Entwurf und Implementierung verteilter Lösungsansätze für Capital Budgeting Probleme mit GLPK und Apache thrift

Development of a distributed system with Apache thrift
to solve Capital Budgeting Problems with GLPK

Bachelorarbeit
zur Erlangung des Grades *Bachelor of Science*

an der
Hochschule Niederrhein
Fachbereich Elektrotechnik und Informatik
Studiengang *Informatik*

vorgelegt von
Jochen Peters
xxxxxxx

Krefeld, den 10. September 2015

Prüfer:	Prof. Dr. Jochen Rethmann
Zweitprüfer:	Prof. Dr. Steffen Goebbels

Zusammenfassung

Das Toolkit *glpk* enthält Methoden der gemischt-ganzzahligen linearen Programmierung (MILP). Diese Methoden lassen sich zur Lösung von Capital Budgeting Problemen (CBP) nutzen. Da *glpk* weder multithreaded fähig ist noch eine Verteilung auf mehrere Rechner implementiert ist, kann es sein Potential auf heutigen Computern und in vernetzten IT-Systemen nicht voll entfalten. Um bei der Lösung von CBP mit kommerziellen Toolkits wie *CPLEX* und *Gurobi* mithalten zu können, haben wir mit dem Verteilungs-Framework *Apache thrift* und der `C`-API von *glpk* eine Möglichkeit geschaffen, um CBP auf beliebig viele Kerne und Rechner zu verteilen. Durch Last-Balancierung und Berücksichtigung der Methoden der MILP bei der Verteilung war es uns sogar möglich, die Verarbeitung um mehr als das Vielfache der genutzten Prozessor-Kerne zu beschleunigen. Der „Branch-and-Cut"-Algorithmus von *glpk*, mit dem primär gearbeitet wird, liefert bei einer ausgewogenen Unterteilung in Teilprobleme, paralleler Verarbeitung und regelmäßigem Austausch der bisher besten Lösung unter den Teilproblemen deutlich schneller eine Lösung, als eine sequenzielle Verarbeitung der Teilprobleme.

Abstract

The toolkit *glpk* supports methods for mixed integer linear programming (MILP). These methods solve Capital Budgeting Problems (CBP). Unfortunately, *glpk* does not support any multithreading and there is no feature to distribute problems via network connections. Today, this is a pitiable sight, because modern computer systems are coupled by networks and support multi threading. To solve CBP with *glpk* like commercial toolkits (*CPLEX* or *Gurobi*) we create a distributed system with *Apache thrift* and the `C`-API of *glpk*. Now, it is possible to use as many cores in a network as you want. With a focus on the MILP methods we implement a load balancing and speed up the solving process in a multiplicative way. Sometimes we have super-linear speedup with a small set of hardware. *glpk* primary uses the "Branch-and-Cut" algorithm. With an intelligent splitting of problems, parallel computing and distributing the actual best solution to all running processes we solve CBP much faster than a sequential processing can do.

Eidesstattliche Erklärung

Name:	Jochen Peters
Matrikelnr.:	xxxxxxx
Titel:	Entwurf und Implementierung verteilter Lösungsansätze für Capital Budgeting Probleme mit GLPK und Apache thrift

Ich versichere durch meine Unterschrift, dass die vorliegende Arbeit ausschließlich von mir verfasst wurde. Es wurden keine anderen als die von mir angegebenen Quellen und Hilfsmittel benutzt.

Die Arbeit besteht aus **50** Seiten.

Krefeld, den 10. September 2015

Unterschrift

Inhaltsverzeichnis

1 Motivation

Während der Praxisphase an der Hochschule wurde ich auf eine Webseite[1] aufmerksam, auf der eine Reihe von Testinstanzen (Chu and Beasley, Glover and Kochenberger, SAC-94) für ein dort beschriebenes „0-1 multidimensional knapsack problem“ aufgelistet sind. Dies ist nur ein anderer Name für eine spezielle Variante des Capital Budgeting Problems (CBP) oder Projekt-Selektions-Problem. Es lässt sich gut mit einem Beispiel beschreiben:

> Sie leiten eine Bank und haben für einen Zeitraum von 10 Monaten für jeden Monat j ein Budget b_j zur Verfügung. Dieses Budget wollen Sie so stark es geht in einige Projekte investieren. Sie haben eine Auswahl von 250 Projekten, die je Projekt i pro Monat j einen Teil des Budgets verbrauchen – dieser Verbrauch sind die Kosten c_{ji}. Nach 10 Monaten bekommen Sie für jedes Projekt i einen Profit p_i. Das Budget reicht leider je Monat nicht aus, um alle Projekte zu unterstützen. Ein Projekt kann nur ganz (1-fach), oder gar nicht bearbeitet werden (x_i ist also binär). Jetzt ist die Frage: Welches Portfolio x an Projekten wählen Sie aus, um nach 10 Monaten den meisten Gewinn L_{opt} zu bekommen?

Mathematisch ist dieses Problem allgemein wie folgt formuliert:

$$L_{opt} = \sum_{i=1}^{250} p_i x_i \quad \text{ist zu maximieren}$$

mit den Randbedingungen:

$$b_j \geq \sum_{i=1}^{250} c_{ji} x_i \qquad 1 \leq j \leq 10$$

$$x_i \, \epsilon \, \{0,1\}$$

Die Variablen x_i bilden das zu findende Portfolio und repräsentieren eine Lösung[2], bei dem der Gewinn L_{opt} maximal ist. Die Variablen p_i, den 10 Budgets b_j, die je Zeitraum j zur Verfügung stehen, sowie einer Kostenmatrix c_{ji}, die je Zeitraum j die Kosten des Projekts i enthält, beschreiben alle Größen des Problems. Das hier beschriebene CBP ist das Multi-Period CBP und ist eine Verallgemeinerung des eindimensionalen Rucksack-Problems, welches bereits zu den NP-schweren Problemen gehört (siehe Garey & Johnson [6], Anhang: Punkt MP9).

Will man den Gewinn aller 2^{250} Portfolios – das sind 10^{75} Projekt-Kombinationen – berechnen, da allgemein kein effizienter Algorithmus bekannt ist, so würde dies extrem lange dauern. Hier ein Rechenbeispiel: Angenommen ein Takt einer 1GHz CPU würde den Gewinn eines Portfolios ausrechnen und man könnte die „Rechenpower“ durch Vervielfältigung der CPUs linear erhöhen, wie viele CPUs würden wie lange rechnen? Nimmt

[1] `http://www.cs.nott.ac.uk/~jqd/mkp/`

[2] ein Vektor von Binärzahlen

man als Beispiel das Alter der Erde, was auf 4,6 Milliarden Jahre geschätzt wird, kommt man auf ca. 10^{17} Sekunden. Mit nur einer 1GHz CPU sind das 10^{26} Berechnungen. Man müsste also die Anzahl der CPUs erhöhen. Bleibt man als Beispiel bei der Erde: Sie hat ein Volumen von ca. $10^{21}m^3$. Wenn man dieses Volumen mit CPUs ausfüllt, z.B. mit 10 CPUs pro Liter, dann kommt man auf ca. 10^{25} CPUs die je 10^{26} (=Alter der Erde) lang arbeiten. Das wären dann schon 10^{51} Berechnungen. Um in dem Beispiel nicht über die Größe der Erde hinausgehen zu müssen, verkleinern wir die Größe der CPUs auf Wassermolekülgröße. In dem Fall passen nicht 10, sondern ca. 10^{25} CPUs in einen Liter. So erreicht man endlich 10^{75} Berechnungen. Wie Sie sicher bemerkt haben, ist dieser Aufwand einen erdgroßen Supercomputer zu bauen, bloß um den maximalen Profit (42 ?) und einen 250-stelligen Binärcode zu ermitteln, etwas zu groß. Auch eine Bank, die auf stetiges Wachstum setzt, muss hier das Limit der Ressourcen der Erde und vor allem der Zeit respektieren. Dies ist eine Überlegung, die man Science-Fiction Autoren überlassen sollte.

Es gibt bisher eine Vielzahl unterschiedlicher Kombinationen von Algorithmen und Techniken, die Aufgaben des CBP lösen. Bis zu einer geringen Anzahl von Projekten kann man auch die Kombinationen an Projekten einfach durchprobieren. Eine Beschleunigung der Berechnung ist durch eine Heuristik möglich. So könnte man z.B. zuerst die Projekte auswählen, die den meisten Profit bringen (Greedy) oder die die geringsten Kosten je Profit haben. So eine Heuristik kann aber auch fehlschlagen und nicht zu einem maximalen Gewinn führen. Der Fokus unserer Arbeit liegt jedoch auf der Verteilung von Teilaufgaben und deren Verarbeitung durch *glpk*-eigene Algorithmen.

Im Vordergrund unserer Arbeit steht die Nutzung von Open-Source-Software ohne jegliche Optimierung in Bezug auf CBP. Sie soll ohne Lizenzgebühren auf handelsüblicher Bürohardware laufen. Optimierungs-Software wie *CPLEX* oder *Gurobi*, die nur im Umfeld der Hochschule kostenfrei genutzt werden können, fallen daher weg. In einer Präsentation von *Gurobi* [12] liegen die Lizenkosten von *Gurobi* und *CPLEX* im 5 stelligen Dollar-Bereich. So fällt für kleinere Unternehmen die Möglichkeit weg, gelegentlich selber Optimierungsprobleme zu lösen. Es gibt zwar Dienste, die man Stunden oder Tageweise mieten kann, aber ob eine Firma Projektkosten, Profite etc. an eine 3. Firma übergeben will ist sehr fraglich. Es handelt sich doch um sehr sensible, für die Firma wichtige Daten, die da übermittelt werden müssten. Eine kostenlose, betriebsinterne Software, die z.B. als Boot-Image oder Bildschirm-Schoner genutzt werden kann, ist da deutlich attraktiver. Das *SETI-at-home* Projekt bot früher einen Bildschirm-Schoner an, der aus dem Internet Radio-Teleskop-Daten bekam, die dann die Rechenleistung von Privat-Computern zur Suche von außerirdischen Signalen nutzte. Aus dieser Idee wurde später das *BOINC*-Projekt, welches eine Verteilung unterschiedlicher Aufgaben ermöglicht[3]. Zu Anfang hatten wir einen eigenen BOINC-Server ins Auge gefasst, entscheiden uns aber für eine deutlich spezialisierte Art der Verteilung mit *Apache thrift*. Durch eine Verteilung wollen wir die Ressourcen einer Firma besser nutzen, da uns bisher noch keine Open-Source-Software zur Optimierung bekannt ist, die die Möglichkeit der Ressourcenverteilung über ein Netzwerk anbietet.

[3] `https://boinc.berkeley.edu/`

2 Problemstellung

Das Capital Budgeting Problem (CBP), bei dem ein Portfolio gesucht wird, dessen Gewinn maximal ist und kein monatliches Budget überschritten wird, lässt sich mit einem *Branch-and-Cut*-Algorithmus verhältnismäßig schnell lösen, obwohl es zu den NP-schweren Problemen gehört. Den *Branch-and-Cut* nutzen wir, in dem wir auf die C-API von *glpk* zugreifen.

Die Auswahl bzw. Nicht-Auswahl eines Projektes bei z.B. 250 Projekten lässt sich als binärer Entscheidungsbaum darstellen (Abbildung 1). In der Tiefe von 250 befindet sich dann die Entscheidung, ob das letzte Projekt für ein Portfolio genommen oder nicht genommen wird.

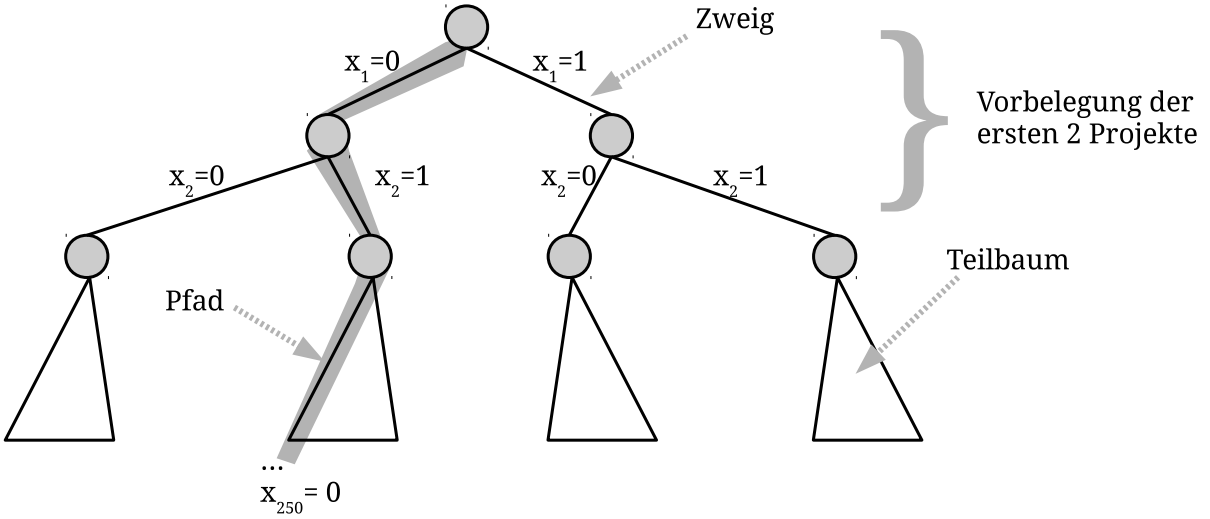

Abbildung 1: Vorbelegung der Projekte als Entscheidungsbaum. Der Pfad von der Wurzel bis zum Blatt ist ein Portfolio.

Das Aufsplitten des Entscheidungsbaums in Zweige (engl.: branches), deren Teilbäume dann jeweils von Rechnern in unserem verteilten System bearbeitet werden, ist die erste Herausforderung unserer Software. Unser Konzept zur Verteilung des CBP sieht vor, dass wir z.B. von den 250 Projekten die ersten 15 Projekte fest vorgeben. Durch diese Vorbelegung entstehen 2^{15} Teilbäume, die als Jobs der Reihe nach an *glpk* übergeben werden. Die gleichmäßig ausbalancierte Verteilung über das Netzwerk sowie die Nutzung aller CPU-Kerne der Rechner ist eine weitere Herausforderung. Mit der C-API von *glpk* (ohne CBP-spezialisierte Algorithmen) und normaler Bürohardware größere CB-Probleme über Nacht oder am Wochenende zu lösen, haben wir als Gesamtziel definiert. Dieses Ziel ließ sich hinterher nur erreichen, in dem zum *Branch-and-Cut* von *glpk* ein verteilter *Branch-and-Bound*-Algorithmus implementiert wurde. Details zu beiden Algorithmen finden Sie im nächsten Kapitel.

3 Lösung

Um die von uns erarbeitete Software zu verstehen, werden wir nun auf einzelne Algorithmen, Begriffe und Techniken näher eingehen.

Um die Suche nach dem Optimum (beim CBP das Maximum des Gewinns) im Entscheidungsbaum (Abbildung 1) zu beschleunigen, bildet man durch eine Vorbelegung der Projekte Teilbäume, in denen jeweils wieder ein Optimum gesucht wird. Dieses Verzweigen (engl.: branching) in Teilbäume ermöglicht die Betrachtung kleinerer Probleme, da weniger Projekte berücksichtigt werden müssen. Für jeden Teilbaum lässt sich mit Methoden der Linearen Programmierung (LP) wie z.B. dem Simplex-Verfahren von 1947 eine obere Grenze berechnen. Seit 1984 existiert für die LP auch ein Algorithmus (Karmarkar [8]) der diese obere Grenze in polynominaler Zeit in Abhängigkeit von der Anzahl der Projekte und Budget-Grenzen (=Bedingungen) löst. Das Simplex-Verfahren ist jedoch sehr leicht zu programmieren und liefert schnell eine Lösung, obwohl es mit den gleichen Eingabegrößen (Anzahl der Variablen und Bedingungen) schlimmstenfalls eine exponentielle Laufzeit hat. Das Simplex-Verfahren hat gegenüber dem Algorithmus von Karmarkar einen bedeutenden Vorteil: während der Berechnung können Bedingungen hinzugefügt werden. Diese Möglichkeit des Hinzufügens begünstigt andere Algorithmen (spezielle Heuristiken) die z.B. aus bestehenden Lösungen neue Bedingungen bilden, die das mathematische Problem weiter einschränken und den Lösungsraum, in dem gesucht werden muss, verkleinern.

Bei der LP ist im Gegensatz zur binären, ganzzahligen Linearen Programmierung (BILP) der gesamte reelle Zahlenraum für die Wahl der Projekte erlaubt. Mit zusätzlichen Bedingungen muss man daher den Raum für die Lösung einschränken, so dass die Wahl der Projekte zwischen 0 und 1 bleibt. Dieses Abbilden der BILP auf die LP wird LP-Relaxierung genannt. Das Maximum, was bei der LP-Relaxierung ermittelt wird, ist immer größer oder gleich dem Maximum der BILP. Der Grund dafür ist, dass Projekte auch „anteilig“ gewählt werden können, wie z.B. 0,3 (30%) von Projekt 23 und 0,8 (80%) von dem Projekt 51.

Mit Hilfe der LP-Relaxierung lässt sich also für Teilbäume eine obere Grenze des Gewinns abschätzen. Sobald sie nun diese (oder anders berechnete) Grenzen (Bounds) dazu nutzen, um bestimmte Teilbäume (z.B. die mit der größeren oberen Grenze) erneut zu verzweigen (engl.: branchen) und wieder eine LP-Relaxierung machen (oder eine andere Berechnung, um Grenzen zu bestimmen), dann nutzen Sie bereits den *Branch-and-Bound*-Algorithmus. Durch ein rekursives Vorgehen bewegen Sie Sich im Entscheidungsbaum immer weiter entlang der „gewinnträchtigsten“ Teilbäume nach unten, bis Sie entweder durch die LP-Relaxierung eine binäre Lösung erhalten, oder das Ende des Baums erreicht haben. In beiden Fällen besitzen Sie dann eine erste, bisher beste Lösung, die Ihnen für die nächsten Untersuchungen der Teilbäume als untere Grenze dient. Sollte nun bei der LP-Relaxierung ein Teilbaum als obere Grenze schlechter als Ihre bisher beste Lösung sein, dann brauchen Sie diesen Teilbaum nicht mehr weiter zu untersuchen. Auf diese Weise können viele Projektkombinationen ausgeschlossen werden und es muss nicht jede Kombination berechnet werden.

Mit *Branch-and-Bound* lassen sich oft sehr schnell Lösungen finden, obwohl Probleme der

BILP zu den NP-schweren Problemen gehören ([7], Seite 35) und daher nicht effizient lösbar sind. Im Kapitel *Aussicht* auf Seite 41 werden wir das nochmal ansprechen, da die Annahme, Teilbäume seien schneller zu lösen als der entsprechende ganze Baum, nicht zwingend richtig ist. Der *Branch-and-Bound*-Algorithmus kann auch schlimmstenfalls so lange bei 250 Projekten brauchen, wie die Erde alt ist.

Eine Erweiterung des *Branch-and-Bound* ist der *Branch-and-Cut*-Algorithmus. Beim *Branch-and-Cut* werden ebenfalls obere Grenzen betrachtet, allerdings werden die Zweige, an denen die Teilbäume hängen, nicht durch ein Vorbelegung von Projekten, sondern durch das Hinzufügen von Schnittebenen (engl.: cutting planes) gebildet. Es handelt sich dabei um zusätzlich Bedingungen, die mit Hilfe von Heuristiken erzeugt werden[4]. Das Gebiet der Heuristiken reicht sehr weit. In einer sehr ausführlichen Arbeit [4] von 2011 wird deutlich, dass nicht zwingend *eine* Heuristik optimal ist, um zusätzliche Bedingungen als Schnittebenen zu erzeugen. So werden häufig mehrere Heuristiken parallel angewendet, die dann[5] durch Meta-Heuristiken [10] ausgewertet werden. Meta-Heuristiken entscheiden, welche Heuristik zur Laufzeit aktuell am besten den Lösungsraum einschränkt, und weißt dieser Heuristik z.B. mehr CPU-Ressourcen zu. In der *glpk*-Dokumentation wird auch davon gesprochen, dass die in *glpk* implementierte Heuristik die hinzugefügten Schnittebenen bewertet und ggf. wieder entfernt. Heuristiken suchen z.B. möglichst schnell eine Lösung, die alle Bedingungen erfüllt, und nehmen diese als untere Grenze, oder sie bilden das duale Problem, welches durch eine Transponierung der Matrix aus Bedingungen und Zielfunktion ein Minimierungs-Problem macht. Eine gültige Lösung des dualen Problems (Minimierung der Budgets) stellt automatisch eine obere Grenzen des ursprünglichen primalen Problems (Maximierung des Gewinns) dar[6]. Auf diese Weise lässt sich auch eine obere Grenze finden, die als neue Bedingung hinzugefügt wird.

