Prozess-Scheduling in einem eindimensionalen Prozessorarray

Studienarbeit

Jochen Seidel

Betreuer: Dipl.-Inform. Dipl.-Math. Jochen Speck
Verantwortlicher Betreuer: Prof. Dr. rer. nat. Peter Sanders

Stand: 10. Juni 2010

Zusammenfassung

In dieser Arbeit wird das Zuteilungsproblem für Mehrprozessorjobs mit einmalig wählbarer, konstant beschränkter Prozessorzahl untersucht. Zunächst wird der nicht-preemptive Fall betrachtet und ein ILP zu dessen Lösung angegeben. Weiter wird ein existierender pseudopolynomieller Algorithmus zur optimalen Lösung des Problems mit Preemption verwendet, um ein voll-polynomielles Approximationsschema für das ($\mathbf{Pm|spdp\text{-}any,pmtn|C_{max}}$)-Problem zu entwickeln. Abschließend wird kurz auf die Erzeugung von Testeingaben und Praxisrelevanz im Vergleich mit Greedy-Heuristiken eingegangen.

Inhaltsverzeichnis

Erklärung

Hiermit erkläre ich, die vorliegende Studienarbeit selbständig verfasst und keine anderen außer den angegebenen Quellen verwendet zu haben.

Karlsruhe, den 10. Juni 2010

(Jochen Seidel)

1 Einleitung

Mehrkernrechner mit gemeinsamem Speicher gewinnen zunehmend an Bedeutung. In heutigen Haushaltsrechnern sind Vierkernprozessoren keine Seltenheit mehr, und auch moderne Grafikkarten arbeiten nach diesem Prinzip. Oft können benachbarte Prozessoren durch geteilten Cache oder spezielle Direktverbindungen schnell miteinander kommunizieren. Daher wird in dieser Studienarbeit das Zuteilungsproblem in einem eindimensionalen Prozessorarray untersucht. Ein Job soll dabei stets auf „benachbarten“ Prozessoren zugeteilt werden, um eventuelle Kommunikationskosten zu minimieren. Wie zu sehen sein wird, ist diese Einschränkung an eine Zuteilung in der Regel „von selbst“ erfüllt.

In dieser Arbeit wird im nächsten Unterabschnitt 1.1 zunächst die Problemstellung formaler gefasst und Bezeichnungen eingeführt. Darauf folgend werden in Unterabschnitt 1.2 dann einige Ergebnisse vorgestellt, die für die folgenden Untersuchungen von Interesse sind.
Die nächsten beiden Abschnitte gehen dann jeweils auf die nicht-preemtive (Abschnitt 2) sowie die preemptive Variante (Abschnitt 3) des Problems ein. Dabei wird jeweils zuerst eine Variante des Problems betrachtet, bei der die Anzahl der Prozessoren, die ein Job benötigt, festgelegt ist, und danach untersucht, wie sich das Problem verhält, wenn dies nicht der Fall ist. Für den preemptiven Fall wird dabei ein pseudopolynomieller optimaler Algorithmus, der von Du und Leung in (DL89) vorgestellt wurde, ausführlich erklärt. Dieser wird dann in Unterabschnitt 3.3 verwendet, um ein voll-polyonmielles Approximationsschema anzugeben.
Zum Vergleich der Qualität des Approximationsschemas mit heuristisch ermittelten Lösungen werden in Unterabschnitt 3.4 zwei einfache Greedy-Heuristiken vorgestellt.
Der letzte Abschnitt 4 befasst sich schließlich mit der Generierung von Probleminstanzen sowie der experimentellen Auswertung bzw. der Praxistauglichkeit der vorgestellten Algorithmen.

1.1 Problemstellung

Es werden verschiedene Varianten des Problems betrachtet. Ein Freiheitsgrad stellt die Art des Jobs dar:

- mit festgelegter Prozessorzahl (*fixed-size Jobs*).

- mit einmalig wählbarer Prozessorzahl (*moldable Jobs*).
- mit mehrmals wählbarer Prozessorzahl (*malleable Jobs*).

Bei *moldable Jobs* und *malleable Jobs* ist die Laufzeit abhängig von der zugeteilten Prozessorzahl. Außerdem kann zwischen preemptiver und nichtpreemptiver Zuteilung gewählt werden. *Malleable Jobs* werden in dieser Arbeit jedoch nicht betrachtet.

Zur Klassifikation der verschiedenen Problemvarianten bietet sich die von (GLLK79) vorgeschlagene Drei-Feld Schreibweise mit Erweiterungen aus (Dro96) an. Hier werden nur die in dieser Arbeit verwendeten Abkürzungen beschrieben.

1.1 Definition (Problemklassifikation $(\alpha|\beta|\gamma)$)**:**
In der Drei-Feld Schreibweise gibt das erste Feld α die ***Maschinenvariante*** *an. Hier wird stets $\alpha = \mathbf{Pm}$ angenommen, d.h. es stehen m identische parallele Maschinen zur Verfügung. Die Prozessoranzahl m wird in den Betrachtungen als konstant, das heißt nicht zur Eingabe gehörend, angenommen.*
Um die ***Jobcharakteristika*** *β zu beschreiben, werden folgende Bezeichnungen verwendet:*

$\mathbf{size_j}$ *Ein Job j muss von einer bestimmten Anzahl $\mathbf{size_j}$ von Prozessoren bearbeitet werden. Die Laufzeit des Jobs j ist also nicht abhängig von der Prozessoranzahl.*

spdp-any *Für jeden Job ist die Prozessoranzahl wählbar, die Laufzeit eines Jobs ist funktional von der gewählten Anzahl von Prozessoren abhängig. Über weitere Eigenschaften des Zusammenhangs werden jedoch keine Annahmen getroffen.*

pmtn *Die Jobs dürfen beliebig unterbrochen werden, die Anzahl der für einen Job verwendeten Prozessoren bleibt jedoch auch über Preemptionsschritte hinweg gleich.*

Im letzten Feld findet man das ***Optimierungskriterium****. $\gamma = \mathbf{C_{max}}$ bedeutet, es wird nach einer Zuteilung mit kürzester Gesamtlaufzeit gesucht.*

In einer Zuteilung darf zu jedem Zeitpunkt auf jedem Prozessor höchstens ein Job ausgeführt werden. Folgende Bezeichnungen werden, wenn nicht anders angegeben und aus dem Zusammenhang klar, im Folgenden verwendet:

1.2 Definition (Bezeichnungen in $(\mathbf{Pm}|\circ|\mathbf{C_{max}})$)**:**
Gegeben sind n Jobs, die auf bis zu m Prozessoren laufen können, sowie für jeden Job die Laufzeiten $t_j^k \in \mathbb{N} \setminus \{0\}$ des Jobs j, wenn er auf k Prozessoren gleichzeitig ausgeführt wird. Für die Variante $(\mathbf{Pm}|\mathbf{size_j}|\mathbf{C_{max}})$ ist die Joblaufzeit eines jeden Jobs fest und wird anstelle von $t_j^{\mathbf{size_j}}$ auch nur mit t_j bezeichnet.

1.2 Bisherige Arbeiten

Es gibt viele Veröffentlichungen zu verschiedensten Varianten des Zuteilungsproblems. Eine umfassende Übersicht inbesondere zu parallelen Problemvarianten gibt (Dro96).

Der preemptive Fall $(\mathbf{Pm}|\mathbf{size_j, pmtn}|\mathbf{C_{max}})$ mit *fixed-size Jobs* ist nach (BDW86) mit Hilfe eines linearen Programms in Polynomialzeit lösbar.
Für *moldable Jobs* wurde in (DL89) gezeigt, dass die preemptive Variante $(\mathbf{Pm}|\mathbf{spdp\text{-}any, pmtn}|\mathbf{C_{max}})$ $\mathcal{NP}$-schwer ist, und für $m \geq 2$ wurde ein pseudopolynomieller Algorithmus angegeben, der in dieser Arbeit zu einem FPTAS erweitert werden soll. Ist m nicht fest, ist das Problem stark $\mathcal{NP}$-schwer.
Die nicht-preemptive Variante $(\mathbf{P5}|\mathbf{size_j}|\mathbf{C_{max}})$ ist ebenfalls stark $\mathcal{NP}$-schwer, weshalb es für dieses Problem (unter der Annahme, dass $\mathcal{P} \neq \mathcal{NP}$) weder einen pseudopolynomiellen Algorithmus noch einen voll-polynomielles Approximationsschema gibt. Für die Varianten mit 2 oder 3 Prozessoren geben (DL89) allerdings pseudopolyinomielle Algorithmen an. Wie der Fall mit $m = 4$ Prozessoren einzuordnen ist, ist noch nicht bekannt.
Die nicht-preemptive Variante $(\mathbf{Pm}|\mathbf{size_j}|\mathbf{C_{max}})$ entspricht dem Rechteckanordnungsproblem im Streifen (siehe Unterabschnitt 2.1). In (Zus08) wird für dieses Problem ein ganzzahliges lineares Programm angegeben. Dies lässt sich zu einem Programm für $(\mathbf{Pm}|\mathbf{spdp\text{-}any}|\mathbf{C_{max}})$ erweitern, worauf nun eingegangen wird.

2 Scheduling ohne Preemption

2.1 Definition (Streifen-Packungsproblem)**:**
Gegeben ist ein Streifen der Breite m, und n Rechtecke $R_1, \ldots R_n$. Gesucht ist eine nicht-überlappende Anordnung der Rechtecke im Streifen mit möglichst geringer Höhe und einer Breite $\leq m$.

Das Streifen-Packungsproblem ist $\mathcal{NP}$-schwer zu optimieren. In (Zus08) wird dafür ein ganzzahliges lineares Programm angegeben. Dieses Programm wählt für die n Rechtecke jeweils die Koordinaten der unteren linken Ecke, und sorgt durch die Nebenbedingungen dafür, dass sich die Rechtecke nicht überlappen oder außerhalb des Streifens angeordnet werden.

