Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

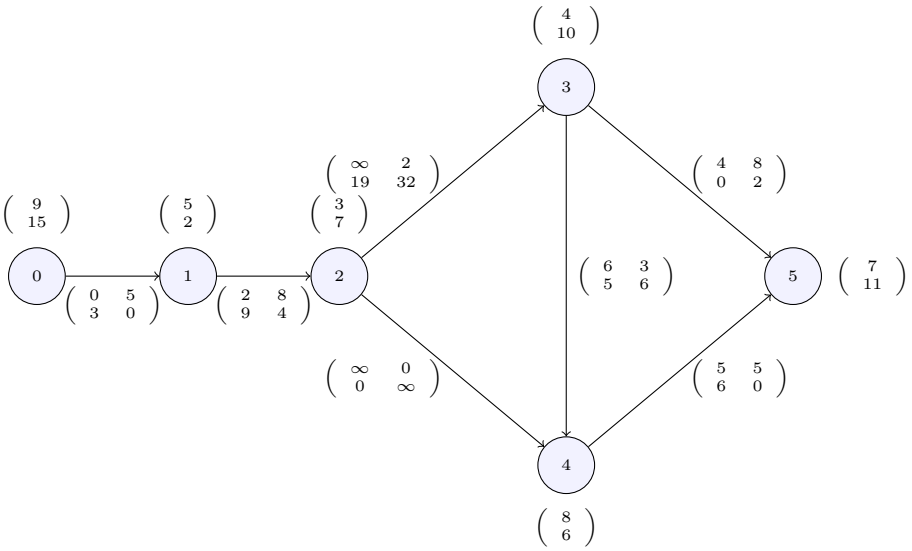
Lehrstuhl Prof. Dr.-Ing. Snelting

Entwicklung eines parallelen PBQP-Lösers mit X10

Studienarbeit von

Christopher Frieler

an der Fakultät für Informatik



Gutachter: Prof. Dr.-Ing. Gregor Snelting

Betreuender Mitarbeiter: Dipl.-Inform. Sebastian Buchwald

Bearbeitungszeit: 24. Februar 2012 – 5. April 2012

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 5. April 2012

Inhaltsverzeichnis

1 Einleitung

In der Vergangenheit wurde die Leistungsfähigkeit von Computern lange Zeit durch eine höhere Integrationsdichte der Transistoren auf einem Chip – laut dem Mooreschen Gesetz findet eine Verdoppelung etwa alle 18 Monate statt – und höhere Taktraten der Prozessoren gesteigert. Diese Technik ist in den letzten Jahren aber an ihre Grenzen gestoßen. Stattdessen wird zunehmend auf Mehrkernprozessoren und Mehrprozessorsysteme gesetzt, die mehrere Berechnungsfäden gleichzeitig anstatt einen besonders schnell ausführen können. Um diesen Vorteil nutzen zu können, sind die Software-Entwickler gezwungen ihre Software entsprechend anzupassen, sodass auch wirklich unabhängige und damit parallel ausführbare Berechnungsfäden vorhanden sind.

Im Rahmen dieser Arbeit wird ein paralleler Löser für das Partitioned Boolean Quadratic Problem (PBQP) entwickelt. Das PBQP ist ein mathematisches Optimierungsproblem. Es lässt sich unter anderem im Compilerbau für die Befehlsauswahl und die Registerallokation anwenden. Beispielsweise wurde es in [5] erfolgreich für die Befehlsauswahl für eingebettete Systeme genutzt. Die Formulierung als PBQP bietet dabei bessere Möglichkeiten mit irregulären Befehlssätzen umzugehen als klassische Techniken wie Termersetzungsverfahren, die etwa in [9] zur Entwicklung des Bottom-Up Pattern Matching genutzt wurden. In [6] wurde das PBQP auch für die Registerallokation eingesetzt. Hier bietet die Formulierung dieses Problems als PBQP ebenfalls bessere Möglichkeiten mit einem uneinheitlichen Registersatz umzugehen als die klassische Graphfärbung nach Chaitin [2].

Das PBQP ist zwar NP-vollständig, eine Unterklasse kann jedoch mit linearem Zeitaufwand gelöst werden. Darüber hinaus existieren gute Heuristiken um auch in allgemeinen PBQPs effizient Lösungen zu suchen. Die vorhandenen Lösungsstrategien für das PBQP versprechen ein großes Parallelisierungspotenzial. Ausgehend von den vorhandenen sequentiellen Algorithmen werden im Rahmen dieser Arbeit verschiedene Parallelisierungen entwickelt, implementiert und evaluiert.

Für die Implementierung wurde die von IBM Research entwickelte, quelloffene Programmiersprache X10 gewählt. Eine Übersicht über X10 findet sich in [3]. Aktuelle Entwicklungen zu X10 können auf der X10-Website ([10]) verfolgt werden. X10 ist eine typsichere, objektorientierte Programmiersprache. Darüber hinaus bietet X10 aber auch spezielle Unterstützung für Mehrfädigkeit von Anwendungen. Dazu stellt X10 Aktivitäten zur Verfügung. Aktivitäten sind sequentielle Berechnungsaufträge, die von einem Pool mehrerer Worker-Threads abgearbeitet werden. Um Parallelität zu erreichen, kann eine Aktivität mehrere neue Aktivitäten erzeugen, die dann von den Worker-Threads parallel bearbeitet werden können. Die Aktivitäten teilen einen gemeinsamen Speicher. Zusätzlich unterstützt X10 mit dem Konzept der „Places“ auch die Verteilung einer Anwendung auf mehrere Berechnungsknoten. Diese Places besitzen jeder einen eigenen Speicher, auf den sie effizient zugreifen können. Der Zugriff auf andere Places ist zwar auch möglich, stellt aber einen ineffizienten Speicherzugriff auf einen entfernten Speicher dar.

Die Entwicklung mit X10 bietet ebenfalls die Möglichkeit, den PBQP-Löser später an das in [7] vorgestellte Framework zur invasiven Programmierung anzubinden und so die Parallelisierungen auch im Rahmen dieses Programmierparadigmas zu untersuchen. Die invasive Programmierung erlaubt es einer Anwendung, ihre Berechnungen ausgehend vom erreichten Grad an Parallelität auf die vorhandene Hardware zu verteilen. Gleichzeitig wird der Anwendung auch ermöglicht, ihr Verhalten zur Laufzeit an die tatsächlich verfügbaren Hardwareressourcen anzupassen und dabei auch Schwankungen der Verfügbarkeit aufgrund von Auslastung, Temperatur oder sogar Defekten zu berücksichtigen.

In Kapitel 2 werden zunächst die theoretischen Grundlagen des PBQP erläutert. Kapitel 3 beschreibt dann einen sequentiellen PBQP-Löser. In Kapitel 4 werden die entwickelten Parallelisierungen vorgestellt und in Kapitel 5 bewertet. Kapitel 6 gibt einen Ausblick auf mögliche Fortführungen dieser Arbeit. Abschließend fasst Kapitel 7 die Ergebnisse der Arbeit noch einmal zusammen.

2 Grundlagen

2.1 Das Partitioned Boolean Quadratic Problem

Das Partitioned Boolean Quadratic Problem (PBQP) ist ein kombinatorisches Optimierungsproblem. Es ist formal definiert als

$$\begin{aligned} min \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} \vec{x}_i^\top \tilde{C}_{ij}\vec{x}_j & \\ s.t. \quad & \vec{x}_i^\top \vec{1} = 1 & \forall i = 1 \dots n \\ & \vec{x}_i \in \{0,1\}^{d_i} & \forall i = 1 \dots n \end{aligned}$$

Dabei sind die Entscheidungsvariablen $\vec{x}_i$ binäre Vektoren, in denen jeweils genau ein Eintrag 1 ist. Die Entscheidung besteht also darin, welcher Eintrag in jedem $\vec{x}_i$ 1 ist. Die $\tilde{C}_{ij}$ sind $(d_i \times d_j)$-Kostenmatrizen. Sie geben an, welche Kosten durch bestimmte einzelne Entscheidungen $(i = j)$ und für Kombinationen aus zwei Entscheidungen $(i \neq j)$ entstehen. Die Gesamtkosten sind zu minimieren.

PBQP ist NP-vollständig, wie bereits in [11] gezeigt wurde. Allerdings kann eine Unterklasse von PBQP in polynomieller Zeit gelöst werden. Außerdem kann mit den weiter unten vorgestellten Reduktionen R1, R2 und der heuristischen Variante von RN eine Näherungslösung gesucht werden, sodass der Aufwand linear in der Anzahl der Entscheidungsvariablen ist. Die gefundene Näherung ist meistens recht gut (vgl. [5]), aber eben nicht zwangsweise die optimale Lösung. Wenn in den Kostenmatrizen auch ∞ als Eintrag zugelassen ist, kann es auch vorkommen, dass überhaupt keine endliche Lösung gefunden wird, obwohl eine existiert.

2.2 Graphrepräsentation

Für diese Arbeit wurde eine leicht andere Formulierung des PBQP gewählt, wie sie auch in [12] verwendet wird. Dabei werden die Kostenmatrizen $\tilde{C}_{ii}$ durch Kostenvektoren $\vec{c}_i$, die nur aus den Diagonaleinträgen der Matrix bestehen, ersetzt. Andere Einträge dieser Matrizen gehen nie in den Zielfunktionswert ein. Außerdem werden für $i < j$ je zwei Matrizen $\tilde{C}_{ij}$ und $\tilde{C}_{ji}$ zu $C_{ij} := \tilde{C}_{ij} + \tilde{C}_{ji}^\top$ zusammengefasst:

$$\begin{aligned} min \quad & \sum_{1\leq i<j\leq n} \vec{x}_i^\top C_{ij}\vec{x}_j + \sum_{1\leq i\leq n} \vec{x}_i^\top \vec{c}_i & \\ s.t. \quad & \vec{x}_i^\top \vec{1} = 1 & \forall i = 1 \dots n \\ & \vec{x}_i \in \{0,1\}^{d_i} & \forall i = 1 \dots n \end{aligned}$$

Diese Form lässt sich als gerichteten Graphen repräsentieren. Der Graph besitzt einen Knoten pro Entscheidungsvariable $\vec{x}_i$, der mit dem zugehörigen Kostenvektor $\vec{c}_i$ markiert ist. Je zwei Knoten zu $\vec{x}_i$ und $\vec{x}_j$ sind durch eine Kante (i, j), die mit der Kostenmatrix C_{ij} markiert ist, verbunden. Man kann sich das PBQP dann so vorstellen, dass in jedem Knoten i eine von d_i Alternativen gewählt werden muss.