3.1 *glpk*

Das GNU Linear Programming Kit – kurz: *glpk* – bietet eine Vielzahl von Routinen und Konfigurations-Möglichkeiten, um eine breite Palette unterschiedlicher Probleme der LP zu lösen, deren Lösungsverlauf zu beobachten und in die zugrunde liegenden Algorithmen einzugreifen. *glpk* kann gleichzeitig mit jedem Variablentyp (binär, integer, reell) in Zielfunktion und Bedingungen umgehen. Das wird auch allgemein als Mixed Integer Linear Programming (MILP oder kurz: MIP) genannt. Zu Anfang hatten wir die Überlegung, in den Open-Source-Code von *glpk* direkt ein zu greifen, um eine Parallelisierung auf mehrere Kerne (multithreading) und später eine Verteilung im Netzwerk zu ermöglichen. Als Vorlage für diese Idee hatten wir ein Patch für ein *glpk*-Plugin [13], welches 2010 von der NASA entwickelt wurde, in Augenschein genommen. Dieses Patch sollte eine Multithreaded-Fähigkeit von *glpk* erlauben. Es war jedoch für eine deutlich ältere Version von *glpk* gedacht, die in keinster Weise mit der aktuellen Version 4.55 aus dem Jahr 2014 vergleichbar ist. Daher entschieden wir uns statt mehrere Threads mehrere Prozesse auf einem Rechner laufen zu lassen. Für die Verteilung der Rechenleistung auf die Kerne ist somit das Scheduling des Betriebssystems zuständig. Außerdem würde ein

[4] Das Simplex-Verfahren ermöglicht das Hinzufügen von weiteren Bedingungen.
[5] das macht aber *glpk* nicht!
[6] schwacher Dualitätssatz [1], Seite 146

Eingriff in *glpk* zur Folge haben, dass Neuerungen von *glpk* in unserer Software nicht mit einfließen würden. Genauso wie beim *glpk*-Plugin der NASA würde der Entwicklungsstand „eingefroren“ und eine Firma, die unsere Software einsetzen will, würde von einer Weiterentwicklung *glpk*s nichts haben. Die Alternative wäre eine aktive Mitentwicklung von *glpk*, die voraussichtlich den zeitlichen Rahmen von 12 Wochen für diese Bachelorarbeit gesprengt hätte.

Ein paar der *Chu-and-Beasley* Instanzen ließen sich auch auf einem CPU-Kern mit dem *glpk* Kommandozeilen-Tool `glpsol` lösen. Anstelle der `C`-API von *glpk* und *Apache thrift* zur Verteilung von Teil-Instanzen, wäre auch eine Kombination eines verteilten Dateisystems, Shell-Skripten, einzelne selbst geschriebene Programme sowie `telnet` und `ssh` möglich. Diese Idee wurde jedoch schnell wieder verworfen, da sie sich nur schwer auf ein anderes Betriebssystem portieren lässt. Für kleine Firmen ohne IT-Abteilung und nur mit Windows-Systemen wäre dies nicht praktikabel. Eine „ready to use“-Binärdatei für Linux-Distributionen, Windows-Systeme und Mac erschien uns deutlich eleganter, unanfälliger und auch performanter, als eine Reihe von Skript-Dateien, Binärdateien und viele Systemvoraussetzungen für eine Datei-Verteilung.

Mit der C-API von *glpk* ist es möglich zur Laufzeit Zwischenergebnisse (obere und untere Grenzen) ab zu fragen und auf diese zu reagieren. Während des *Branch-and-Cut*-Algorithmus lassen sich außerdem Bedingungen wie die Einhaltung von neu berechneten Grenzen hinzufügen und verändern. Ebenso ist es möglich, eigenen Code zu hinterlegen, der nach erreichen eines Zeitlimits Zwischenergebnisse (z.B. die obere Grenze) abspeichert. Eine weitere Einflussmöglichkeit, die wir eingesetzt haben, war die Nutzung des Presolvers beim Aufruf der Methode `glp_intopt()` zur Lösung von MIP-Problemen. Laut der Dokumentation von *glpk* „verbessert“ der Presolver die Bedingungen des Problems. Da wir nicht integer, sondern binäre Variablen wollen, ließ sich noch die Heuristik „Fischetti–Monaci Proximity Search“ [5] (siehe auch *glpk* Dokumentation[9], Seite 64) aktivieren, die den Presolver deutlich verbessern soll. Die Einflussmöglichkeit bestand nun in der Art, dass dieser Presolver zusammen mit der Heuristik erst ab einer bestimmten Laufzeit Sinn macht, da für einen positiven Effekt in der Laufzeit Presolver und Heuristik selbst ein paar Sekunden laufen müssen, bevor der reguläre *Branch-and-Cut*-Algorithmus beginnt. Wir testeten Presolver und Heuristik mit ein paar der *Chu-and-Beasley* Instanzen, die mit dem Kommandozeilen-Tool `glpsol` Stunden zur Lösung brauchten. Diese ließen sich tatsächlich mit `glpsol` und aktivem Presolver und Heuristik in wenigen Minuten lösen.

Mit der `C`-API von *glpk* sind somit viele Techniken der MILP und auch LP nutz-, realisier- und anpassbar.

3.2 *Apache thrift*

Das Framework *Apache thrift* [14] wurde 2007 ursprünglich von *facebook* entwickelt. Es ist dem *RPC* (Remote procedure call) von Sun Microsystems sehr ähnlich, weist aber folgende Stärken auf:

- der Codegenerator produziert auf jedem Betriebssystem einen multithreaded-Server
- es ist möglich, nicht nur `C` und `C++` tauglichen Code zu erzeugen, sondern auch z.B. für Python, Java, Javascript, PHP, Ruby, Erlang, Perl, Haskell und C#

- die Weiterentwicklung des Codes ist an keine Firma gebunden

Bei *Apache thrift* (kurz: *thrift*) wurden die Schwachstellen von RPC weitestgehend beseitigt. So soll durch die große Open-Source-Community, der der Quellcode unter der Apache Lizenz zur Verfügung steht, eine Stagnation bei der Weiterentwicklung des Frameworks verhindert werden.

Der Codegenerator von *thrift* kann die Struktur- und Service-Definitionen der thrift-IDL in eine Vielzahl unterschiedlicher Sprachen übersetzen. In `C++` werden aus den Struktur-Definitionen Klassen, die beiden Verbindungspartnern bekannte sind. In den Service-Definitionen werden Methoden beschrieben, die dann beim Client als Methoden eines Client-Objekts zur Verfügung stehen. Ähnlich wie bei RPC, wird für die Sprache `C++` auch eine *Skeleton*-Datei erzeugt, in dem man nur noch die Programmlogik der Methoden serverseitig einfügen muss. Diese sollte man jedoch als Vorlage für seinen eigenen Code verstehen, da eine neue Übersetzung durch den Codegenerator in der Default-Einstellung diese Datei wieder überschreiben würde.

Für einen multithreaded-Server, der also mehrere Anfragen parallel bearbeiten kann, bedient sich *thrift* im wesentlichen der `libevent`-Bibliothek, der thread-Implementierung des *boost*-Frameworks, sowie dessen Shared-Pointern. Sowohl `libevent` als auch das *boost*-Framework ist für viele Programmiersprachen und Betriebssysteme verfügbar.

Eine der angenehmsten Eigenschaften von *thrift* ist die Übersetzung von grundlegenden Datentypen in die entsprechende Zielsprache. Gibt man in der thrift-IDL z.B. als Datentyp `list<list<double> >` an, erhält man beim Erzeugen des `C++`-Codes: `vector<vector<double> >`. Es wird keine `list` erzeugt, da es in anderen Programmiersprachen als Datentyp auf jeden Fall so etwas wie ein Feld bzw. Vektor gibt. Dies entspricht dem kleinsten gemeinsamen Container-Datentyp.

Die IDL-Beschreibung für unser *thrift*-Client/Server-System schaut so aus:

```
1  // Enthält die gesamten Daten zur Lösung des Problems
2  struct Instance {
3      1: list<i32> _budgets, // Budget-Grenzen je Zeitraum
4      2: list<i32> _profits, // Profit je Projekt
5      3: list<list<i32> > _prices // Kosten je Zeitraum und Projekt
6  }
7
8  struct Job {
9      1: i32 _opt,         // bisher bekanntes Optimum
10     2: list<bool> _fix, // Vorbelegung
11     3: bool _end,        // Das ist ein Flag, um mit einem Job einem
12                          // blockierten Worker mit zu teilen, dass
13                          // keine weiteren Jobs folgen.
14     4: bool _timeout,    // Markierung, dass Job nicht fertig wurde
15     5: i32 _bestBound    // obere Grenze des Teilbaums
16 }
17
18 /* Die Methoden eines Servers werden in sogenannten
19  * Services zusammengefasst. Ein Server kann mehrere
```


```
20  * Services enthalten.
21  */
22 service RemoteGlpk {
23
24     // Methode zur Initialisierung
25     void input(
26         1:Instance data,
27
28         // Flag, damit Worker weiss, dass er nach erreichen des
29         // Zeitlimits obere Grenze speichern muss
30         2:bool transportBound,
31
32         // zur Steuerung, ab welcher Vorbelegung der Presolver
33         // einsetzen soll
34         3:double presolverStartPercent
35     ),
36
37     // übertragen und starten eines Jobs
38     Job solve(1:Job best, 2:Job newJob, 3:i32 zeitlimit),
39
40     // abbrechen von solve()
41     void kill(),
42
43     // austauschen der bisher besten Lösung
44     Job update(1:Job best)
45 }
```

thrift kann synchrone Aufrufe an einen Server senden, bei denen auf eine Antwort des Servers gewartet wird. Mit diesem blockierenden Aufruf verhindern wir, dass der Client, der für die Verteilung von Teilbäumen in Form von Jobs zuständig ist, dem Server weitere Jobs zustellt. Um trotz des blockierenden Aufrufs eine parallele Verarbeitung zu ermöglichen, setzten wir für jede Verbindung einen Thread ein, welcher ein Stellvertreter (Proxy) für den Methodenaufruf an den thrift-Server ist (Proxy-Entwurfsmuster, Abbildung 2).

Aufgrund der mangelnden multithreaded-Fähigkeit von *glpk*, welches in der Speicherung von Daten in globalen Variablen begründet ist, haben wir für den *thrift*-Server zu Anfang eine nicht multithreaded-Version genommen. Um trotzdem alle 4 CPU-Kerne eines Rechners nutzen zu können, starteten wir die Server 4 mal – jeweils auf einem anderen Port. Der Client, der die Jobs erzeugt, muss daher zu einem Rechner 4 Verbindungen aufbauen. In unserer aktuellen Version setzen wir aber wieder einen Multithreaded-Server ein, der aber immer noch 4 mal auf einem Rechner gestartet wird. Durch die Multithreadedfähigkeit des Servers ist es uns möglich, mit einer 2. Verbindung Einfluss auf die Verarbeitung des Jobs zu nehmen, wie z.B. mit der `kill()` Methode zum Abbruch des Jobs oder mit `update()` zum Abgleich der bisher besten Lösung[7].

[7] In `solve()` wird *glpk* benutzt, was die bisher beste Lösung als untere Grenze in einer zusätzlichen Bedingung (neben den Budget-Grenzen) nutzt.

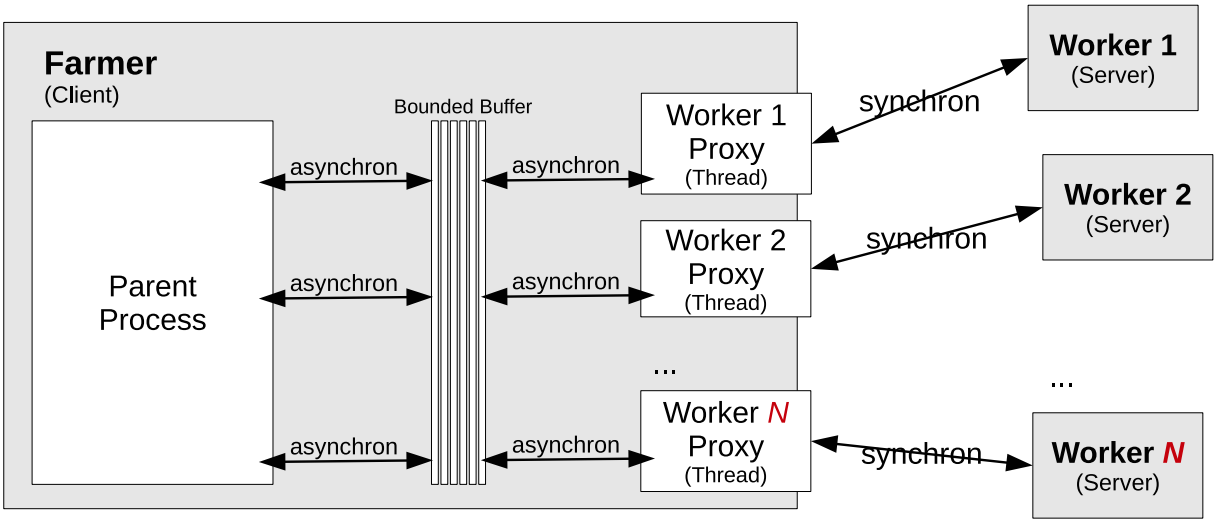

Abbildung 2: Asynchroner Aufruf durch die Nutzung eines „Worker-Proxys“ beim Farmer

3.3 Konzept

Unser Konzept zur Verteilung des CBP sieht vor, dass wir z.B. von den 250 Projekten die ersten 15 Projekte fest vorgeben. Durch diese Vorbelegung entstehen 2^{15} Teilbäume, die als Jobs der Reihe nach an *glpk* übergeben werden. Um eine Verteilung (und somit eine parallele Verarbeitung) zu ermöglichen, ließen wir auf jedem Rechner in der Anzahl seiner Kerne *thrift*-Server laufen. Neben der Übergabe des Teilbaums wird auch ein Zeitlimit, eine bisher beste Lösung und eine mit dieser Vorbelegung obere Grenze übergeben. Die bisher beste Lösung und die obere Grenze stellen die Grenzen dar, in denen das Maximum als neue beste Lösung des Teilbaums zu erwarten ist – daher kann man daraus auch eine zusätzliche Bedingung neben den Bedingungen der Budget-Grenzen erzeugen. Der einzelne Client, der die Jobs an die *thrift*-Server sendet, bezeichnen wir als Farmer, während wir die Server, die die Jobs mit *glpk* bearbeiten, als Worker bezeichnen. Leider ist *glpk* nicht thread-fähig, so dass wir die CPU-Kerne der Rechner nur durch die Nutzung mehrerer Worker-Prozesse auf einem Rechner nutzen können. In der Abbildung (3, Seite 17) im Kapitel *Architektur* sind diese Worker-Prozesse auf den Rechnern im Netzwerk als **Worker N** dargestellt.

Eine Aufteilung und parallele Verarbeitung von CBPs mit *glpk* ist so möglich. Da uns z.B. nicht 2^{15} Kerne zur Verfügung standen, sondern z.B. nur 32, wurden die vielen Jobs, die sehr unterschiedlich lang dauerten, verhältnismäßig ungleichmäßig an die 32 Worker verteilt. Es kam oft zu der Situation, dass am Ende nur noch ein Worker an einem Job arbeitete ... und das für deutlich über 50% der gesamten Verarbeitungszeit. Daher entschieden wir uns für ein Zeitlimit, was angibt wie lange ein Job auf dem Worker arbeiten darf. Wird dieses Limit erreicht, dann speichern wir die bisherigen Grenzen von *glpk* und erstellen aus diesem Job neue Jobs mit erweiterter Vorbelegung. Mit anderen Worten: wir „branchen“ erneut und verzweigen den Teilbaum des Jobs in neue Teilbäume – also in neue Jobs. Wir entschieden uns, dass man über die Kommandozeile angeben kann, bis in welche Pfadtiefe des Entscheidungsbaums (in %) diese Erweiterung der Vorbelegung stattfinden darf. Ab dem Erreichen dieser Tiefe wird den Jobs eine beliebige Verarbeitungsdauer ge-

währt[8]. Als weitere Option ist es möglich, die Vergrößerung der Vorbelegung an zu geben. Gibt man z.B. eine 10 an, so werden $2^{10} = 1024$ neue Jobs aus dem Alten erzeugt – es werden also 10 weitere Projekte vorbelegt. Die Abbildung (3, Seite 17) enthält dieses Beispiel. Mit unserem Programm kann man außerdem durch die Angabe eines Faktors das Zeitlimit verändern. So ist es möglich auszuprobieren, ob man mit einem Zeitlimit von einigen Sekunden die ersten Jobs starten sollte und dann bei einer größeren Vorbelegung das Zeitlimit herunter setzt ($0 <$ Faktor < 1), oder ein kleines Zeitlimit vorteilhafter ist, welches bei jeder neuen Vorbelegung erhöht wird (Factor > 1).

Wenn der *Branch-and-Cut* auf nur einem Kern ohne Verteilung läuft, wird ständig die obere Grenze mit der bisher besten Lösung verglichen. Dieses Vergleichen ist im Grunde der Teil des *Branch-and-Cut*, der mit dem *Branch-and-Bound*-Algorithmus identisch ist. Daher entschieden wir uns, dass dieser Wert zentral beim Farmer für alle Worker zur Verfügung stehen sollte, damit die parallel laufenden *Branch-and-Cuts* der Worker nicht nur ihre lokal bisher beste Lösung verwenden bzw. nicht nur die bisher beste Lösung, die bei der Übertragung des Jobs an den Worker vom Farmer übermittelt wird. Um den Austausch zwischen den Workern zu ermöglichen, verwenden wir den Farmer als eine Art Relais-Station. In kurzen Abständen pingt der Farmer der Reihe nach alle Worker an, erfragt deren bisher beste Lösung und ein Abgleich findet statt (siehe auch Abbildung 6, Seite 21). So existiert, wenn auch mit einem kleinen Versatz, immer eine bisher beste Lösung für eine untere Grenze bei den Workern. Mit einer Callback-Funktion, die man bei *glpk* hinterlegen kann, wird während des *Branch-and-Cut* regelmäßig diese übermittelte bisher beste Lösung L^t_{opt} als neue untere Grenze in der Berechnung aufgenommen. Sollte diese bisher beste LösungL^t_{opt} schlechter als die lokal bisher beste Lösung L^t_{lokal} des Workers sein, so wird diese lokal bisher beste Lösung genommen. Wenn der Framer dann erneut nach einer bisher besten Lösung des Workers fragt, so greift dieser auf die neue bisher beste Lösung L^{t+1}_{opt} des Workers zu, die in der Callback-Funktion zuvor hinterlegt wurde. Im Kapitel *Architektur* ist unter der Abbildung (6) auf Seite 21 das Verfahren genauer dargestellt. Auf diese Art realisieren wir eine verteilte Version des *Branch-and-Bound*-Algorithmus.

Ein aktives Versenden der lokalen bisher besten Lösung zwischen den Workern hielten wir für unsinnig. Der Grund dafür ist der Farmer, bei dem die Teillösungen der Jobs zusammenlaufen und auf dem so prinzipiell ständig neue Kandidaten für die bisher beste Lösung eingehen. Eine Ring-Struktur, bei der diese bisher beste Lösung als untere Grenze zwischen den Workern durchgereicht wird, ist jedoch nicht völlig abwegig und sollte in Zukunft einmal näher untersucht werden.

Zur Lösung kann *glpk* einen Presolver einsetzen, der in der Lage ist, die Bedingungen neu zu formulieren, zu vereinfachen und ggf. durch Linearkombination effizientere zu erzeugen. Dies geht z.B. durch die Betrachtung des dualen Problems([7], Seite 243), welches aus dem Maximierungsproblem des Capital Budgeting Problems ein Minimierungsproblem macht. Durch die Angabe, dass es sich um binäre Variablen handelt, wird der Presolver von *glpk* noch deutlich schneller. Außerdem lässt sich noch eine auf binäre Probleme spezialisierte

[8]Es ist für die Zukunft angedacht, dass in Extremfällen, wenn der Farmer keine offenen Jobs mehr hat und nur noch ein Worker sehr lange arbeitet, diese Pfadtiefe überschritten werden kann und der einzelne Job neu aufgeteilt wird.

Heuristik – „Fischetti–Monaci Proximity Search“ [9] – anwenden, die den Presolver unterstützt und auch obere Grenzen finden kann, welche als zusätzliche Bedingungen hinzugefügt werden. Da diese Heuristik selber etwas Zeit benötigt (60 Sekunden ist per Default in *glpk* als Limit für diese Heuristik eingestellt), lässt sich in unserem Programm über die Kommandozeile als Option eine Pfadtiefe des Entscheidungsbaums (in %) angeben, ab dem der Presolver (inkl. Heuristik) aktiv werden darf.

Als weiteren Verbesserungsversuch sortierten wir in aufsteigender Reihenfolge die Projekte nach der Summe ihrer Kosten pro Profit. Die Projekte, die dann zuerst von uns vorbelegt werden und zuerst von einer Iteration durch die möglichen Kombinationen betroffen sind, beeinflussen den Gesamt-Gewinn am stärksten. Ursprünglich war von uns eine eigene LP-Relaxierung geplant, um auch bei der Vorbelegung bereits ein *Branch-and-Cut* machen zu können. Dies haben wir aus Zeitgründen nicht mehr geschafft. Als Überbleibsel ist jedoch eine initiale Vorbelegung der Projekte übrig geblieben, die einer „gerundeten[10] Simplex-Lösung“ entspricht. So sollte erreicht werden, dass die Vorbelegung möglichst dicht an einer möglichen finalen Lösung liegt. Wir haben durch Tests an Instanzen, die in nur wenige Minuten zu berechnen waren, festgestellt, dass die LP-Lösung (Simplex) zu einem großen Teil bereits Projekte in binärer Form auswählt. Ein Vergleich mit der BILP-Lösung ergab, dass diese Projektwahl zum Teil identisch war. Wir haben leider diesen Punkt aus Zeitgründen nicht mehr näher untersuchen können.

