ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Theoretische Informatik

Analyse von Parallelen Automatischen Algorithmen-Konfiguratoren

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Juri Schulte
juri.schulte@uni-ulm.de

Gutachter:
Prof. Dr. Uwe Schöning

Betreuer:
Adrian Balint

2011

Fassung 21. Dezember 2011

Satz: PDF-LaTeX 2ε

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

In dem modernen Bemühen Handlungsabläufe schlank und kostengünstig zu halten spielen performante Algorithmen eine essentielle Rolle. So werden Lieferrouten optimiert, um Zeit zu sparen, benötigte Produktionsbestände vorausberechnet, um die Lagerkosten gering zu halten und Arbeitspläne ideal auf die Arbeiter zugeschnitten. Das Optimierungspotenzial solcher und anderer Algorithmen liegt dabei besonders für heuristische Algorithmen, welche versuchen Lösungen für NP-harte Probleme zu finden, stark in einer geeigneten Wahl der Parameter. So kann es bei einer schlechten Wahl Tage, wenn nicht Wochen dauern, eine zufriedenstellende Lösung zu finden, jedoch bei einer guten Wahl nur Sekunden. Aus diesem Grund wird viel Zeit, teils mehr als in den Algorithmus selber, in die Konfiguration der Parameter investiert. Heutzutage wird dieser Optimierungsvorgang sehr häufig manuell durch einen erfahrenen Entwickler per Ausprobieren durchgeführt. Bei einer Vielzahl von Parametern wird dabei, aufgrund fehlender Entwicklungszeit, meist jeder Parameter isoliert, für sich, getestet und nur auf einer geringen Anzahl an Probleminstanzen evaluiert. Abhängigkeiten zwischen Parametern werden meist vernachlässigt. Entsprechend gering ist oft die Güte der gefundenen Lösung und entsprechend hoch der Bedarf nach automatisierten schnellen Verfahren.

> "As an analogy, consider Sudoku puzzles. While humans can in principle solve these, computer algorithms can do so much more efficiently. While Sudoku is a fun exercise for the brain, countless days of experimenting with different heuristics and parameter settings is *not*. " ([6], Kapitel 1, Seite 5)

Im Folgenden wird der Ablauf einer automatischen Algorithmenkonfiguration beschrieben und mit bereits existierenden Konfiguratoren näher erläutert. In Kapitel 3 wird dann eine eigene Implementierung vorgestellt und mit der, in Kapitel 4 beschriebenen Experimente, hinsichtlich der Parallelisierungsgüte, getestet. Die Ergebnisse finden sich in Kapitel 5.

1.1 Automatische Algorithmen-Konfiguration

Für die automatische Konfiguration eines Algorithmus wird, neben dem Algorithmus selbst, eine Menge an Probleminstanzen benötigt auf welchen der Algorithmus ausgeführt wird. Die gegebene Menge dieser Instanzen sollte eine heterogene Teilmenge des Anwendungsbereiches abdecken. Werden beispielsweise nur einfache Probleminstanzen gewählt, liefert der Algorithmus, mit der ermittelten Parameterkonfiguration auf schweren Probleminstanzen mit hoher Wahrscheinlichkeit unbrauchbare Ergebnisse. Dieser Aspekt wird auch als *Overconfidence* bezeichnet. Weiterhin sollte die Instanzmenge so klein wie möglich gehalten werden. Viele ähnliche Instanzen kosten wertvolle Rechenzeit und liefern keine zusätzlichen Informationen über die Güte einer Parameterkonfiguration - auch *Overtuning* genannt. Als konkretes Beispiel könnte man aus der Videokompression den MPEG-4 Codec anführen. Parameter, wie beispielsweise die Auflösung, die Framerate und die Bitrate, beeinflussen maßgeblich die Güte des komprimierten Videos. Der Codec stellt also den zu konfigurierenden Algorithmus dar, welcher hinsichtlich des Verhältnisses zwischen Qualität und Speicherbedarf optimiert werden soll. Die Probleminstanzen sind Videodateien. Diese Videodateien sollten so gewählt werden, dass Overtuning und Overconfidence vermieden werden. Es sollten also nicht zu lange Dateien verwendet werden und nicht nur Schwarz/Weiß-Videos, wenn später auch Farbfilme komprimiert werden.

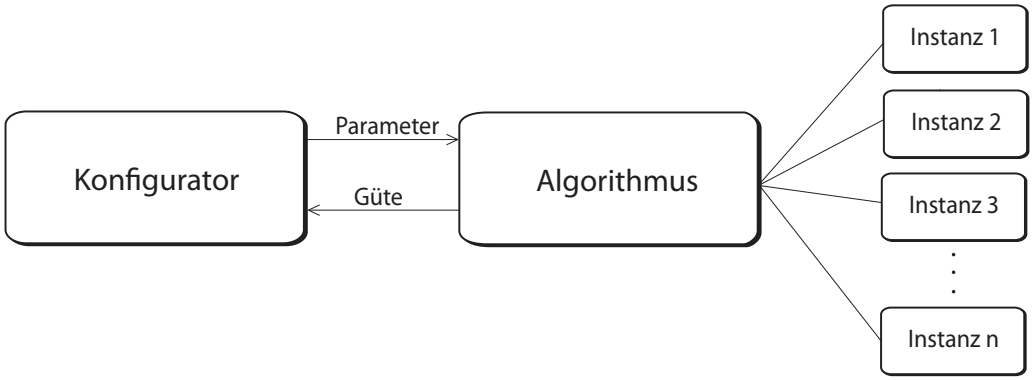

Abbildung 1.1: Allgemeiner Algorithmen-Konfigurationsablauf

In Abbildung 1.1 wird der Ablauf einer Algorithmen-Konfiguration dargestellt. Das eigentliche Programm für die Konfiguration wird als Konfigurator bezeichnet und ruft den zu optimierenden Algorithmus, mit einer entsprechenden Parameterkonfiguration auf einer Pro-

bleminstanz, auf. Als Ergebnis liefert der Algorithmus die Güte der Lösung, also zum Beispiel die Zeit, in der eine Lösung gefunden wurde. Um dabei Parameterkonfigurationen gegeneinander abwägen zu können und gleichzeitig eine stetige Verbesserung zu erzielen, teilt sich ein Konfigurator in zwei Bestandteile auf: Die *Racing*-Prozedur entscheidet auf einer gegebenen Menge an Parameterkonfigurationen, welches die jeweils besten sind. Der zweite Teil ist eine *Such*-Prozedur, welches anhand der Güte von Parameterkonfigurationen neue Konfigurationen ermittelt, die wiederum durch die Racing-Prozedur geprüft werden.

1.1.1 Racing Konfiguratoren

Racing Konfiguratoren verzichten auf die Such-Prozedur und konzentrieren sich auf eine möglichst effiziente Analyse einer endlichen Menge an Parameterkonfigurationen $\mathcal{P}$ auf festgelegten Probleminstanzen $\mathcal{I}$. Dabei wird zunächst jeder Konfiguration aus $\mathcal{P}$ die gleiche Anzahl an Läufen, auf den gleichen Instanzen aus $\mathcal{I}$, gewährt. Schneiden nach diesem Durchgang bereits Parameterkonfigurationen besonders schlecht ab, werden sie aus $\mathcal{P}$ entfernt. Iterativ werden nun allen übrig gebliebenen Konfigurationen aus $\mathcal{P}$ weitere Läufe auf Instanzen aus $\mathcal{I}$ zugeteilt und anschließend schlechte Konfigurationen aus $\mathcal{P}$ entfernt. Dieser Vorgang wiederholt sich, bis eine beste Konfiguration gefunden oder ein gegebenes Zeitlimit überschritten wird.