Jede Alternative erzeugt Kosten gemäß des Kostenvektors $\vec{c}_i$. Außerdem fallen für jede Kombination aus zwei Entscheidungen noch Kosten gemäß der Kostenmatrix C_{ij} an, wobei durch die Auswahl im Quellknoten der Kante eine Zeile und durch die Auswahl im Zielknoten eine Spalte der Matrix gewählt wird.

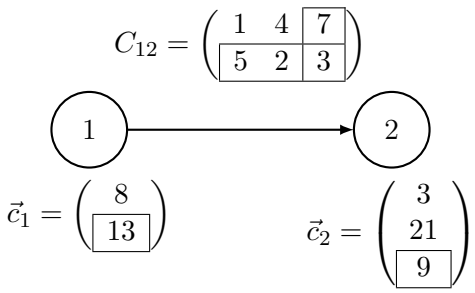


Abbildung 1: Ein kleiner PBQP-Graph mit einer eingezeichneten Auswahl.

In Abbildung 1 ist ein kleiner PBQP-Graph dargestellt. Es wurde in Knoten 1 die zweite Alternative ($\vec{x}_1 = (0\ 1)^\top$) und in Knoten 2 die dritte Alternative ($\vec{x}_2 = (0\ 0\ 1)^\top$) gewählt. Dadurch wird auch der Eintrag in der zweiten Zeile und dritten Spalte von C_{12} gewählt. Die entstehenden Kosten sind also $13 + 9 + 3 = 25$.

2.3 Normalform

Jede Matrix C_{ij} kann normalisiert werden, sodass in jeder Zeile und jeder Spalte jeder Kostenmatrix das Minimum der Einträge gleich 0 ist. Dazu wird in jeder Zeile der minimale Eintrag von jedem Eintrag subtrahiert und zum entsprechenden Eintrag von $\vec{c}_i$ hinzu addiert. Dadurch ist in jeder Kostenmatrix in jeder Zeile mindestens ein Eintrag gleich 0. Analog wird für die Spalten von C_{ij} und den Vektor $\vec{c}_j$ vorgegangen. Die so entstehende Form des PBQP bezeichnen wir als Normalform. An den Gesamtkosten jeder Auswahl ändert sich durch die Normalisierung nichts. Normalisiert man die Matrix aus dem vorhergehenden Beispiel, ergibt das den Graph aus Abbildung 2. Die eingezeichnete Auswahl kostet immer noch $15 + 10 + 0 = 25$.

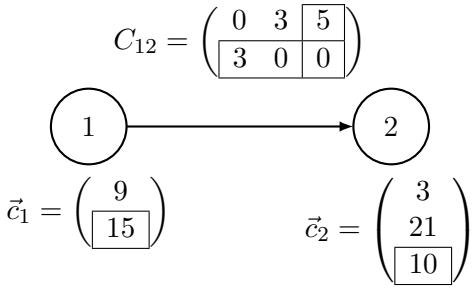


Abbildung 2: PBQP-Graph nach der Normalisierung.

Kostenmatrizen der Form

$$C_{ij} = \begin{pmatrix} u_1 + v_1 & \cdots & u_{d_j} + v_1 \\ \vdots & \ddots & \vdots \\ u_1 + v_{d_i} & \cdots & u_{d_j} + v_{d_i} \end{pmatrix}$$

werden durch die Normalisierung vollständig zur Null-Matrix. Wir bezeichnen sie als unabhängige Matrizen, weil sie keinerlei Beziehung zwischen den Entscheidungsvariablen $\vec{x}_i$ und $\vec{x}_j$ herstellen. Stellt sich eine Matrix während der Normalisierung als unabhängig heraus, bezeichnen wir auch die zugehörige Kante in der Graphrepräsentation als unabhängig; unabhängige Kanten können gelöscht werden.

2.4 Lösen von PBQP-Instanzen

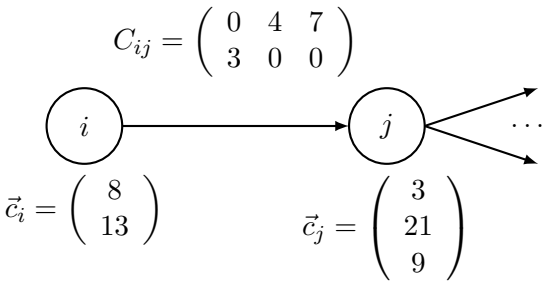
Zum Lösen einer PBQP-Instanz werden zunächst alle Kanten normalisiert, um alle unabhängigen Kanten zu identifizieren und zu löschen. Danach wird der Graph durch eine Folge von Reduktionen, die jeweils einen Knoten und seine inzidenten Kanten aus dem Graph entfernen, sukzessive verkleinert, bis nur noch einzelne Knoten vom Grad 0 vorhanden sind. Die Lösung der PBQP-Instanz ist dann die Lösung auf diesen restlichen Knoten. Diese Lösung ist trivial, da alle Knoten unabhängig voneinander, lokal optimiert werden können. Für gewöhnlich ist zusätzlich zum Lösungswert auch die Belegung der Entscheidungsvariablen $\vec{x}_i$ in jedem Knoten interessant. Für die übrig gebliebenen Knoten vom Grad 0 sind diese Belegungen bereits bekannt. Für die Knoten, die durch Reduktionen weggefallen sind, müssen die Belegungen noch ermittelt werden. Dazu werden die Reduktionen dieser Knoten in einer abschließenden Rückwärtspropagation nach und nach wieder rückgängig gemacht. Ein Knoten kann dabei bearbeitet werden, sobald die Belegungen seiner Nachbarknoten vom Zeitpunkt der Reduktion bekannt sind.

2.4.1 Die Reduktion R1

Die Reduktion R1 kann auf Knoten vom Grad 1 angewendet werden und entfernt diese aus dem Graph. Sei i ein solcher Knoten, j sein Nachbar und die Kante zwischen ihnen oBdA (i, j)[1]. Dann wird zuerst der Kostenvektor $\vec{c}_i$ zu jeder Spalte der Kostenmatrix C_{ij} addiert. Danach wird zu jeder Wahlmöglichkeit in Knoten j (also jedem Eintrag in dessen Kostenvektor) das Minimum der zugehörigen Spalte in der Kostenmatrix von C_{ij} ermittelt und zum entsprechenden Eintrag in $\vec{c}_j$ addiert. Damit sind die Kosten, die durch optimale Wahl des $\vec{x}_i$ zu jeder möglichen Wahl von $\vec{x}_j$ entstehen, in Knoten j berücksichtigt und Knoten i und Kante (i, j) können aus dem Graph entfernt werden.

[1] Im Fall $j < i$ wäre diese Kante (j, i), sie ließe sich aber ohne Probleme durch transponieren der Kostenmatrix in (i, j) überführen.

Hierzu ein Beispiel:



Wir wollen die R1-Reduktion auf i anwenden. Als erstes wird $\vec{c}_i$ auf jede Spalte von C_{ij} addiert:

$$C_{ij} = \begin{pmatrix} 8 & 12 & 15 \\ 16 & 13 & 13 \end{pmatrix}.$$

Die Minima in jeder Spalte sind 8, 12 und 13 und werden auf $\vec{c}_j$ addiert:

$$\vec{c}_j = \begin{pmatrix} 11 \\ 33 \\ 22 \end{pmatrix} \text{ und } C_{ij} = \begin{pmatrix} 0 & 0 & 2 \\ 8 & 1 & 0 \end{pmatrix}.$$

Es bleibt der Graph ohne Knoten i und Kante (i, j) zu lösen.

Die Rückwärtspropagation einer R1-Reduktion auf Knoten i muss durchgeführt werden, nachdem die Auswahl im Nachbarknoten j, also das $\vec{x}_j$ bekannt ist. Dann wird ein (nicht zwingend eindeutiger) minimaler Eintrag in der zur 1-Stelle in $\vec{x}_j$ gehörenden Spalte von C_{ij} gesucht. Sein Zeilenindex gibt die Auswahl in Knoten i, also die 1-Stelle in $\vec{x}_i$ an.

Angenommen wir haben im vorangegangenen Beispiel in Knoten j die dritte Alternative gewählt, d.h. $\vec{x}_j = (0\ 0\ 1)^\top$ ist die ermittelte Entscheidung in Knoten j. Dann schauen wir bei der Rückwärtspropagation in die dritte Spalte von C_{ij} und finden dort den minimalen Eintrag „0“ in der zweiten Zeile. Das bedeutet, dass in Knoten i die zweite Alternative gewählt wird, also dass $\vec{x}_i = (0\ 1)^\top$ bezüglich der Wahl in j optimal ist.

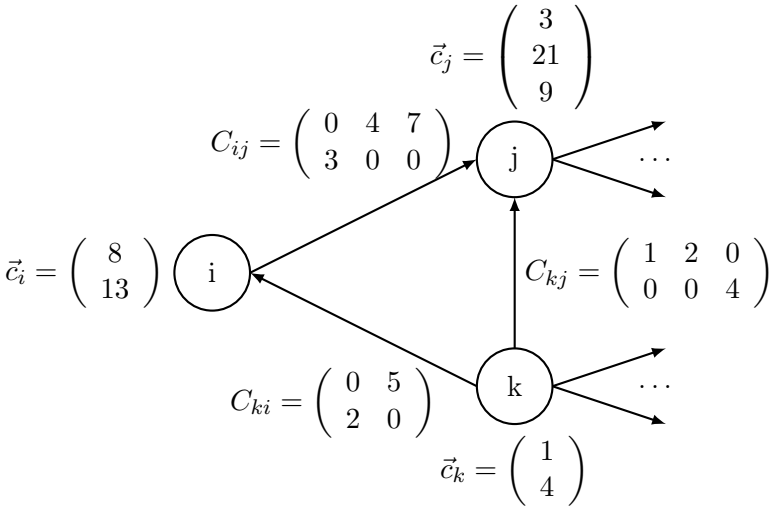
2.4.2 Die Reduktion R2

Die Reduktion R2 kann auf Knoten vom Grad 2 angewendet werden und entfernt diese aus dem Graph. Sei i ein solcher Knoten, j und k seien seine Nachbarn und die verbindenden Kanten seien (i, j) und (k, i)[2]. Dann wird zu jedem Paar möglicher

[2] Auch hier können die Kanten bei Bedarf durch Transponieren der Kostenmatrix umgedreht werden.

