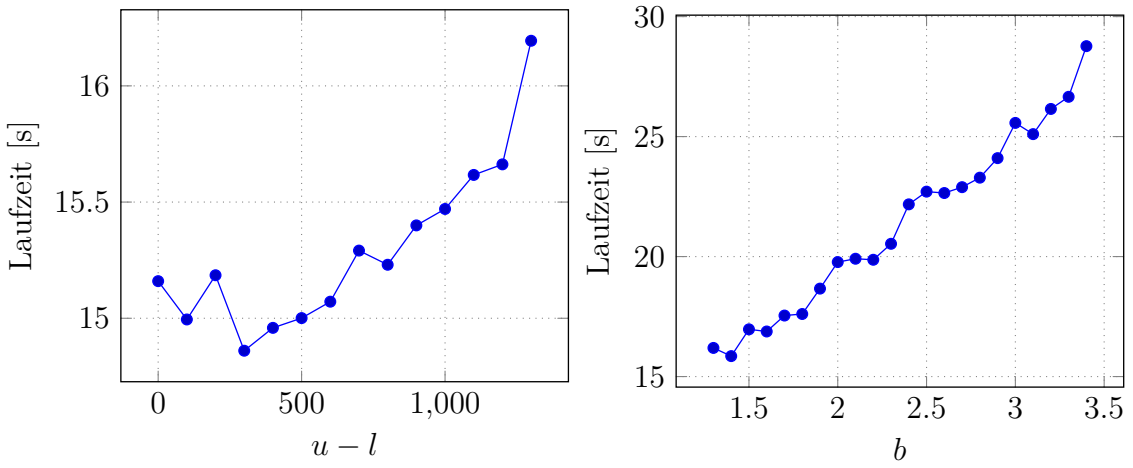
(a) Laufzeit der Instanzen von Pisinger (b) Speedup der Instanzen von Pisinger

Abbildung 5: Shared-Memory-Ergebnisse des Generators von Pisinger

auch bei diesem Generator schnell, dass sich diese kaum zwischen dem sequenziellen und parallelen Algorithmus unterscheidet. Konkrete Laufzeitergebnisse sind in Abb. 5 visualisiert. Dargestellt ist der Durchschnitt der Messergebnisse von 50 Instanzen der Größe 350. Somit ist die Instanz größer als beim Generator von Martello, kann jedoch fast genauso schnell gelöst werden. Dies ist darauf zurückzuführen, dass beim Branch-and-Bound-Verfahren mehr Kandidaten aufgrund oberer beziehungsweise unterer Schranken verworfen werden können. Das sorgt wiederum dafür, dass es sich lohnt, pro Iteration deutlich weniger Kandidaten zu evaluieren. Durch die zusätzliche Kommunikation werden die unteren Schranken häufiger zwischen den einzelnen Prozessoren ausgetauscht. Die Tatsache, dass pro Runde weniger Kandidaten evaluiert werden, erklärt auch die höhere Imbalance-Schranke gegenüber den Instanzen von Martello. Es werden in jeder Runde weniger neue Kandidaten evaluiert, sodass eine geringe Imbalance nicht so schlimm ist und beispielsweise in der Folgeiteration ausgeglichen werden kann. Man stellt allerdings auch fest, dass die häufigere Kommunikation zu einer deutlichen Verschlechterung des Speedups führt. Änderungen der Parameter haben bei diesen Instanzen ebenfalls einen großen Einfluss auf die Gesamtlaufzeit, wobei hier eine Änderung der Imbalance-Schranke die größte Auswirkung auf die Laufzeit hat. Konkrete Zahlen für die Auswirkungen auf die Laufzeit sind in Abb. 6 dargestellt. Für $b < 1.3$ sind keine Werte angegeben, da die Berechnung das gesetzte Zeitlimit von sechs Minuten pro Instanz überschreitet. Verändert man l und u gleichzeitig um den gleichen Wert, so entsprechen die Auswirkungen denen des Generators von Martello. Änderungen des Bereichs, welcher sich durch die Differenz der Schranken l und u definiert, wirken sich lediglich moderat auf die Laufzeiten aus.