2.1 Fixierte Prozessorzahlen

In der Problemvariante ($\mathbf{Pm|size_j|C_{max}}$) sollen n Jobs mit ihren Laufzeiten t_j und Prozessorenbedarf $\mathbf{size_j}$ so auf m Prozessoren verteilt werden, dass die Gesamtlaufzeit minimal ist. Dies entspricht genau dem Streifen-Packungsproblem, indem jedem Job j ein Rechteck mit Länge t_j und Breite $\mathbf{size_j}$ zugeordnet wird. Der Streifen, auf dem die Rechtecke mit möglichst geringer Höhe angeordnet werden müssen, hat dann genau Breite m, und die Höhe der Anordnung entspricht der Länge der entsprechenden Zuteilung. Die Jobs werden auf diese Weise offensichtlich auf benachbarte Prozessoren zugeteilt.

In dem ganzzahligen linearen Programm sollen die Entscheidungsvariablen $x_{j,p,q} \in \{0,1\}$ gewählt werden. Wird in der Lösung eine der Variablen $x_{j,p,q} = 1$ gesetzt, so bedeutet dies, dass Job j ab Prozessor p und Zeitpunkt q ausgeführt wird. Der spätest-mögliche Zeitpunkt, zu dem ein Job zugeteilt wird, muss bei dieser Formulierung des Programms beschränkt werden, um die Anzahl der auftretenden Variablen zu begrenzen. Man benötigt also eine obere Schranke für die Länge der Zuteilung.

2.2 Beobachtung (Obere Schranke für Opt)**:**
Die Länge $\textsc{Opt}(n,m,t)$ *einer optimalen Zuteilung*[1] *ist nicht größer als*

$$U := \sum_j t_j$$

Begründung: Eine gültige Zuteilung besteht bereits darin, alle Jobs ohne Ausnutzen von Parallelismus nacheinander abzuarbeiten. ◇

Das ILP mit Zielfunktion l hat dann die Nebenbedingungen

[1] Auch wenn wir uns natürlich für die tatsächliche Zuteilung interessieren, wird mit $\textsc{Opt}(n,m,t)$ auch die *Länge* einer solchen bezeichnet.

$$\sum_p \sum_q x_{j,p,q} = 1 \text{ für jeden Job } j \tag{1}$$

$$t_j + q \sum_p x_{j,p,q} \leq l \text{ für jeden Job } j \text{ und alle } q \tag{2}$$

$$\sum_j \sum_p \sum_q a_{j,p,q,r,s} \cdot x_{j,p,q} \leq 1 \text{ für jeden Punkt } (r,s) \tag{3}$$

Gleichung 1 stellt sicher, dass jeder Job genau einmal ausgeführt wird, während durch Gleichung 2 die Laufzeiten der Jobs innerhalb der Gesamtlaufzeit l liegen müssen.
Die Koeffizienten $a_{j,p,q,r,s}$ in Gleichung 3 sind 1, wenn bei Ausführung von Job j auf den $\mathbf{size_j}$ Prozessoren ab Prozessor p und Zeitpunkt q der Prozessor-Zeitpunkt (r,s) durch diese Job belegt wäre. Durch die 3. Gleichung wird so ausgeschlossen, dass ein Prozessor mehr als einen Job gleichzeitig ausführt.

Als Bereiche für die Indizes der Entscheidungsvariablen $x_{j,p,q}$ ergeben sich insgesamt also $1 \leq j \leq n$ für die Jobs, $0 \leq p < m - \mathbf{size_j}$ für den ersten von Job j verwendeten Prozessor sowie $0 \leq q < U - t_j$ für seinen Startzeitpunkt. Die Punkte (r,s) in Gleichung 3, die auf Überlappungen geprüft werden müssen, sind diejenigen aus dem Rechteck $\{0,\ldots,m-1\} \times \{0,\ldots,U-1\}$.

2.2 Wählbare Prozessorzahlen

Um das $(\mathbf{Pm|spdp\text{-}any|C_{max}})$-Problem zu lösen, kann die Formulierung des Programms abgewandelt werden. In der abgeänderten Formulierung entspricht dann ein Job mit den Laufzeiten t_j^k nicht mehr nur einem einzelnen Rechteck, sondern einer ganzen *Menge von m Rechtecken* mit den Breiten $1,\ldots,m$ und den Höhen $t_j^1,\ldots,t_j^m$. Den n Jobs des ursprünglichen Problems entsprechen also n Rechteckmengen mit je m Elementen, und es wird aus jeder der n Mengen jeweils genau eines der m Rechtecke angeordnet. Auch hier werden Jobs offensichtlich stets auf benachbarten Prozessoren ausgeführt.
Man sieht leicht, dass jedes $(\mathbf{Pm|spdp\text{-}any|C_{max}})$-Problem mindestens so schwierig wie das Streifen-Packungsproblem ist, indem man jedem Rechteck einen Job zuordnet, der nur auf einer bestimmten Prozessorenzahl effizient[2] ausgeführt werden kann.

[2] z.B. indem man für alle übrigen Prozessorzahlen eine Laufzeit $> \sum_j \min_k t_j^k$ wählt (siehe Beobachtung 2.3)

In dem modifizierten ganzzahligen linearen Programm müssen die Entscheidungsvariablen $x_{j,k,p,q} \in \{0,1\}$ gewählt werden. Wird in der Lösung eine der Variablen $x_{j,k,p,q} = 1$ gesetzt, so bedeutet dies nun, dass Job j **auf** k **Prozessoren** ab Prozessor p und Zeitpunkt q ausgeführt wird.

Die Bereiche für die Indizes der Entscheidungsvariablen $x_{j,k,p,q}$ und die Grenzen der Rechteckpunkte (r,s) ergeben sich ähnlich wie eben, hinzu kommt jedoch, dass $1 \leq k \leq m$ für die Anzahl der von Job j verwendeten Prozessoren steht. Man benötigt außerdem wieder eine

2.3 Beobachtung (Obere Schranke für Opt)**:**
Die Länge $\textsc{Opt}(n,m,t)$ *einer optimalen Zuteilung ist nicht größer als*

$$U := \sum_j \min_k t_j^k$$

Begründung: Wie in Beobachtung 2.2 werden die Jobs nacheinander ohne Ausnutzen von Preemption oder Parallelismus abgearbeitet, und zwar mit der jeweils kürzestmöglichen Laufzeit. ◇

Die Bedingungen für das neue Minimierungsproblem mit Zielfunktion l lauten dann:

$$\sum_k \sum_p \sum_q x_{j,k,p,q} = 1 \text{ für jeden Job } j \tag{4}$$

$$\sum_k \sum_p \sum_q (t_j^k + q) \cdot x_{j,k,p,q} \leq l \text{ für jeden Job } j \tag{5}$$

$$\sum_j \sum_k \sum_p \sum_q a_{j,k,p,q,r,s} \cdot x_{j,k,p,q} \leq 1 \text{ für jeden Punkt } (r,s) \tag{6}$$

Gleichung 4 stellt wieder sicher, dass jeder Job genau einmal ausgeführt wird, und Gleichung 5 legt fest, dass die Laufzeiten der Jobs innerhalb der Gesamtlaufzeit l liegen müssen.
Die Koeffizienten $a_{j,k,p,q,r,s}$ in Gleichung 6 (die wieder ausschließt, dass ein Prozessor mehrere Jobs gleichzeitig bearbeitet) sind nun also 1, wenn bei Ausführung von Job j auf den k Prozessoren ab Prozessor p und Zeitpunkt q der Prozessor-Zeitpunkt (r,s) durch diese Job belegt wäre.
Wie man sieht, ist die Größe des linearen Programms auch von der Güte der verwendeten oberen Schranke abhängig. Größere Probleminstanzen lassen sich auf diese Weise verständlicherweise nicht in angemessener Zeit lösen.

Es sei noch darauf hingewiesen, dass in (Zus08) außerdem ein Branch-and-Bound-Algorithmus zur Lösung des Problems angegeben wird. Auch dieser Algorithmus kann so erweitert werden, dass er jeweils ein Rechteck aus n Rechteckmengen auswählt, was hier jedoch nicht betrachtet wird.
Stattdessen soll nun untersucht werden, inwieweit Preemption das Problem verändert.

3 Scheduling mit Preemption

Dass (DL89) einen pseudopolynomiellen Algorithmus für die Problemvariante ($\mathbf{Pm|spdp\text{-}any,\ pmtn|C_{max}}$) angegeben haben, wurde bereits angesprochen. Der dort vorgestellte Algorithmus basiert darauf, dass mehrere ($\mathbf{Pm|size_j,\ pmtn|C_{max}}$)-Probleminstanzen erzeugt und gelöst werden, weshalb nun zunächst auf diese Variante eingegangen werden soll.

3.1 Fixierte Prozessorzahlen

In (BDW86) wurde das ($\mathbf{Pm|size_j,\ pmtn|C_{max}}$)-Problem untersucht. Die Autoren geben das folgende lineare Programm Fixed-Opt(n, m, t) an, um das Problem zu lösen:

$$\min \sum_{q \in Q} x_q \qquad \text{Zielfunktion} \tag{7}$$

$$\sum_{q \in Q_j} x_q = t_j \qquad \text{für alle Jobs } j \tag{8}$$

$$x_q \geq 0 \qquad \text{für alle } q \in Q \tag{9}$$

Dabei bezeichnet Q die Menge aller zulässigen Aufteilungen von Jobs auf Prozessoren. Eine Aufteilung heißt zulässig, wenn nicht mehr als m Prozessoren benötigt werden, um alle Jobs dieser Aufteilung gleichzeitig zu bearbeiten. Die Elemente $q \in Q$ sind also im Wesentlichen nichtleere Mengen von Jobs, für die $\sum_{j \in q} \mathbf{size_j} \leq m$ gilt.

Q enthält demnach höchstens $\mathcal{O}((n+1)^m)$ Elemente, da für jede zulässige Menge höchstens m Prozessoren gleichzeitig arbeiten. Jedem dieser Prozessoren kann also entweder einer der $1 \ldots n$ „echten“ Jobs oder ein virtueller „Idle“-Job zugeordnet werden, der nicht zur Menge der zuzuteilenden Jobs beiträgt, sondern nur zum Abzählen der Größe von Q gedacht wird.

Q_j ist schließlich die Menge derjenigen Jobaufteilungen, die Job j enthalten. Gleichung 8 sichert also zu, dass jeder Job in genau so vielen Zeiteinheiten läuft, wie er insgesamt benötigt.