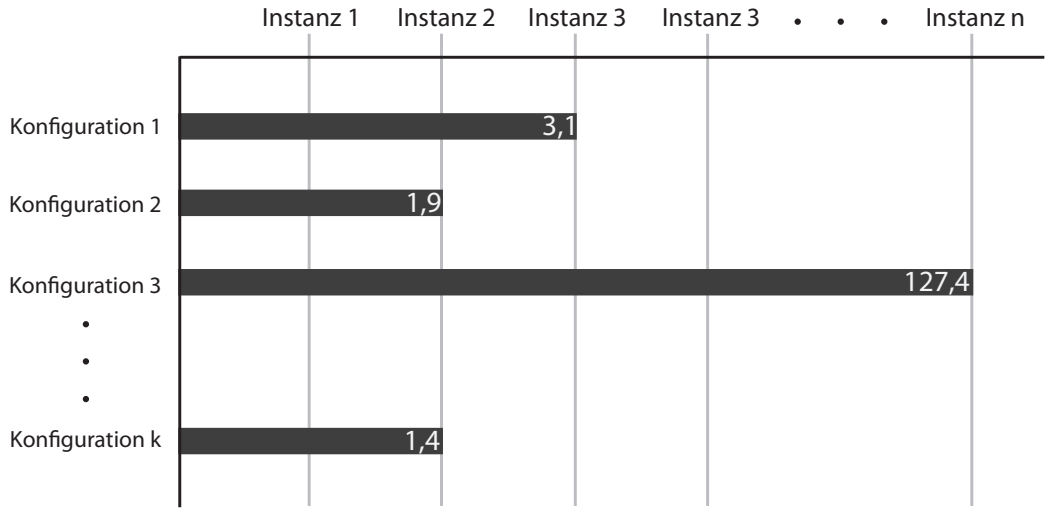

Abbildung 1.2: Möglicher Lauf eines Racing Konfigurators

Abbildung 1.2 zeigt das fiktive Ergebnis einer allgemeinen Racing-Konfiguration. Während Konfiguration 2 und k auf nur zwei Probleminstanzen evaluiert und danach nicht weiter betrachtet wurden, stellt Konfiguration 3 die ermittelte beste Konfiguration dar, die in jedem Iterationsschritt zusätzliche Läufe erhalten hat.

Eine spezielle Variante dieser Racing-Konfiguratoren ist F-Race ([5]). F-Race verwendet nach jeder Iteration den Friedman-Test, einen statistischen, nicht-parametrischen Test, um signifikante Unterschiede in der Güte der Parameterkonfiguration untereinander aufzudecken.

Insgesamt bietet ein Racing-Konfigurator zwar eine effiziente Möglichkeit Parameterkonfigurationen zu vergleichen, ist aber aufgrund fehlender Suchstrukturen, eher für Algorithmen mit kleinem Parameterraum geeignet. Bei kleinen diskreten Suchräumen besteht $\mathcal{P}$ aus einer vollständigen Aufzählung aller möglichen Parameterkonfigurationen. Bei größeren Suchräumen wird $\mathcal{P}$ durch ein Sampling-Verfahren, wie bspw. das Latin hypercube sampling erstellt. Trotzdem $\mathcal{P}$ den Parameterraum grob abdecken kann, findet keine sukzessive Verfeinerung der Ergebnisse statt.

1.1.2 Konfiguratoren basierend auf genetischen Algorithmen

Konfiguratoren, die auf genetischen Algorithmen basieren, orientieren sich an biologischen Populationsbeobachtungen. Zentraler Bestandteil bildet die Auslese besonders starker Individuen, auch als *Selektion* oder “Survival of the Fittest” [1] bezeichnet. Neben der Selektion bilden die *Mutation* und die *Kreuzung* zwei wesentliche Aspekte genetischer Konfiguratoren.

Wie aus Abbildung 1.3 ersichtlich, wird bei der Kreuzung aus zwei existierenden “Eltern”-Parameterkonfigurationen, eine neue “Kind”-Konfiguration erstellt. Dabei werden dem “Kind” zufällig Werte der “Eltern” zugeteilt. Dieses Vererben von Anlagen ist essentiell, um gute Anlagen im Genpool zu halten oder genauer: um gute Parameterwerte beizubehalten. Neben der Kreuzung werden Parameterwerte nur durch Mutation verändert. Tritt eine Mutation auf, wird jeder Parameterwert mit einer prozentualen Wahrscheinlichkeit, durch einen zufälligen neuen Wert ersetzt.

Ein Zusammenspiel dieser drei Grundprinzipien wird in [2] beschrieben. In diesem geschlechtsbasierten Ansatz wird die Population (feste Menge von Parameterkonfigurationen)

[1] Aus dem Werk “The Origin of Species” (1859) von Charles Darwin

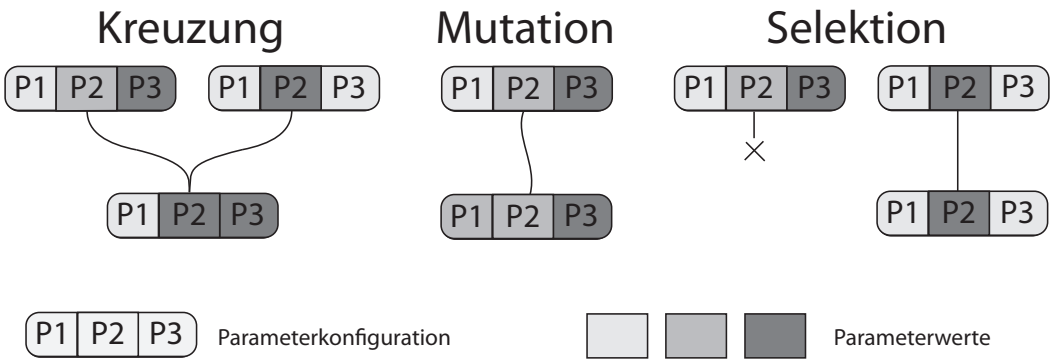

Abbildung 1.3: Grundfunktionen eines genetischen Konfigurators

in zwei Gruppen gegliedert. Die Individuen der Gruppe $\mathcal{A}$ werden untereinander verglichen und die besten x% mit einem Individuum aus Gruppe $\mathcal{B}$ gekreuzt. Bei der Kreuzung treten zu y% Mutationen auf. Das entstandene Kind kommt mit einer 50% Wahrscheinlichkeit in Gruppe $\mathcal{A}$, sonst in $\mathcal{B}$. Bei jeder Kreuzung wird das entsprechende Individuum älter und wird ab einem gewissen Alter aus der zugehörigen Gruppe entfernt.

Vorteile dieser Konfiguratoren sind zum einen die Einfachheit, zum anderen die Robustheit. Neben dem Alter und der Gruppenzugehörigkeit der jeweiligen Parameterkonfiguration muss nichts gespeichert oder konsistent gehalten werden. Die stattfindenen Veränderungen an den Konfigurationen sind simpel und zufallsbedingt. Entsprechend wenig Fehler können selbst auf komplexen Parameterumgebungen auftreten. Nachteilig ist allerdings die feste Ausgangspopulation. Sie muss manuell erstellt werden und bestimmt maßgeblich die Geschwindigkeit, in der eine gute Parameterkonfiguration gefunden wird.