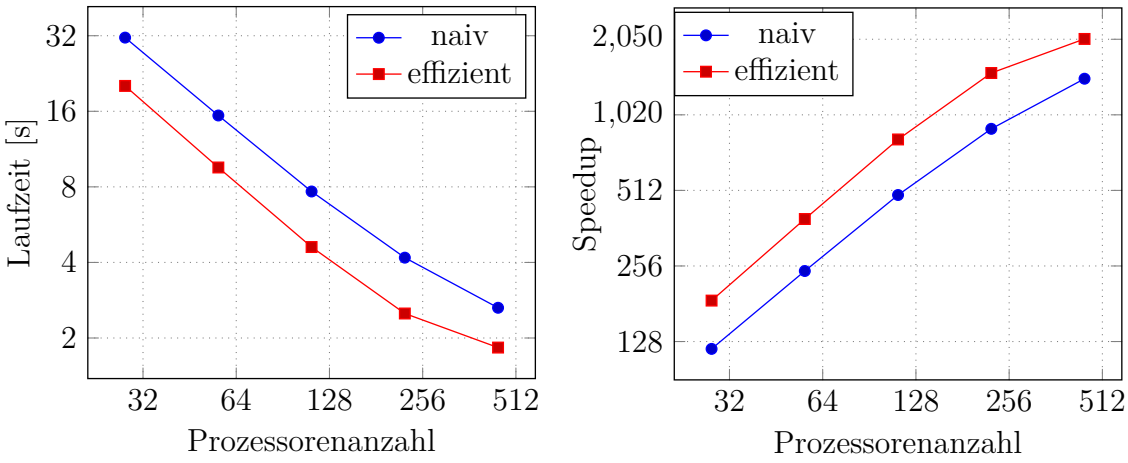
5.3 Verteilte Ergebnisse

Die verteilten Experimente wurden auf einem Cluster – dem bwUniCluster – durchgeführt. Ein einzelner Knoten besitzt auf diesem Cluster 28 Kerne. Jede einzelne Maschine ist mit folgenden Komponenten ausgestattet: Als Prozessor werden auf jedem Knoten 2 Intel Xeon E5-2660 v4 eingesetzt. Zudem besitzen die Maschinen jeweils 128 GB RAM und als Betriebssystem kommt ein Red Hat Enterprise Linux 7.7 (Maipo) zum Einsatz.



(a) Änderungen des Bereichs $u - l$ (b) Änderungen Imbalance-Schranke b

Abbildung 6: Laufzeitänderungen durch Modifikation der Parameter (shared memory, Pisingers Generator)