Entscheidungen in Knoten j und k die optimale Entscheidung in Knoten i bzw. die durch sie entstehenden Kosten berechnet. Dazu wird eine neue $(d_j \times d_k)$-Matrix aufgestellt. Deren Eintrag in der Zeile a und der Spalte b entspricht den Kosten, die durch die optimale Entscheidung in Knoten i entstehen, wenn in Knoten j und k die Möglichkeiten a bzw. b ausgewählt wurden. Um den Eintrag zu berechnen, müssen $\vec{c}_i$, die a-te Spalte von C_{ij} (Spalte, da j der Zielknoten der Kante (i, j) ist) und die b-te Zeile von C_{ki} (Zeile, da k der Quellknoten der Kante (k, i) ist) addiert und dann der minimale Eintrag ausgewählt werden. Die neue Matrix gibt Kosten an, die von der Kombination aus Entscheidungen in Knoten j und k abhängen. Sie bildet also eine neue Kante (j, k) oder, falls zwischen diesen Knoten schon eine Kante existiert, wird sie zu dieser – bei entgegengesetzter Richtung transponiert – addiert. Die neue oder geänderte Kante sollte dann wieder normalisiert werden, um sie, falls sie unabhängig ist, löschen zu können.

Auch hierzu ein Beispiel:



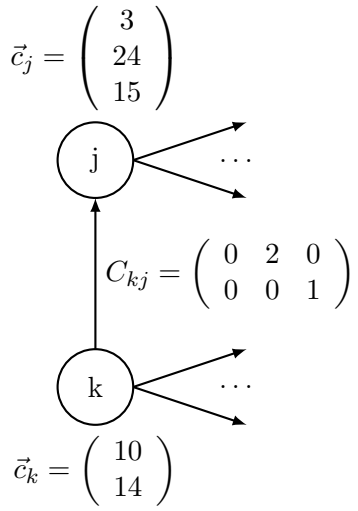
Wir wollen R2 auf Knoten i anwenden. Dazu stellen wir eine neue 3×2-Matrix auf. Um den ersten Eintrag zu ermitteln, addieren wir $\vec{c}_i$, die erste Spalte von C_{ij} und die erste Zeile von C_{ki}: $(8\ 21)^\top$. Der minimale Eintrag und damit der Matrixeintrag ist also 8. Analog berechnen wir die anderen Einträge und erhalten als neue Matrix

$$C_{temp} = \begin{pmatrix} 8 & 10 \\ 12 & 13 \\ 15 & 13 \end{pmatrix}.$$

Diese Matrix muss nun zur vorhandenen Kante zwischen j und k addiert werden. Da die vorhandene Kante allerdings von k nach j zeigt, wird die Matrix vorher transponiert. Wir erhalten als neue Matrix

$$C_{kj} = \begin{pmatrix} 9 & 14 & 15 \\ 10 & 13 & 17 \end{pmatrix}.$$

Knoten i und seine inzidenten Kanten können gelöscht werden. Die geänderte Kante muss noch normalisiert werden und als Ergebnis bleibt:



Um die optimale Entscheidung für Knoten i zu ermitteln müssen in der Rückwärtspropagation, nachdem die Entscheidungen in Knoten j und k bekannt sind, $\vec{c}_i$, die durch die 1-Stelle in $\vec{x}_j$ ausgewählte Spalte von C_{ij} und die durch die 1-Stelle in $\vec{x}_k$ ausgewählte Zeile von C_{ki} addiert werden. Die Position des minimalen Eintrags in diesem Vektor entspricht dann der Auswahl in Knoten i, also der 1-Stelle in $\vec{x}_i$.

In unserem Beispiel nehmen wir an, es seien in Knoten j die dritte Alternative, also $\vec{x}_j = (0\ 0\ 1)^\top$, und in Knoten k die erste Alternative, also $\vec{x}_k = (1\ 0)^\top$ gewählt worden. Wir addieren also die dritte Spalte von C_{ij} und die erste Zeile von C_{ki} zu $\vec{c}_i$ und erhalten $(15\ 18)^\top$. Der minimale Eintrag, die 15, steht an erster Stelle. Daher muss in Knoten i die erste Alternative gewählt werden, also $\vec{x}_i = (1\ 0)^\top$.

2.4.3 Die Reduktion RN

Existiert kein Knoten vom Grad 1 oder 2 mehr, können also R1 und R2 nicht angewendet werden, kommt die RN-Reduktion zum Einsatz. Dazu kann ein beliebiger Knoten gewählt werden. Alle seine inzidenten Kanten werden durch die RN-Reduktion aus dem Graph entfernt und für den Knoten selbst wird die Lösung festgelegt. Um möglichst viele Kanten aus dem Graph zu entfernen und so den Graph möglichst stark zu verkleinern, wird ein Knoten mit maximalem Grad gewählt. Für diesen kann dann heuristisch oder durch Ausprobieren aller Möglichkeiten eine Lösung bestimmt werden.

Als heuristische Variante wurde die Heuristik aus [5] implementiert, die dort recht gute Ergebnisse lieferte. Dabei wird der gewählte Knoten lokal, nur unter Berücksichtigung seiner direkten Nachbarschaft optimiert. Dazu wird jede Alternative a in diesem Knoten mit ihren lokal entstehenden Kosten bewertet. Für diese Bewertung

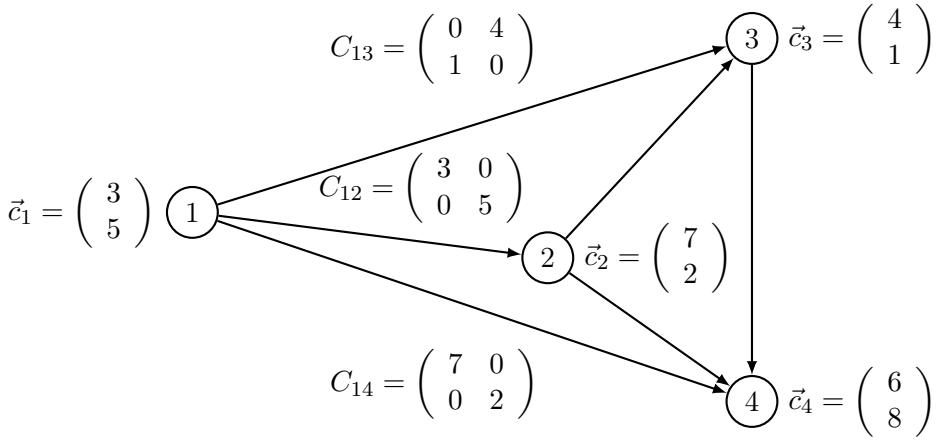
wird so getan, als ob alle Nachbarn des gewählten Knotens keine weiteren Nachbarn besäßen und dann werden virtuelle R1-Reduktionen für alle Nachbarn durchgeführt. Es werden also für jeden Nachbarknoten dessen Kostenvektor und die a-te Zeile oder Spalte der Kostenmatrix der verbindenden Kante – je nachdem, ob der gewählte Knoten Ziel- bzw. Quellknoten dieser Kante ist – addiert, der minimale Eintrag dieser Summe gewählt und zum a-ten Eintrag des Kostenvektors des gewählten Knoten addiert. Anschließend wird die am besten bewertete Alternative a^* ausgewählt, die a^*-te Zeile oder Spalte der Kostenmatrix jeder adjazenten Kante auf den Kostenvektor des zugehörigen Nachbarn addiert und die Kanten werden gelöscht. Dieses Verfahren liefert natürlich im Allgemeinen kein globales Optimum des PBQP mehr, führt aber zu linearem Zeitaufwand in der Anzahl der Knoten und erfahrungsgemäß guten Ergebnissen. Es ist außerdem zu beachten, dass die Wahl, welcher Knoten reduziert wird, das Ergebnis sowie die weiteren möglichen Reduktionen beeinflusst. Es ist daher nötig, die Wahl des Knotens auch bei mehreren Knoten mit maximalem Grad eindeutig zu machen, um verschiedene Implementierungen eines PBQP-Lösers vergleichen zu können.

Abbildung 3 zeigt hierzu wieder ein Beispiel. Dort soll Knoten 1 durch die RN-Reduktion gelöst werden. Dabei werden die Kanten (2, 3), (2, 4) und (3, 4) und ihre Kostenmatrizen ignoriert. Deshalb sind diese Matrizen im Beispiel bereits weggelassen. Jetzt werden für die Knoten 2, 3 und 4 R1-Reduktionen durchgeführt. Dabei entsteht eine Bewertung für die erste Alternative in Knoten 1, indem die erste Zeile von C_{12} und $\vec{c}_2$ addiert werden; der minimale Eintrag ist 2. Weiterhin werden die erste Zeile von C_{13} und $\vec{c}_3$ addiert; der minimale Eintrag des Ergebnisses ist 4. Außerdem werden die erste Zeile von C_{14} und $\vec{c}_4$ addiert; der minimale Eintrag dieses Ergebnisses ist 8. Diese errechneten Minima werden auf die Kosten für die Alternative 1 in Knoten 1 addiert und es ergibt sich eine Bewertung von $3 + 2 + 4 + 8 = 17$. Analog erhält Alternative 2 eine Bewertung von 19. Es wird also Alternative 1 ausgewählt, d.h. $\vec{x}_2 = (1\ 0)^\top$. Außerdem werden die erste Zeile von C_{12} auf $\vec{c}_2$, die erste Zeile von C_{13} auf $\vec{c}_3$ und die erste Zeile von C_{14} auf $\vec{c}_4$ addiert und die Kanten entfernt. Als Ergebnis der RN-Reduktion bleibt der in Abbildung 3 dargestellte PBQP-Graph.