Als weitere Option ist unserem Programm die Art der Abarbeitung der Jobs mit Zeitüberschreitung zu übergeben. Gibt man „depth“ an, so wird eine Tiefensuche gemacht. Es werden dann die Jobs zuerst abgearbeitet, die **zuletzt** eine Zeitüberschreitung hatten: Dies hat einen bedeutenden Vorteil gegenüber einer Breitensuche, da im schlimmsten Fall der Buffer, der die Jobs enthält, nur additiv mit der Pfadtiefe ansteigt, sollten ständig Zeitüberschreitungen stattfinden (Abbildung 5). Mit einer Breitensuche, bei der zuerst die Jobs abgearbeitet werden, die **als erstes** ihr Zeitlimit erreichten, können schlimmstenfalls in der untersten (z.B. die 250.) Pfadebene 2^{250} Jobs zur Abarbeitung bereit stehen. Diese Menge ist nicht speicherbar! Bisher ist uns dies nur bei Instanzen passiert, die trotz unserer Verteilung länger als 1 Tag laufen. Unser Programm weist an dieser Stelle eine Schwäche auf. In einer aktuellen Version versuchen wir, bei Speicherproblemen auf die Tiefensuche automatisch um zu schalten. Allerdings fehlt uns noch ein Konzept, ab wann wir wieder zurück auf die Breitensuche umschalten. Durch die Jobs, die durch die Tiefensuche entstehen, bleibt das Programm immer sehr nahe an einem vorgegebenen Limit an Jobs, ab dem wir auf die Tiefensuche umgeschaltet haben.

[9] siehe *glpk* Dokumentation[9], Seite 64

[10] ein Wert wie 0.5 bei der Projektwahl wird auf 1, und darunter auf 0 gerundet, so dass der Lösungvektor einem binärem Ergebnis entspricht

4 Architektur

Um das von uns entwickelte und getestete verteilte System besser verstehen zu können, werden in diesem Kapitel die verwendeten Strukturen dargestellt.

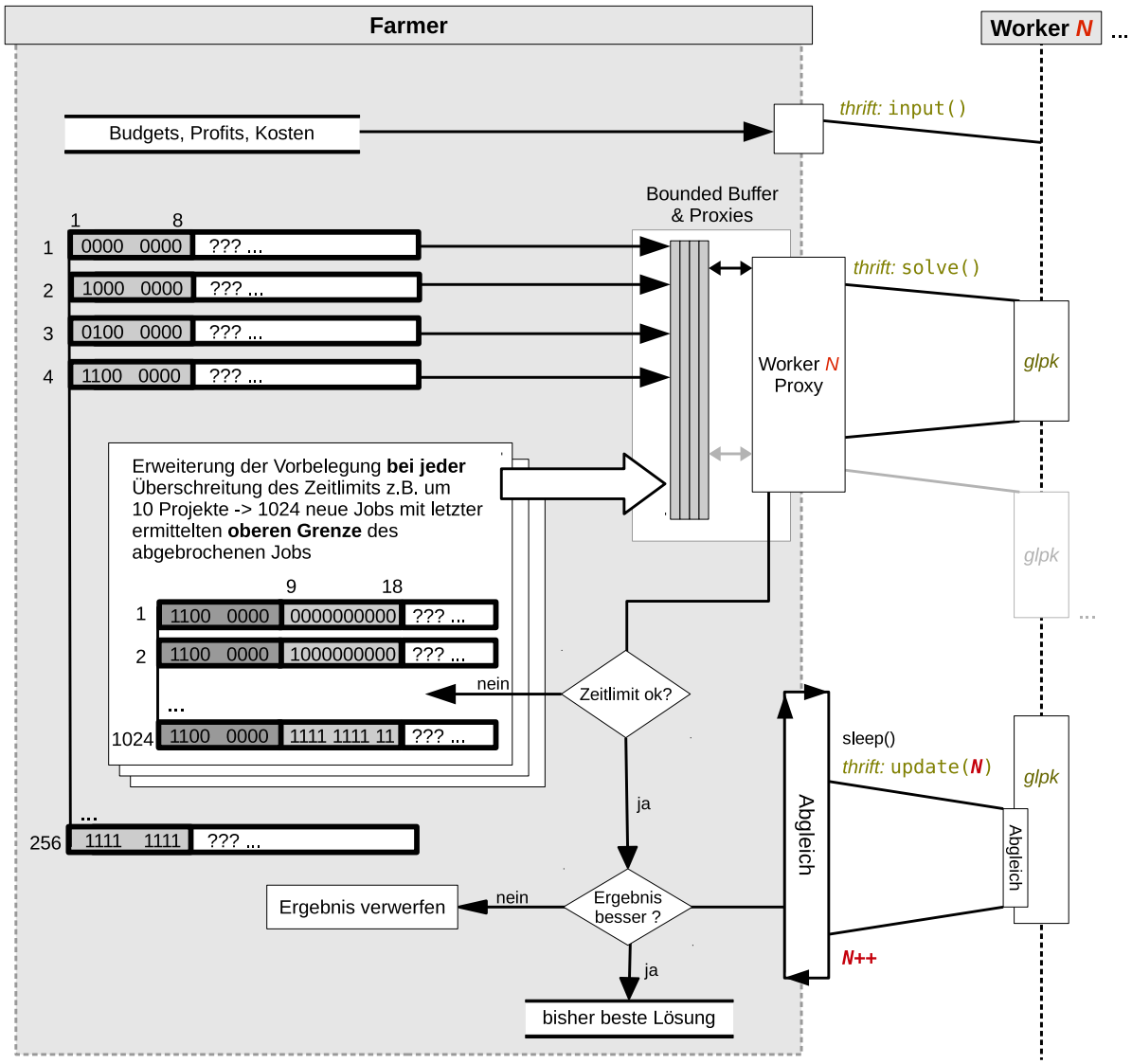

Abbildung 3: Konzept zur Verteilung von Jobs für *glpk* mit *Apache thrift*

In der Abbildung (3) wird der strukturelle Aufbau unseres verteilten Systems gezeigt. Die Logik zur Reihenfolge, in der der *Bounded Buffer* mit Jobs gefüllt wird (Breiten- oder Tiefensuche) ist nicht mit angegeben. Ebenso ist die Logik nicht dargestellt, welche ab einer bestimmten Vorbelegung kein Zeitlimit mehr setzt sowie den Presolver von *glpk* aktiviert. Das Abgleichen der bisher besten Lösung auf einem Worker ist ebenfalls nur angedeutet und wird im Kapitel *Quasi globale bisher beste Lösung* genauer beschrieben. Die darin beteiligte Callback-Funktion von *glpk*, mit der eine neue untere Grenze erstellt wird, würde ebenfalls den Rahmen Grafik sprengen.

Die Abbildung (4) zeigt den Aufbau unseres Quellcodes und dient als Referenz, um nachfolgende Beschreibungen besser zu verstehen.

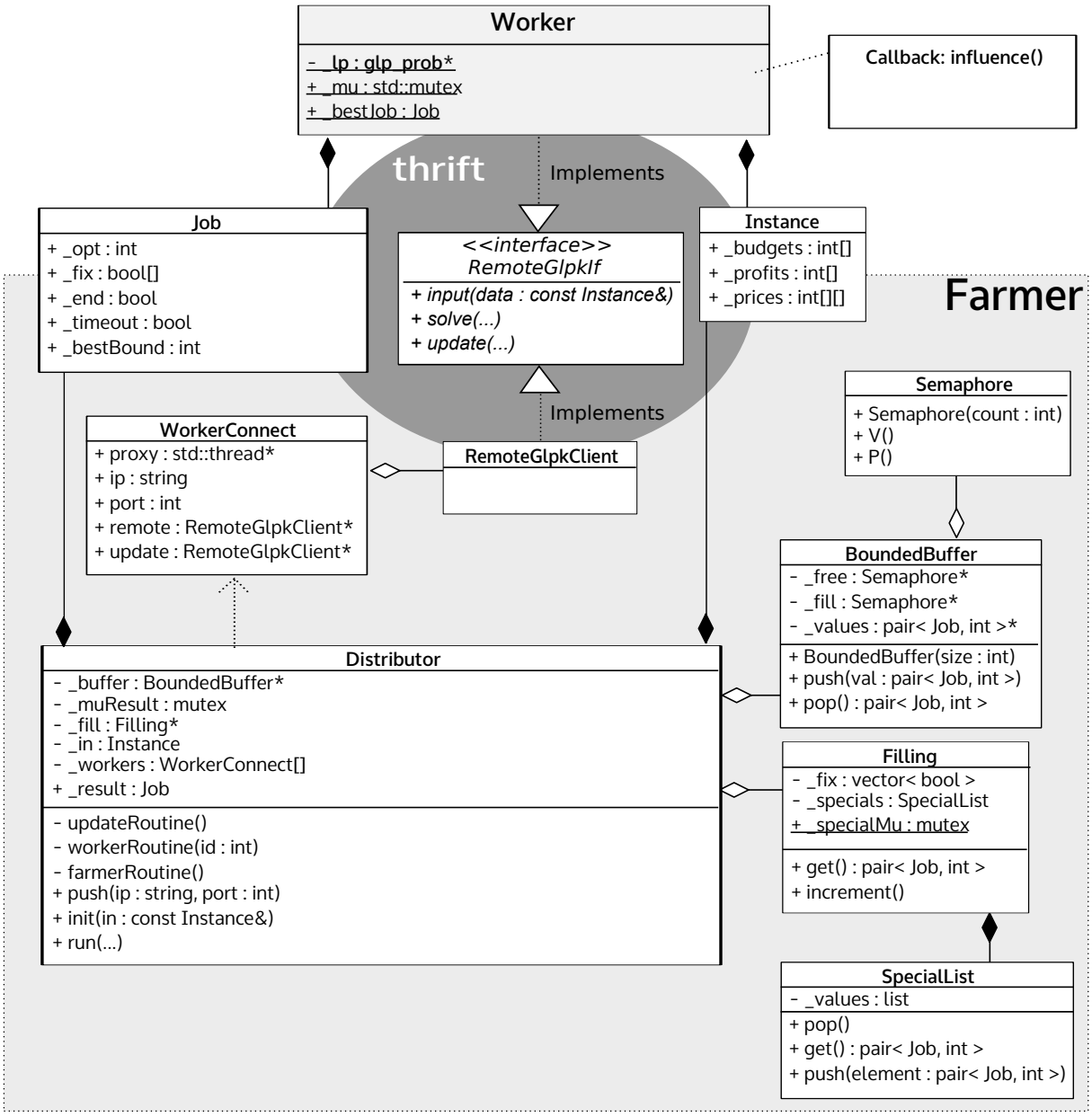

Abbildung 4: UML-Diagramm unserer Software

4.1 Farmer/Worker

Das Farmer/Worker-Modell ist die Grundlage, auf der unsere Software aufgebaut ist. Der Farmer erzeugt Jobs, die dann von Workern abgearbeitet werden. Auf einem Rechner mit mehreren Kernen lassen sich auf diese Weise parallel Jobs auf mehrere Worker-Threads verteilen.

Das Modell lässt sich auch in einem Netzwerk realisieren, in dem die Worker als Server Rechenleistung bereitstellen, und ein Client durch seine Anfragen an die Server, die Rolle des Farmers übernimmt. Ein Worker läuft dann als Prozess auf einem Server, während die Worker-Threads nur noch eine stellvertretende Rolle (Proxies) einnehmen (Abbildung 2, Seite 14).

Zur asynchronen Abarbeitung von Jobs (Abbildung 2) verwenden wir einen *Bounded Buffer*, da wir diese Lösung des Erzeuger-Verbraucher-Problems bereits im Studium kennengelernt haben. Dieses Problem, welches in unserer thread-basierten Farmer/Worker Implementierung auftaucht, wird in *Bounded Buffer* mit zwei „normalen"- und einer binären-Semaphore (Mutex) gelöst. Mit Mutex wird der kritische Abschnitt (Zugriff auf das Feld mit den Jobs) geschützt, während die Semaphoren[11] für das Blockieren und Freigeben beim Hinzufügen `_fill` bzw. der Entnahme `_free` von Jobs zuständig sind.

4.2 Last-Balancierung

Das Herzstück unserer Bemühungen einer Verteilung war die Last-Balancierung. Während das Farmer/Worker-Modell unsere Orientierung ist, *wie* wir Aufgaben/Jobs verteilen, so bestimmt die Last-Balancierung, *was* für Jobs verteilt werden. Außerdem werden *Parameter* von uns definiert, die auf die *Verteilung* und *Verarbeitung* der Jobs Einfluss nehmen bzw. auf die zur Laufzeit Einfluss genommen werden.

Statisch

Einer der ersten Parameter zur Last-Balancierung war die Anzahl der Projekte, die der Farmer fest vorbelegt. Diesen Parameter nannten wir K. Bei z.B. 16 Workern wählten wir zu Anfang fest (statisch) $K = \log_2(16) = 4$, um jedem Worker einen Job mit gleicher Größe zu geben. Es wurden dann vom Farmer alle 16 Kombinationen, die durch die Wahl/Abwahl von 4 Projekten möglich sind, vorbelegt. Jeder Worker bekam dann bei z.B. einer Instanz mit 250 Projekten nur ein Teilproblem mit 246 offenen Projekten. Setzt man K nun deutlich höher an (z.B. 10 oder 15), werden sehr viele, etwas kleinere Jobs geschaffen und es besteht weniger die Gefahr, dass ein Worker nach seiner Job-Verarbeitung für lange Zeit nichts zu tun hat. Die Auslastung der 16 Worker ist auf diese Weise gleichmäßiger verteilt und es können mehr Berechnungen zur selben Instanz in gleicher Zeit stattfinden.

Dynamisch

Für eine dynamische Last-Balancierung haben wir ein Zeitlimit (Timeout) definiert, mit dem wir bestimmen, ab wann ein Job eine Teil-Instanz bekommen hat, die besser auf

[11] Abbildung 4, Klasse `BoundedBuffer`

mehrere kleinere Teil-Instanzen aufgeteilt werden sollte. So ist es uns möglich beim Erreichen des Zeitlimits zur Laufzeit das K zu dieser „ungünstigen“ Vorbelegung zu vergrößern und in mehrere Jobs aufzuteilen.

Als weiteren dynamischen Parameter haben wir mit diesem Zeitlimit gearbeitet und sowohl eine Erhöhung des Limits als auch eine Herabsetzung je K ausprobiert. Mehr dazu finden Sie im Kapitel *Modifikation des Zeitlimits*.

Einen Einfluss auf die optimale Abarbeitung der Jobs hatte auch die Auswahl, in welcher Reihenfolge die Jobs abgearbeitet werden sollen. Da jede Erhöhung von K tiefere Ebenen im Binärbaum der Vorbelegung bedeutet, führt eine Breitensuche im Extremfall zu einem exponentiellen Anstieg der noch zu bearbeitenden Jobs je Erhöhung von K. Bei einer Tiefensuche hingegen, bei der die neu erzeugten Jobs umgehend von den Workern abgearbeitet werden, kommt zu jeder Erhöhung von K nur die Menge der neuen Jobs hinzu (Abbildung 5).

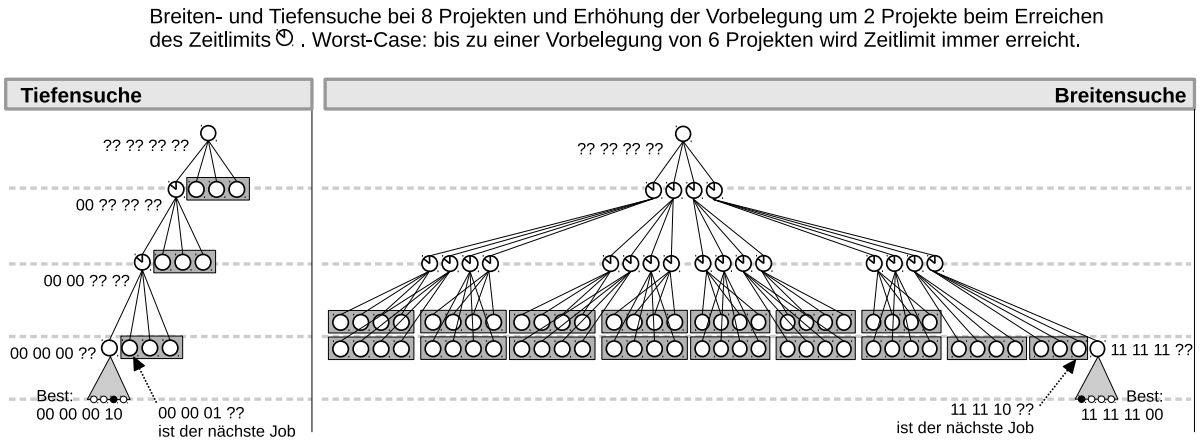

Abbildung 5: Vergleich der Breiten- mit der Tiefensuche. Anzahl der offenen Jobs (grau hinterlegte Kreise) kann im Worst-Case exponentiell zu der Anzahl der Überschreitung des Zeitlimits zunehmen.

Auf diese Weise ist die Last im Sinne einer Speicherbelastung gut aufteilbar. Leider sind wir nicht mehr dazu gekommen, eine sinnvolle dynamische Umschaltung der Suche zu implementieren. Es ist nur eine statische Auswahl beim Starten unseres Programms möglich.

4.3 „Quasi globale“ bisher beste Lösung

Weil dies ein sehr spezieller Punkt ist, wollen wir noch mal im Detail auf den Austausch der bisher besten Lösung unter den Workern eingehen. Dies läuft über einen Thread `Distributor::updateRoutine()` beim Farmer ab. In der Abbildung (3) ist ein Sequenzdiagramm (**Abgleichen**) dazu dargestellt. In regelmäßigen Zeitintervallen (als `sleep()` in Abbildung 6 dargestellt) wird jeder Worker mit der Routine `RemoteGlpkClient::update()` angesprochen. Diese Routine sorgt dafür, dass auf dem Farmer und auf dem Worker die selbe bisher beste Lösung (= untere Grenze) ist. Bei diesem Abgleich gewinnt der größere Wert. Wir entschieden uns für dieses Konzept, weil

beim Farmer durch die Lösungen der Jobs ohnehin eine zentrale Sammelstelle für Lösungen existiert, aus der die bisher beste Lösung zum Schluss als Optimum hervorgeht. In der Klasse `Distributor` wird die Variable `_result` für die bisher beste Lösung mit einer Mutex-Semaphore `_muResult` geschützt, weil auch die Threads `workerRoutine()` nach Abarbeitung eines Jobs auf diese Variable zugreifen und ihr Ergebnis – nach einem Vergleich – ggf. dort ablegen.

Serverseitig – also beim Worker – wird die durch `update()` übermittelte bisher beste Lösung des Farmers ebenfalls verglichen und zwischengespeichert (`Worker::_bestJob`). Auch hier ist eine Mutex-Semaphore `Worker::_mu` beteiligt, da parallel in einem Thread `Worker::solve()` die Berechnung eines Jobs läuft (laufen kann). Die bisher beste Lösung wird durch diesen Thread immer dann angefasst, wenn *glpk* in die Callback-Funktion `influence()` springt. Jedes mal, wenn eine neue, lokale bisher beste Lösung vorliegt, wird dann diese mit der durch `update()` übertragene bisher Besten verglichen und ggf. ausgetauscht. Ebenso wird mit der Callback-Funktion diese neue, lokale bisher beste Lösung als neue untere Grenze in den Bedingungen der Berechnung des Jobs eingebaut.

Schematisch ist dieser Austausch in Abbildung (6) dargestellt. Auf diese Weise lässt sich ein verteilter *Branch-and-Bound*-Algorithmus realisieren. Um den Einfluss des regelmäßigen Abgleichs der bisher besten Lösung untersuchen zu können, haben wir den Zeitintervall des `sleep()` über einen Parameter beim Aufruf unseres Programms anpassbar gemacht. Mehr dazu finden Sie im Kapitel *Einfluss des Sync.-Intervalls*.