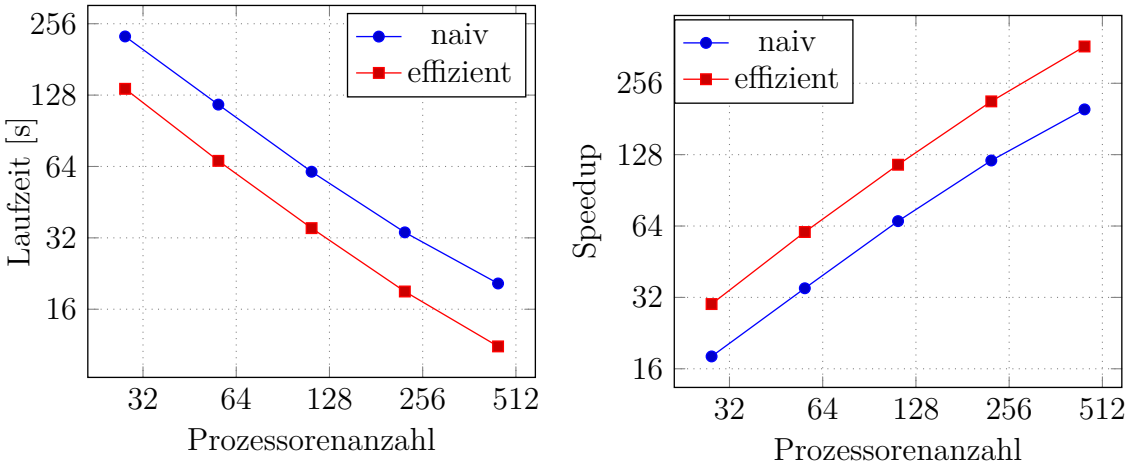


(a) Laufzeit der Instanzen von Martello (b) Speedup der Instanzen von Martello

Abbildung 7: Verteilte Ergebnisse des Generators von Martello

5.3.1 Laufzeiten für Instanzen basierend auf dem Generator von Martello

Auf dem Cluster erhalten wir im Vergleich zum gemeinsamen Speicher ähnliche Ergebnisse. Die Laufzeiten sind in Abb. 7 dargestellt. Die Wahl der Instanzen sowie der Parameter entspricht der Wahl in Abschnitt 5.2.2. Die Auswirkungen auf die Laufzeit durch Änderung der Parameter ändern sich gegenüber dem verteilten Speicher ebenfalls nicht. Es taucht lediglich ein interessantes Phänomen auf, welches (aufgrund der maximalen Anzahl Prozessoren) nur auf verteiltem Speicher sichtbar wird: Für 448 Prozessoren stellt man fest, dass der Speedup für den effizienten Algorithmus nicht mehr so stark ansteigt wie für weniger Prozessoren. Das kann dadurch erklärt werden, dass es ab hier zu einer Erhöhung der Anzahl evaluierter Kandidaten kommt, da mit zunehmender Prozessoranzahl auch die Anzahl der pro Iteration evaluierten Kandidaten zunimmt. Verringert man die Größe der Instanz, so taucht dieses Phänomen auch auf verteiltem Speicher auf.



(a) Laufzeit der Instanzen von Pisinger

(b) Speedup der Instanzen von Pisinger

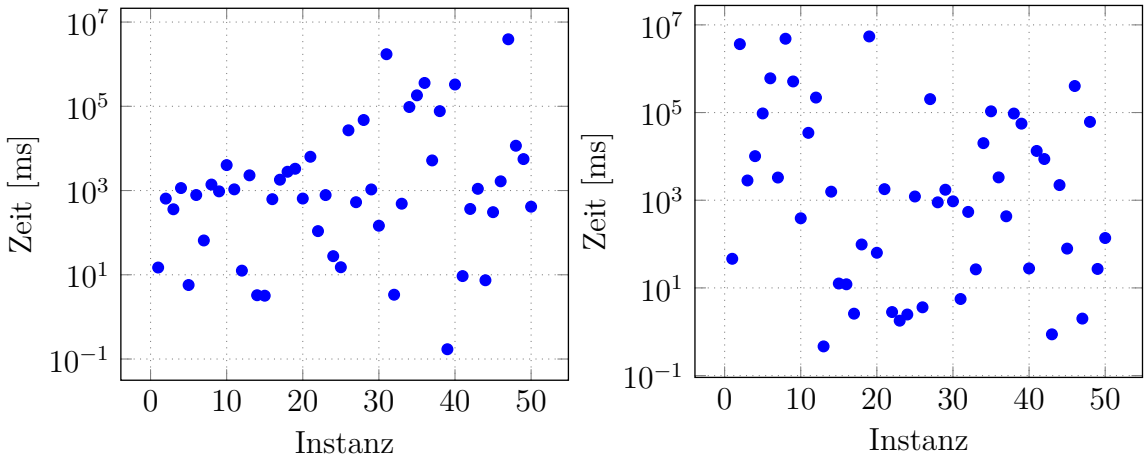
Abbildung 8: Verteilte Ergebnisse des Generators von Pisinger

5.3.2 Laufzeiten für Instanzen basierend auf dem Generator von Pisinger

Für die Instanzen des Generators von Pisinger ist auch auf dem Cluster ein deutlicher Speedup zu verzeichnen. Wir stellen auch hier fest, dass die Anzahl der insgesamt evaluierten Kandidaten von der Anzahl der Prozessoren weitgehend unabhängig ist. Bemerkenswert ist, dass wir einem perfekten Speedup sehr nahe kommen (vgl. Abb. 8). Die Auswirkungen auf die Laufzeit durch eine Modifikation der Parameter sind auch hier den Auswirkungen auf verteiltem Speicher sehr ähnlich. Auch bei diesen Instanzen wurde zunächst festgestellt, dass bei Verwendung sehr vieler Prozessoren der Speedup nicht mehr so gut ist. Dies lag auch hier daran, dass mit steigender Anzahl Prozessoren zu viele Kandidaten evaluiert wurden. Daher wurde die Anzahl der Gegenstände für die Messungen auf verteiltem Speicher von 350 auf 400 erhöht. Diese Instanzgröße ist groß genug, um auch für viele Prozessoren hohe Speedups zu ermöglichen.