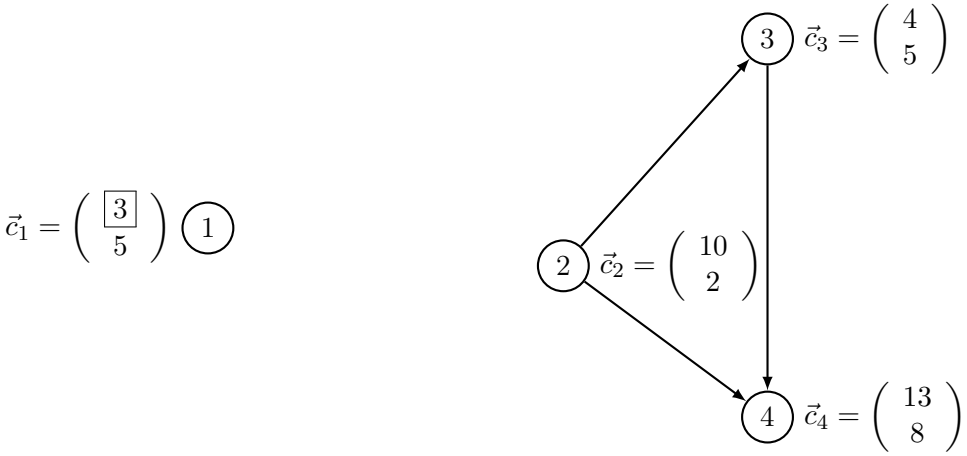
Alternativ zur Heuristik können auch alle Alternativen im RN-Knoten durchprobiert werden. Dazu werden einfach alle Alternativen im ausgewählten Knoten einmal ausgewählt, das Rest-PBQP gelöst und die beste Variante als Lösung genommen. Die Optimalität der Lösung ist damit natürlich gewährleistet, allerdings ist der Zeitaufwand exponentiell in der Anzahl der Knoten bzw. Entscheidungsvariablen, da in den Rest-PBQPs jeweils wieder Verzweigungen aufgrund von RN-Reduktionen vorkommen können und so ein recht großer Suchbaum entsteht.

Für dieses Probieren bietet es sich an, die Auswahl des RN-Knotens so anzupassen, dass bei gleichem Knotengrad der Knoten mit den wenigsten Auswahlmöglichkeiten genommen wird. Das kann den Suchbaum, der durch wiederholtes Verzweigen entsteht, deutlich verkleinern und damit das Verfahren beschleunigen.

Dieses Verfahren lässt sich durch eine Branch-And-Bound-Strategie wie in [6] beschrieben weiter verbessern, indem als untere Schranke die minimalen Einträge und als obere Schranke die maximalen Einträge aller Kostenvektoren und -matrizen sum-



(a) PBQP-Graph in dem nur noch RN-Reduktionen möglich sind.



(b) PBQP-Graph nach Anwendung der RN-Reduktion auf Knoten 1.

Abbildung 3: Beispiel für eine RN-Reduktion unter Verwendung der Heuristik.

miert werden. Als kleine Verbesserung kann bei Knoten vom Grad 0 auch der minimale Eintrag des Kostenvektors in die obere Schranke eingehen. Die Schranken müssen dabei nach jeder Reduktion angepasst werden. Noch deutlich bessere Ergebnisse erzielt man jedoch, indem man wie in [6] beschrieben zunächst eine heuristische Lösung ermittelt und diese als obere Schranke wählt. Auf diese Weise hat man von Anfang an eine recht scharfe obere Schranke, die auch während des Lösens nicht mehr angepasst werden muss.

3 Sequentieller PBQP-Löser

Für diese Arbeit wurde zunächst ein sequentieller PBQP-Löser mit X10 entwickelt. Dieser diente dann als Ausgangspunkt für die vorgenommenen Parallelisierungen und als Referenzversion für die Laufzeitmessungen. Der sequentielle Löser orientiert sich stark am „Karlsruher PBQP-Solver“ (kaps), der im Rahmen von [1] entstand, und beinhaltet daher die gleichen Phasen des Algorithmus.

Als erstes findet die Normalisierung des PBQP statt. Dabei wird über alle Kanten iteriert und für jede Kante die in Abschnitt 2.3 beschriebene Normalisierung durchführt. Stellt sich eine Kante dabei als unabhängig heraus, wird sie aus dem PBQP-Graph gelöscht.

Anschließend werden die Knoten als Vorbereitung für die nächsten Phasen abhängig von ihrem Grad in verschiedene Partitionen aufgeteilt. Es gibt je eine Partition für Knoten vom Grad 0, 1 und 2, sowie eine weitere für alle Knoten mit Grad 3 oder höher. Die Zugehörigkeit eines Knotens zu einer dieser Partitionen entspricht also der möglichen Reduktion für diesen Knoten. Um für eine RN-Reduktion effizient einen Knoten mit maximalem Grad und möglichst wenig Auswahlmöglichkeiten zu finden, ist die Partition für Knoten vom Grad 3 und höher als Max-Heap organisiert. Wird im Verlauf des Algorithmus ein Knoten reduziert, wird er aus seiner Partition entfernt und auf einen zusätzlichen Stack für die Rückwärtspropagation gelegt. Außerdem müssen die Nachbarn des reduzierten Knotens gegebenenfalls in eine andere Partition verschoben werden, falls sich ihr Grad geändert hat.

Jetzt werden nacheinander alle möglichen Reduktionen R1, R2 und RN durchgeführt, bis nur noch Knoten vom Grad 0 übrig sind. Dabei werden R1-Reduktionen gegenüber den anderen bevorzugt durchgeführt und R2-Reduktionen bevorzugt gegenüber RN-Reduktionen. Die reduzierten Knoten landen auf dem Stack für die Rückwärtspropagation.
Das Verhalten für die RN-Reduktion lässt sich durch ein Strategie-Objekt steuern, das eine der beiden Alternativen für das Verhalten bei einer RN-Reduktion auswählt. Es kann entweder die Heuristik angewendet werden, um eine Auswahl für den zu reduzierenden Knoten zu treffen oder es können auch alle Möglichkeiten dieses Knotens durchprobiert werden (vgl. Abschnitt 2.4.3). Bei letzterer Strategie wird für jede Möglichkeit eine Kopie des PBQP und des Lösers angelegt, die Möglichkeit wird eingesetzt und die Kopie wird gelöst. Abschließend wird die Möglichkeit gewählt, deren Kopie mit den geringsten Kosten gelöst wurde. Da beim Lösen der Kopien weitere RN-Entscheidungen getroffen worden sein könnten, werden zusätzlich zur besten Auswahl des einen zu reduzierenden Knoten auch alle weiteren RN-Entscheidungen, die zur optimalen Lösung geführt haben, gespeichert und später verwendet.

Nachdem alle Reduktionen durchgeführt sind, wird die Lösung bestimmt. Dazu wird in jedem der übrig gebliebenen Knoten vom Grad 0 die Alternative mit minimalen Kosten gewählt und diese Kosten werden zur Gesamtlösung addiert.

Als letzte Phase folgt noch die Rückwärtspropagation. Dazu wird der Stack der reduzierten Knoten abgearbeitet und für jeden Knoten wie in Abschnitt 2.4 beschrieben

die Lösung gewählt. Durch den Stack ist sichergestellt, dass die nötigen Nachbarknoten alle schon gelöst sind, weil die Knoten in der umgekehrten Reihenfolge, wie sie reduziert wurden, bearbeitet werden.

Optional kann das Branch-and-Bound-Verfahren aktiviert werden. Dann werden nach der Normalisierung die Schranken berechnet. Für die untere Schranke werden die minimalen Einträge aller Knoten und Kanten addiert. Zur Berechnung der oberen Schranke wird ein zusätzlicher Solver mit heuristischer RN-Strategie erzeugt, der eine Kopie des PBQP löst. Bei aktiviertem Branch-and-Bound-Verfahren wird zusätzlich bei jeder Reduktion die untere Schranke angepasst.

Wenn nach einer Reduktion die untere Schranke die obere übersteigt, wird der Lösungsvorgang abgebrochen. Es werden keine weiteren Reduktionen mehr durchgeführt. Als Gesamtlösung wird ∞ zurückgeliefert, um die bei den RN-Reduktionen getroffenen Entscheidungen als Irrweg zu markieren. Die gesamte Rückwärtspropagation entfällt.

Das UML-Aktivitätsdiagramm in Abbildung 4 veranschaulicht noch einmal den Algorithmus.

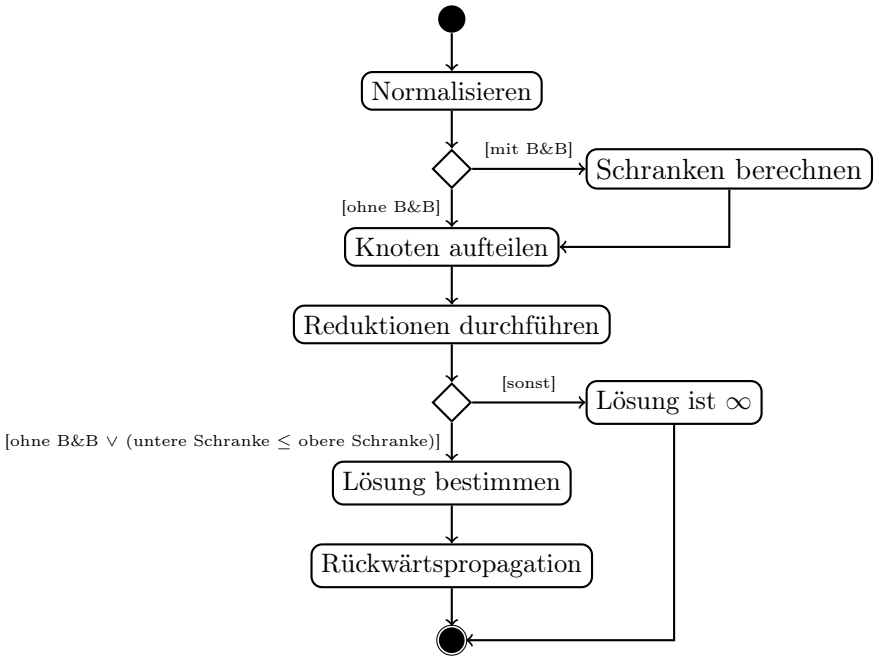


Abbildung 4: Der Ablauf des sequentiellen Lösungsalgorithmus für PBQP als UML-Aktivitätsdiagramm.