1.1.3 Konfiguratoren basierend auf lokaler Suche

Im Gegensatz zu Racing Konfiguratoren und Konfiguratoren mit genetischen Algorithmen, arbeiten Konfiguratoren, die auf lokaler Suche basieren, mit speziellen Suchalgorithmen. Die lokale Suche ist ein Oberbegriff von metaheuristischen Suchalgorithmen, die im Allgemeinen auf sogenannten Nachbarschaften agieren. Parameterkonfigurationen sind Nachbarn, wenn sie sich nur hinsichtlich eines Parameterwertes unterscheiden. Eine Nachbarschaft bezeichnet die Menge aller Nachbarn. Die lokale Suche startet mit einer beliebigen Parameterkonfiguration und sucht in der Nachbarschaft nach einem Nachbarn mit einer höheren Güte. Wird so eine Konfiguration gefunden, wird auf der neuen Nachbar-

schaft weitergesucht. Abbildung 1.4 zeigt, wie ein solcher Vorgang aussehen könnte. Dabei können Nachbarschaften unterschiedlich groß sein und aufgrund fehlender Rechenzeit, oftmals nicht alle Nachbarn betrachtet werden. Die Parameterkonfiguration am Koordinatenursprung ist die Startkonfiguration. Der Nachbar, mit der jeweils höchsten Güte, wird als Schnittpunkt zweier anliegender Nachbarschaften dargestellt.

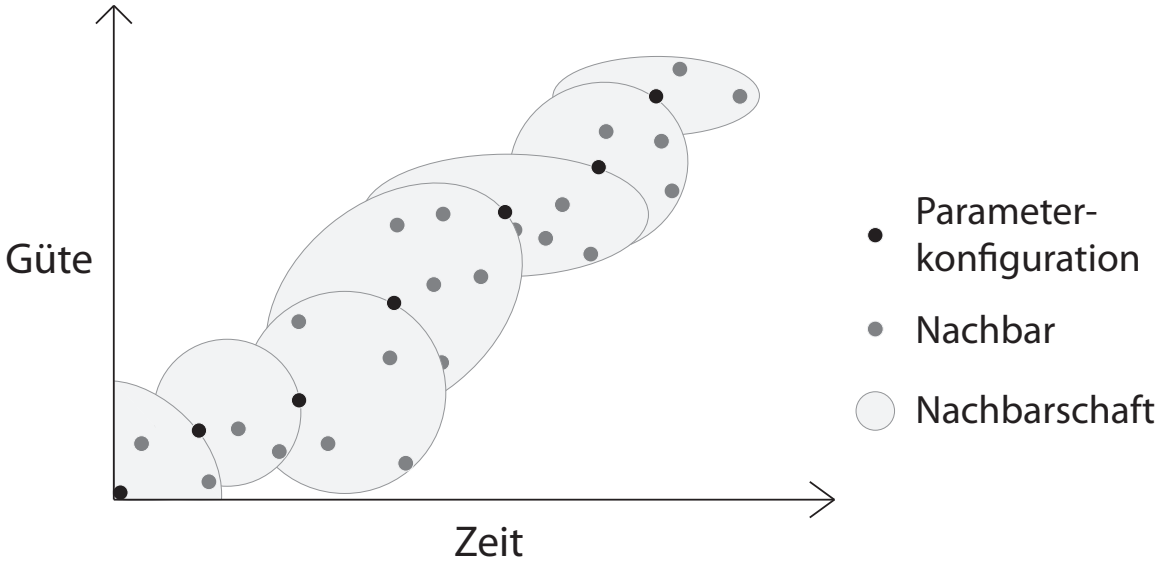

Abbildung 1.4: Lokale Suche

Auf Basis der iterativen lokalen Suche wurden in [6] die zwei Konfiguratorenvarianten BasicILS und FocusedILS vorgestellt, auf die in Kapitel 2 näher eingegangen wird. Prinzipiell unterscheiden sich die beiden Ansätze nur hinsichtlich der Racing-Prozedur voneinander. Während BasicILS zwei Parameterkonfigurationen vergleicht, indem es beide auf einer festgelegten Anzahl an Probleminstanzen evaluiert, teilt FocusedILS Konfigurationen dynamisch Läufe zu. Die Suche ist eine spezielle Variante der allgemeinen lokalen Suche. Die Nachbarschaft wird nacheinander, Nachbar für Nachbar, evaluiert und gewechselt, sobald ein Nachbar eine höhere Güte aufweist.

Auch wenn beide Konfiguratoren schnell gute Parameterkonfigurationen ermitteln, so bilden sowohl der Racing-, als auch der Suchteil, rein sequentielle Verfahren. Da direkt der erste bessere Nachbar weiterverwendet wird, werden beim Racing immer nur zwei Konfigurationen miteinander verglichen. In FocusedILS werden sogar die Läufe, auf denen diese Konfigurationen evaluiert werden, nur schrittweise zugeteilt.

Insgesamt stellen Konfiguratoren mit lokalen Suchalgorithmen effiziente Verfahren der Parameteroptimierung dar. Die Suche verfeinert gefundene Konfigurationen stetig und kann in speziellen Varianten sogar lokale Minima vermeiden. Die Racing-Prozedur kann meist unabhängig von der Such-Prozedur verändert werden und bietet eine Vielzahl von möglichen Ansätzen. Einzig und allein der meist sequentielle Aufbau ist ein Nachteil. Gerade in einem so rechenaufwendigen Bereich wie der Parameteroptimierung, sollten alle zur Verfügung stehenden Prozessoren genutzt werden können.

1.2 Parallelisierungsgrad und -güte

Wie bereits erläutert, bieten die tatsächlichen Implementierungen von Konfiguratoren mit lokalen Suchalgorithmen, meist keine Möglichkeit der Parallelität und das, obwohl theoretisch eine Nachbarschaftssuche parallelisiert werden könnte, indem die Anzahl der betrachteten Nachbarn vergrößert wird. Allerdings muss entsprechend eine Racing-Prozedur implementiert werden, die entweder mehrere Nachbarn gleichzeitig oder viele Nachbarn sehr schnell, betrachten kann.

Im Gegensatz hierzu sind Racing Konfiguratoren und Konfiguratoren mit genetischen Algorithmen in der Theorie nativ parallel. Beide haben immer eine Menge an Parameterkonfigurationen, die evaluiert werden müssen. Umso größer die Menge dabei ist, umso höher ist auch der Paralellisierungsgrad. Bei genetischen Algorithmen bildet dabei die gleichzeitig stattfindende Kreuzung und Mutation einen Flaschenhalseffekt. Alle Läufe müssen abgeschlossen sein, um ein Ranking der Individuen der kompetitiven Gruppe aufzustellen. Bis dies der Fall ist, liegen immer größere Teile, der zur Verfügung stehenden Rechenressourcen, brach. Wenn also beispielsweise 100 Prozessoren zur Verfügung stehen und auf einem Prozessor der letzte Lauf einer Parameterkonfiguration für das Ranking läuft, sind 99 freie Prozessoren untätig. Da dies bei großen Populationsmengen allerdings nur einen Bruchteil der Konfigurationszeit beanschlagt, ist die Parallelisierungsgüte hoch. Das Äquivalent bildet bei Racing-Konfiguratoren die stetige Analyse der Parameterkonfigurationen durch Testverfahren, wie den Friedman-Test. Für eine Implementierung müsste prinzipiell also einzig und allein eine Routine für das Auslagern von Läufen auf freie Prozessoren geschrieben werden. Entsprechend verwunderlich ist es, dass dieser Aspekt noch keine allgemeine Beachtung gefunden hat.

Im Weiteren werden Implemetierungsansätze für parallele Konfiguratoren, die auf der iterativen lokalen Suche basieren, vorgestellt und analysiert. Dazu ist ein tieferer Einblick in die Methoden und Abläufe von BasicILS und FocusedILS grundlegend und wird im Folgenden beschrieben.