5.4 Zusammensetzung der Laufzeiten

Ein spezieller Fokus soll auf die Lastbalancierung gelegt werden, da diese Kern der Arbeit ist. Man erkennt, dass derjenige Algorithmus, welcher neu erstellte Kandidaten an einen zufälligen Prozess schickt, sehr viel Zeit für das Senden beziehungsweise Empfangen der Kandidaten benötigt. Dies ist beim optimierten Algorithmus nicht der Fall und es kommt zu einem merklichen Speedup. Wir betrachten hier nur die Aufteilung der Laufzeit für den Algorithmus von Pisinger. Bei diesem zeigt sich, dass das Versenden der Kandidaten an einen zufälligen Prozessor für Algorithmus 5 etwa 5% der Gesamtlaufzeit ausmacht. In Algorithmus 4 hingegen macht das Versenden etwa 35% der Gesamtlaufzeit aus. Es sei an dieser Stelle angemerkt, dass Algorithmus 4 nicht ganz dem Ansatz, neu entstandene Kandidaten an einen zufälligen Prozessor zu senden, entspricht. Wir wählen nicht für jeden neu erzeugten Kandidaten zufällig einen Prozessor aus, sondern fassen die neuen Kandidaten zu logarithmisch vielen Gruppen zusammen und wählen dann für jede Gruppe einen zufälligen Zielprozessor aus. Wählt man für jeden Kandidaten einen zufälligen Prozessor aus, so ist nochmals eine Erhöhung der Laufzeit zu messen. Zu beachten ist



(a) Laufzeitaufteilung bei Pisinger

(b) Laufzeitaufteilung bei Martello

Abbildung 9: Laufzeitaufteilung der Messreihen

auch, dass die Lastbalancierung deutlich schlechter wird, wenn man nur bei hoher Imbalance umverteilt. Da aber die Anzahl der gesendeten Kandidaten beim naiven Algorithmus etwa um den Faktor 100 bis 1000 höher ist, ist die Laufzeit des effizienten Algorithmus trotz höherer Imbalance geringer.

Für die beiden Instanzgeneratoren ist ein weiteres interessantes Phänomen zu beobachten. Wir stellen fest, dass die Dauer, die Lösung einer bestimmten Instanz zu ermitteln, stark von der jeweiligen Instanz abhängig ist. So haben einige wenige Instanzen einen großen Anteil an der Gesamtlaufzeit, wohingegen andere Instanzen innerhalb weniger Millisekunden gelöst werden können. Die Aufteilung der Laufzeit ist in Abb. 9 visualisiert.

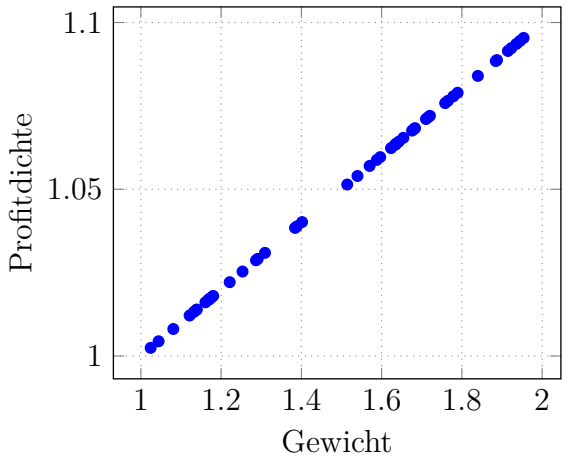
5.5 Abhängigkeiten der Laufzeiten vom Instanzgenerator

Wir haben uns in dieser Arbeit bewusst dazu entschieden, Generatoren zu verwenden, die schwer zu lösende Instanzen generieren. Jedoch gibt es neben diesen Generatoren auch solche, welche Instanzen generieren, die deutlich einfacher zu lösen sind. Ändert man beispielsweise Algorithmus 6 dahingehend ab, dass Zeile 5 durch den Code