4 Parallelisierungen

Der Lösungsalgorithmus für PBQP bietet ein großes Parallelisierungspotenzial, da verschiedene Teile des Graphen unabhängig von einander bearbeitet werden können. Dazu muss allerdings erst eine Aufteilung des PBQP-Graphen berechnet werden. Der Berechnungsaufwand um eine Aufteilung zu erhalten sollte dabei möglichst klein sein, am besten sogar linear, sodass der Geschwindigkeitsvorteil, der durch die anschließende Parallelisierung entsteht, so wenig wie möglich beeinträchtigt wird. Im Folgenden werden fünf implementierte Parallelisierungen beschrieben, die unabhängig voneinander in den Algorithmus aufgenommen werden können.

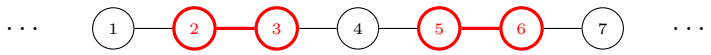
4.1 Normalisierung

Die Normalisierung einer Kante greift nur auf die Kostenmatrix der Kante und die Kostenvektoren ihrer beiden inzidenten Knoten zu. Je zwei Kanten können daher parallel normalisiert werden, wenn sie keinen inzidenten Knoten gemeinsam haben. Eine Menge von parallel normalisierbaren Kanten entspricht damit genau der Definition eines Matchings im PBQP-Graphen. Grundlagen zum Thema Matchings können z.B. in [8] nachgelesen werden. Die Normalisierung bietet damit ein großes Parallelisierungspotenzial.

Um die Normalisierung zu parallelisieren wird nun wiederholt ein Matching aus den noch nicht normalisierten Kanten aufgebaut und die Kanten des Matchings werden parallel normalisiert. Ideal wäre dabei natürlich ein optimales Matching, um größtmögliche Parallelität zu erreichen. Ein optimales Matching ist ein kardinalitätsmaximales Matching, d.h. es gibt kein Matching, das aus mehr Kanten besteht. Das finden eines optimalen Matchings ist allerdings zu aufwändig, in allgemeinen, ungewichteten Graphen $G = (V, E)$ liegt der Aufwand bei $O(|V|^2 \cdot |E|^2)$. Stattdessen lässt sich aber in $O(|E|)$ ein maximales Matching M finden, also eines, dem keine Kante mehr hinzugefügt werden kann, ohne eine andere zu entfernen oder die Matching-Eigenschaft zu zerstören. Insbesondere ist jedes optimale Matching auch maximal. Um ein maximales Matching zu finden kann einfach ein Greedy-Verfahren angewandt werden, das über alle Kanten iteriert und jede Kante, zu der noch keine adjazente Kante im Matching ist, dem Matching hinzufügt. Es lässt sich leicht zeigen, dass ein so gefundenes maximales Matching mindestens halb so viele Kanten wie ein optimales Matching enthält.

Beweis. *Sei M ein maximales Matching, M^* ein optimales. Dann existiert zu jeder Kante in M^* mindestens eine Kante in M, mit der sie einen inzidenten Knoten gemeinsam hat, denn sonst könnte sie einfach zu M hinzugenommen werden und M wäre im Widerspruch zur Voraussetzung nicht maximal. Da aber jede Kante in M nur zwei inzidente Knoten hat und somit nur die Hinzunahme von höchstens zwei Kanten aus M^* verhindern kann, muss M mindestens halb so viele Kanten enthalten wie M^*.* □

Um die Qualität des Matchings noch zu verbessern, wird eine Tiefensuche genutzt. Dazu wird über alle Knoten iteriert und an jedem Knoten, zu dem noch keine inzidente Kante im Matching ist, eine Tiefensuche gestartet. Diese Tiefensuche folgt nur Kanten, die noch nicht normalisiert wurden. Eine besuchte Kante wird zum Matching hinzugefügt, wenn sich für ihre inzidenten Knoten noch keine inzidenten Kanten im Matching befinden. Das verbessert insbesondere das Verhalten bei langen Ketten im Graph, da bei der Tiefensuche jede zweite Kante in das Matching aufgenommen wird. Bei zufälliger Betrachtung der Kanten könnte im schlimmsten Fall nur jede dritte aufgenommen werden (siehe Abbildung 5).



(a) Durch das Greedy-Verfahren erzeugtes, maximales Matching. Nur jede dritte Kante wurde ausgewählt.

(b) Durch Tiefensuche ab Knoten 1 erzeugtes, maximales Matching. Jede zweite Kante wurde ausgewählt.

Abbildung 5: Zwei unterschiedlich erzeugte, maximale Matchings (rot und fett dargestellt).

Eine bessere untere Schranke für die Größe des Matchings als 50% eines optimalen Matchings lässt sich dadurch aber leider nicht zusichern. Es lassen sich immer noch Graphen konstruieren, bei denen auch die Anwendung der Tiefensuche nur zu halb so großen Matchings wie einem optimalen führen können.

4.2 Zusammenhangskomponenten

Besteht der PBQP-Graph aus mehreren Zusammenhangskomponenten, so lassen sich diese vollkommen unabhängig und damit sehr gut parallel lösen. Die Zusammenhangskomponenten können mithilfe der Union-Find-Datenstruktur in $O(n \cdot \alpha(n))$ ermittelt werden, wobei α die inverse Ackermannfunktion ist. Ein einfacher Algorithmus zur Berechnung der Zusammenhangskomponenten eines Graphen findet sich z.B. in [4]. Danach werden die Zusammenhangskomponenten parallel gelöst. Triviale Zusammenhangskomponenten, die nur aus einem einzelnen Knoten vom Grad 0 bestehen, werden direkt gelöst, um übermäßigen Overhead zu vermeiden. Für alle nicht-trivialen Zusammenhangskomponenten wird der PBQP-Lösungsalgorithmus gestartet. Die Berechnung der Zusammenhangskomponenten wird erst nach der Normalisierung durchgeführt, um unabhängige Kanten vorher zu löschen und damit eventuell mehr Zusammenhangskomponenten zu erhalten.

Es wäre auch denkbar, nach R2- und RN-Reduktionen erneut eine Aufteilung vorzunehmen, wenn die Zusammenhangskomponente durch die Reduktion wieder in mehrere Komponenten zerfallen ist. Es lässt sich allerdings nur durch erneute Anwendung des Aufteilungsalgorithmus auf die gesamte Komponente prüfen, ob tatsächlich neue

Komponenten entstanden sind. Da dies aber im Allgemeinen nur selten passiert, wäre der Aufwand im Verhältnis zum Nutzen recht hoch. Auf eine Implementierung dieser erneuten Aufteilung wurde daher verzichtet.

4.3 Reduktionen und Rückwärtspropagation

Jede Reduktion betrifft nur den einen Knoten, der reduziert wird, sowie seine inzidenten Kanten und seine Nachbarknoten. Sind diese betroffenen Daten für zwei Reduktionen disjunkt, so können die Reduktionen parallel ausgeführt werden. Einzig beim Entfernen von Kanten oder dem reduzierten Knoten aus dem PBQP, sowie beim Anpassen der unteren Schranke, wenn das Branch-and-Bound-Verfahren benutzt wird, müssen sich die parallelen Aktivitäten synchronisieren. Damit ist auch eine Parallelisierung der Reduktionen innerhalb einer Zusammenhangskomponente möglich. Das ist besonders wichtig, wenn nur wenige oder sogar nur eine Zusammenhangskomponente vorhanden ist. Da eine RN-Reduktion immer erst durchgeführt wird, wenn keine R1- und R2-Reduktionen mehr möglich sind, und die Wahl des RN-Knotens bei Anwendung der Heuristik das Ergebnis beeinflusst, werden nur alle möglichen R1- und R2-Reduktionen parallelisiert. Eine Bevorzugung der R1- gegenüber den R2-Reduktionen gibt es hier nicht mehr. Danach folgt eine einzelne RN-Reduktion.

Um nun parallel durchführbare Reduktionen zu finden, wird die Knotenmenge partitioniert und die Partitionen werden parallel bearbeitet. Dabei dürfen Reduktionen nicht auf Randknoten, also Knoten, die eine inzidente Kante in eine andere Partition haben, durchgeführt werden, weil es sonst zu konkurrierenden Zugriffen aus mehreren Partitionen kommen könnte. Stattdessen sind Reduktionen nur auf inneren Knoten erlaubt, sodass ihre Nachbarn in der gleichen Partition liegen. Deshalb ist es wünschenswert eine derartige Partitionierung zu finden, dass die Randknoten möglichst hohen Grad haben und somit gar nicht für eine R1- oder R2-Reduktion in Frage kommen.

Ein Ansatz wäre die Kanten invers zum Grad ihrer inzidenten Knoten zu bewerten und dann einen minimalen Schnitt im Graphen zu berechnen. Eine mögliche Bewertungsfunktion wäre z.B. $w : E \to \mathbb{Q}, (i, j) \mapsto \frac{1}{deg(i)+deg(j)}$. Diese Funktion führt dazu, dass der minimale Schnitt aus wenig Kanten besteht, es also nur wenig Randknoten gibt, und diese Randknoten zusätzlich hohen Grad haben. Allerdings liefert diese Art der Partitionierung immer nur zwei Partitionen, die dann erneut aufgeteilt werden müssen. Ohnehin ist die Berechnung eines minimalen Schnittes sehr zeitaufwändig. Darüber hinaus wird sich der Grad vieler Knoten durch Reduktionen ihrer Nachbarn während des Verfahrens noch verringern, sodass Knoten mit ursprünglich hohem Grad nicht immer geeignete Randknoten sind. Es muss stattdessen abgeschätzt werden, welche Knoten und Kanten durch R1- und R2-Reduktionen wegfallen werden.