2 Konfiguratoren basierend auf Iterativer Lokaler Suche

Da die iterative lokale Suche die Grundlage sowohl für BasicILS, als auch für FocusedILS bildet, müssen einige Mechanismen detaillierter betrachtet werden. Zum Anfang einer jeden lokalen Suche werden r Zufallskonfigurationen erstellt und gegen die Startkonfiguration verglichen. Wurde eine besonders schlechte Startkonfiguration gewählt, kann so bereits, ohne eine Betrachtung der Nachbarschaft, eine teils signifikante Verbesserung erzielt werden. Desweiteren wird ein Mechanismus geboten, um lokale Minima zu vermeiden. Findet eine Parameterkonfiguration in der Nachbarschaft keinen besseren Nachbarn, wird die Nachbarschaftskonfiguration s-mal an einem Parameterwert verändert und damit eine neue Nachbarschaft geschaffen. Abbildung 2.1 zeigt den Ablauf einer solchen möglichen *Perturbation* mit $s = 3$. Mit einer vorher festgelegten Wahrscheinlichkeit, wird an Stelle einer Perturbation, eine ganz neue Konfiguration gewählt.

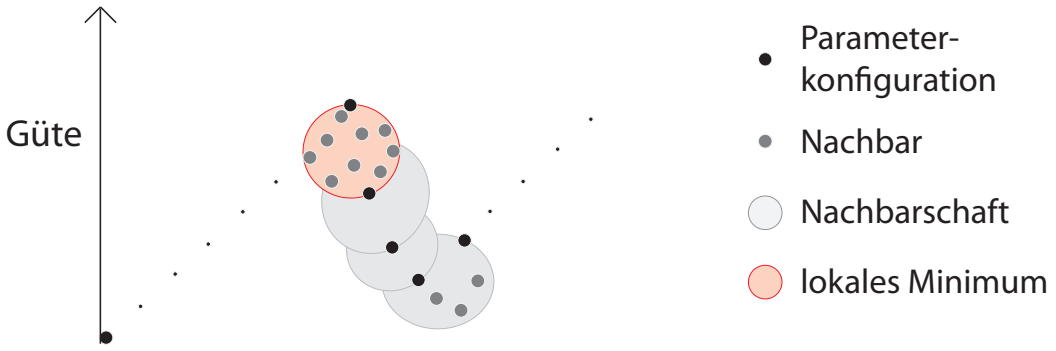

Abbildung 2.1: Ablauf einer Perturbation in der lokalen Suche

Um nun genauer auf die Racing-Prozeduren der beiden Konfiguratorenvarianten eingehen zu können, werden einige Grundbegriffe benötigt. Der *Inkumbent* bezeichnet die derzeit

beste Konfiguration in einem Konfiguratorlauf, der *lokale Inkumbent* die jeweils Beste, bis zu einem lokalen Minimum. Unter *Capping* versteht man die frühzeitige Beendigung von Läufen einer Parameterkonfiguration. Dabei besitzt jeder Lauf eine *Cutoff*-Zeit, die angibt, wieviel Zeit einem Lauf zugeteilt wird, um eine Lösung zu finden. Das optimale Verhältnis zwischen Cutoff-Zeit und der Anzahl an Probleminstanzen ist dabei ein schwieriges Problem. Werden die Cutoff-Zeit und die Anzahl an Probleminstanzen zu großmütig ausgewählt, kostet die Evaluation von schlechten Konfigurationen viel Zeit. Werden beide Variablen hingegen zu gering eingestellt, können selbst gute Konfigurationen keine Lösungen finden. In [7] werden für diese und andere Problematiken beim Einstellen eines Konfigurators Szenarien beschrieben und analysiert.

2.1 BasicILS

Sowohl bei BasicILS, als auch bei FocusedILS, findet der Vergleich zweier Parameterkonfigurationen in der sogennanten “better”-Methode statt. Sie erhält die zwei Konfigurationen als Parameter und gibt *true* zurück, wenn die erste Konfiguration die Bessere ist, *false* anderenfalls. Die Güte einer Konfiguration wird in Form der Kosten bestimmt. Sind die Kosten hoch ist die Güte gering. Um die Kosten zu bestimmen, werden beide Konfigurationen auf einer festgelegten Anzahl $\mathcal{N}$ an Probleminstanzen evaluiert. Diese Anzahl an Läufen erhält jede betrachtete Konfiguration in der gleichen Reihenfolge, auch als *Parcours* definiert.

```
1 public better(Configuration config1 , Configuration config2) {
2         float cost_config1 = costOnNRuns(config1);
3         float cost_config2 = costOnNRuns(config2);
4         return cost_config1 <= cost_config2;
5 }
```

In Fällen in denen Parameterkonfigurationen nach $x \leq \mathcal{N}$ Läufen bereits wesentlich schlechter als der Inkumbent sind, tritt der Capping-Mechanismus in Kraft. Nach jedem Lauf wird berechnet, ob die Kosten der Konfiguration $\geq$ den Kosten des Inkumbenten $*$ einem Capping-Faktor sind. Ist dies der Fall, werden der Konfiguration keine weiteren Läufe zugeteilt und die Kosten auf einen maximalen Wert - den erfolgreichen Läufen festgelegt.

2.2 FocusedILS

Im Gegensatz zu BasicILS ist in FocusedILS die Anzahl der Läufe, die jede Konfiguration erhält, nicht fest. Jede betrachtete Parameterkonfiguration erhält zwar anfangs einen Lauf, muss sich aber für weitere Läufe gegen andere Konfigurationen durchsetzen. Im folgenden Pseudocodefragment wird die “better”-Methode von FocusedILS vorgestellt, allerdings vereinfacht unter der Annahme, dass einer der Konfigurationen, in diesem Fall “recentBest”, bereits mehr Läufe zugeteilt bekommen wurde. Die Methode #Runs gibt die Anzahl der Läufe einer Konfiguration zurück, costOn#Runs entsprechend die Kosten auf dieser Anzahl. Genau wie bei BasicILS, absolviert jede Konfiguration den gleichen Parcours.

```
1  public better(Configuration recentBest , Configuration competitor {
2          addRun(competitor);
3          while (costOn#Runs(competitor, #Runs(competitor)) <
4                  costOn#Runs(recentBest, #Runs(competitor))) {
5                  if( #Runs(recentBest) <= #Runs(competitor)) {
6                          return false;
7                  }
8                  addRun(competitor);
9          }
10         addBBonusRuns(recentBest);
11         return true;
12 }
```

Eine neue Parameterkonfiguration kann also nur dann besser sein als eine Konfiguration mit l Läufen, wenn sie auf allen $1 \dots i$ Läufen, $i = 1 \dots l$ durchsetzen kann. Entsprechend deutlich wird die Relevanz der ersten Probleminstanzen im Parcours. Da jede neue Konfiguration auf diesen Instanzen verglichen wird, setzen sich Konfigurationen durch, die diese schnell lösen können. Parameterkonfigurationen, die auf diesen Instanzen schlecht abschneiden, werden verworfen, obwohl sie ggf. auf allen anderen Instanzen gute Ergebnisse erzielen würden. Um diese Problematik zu entgehen wurde in [8] ein Verfahren für eine zufällige Auswahl der Instanzen vorgestellt.

ROAR (Randomized Online Approximate Racing) unterscheidet sich insofern von FocusedILS, dass Konfigurationen direkt mit dem jeweiligen Inkumbenten verglichen werden. Einen Parcours gibt es dabei nur für den Inkumbenten. Konfigurationen erhalten ihre Läufe,

indem sie zufällig auf einer Probleminstanz, auf der der Inkumbent bereits lief, ausgeführt werden. So wird garantiert, dass Inkumbent und Herausforderer jeweils eine gemeinsame, aber variable Schnittmenge an Läufen haben, die Fokussierung auf spezielle Probleminstanzen aber vermieden wird.