Das lineare Programm besteht somit aus den n Gleichungen 8 und $\mathcal{O}((n+1)^m)$ Variablen x_q mit zugehöriger Nichtnegativitätsbedingung. Daher ist es für festes m in Polynomialzeit lösbar. Die Lösung des linearen Programms gibt dann für jede zulässige Menge von Jobs an, wie lang sie laufen soll. Hierbei ist nicht explizit angegeben, auf welchen Prozessoren die Jobs laufen sollen. Diese kann man nun aber natürlich so wählen, dass für einen Job benachbarte Prozessoren verwendet werden. Über Preemptionsschritte hinweg kann diese Zuteilung jedoch wechseln.

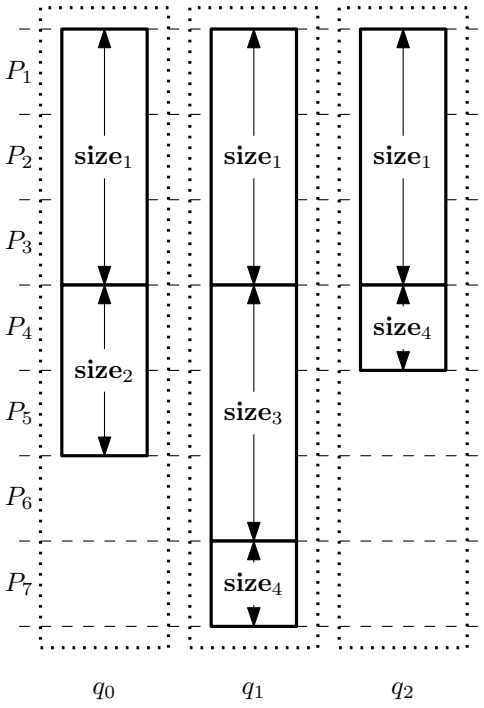


Abbildung 1: Drei Beispiele für zulässige Aufteilungen mit 7 Prozessoren

3.2 Wählbare Prozessorzahlen

Für die (**Pm|spdp-any, pmtn|C_{max}**)-Variante des Problems wurde in (DL89) ein pseudopolynomieller Algorithmus angegeben.

Grundidee Die Idee für den Algorithmus ergibt sich aus Zusammenhängen zwischen (**Pm|size$_j$, pmtn|C_{max}**)- und (**Pm|spdp-any, pmtn|C_{max}**)-Problemen. Um diese aufzuzeigen werden nun (**Pm|size$_j$, pmtn|C_{max}**)-Instanzen und deren Zuteilungen genauer betrachtet. Das Schema, nach dem eine solche Zuteilung aufgebaut ist, lässt sich durch die Aufteilungen der Prozessoren und die Laufzeiten dieser Aufteilungen beschreiben.

3.1 Definition (preemptives Zuteilungsschema)**:**
Ein Paar (K, L), wobei L eine Liste von nichtnegativen reellen Laufzeiten,

und K eine Liste mit Multimengen von Prozessoren ist, heißt preemptives Zuteilungsschema (für m Prozessoren), wenn die Summe der Elemente jeder einzelnen Multimenge m nicht übersteigt, und die Längen der beiden Listen gleich sind.

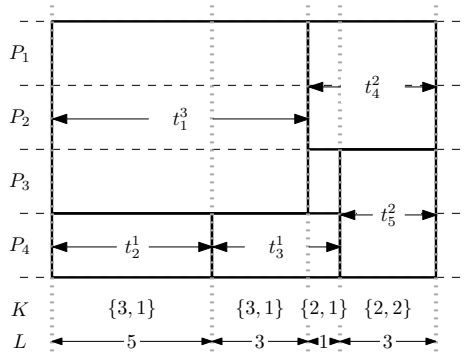


Abbildung 2: Zuteilung mit zugehörigem Zuteilungsschema

Die Multimengen in K geben dabei an, wie viele Prozessoren parallel an einem Job arbeiten. Die Multimenge $\{3, 1\}$ bedeutet also Beispielsweise, dass ein 3-Prozessor-Job und ein 1-Prozessor-Job gleichzeitig bearbeitet wird. Der entsprechende Eintrag in der Liste L gibt an, wie lange diese Zuteilung bearbeitet wird. In preemptiven Zuteilungen müssen diese nicht notwendigerweise ganzzahlig sein.
In einem Zuteilungsschema ist also nicht festgehalten, welche Jobs laufen, sondern nur, wie viele Prozessoren wie lange parallel arbeiten. Zu jeder Zuteilung lässt sich leicht ein Zuteilungsschema angeben. Umgekehrt entspricht natürlich nicht jedes Zuteilungsschema einer gültigen Zuteilung. (DL89) geben eine Bedingung dafür an, unter welchen Umständen ein Zuteilungsschema zulässig für ein ($\mathbf{Pm|size_j,\ pmtn|C_{max}}$)-Problem ist, es also eine Zuteilung gibt, die dieses Zuteilungsschema ergibt. Aus dieser kann dann das folgende Lemma geschlossen werden.

3.2 Lemma:
Die Länge einer optimalen Zuteilung zweier ($\mathbf{Pm|size_j,\ pmtn|C_{max}}$)-Probleme ist gleich, wenn für jedes $1 \leq k \leq m$ sowohl die Joblaufzeiten der $(\lfloor m/k \rfloor - 1)$ größten k-Prozessor-Jobs als auch die Gesamtlaufzeit aller übrigen k-Prozessor-Jobs in beiden Probleminstanzen gleich ist.

Der Algorithmus nutzt nun aus, dass jede Zuteilung für eine Instanz der ($\mathbf{Pm|spdp\text{-}any,\ pmtn|C_{max}}$)-Variante auf natürliche Art und Weise einer Zuteilung für ein ($\mathbf{Pm|size_j,\ pmtn|C_{max}}$)-Problem entspricht, indem die Prozessoranzahlen auf die in der Zuteilung verwendete Anzahl festgelegt werden. Wegen Lemma 3.2 ergibt aber nicht jede mögliche Festlegung der Jobs auf Prozessoranzahlen eine andere Zuteilungslänge, sondern höchstens diejenigen Festlegungen, die ein unterschiedliches preemptives Zuteilungsschema ergeben. Im Abschnitt zur Laufzeit des Algorithmus wird erläutert, dass dies

nur pseudopolynomiell viele sind. Diese werden dann mit dem optimalen Algorithmus aus Unterabschnitt 3.1 gelöst. Daher gelten die dort gemachten Bemerkungen über Zuteilung der Jobs auf benachbarte Prozessoren auch hier.

Beschreibung des Algorithmus Im Folgenden sei $n_k := \lfloor \frac{m}{k} \rfloor - 1$, d.h. $n_k = 0$ für $k > \frac{m}{2}$. n_k gibt also an, von wie vielen größten k-Prozessor-Joblaufzeiten die Länge einer optimalen Zuteilung abhängt. Die Abhängigkeit der Länge einer optimalen Zuteilung von den $H := \sum_{k=1}^{m}(n_k + 1) \in \mathcal{O}(m \log m)$ Zahlen bietet einen Ansatz zur dynamischen Programmierung auf einer H-dimensionalen Tabelle F.
An Stelle $F[j, x_1^1, \ldots, x_{n_1}^1, x_*^1, \ldots, x_1^{\lfloor m/2 \rfloor}, \ldots x_{n_{\lfloor m/2 \rfloor}}^{\lfloor m/2 \rfloor}, x_*^{\lfloor m/2 \rfloor}, \ldots, x_*^{m-1}]$ der Tabelle wird gespeichert, wieviel Zeit für m-Prozessor-Jobs verbraucht werden muss, wenn:

- In einem Schedule mit den Jobs 1 bis j
- der größte 1-Prozessor-Job x_1^1 Zeiteinheiten,
- der zweitgrößte 1-Prozessor-Job x_2^1,
  $\vdots$
- alle restlichen 1-Prozessor-Jobs zusammen x_*^1 Zeiteinheiten,
  $\vdots$
- der $n_{\lfloor m/2 \rfloor}$t-größte $\lfloor \frac{m}{2} \rfloor$-Prozessor-Job $x_{n_{\lfloor m/2 \rfloor}}^{\lfloor m/2 \rfloor}$,
- alle restlichen $\lfloor \frac{m}{2} \rfloor$-Prozessor-Jobs zusammen $x_*^{\lfloor m/2 \rfloor}$,
  $\vdots$
- und alle $(m-1)$-Prozessor-Jobs zusammen x_*^{m-1}

Zeiteinheiten in Anspruch nehmen.

Die Laufzeit der m-Prozessor-Jobs wird also in Abhängigkeit von j sowie für jedes $k < m$ abhängig von den Laufzeiten der n_k größten k-Prozessor-Jobs und der Laufzeit aller übrigen k-Prozessor-Jobs zusammen gespeichert. Dieser Eintrag zusammen mit den Werten $X = (x_1^1, \ldots, x_*^{m-1})$ des Indexvektors beschreibt also ein bestimmtes Zuteilungsschema.
Die Einträge in der j-ten Zeile hängen lediglich von den Einträgen der vorherigen Zeile und (natürlich) den Laufzeiten t_j° von Job j ab.