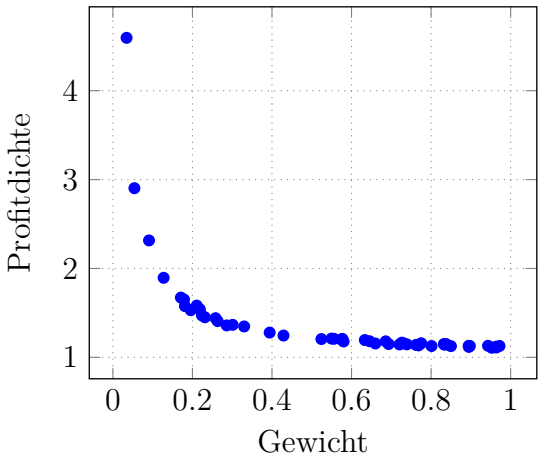
$$items.add(r(weight + 0.1 + 0.025 \cdot z_1, weight)$$

ersetzt wird (wobei *weight* durch $0.01 + z_2$ definiert ist und z_1 sowie z_2 zwei beliebig gewählte Zufallszahlen darstellen), so ändern sich die Laufzeiten gravierend. Es sind plötzlich auch Instanzen mit über 10^6 Gegenständen in wenigen Sekunden lösbar. Wir stellen weiter fest, dass es bei Verwendung von mehreren Prozessoren zu keinen beziehungsweise nur sehr geringen Speedups kommt. Vergleicht man diese Resultate mit den beiden obigen Instanzgeneratoren, so mag es verwundern, dass sich das Verhalten in Bezug auf Laufzeit und Speedup derart unterscheidet. Betrachtet man jedoch andere Publikationen, so stellt man fest, dass dies ein übliches Phänomen bei einer Parallelisierung von Branch-and-Bound-Algorithmen ist (vgl. Crainic u. a. [2006]). In den 1980er Jahren gab es sogar eine große Menge an Publikationen, welche sich mit den Auswirkungen jener Parallelisierung auf die Laufzeit beschäftigen,

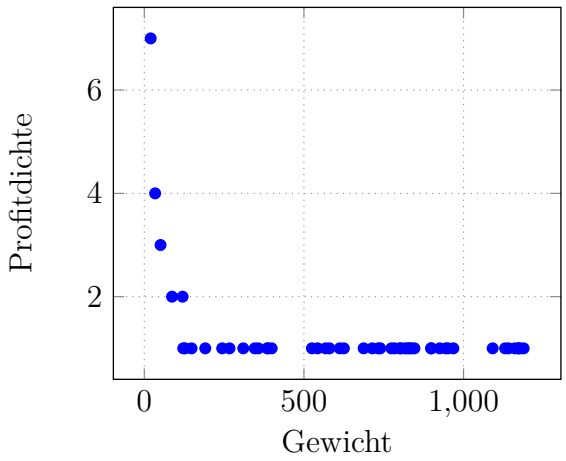
zum Beispiel Li und Wah [1986]. Bei der Parallelisierung von Branch-and-Bound-Algorithmen sind sowohl Verlangsamungen als auch superlineare Speedups feststellbar. Dies ist darauf zurückzuführen, dass durch das Parallelisieren die Reihenfolge, in welcher die Kandidaten evaluiert werden, verändert wird. Daher werden mitunter nicht die gleichen Kandidaten betrachtet. Denn durch die Veränderungen der Evaluationsreihenfolge verbessern sich gegebenenfalls die unteren beziehungsweise oberen Schranken schneller. Dadurch werden Kandidaten, die der sequenzielle Algorithmus in Betracht zieht, bei der parallelen Ausführung verworfen. Jedoch kann es auch passieren, dass die Lösung sehr leicht gefunden beziehungsweise der Entscheidungsbaum sehr stark verkleinert werden kann. Eine Parallelisierung führt in diesem Fall dazu, dass zu viele Kandidaten evaluiert werden. Die genauen Speedups sind daher sehr stark von den jeweiligen Instanzgeneratoren abhängig. Die Schwere der Instanzen eines Generators kann man beispielsweise durch Betrachtung des Verhältnisses zwischen Gewicht und Profitdichte ermitteln. Betrachten wir dazu Abb. 10. Beim Generator von Martello stellen wir fest, dass eine Sortierung nach Profitdichte im Wesentlichen einer Sortierung nach Gewichten entspricht. Nach Kellerer u. a. [2004] ist dies ein Indiz für schwere Instanzen. Anders verhält es sich beim Generator, welcher durch Modifikation von Algorithmus 6 erstellt wurde. Hier liegt es auf der Hand, den Gegenstand oben links sofort einzupacken. Auf die Gegenstände in der unteren rechten Ecke kann man hingegen mit großer Wahrscheinlichkeit verzichten. Der Generator von Pisinger erhält seine Schwere dadurch, dass viele Elemente mit gleicher Profitdichte existieren. Dadurch ist eine Sortierung der Gegenstände nach Profitdichte nicht wirklich hilfreich.