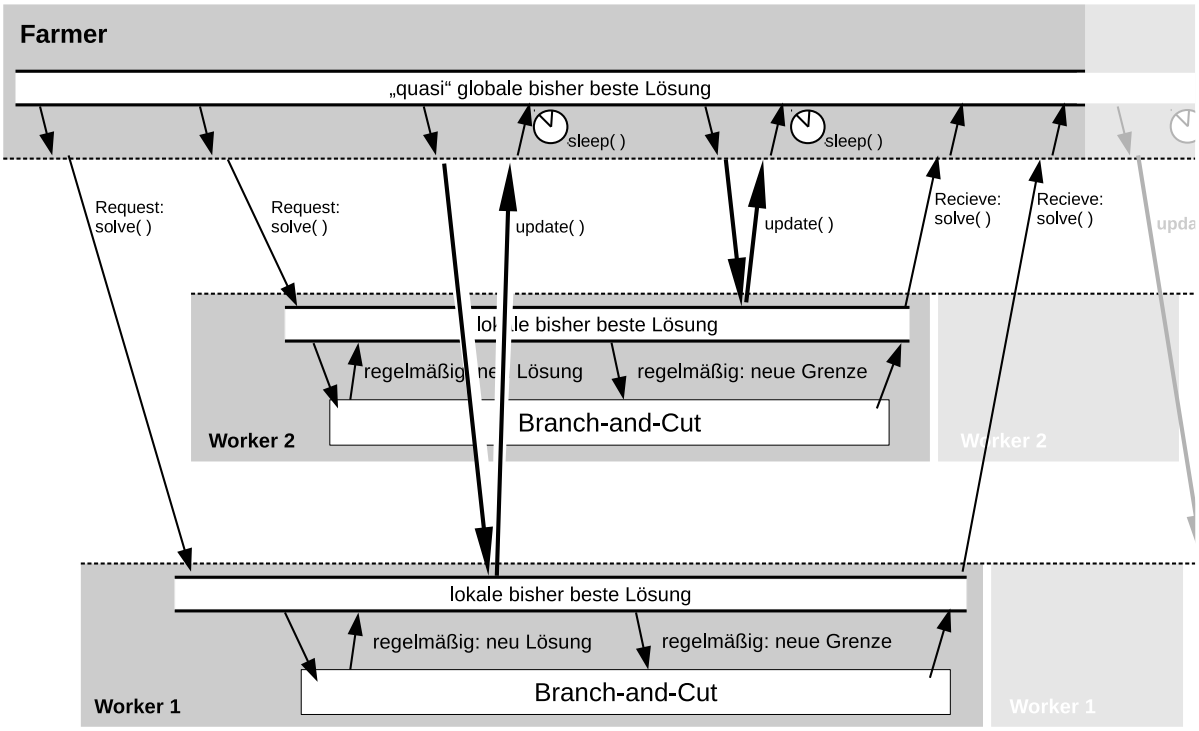

Abbildung 6: Durchreichen der bisher besten Lösung von einem *Branch-and-Cut* eines Workers zum Nächsten

4.4 Probleme der Kopplung einzelner Elemente

Die von uns implementierte dynamische Last-Balancierung sieht vor, dass ein Worker von sich aus nach einem Zeitlimit den Job, den sein Worker-Proxy aus dem *Bounded Buffer* entnommen hatte, für *abgebrochen* erklärt. Diese Situation ist nicht ganz unproblematisch, da ein abgebrochener Job in neue Jobs aufgeteilt und in den Buffer zurückgeführt werden müssen. Auf die Art wird der Worker ebenfalls zu einem Erzeuger und da der Farmer die vom Worker erzeugten Jobs berücksichtigen muss, schlüpft dieser in die Rolle des Verbrauchers. Ebenso muss der Farmer berücksichtigen, dass er nach der Verteilung aller ihm bekannten Jobs sich nicht beenden darf bzw. in unserer Implementierung keine *finalen Jobs* den Worker-Proxies geben darf. Das Konzept der *finalen Jobs* dient dazu, die Worker-Proxies, die durch den *Bounded Buffer* blockiert sind, aufzuwecken und zu beenden. Als Flag, um einen Job zu einem *finalen Job* zu machen, dient das `Job::_end`-Attribut. Für das Problem der abgebrochenen Jobs, die den Worker zum Erzeuger und den Farmer zum Konsumenten machen, ist uns kein adäquates Modell bekannt. Wir haben diese umgekehrte Erzeuger-Verbraucher-Beziehung durch eine zweite, mit einer Mutex-Semaphore geschützten Liste (`_specials`) realisiert, die in der Klasse `Filling` enthalten ist. `Filling` ist für die Erzeugung von neuen Jobs und vor allem für die Vorbelegung der Projekte in den Jobs verantwortlich. Der Farmer bekommt die Jobs von `Filling`.

Durch eine Refaktorierung von `Distributor`, `Filling` und `BoundedBuffer` sollte es in Zukunft möglich sein, eine deutlich knappere, aber weniger modulare Architektur zu bilden, da die Aufgaben der drei Klassen zu stark miteinander verwoben sind. Der Modularisierungsgedanke, um evtl. ein zukünftiges Strategie-Muster für unterschiedliche Jobs (nicht nur CBP) zu ermöglichen, ist auch in der bisher implementierten, modularen Architektur sehr schwer.

Insbesondere die Last-Balancierung lässt sich aufgrund der vielen Kriterien, nach denen man die bisherige Job-Verarbeitung bewerten kann, enorm modifizieren. Zusammen mit der Entscheidung, welche Daten eines Jobs für andere laufende oder zukünftige Jobs interessant sind, gibt es einen Modularisierungsbedarf, der durch unseren bisherigen Quellcode noch nicht erfüllt ist.

5 Ergebnisse

Als Test-Instanzen wählten wir ein paar der für CBP sehr populären[12] *Chu-and-Beasley* Instanzen, die aus Zufallszahlen mit 5, 10 und 30 Zeiträumen und 100, 250 sowie 500 Projekten bestehen. Diese Instanzen sind jeweils noch in 3 Tightness-Level unterteilt: 0.25, 0.50 und 0.75. Diese Zahlen geben exakt an, wie das Verhältnis Budget des Monats zur Summe der Kosten des Monats über alle Projekte ist. Die 0.25er Instanzen lassen sich in der Regel nicht so schnell lösen. So sind z.B. einige dieser Instanzen mit 500 Projekten bisher noch nie gelöst worden. Mehr zu den Instanzen finden Sie im Anhang auf Seite 44.

Zur Vernetzung der Rechner existiert ein 1 Gbit Ethernet-Netzwerk. Alle 20 Rechner sind in ihrem eigenem Netzwerk-Segment. Neben dem Austausch von Daten unseres verteilten Systems geschehen auch gelegentlich NFS-Dateizugriffe, die nur einen vergleichsweise geringen Traffic verursachen. Kriterien wie z.B. Transparenz und Ausfallsicherheit standen in unserem verteilten System nicht im Vordergrund, sondern eher die Möglichkeit, Rechenleistung und Speicherplatz (Arbeitsspeicher) zu verteilen und nach Bedarf zu skalieren (mehr PCs statt einen Supercomputer aufrüsten).

Auf jedem Rechner stehen 8 GB Arbeitsspeicher und eine 64bit 4-Kern Intel i5-2500 (bzw. 3550)[13] CPU mit einer 3.3 GHz Taktung zur Verfügung. Jeder Kern kann 6144 KB Cache nutzen. Die CPU-Geschwindigkeit wird unter Linux mit 26340 (bzw. 26399) BogoMips[14] angegeben.

Leider stand auf den Rechnern kein *Apache thrift* [14] zur Verfügung und die *glpk*-Version 4.38 stammt aus dem Jahr 2008. Aus diesem Grund wurden beide Bibliotheken in einer aktuellen Version (*glpk*: 4.55, *thrift*: 0.9.2) auf einem andren Rechner kompiliert und statisch gelinkt. Als Compiler wurde unter Linux *gcc*-Version 4.9.2 genutzt, und unter Windows 8.1 Visual C++ Version 12.

Die Abbildungen (7) und (8), zeigen die Laufzeiten (in Sekunden) unseres verteilten Lösungsansatzes mit *Apache thrift* und *glpk*. Leider haben wir es in der begrenzten Zeit nicht geschafft, alle mehrfach zu messen um Mittelwerte und Standardabweichung angeben zu können. Durch andere Netzwerk-Zugriffe kann die Dauer des Austauschs der Datenpakete variieren, oder es könnte das Betriebssystem wie z.B. das Scheduling den Workern weniger Priorität geben, weil andere Prozesse das Betriebssystem bevorzugt[15]. Sollte durch den Scheduler ein Job (und in der Summe werden Millionen davon erzeugt) nur leicht länger als das Zeitlimit laufen, werden durch seine Überschreitung tausende neue Jobs mit neuer Vorbelegung erzeugt. Unter *Mehrfachmessung* auf Seite 27 haben wir die Streuung der Laufzeit näher untersucht. Die Laufzeiten variierten bei den Instanzen, die wir mehrfach gemessen haben, zwischen 4 bis 10%.

Die Spalte „glpsol“ stellt in den Abbildungen die konventionelle Nutzung von *glpk*-Version

[12] siehe unter Literatur: [4], [11], [15] und [2]

[13] Es stand noch ein 2. Rechnerpool zur Verfügung mit vergleichbarer Hardware

[14] Mips = Millions of Instructions per Second. Das *Bogo* ist eine Abkürzung für „bogus“ (englisch: unecht) und soll andeuten, dass es sich nicht um ein wissenschaftlich exakt erfassten Wert handelt, weil er beim Booten gemessen wird wo die Taktfrequenz variiert.

[15] Es trat in der Vergangenheit z.B. die Situation auf, dass eine Reinigungskraft einen Stuhl auf die Tastatur stellte. Wie lange und in welcher Form reagiert das Betriebssystem darauf? Ein Pessimist kann sich hier eine Vielzahl von Möglichkeiten vorstellen.

Laufzeit [s] bei 20s Zeitlimit je Job

Instanz / Worker:	glpsol 1	Breitensuche 8	Breitensuche 16	Breitensuche 32	Tiefensuche 8	Tiefensuche 16	Tiefensuche 32
OR5x250							
0.75_1	2.248	748	374	248	725	515	340
0.75_4	900	385	265	157	407	234	159
0.50_1	3.238	535	253	167	1.007	760	488
0.50_4	3.171	816	438	294	1.141	959	290
0.50_7	3.496	722	417	242	730	419	791
OR30x100							
0.75_1	943	100	81	81	112	91	96
0.75_4	140	62	51	48	66	52	50
0.75_7	3.478	172	124	84	221	106	100
0.50_1	378.696	1.434	668	350	1.336	1.549	970
0.50_4	23.390	528	177	105	515	171	105
0.50_7	172.379	1.013	587	259	1.065	617	264
OR10x250							
0.75_1	>5,5d	13.062	6.653	3.252	23.607	11.383	3.905
0.75_4	>5,5d	50.684	2.811	1.888	78.485	2.859	1.460
0.75_7	62.946	1.137	611	330	1.160	626	1.632
0.50_1	>5,5d	40.527	22.384	10.054	-	113.108	65.986
0.50_4	Speicherüberlauf >11d	-	-	62.795	-	-	153.899
0.50_7	>5d	90.966	50.329	22.627	137.916	70.709	34.106
OR5x500							
0.75_1	3.175	2.956	1.526	810	3.052	1.587	852
0.75_4	>5,5d	22.384	11.409	5.594	32.817	16.503	8.760
0.75_7	30.366	6.114	3.195	1.864	12.422	5.519	2.858

Speedup

	Breitensuche 8	Breitensuche 16	Breitensuche 32	Tiefensuche 8	Tiefensuche 16	Tiefensuche 32
1	3	6	9	3	4	7
1	2	3	6	2	4	6
1	6	13	19	3	4	7
1	4	7	11	3	3	11
1	5	8	14	5	8	4
1	9	12	12	8	10	10
1	2	3	3	2	3	3
1	20	28	41	16	33	35
1	264	567	1.082	283	244	390
1	44	132	223	45	137	223
1	170	294	666	162	279	653
1	36	71	146	20	42	122
1	9	169	252	6	166	325
1	55	103	191	54	101	39
1	12	21	47	-	4	7
1	-	-	15	-	-	6
1	5	9	19	3	6	13
1	1	2	4	1	2	4
1	21	42	85	14	29	54
1	5	10	16	2	6	11

3h = 10.800s 10h = 36.000s 4,5d = 108h

1d = 86.400s 2d = 172.800s 108h = 388.800s

= super-linear

Abbildung 7: Verteilung des Capital Budgeting Problems mit *glpk* und *Apache thrift*. Jobs werden nach 20 Sekunden abgebrochen und dann neu verteilt. Die linke Tabelle enthält die Laufzeit und die rechte den Speedup.

Laufzeit [s] bei 40s Zeitlimit je Job

Instanz / Worker:	glpsol 1	Breitensuche 8	Breitensuche 16	Breitensuche 32	Tiefensuche 8	Tiefensuche 16	Tiefensuche 32
OR5x250							
0.75_1	2.248	856	587	409	883	518	385
0.75_4	900	507	343	265	511	344	274
0.50_1	3.238	595	340	220	1.437	997	213
0.50_4	3.171	1.034	627	492	1.501	1.039	454
0.50_7	3.496	939	519	425	971	563	424
OR30x100							
0.75_1	943	151	144	122	175	120	122
0.75_4	140	98	90	89	113	90	89
0.75_7	3.478	228	150	131	240	144	134
0.50_1	378.696	1.798	841	428	1.472	1.801	613
0.50_4	23.390	650	216	146	635	205	135
0.50_7	172.379	1.427	726	307	1.332	720	356
OR10x250							
0.75_1	>5,5d	17.380	8.205	4.533	27.238	12.581	4.361
0.75_4	>5,5d	57.131	3.410	1.774	95.347	3.390	1.776
0.75_7	62.946	1.308	714	412	1.332	729	423
OR5x500							
0.75_1	3.175	3.648	1.866	1.045	3.798	2.065	1.224
0.75_4	>5,5d	25.175	12.814	6.688	39.370	19.057	10.260
0.75_7	30.366	8.660	4.283	1.328	15.211	6.612	2.504

Laufzeit [s] bei 90s Zeitlimit je Job

Breitensuche 8	Breitensuche 16	Breitensuche 32	Tiefensuche 8	Tiefensuche 16	Tiefensuche 32
1.467	872	733	1.348	931	725
649	526	521	733	528	500
789	459	399	824	467	391
1.470	966	807	1.546	932	786
1.300	970	902	1.326	941	881
249	229	222	219	224	211
144	133	138	138	131	128
329	262	243	405	254	248
2.499	946	694	1.935	990	580
512	318	278	515	297	268
1.786	801	450	1.564	787	511
22.298	10.824	5.560	30.376	14.514	5.775
71.611	4.514	2.333	114.635	4.551	2.396
1.732	965	596	1.779	987	590
4.777	2.636	1.712	5.184	2.786	1.641
32.719	16.824	8.617	54.395	23.967	10.106
7.038	4.120	2.337	6.356	5.763	2.163

Abbildung 8: Jobs werden nach 40 bzw. 90 Sekunden abgebrochen und dann neu verteilt.

4.55 dar, da es von sich aus weder mehrere Kerne nutzen kann, noch eine Netzwerk-Unterstützung hat. Fairerweise haben wir Presolver und Proxy-Heuristik aktiviert, um einen echten Vergleich zu unserer Lösung zu haben.

Da unser Programm eine Vielzahl von Möglichkeiten bietet, die jedoch hier den Rahmen sprengen würden, haben wir uns in den Tabellen (Abbildung 7) und (Abbildung 8) nur für die Breiten- und Tiefensuche mit einem Zeitlimit je Job von 20, 40 und 90 Sekunden entschieden. Eine Erhöhung dieses Limits findet nicht statt, sondern nur eine Erhöhung um 10 Projekte (= 1024 neue Jobs zur Abarbeitung) bei der Vorbelegung. Der Presolver setzt ein, wenn 50% der Projekte vorbelegt sind und das Zeitlimit wird bei 85% der Vorbelegung nicht mehr gesetzt. Die Größe der ersten Vorbelegung orientiert sich an der Anzahl der zur Verfügung stehenden Worker.

Instanz / Worker:	**Zeiträume**	**Projekte**	**glpsol 1**	**Breitensuche (200ms * 1,5) 8**	**16**	**32**
OR5x250-0.75_1	5	250	2.248	1.375	701	375
OR5x250-0.75_4	5	250	900	382	191	132
OR5x250-0.50_1	5	250	3.238	2.331	818	256
OR5x250-0.50_4	5	250	3.171	1.279	738	426
OR5x250-0.50_7	5	250	3.496	1.734	826	477
OR30x100-0.75_1	30	100	943	128	76	178
OR30x100-0.75_4	30	100	140	69	45	36
OR30x100-0.75_7	30	100	3.478	164	147	138
OR30x100-0.50_1	30	100	378.696	1.549	844	803
OR30x100-0.50_4	30	100	23.390	538	199	164
OR30x100-0.50_7	30	100	172.379	979	503	515
OR10x250-0.75_1	10	250	>5,5d	173.684	> 290.000	> 290.000
OR10x250-0.75_7	10	250	62.946	1.389	-	-

```
3 h =  10.800s
10 h =  36.000s
1 d =  86.400s
2 d = 172.800s
4,5d =     108h
108h = 388.800s
```

Abbildung 9: Messung in Sekunden: Jobs werden anfänglich nach 200 Millisekunden abgebrochen und dann nach jedem Abbruch mit 1,5-fach höherem Zeitlimit neu verteilt.

Die Abbildung (9) zeigt eine vollkommen andere Konfiguration, bei der das Zeitlimit 200ms ist, welches jeweils um den Faktor 1.5 erhöht wird. Die erste Vorbelegung besteht aus 18 Projekten (2^{18} Jobs) und erhöht sich immer um 5 wenn das Zeitlimit erreicht ist. Sind jedoch mehr als 40% der Projekte vorbelegt, dürfen die Jobs beliebig lange laufen. Der Presolver setzt bereits bei einer Pfadtiefe von 30% ein. Bei größeren Instanzen hat sich diese Konfiguration als so ungünstig erwiesen, so dass wir keine weiteren Messungen gemacht haben.

Im Laufe der Messungen haben wir uns aus Zeitnot entschieden, bestimmte Konfigurationen nicht weiter zu messen. Es zeigte sich zum Beispiel, dass die konventionelle 1-Worker-Version `glpsol` die 8GB Arbeitsspeicher des Rechners überschritt und sich daher frühzeitig beendete. Andere Instanzen waren nach 11 Tagen immer noch nicht fertig. Ebenso zeigte sich, dass die Laufzeiten mit den Zeitlimits von 40 und 90 Sekunden tendenziell schlechter waren, als die der 20-Sekunden-Versionen. Die Messergebnisse sind trotzdem sinnvoll, da uns nicht interessiert, ob die konventionelle 1-Worker-Version 7 Tage zur Berechnung braucht, während der 32-fache CPU-Einsatz (8 Rechner) „nur“ einen halben Tag braucht (und nicht 7/32 Tage). Grundsätzlich sieht man, dass die Verteilung z.B. auf nur 8 Worker bereits eine enorme Verkürzung der Laufzeiten bringt – manchmal sogar um mehr als das 8-fache (Abbildung 7; Super-linearer Speedup ist orange markiert). Ähnlich

gut skaliert die Anzahl der Worker, so dass eine Verdopplung der Worker zum Teil eine Halbierung der Laufzeit bedeutet – und genau diese Verkürzung der Laufzeit wollten wir erreichen!

64 Worker

Mit 64 Workern und leicht modifizierten Parametern haben wir ein paar größere Instanzen lösen können, die mit `glpsol` nach 5 Tagen noch nicht fertig waren (OR10x500-0.75_4 brach z.B. nach 5 Tagen wegen Überschreiten des Arbeitsspeichers ab):

Instanz	Zeiträume	Projekte	Laufzeit
OR5x500-0.50_1	5	500	4.399
OR10x500-0.75_1	10	500	97.199
OR10x500-0.75_4	10	500	353.366

5.1 Einflüsse der Parameter

Die folgenden Abbildungen sind zusammen mit einer Reihe von Bash-Shell-Skripten, `gnuplot`, dem Kommandozeilen-Tool `awk` sowie der Umleitung von `stdout` und `stderr` in Logging-Dateien entstanden. Zum Starten, Überwachen und Anlegen von Benchmark-Szenarien sind ebenfalls Bash-Shell-Skripte, `ssh`, `nohub` und eine reformatierte Ausgabe von `ps` unter Linux zum Einsatz gekommen. Jede Abbildung soll exemplarisch den Einfluss bei der Änderung der Kommandozeilen-Parameter zeigen. Sollten die Parameter nicht explizit von uns angegeben worden sein, so sind sie:

Wert	8 Worker	16 Worker	32 Worker
initiales K	5	6	8
K-Erweiterung	10	10	10
Zeitlimit	20s	20s	20s
Zeitlimit-Faktor	1	1	1
Suche	breit	breit	breit
Presolver ab	50%	50%	50%
kein Zeitlimit ab	85%	85%	85%
Sync.-Intervall	40ms	40ms	40ms
Transport der oberen Grenze	ja	ja	ja

In den nächsten Kapiteln werden sie bei vielen Messreihen den Hinweis lesen, dass weiteren Messungen sinnvoll sind. Leider konnte ich aus gesundheitlichen Gründen nicht weitere Messungen an der Hochschule durchführen und im späteren Verlauf dieser Arbeit wurden

Wartungsarbeiten und System-Updates an den Rechnern durchgeführt, so dass falls noch Zeit vor der Abgabe bestand keine vergleichbaren Messergebnisse zustande gekommen wären und ein großer Teil hätte neu gemessen werden müssen. Auch die jetzt vorliegenden Ergebnisse sind unter großem Stress entstanden und zeigen daher ein paar Messlücken. Trotzdem konnte überall eine auswertbare Größe gemessen werden.