Um diesen Problemen zu begegnen wurde ein anderes Verfahren für die Partitionierung implementiert. Abbildung 6 zeigt ein Beispiel einer solchen Partitionierung und dient im Folgenden zur Veranschaulichung des Verfahrens.

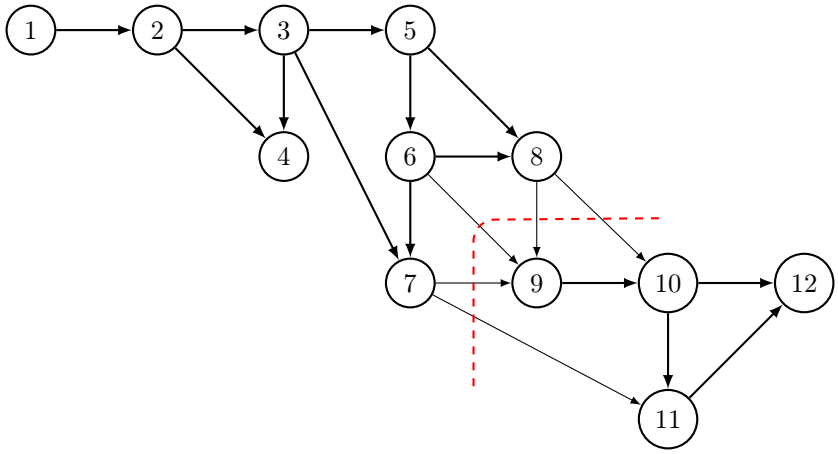


Abbildung 6: Ein zusammenhängender PBQP-Graphen, der an der gestrichelten Linie in zwei Partitionen unterteilt wurde.

Die Partitionen werden nacheinander aufgebaut. Es wird an einem Knoten vom Grad 1 oder 2 begonnen, der noch zu keiner Partition gehört. Dieser lässt sich in jedem Fall reduzieren, daher wird er zur Partition hinzugenommen. Knoten 1 im Beispiel könnte ein solcher Knoten sein. Von diesem Knoten aus startet dann eine Breitensuche, bei der nur von solchen Knoten aus weiter gesucht wird, die in die Partition aufgenommen wurden. Ein besuchter Knoten wird dabei zur Partition hinzugenommen, wenn sein Grad abzüglich der Anzahl seiner Nachbarn, die bereits in der selben Partition liegen, echt kleiner als 3 ist, er also nach Anwendung der Reduktionen auf schon vorhandenen Knoten in der Partition selbst reduziert werden kann. Im Beispiel würde also Knoten 2 besucht. Dessen Grad ist 3, die Kante zu Knoten 1 wird jedoch abgezogen, weil Knoten 1 bereits in der Partition ist. Das entspricht der Tatsache, dass Knoten 2 nach der Reduktion von Knoten 1 nur noch Grad 2 hat und dann reduziert werden kann. Nachdem Knoten 2 in die Partition aufgenommen wurde, werden Knoten 3 und 4 besucht. Knoten 3 kommt nicht in die Partition, da er Grad 4 und nur eine Kante in die Partition hat, nämlich die zu Knoten 2. Knoten 4 wird in die Partition aufgenommen und jetzt muss sein Nachbar Knoten 3 erneut besucht werden. Tatsächlich besitzt dieser jetzt zwei Kanten in die Partition und wird auch hinzugenommen.

Da durch die Anwendung der R2-Reduktion manchmal eine neue Kante hinzukommen kann, stellt dieses Verfahren eine leichte Überapproximation der reduzierbaren Knoten dar. Wird im Beispiel nach Knoten 3 der Knoten 5 besucht, wird dieser als reduzierbar angenommen, weil er nach dem Wegfall von Knoten 3 nur noch Grad 2 hat. Die R2-Reduktion von Knoten 3 kann aber zwischen Knoten 5 und 7 eine zusätzliche Kante erzeugen. Solche Fälle lassen sich jedoch nicht erkennen, bevor die Reduktionen durchgeführt wurden, schließlich könnte die entstandene Kante zwischen Knoten 5 und 7 auch unabhängig sein und gleich wieder wegfallen.

Um konkurrierende Zugriffe auf Knoten zu vermeiden, dürfen nur innere Knoten der Partitionen reduziert werden, sodass alle betroffenen Nachbarn in der gleichen Par-

tition liegen. Wenn die Partition nicht weiter ausgedehnt werden kann, wird deshalb noch versucht, eine „Grenzschicht“ um die Knoten herum in die Partition aufzunehmen, d.h. es werden noch alle Kanten, die aus der Partition herausführen, verfolgt und der erreichte Knoten wird ebenfalls in die Partition aufgenommen, wenn er sich noch in keiner anderen befindet. Nachdem also Knoten 5 hinzugenommen wurde, kann die Partition nicht weiter vergrößert werden. Die Knoten 6, 7 und 8, die von den Reduktionen der Knoten 3 und 5 betroffen sind, werden aber noch aufgenommen.

Gleichzeitig kann es dadurch aber auch passieren, dass eine R1- oder R2-Reduktion nicht durchgeführt werden darf, weil ein Nachbarknoten bereits in einer anderen Partition liegt. Diese Reduktion kann dann erst später durchgeführt werden. Im Beispiel passiert das, wenn ausgehend von Knoten 12 die nächste Partition aufgebaut wird. Die Knoten 11 und 10 werden als reduzierbar hinzugenommen. Die zusätzliche Grenzschicht würde aus den Knoten 7, 8 und 9 bestehen, Knoten 7 und 8 sind aber schon in der ersten Partition. Dadurch ist Knoten 11 kein innerer Knoten und darf vorerst nicht reduziert werden.

Nach diesem Schema werden nacheinander weitere Partitionen aufgebaut, bis alle Knoten vom Grad 1 und 2 in einer Partition liegen. Dann werden die Partitionen parallel abgearbeitet, indem alle in ihnen möglichen R1- und R2-Reduktionen durchgeführt werden. Es werden solange neue Partitionierungen gesucht und reduziert, bis nur noch RN-Reduktionen möglich sind. Dann wird eine RN-Reduktion durchgeführt und das Verfahren wiederholt sich, indem eine neue Partitionierung gesucht wird, bis schließlich alle Reduktionen durchgeführt und nur noch Knoten vom Grad 0 übrig sind.

Um Parallelität auch dann zu erzwingen, wenn eigentlich nur eine große Partition gefunden würde, die alle Knoten enthält, kann zusätzlich eine obere Schranke für die Größe einer Partition angegeben werden. Die Breitensuche bricht dann auch ab, sobald diese Schranke erreicht ist. Als Wert für diese Schranke bietet sich zum Beispiel die Anzahl der Knoten geteilt durch die Anzahl vorhandener Worker-Threads an, sofern diese verfügbar ist.

Die gefundenen Partitionierungen liefern auch gleichzeitig eine Möglichkeit, die Rückwärtspropagation zu parallelisieren. Es dürfen nämlich genau die Reduktionen parallel rückgängig gemacht werden, die auch parallel durchgeführt wurden. Das sind eben die, die bei der selben Partitionierung in unterschiedlichen Partitionen stattgefunden haben. Die Rückwärtspropagation verfolgt also die Partitionierungen in umgekehrter Reihenfolge. In jeder Partitionierung können die Partitionen parallel bearbeitet werden. Innerhalb jeder Partition müssen die Reduktionen dann wiederum in umgekehrter Reihenfolge rückgängig gemacht werden.

Die vorgestellte Art der Partitionierung respektiert auch Zusammenhangskomponenten, d.h., auch Zusammenhangskomponenten werden durch diese Partitionierung getrennt. Allerdings ist es nicht sinnvoll auf die vorherige Aufteilung der Zusammenhangskomponenten zu verzichten, weil das Lösen von Zusammenhangskomponenten vollkommen unabhängig und somit parallel geschehen kann. So werden im Gegensatz zur hier vorgestellten Partitionierung auch parallele RN-Reduktionen erlaubt.

4.4 Bestimmung der Lösung

Nachdem alle Reduktionen durchgeführt sind, müssen zur Bestimmung der Lösung die übrig gebliebenen Knoten vom Grad 0 gelöst werden, indem in jedem dieser Knoten die günstigste Alternative ausgewählt wird und die Kosten addiert werden. Auch dies lässt sich parallelisieren. Dazu könnten die vorhandenen Worker-Threads sich immer einen Knoten holen und diesen bearbeiten, bis alle Knoten gelöst sind. Dieses Verfahren führt zu einem guten Lastausgleich zwischen den Threads. Allerdings müssten sie sich beim Zugriff auf die Liste der Knoten synchronisieren. Geschickter ist es, die Liste der Knoten vom Grad 0 einfach in mehrere, gleich große Abschnitte aufzuteilen. Diese Abschnitte können dann parallel gelöst werden, ohne dass zusätzlicher Synchronisationsaufwand beim Zugriff auf die Liste entsteht. Ein Lastausgleich findet dabei natürlich nicht statt. Im Rahmen dieser Arbeit wurde die zweite Variante implementiert.

Zum Aufsummieren der Gesamtlösung müssen sich die parallelen Aktivitäten an der Summationsvariable synchronisieren. Auch dieses Problem lässt sich abmildern, indem jede parallele Aktivität ihre eigene Summationsvariable bekommt. Die Teilsummen müssen dann am Ende noch einmal addiert werden. Das kann wiederum parallel geschehen, indem die Teilsummen hierarchisch addiert werden, oder, wie im Rahmen dieser Arbeit implementiert, einfach sequentiell geschehen. Um festzulegen, wie viele Teilmengen es geben soll, lässt sich wieder die Anzahl der verfügbaren Worker-Threads heranziehen.

4.5 Parallelisierung der Reduktion RN

Wird als RN-Strategie das Durchprobieren aller Möglichkeiten des RN-Knotens gewählt, lässt sich auch dies sehr einfach parallelisieren. Da bereits für jede Variante, die probiert wird, eine Kopie des PBQP-Graphen und des Lösers erzeugt wurde, können die Löser problemlos parallel fortgesetzt werden.