Im Weiteren wird für alle dieser drei vorgestellten Konfiguratorenvarianten eine parallelle Implementierung präsentiert.

3 Parallele Konfiguratoren basierend auf Iterativer lokaler Suche

Für die Implementierungen der parallelen BasicILS -und FocusedILS-Varianten wurde die Programmiersprache JAVA [1] verwendet. Zum einen bietet JAVA durch seine Plattformunabhängigkeit eine hohe Portabilität, zum anderen wurde EDACC (Experiment Design and Administration for Computer Clusters) [3] verwendet, um Konfigurationsexperimente zu verwalten und Läufe auf zur Verfügung stehenden Prozessoren zu verteilen. EDACC bietet bereits eine Schnittstelle für JAVA-basierte Konfiguratoren an und ist selbst eine größtenteils auf JAVA basierende Anwendung.

Um einige allgemeine Problematiken im Umgang mit einer dynamischen Anzahl an Prozessoren zu vermeiden, besitzen alle Implementierungsansätze zwei spezielle Mechanismen. Der erste dient dazu, trotz schwankender Anzahl an verfügbaren Prozessoren, eine volle Auslastung der Rechenressourcen zu erzielen. Dazu wird in regelmäßigem Abstand die Auslastung ermittelt. Die Auslastung ergibt sich über die Anzahl der Läufe, die berechnet werden oder auf die Berechnung warten, im Verhältnis zur Prozessoranzahl, konkret $\frac{\text{Anzahl Läufe}}{\text{Anzahl Prozessoren}}$. Da für eine faire Verteilung jeder Lauf die gleiche Rechenleistung zur Verfügung gestellt bekommen muss, berechnet ein Prozessor immer nur einen Lauf. Tritt nun der Fall auf, dass die Auslastung ≤ 1 sein sollte, werden neue Läufe gestartet, indem die Nachbarschaftssuche um neue Nachbarn erweitert wird. Um dabei eine volle Ressourcenauslastung zu erlangen, sollte mindestens $\lceil \frac{\text{Anzahl Prozessoren} - \text{Anzahl Läufe}}{\text{Initialläufe einer neuen Parameterkonfiguration}} \rceil$ neue Nachbarn gestartet werden. Je nach Komplexität der Parameterumgebung kann das Erzeugen solcher neuen Parameterkonfigurationen sehr aufwendig sein und sollte entsprechend bereits vor potenziellen Engpässen stattfinden. Für diesen Fall haben alle Konfiguratoren einen sogenannten *Job Buffer*. Dieser hält stetig eine Menge von Konfigurationen mit zugehörigen Läufen vorrätig. Nimmt die Menge dieser Konfigurationen ab, werden umgehend neue

[1] 1991 im Auftrag von Sun Microsystems entwickelte objektorientierte Programmiersprache

erzeugt. Die Größe dieses Job Buffers ist frei definierbar, sollte aber idealerweise über der dem Zweifachen der Prozessoranzahl liegen.

Ein weiterer wichtiger Mechanismus dient der Reduzierung des benötigten Speichers. Eine Algorithmen-Konfiguration mit BasicILS oder FoucsedILS hatte nach einigen Tagen den Arbeitsspeicher bereits voll aufgebraucht und verlor immens an Performanz. Da EDACC bereits erstellte Parameterkonfigurationen und die Ergebnisse von Läufen effizient in einer SQL-Datenbank speichert, lag der Fokus auf der Reduzierung irrelevanter Informationen. Gerade bei stark parallelisierten Konfiguratoren können bei einer hoher Prozessorenanzahl und einer geringen Cutoff-Zeit in wenigen Stunden, Millionen von Läufen berechnet und gespeichert werden, welche entsprechend schnell die Datenbankkapazitäten aufbrauchen. Als Gegenmaßnahme werden Parameterkonfigurationen, die durch die Capping-Prozedur als unbrauchbar deklariert werden, samt aller zugehörigen Läufe aus der Datenbank entfernt. Dies könnte ohne weitere Maßnahmen allerdings dazu führen, dass gelöschte Parameterkonfigurationen wieder aufgegriffen werden, schließlich kann der Nachbar eines Nachbarn wieder die ursprüngliche Konfiguration darstellen. Um dies zu vermeiden, wurde eine Prüfsumme eingeführt, die jede Parameterkonfiguration eindeutig bestimmt. Eine Liste im Konfigurator verwaltet alle Prüfsummen von bereits betrachteten Konfigurationen und verhindert, dass eine zweimal gestartet wird. In der Datenbank verbleiben also nur die Inkumbenten und die ähnlich guten Konfigurationen. Die Liste ist dabei die einzige verwendete Datenstruktur, die im Verlaufe einer Algorithmen-Konfiguration stetig wächst. Alle anderen benötigten Daten werden dem Konfigurator über einen Cache bereit gestellt und freigegeben, sobald sie keine Verwendung mehr finden.

3.1 Parallel BasicILS

Parallel BasicILS unterscheidet sich von BasicILS, neben den bisher erwähnten Mechanismen, vor allem in der Racing-Prozedur. Anstatt auf die Läufe zweier Parameterkonfigurationen zu warten, wird eine ganze Liste von betrachteten Parameterkonfigurationen in periodischen Zeitabständen überprüft. Alle Läufe, die bis zu so einer Überprüfung abgeschlossen sind, werden den zugehörigen Konfigurationen zugewiesen und anschließend durch die Capping-Prozedur analysiert. Ist die Konfiguration vielversprechend genug, bleibt sie in der Liste erhalten und tritt gegen den lokalen Inkumbenten an, falls alle Läufe abgeschlossen sind. Anderenfalls wird sie aus der Liste und der Datenbank entfernt.

```
1  public better(List<Configuration> consideredConfigs) {
2          for(Configuration config : consideredConfigs) {
3                  checkForFinishedRuns(config);
4                  if(isCapped(config)) {
5                          deleteDataFromDatabase(config);
6                          consideredConfigs.remove(config);
7                          continue;
8                  }
9                  if(#Runs(config) == #Runs(localIncumbent)) {
10                         if(betterThanLocalIncumbent(config)) {
11                                 setNewLocalIncumbent(config);
12 }       }       }       }
```

Die “betterThanLocalIncumbent”-Methode vergleicht dabei lediglich die Kosten der Konfiguration und dem lokalen Inkumbent, die sich aus den vorhandenen Läufen ergeben. Der lokale Inkumbent durchläuft ein ähnliches Verfahren, um gegen den Inkumbenten verglichen zu werden. Die Capping-Prozedur entfällt in diesem Fall.