5.1.1 Mehrfachmessung

Durch das mehrfache Messen einer von uns willkürlich ausgewählten *Chu-and-Beasley* Instanz wollten wir untersuchen, wie stark die Laufzeit unseres verteilten System variiert. Die Abbildungen (10) und (11) zeigen die Ergebnisse dieser Messung. Durch den Scheduler des Betriebssystems können die Jobs auf den Workern in ihrer Laufzeit leicht variieren, da z.B. Zeit für einen anderen Thread (z.B. die `update()` Methode) zur Verfügung gestellt werden muss. So kommt es vereinzelt zu Zeitlimit-Überschreitungen von Jobs, durch die dann neue Jobs erzeugt werden. Die Abbildung (11) zeigt, wie oft Jobs Zeitlimits überschreiten. Sollte eine Überschreitung nur einmal mehr passieren als bei einer anderen Messung, dann werden bei dieser Messreihe 1024 weitere Jobs erzeugt. Das ist der Grund, warum die Laufzeiten auch in den weiteren Messungen um bis zu 10 % variieren.

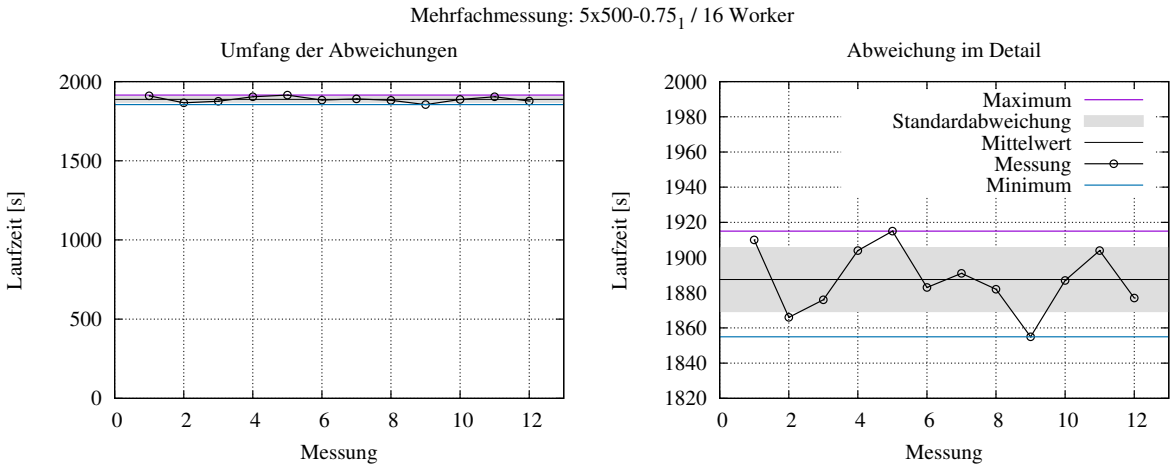

Abbildung 10: Eine Instanz wurde 12 mal gestartet und die Laufzeit in Sekunden gemessen.

5.1.2 Einfluss der Vorbelegung

Die Abbildung (12) zeigt 2 Messreihen, bei denen wir die Vorbelegung der Projekte unterschiedlich groß wählten. Als wir noch keine Last-Balancierung durch Zeitlimit und automatischer Erhöhung der Vorbelegung K hatten, war der Einfluss der Größe der Vorbelegung sehr stark. Zwar hatte damals eine größere Vorbelegung nicht zwingend die Laufzeit verringert[16], aber ein kleiner Einfluss ($K \pm 3$) war für die Laufzeit entweder ver-

[16] Kleinere Teilbäume bearbeiten heißt im *Brunch-and-Cut* nicht automatisch eine kürzere Laufzeit. Im Kapitel *Schwachstellen* sprechen wir dies nochmal an.

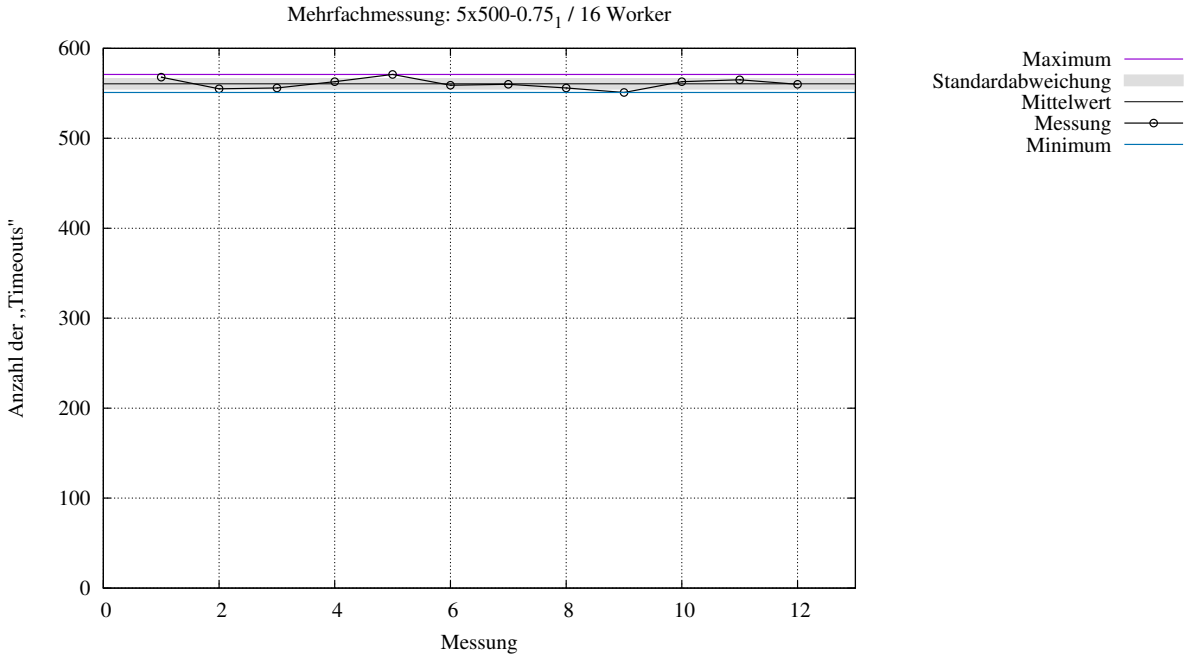

Abbildung 11: Eine Instanz wurde 12 mal gestartet und die Anzahl der Zeitlimit-Überschreitungen (Timeouts) gemessen.

heerend (z.B. 20 mal längere Laufzeit) oder extrem positiv (z.B. 1/10 der Laufzeit). Diese Messungen von damals sind hier nicht dargestellt. Für Instanzen, die in nur wenigen Sekunden bis Minuten gelöst werden können, ist trotz Last-Balancierung der Einfluss der ersten Vorbelegung in der aktuellen Version relevant, da in diesen Fällen innerhalb des Zeitlimits von 20s bereits Jobs mit dieser initialen Vorbelegung fertig werden.

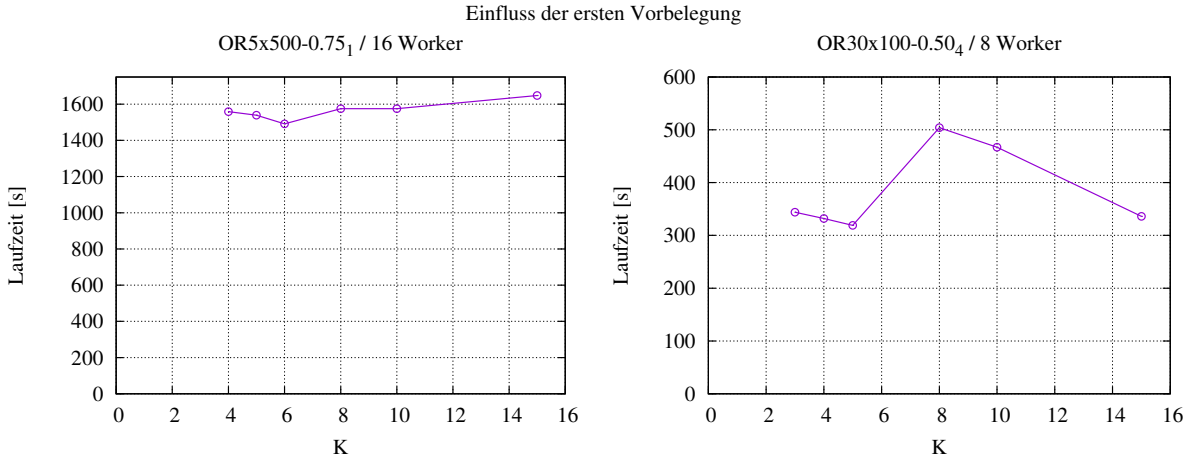

Abbildung 12: Zwei Messreihen, bei denen wir die initiale Vorbelegung K veränderten.

5.1.3 Größe der neuen Vorbelegung

Beim Erreichen des gesetzten Zeitlimits wird beim Worker ein Job abgebrochen. Diese abgebrochenen Jobs werden mit einer neuen, größeren Vorbelegung zu neuen Jobs. Bei

einer Erweiterung von z.B. 10 Projekten, werden 1024 neue Jobs erzeugt. Die Abbildung (13) zeigt 2 Messreihen, bei denen wir die Größe der neuen Vorbelegung variiert haben (K ist bei beiden initial 5). Es zeigt sich, je schneller wir beim Erreichen eines Zeitlimits im Entscheidungsbaum nach unten gehen, desto kürzer ist die Laufzeit. Leider haben wir diese Messungen aus zeitlichen Gründen nicht mehr mit 4, 8, 16 und 32 Workern und einer feineren Einteilung der K-Erweiterung gemacht. Mit der jetzigen Anzahl kann man lediglich einen Verlauf erahnen.

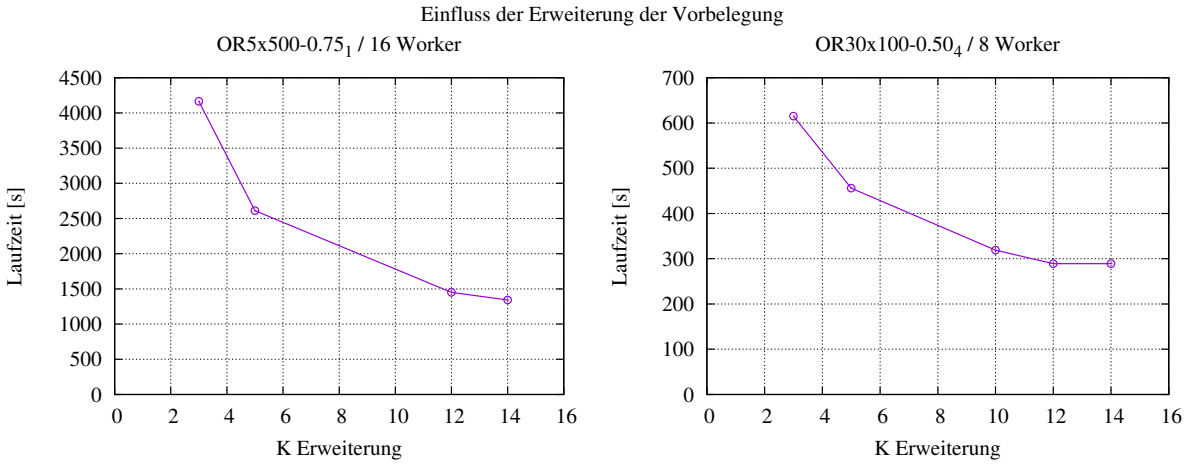

Abbildung 13: Zwei Messreihen, bei denen wir die Größe der Erweiterung der Vorbelegung veränderten.

5.1.4 Zu kleines Zeitlimit

Wie bereits erwähnt, ist eine Konfiguration mit einem Zeitlimit für die Jobs unter einer Sekunde bei größeren Instanzen nicht gut. Es kommt bis zur vorgegebenen Pfadtiefe, ab dem das Zeitlimit aufgehoben ist, ständig zu Überschreitungen des Zeitlimits. Dies führt zu einer beinahe unbalancierten Verteilung der Jobs, die darin gipfelt, dass nur noch ein Job für sehr lange Zeit ein Teilproblem abarbeitet. Die Abbildung (14) ist durch ein regelmäßiges schreiben der Werte (alle 30s) in eine Log-Datei entstanden [17] und zeigt so einen Extremfall von verhungernden Workern, was bei einem kleinen Zeitlimit sehr oft auftrat.

Die Abbildung (15) zeigt beispielhaft den Einfluss des Zeitlimits bei einer Instanz, die mit 16 Workern[18] etwas über 20 Minuten rechnet und einer Instanz, die mit 8 Workern[19] ca 5 Minuten rechnet. In beiden Fällen ist ein sehr kleines Zeitlimit nicht gut. Auch hier wären weitere Messungen wünschenswert mit einer Variation der Anzahl der Worker und weiteren Zeitlimits.

[17] initiales K: 18; K-Erweiterung: 5; Zeitlimit: 200ms * 1,5; Presolver bei: 30%; kein Zeitlimit ab: 40%
[18] K-Erweiterung: 15; Faktor: 1; Breitensuche
[19] K-Erweiterung: 12; Faktor: 0,75; Tiefensuche

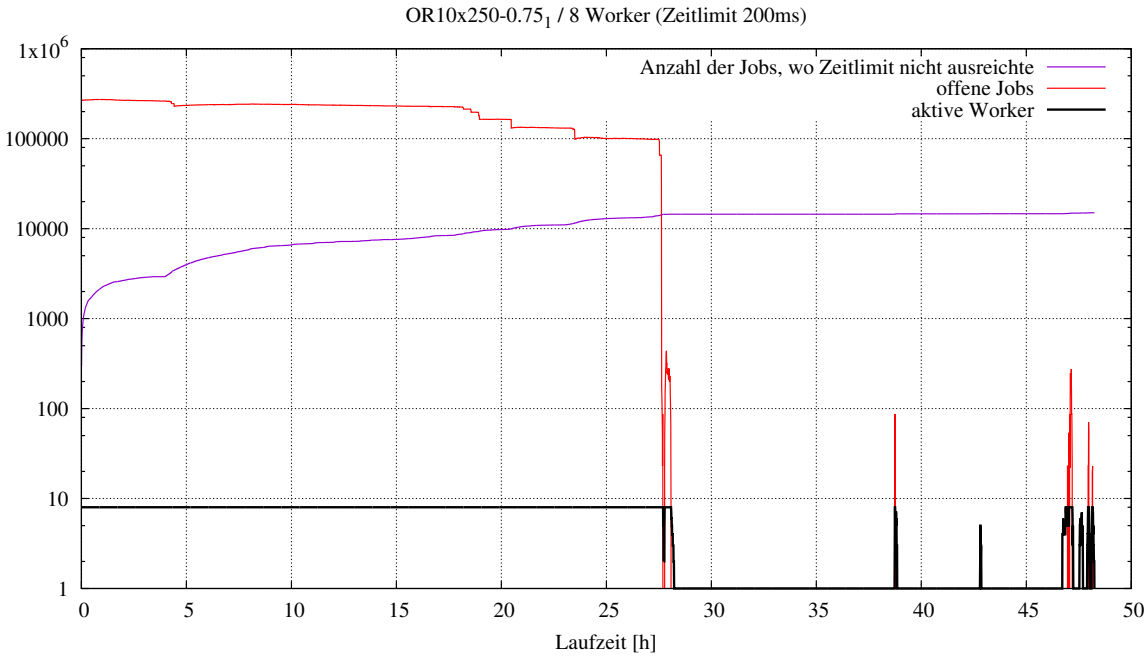

Abbildung 14: Eine Messreihe mit unpassendem, zu kleinem Zeitlimit (200ms) für diese Instanz. Die Anzahl der aktiven Worker bricht über lange Zeit ein.

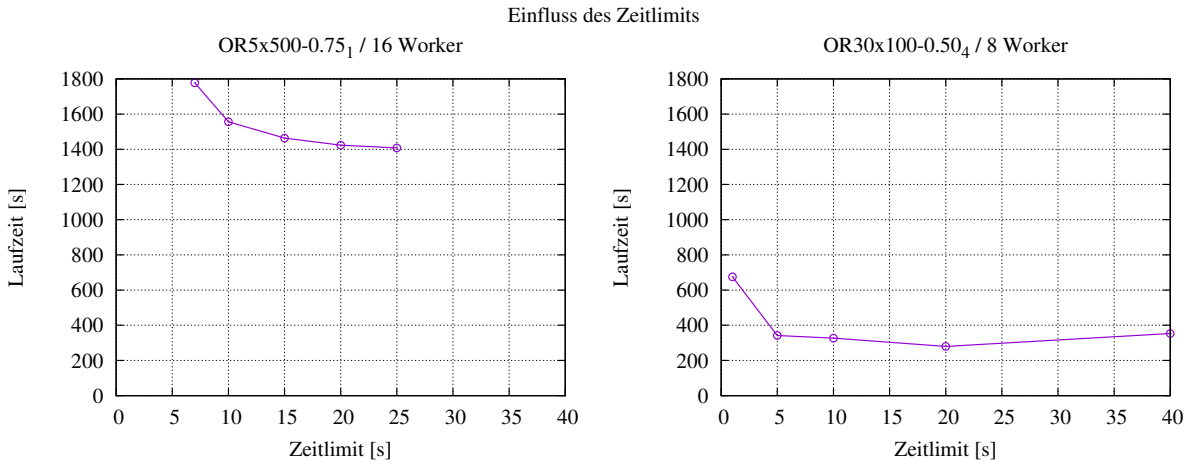

Abbildung 15: Zwei Messreihen, bei denen wir die Größe des Zeitlimits veränderten.

5.1.5 Modifikation des Zeitlimits

Die Abbildung (16) zeigt, dass unsere Überlegungen zur Anpassung des Zeitlimits von 20s auf keinen fruchtbaren Boden traf. Ist einmal ein gutes Zeitlimit gefunden, macht eine stufenweise Vergrößerung (Faktor > 1) oder Verkleinerung ($0 <$ Faktor < 1) je Größe der Vorbelegung keinen Sinn. Leider sind wir nicht dazu gekommen zu testen, ob man alternativ statt neue Jobs mit größerer Vorbelegung zwischendurch den selben Job zuerst mit einem z.B. doppelt so großem Zeitlimit starten sollte. Dies könnte die Anzahl der zu bearbeitenden Jobs reduzieren und könnte vor Speicherplatzproblemen beim Farmer schützen. Besonders diese Messreihe war schlecht gewählt und müsste wiederholt werden. Bei der Messung mit 16 Workern reichten 20s bei einer Verkleinerung des Zeitlimits nicht aus. So ist hier **nur** beim Faktor 0,75 mit 100s Zeitlimit gestartet worden, was die Vergleichbarkeit mit den restlichen Werten anzweifeln lässt.

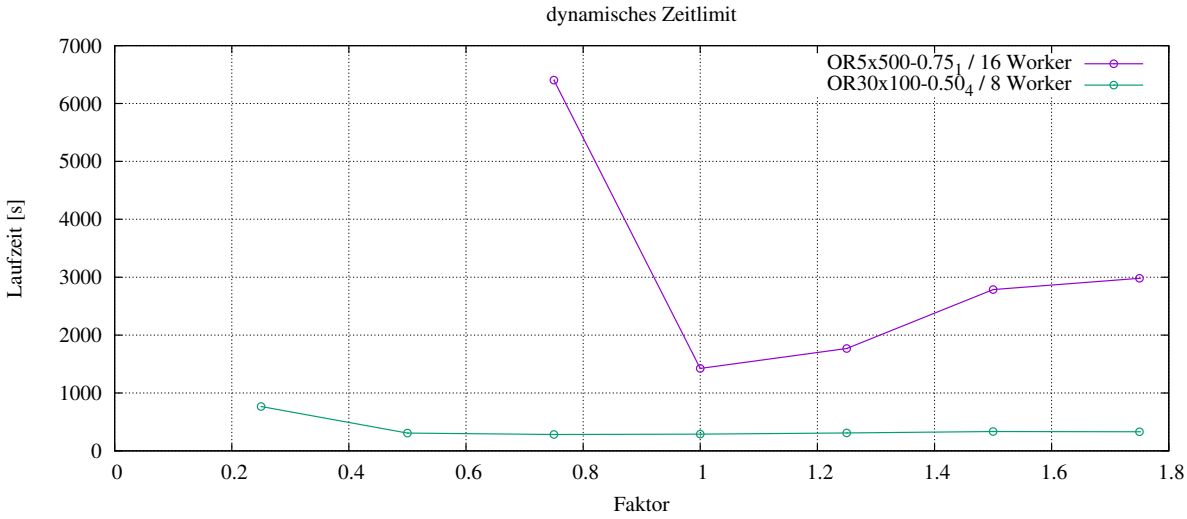

Abbildung 16: Zwei Messreihen, bei denen die Größe des Zeitlimits für neue Jobs über einen Faktor verändert wird, sobald das Zeitlimit für einen Job überschritten wird. Die Instanz mit 16 Worker arbeitet mit einer Erweiterung von K_{erw}=15, die andere mit K_{erw}=12.