5 Messung und Auswertung

Als Testplattform für die Messungen kam ein Notebook mit einem Intel Core i7-2630QM als Prozessor zum Einsatz. Dieser Prozessor verfügt über acht Kerne, vier davon sind allerdings nur virtuell durch Intels Hyperthreading-Technologie vorhanden. Die Kerne sind nominell mit 2,0 GHz getaktet; mittels Turbo-Boost können einzelne Kerne aber eine Geschwindigkeit von bis zu 2,9 GHz erreichen, wenn nicht alle Kerne aktiv sind. Dadurch kann auch bei theoretisch perfekter Parallelität nicht die achtfache Geschwindigkeit erreicht werden. Jeder der acht Kerne besitzt 32 KB L1-Cache, je zwei Kerne, ein echter und ein virtueller, teilen sich 256 KB L2-Cache und zusätzlich ist noch ein 6 MB großer L3-Cache vorhanden, der von allen Kernen gemeinsam genutzt wird. Das Notebook verfügte über 6 GB DDR3-Arbeitsspeicher und lief unter Ubuntu 11.10 (x64).

Auf diesem Testrechner wurden verschiedene Instanzen des PBQP gelöst. Die Testfälle wurden dabei zufallsgesteuert von einem Testfallgenerator erstellt. Dieser Generator erzeugt zunächst eine vorgegebene Anzahl Knoten. Jeder dieser Knoten besitzt zwischen einer und zehn Auswahlmöglichkeiten. Die Wahrscheinlichkeit für nur eine Möglichkeit beträgt dabei $\frac{1}{19}$, für zwei oder mehr Möglichkeiten beträgt sie jeweils $\frac{2}{19}$. Die Kosten jeder Möglichkeit sind auf dem Intervall von 100 bis 200 gleichverteilt. Dann wird eine vorgegebene Anzahl von Kanten generiert. Dazu werden gleichverteilt zwei unterschiedliche Knoten ausgewählt und mit einer Kante verbunden. Die Dimension der Kostenmatrix ist dabei durch die inzidenten Knoten festgelegt, die Einträge der Kostenmatrix sind wieder auf dem Intervall von 100 bis 200 gleichverteilt. Entsteht zwischen zwei Knoten mehr als eine Kante, so werden diese Kanten beim Aufbau des PBQP-Graphen zusammengefasst. Zusätzlich kann noch eine Mindestanzahl von Zusammenhangskomponenten angegeben werden. Dann wird die Knotenliste in entsprechend viele Abschnitte unterteilt und es werden nur Kanten innerhalb der Abschnitte erzeugt. Selbstverständlich können mehr Zusammenhangskomponenten entstehen, wenn nicht alle Knoten in einem Abschnitt zusammenhängend verbunden werden.

Es wurden vier Konfigurationen des Lösers getestet, nämlich eine sequentielle und eine mit allen entwickelten Parallelisierungen jeweils mit beiden RN-Strategien; bei der optimalen Strategie, die alle Möglichkeiten des RN-Knotens durchprobiert, war auch das Branch-and-Bound-Verfahren aktiviert. Zusätzlich wurden die parallelen Löser sowohl mit vier als auch mit acht X10-Worker-Threads getestet. Der sequentielle Löser bekam selbstverständlich nur einen Worker-Thread. Dabei wurde auch die Umgebungsvariable „X10_STATIC_THREADS“ gesetzt, sodass die X10-Laufzeitumgebung nicht eigenmächtig zusätzliche Threads erzeugt. Trotzdem waren immer einige Threads mehr vorhanden; diese zusätzlichen Threads sind für die Garbage-Collection verantwortlich und führen im Vergleich zum Lösungsalgorithmus nur wenige Berechnungen aus.

Bei den Tests waren immer wieder Schwankungen der Ausführungszeiten um bis zu 5 % zu beobachten. Unter Verwendung von vier Worker-Threads lagen einige Ausreißer sogar 15 % über dem besten, mit der gleichen Konfiguration und Einga-

be gemessenen Wert. Für die Auswertung wurde daher immer der beste Wert von mindestens vier Messungen herangezogen. Für die Schwankungen ist vermutlich die zufällige und zudem wechselnde Zuordnung der Worker-Threads zu Prozessorkernen verantwortlich. Wechselt ein Thread den Prozessorkern, gehen die Inhalte des L1- sowie evtl. des L2-Caches verloren. Gleichzeitig kann der Wechsel des Prozessorkerns aber auch von Vorteil sein, wenn dadurch die Temperatur in einzelnen Kernen niedrig gehalten wird, weil die Taktrate durch Intels Turbo-Boost dynamisch bei niedriger Betriebstemperatur angehoben werden kann. Das würde auch erklären, warum die Schwankungen bei vier Worker-Threads auf acht Prozessorkernen am stärksten sind. Dann ist nämlich immer die Hälfte der Prozessorkerne für einen Wechsel frei.

Testfall 1 ist ein PBQP-Graph mit insgesamt 1800 Knoten und 1880 Kanten die sich auf sechs große, sowie einige sehr kleine und triviale Zusammenhangskomponenten verteilen. Die gemessenen Ausführungszeiten finden sich in Tabelle 1. Bei der heuristischen Variante und vier Threads ist ein SpeedUp von $1,810$ durch die Parallelisierung zu beobachten. Dieser wächst aber nur wenig auf $2,386$ wenn man die Thread-Anzahl verdoppelt. Das volle Potenzial von acht Kernen kann also nicht ausgenutzt werden. Bei der optimalen RN-Strategie ist die parallele Variante um Größenordnungen schneller als die sequentielle. Das liegt nicht nur an der Parallelität, sondern auch daran, dass durch die Aufteilung in Zusammenhangskomponenten mehrere kleine Suchbäume anstatt eines einzigen, riesigen Suchbaums entstehen. Aktiviert man im sequentiellen Löser die Aufteilung in Zusammenhangskomponenten, aber ohne diese dann parallel zu bearbeiten, ergibt sich eine Ausführungszeit von ca. 270 s gegenüber mehr als 15 Minuten ohne eine solche Aufteilung. Die Aufteilung in Zusammenhangskomponenten lohnt sich also nicht nur zur Parallelisierung, sondern auch in sequentiellen Lösern. Der parallele Löser bleibt aber immer noch um das $1,937$-fache bei vier Threads bzw. um das $2,271$-fache bei acht Threads schneller.

Konfiguration	**Worker-Threads**	**Zeit**	**SpeedUp**
heuristisch, sequentiell	1	0,494 s	–
heuristisch, parallel	4	0,273 s	1,810
heuristisch, parallel	8	0,207 s	2,386
optimal, sequentiell	1	> 15 min	–
optimal, sequentiell[1]	1	270,461 s	–
optimal, parallel	4	139,649 s	1,937
optimal, parallel	8	119,086 s	2,271

[1] Mit Aufteilung in Zusammenhangskomponenten. Dient hier als Referenzwert zur Berechnung des SpeedUp.

Tabelle 1: Messungen zu Testfall 1.

Im Testfall 2 besteht der PBQP-Graph aus 300 Knoten und 299 Kanten und ist fast vollständig zusammenhängend. Das entspricht in etwa einer der großen Zusammenhangskomponenten aus Testfall 1. Die Parallelisierung von Zusammenhangskomponenten kommt hier also nicht zum Tragen. Die Messergebnisse befinden sich in Tabelle 2. Für die heuristische Variante ist nur ein geringer SpeedUp von $1,189$ bzw.

1,370 erkennbar. Bei der optimalen RN-Strategie zeigt die Parallelisierung aber einen deutlicheren SpeedUp von 1,952 bzw. 2,969. Das zeigt, dass in diesem Testfall vor allem die Parallelisierung der optimalen RN-Strategie einen Vorteil bringt.

Konfiguration	**Worker-Threads**	**Zeit**	**SpeedUp**
heuristisch, sequentiell	1	0,063 s	–
heuristisch, parallel	4	0,053 s	1,189
heuristisch, parallel	8	0,046 s	1,370
optimal, sequentiell	1	4,928 s	–
optimal, parallel	4	2,525 s	1,952
optimal, parallel	8	1,660 s	2,969

Tabelle 2: Messungen zu Testfall 2.

Der Testfall 3 besteht aus 1000 Knoten und 9903 Kanten und besteht nach der Normalisierung aus einer sehr großen Zusammenhangskomponente und 104 Einzelknoten. Aufgrund der vielen Kanten sind hier sehr viele RN-Reduktionen nötig und die optimale RN-Strategie ist in diesem Testfall nicht mehr sinnvoll. Stattdessen wurde nur die heuristische Variante gemessen. Tabelle 3 zeigt die Messergebnisse. Ein Geschwindigkeitsvorteil der parallelen Variante ist kaum auszumachen. Mit acht Worker-Threads nimmt die Geschwindigkeit des parallelen Lösers sogar wieder ab. Der Grund dafür sind das Fehlen mehrerer Zusammenhangskomponenten und der damit verbundenen Parallelität sowie die vielen RN-Knoten, die eine Partitionierung des Graphen und parallele R1- und R2-Reduktionen verhindern. Eine separate Messung der Normalisierung zeigt, dass diese Phase durchaus von der Parallelisierung profitiert. Bei der Normalisierung wird mit vier bzw. acht Worker-Threads ein Geschwindigkeitsvorteil von 1,537 bzw. 1,811 erzielt. Dieser Vorteil wird aber durch die Berechnungen zur Partitionierung des Graphen wieder zunichte gemacht.

Konfiguration	**Worker-Threads**	**Zeit**	**SpeedUp**
heuristisch, sequentiell	1	0,992 s	–
heuristisch, parallel	4	0,855 s	1,160
heuristisch, parallel	8	0,927 s	1,070
nur Normalisierung, sequentiell	1	0,661 s	–
nur Normalisierung, parallel	4	0,430 s	1,537
nur Normalisierung, parallel	8	0,365 s	1,811

Tabelle 3: Messungen zu Testfall 3.