3.2 Parallel FocusedILS

Parallel FocusedILS überprüft, ähnlich wie Parallel BasicILS, immer eine Liste von Konfigurationen. Dabei wird die Capping-Prozedur allerdings erst durchlaufen, wenn alle Läufe einer Konfiguration abgeschlossen sind. Dies führt dazu, dass das verfrühte Verwerfen von guten Konfigurationen mit einer größeren Zahl von Bonusläufen abnimmt. Bonusläufe bezeichnen die Läufe, die eine Konfiguration zugeteilt bekommt, wenn sie sich gegen eine andere Konfiguration, hier den lokalen Inkumbenten, durchsetzen kann. Die Anzahl dieser Bonusläufe wird in Parallel FocusedILS über eine Exponentialverteilung ermittelt. Eine Konfiguration erhält im i-ten Vergleich 2^{i+1} zusätzliche Läufe. So können vielversprechende Konfigurationen schneller gegen den lokalen Inkumbenten verglichen werden. Der lokale Inkumbent erhält dabei jeweils einen zusätzlichen Lauf für jede Konfiguration, die er schlägt. Da dies in kurzer Zeit bereits eine große Anzahl an Läufen sein kann, wird eine obere Grenze für den Parcours festgelegt.

```
1  public better(List<Configuration> consideredConfigs) {
2          for(Configuration config : consideredConfigs) {
3                  checkForFinishedRuns(config);
4                  if(!allRunsAreFinished(config)) continue;
5                  if(isCapped(config)) {
6                          deleteDataFromDatabase(config);
7                          consideredConfigs.remove(config);
8                          continue;
9                  }
10                 if(costOn#Runs(config, #Runs(config)) <
11                         costOn#Runs(localIncumbent, #Runs(config)){
12                         if(#Runs(config) == #Runs(localIncumbent)){
13                                 setNewLocalIncumbent(config);
14                         } else {
15                                 addBonusRuns(config);
16 }       }       }       }
```

Um der Problematik der Parcoursfokussierung zu entgehen, bietet parallel FocusedILS eine Implementierung der ROAR-Prozedur an. So kann optional eingestellt werden, dass neue Konfigurationen nicht immer schrittweise den Parcour absolvieren, sondern zufällig auf Probleminstanzen des lokalen Inkumbenten ausgeführt werden. Der lokale Inkumbent hat dazu immer mindestens gleich viele Läufe, wie die betrachteten Konfigurationen und der Inkumbent mindestens so viele, wie der lokale Inkumbent. Da dies implizit bei jedem Start eines Laufes sichergestellt wird, gewinnt der lokale Inkumbent keine weiteren Läufe bei Vergleichen mit anderen Konfigurationen.

4 Experimente

Um Parallel BasicILS und Parallel FocusedILS mit und ohne ROAR-Prozedur hinsichtlich ihrer Verwendbarkeit beurteilen zu können, wurden vier Experimente durchgeführt, die sich jeweils in zwei Teile gliederten. In beiden Teilen stand eine feste Rechenkapazität von 384, bzw. 576 Stunden zur Verfügung. Im ersten Teil bekam ein Konfiguratorlauf die volle Rechenzeit zugewiesen. Im zweiten Teil wurde die Rechenzeit auf 24 einzelne Konfiguratorenläufe aufgeteilt. Dadurch konnte eine Aussage über die Ergebnisqualität hinsichtlich der verwendeten Rechenkapazitäten getroffen werden. Als Startkonfigurationen wurden bereits optimierte Parameterkonfigurationen verwendet. Der Capping-Faktor lag bei 1,5. Für Parallel BasicILS wurde der Parcours auf 250 Probleminstanzen festgelegt, bei parallel FocusedILS lag die obere Grenze bei 1000. Um eine möglichst generelle Aussage über die Kosten einer Konfiguration treffen zu können, wurden die gefundenen Konfigurationen auf einer weiteren festgelegten Menge von Probleminstanzen, auch als Testinstanzen bezeichnet, evaluiert. Als Kostenfunktion wurde PAR10 gewählt. Sie teilt Läufen die Kosten $Cutoff * 10$ zu, falls in der Cutoff-Zeit keine Lösung gefunden wird. Parameterkonfigurationen, mit denen mehr Probleminstanzen gelöst werden können, sollen so einen Vorteil erhalten.

4.1 Problemspezifikation

Die zu konfigurierenden Algorithmen bildeten die SAT-Solver Captain Jack [9] und Sparrow [4]. Sparrow besitzt insgesamt sieben Parameter, von denen vier konfiguriert wurden. Dabei handelte es sich um zwei kontinuierliche und zwei diskrete Parameter. Die Probleminstanzen bildeten 3sat10k-Instanzen für die jeweils eine Cutoff-Zeit von 75 Sekunden bereit gestellt wurde. Aufgrund der hohen Cutoff-Zeit stand in diesem Experiment in beiden Teilen eine Rechenkapazität von 576 Stunden zur Verfügung.

Die anderen drei Experimente verwendeten den Captain Jack SAT-Solver, der mit seinen 64 Parametern einen wesentlich größeren Parameterraum aufweist. Von diesen 64 Parametern wurden 33 Parameter, darunter 1 Diskreter, 5 Ordinale und 27 Kontinuierliche, konfiguriert. Die Probleminstanzen bildeten 5sat100-Instanzen mit einer Cutoff-Zeit von 5 Sekunden, 7sat60-Instanzen mit einer Cutoff-Zeit von 20 Sekunden und 3sat1k-Instanzen, für die eine Cutoff-Zeit von 30 Sekunden zur Verfügung stand. Die Experimente mit den 5sat100 -und 7sat60-Probleminstanzen erhielten eine Rechenkapazität von jeweils zweimal 384 Stunden, das Experiment mit den 3sat1k-Instanzen zweimal 576 Stunden.

4.2 Hardware und Systeme

Für die Berechnungen wurde die Rechenkapazität des bwGrid [1] genutzt. “Das bwGRiD ist eine dezentrale Ansammlung von Rechenknoten, die von acht Universitäten des Landes Baden-Württemberg betrieben werden” ([1]) und stellt über 11000 Prozessoren bereit. Für die Experimente wurden ausschließlich Rechnerknoten verwendet, die mit zwei Quad-Core Intel Xeons mit 2,8 GHz ausgestattet waren. Die Bereitstellung solcher Rechnerknoten geschah über standortspezifische Login-Knoten des bwGrids, auf denen Scientific Linux v5.5 [1] lief.

Die berechneten Ergebnisse wurden über eine EDACC-Prozedur direkt in eine mySQL-Datenbank gespeichert. Die Datenbank läuft auf einem Server mit dem Betriebssystem Ubuntu v10.04. Auf dem selbigen Server wurden auch die Konfiguratoren ausgeführt, um unnötige Paketumlaufzeiten zu vermeiden.

[1] Erstmals 2004 erschienendes Linux-basiertes Betriebssystem, welches durch seine wissenschaftlichen Softwarekomponenten besonders in der Forschung Anwendung findet

5 Ergebnisse

Nach weit über 10000 verbrauchten Rechenstunden und mehr als 300 analysierten Konfiguratorläufen, konnten einige relevante Erkenntnisse über die Nützlichkeit und Besonderheit der vorgestellten parallelen automatischen Algorithmen-Konfiguratoren festgehalten werden. Wichtig für diese Erkenntnisse war die Tatsache, dass sich eine Tendenz in der Relation der Rechenzeit zu den Konfigurationskosten abbildete. Anfangs wurden schnell bessere Konfigurationen gefunden, mit der Zeit nahm dies allerdings exponentiell ab. Besonders die ersten Rechenstunden sind also von entscheidender Bedeutung. Trotzdem sollte kontinuierlich eine sukzessive Verbesserung sichtbar sein. Die genauen Ergebnisse werden in den folgenden Unterkapiteln dargestellt. Der Verlauf des Inkumbenten wird dabei grafisch veranschaulicht. Dabei wurde aufgrund der unterschiedlichen Rechenzeiten in den beiden Experimentteilen, eine logarithmische Skalierung der X-Achse verwendet. Um die Übersicht des Graphen zu gewährleisten wurden die 24 Einzelläufe zusammengefasst. So bezeichnet “seq-top$X$” den Kostendurchschnitt der besten $X\%$ der Konfiguratorläufe. Da es sich bei Konfiguratoren um Ansätze des maschinellen Lernens handelt, werden zwei Mengen an Probleminstanzen unterschieden. Die Erste stellt Trainingsinstanzen dar, auf denen ein Konfigurator “lernt” und die zweite Menge definiert Testinstanzen auf denen die Ergebnisse kontrolliert werden. In den weiteren Experimenten bestanden beide Mengen aus jeweils 250 Probleminstanzen.