Ein neuer Eintrag $F[j, X]$ kann sich dadurch ergeben, dass j als m-Prozessor-Job zugeteilt wird, was entsprechend bedeutet, dass der Eintrag

an derselben Stelle in der vorherigen Zeile um t_j^m vergrößert wird (d.h. $F[j, X] \leftarrow F[j-1, X] + t_j^m$).
Unter Umständen würde der Tabelleneintrag $F[j, X]$ jedoch kleiner, wenn Job j auf nur $k < m$ Prozessoren ausgeführt wird. Das Zuteilungsschema, das diesem Tabelleneintrag entspricht, unterscheidet sich also nur in der Länge der m-Prozessor-Jobs. Damit ist aber auch die Länge einer Zuteilung für dieses Schema kleiner, denn die m-Prozessor-Jobs können nur sequentiell abgearbeitet werden. Das Schema für die übrigen Jobs ist aber dasselbe, und somit auch gleich lang.
Ein Job kann nur dann als k-Prozessor-Job zugeteilt werden, wenn die Laufzeit des Jobs $t_j^k = x_\circ^k$, d.h. j einer der n_k größten k-Prozessor-Jobs ist, oder wenn er kleiner als die n_k größten k-Prozessor-Jobs und nicht größer als die für übrige k-Prozessor-Jobs zur Verfügung stehende Zeit x_*^k ist. In diesem Fall erhält man den Eintrag $F[j, X]$ mit Hilfe desjenigen Eintrags aus der vorherigen Zeile, der sich ergibt, indem dort die übrigen k-Prozessor-Jobs ggfs. „verschoben“ werden. Mit anderen Worten: $F[j, X] \leftarrow F[j-1, X']$ für ein geeignetes X'. Für die genaue Vorgehensweise zur Bestimmung von X' siehe die Beschreibung der Funktion k-Proc.
Wird ein Eintrag in Zeile j ausgefüllt, wird außerdem festgehalten, auf wie vielen Prozessoren Job j laufen muss, um den Eintrag zu erhalten.

Wurde die letzte (n-te) Zeile der Tabelle berechnet, steht für die gültigen Einträge fest, welche Festlegung der Jobs auf Prozessorzahlen zu diesem Eintrag geführt haben. Mit dem Algorithmus Fixed-Opt kann dann eine optimale Zuteilung für die entsprechenden Probleminstanzen berechnet werden. Die beste so gefundene Zuteilung ist optimal für das Problem und wird ausgegeben.

```
OPT(n, m, t)
 1  for all t_j^k
 2      do if t_j^k > Σ_j min_k t_j^k
 3          then t_j^k ← ∞ ▷ Lange Joblaufzeiten ignorieren
 4  F[0, ∘] ← ∞
 5  F[0, ..., 0] ← 0 ▷ Initialisierung
 6  for Job j ← 1 to n
 7      do for all Indices X = (x_1^1, ..., x_*^{m-1})
 8          do F[j, X] ← min{
 9              F[j − 1, X] + t_j^m,            ▷ m-Prozessor-Job
10              k-Proc(m − 1, j, t, F, X),      ▷ (m − 1)-Prozessor-Job
                ⋮
11              k-Proc(1, j, t, F, X)           ▷ 1-Prozessor-Job
            }
12          Halte fest, welche Wahl minimal war.
13  for all Indices X = (x_1^1, ..., x_*^{m-1})
14      do if F[n, X] ≠ ∞
15          do Fixiere Prozessorzahlen gemäß der Wahl in
               Zeilen 9 bis 11. Erhalte Laufzeiten t'
16             FIXED-OPT(n, m, t')
17  return die kleinste so erhaltene Zuteilung
```

Die in Beobachtung 2.3 angegeben obere Schranke für die Gesamtlänge einer Zuteilung gilt auch im preemptiven Fall, da jede nicht-preemptive Zuteilung auch eine gültige preemptive Zuteilung ist. Die in Zeile 3 vorgenomme Reduzierung der Eingabe ist daher zulässig, denn längere Jobs würden für eine optimale Zuteilung niemals ausgewählt. Dies ist eine leichte Abwandlung des Algorithmus von (DL89), die benötigt wird, um für das später in Unterabschnitt 3.3 vorgestellte FPTAS polynomielle Laufzeit zu erhalten.

Die von (DL89) angegebenen Bereiche der Indizes $x_\circ^k$ sind recht ungenau abgeschätzt. Sie lassen sich zur Implementierung mit einer einfachen Überlegung stark einschränken, denn der größte k-Prozessor-Job kann selbstverständlich nicht mehr Zeit als $\max_j t_j^k$, also der Längste im System vorhandene k-Prozessor-Job benötigen. Dies bedeutet nichts anderes als

$$x_1^k \leq \max_j t_j^k.$$

Analog kann man so fortfahren, bis schließlich die Laufzeit aller übrigen k-

Prozessor-Jobs nicht größer als die Summe der verbleibenden $n-n_k$ kleinsten k-Prozessor-Laufzeiten sein kann.

Bestimmen eines Tabelleneintrags Im Algorithmus OPT wird zur Bestimmung eines Tabelleneintrags die Funktion k-Proc aufgerufen. Diese ordnet Job j entweder als einen der n_k größten oder einen der übrigen k-Prozessor-Jobs ein. Ist beides nicht möglich, wird ∞ zurückgegeben.

k-Proc(k, j, t, F, X)

▷ Entweder als i-t-größten Job einordnen

1 **for** $i \leftarrow 1$ **to** n_k

2 **do if** $t_j^k = x_i^k$

3 **then** ▷ Job j soll i-t-größter k-Prozessor-Job werden

4 $x_l^k \leftarrow x_{l+1}^k$ für alle $l \geq i$

5 $y_{max} \leftarrow min\{x_{n_k}^k, x_*^k\}$

6 **return** $\min_{y=0}^{y_{max}}\{F[j-1, X : x_{n_k}^k \leftarrow y : x_*^k \leftarrow (x_*^k - y)]\}$ [a]

▷ Oder Job j in der Summe x_*^k unterbringen

7 **if** $t_j^k \leq x_*^k$ **and** $t_j^k < x_{n_k}^k$

8 **then return** $F[j-1, X : x_*^k \leftarrow (x_*^k - t_j^k)]$

▷ Wenn beides nicht geht:

9 **return** ∞

[a] Die Schreibweise $X : a \leftarrow b$ soll bedeuten, dass in der Koordinate X der Eintrag an Stelle a durch den Wert b ersetzt wird.

Das Verrücken der Joblaufzeiten in Zeile 4 trägt der Tatsache Rechnung, dass vor der Zuteilung von Job j als i-t-größter k-Prozessor-Job ein anderer an dieser Stelle stand.
Die Laufzeit des n_k-t-größten k-Prozessor-Jobs muss jedoch rekonstruiert werden, da dieser durch die getroffene Zuteilung von j in die Summe x_*^k „verdrängt“ wird. In Zeile 6 wird daher der bestmögliche n_k-t-größte Job in der $(j-1)$-ten Tabellenzeile gesucht, der durch die Wahl von Job j als i-t-größter Job in die Summe der übrigen k-Prozessor-Jobs „verdrängt“ wurde. Der Eintrag für diese Zuteilung in Tabellenzeile j hätte sich nämlich dadurch ergeben, dass stattdessen y Zeiteinheiten weniger für x_*^k in Tabellenzeile $(j-1)$ zur Verfügung gestanden hätten.
Die Laufzeit dieses verdrängten Jobs kann natürlich weder die des nächstgrößeren, noch die zur Verfügung stehende Restlaufzeit x_*^k für k-Prozessor-Jobs überschreiten. Somit läuft y bis zu dem in Zeile 5 gewählten Wert y_{max}.

War es nicht möglich, Job j als einen der n_k größten k-Prozessor-Jobs zuzuteilen, so bleibt noch die Möglichkeit, ihn in Zeile 8 als einen der übrigen k-Prozessor-Jobs, die insgesamt x_*^k Zeiteinheiten verbrauchen, unterzubringen. Ist auch dies nicht möglich, kann der Job nicht mit den durch X vorgegebenen Laufzeiten als k-Prozessor-Job zugeteilt werden, und es wird ∞ zurückgegeben.

Laufzeit des Algorithmus Pro Tabellenzeile sind $\mathcal{O}(M^{H-1})$ Tabelleneinträge zu bestimmen, wobei $M \leq \sum_j \min_k t_j^k$ wegen Zeile 3 im Algorithmus Opt ist.

Jeder Eintrag in der Tabelle lässt sich in Zeit $\mathcal{O}(H+M)$ bestimmen, denn im Wesentlichen müssen die m Aufrufe von k-Proc bearbeitet werden. In einem solchen wird die Schleife aus Zeile 1 höchstens $\mathcal{O}(H)$ mal durchlaufen, und in höchstens einem der Durchläufe wird in Zeile 6 erneut über $\mathcal{O}(M)$ Elemente iteriert. Für die Erzeugung der insgesamt n Tabellenzeilen ergibt sich daher eine Laufzeit von $\mathcal{O}(n(H+M)M^{H-1}) \subseteq \mathcal{O}(nHM^H)$.
Hinzu kommen noch die $\mathcal{O}(M^{H-1})$ Aufrufe des Polynomialzeitalgorithmus Fixed-Opt aus Zeile 16 im Algorithmus Opt. Insgesamt ergibt sich damit für Opt eine asymptotische Laufzeit von $\mathcal{O}(\mathbf{poly}(n) \cdot HM^H)$. Da $H \in \mathcal{O}(m \log m)$ als konstant angenommen wird, und M polynomiell in der Größe der Eingabe ist, ist die Laufzeit des Algorithmus pseudopolynomiell.

3.3 Voll-polynomielles Approximationsschema

Der eben vorgestellte Algorithmus lässt sich zu einem voll-polynomiellen Approximationsschema (FPTAS, fully-polynomial time approximation scheme) erweitern.