5.1.6 Aufheben des Zeitlimits

Da wir nicht Gefahr laufen wollten den Arbeitsspeicher des Farmers mit Milliarden von Jobs und riesigen Vorbelegungen zu Überfluten, haben wir als Parameter die Eingabe eines Prozentsatzes ermöglicht. Dieser Prozentsatz gibt an, ab welcher Größe der Vorbelegung das Zeitlimit aufgehoben ist. Diese Jobs, wo z.B. 85% der Projekte bereits vorbelegt sind, können dann beliebig lange laufen. Bei einem zu kleinen Zeitlimit kann es aber passieren, dass Worker „verhungern“ (Abbildung 14). Dies könnte man in einer Weiterentwicklung unserer Software verhindern, wenn man dem Farmer eine aktivere Rolle bei der Kontrolle der Worker gibt. So wäre es möglich, dass bei einer schlechten Auslastung der Worker solche sehr lang laufenden Jobs vom Farmer trotzdem abgebrochen und mit größerer Vorbelegung neu verteilt werden.

Die Abbildung (17) zeigt bei 2 Instanzen, wie der Einfluss einer zu frühen Aufhebung des Zeitlimits ist. Auch hier wäre in Zukunft eine feinmaschigere Messung sinnvoll, da die Messung mit 8 Workern[20] bei 50% und die andere Messung mit 16 Workern[21] bei 20% und 30% große Änderungen der Laufzeit zeigen und Details in den Bereichen wünschenswert sind (genauer Verlauf, Maxima?, Minima?).

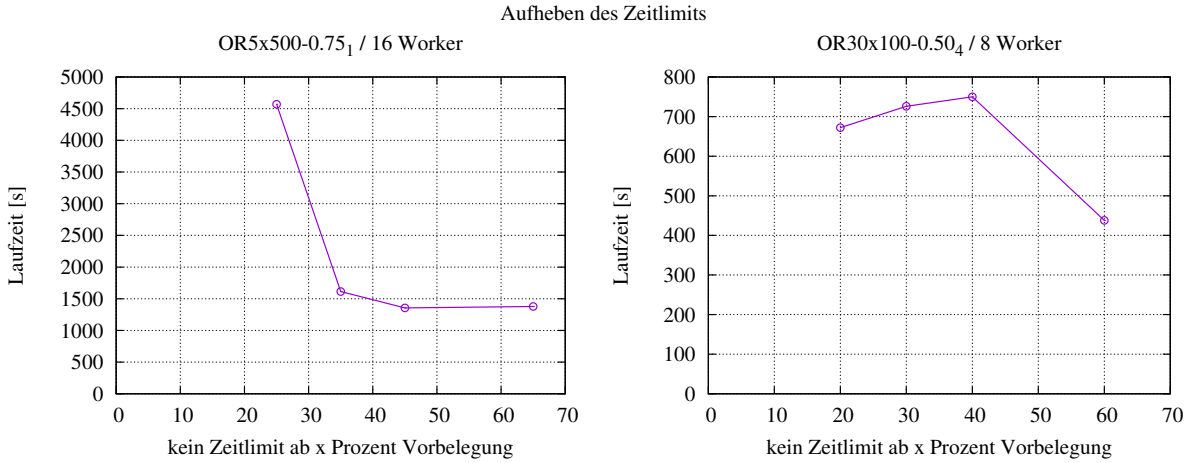

Abbildung 17: Zwei Messreihen, bei denen wir den prozentualen Anteil der vorbelegten Projekte, ab dem das Zeitlimit aufgehoben wird, angepasst haben.

5.1.7 Einfluss des Synchronisierungs-Intervalls

Abbildung (18) soll den Einfluss eines veränderten Sync.-Intervalls (die `sleep()`-Dauer in Abbildung 6, Seite 21) für die „quasi“ globale bisher beste Lösung zeigen. Ein Einfluss ist im Grunde nicht erkennbar. Diese Messung hätten wir besser nochmal mit deutlich höheren Intervallen machen sollen, da eine Erhöhung eine Annäherung an eine Abschaltung dieser Funktion ist. Die Schwankungen der Laufzeit durch die Intervallgröße geht sehr wahrscheinlich in der Streuung der Messung unter. Eine Abschaltung der Synchronisation haben wir in einer anderen Messung untersucht (leider auch eine andere Instanz).

5.1.8 Zu früher Presolver

Der Einfluss des Presolvers von *glpk* ist doch fragwürdig. Es hat sich zwar gezeigt, dass er bei Instanzen, die mit `glpsol` über ein paar Minuten laufen, Vorteile bringt, da wir aber ein Zeitlimit von 20 Sekunden für die vorliegenden Test-Instanzen für optimal halten, ist ein Einsatz des Presolvers erst bei einer hohen Vorbelegung bzw. wenn das Zeitlimit ignoriert wird sinnvoll. Die Abbildung (19) soll den Einfluss des Presolvers zeigen, und wie er ab einem bestimmten Prozentsatz an vorbelegten Projekten aktiv wird und die Laufzeit beeinflusst. Diese Messreihe soll sich nur auf einen sehr frühen Einsatz des Presolvers beziehen. Leider sind von uns nur wenige Messungen gemacht worden. Aus den

[20] initiales K: 5; K-Erweiterung: 12; Zeitlimit: 5s; Faktor: 0,75; Presolver setzt sofort ein, wenn Zeitlimit aufgehoben ist; Tiefensuche

[21] initiales K: 5; K-Erweiterung: 10; Zeitlimit: 10s; Faktor: 1; Presolver setzt sofort ein, wenn Zeitlimit aufgehoben ist; Breitensuche

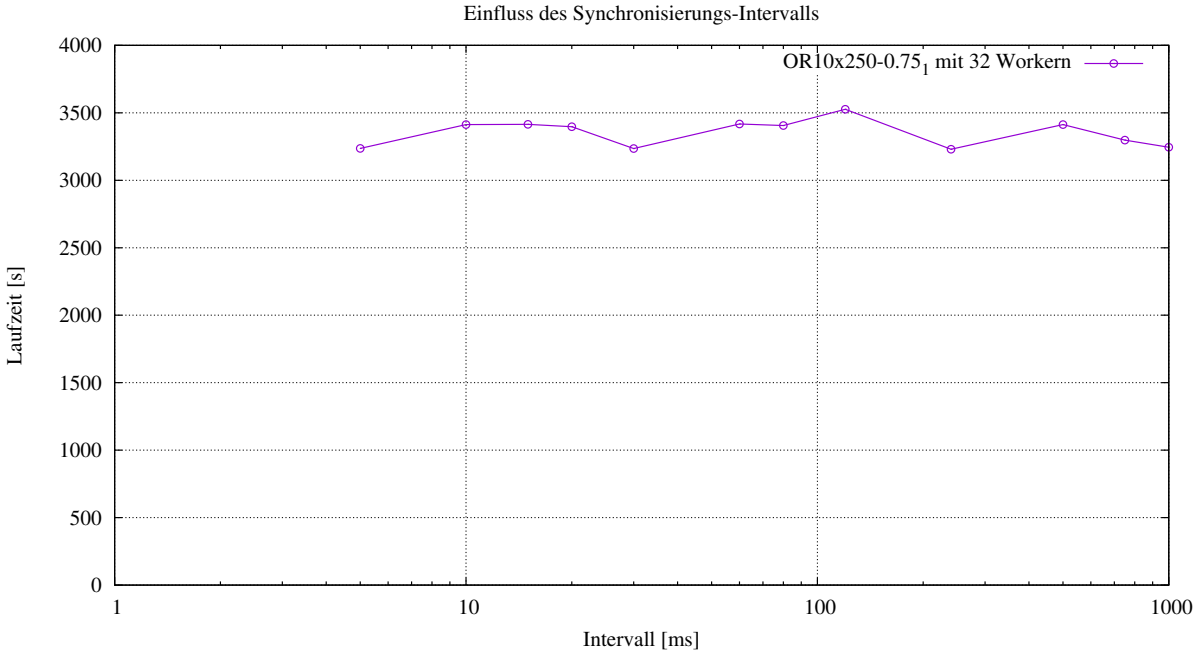

Abbildung 18: Der Einfluss des Sync.-Intervalls ist bei der Wahl der gemessenen Intervall-Zeiten nicht erkennbar.

Messdaten geht hervor (nicht in der Abbildung dargestellt!), dass bei allen Messungen nie einer Vorbelegung (>85%) erreicht wurde, bei der das Zeitlimit aufgehoben wird[22]. Die Maximale Belegung geht bei der Instanz mit 500 Projekten nur bis 261 (52% Vorbelegung) und bei der mit 100 Projekten bis 41 (also 41% Vorbelegung). Beim Messwert bei 70% (Messung mit 8 und 16 Workern) und 50% (Messung mit 8 Workern) ist der Presolver gar nicht aktiv geworden, was eine überaus gute Laufzeit zur Folge hatte. Da bei über 50% Vorbelegung aus unserer Sicht der Presolver in der Berechnung erst spät einsetzt, interessieren diese Messwerte bei dieser Untersuchung „zu *früher* Presolver“ nicht. Da aber die Laufzeiten bei diesen „Presolver-freien“ Berechnungen so gut waren, haben wir später noch eine Messreihe gemacht, bei der der Presolver komplett abgeschaltet ist.

5.1.9 Vorsortierung

Die Sortierung der Projekte zu Anfang lässt sich streng genommen nicht über einen Programmparameter steuern. Mit einem kleinen Eingriff im Programmcode konnten wir aber auch diesen Einfluss testen. Wir haben zur Sicherheit gleich mehrere Messungen gemacht, um einen Einfluss durch den Scheduler auf die Laufzeit oder andere Einflüsse ausschließen zu können. Die Abbildung (20) zeigt, dass eine Vorsortierung der Projekte, so dass Projekte mit geringen Kosten pro Profit zuerst vorbelegt werden, einen positiven Einfluss auf die Laufzeit hat.

[22] Es sind wirklich sehr viele Daten, die nicht gut an dieser Stelle gezeigt werden können.

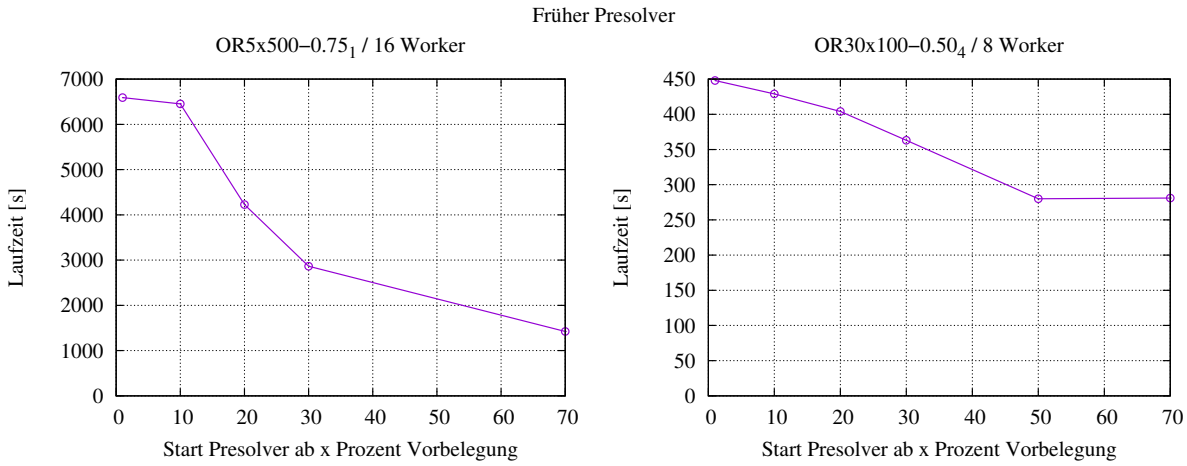

Abbildung 19: Setzt der Presolver von *glpk* bei einem zu großem Teilbaum ein (kleiner Prozentsatz ist vorbelegt) ist der Einfluss des Presolvers auf die Laufzeit negativ zu bewerten.

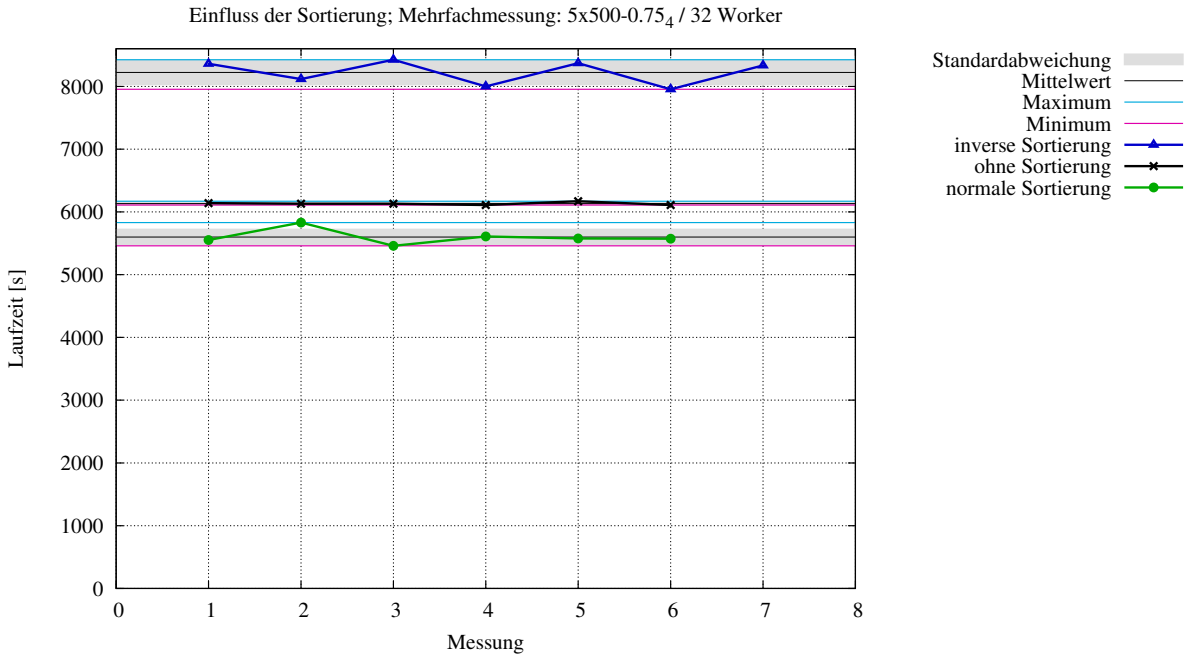

Abbildung 20: Einfluss auf die Laufzeit ohne, mit und inverser Sortierung der Projekte nach der Summe ihrer Kosten pro Profit.

5.1.10 Kein Abgleich der unteren Grenze

Die untere Grenze bzw. die bisher beste Lösung wird in den Workern regelmäßig angeglichen. Die Abbildung (21) zeigt ein Beispiel, welchen Einfluss die Abschaltung auf die Laufzeit und der Menge der Zeitlimit-Überschreitungen hat. Wie erwartet, ist das nicht gut, weil so unser verteilter *Branch-and-Bound*-Algorithmus abgeschaltet ist. Während bei der dargestellten Instanz mit Abgleich der unteren Grenze unsere Software eine Laufzeit von **1423 Sekunden** hat, liegt die Laufzeit ohne diesen Abgleich bei **4834 Sekunden!** Hätten wir noch den Einfluss des Sync.-Intervalls mit deutlich größeren Intervallen bei diesen Instanzen gemessen, gehen wir von einer Annäherung an diese Laufzeiten ohne Synchronisation der bisher besten Lösung aus.

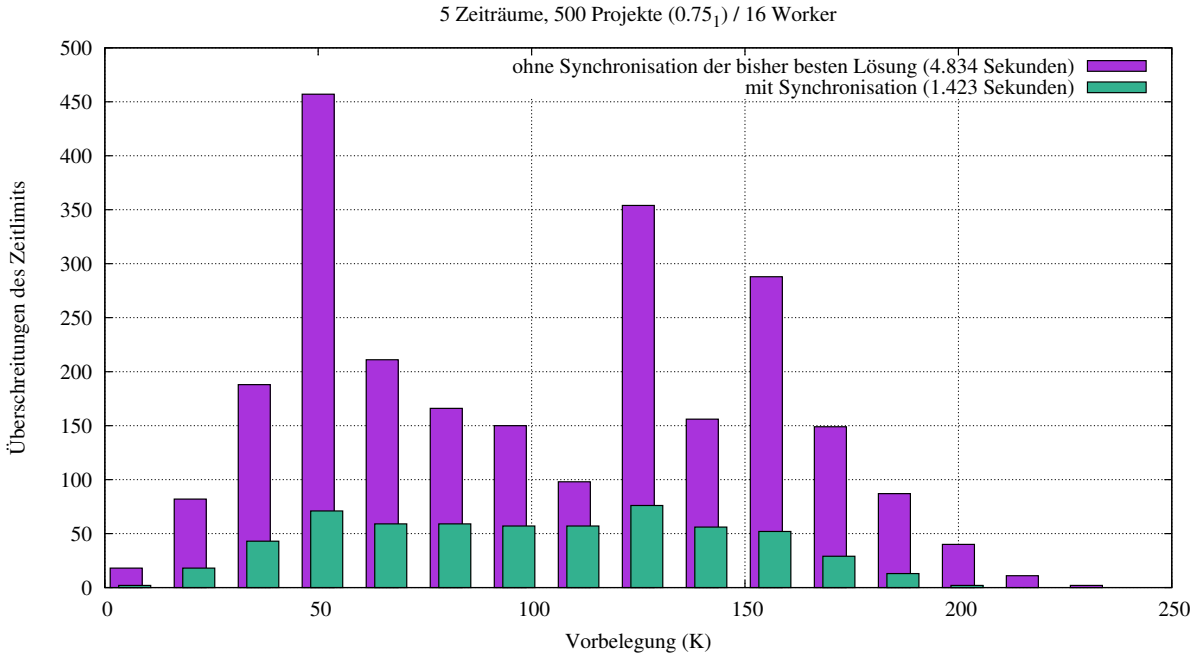

Abbildung 21: Schaltet man die Synchronisation der bisher besten Lösung zwischen den Workern ab, hat dies einen sehr negativen Einfluss auf die Laufzeit und die Anzahl der Zeitlimit-Überschreitungen.

5.1.11 Transport der oberen Grenze in neue Jobs

Die obere Grenze, die ein Worker zu einer Vorbelegung ermittelt hat, wird beim Erreichen des Zeitlimits an die neu erzeugten Jobs weitergegeben. Die Abbildung (22) zeigt ein Beispiel, welchen Einfluss eine Abschaltung dieser Funktion auf die Laufzeit hat. Zwar streuen diese Werte sehr stark, liegen aber trotzdem noch so weit auseinander, dass man das Weitergeben der oberen Grenze als leichten Vorteil in der Laufzeit ansehen kann. Um den Einfluss genauer zu untersuchen, sollten in Zukunft auch weitere Messergebnisse mit anderen Instanzen und einer anderen Anzahl an Workern gemessen werden.

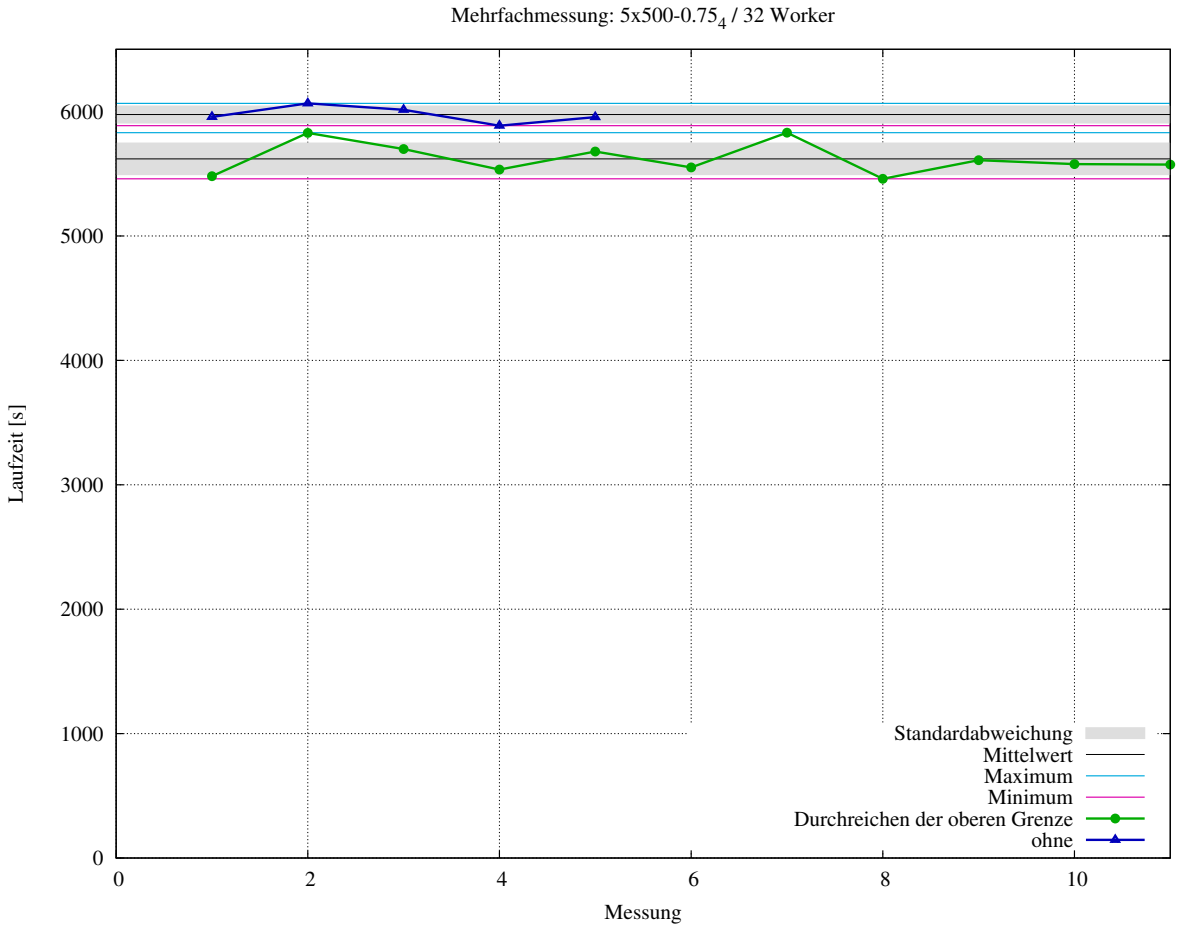

Abbildung 22: Einfluss auf die Laufzeit, wenn die obere Schranke nicht ermittelt und an neue Jobs übergeben wird.