5.1 Captain Jack - 5sat100

Die 5sat100-Probleminstanzen stellten mit ihren 100 Variablen und 2111 Klauseln bei einer geringen Cutoff-Zeit von 5 Sekunden, die größte Herausforderung für die Konfiguratoren dar. Die Startkonfiguration löste einen Lauf in durchschnittlich bereits 0,249 Sekunden. Entsprechend schnell mussten neue Parameterkonfigurationen und Läufe erzeugt werden, um keine Rechenressourcen zu verschwenden.

Parallel BasicILS erzielte dabei im 384h-Lauf eine Auslastung von 15%. Die 24 einzelnen Läufe lagen mit durschnittlich 8% sogar gerade mal bei der Hälfte. Der Grund dafür liegt in der bereits sehr guten Startkonfiguration in Verbindung mit dem geringen Capping-Faktor. Neue Konfigurationen werden bereits verworfen, wenn sie für zwei Probleminstanzen keine Lösungen finden. Entsprechend schnell müssen neue Parameterkonfigurationen erstellt werden und es kommt zu Engpässen trotz Job Buffers. Desweiteren findet die Überprüfung der Auslastung in einem 500ms-Zyklus statt. Bei Laufzeiten von unter 250 ms können so signifikante Verzögerungen entstehen. Trotz der geringen Auslastung, erzielte Parallel BasicILS durchgehed brauchbare Ergebnisse. Besonders der 384h-Lauf erreichte auf den Trainingsinstanzen mit 0,188 sehr geringe Kosten und damit ein sehr gutes Ergebnis gegenüber der Startkonfiguration mit 0,249. Aus Abbildung 5.1 wird ersichtlich, dass die zusätzliche Rechenzeit des 384h-Laufes bis zum Ende zu einer Verfeinerung der Parameterkonfiguration führt. Trotz der geringeren Rechenzeit finden auch 23, der 24 Einzelläufe

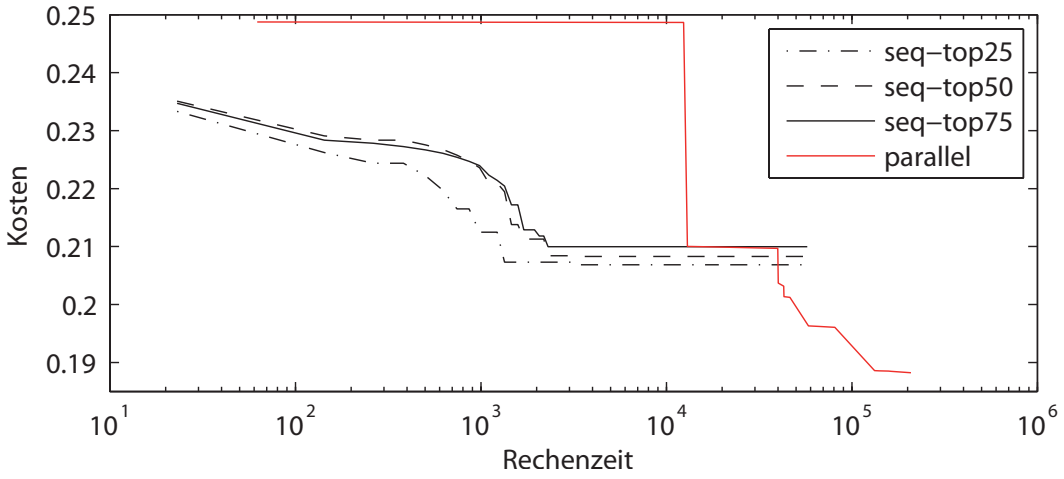

Abbildung 5.1: Captain Jack - 5sat100-Train: Konfiguration durch Parallel BasicILS

eine Parameterkonfiguration, die die Startkonfiguration schlägt. Auf den Testinstanzen hingegen schaffen es nur zwei der Einzelläufe eine Verbesserung zu erzielen. Der 384h-Lauf setzt sich auch auf den Testinstanzen durch, allerdings mit wesentlich höherern Kosten. Die genauen Werte finden sich in Tabelle 5.1.

Parallel FocusedILS weist mit einer Auslastung von 12% im 384h-Lauf und durchschnittlich 5% in den Einzelläufen, noch schlechtere Werte auf als Parallel BasicILS. Dies liegt, neben den bereits angesprochenen Gründen, stark an der Parcoursfokussierung. Wurde erstmals eine Parameterkonfiguration gefunden, die die ersten beiden Parcourinstanzen

sehr schnell lösen kann, werden neue Konfigurationen mit hoher Wahrscheinlichkeit durch die Capping-Prozedur gelöscht. In vielen Läufen beweist der Inkumbentenverlauf genau diese Tendenz. Im 14. Einzellauf besaß der letzte eingetragene Inkumbent beispielsweise Kosten von 0.009497 auf den ersten beiden Instanzen. Entsprechend schlechter fallen auch die Ergebnisse aus. Mit 0,284 findet der 384h-Lauf keine Parameterkonfiguration, die besser ist als die Startkonfiguration. Die Einzelläufe erzielen, wie aus Abbildung 5.2 erkennbar, sogar noch schlechtere Ergebnisse. Ähnlich ist es auf den Testinstanzen.

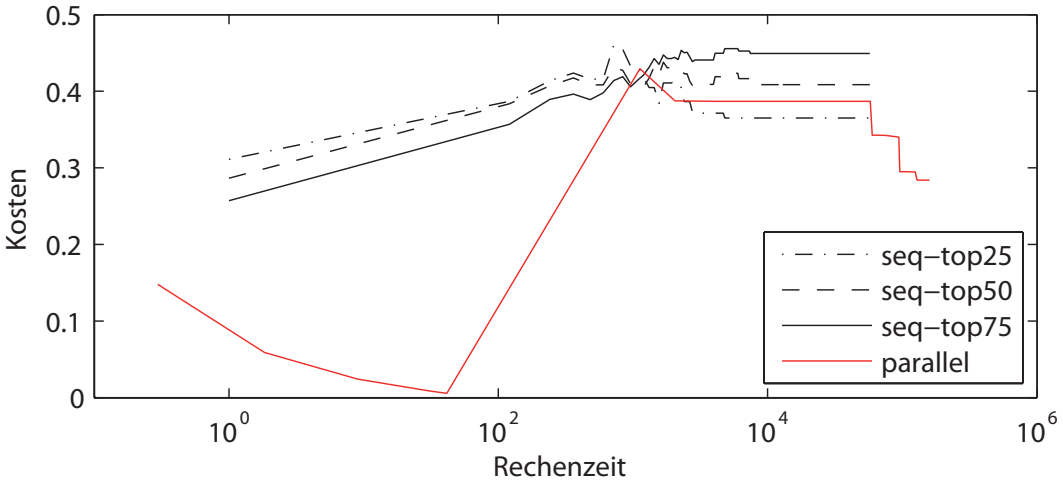

Abbildung 5.2: Captain Jack - 5sat100-Train: Konfiguration durch Parallel FocusedILS

Trotz der geringeren Güte, weist auch Parallel FocusedILS die Tendenz auf, die Ergebnisse schrittweise zu verbessern. Besonders deutlich wird dies im 384h-Lauf, ab ca. 10^5 Sekunden.