Testfall 4 besteht im Vergleich zum vorherigen aus einem wesentlich dünneren Graphen; er besitzt 5000 Knoten aber nur 2997 Kanten, die vom Testgenerator auf 10 Zusammenhangskomponenten verteilt wurden. Da der Graph so dünn ist, entstehen nach der Normalisierung aber weit mehr Zusammenhangskomponenten sowie 1754 Einzelknoten. Die Messergebnisse befinden sich in Tabelle 4. Bei Verwendung der heuristischen RN-Strategie verdoppelt sich die Geschwindigkeit durch die Parallelisierung. Mit der optimalen RN-Strategie ergibt sich ein SpeedUp von 4,549 mit vier Worker-Threads und sogar 5,484 mit acht Worker-Threads. Auf einem derart dünnen Graphen tragen vor allem die Partitionierung des Graphen und die parallelen R1- und R2-Reduktionen zur Beschleunigung bei und es wird ein deutlich größerer Geschwindigkeitsvorteil als im vorhergehenden Testfall erreicht.

Konfiguration	**Worker-Threads**	**Zeit**	**SpeedUp**
heuristisch, sequentiell	1	0,663 s	–
heuristisch, parallel	4	0,347 s	1,911
heuristisch, parallel	8	0,307 s	2,160
optimal, sequentiell	1	3,016 s	–
optimal, parallel	4	0,663 s	4,549
optimal, parallel	8	0,550 s	5,484

Tabelle 4: Messungen zu Testfall 4.

Bis auf Testfall 1 ist der Geschwindigkeitsvorteil durch die Parallelisierung mit optimaler RN-Strategie überall größer als mit der heuristischen Strategie. In Testfall 1 kam aber eine variierte Version des sequentiellen Lösers zum Einsatz, bei der zusätzlich eine Aufteilung in Zusammenhangskomponenten vorgenommen wurde. Das zeigt, dass auch die Parallelisierung der optimalen RN-Strategie stets zu einer Verkürzung der Ausführungszeit führt.

Auch die Heuristik und die optimale RN-Strategie lassen sich anhand der gefunden Lösungen vergleichen (siehe Tabelle 5). In Testfall 1 weicht die heuristisch gefundene Lösung nur um 0,25 % von der optimalen ab. In den Testfällen 2, 3 und 4 findet die Heuristik sogar die optimale Lösung, und das bei deutlich geringerer Ausführungszeit.

Testfall	**heuristisch**	**optimal**
1	184.668	184.199
2	29.711	29.711
3	865.846	865.846
4	365.084	365.084

Tabelle 5: Heuristisch gefundene und optimale Lösungen in den vier Testfällen.

6 Ausblick

Aus Zeitgründen konnten im Rahmen dieser Studienarbeit zwei Ideen nicht weiter verfolgt werden. Zum einen könnte man die Synchronisationsmechanismen optimieren und dadurch möglicherweise noch weitere Geschwindigkeitsvorteile erreichen. So wird im implementierten Löser das X10-Konstrukt „atomic“ verwendet, um kritische Abschnitte und Daten vor konkurrierenden Zugriffen zu schützen. Dieser Mechanismus blockiert global alle kritischen Abschnitte, sodass es zu einer stärkeren Behinderung der parallelen Aktivitäten untereinander kommt. Einzelne kritische Abschnitte ließen sich unabhängig voneinander mit Locks schützen. Diese können jedoch nicht serialisiert werden, was das Kopieren des PBQP-Graphen und -Lösers und damit auch eine Aufteilung der Daten auf mehrere Berechnungsknoten verhindert. Zukünftige Weiterentwicklungen könnten sich dieser Problematik annehmen, um noch bessere Ergebnisse zu erzielen.

Des Weiteren steht ein Anschluss an das in [7] vorgestellte Framework zur invasiven Programmierung noch aus. Die entwickelten Parallelisierungen weisen aber auch für die invasive Programmierung einige gute Möglichkeiten auf. Entscheidend sind dabei, wie gut die benötigten Ressourcen ermittelt werden können und welche Daten kopiert werden müssen.

Bei der parallelen Normalisierung können anhand der Größe des gefundenen Matchings die benötigten Ressourcen zur parallelen Bearbeitung bestimmt und dann angefordert werden. Beim Belegen der Hardware müssen nur die zu normalisierende Kante und ihre inzidenten Knoten kopiert werden. Allerdings müssen diese nach der Normalisierung auch vollständig wieder zurück kopiert und in den PBQP-Graphen übernommen werden.

Die Aufteilung in Zusammenhangskomponenten ermöglicht ebenfalls eine einfache Entscheidung über die benötigten Rechen-Ressourcen; für jede Zusammenhangskomponente wird ein Prozessor gebraucht. Zu jedem Prozessor muss jeweils nur die eine Zusammenhangskomponente, die gelöst werden soll, kopiert werden. Im Anschluss an die Berechnungen werden nur die endgültige Lösung und die in den Knoten gewählten Alternativen am ursprünglichen Prozessor benötigt. Die gleichen Vorteile gelten auch für die Parallelisierung der optimalen RN-Strategie. Es wird genau ein Prozessor für jede Auswahlmöglichkeit im RN-Knoten gebraucht und nur der Lösungswert sowie die im weiteren Verlauf noch getroffenen RN-Entscheidungen müssen zurück kopiert werden.

Die Partitionierung einzelner Zusammenhangskomponenten ermöglicht in Bezug auf den Hardwarebedarf ein umgekehrtes Vorgehen. Hier kann auf Basis der zur Verfügung stehenden Hardware die maximale Größe und damit indirekt die Anzahl der Partitionen gesteuert werden. Ein Problem ergibt sich hier aber im Zusammenspiel mit dem Branch-and-Bound-Verfahren. Dabei müssen alle verteilten Prozessoren auf die zentral gespeicherte untere Schranke zugreifen, um sie nach jeder Reduktion anzupassen. Das erschwert ein invasives Vorgehen in diesem Fall.

7 Fazit

In dieser Arbeit wurden vorhandene Lösungsstrategien für PBQP parallelisiert. Dazu wurden fünf verschiedene Parallelisierungen entwickelt. Grundsätzlich ist durch jede der Parallelisierungen eine Beschleunigung möglich. Die Geschwindigkeitsvorteile fallen jedoch abhängig von der Struktur des vorliegenden PBQP-Graphen unterschiedlich stark aus. Wenn der Graph aus mehreren Zusammenhangskomponenten besteht, verkürzt deren parallele Bearbeitung die Ausführungszeit deutlich. Darüber hinaus konnte auch gezeigt werden, dass sich die Aufteilung eines PBQP-Graphen in Zusammenhangskomponenten bei Verwendung der optimalen RN-Strategie auch in einem sequentiellen PBQP-Löser lohnt. Die Partitionierung innerhalb einer Zusammenhangskomponente verfehlt in sehr dichten PBQP-Graphen deutlich ihre Wirkung, weil kaum parallele R1- und R2-Reduktionen möglich sind. In großen, dünn besetzten Zusammenhangskomponenten führt die Partitionierung jedoch zu sehr guten Ergebnissen. Auch die Parallelisierung der optimalen RN-Strategie zeigt eine zusätzliche Beschleunigung. Gegenüber den Messungen mit der heuristischen Strategie sind mit der optimalen größere Geschwindigkeitsgewinne durch die Parallelisierung zu verzeichnen. Insgesamt konnte der vorhandene PBQP-Lösungsalgorithmus also erfolgreich durch die Parallelisierungen beschleunigt werden.

Literaturverzeichnis

[1] Buchwald, S., Zwinkau, A., Bersch, T.: SSA-based register allocation with PBQP. In: Knoop, J. (ed.) Compiler Construction, Lecture Notes in Computer Science, vol. 6601, pp. 42–61. Springer Berlin / Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-19861-8_4

[2] Chaitin, G.: Register allocation and spilling via graph coloring. SIGPLAN Not. 39(4), 66–74 (Apr 2004), http://dx.doi.org/10.1145/989393.989403

[3] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. SIGPLAN Not. 40(10), 519–538 (2005), http://dx.doi.org/10.1145/1103845.1094852

[4] Cormen, T.H., Leiserson, C.E., Rivest, R., Stein, C., Molitor, P.: Algorithmen-Eine Einführung. Oldenbourg Wissensch.Vlg (2010), http://books.google.de/books?id=Yk6bqjYjPfgC

[5] Eckstein, E., König, O., Scholz, B.: Code instruction selection based on SSA-graphs. In: Krall, A. (ed.) Software and Compilers for Embedded Systems, Lecture Notes in Computer Science, vol. 2826, pp. 49–65. Springer Berlin / Heidelberg (2003), http://dx.doi.org/10.1007/978-3-540-39920-9_5

[6] Hames, L., Scholz, B.: Nearly optimal register allocation with PBQP. In: Lightfoot, D., Szyperski, C. (eds.) Modular Programming Languages, Lecture Notes in Computer Science, vol. 4228, pp. 346–361. Springer Berlin / Heidelberg (2006), http://dx.doi.org/10.1007/11860990_21

[7] Hannig, F., Roloff, S., Snelting, G., Teich, J., Zwinkau, A.: Resource-aware programming and simulation of MPSoC architectures through extension of X10. In: Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems. pp. 48–55. SCOPES '11, ACM, New York, NY, USA (Jun 2011), http://pp.info.uni-karlsruhe.de/uploads/publikationen/hannig11scopes.pdf

[8] Harris, J., Hirst, J., Mossinghoff, M.: Combinatorics and graph theory. Undergraduate texts in mathematics, Springer (2008), http://books.google.de/books?id=CxSoZcNymacC

[9] Hoffmann, C.M., O'Donnell, M.J.: Pattern matching in trees. J. ACM 29(1), 68–95 (Jan 1982), http://dx.doi.org/10.1145/322290.322295

[10] IBM: X10 website, http://www.x10-lang.org

[11] Jakschitsch, H.: Befehlsauswahl auf SSA-Graphen. Master's thesis, Universität Karlsruhe (TH) (Nov 2004), http://www.info.uni-karlsruhe.de/papers/da_jakschitsch.pdf

[12] Scholz, B., Eckstein, E.: Register allocation for irregular architectures. SIGPLAN Not. 37(7), 139–148 (Jun 2002), http://dx.doi.org/10.1145/566225.513854