(a) Generator von Martello



(b) Algorithmus 6 (modifiziert)



(c) Generator von Pisinger

Abbildung 10: Vergleich der Instanzgeneratoren

6 Schlussfolgerungen

6.1 Zusammenfassung

Wir haben basierend auf einer Implementierung eines B+-Baums von Bingmann [2018] sowie eines verteilten Selektionsalgorithmus von Hübschle-Schneider und Sanders [2019] eine verteilte Prioritätswarteschlange realisiert. Diese wurde für einen Löser des parallelen Branch-and-Bound-Rucksackproblems verwendet. Ein besonderer Fokus lag auf der Lastbalancierung. Es wurde gezeigt, dass die Lastbalace eine entscheidende Rolle für die Laufzeit spielt. Eine Umverteilung ist nur bei hoher Imbalance notwendig, weswegen das Senden neuer Elemente an einen zufälligen Prozessor ineffizient ist. Wir haben damit gezeigt, dass ein geschicktes und kontrolliertes Umverteilen von Elementen signifikante Verbesserungen in der Laufzeit bewirken kann.

6.2 Mögliche Erweiterungen

Ein offenes Problem ist das adaptive Gestalten der Parameter. Dies betrifft einerseits die Wahl von l und u, also der unteren beziehungsweise oberen Schranke für den Selektor. Andererseits kann eine geschickte Wahl der Imbalance-Schranke eine deutliche Verbesserung der Laufzeit bewirken. In dieser Arbeit wurde die Anzahl der pro Iteration evaluierter Kandidaten meist basierend auf der Prozessorenanzahl gewählt. Dies ist allerdings keineswegs immer optimal. Zudem unterscheidet sich die optimale Wahl der Schranken von Instanz zu Instanz. Hier könnte man der Frage auf den Grund gehen, wie die Schranken für jede Instanz adaptiv gewählt werden können, um noch bessere Speedups zu erreichen.

Literatur

[bwUniCluster] *Steinbuch Centre for Computing (SCC) - bwUniCluster*. https://www.scc.kit.edu/dienste/bwUniCluster.php. – Accessed: 2020-03-14

[Arora und Barak 2009] Arora, Sanjeev ; Barak, Boaz: *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009. – URL http://www.cambridge.org/catalogue/catalogue.asp?isbn=9780521424264. – ISBN 978-0-521-42426-4

[Balas und Zemel 1980] Balas, Egon ; Zemel, Eitan: An Algorithm for Large Zero-One Knapsack Problems. In: *Operations Research* 28 (1980), Nr. 5, S. 1130–1154. – URL https://doi.org/10.1287/opre.28.5.1130

[Bingmann 2018] Bingmann, Timo: *TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers*. 2018. – https://panthema.net/tlx, retrieved Oct. 7, 2020

[Comer 1979] Comer, Douglas: The Ubiquitous B-Tree. In: *ACM Comput. Surv.* 11 (1979), Nr. 2, S. 121–137. – URL https://doi.org/10.1145/356770.356776

[Crainic u. a. 2006] Crainic, Teodor G. ; Cun, Bertrand L. ; Roucairol, Catherine: *Parallel Branch-and-Bound Algorithms*. Kap. 1, S. 1–28. In: *Parallel Combinatorial Optimization*, John Wiley & Sons, Ltd, 2006. – URL https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470053928.ch1. – ISBN 9780470053928

[Hübschle-Schneider und Sanders 2016] Hübschle-Schneider, Lorenz ; Sanders, Peter: Communication Efficient Algorithms for Top-k Selection Problems. In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Chicago, IL, USA, 23–27 May 2016*, IEEE, Piscataway (NJ), 2016, S. 659–668. – ISBN 978-1-5090-2140-6