5.1.12 Abschaltung des Presolvers

Um den positiven Einfluss des Presolvers auf die Laufzeit zu erkennen, haben wir ihn bei einer Instanz mit über einer Stunde Laufzeit abgeschaltet. Diese Instanz haben wir dann mehrfach mit und ohne Presolver gemessen. Die Ergebnisse sind in Abbildung (23) zusammengestellt. Dabei ist herausgekommen, dass der Einfluss des Presolvers nicht erkennbar ist und in der Streuung der Messdaten untergeht. Der Mittelwert der Messdaten ließe sogar zu, dass der Einsatz des Presolvers bei Jobs ab 50% Vorbelegung (in diesem Fall über 250 Projekte) eher nachteilig ist. Ein Blick auf die Messdaten (hier nicht gezeigt, weil zu umfangreich) zeigt, dass bei allen Messungen beider Messreihen (mit und ohne Presolver) die Vorbelegung bei knapp über 50% liegt. Es sieht ganz so aus, als wenn eine komplizierte Heuristik im Mittel auch nicht bessere Laufzeiten bringt, als eine simple Vorbelegung der Projekte und eine Verteilung dieser neuen Teil-Instanzen im Netzwerk. Wie sieht dieses Verhalten bei einer anderen Vergrößerung von K aus? Wie verhält es sich mit mehreren Workern? Wie ist die Abhängigkeit von der Vorbelegung in % zu bewerten sowie ein Nutzung von deutlich höheren Zeitlimits? Das sind leider noch offene Fragen die mit viele weiteren Messungen in Zukunft geklärt werden müssen.

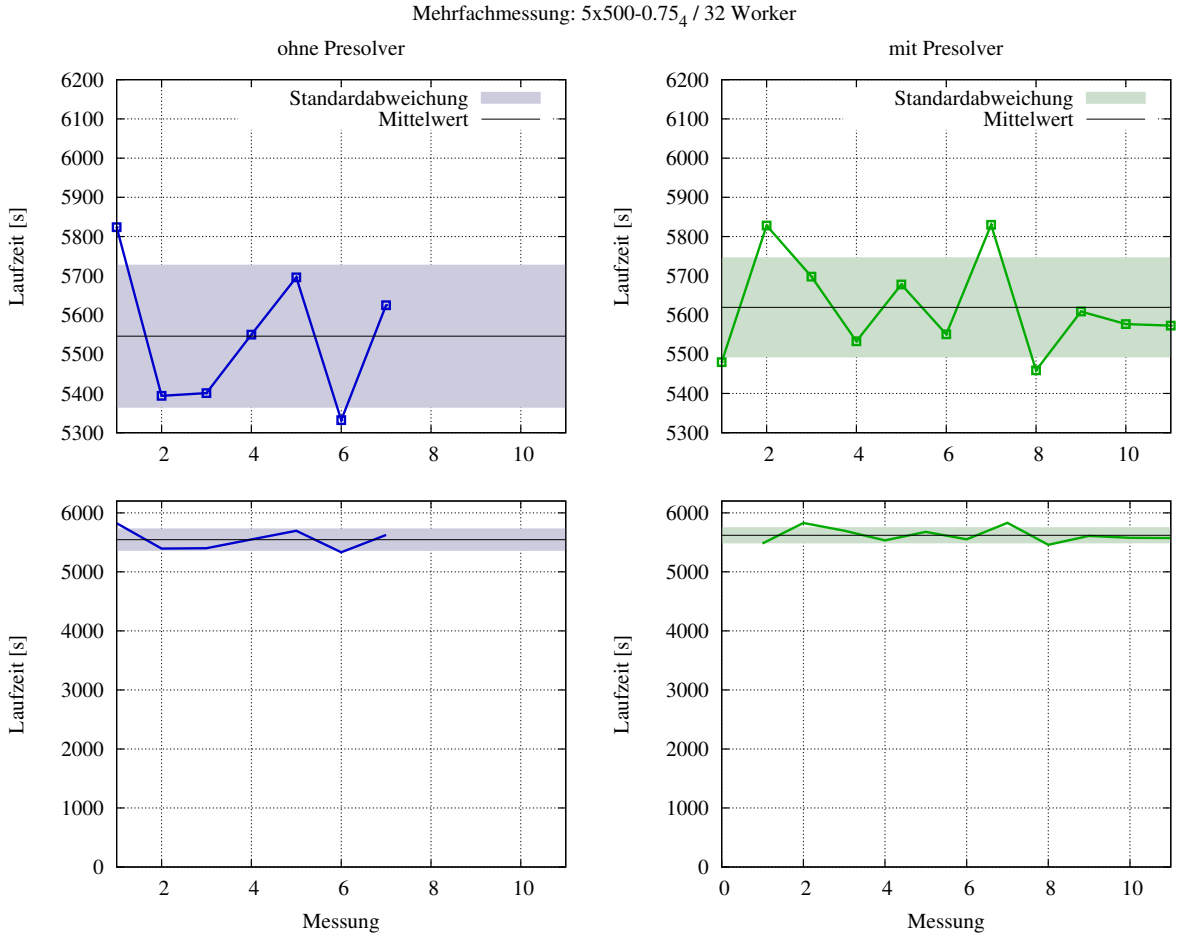

Abbildung 23: Einfluss auf die Laufzeit, wenn der Presolver abgeschaltet ist.

5.2 Entwicklungs-Historie

Farmer, Bounds und Vorbelegung – Worker-Thread

Die erste Version unserer Lösung nutzte weder *glpk* noch *thrift*. Farmer und Worker waren Threads eines Programms, wobei nur so viele Worker-Threads erzeugt wurden, wie noch CPU-Kerne frei waren. Die Vorbelegung wurde zusammen mit einer berechneten oberen Grenze in einer Liste abgelegt und der Reihe nach in den *Bounded Buffer* gelegt. Jeder Worker holte sich seine Jobs daraus. Jeder Worker machte eine obere Grenzen-Berechnung auf der Grundlage einer Projekt-Sortierung und nutzte einen *Branch-and-Bound*. Der Worker speicherte in einer thread-eigenen Variable seine bisher beste Lösung.

Worker als multithreaded-thrift-Server

Im nächsten Entwicklungsschritt wurden aus den Worker-Threads die bereits erwähnten Worker-Proxys. Jede Vorbelegung wurde über das Netzwerk mit *thrift* übertragen und dann auf einem multithreaded-Server ohne *glpk* und nur mit einem *Branch-and-Bound* verarbeitet.

Branch-and-Bound nutzt *glpk* für Upper Bound

Da die Berechnung der oberen Grenze auf der Grundlage einer Projekt-Sortierung zu

schlecht war, machten wir stattdessen mit *glpk* eine LP-Relaxierung.

Ab definierter Subproblemgröße *glpk* nutzen

Leider war der verwendete *Branch-and-Bound*-Algorithmus trotz der Verteilung auf mehrere Rechner sehr langsam. So entscheiden wir, dass wir ab einer bestimmten Menge an vorbelegten Projekten *glpk* die Lösung des „Branches“ überlassen.

Subprobleme komplett mit *glpk* lösen

Es hatte sich gezeigt, dass der *Branch-and-Cut*-Algorithmus von *glpk* so schnell war, dass unser *Branch-and-Bound* komplett von *glpk* abgelöst werden sollte. Wir taten dies und die Berechnung des Maximums wurde viel schneller fertig.

Regelmäßiger Austausch der unteren Grenzen

Durch die *quasi globale bisher beste Lösung*, die durch den Abgleich der lokalen besten Lösungen der Worker realisiert ist, konnte ein verteilter *Branch-and-Bound* realisiert werden. Bei jedem Worker wird während des *Branch-and-Cut* die untere Grenze regelmäßig hoch gesetzt, sollte eine bessere bisher beste Lösung vom Farmer an den Worker gesendet worden sein.

Gerundete Simplex-Lösung als Initialwert

Es gab die Überlegung, selber mit der LP-Relaxierung die Vorbelegung durch den Farmer intelligent zu gestalten. Jede weitere Wahl (bzw. Nicht-Wahl) eines Projekts sollte so sein, dass die Projekte mit dem Wert nahe 0,5 als nächstes vorausgewählt werden. Dies wurde auf unbestimmte Zeit verschoben, siehe *Aussicht*, Seite 41.

So ist der aktuelle Stand, dass zwar weiterhin ganz normal alle Kombinationen von Projekten durchlaufen werden, aber die erste Vorbelegung orientiert sich an der Simplex-Lösung.

Die „Schweren“ zuerst

Ein absoluter Misserfolg war eine Sortierung der Projekte anhand der Simplex-Lösung. So wurden alle Projekte, die nahe 0,5 (also nicht eindeutig 0 oder 1) waren, an den Anfang gestellt. Wir hatten die Vermutung, dass diese Projekte am stärksten das Ergebnis der Maximierung beeinflussen und so sollten diese Projekte durch unsere Iteration aller Kombinationen am häufigsten zwischen 0 und 1 schwanken. Die Laufzeit hat sich dadurch enorm verschlechtert, so dass wir überlegten, die Sortierung genau anders herum zu machen. Die ergab aber im Prinzip eine Vorbelegung, die zu Anfang komplett aus 1 besteht. Dies hatten wir bereits zwischenzeitlich getestet und bei der ersten Vorbelegung der Projekte ergab sich auch kein besseres Laufzeitverhalten.

Vergrößern der Vorbelegung: dynamische Last-Balancierung

Nach einigen Tests mit unterschiedlich großen Vorbelegungen ergab sich, dass es weder für jede Instanz eine optimale Länge der Vorbelegung gibt, noch für jede Teilinstanz. Wir entscheiden, dass nach einem Zeitlimit der Job (= die Teilinstanz) abgebrochen wird, und neue Jobs mit einer größeren Vorbelegung erzeugt werden.

Fixe Variablen und immer gleiche budget/cost Matrix

Zu Anfang hatten wir auf den Workern *glpk* immer wieder neu initialisiert und eine zuvor durch die Vorbelegung verkleinerte Kostenmatrix inkl. reduzierte Budgets übergeben. Als Alternative dazu konnte man die Matrix einmal auf allen Worker komplett füllen, und nur bei der Übermittlung des Jobs mit seiner Vorbelegung die Variablen in *glpk* entsprechend fixieren. Es zeigte sich, dass dies bei einer Vorbelegung von mehr als 15 Projekten bei den 10x100 (10 Zeiträume, 100 Projekte[23]) Instanzen von *Chu-and-Beasley* die Laufzeit geringfügig kürzer war.

Rettung der oberen Grenze

Jede Überschreitung des gesetzten Zeitlimits machte bisherige Berechnungen der oberen Grenze zunichte. Wir entschieden uns, dass in dem Fall für die zukünftigen neuen Jobs die im *Branch-and-Cut* ermittelte obere Grenze übernommen wird. Leider muss *glpk* diese Grenze erst aufwändig im bisherigen Entscheidungsbaum suchen. Dies ist so aufwändig, dass es erst aber einer größeren Vorbelegung sinnvoll ist. Es ist eine Überlegung wert, in Zukunft den Einsatz ebenfalls mit einem Parameter zu steuern.

Presolver und Heuristik

Nach gründlichem Lesen der *glpk*-Dokumentation sahen wir ein Feature, mit dem sich der *Branch-and-Cut* beschleunigen lässt. Leider kostet die Nutzung des Features zusätzliche Zeit, die durch deren Nutzung gewonnen werden muss. Es handelt sich um einen Presolver zusammen mit einem Fischetti–Monaci Proximity Search, die obere Grenzen ermitteln und die Constraints, also die Ungleichungen zu den Budget-Grenzen „optimieren“. Leider ist dieses Verfahren in der *glpk*-Dokumentation nicht genau erläutert. Es ist jedoch in dieser Veröffentlichung [5] aus dem Jahr 2013 vom Entwickler näher beschrieben. Es wird eine Stellvertreter-Zielfunktion gesucht, die nahe an der ursprünglichen Zielfunktion ist, jedoch deutlich schneller ein Optimum liefert. Es wird davon ausgegangen, dass das Optimum der Stellvertreter-Zielfunktion nahe der originalen Zielfunktion ist und der Algorithmus macht dann eine lokale Suche[24].

Sortierung der Projekte nach Summe der Kosten je Profit

Eine Vorsortierung der Projekte nach der Summe ihrer Kosten in Abhängigkeit ihres Profits, so wie wir es in eine der ersten Versionen zur Bound-Berechnung hatten, brachte bei den 5x100 und 10x100 Instanzen einen geringen Vorteil. Wir haben später eine Testreihe gemacht, wo wir den positiven Effekt dieser Sortierung erkennen konnten (Abbildung 20, Seite 34).

automatische dynamische Last-Balancierung

Als weiteres Feature, was noch weiter untersucht werden konnte, haben wir eine Modifizierung des Zeitlimits durch einen Faktor ermöglicht. Sollten bestimmte Features nicht ab einer bestimmten Größe der Vorbelegung, sondern anhand der zur Verfügung stehenden Ausführungszeit an- oder ausgestellt werden, so wäre dies nun im Programmcode möglich.

[23] gelten als leicht

[24] das entspricht in etwa einer sehr sehr großen Vorbelegung, bei der nur vereinzelt Projekte ab- oder hinzugewählt werden.

6 Fazit

Wir sehen unser Ziel erreicht, da wir mit einer Open-Source-Lösung – also ohne für das Problem spezifische Software – und bestehender Hardware eine Vielzahl von CB-Problemen in deutlich kürzerer Zeit lösen können, als dies *glpk* als sequenzielles Programm bisher konnte. So lassen sich in Büros über Nacht oder über das Wochenende hinweg Berechnungen ohne Lizenzkosten und ohne aufwändigem Hardwareeinsatz lösen. Ein weiterer Vorteil ist die Nutzung des Arbeitsspeichers aller Rechner, so dass hier kein einzelner Rechner mit enormen Mengen an Speicher ausgerüstet werden muss. Ebenso ist uns eine Unabhängigkeit von Systemarchitektur (32/64bit, Arm, Single- oder Multicore) und Betriebssystem (Windows, Mac oder Linux) gelungen.

6.1 Schwachstellen

Da weniger ein neuer Algorithmus zur Lösung von CBP im Fokus der Arbeit stand, sondern eine mit kommerzieller Software vergleichbare Anwendung, wäre zum Vergleich von Laufzeiten die `MIPLIB`[25] als Test- und Benchmark interessant. Die Laufzeiten von z.B. *CPLEX* und *Gurobi* werden regelmäßig auf der Webseite `http://plato.asu.edu/bench.html` veröffentlicht. *Chu-and-Beasley* ist leider in der `MIPLIB` nicht enthalten, was einen transparenten Vergleich erschwert. Bisher haben wir zum Vergleich diese Quelle [2], wo ein spezieller Algorithmus für die CBP von *Chu-and-Beasley* genutzt wird. Es wird da eine 2-Kerne CPU mit 3GHz und 2GB RAM genutzt, die dann bis zu 100 Tage an Instanzen mit 10 Zeiträumen und 500 Projekten arbeitet. Einige Instanzen, die wir in weniger als einem Tag auf 32 Kernen gelöst bekommen, sind dort mit speziellen Algorithmen für CBP bereits in wenigen Minuten gelöst.

Das Farmer/Worker-Modell ist nicht besonders elegant von uns implementiert worden. Mit einem `sleep()` von 500ms wird eine Stelle am Ende der Verarbeitung der Jobs überbrückt, die eigentlich durch eine Semaphore deutlich günstiger[26] gelöst werden könnte. Bei unserer Software handelt es sich noch um einen Prototypen, und das Problem im Farmer/Worker-Modell wurde von uns erst sehr spät erkannt. Um nicht einen großen Teil neu programmieren zu müssen, der evtl. neue unvorhersehbare Probleme durch kleine Bugs verursacht, entschieden wir uns für diesen pragmatischen Workaround mit dem `sleep()`. Ebenso wird auf Speicherüberschreitungen, die beim Worker und beim Farmer auftreten können, noch nicht reagiert. Das Programm liefert bei einer Speicherüberschreitung eines Worker oder beim Farmer kein – und somit auch kein falsches – Ergebnis.

Prinzipiell könnte es passieren, dass versehentlich mehrere Farmer gestartet werden, die auf die selben Worker zugreifen. Da ein Farmer zu Anfang die Instanz mit Kosten, Budgets und Profiten an die Worker verteilt, würden die Worker den Farmern falsche Ergebnisse liefern oder es wird auf einem Worker 2x `solve()` aufgerufen, und das nicht thread-fähige *glpk* läuft in 2 Threads. Hierzu könnte man noch einen Schutz-Mechanismus einbauen.

Es hat sich gezeigt, dass durch eine Vorbelegung der Projekte der *Branch-and-Cut* nicht zwingend schneller fertig wird. Es gab Fälle, wo ein Teilbaum viel mehr Zeit zur Lösung

[25] `http://miplib.zib.de/`

[26] günstiger im Sinne von: schneller als 500ms

brauchte, als wenn der gesamte Baum gelöst wird. Unser ganzer Ansatz der Verteilung geht davon aus, dass Teilbäume im Mittel mit *Branch-and-Cut* schneller gelöst werden können als das Gesamtproblem. So wie es aussieht, stimmt diese Annahme wohl, ohne das wir dieses Phänomen näher untersuchen konnten. Es bleibt die Frage offen, ob es tatsächlich Glück ist, dass bei die Aufteilung einer *Chu-and-Beasley*-Instanz in Teilbäume im Schnitt die Teilbäume schneller mit einem *Branch-and-Cut* zu lösen sind, wenn bei der parallelen Verarbeitung der Teilbäume eine untere Grenze ausgetauscht wird (also: so wie wir es machen).

6.2 Aussicht

Insbesondere die breite Palette an Konfigurationen unseres Programms sollte weiter ausgereizt werden. Welche Parameter lassen sich zur Laufzeit automatisch optimal einstellen? Wie lässt sich in *glpk* schnell die Spanne zwischen der bisher besten Lösung und der kleinsten oberen Grenze „gap“ ermitteln, so dass man auch bei einer Verteilung eine globale obere Grenze hat? Mit den Werten kann man den Fortschritt der Lösungssuche besser einschätzen. Weitere nützliche Features wären:

- Erkennen des Ausfalls von Workern und Neuverteilung ihrer Jobs, da es nicht passieren darf, dass eine Berechnung hängen bleibt, weil z.B. ein Netzwerk-Problem mit einem Rechner auftrat.
- Ein aktives Unterbrechen der Worker durch den Farmer beim Überschreiten des Zeitlimits (nicht den Worker entscheiden lassen) wäre besser. So kann der Farmer, der die Anzahl der aktiven Worker kennt, aktiv einen Worker abbrechen und dessen Aufgabe neu verteilen.
- Snapshots erzeugen, so dass die verteilte Berechnung unterbrochen und wieder aufgenommen werden kann (z.B. bei geplantem oder sogar ungeplantem Stromausfall).
- nachträgliches Hinzufügen von Workern
- GUI für Windows

Wie wir im Kapitel *Konzept* bereits erwähnt haben, suchen wir noch eine Idee für eine praktikable automatische Umschaltung zwischen Breiten- und Tiefensuche. Ein Strategie-Entwurfsmuster für unterschiedliche Konzepte sollte an dieser Stelle außerdem leicht zu implementieren sein. Eine mögliche Idee ist es, nach der Umschaltung von Breiten- auf Tiefensuche gelegentlich (zufällig) auch mal ein Job zu einem Teilbaum der Breitensuche zu nehmen. Eine andere Möglichkeit wäre, dass beim Hinzufügen der Jobs die mit einer bestimmten Vorbelegung vom Farmer zuerst genommen werden. Dies entspräche einer Tiefensuche ab einer bestimmten Baumtiefe.

Unser bisheriges Konzept besteht aus einer Vorbelegung von Projekten, bei der die ersten K Projekte mit allen 2^K Kombinationen aus 0 und 1 vorbelegt werden. Durch unsere Sortierung der Projekte nach der Summe ihrer Kosten je Profit, findet eine Priorisierung der Vorbelegung statt.