3.3 Definition (voll-polynomielles Approximationsschema)**:**
Ein Approximationsschema AS *ist ein Algorithmus, der als Eingabe eine Instanz I der Länge n eines Optimierungsproblems, sowie eine Zahl $\varepsilon > 0$ hat, und dessen Ausgabe* $\mathrm{AS}(\varepsilon, I)$ *sich höchstens um den Faktor* $(1+\varepsilon)$ *von einer optimalen Lösung für I unterscheidet. Das heißt* $\mathrm{AS}(\varepsilon, I) \leq (1+\varepsilon) \cdot \textsc{Opt}(I)$. *Ein Approximationsschema heißt* polynomiell*, wenn die Laufzeit des Algorithmus polynomiell in der Eingabelänge n ist, und* voll-polynomiell*, wenn die Laufzeit zusätzlich polynomiell in $1/\varepsilon$ ist.*

Für den Entwurf eines voll-polynomiellen Approximationsschemas werden die folgenden Beobachtungen hilfreich sein:

3.4 Beobachtung (Steigung von OPT)**:**
Die Steigung von $\textsc{Opt}(n, m, t)$*, d.h. die Länge einer optimalen Zuteilung als Funktion in Abhängigkeit der Joblaufzeiten* t_j^k*, ist in allen* t_j^k *mindestens* 0 *und höchstens* 1*.*

Begründung: Die Steigung kann nicht negativ sein, denn könnte man die Gesamtlaufzeit verkleinern, indem man eines der t_j^k vergrößert, so würde die gleiche Zuteilung das ursprüngliche Problem in kürzerer Zeit lösen. Ebenso kann die Steigung nicht größer als 1 werden, denn im schlimmsten Fall kann die zusätzliche Laufzeit nach dem Ende der ursprünglichen Zuteilung abgearbeitet werden. ◇

3.5 Beobachtung (Untere Schranke für OPT)**:**
Es gilt

$$\textsc{Opt}(n, m, t) \geq \max_j \min_k t_j^k \geq \frac{1}{n} \cdot \sum_j \min_k \{t_j^k\}.$$

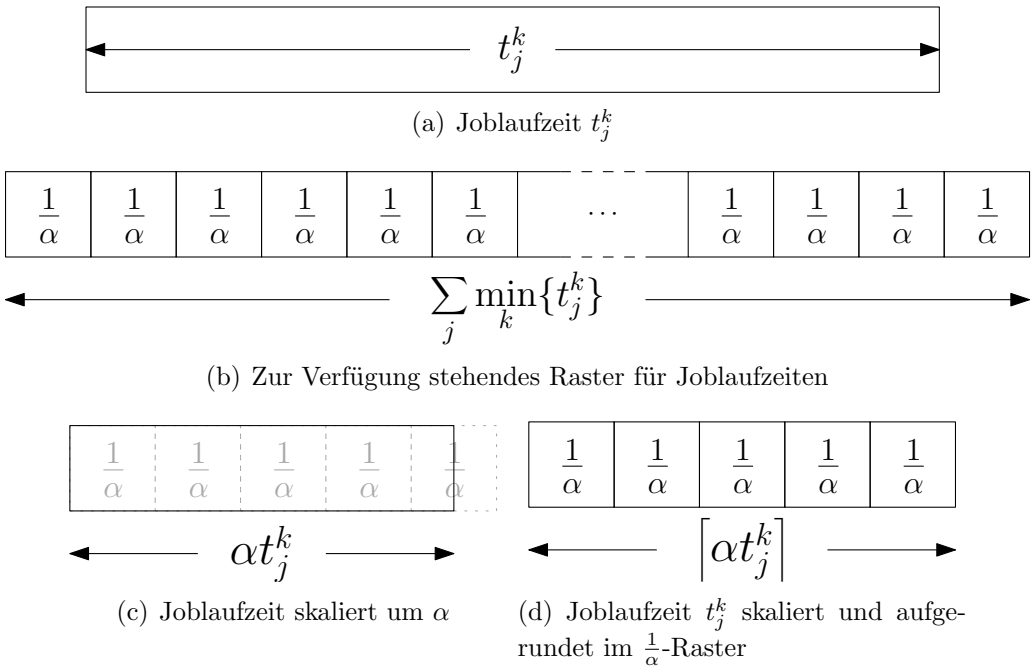
Begründung: Es wird mindestens der kleinstmögliche Job zugeteilt. ◇

Die Grundidee des Algorithmus ist es, die Joblaufzeiten um einen Faktor $\alpha \leq 1$ zu skalieren, sodass die Joblaufzeiten nur noch polynomielle Größe in n haben. Für diese neuen Joblaufzeiten $t_j'^k := \lceil \alpha t_j^k \rceil$ wird dann mit dem vorgestellten pseudopolynomiellen Algorithmus eine optimale Zuteilung bestimmt. Die gefundene Zuteilung (entsprechend um $\frac{1}{\alpha}$ Skaliert) ist dann auch eine gültige Lösung für das ursprüngliche Problem. Die Joblängen wurden ja aufgerundet, weshalb ein Job höchstens zu viel, nicht aber zu wenig Laufzeit zugeteilt bekommt. Es ergibt sich der folgende Algorithmus:

FPTAS(ε, n, m, t)
1 $S \leftarrow \sum_j \min_k \{t_j^k\}$
2 $\alpha \leftarrow n^2/\varepsilon S$
3 $t_j'^k \leftarrow \lceil \alpha t_j^k \rceil$
4 $z \leftarrow \textsc{Opt}(n, m, t')$ ▷ Zuteilung für reduziertes Problem
5 **return** $z \cdot \frac{1}{\alpha}$ ▷ Zuteilung für ursprüngliches Problem

Bezüglich der Zuteilung von Jobs auf benachbarte Prozessoren in der gefundenen Zuteilung gelten wieder die gleichen Bemerkungen wie schon in Unterabschnitt 3.1.

Approximationsgüte Durch die Rundung der Joblaufzeiten t' kann einem Job so mehr Zeit zugeteilt werden, als er ursprünglich benötigt hat. Der



(a) Joblaufzeit t_j^k

(b) Zur Verfügung stehendes Raster für Joblaufzeiten

(c) Joblaufzeit skaliert um α

(d) Joblaufzeit t_j^k skaliert und aufgerundet im $\frac{1}{\alpha}$-Raster

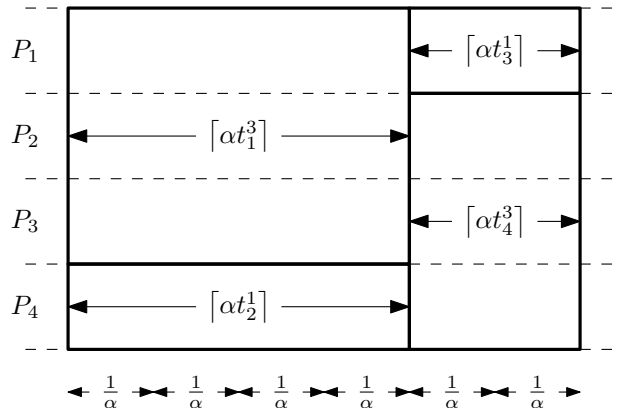
Abbildung 3: Rasterung der Joblängen

Rundungsfehler für einen zugeteilten Job ist allerdings unabhängig von k höchstens:

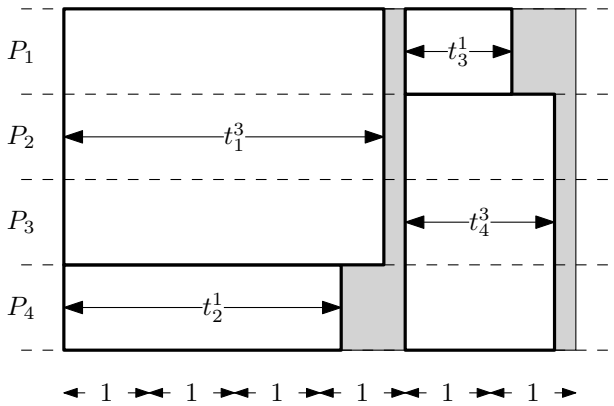
$$\underbrace{t_j'^k - t_j^k}_{\text{Rundungsfehler}} = \frac{\lceil \alpha t_j^k \rceil}{\alpha} - t_j^k \leq \frac{\alpha t_j^k + 1}{\alpha} - t_j^k = \frac{1}{\alpha}$$

Fasst man eine optimalen Zuteilung für das ursprüngliche Problem mit den Joblaufzeiten t_j^k als Zuteilung für das neue Problem (mit den $t_j'^k$) auf, indem man die Laufzeiten in der Zuteilung entsprechend rundet, so ändert sich die Länge dieser Zuteilung wegen Beobachtung 3.4 für jeden der n Jobs höchstens um den Rundungsfehler. Der Gesamtfehler der Approximationslösung ist demnach höchstens n-mal so groß wie der Rundungsfehler pro Job, da sie optimale Lösung für das neue Problem ist.
Wählt man wie in Zeile 2 des Algorithmus angegeben

$$\alpha = \frac{n^2}{\varepsilon \cdot \underbrace{\sum_j \min_k \{t_j^k\}}_{S}},$$



(a) Durch FPTAS berechnete Zuteilung



(b) Verlust durch Rundung

Abbildung 4: Auftretender Verschnitt durch Aufrunden der Joblängen

so ergibt sich die Beschränkung des absoluten Fehlers durch $\varepsilon \cdot \textsc{Opt}$:

$$\begin{aligned} F_{abs} &\leq \frac{n}{\alpha} && \text{wegen Beobachtung 3.4} \\ &= \frac{\varepsilon}{n} \cdot \sum_j \min_k t_j^k && \text{Wahl von } \alpha \\ &\leq \varepsilon \cdot \textsc{Opt}(n, m, t) && \text{wegen Beobachtung 3.5} \end{aligned}$$

Vom Algoritmus Opt werden im reduzierten Problem diejenigen Joblängen, die größer als $\sum_j \min_k t_j'^k$ sind, nicht beachtet werden. Die Abschätzungen für die Approximationsgüte gelten dennoch auch für die so modifizierte

Probleminstanz, da eine optimale Lösung berechnet wird. Die ignorierten Joblängen werden in einer solchen nämlich nicht auftreten.
Die Approximationsgüte ist also

$$\text{FPTAS}(\varepsilon, n, m, t) \leq \textsc{Opt}(n, m, t) + F_{abs} \leq (1 + \varepsilon) \cdot \textsc{Opt}(n, m, t).$$

Laufzeit des Approximationsalgorithmus Durch die Problemreduzierung ändern sich n, m, und somit auch H aus der Laufzeit von Opt nicht. In dem reduzierten Problem ist nun jedoch

$$\begin{aligned} M &= \sum_j \min_k t_j'^k = \sum_j \min_k \lceil \alpha t_j^k \rceil \\ &= \sum_j \min_k \left\lceil \frac{n^2 t_j^k}{\varepsilon S} \right\rceil \\ &\leq \sum_j \min_k \left\{ \frac{n^2 t_j^k}{\varepsilon S} + 1 \right\} \end{aligned}$$