[Hoefler u. a. 2010] Hoefler, Torsten ; Siebert, Christian ; Lumsdaine, Andrew: Scalable communication protocols for dynamic sparse data exchange. In: Govindarajan, R. (Hrsg.) ; Padua, David A. (Hrsg.) ; Hall, Mary W. (Hrsg.): *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, ACM, 2010, S. 159–168. – URL https://doi.org/10.1145/1693453.1693476

[Horowitz und Sahni 1974] Horowitz, Ellis ; Sahni, Sartaj: Computing Partitions with Applications to the Knapsack Problem. In: *J. ACM* 21 (1974), Nr. 2, S. 277–292. – URL https://doi.org/10.1145/321812.321823

[Hübschle-Schneider und Sanders 2019] Hübschle-Schneider, Lorenz ; Sanders, Peter: Communication-Efficient (Weighted) Reservoir Sampling. In: *CoRR* abs/1910.11069 (2019). – URL http://arxiv.org/abs/1910.11069

[Karp und Zhang 1993a] Karp, Richard M. ; Zhang, Yanjun: Randomized Parallel Algorithms for Backtrack Search and Branch-and-Bound Computation. In: *J. ACM* 40 (1993), Nr. 3, S. 765–789. – URL https://doi.org/10.1145/174130.174145

[Karp und Zhang 1993b] Karp, Richard M. ; Zhang, Yanjun: Randomized Parallel Algorithms for Backtrack Search and Branch-and-Bound Computation. In: *J. ACM* 40 (1993), Nr. 3, S. 765–789. – URL https://doi.org/10.1145/174130.174145

[Kellerer u. a. 2004] Kellerer, Hans ; Pferschy, Ulrich ; Pisinger, Da-

vid: *Knapsack problems*. Springer, 2004. – URL https://doi.org/10.1007/978-3-540-24777-7. – ISBN 978-3-540-40286-2

[Khuri u. a. 1994] Khuri, Sami ; Bäck, Thomas ; Heitkötter, Jörg: The zero/one multiple knapsack problem and genetic algorithms. In: Berghel, Hal (Hrsg.) ; Hlengl, Terry (Hrsg.) ; Urban, Joseph E. (Hrsg.): *Proceedings of the 1994 ACM Symposium on Applied Computing, SAC'94, Phoenix, AZ, USA, March 6-8, 1994*, ACM, 1994, S. 188–193. – URL https://doi.org/10.1145/326619.326694

[Korte und Vygen 2012] Korte, Bernhard ; Vygen, Jens: *Combinatorial Optimization: Theory and Algorithms*. 5th. Springer Publishing Company, Incorporated, 2012. – ISBN 3642244874

[Li und Wah 1986] Li, G. ; Wah, B. W.: Coping with Anomalies in Parallel Branch-and-Bound Algorithms. In: *IEEE Transactions on Computers* C-35 (1986), June, Nr. 6, S. 568–573. – ISSN 2326-3814

[Martello und Toth 1990] Martello, Silvano ; Toth, Paolo: *Knapsack Problems: Algorithms and Computer Implementations*. USA, 1990. – ISBN 0471924202

[Pisinger 1995] Pisinger, David: *Algorithms for Knapsack Problems*. 1995

[Pisinger 1999] Pisinger, David: Core Problems in Knapsack Algorithms. In: *Operations Research* 47 (1999), Nr. 4, S. 570–575. – URL https://doi.org/10.1287/opre.47.4.570

[Pisinger 2002] Pisinger, David: An expanding-core algorithm for the exact 0–1 knapsack problem. In: *European Journal of Operational Research* 87 (2002), 12, S. 175–187

[Sanders 1997] Sanders, Peter: *Lastverteilungsalgorithmen für parallele Tiefensuche*, Dissertation, 1997

[Sanders u. a. 2019] Sanders, Peter ; Mehlhorn, Kurz ; Dietzfelbinger, Martin ; Dementiev, Roman: *Sequential and Parallel Algorithms and Data Structures - The Basic Toolbox*. 1. ed. Springer International Publishing, Berlin, 2019. – ISBN 978-3-030-25208-3

[Scheithauer 2018] Scheithauer, Guntram: *Introduction to Cutting and Packing Optimization*. 1. ed. Springer International Publishing, 2018. – ISBN 978-3-319-64402-8