Welches Projekt zuerst mit 0 oder 1 belegt wird, und ob zuerst mit 0 oder zuerst mit 1 belegt wird, ist mit einer LP-Relaxierung voraussichtlich zu verbessern. So könnte man z.B. bei jeder Vorbelegung eines weiteren Projekts zuerst eine LP-Relaxierung machen,

und das Projekt dessen Auswahl am „Unbestimmtesten“ ist (genau zwischen 0 und 1: 0,5) als erstes fest mit 0 oder 1 vorbelegen. Wenn wir z.B. $K = 5$ haben, geschieht dieser Vorgang noch weitere 4 mal. Bei jeder LP-Relaxierung erhalten wir außerdem eine obere Grenze, mit der wir bereits beim Farmer Teilbäume bevorzugen können (Abbildung 24). Bei BILP ist dieses Verfahren sehr simple, da nur 0 oder 1 zur Auswahl steht und die Variable entsprechend festgehalten werden muss. Alle Kombinationen sind so in der Vorbelegung der Projekte noch möglich, sofern man diese nicht verwirft[27]. In diesem Verfahren verbirgt sich jedoch die Chance, bereits beim Farmer einen *Branch-and-Cut*-Algorithmus an zu wenden, der auch eine Art Vorbelegung von Integer-Werten ermöglicht. Anstelle einer fixierten Variable, wird eine weitere Bedingung als sogenannte *Cut-Ebene* beigefügt. Kommt bei einer von K vielen LP-Relaxierungen nun 3,5 mal das „i-te Projekt“ heraus, so wird dieses Projekt i nicht im Job **A** mit 3 und in Job **B** mit 4 vorbelegt, sondern es wird eine Bedingung mit $3 \geq x_i$ bzw. in Job **B** eine Bedingung mit $x_i \geq 4$ hinzugefügt. So ist nicht nur unser Multi-Period Capital Budgeting Problem lösbar, sondern es wäre auch allgemein ILP lösbar. Um ein verteiltes ILP zu ermöglichen, muss unsere Software jedoch stark angepasst werden. Der Code für den Worker muss in der Lage sein, mehr als nur einen binären Vektor als Vorbelegung zu verarbeiten. Die Interface-Definition von *thrift* muss um weitere Datentypen ergänzt werden, die nun bei einem neuen Job übertragen werden müssen. Die `Filling`-Klasse muss um die neue Logik der Vorbelegung erweitert werden. Prinzipiell ist eine Erweiterung also möglich.

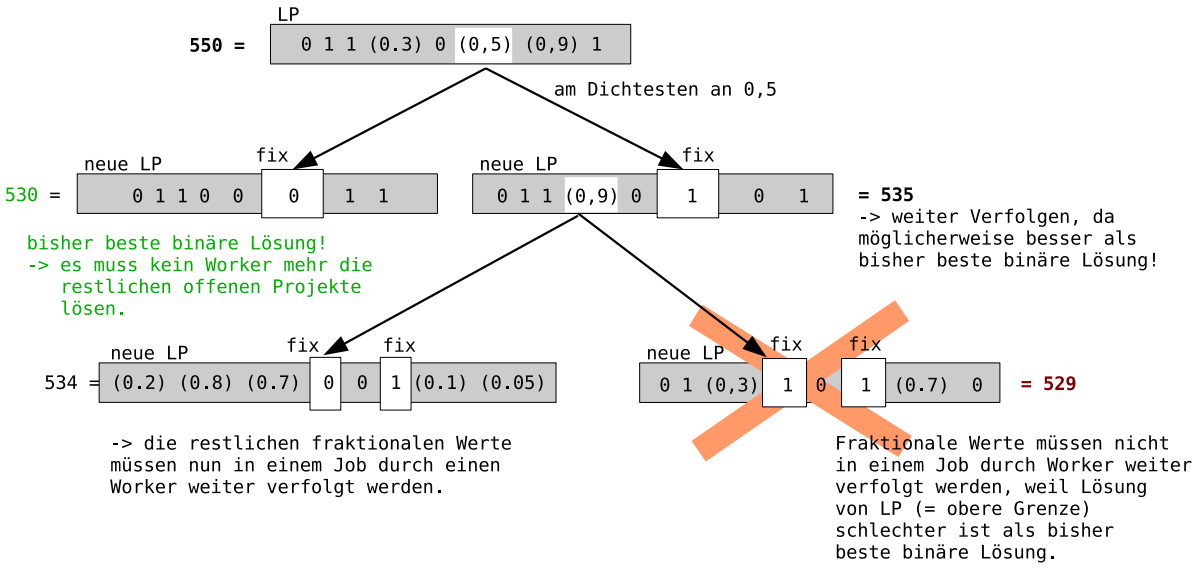

Abbildung 24: Nutzung der LP-Relaxierung bei der Vorbelegung von 2 Projekten von insgesamt 8 Projekten. Anstatt dass 4 Jobs erzeugt werden, wird in dem Beispiel zuerst nur ein Job erzeugt. Zum Einen, weil bei der ersten Vorbelegung ein „Branch“ bereits eine binäre Lösung liefert, und zum Anderen, weil bei der 2. Vorbelegung ein „Branch“ als obere Grenze eine schlechtere Lösung erwartet.

Eine weitere Verbesserung wäre die Nutzung eines Brokers, der zwischen Farmer und Worker agiert. Es trat bei uns der Fall auf, dass die Anzahl der zu bearbeitenden Jobs auf über 50 Millionen an stieg. Dies führte beim Farmer mit 8 GB RAM zu einem Speicherüberlauf

[27] Das Ergebnis der LP-Relaxierung könnte ja schlechter sein, als die bisher beste Lösung.

und er brach ab. Mit einem Broker könnte man die Speichernutzung eleganter verteilen, so dass ein Farmer z.B. mit 3 Brokern statt 3 Workern kommuniziert (Abbildung 25). Jeder Broker agiert wie einer der bisherigen Farmer und nutzt eine Konfiguration, die für die ihm zugeteilten Worker optimal ist, was in heterogenen Büroumgebungen sinnvoll ist.

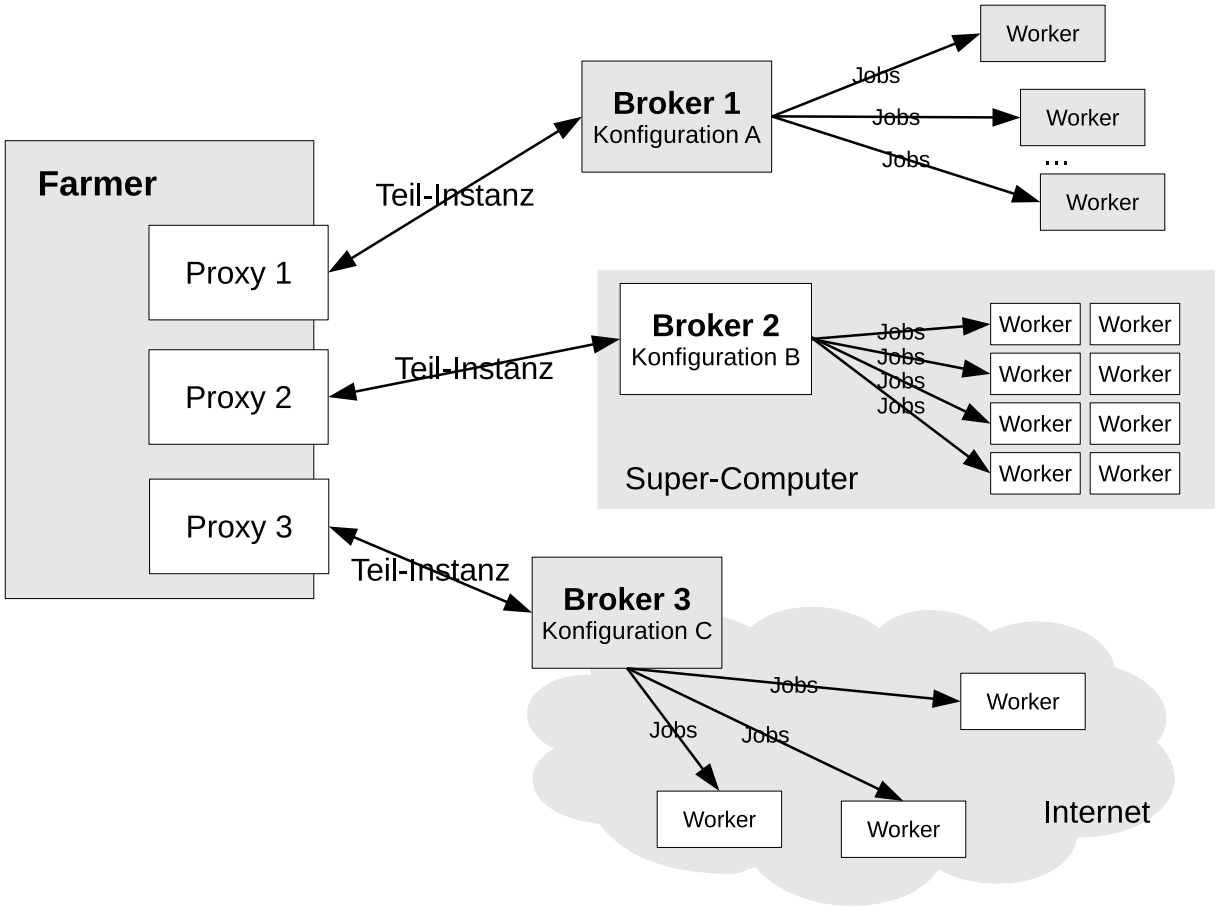

Abbildung 25: Mit einem Broker, der für einen Farmer so ähnlich wie ein Worker agiert, ließe sich die Last eines Farmers (Millionen von Jobs) besser verteilen.

7 Anhang

7.1 Die *Chu-and-Beasley* Test-Instanzen

Um ein „Kräftemessen“ der Algorithmen beim CBP zu ermöglichen, gibt es international anerkannte Test-Instanzen ([4], [11], [15] und [2]).

Wir beschränkten uns auf die „Operation Research“ Instanzen von *Chu-and-Beasley* [3], welche aus Zufallszahlen für Profit, Kosten und Budgets bestehen. Es existieren jeweils 10 Test-Instanzen für 5, 10 und 30 Zeiträume, in denen 100, 250 sowie 500 Projekte platziert sind. Eine Test-Instanz sieht dann als Dateiname z.B. so aus:

`OR5x250-0.75_6.dat`

Um eine Art Schwierigkeitsgrad angeben zu können, gibt es dann noch drei „Tightness“-Versionen (0.25, 0.50 und 0.75) – in diesem Beispiel ist es 0.75. Die *Chu-and-Beasley* Test-Instanzen umfassen somit für drei unterschiedlich lange Zeiträume, drei unterschiedliche Projektmengen und drei „Tightness“-Versionen je 10 Test-Instanzen; das sind 270 Dateien insgesamt. Da wir diese Instanzen auch verwenden, gehen wir noch näher auf die „Tightness“ ein. Ein Wert von 0.75 oder besser $\frac{3}{4}$ bedeutet, dass in einer Zeitperiode das Verhältnis von Budget zur Summe der Projektkosten exakt 3:4 ist. Es gilt also für jeden Zeitraum j dieser Instanz für alle n Projekte:

$$\frac{3}{4} \quad = \quad b_j / \sum_{i=1}^{n} c_{ji}$$

Bei den 0.25-Instanzen ist die Summe der Kosten 4 mal höher als das zur Verfügung stehende Budget. Bei den 0.50-Instanzen ist diese Summe nur 2 mal so groß.

Einige dieser Instanzen sind noch ungelöst. Die OR-Gruppen konzentrieren sich daher bei der Angabe, wie effizient ihre Algorithmen sind, auf das „gap“ nach einer festgelegten Laufzeit. Um zu verstehen, was mit dem *gap* gemeint ist, müssen wir zwei Begriffe erklären: die *upper bound* und die *lower bound*. Da wir es beim CBP mit einem Maximierungsproblem zu tun haben, ist die bisher beste Lösung immer die untere Grenze (= lower bound) des Lösungsraums. Um nicht jedes Projekt-Portfolio berechnen zu müssen, schätzen die Algorithmen oberen Grenzen (= upper bound) für unvollständig berechnete Portfolios ab. Bei einem unvollständig berechneten Portfolio sind bestimmte Projekte bereits fest aus- bzw. abgewählt, während andere Projekte noch unbestimmt sind. Anhand dieser oberen Grenzen lassen sich die noch unbestimmten Projekt-Selektionen verwerfen, sollte deren Abschätzung schlechter als die *lower bound* sein. Je näher diese Grenzen zusammen liegen, desto näher ist man an einem Optimum. Man ist mit der Berechnung fertig, sobald keine obere Grenze mehr existiert, die besser als die bisher beste Lösung – der unteren Grenze – ist.

Mit *gap* wird nun die prozentuale Distanz bezeichnet, wie weit die Grenzen von der bisher besten Lösung (>0) entfernt sind:

$$\text{gap} = \frac{|\text{best\_mip} - \text{best\_bnd}|}{|\text{best\_mip}| + \text{DBL\_EPSILON}} \quad (1)$$

Diese Gleichung stammt aus der Dokumentation[9] von *glpk*. `best_mip` bezeichnet die bisher beste Lösung, `best_bnd` ist die beste obere Grenze und `DBL_EPSILON` ist in `C` und `C++` die kleinste im Datentyp `double` darstellbare Zahl (ein Trick, um nicht durch 0 zu teilen).

Alle Lösungsansätze, die wir fanden, konzentrieren sich auf folgende Fragen:

- Welche Projekte wählt man in einer initialen Phase aus, die in späteren Heuristiken vielversprechende Ergebnisse liefern?
- Wie lässt sich der Lösungsraum am schnellsten verkleinern?
- Lassen sich die Bedingungen (constraints) reduzieren?
- Lassen sich aus Teilergebnissen neue Bedingungen bilden, die schneller den Lösungsraum einschränken?
- Lassen sich aus den bisherigen Bedingungen neue Bedingungen und Grenzen bilden?
- Welche Bedingung schränkt den Lösungsraum am stärksten ein?
- Lassen sich während der Berechnung Daten erheben, die eine dynamische Wahl einer anderen Heuristik ermöglicht?
- Bei einer Verteilung auf mehrere CPUs[4]: Welche Daten sollten ausgetauscht werden, um schneller eine Lösung oder Grenze zu finden?

7.2 Glossar

Ameisen Die Nutzung von Ameisen als Bild auf der Titelseite und als leichter Schatten auf fast jeder Seite dient nur zur leichteren Identifizierung dieser Arbeit. Da LaTeX eine Standardisierung im Layout forciert, ist das Wiedererkennen und Erinnern an die Inhalte aus meiner Sicht beeinträchtigt.

BILP Binary Integer Linear Programming

constraint engl. für Bedingung wie z.B. eine Ungleichung, die die Budget-Grenze eines Monats festlegt.

Farmer Der Farmer ist der Teil unseres verteilten Systems, der die Aufgaben (Jobs) erzeugt. Er ist bei uns als einzelner *thrift*-Client realisiert.

Heuristik Dieser Begriff wird gerne für Lösungsverfahren (also: Algorithmen) genommen, die aus bestehenden Problemen und Teil-Ergebnissen aufgrund von Annahmen neue Probleme und neue Teil-Ergebnisse erzeugen, die eine weitere Lösungssuche voraussichtlich erleichtern.

ILP Integer Linear Programming

Job/Teilproblem Ein Job besteht bei uns aus einem bool-Vektor, der die Vorbelegung repräsentiert, zwei Flags für die Markierungen *Zeitlimit erreicht* und *Worker-Proxy beenden* sowie zwei Integer-Werten, die die beiden Grenzen (bisher beste Lösung, beste obere Grenze des Teilbaums) enthält.

Kostenmatrix Das zweidimensionale Feld aus den Kosten eines Projekts je Monat bildet eine Matrix.

lower bound (beste untere Grenze) Die bisher beste Lösung aus allen Teilproblemen stellt automatisch eine untere Grenze des Maximierungsproblems dar. Sie wird in englischen Publikationen „best feasible solution“ genannt.

MILP Mixed Integer Linear Programming

Optimum In einem Maximierungsproblem ist das durch die Zielfunktion gesuchte Optimum ein Maximum. Im CBP ist dies der Gesamt-Gewinn.

OR operation research (Verfahrensforschung)

Projekt-Portfolio Die Lösung des Capital Budgeting Problems besteht nicht aus einem maximal zu erreichenden Gewinn, sondern viel mehr aus der Auswahl an Projekten, aus der man den maximalen Gewinn erhält. Dieser binäre Vektor, in dem eine 1 die Wahl eines Projektes bedeutet, wird auch State oder Portfolio genannt.

Strategie-Muster Das Strategie-Entwurfsmuster ist dafür da, um in einer Software unterschiedliche Verfahren und ggf. Datenstrukturen leicht austauschbar zu machen.

upper bound (beste obere Grenze) Viele Algorithmen und Heuristiken suchen nicht primär nach dem Lösungsvektor, sondern Versuchen bei einem Maximierungsproblem eine obere Grenze (z.B. des Gesamt-Gewinns) zu ermitteln. Mit dieser Grenze können zukünftige Berechnungen vermieden werden, wenn andere Rechenwege bereits eine bessere, gültige Lösung ergaben. Ebenso kann mit einer solchen Grenze

eine Abschätzung des bisherigen Rechenwegs gemacht werden, ob dieser Rechenweg z.B. die Grenze am Stärksten herabsetzte.

Worker Der Worker ist der Teil unseres verteilten Systems, der die Jobs des Farmers verarbeitet. Er entspricht bei uns einem von vielen *thrift*-Servern.

Worker-Proxy Für jeden Worker gibt es einen *Stellvertreter-Thread*, der die Verbindung zum Worker repräsentiert. Dies ist das Proxy-Entwurfsmuster.

7.3 Lizenzhinweise und Quellcode

Alle im Abbildungsverzeichnis aufgezählten Abbildungen und Tabellen sind von mir erstellt worden und werden von mir unter der **Creative Commons Lizenz CC-BY-SA** (Namensnennung, Weitergabe unter gleichen Bedingungen) freigegeben.

Der von mir erstellte und gezeigte Quellcode (siehe `https://sourceforge.net/p/bat2015`) wird von mir unter der **2-Klausel-BSD Lizenz** freigegeben:

Abbildungsverzeichnis

Literaturverzeichnis

[1] **D. Bertsimas, J. N. Tsitsiklis:** *Introduction to Linear Optimization.* Athena Scientific, 1. Auflage, Belmont 1997

[2] **S. Boussier, M. Vasquez, Y. Vimont, S. Hanafi, P. Michelon:** *A multi-level search strategy for the 0-1 Multidimensional Knapsack Problem.* In: *Discrete Applied Mathematics.* 158(2): 97-109, 2010

[3] **P. C. Chu, J. E. Beasley:** *A genetic algorithm for the generalized assignment problem.* In: *Computers Operations Research.* 24(1): 17-23, 1997

[4] **J. H. Drake, E. Özcan, E. K. Burke:** *Controlling Crossover in a Selection Hyper-heuristic Framework.* In: *Computer Science Technical Report No. NOTTCS-TR-SUB-1104181638-4244.* University of Nottingham, Nottingham 2011

[5] **M. Fischetti, M. Monaci:** *Proximity Search for 0-1 Mixed-Integer Convex Programming.* In: *Journal of Heuristics 12/2014.* 20(6):709-731, Abrufdatum 01.09.2015, `http://www.dei.unipd.it/~fisch/papers/proximity_search.pdf`, University of Padova 2013

[6] **M. R. Garey, D. S. Johnson:** *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1. Auflage, New York 1979

[7] **J. Kallrath:** *Gemischt-ganzzahlige Optimierung: Modellierung in der Praxis.* Vieweg, 1. Auflage, Braunschweig 2002

[8] **N. Karmarkar:** *A New Polynomial Time Algorithm for Linear Programming.* In: *Combinatorica.* 4:373-395, Murray Hill 1984

[9] **A. Makhorin:** *glpk - GNU Linear Programming Kit.* Version 4.55, `https://www.gnu.org/software/glpk/`, August 2014

[10] **P. Merz:** *Moderne Heuristische Optimierungsverfahren: Meta-Heuristiken.* In: *Vorlesungsfolien Sommersemester 2003.* Universität Tübingen, Abrufdatum 04.08.2015, `http://www.ra.cs.uni-tuebingen.de/lehre/ss03/vs_mh/mh-v1.pdf`, 2003

[11] **J. Puchinger, G. R. Raidl, U. Pferschy:** *The Multidimensional Knapsack Problem: Structure and Algorithms.* In: *Technical Report 186–1–07–01.* TU Wien, Abrufdatum 14.08.2015, `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.219.5127`, 2007

[12] **C. Riche:** *Migrating an existing model to Gurobi Optimizer.* Gurobi Optimization, Abrufdatum 06.08.2015, `http://www.gurobi.com/pdfs/modelmigration.pdf`, 2012

[13] **J. L. Rios:** *DWSOLVER (Dantzig-Wolfe Solver).* Version 1.0, United States Government National Aeronautics and Space Administration (NASA), 2010, `https://github.com/alotau/dwsolver`

[14] **M. Slee, A. Agarwal, M. Kwiatkowski:** *Thrift: Scalable Cross-Language Services Implementation.* Version 0.9.2, `https://thrift.apache.org/`, November 2014

[15] **M. Vasquez, Y. Vimont:** *Improved results on the 0-1 multidimensional knapsack problem.* In: *European Journal of Operational Research.* 165(1): 70-81, 2005