Nun wird (wie auf Seite 14 angekündigt) benötigt, dass Opt diejenigen Joblaufzeiten $t_j'^k > \sum_j \min_k \lceil \alpha t_j^k \rceil =: S'$ nicht beachtet. Dies bedeutet, dass in obiger Gleichung für die t_j^k gilt:

$$t_j^k \leq \frac{1}{\alpha} \sum_j \min_k \lceil \alpha t_j^k \rceil \leq \frac{1}{\alpha} \sum_j \min_k \{\alpha t_j^k + 1\} = S + \frac{n}{\alpha}. \qquad (*)$$

Dies bedeuted wiederum für M:

$$\begin{aligned} M &\leq \sum_j \min_k \left\{ \frac{n^2 t_j^k}{\varepsilon S} + 1 \right\} && \text{Gleichung von oben} \\ &\leq \sum_j \min_k \left\{ \frac{n^2 (S + \frac{n}{\alpha})}{\varepsilon S} + 1 \right\} && \text{wegen } (*) \\ &= n \left(\frac{n^2}{\varepsilon} + \frac{n^3}{\alpha \varepsilon S} + 1 \right) = \frac{n^3}{\varepsilon} + n^2 + n \end{aligned}$$

Mit der Laufzeit aus Abschnitt 3.2 ist die Gesamtlaufzeit des Algorithmus FPTAS dann

$$
\begin{aligned}
&\in \mathcal{O}(\mathbf{lp}(n) \cdot nHM^H) && \text{Laufzeit von OPT} \\
&= \mathcal{O}(\mathbf{lp}(n) \cdot H(n^4/\varepsilon + n^3 + n^2)^H) && M \text{ eingesetzt} \\
&\subseteq \mathcal{O}(\mathbf{poly}(n, 1/\varepsilon)) && \text{für konstante Prozessorzahl } m
\end{aligned}
$$

Dabei ist $\mathbf{lp}(n)$ die polynomielle Laufzeit zum Lösen eines linearen Programms FIXED-OPT, und $H \in \mathcal{O}(m \log m)$ die Größe der Tabelle des Algorithmus OPT. Insgesamt ergibt sich der folgende

3.6 Satz:
Der Algorithmus FPTAS(ε, n, m, t) *ist ein voll-polynomielles Approximationsschema für das Zuteilungsproblem* ($\mathbf{Pm|spdp\text{-}any,pmtn|C_{max}}$).

Heuristische Verbesserung der Lösung Die gefundene Zuteilung kann, wie in Abbildung 4 erkenntlich, Leerlaufzeiten aufgrund der auftretenden Rundungsfehler enthalten. Man kann dies durch Bestimmen einer optimalen Lösung für die gefundene Prozessorzuteilung vermeiden. Die asymptotische Laufzeit ändert sich dadurch nicht, die Qualität der Lösungen nimmt aber, wie in Unterabschnitt 4.3 noch erläutert wird, deutlich zu.

3.4 Greedy-Heuristiken

In Hinblick auf die experimentelle Auswertung der Algorithmen ist der Vergleich mit heuristisch ermittelten Lösungen interessant. Probleme mit fixierter Prozessorzahl und Preemption sind wie in Unterabschnitt 3.1 besprochen in Polynomialzeit optimal lösbar. Es liegt daher nahe, mittels einer schnellen Heuristik die Prozessorzahlen der Jobs zu fixieren, und das daraus entstehende ($\mathbf{Pm|size_j,pmtn|C_{max}}$)-Problem optimal zu lösen.

Eine einfache Heuristik besteht darin, für Jobs diejenige Prozessorzahl zu wählen, welche den größten Speedupzuwachs, und damit die größte Leistungssteigerung ergibt:

GREEDY-GAIN(n, m, t)

1 $spdp[j,k] \leftarrow t_j^k / t_j^1$
2 $\mathbf{size_j} \leftarrow \arg\max_k(spdp[j,k] - spdp[j,k-1])$ [a]
3 **return** FIXED-OPT$(n, m, t_j^{\mathbf{size_j}})$

[a] Wird das Maximum für mehrere k angenommen, wird das Kleinste gewählt

Ein anderes Auswahlkriterium ist der Quotient $\frac{spdp[j,k]}{k}$ von Speedup zu verwendeter Prozessorzahl, d.h. die Effizienz der für den Job gewählten Prozessoranzahl. In der folgenden Auswertung wird diese Heuristik mit GREEDY-EFFICIENCY bezeichnet.

4 Experimentelle Auswertung

Die in Abschnitt 2 vorgestellten linearen und ganzzahligen linearen Programme werden schon für relativ kleine Eingaben umfangreich. Daher ist vor allem im Falle des ganzzahligen Programms klar, dass sich das Finden einer optimalen Lösung nur dann auszahlt, wenn die Laufzeiten der Jobs deutlich größer als die Zeit zum Finden der Lösung ist.

Ähnliches gilt für den pseudopolynomiellen Algorithmus aus Unterabschnitt 3.2 für das ($\mathbf{Pm|spdp\text{-}any,pmtn|C_{max}}$)-Problem, denn dieser löst ja pseudopolynomiell viele lineare Programme, um die optimale Zuteilung zu finden. Interessant könnte der Algorithmus hingegen sein, wenn die gleichen Jobs wieder und wieder (mit gleichen Laufzeiten, aber ggfs. unterschiedlichen Eingaben) ausgeführt werden sollen. Dies kann beispielsweise in numerischen oder Grafikanwendungen sowie in Echtzeitanwendungen gegeben sein. In diesen Fällen kann es durchaus von Interesse sein, die Dauer für die Abbarbeitung eines wiederkehrenden „Mehrprozessor-Jobbündels“ zu minimieren. Das vorgestellte Approximationsschema bietet hierbei eine Möglichkeit, definierte Einbußen in der Qualität der gefundenen Lösung zu machen, um sowohl Laufzeit als auch Speicherbedarf zum Berechnen einer Lösung zu verringern.

Ein großes Problem stellt jedoch das exponentielle Verhalten der Laufzeit und des Speicherbedarfs in der Anzahl m der verfügbaren Prozessoren dar. Dies zeigt sich deutlich in den Ergebnissen der Laufzeitmessung.

4.1 Problemgenerierung

Zur experimentellen Auswertung der implementierten Algorithmen benötigt man selbstverständlich Testeingaben. Die Eingabedaten für Zuteilungsprobleme sind unabhängig davon, ob zur Lösung des Problems Preemption erlaubt ist, oder nicht. Daher sind in diesem Abschnitt immer sowohl die preemptive als auch die nicht-preemptive Variante gemeint.
Um die Güte der Algorithmen zu beurteilen ist es nun sinnvoll, keine ausschließlich zufälligen Eingaben (Joblaufzeiten) zu verwenden, sondern sie so zu wählen, wie sie in realen Schedulingproblemen auftreten können.

Für ($\mathbf{Pm}|\mathbf{size_j}|\mathbf{C_{max}}$)-Probleme besteht die Eingabe jedoch lediglich aus den n Werten $\mathbf{size_j}$ sowie den n zugehörigen Joblaufzeiten t_j. Ein Zusammenhang zwischen diesen Werten lässt sich ohne zusätzliche Annahmen über zu erwartende reale Eingaben nur schwerlich konstruieren.
Interessanter sind die Eingaben zu ($\mathbf{Pm}|\mathbf{spdp\text{-}any}|\mathbf{C_{max}}$)-Problemen. Hierbei sind nämlich für jeden der n Jobs m Laufzeiten zu wählen. Es erscheint jedoch beispielsweise wenig plausibel, dass ein Job langsamer wird, wenn er auf mehr Prozessoren zugeteilt wird, denn er könnte ja auch einfach weniger Prozessoren verwenden, als ihm zur Verfügung stehen.
Eine Forderung, die man deswegen mit gutem Gewissen an die Laufzeiten t_j^k stellen kann, ist also *monotones Fallen in der Prozessorzahl k*.

Das amdahl'sche Gesetz (Amd67) bietet eine Möglichkeit, mit wenigen Parametern solche Testdaten zu generieren. Es basiert darauf, dass von einem Programm nur ein gewisser Anteil p parallel auf allen zur Verfügung stehenden Prozessoren ausgeführt werden kann. Der restliche Teil $(1-p)$ kann nur sequentiell ausgeführt werden. Für jeden Job muss also der Anteil von parallelem Code p_j sowie die Ausführungsgeschwindigkeit auf einem Prozessor t_j^1 gewählt werden. Es ergibt sich die bekannte Laufzeit

$$t_j^k = t_j^1 \cdot \left((1-p_j) + \frac{p_j}{k}\right)$$

Nun gilt das amdahl'sche Gesetz als eine eher pessimistische Abschätzung. Dies liegt daran, dass der Speedup nach diesem Modell höchstens linear sein kann. Superlineare Speedups, die in der Praxis beispielsweise aufgrund von Cache-Effekten durchaus auftreten können, werden so außer acht gelassen.

Eine mögliche Verallgemeinerung wäre, Laufzeiten zu erzeugen, die konvex und monoton fallend in k sind. Anschaulich entspricht dies der Einschränkung, dass die Beschleunigung für die nächste Prozessorenzahl nicht

größer sein soll, als die letzte. Diskrete monoton fallende konvexe Funktionen lassen sich durch die absteigend sortierte Liste der Laufzeitdifferenzen $t_j^{k+1} - t_j^k$ angeben, die zufällig erzeugt werden kann.

Die erwähnten Cache-Effekte wirken sich in der Praxis allerdings eher als plötzlicher nicht-konvexer Sprung aus. Solche Sprünge ergeben sich daraus, dass ab einer bestimmten Prozessorzahl die immer kleiner werdenden zu lösenden Teilprobleme in die nächstkleinere Stufe der Cache-Hierarchie passen. So gelangt man schließlich zu stückweise konvexen Funktionen mit einer Anzahl von Sprungstellen, welche die Anzahl der Stufen der zu modellierenden Cache-Hierarchie nicht übersteigt.

Getestete Eingaben Es wurden nach zwei Verfahren Datensätze mit jeweils 3, 4 und 8 Prozessoren und maximalen Jobgrößen 10 sowie 15 generiert. Das eine beruht auf dem oben vorgestellten Amdahl'schen Gesetz (Datensatz „Amdahl"), bei dem zweiten wurde lediglich monotones Fallen der Joblaufzeiten in der Prozessorzahl gefordert (Datensatz „Monotone"). Für jede mögliche Kombination dieser Parameter wurden 10 verschiedene Datensätze erzeugt und – wenn möglich – optimal gelöst. Außerdem wurde eine 4- und eine 8-Approximation berechnet, d.h. das FPTAS mit $\varepsilon = 3$ bzw. 7 gestartet. Ebenfalls wurden 2-Approximationen betrachtet. Diese haben aber nur in sehr wenigen Fällen zu einer Reduktion des Problems geführt und sind daher nicht aufgeführt. Das kann vorkommen, wenn der berechnete Skalierungsfaktor $\alpha > 1$ ist, was bedeutet, dass die Laufzeit des FPTAS größer wäre, als die des optimalen Algorithmus. Man würde dann natürlich den optimalen Algorithmus zur Lösung verwenden. Traten solche Probleminstanzen bei der 4- oder 8-Approximation auf, so wurden sie daher als „nicht gelöst" betrachtet und für die Analyse verworfen, da sich der Approximationsalgorithmus weder in Laufzeit noch in Lösungsqualität vom optimalen unterscheiden würde. Falls einer der Approximationsalgorithmen für eine Jobanzahl keinen der Datensätze lösen könnte, wurde er nicht in die Betrachtung einbezogen und im Diagramm nicht aufgetragen. In den Diagrammen ist an der x-Achse in Klammern vermerkt, wie viele Testeingaben von den eingezeichneten Algorithmen für die jeweilige Jobanzahl gelöst werden konnten, falls dies nicht für alle 10 möglich war.

4.2 Laufzeit

Die Messungen in Abbildung 5 zeigen, dass schon bei geringer Prozessorenzahl optimale Lösungen nur schwer zu berechnen sind, denn nicht alle Datensätze konnten mit dem zur Verfügung stehenden Speicher gelöst werden.

Auch die Art der Eingabe scheint einen Einfluss auf die Laufzeit zu haben, denn die Eingaben aus dem Datensatz „Monotone“ konnten für mehr als drei Prozessoren nur in wenigen Fällen optimal gelöst werden. Wie erwartet lässt sich die benötigte Laufzeit pro Datensatz durch Approximation deutlich senken. Der Laufzeitgewinn durch die 8-Approximation ist allerdings deutlich größer als der durch die 4-Approximation.

Deutlich wird auch, dass der pseudopolynomielle Algorithmus exponentiell in der Prozessoranzahl m ist. Bereits für vier Prozessoren wird der Speicherverbrauch für die offenbar schwierigeren Eingaben aus dem „Monotone“-Datensatz schnell zu groß. Daher konnte nur noch die 8-Approximation mit vertretbarem Speicher- und Zeitaufwand gelöst werden.

Die beiden Greedy-Heuristiken laufen erwartungsgemäß deutlich schneller als die übrigen Algorithmen. Da sich die Laufzeiten kaum unterscheiden, ist in den Diagrammen nur die Greedy-Gain-Heuristik eingetragen.

4.3 Heuristische Verbesserung der Approximation

Wie in Unterabschnitt 3.3 bereits erwähnt, kann die Qualität der Approximationslösung durch Ausführen des polyonmiellen Algorithmus zur optimalen Lösung von ($\mathbf{Pm|size_j,\ pmtn|C_{max}}$)-Problemen deutlich gesteigert werden. Dies trifft sowohl auf die Testfälle des Amdahl-, als auch auf die des Monotone-Datensatzes zu. Auf diese Weise wird der entstehende Verschnitt durch Rundung vollständig eleminiert, und lediglich die nicht-optimale Prozessorzuteilung beeinflusst nun noch die Lösungsqualität. Wie in Abbildung 6 zu sehen ist, liefern die so verbesserten Approximationsalgorithmen für größere Eingaben dann Lösungen, die sehr nah an der optimalen liegen.

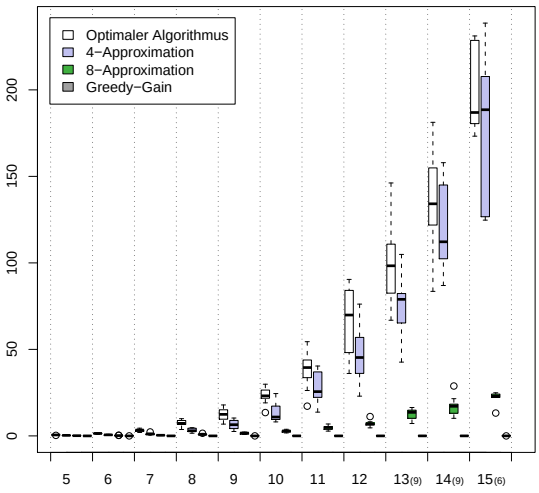
4.4 Ausgabequalität

Es stellt sich natürlich die Frage, wie stark die Qualität der Näherungslösungen von der optimalen abweicht. Daher zeigt Abbildung Abbildung 7, um welchen Faktor sich die Zuteilungslängen der Approximationsalgorithmen von der optimalen unterscheiden.

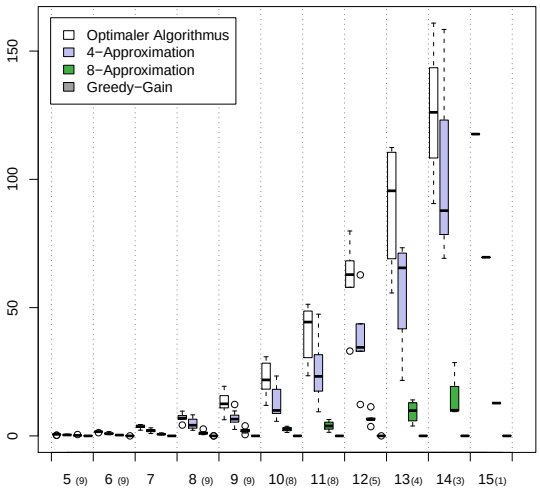
Um die Qualität der Approximation auch in denjenigen Fällen einschätzen zu können, in denen der pseudopolynomielle Algorithmus eine optimale Lösung aufgrund des hohen Speicherverbrauchs nicht berechnen konnte, wurde für Fälle mit weniger als 10 Jobs mit Hilfe eines Exponentialzeitalgorithmus, dessen Speicherverbrauch geringer als der des pseudopolynomiellen Algorithmus ist, eine optimale Lösung berechnet.

Für die getesteten Eingaben lässt sich in Abbildung 7 (a) und (b) erkennen, dass die erreichte Approximationsgüte deutlich besser ist, als die gegebene Garantie.
In den abgebildeten Fällen (c) und (d) konnte die optimale Lösung für mehr als 11 bzw. 9 Jobs in keinem der Testfälle mehr berechnet werden, weshalb die Approximationsgüte nicht angegeben werden kann.

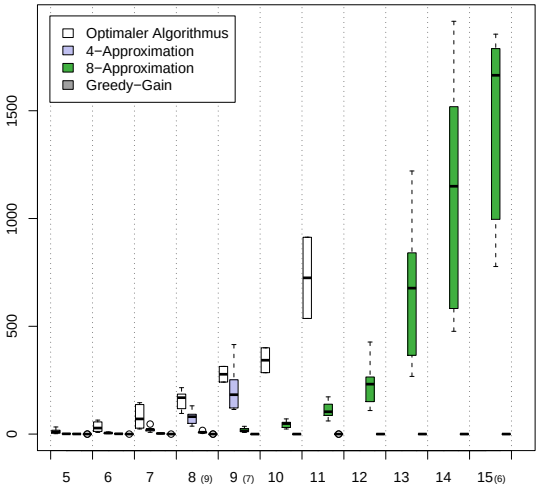
Für die Amdahl-Datensätze, dessen Jobs wie erwähnt nur linearen Speedup haben können, finden beide Greedy-Heuristiken vor allem für größere Eingabedaten zuverlässig die optimale Lösung. Dies liegt daran, dass es in den meisten Fällen am besten ist, jedem Job einen einzelnen Prozessor zuzuteilen, was die Heuristiken bei nicht-superlinearem Speedup tun.
Auch die monoton fallenden Eingabedatensätze werden mit ansteigender Problemgröße zunehmend besser gelöst. Die Approximationsalgorithmen scheinen jedoch ein regelmäßigeres Verhalten in der Qualität der gefundenen Zuteilungen zu zeigen.



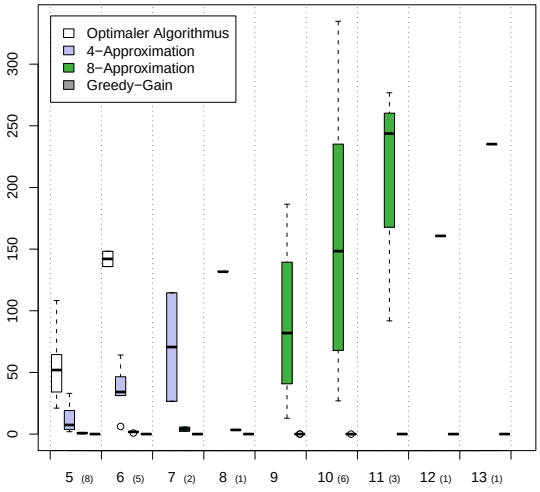
(a) „Amdahl“, 3 Prozessoren, Jobgröße ≤ 15



(b) „Monotone“, 3 Prozessoren, Jobgröße ≤ 15

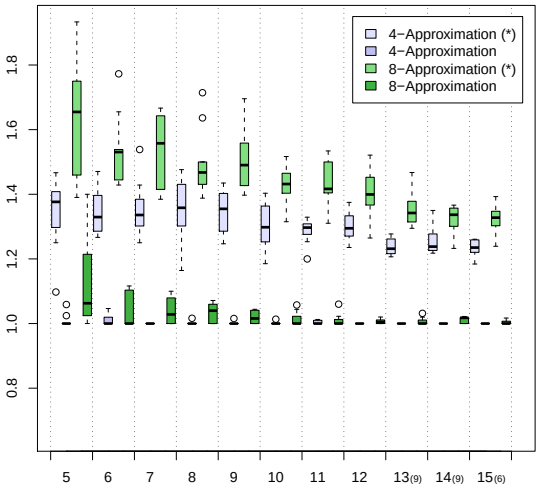


(c) „Amdahl“, 4 Prozessoren, Jobgröße ≤ 10

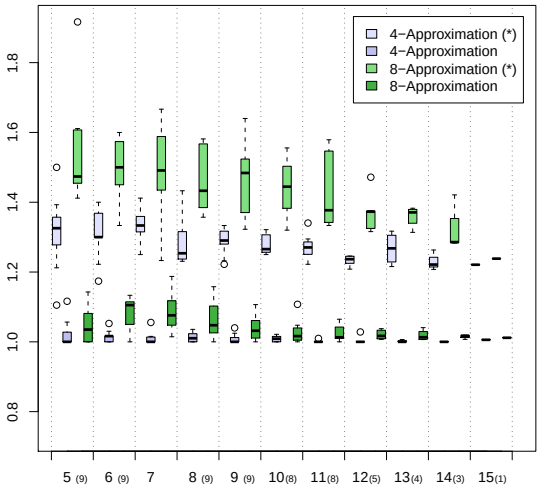


(d) „Monotone“, 4 Prozessoren, Jobgröße ≤ 10

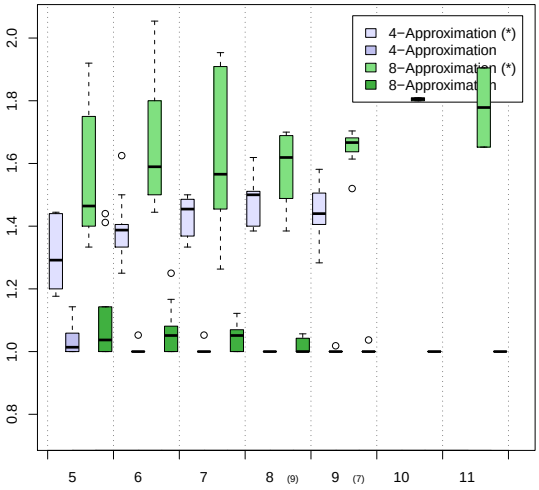
Abbildung 5: Beispielhafte Ergebnisse der Laufzeitmessung. Die horizontale Achse zeigt die Anzahl der Jobs sowie die Größe des gelösten Testdatensatzes. Die vertikale zeigt die Laufzeit der Algorithmen in Sekunden.



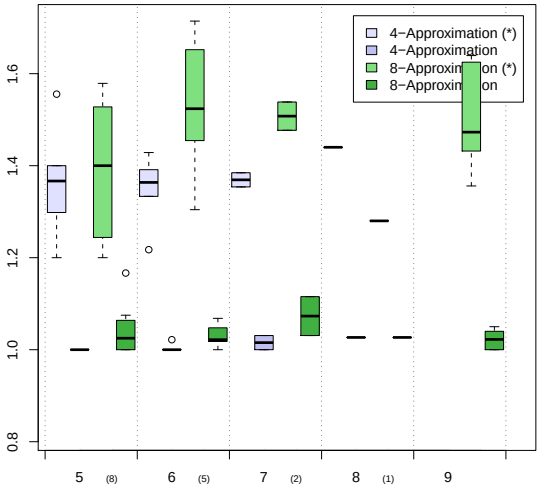
(a) „Amdahl“, 3 Prozessoren, Jobgröße ≤ 15



(b) „Monotone“, 3 Prozessoren, Jobgröße ≤ 15

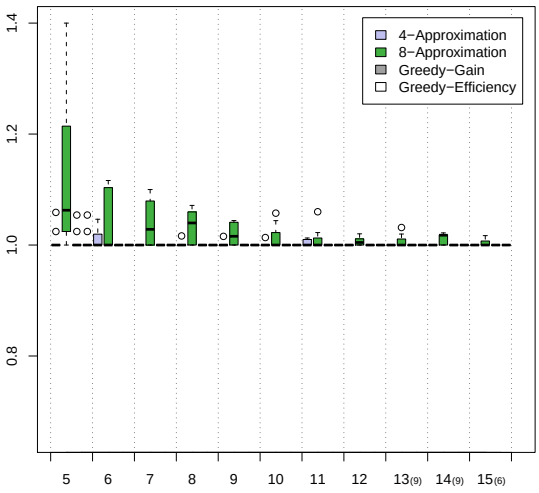


(c) „Amdahl“, 4 Prozessoren, Jobgröße ≤ 10

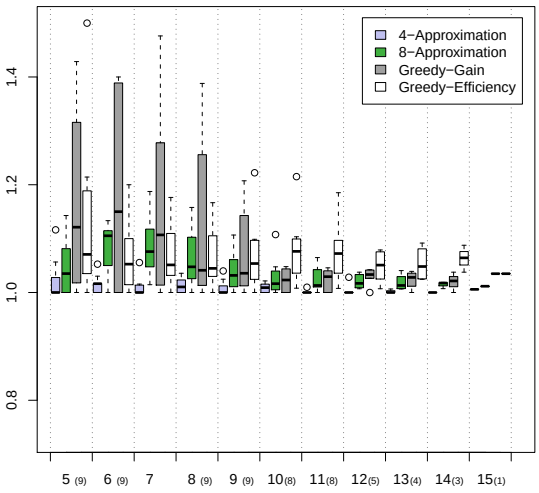


(d) „Monotone“, 4 Prozessoren, Jobgröße ≤ 10

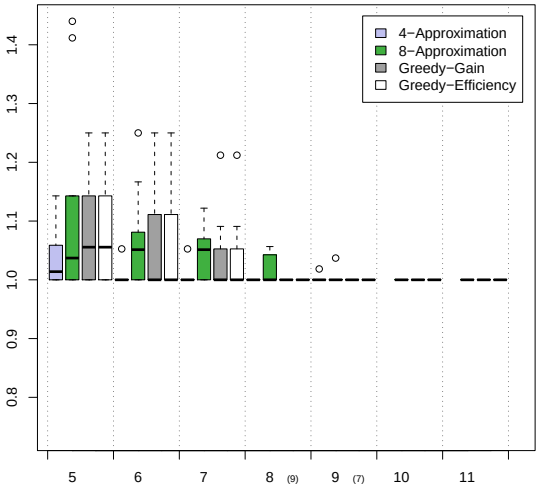
Abbildung 6: Heuristische Verbesserung der Approximationsergebnisse. Die unverbesserte Variante des Algorithmus ist mit (*) markiert.



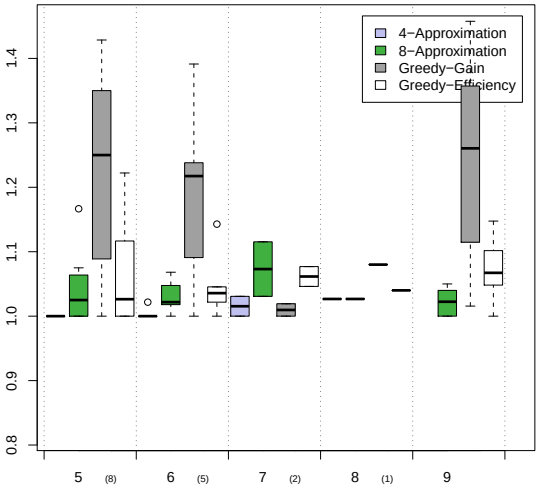
(a) „Amdahl“, 3 Prozessoren, Jobgröße ≤ 15



(b) „Monotone“, 3 Prozessoren, Jobgröße ≤ 15



(c) „Amdahl“, 4 Prozessoren, Jobgröße ≤ 10



(d) „Monotone“, 4 Prozessoren, Jobgröße ≤ 10

Abbildung 7: Beispielhafte Ergebnisse der Qualitätsmessung. Auch hier zeigt die x-Achse die Anzahl der Jobs und in Klammern die Größe des gelösten Testdatensatzes. Vertikal ist die erreichte Approximationsgüte aufgetragen.

Literatur

Amd67 Amdahl, Gene M.: Validity of the single processor approach to achieving large scale computing capabilities. In: *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*. New York, NY, USA : ACM, 1967, S. 483–485

BDW86 Blazewicz, J. ; Drabowski, M. ; Weglarz, J.: Scheduling Multiprocessor Tasks to Minimize Schedule Length. In: *Computers, IEEE Transactions on* C-35 (1986), May, Nr. 5, S. 389–393. http://dx.doi.org/10.1109/TC.1986.1676781. – DOI 10.1109/TC.1986.1676781. – ISSN 0018–9340

DL89 Du, Jianzhong ; Leung, Joseph Y.-T.: Complexity of Scheduling Parallel Task Systems. In: *SIAM J. Discrete Math.* 2 (1989), Nr. 4, 473-487. http://dblp.uni-trier.de/db/journals/siamdm/siamdm2.html#DuL89a

Dro96 Drozdowski, Maciej: Scheduling multiprocessor tasks – An overview. In: *European Journal of Operational Research* 94 (1996), Nr. 2, 215 - 230. http://dx.doi.org/10.1016/0377-2217(96)00123-3. – DOI 10.1016/0377–2217(96)00123–3. – ISSN 0377–2217

GLLK79 Graham, R.L. ; Lawler, E.L. ; Lenstra, J.K. ; Kan, A.H.G.Rinnooy: Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey. Version: 1979. http://dx.doi.org/10.1016/S0167-5060(08)70356-X. In: P.L. Hammer, E.L. J. (Hrsg.) ; Korte, B.H. (Hrsg.): *Discrete Optimization II, Proceedings of the Advanced Research Institute on Discrete Optimization and Systems Applications of the Systems Science Panel of NATO and of the Discrete Optimization Symposium co-sponsored by IBM Canada and SIAM Banff, Aha. and Vancouver* Bd. 5. Elsevier, 1979. – DOI 10.1016/S0167–5060(08)70356–X. – ISSN 0167–5060, 287 - 326

Zus08 *Rechteck-Anordnungen im Streifen*. Vieweg+Teubner, 2008 . – 115–156 S. – ISBN 978–3–8351–0215–6 (Print) 978–3–8351–9237–9 (Online)