Wird bei Parallel FocusedILS die ROAR-Prozedur aktiviert, verbessert sich die Auslastung auf 15% im 384h-Lauf und auf 14% in den Einzelläufen. Die gefundenen Parameterkonfigurationen sind leicht besser (Abbildung 5.3), in der Tendenz allerdings sehr ähnlich. Durch das schlechtere Abschneiden auf den Trainingsinstanzen gleicht sich die Ergebnisqualität aus.

Insgesamt erzielen sowohl die Einzelläufe, als auch die 384h-Läufe aller drei Konfiguratoren gute Ergebnisse, auch wenn nicht alle die Startkonfiguration verbessern können. Besonders die Erkenntnis, dass die zusätzliche Rechenzeit in den 384h-Läufen zu einer stetigen, wenn auch langsameren, Verbesserung führt, unterstützt die Notwendigkeit von stark parallelisierten automatischen Algorithmen-Konfiguratoren.

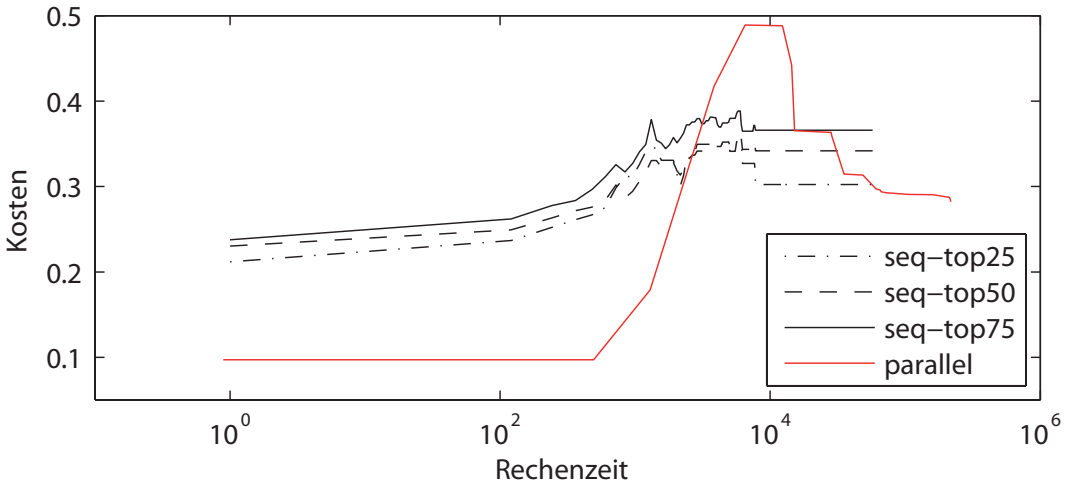

Abbildung 5.3: Captain Jack - 5sat100-Train: Konfiguration durch Parallel FocusedILS mit ROAR

	parallel		seq-top25		seq-top50		seq-top75	
	Train	Test	Train	Test	Train	Test	Train	Test
pBasic	0,188	0,416	0,207	0,53	0,208	0,506	0,21	0,505
pFocused	0,284	0,48	0,365	0,478	0,409	0,496	0,449	0,496
pFocused mit ROAR	0,282	0,53	0,302	0,553	0,342	0,521	0,366	0,521

	Train	Test
default	0,2488	0,439

Tabelle 5.1: Captain Jack - 5sat100: Trainings- und Testergebnisse

5.2 Captain Jack - 7sat60

Bei den 7sat60-Probleminstanzen, mit ihren 60 Variablen und 5267 Klauseln, lag die Cutoff-Zeit bereits bei 20 Sekunden. Entsprechend höher fiel auch die durchschnittliche Auslastung aus. Parallel BasicILS erreichte mit knapp 50% im 384h-Lauf eine ansehnliche Auslastung, genau wie die Einzelläufe mit 36%. Die doch recht hohe Differenz lässt sich mit der Evaluation der Startkonfiguration erklären. Da die Einzelläufe gerade einmal eine Ausführungszeit von acht Minuten aufweisen, beeinflusst die anfängliche Evaluation der Startkonfiguration, bei der nicht alle Ressourcen voll ausgelastet werden können, das Ergebnis signifikant.

Sämtliche Konfiguratorenläufe mit Parallel BasicILS verbesserten die Startkonfiguration deutlich, wie in Abbildung 5.4 ersichtlich, um durchschnittlich eine Sekunde, also ca. 57%. Auf den Testinstanzen wendete sich die Güte der Ergebnisse aber ins Gegenteil und die Startkonfiguration stellte die besten Kosten. Generell wies die Startkonfiguration stabilere Kosten auf, während sich die Kosten der ermittelten Parameterkonfigurationen auf den Testinstanzen teils verdreifachten. Trotz der geringeren Rechenzeit eines Einzellaufes, erzielte

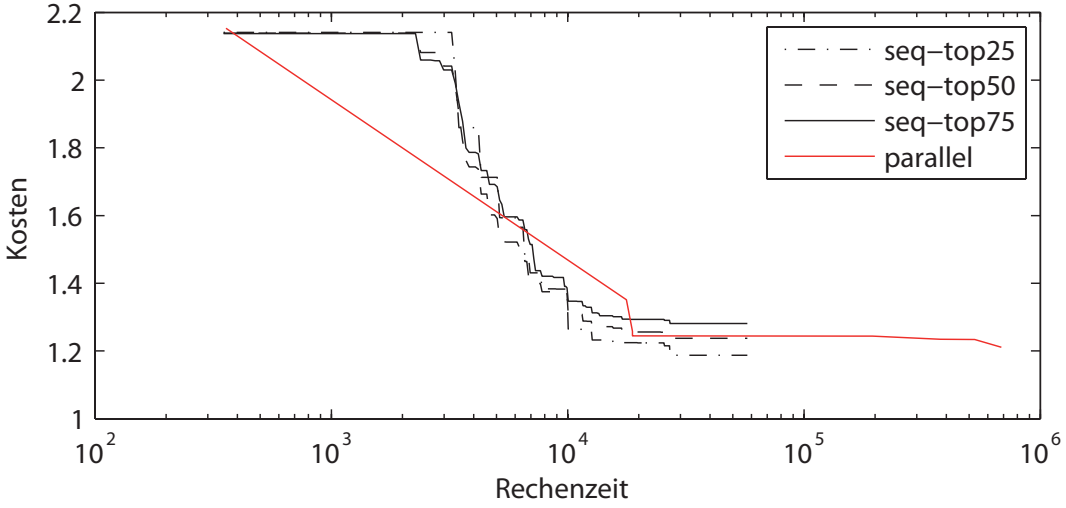

Abbildung 5.4: Captain Jack - 7sat60-Train: Konfiguration durch Parallel BasicILS

jeder Einzellauf ähnlich gute, wenn nicht sogar bessere Ergebnisse wie der 384h-Lauf. Dies ist zum Teil auf die schlechte Startkonfiguration auf den Trainingsinstanzen zurückzuführen. Dadurch gab es eine Vielzahl von besseren Konfigurationen, von denen Einige entsprechend schnell gefunden wurden. Einen weiteren Faktor stellt der Zufall. Da neben der Nachbarschaftssuche auch immer mit einer gewissen Wahrscheinlichkeit neue Parameterkonfigurationen betrachtet werden, kann der 384h-Lauf zufälligerweise eher schlechte Konfigurationen erhalten haben. Die schrittweise Verbesserung fand entsprechend langsam statt.

Bei Parallel FocusedILS basieren die ähnlichen Ergebnisse auf anderen Umständen. Der 384h-Lauf wählte mit einer auf zwei Läufen evaluierten Parameterkonfiguration eine Konfiguration (in Abbildung 5.5 bei ca. 100 Sekunden), die sich auf 1000 Läufen als äußerst schlechte Wahl herausstellte. Trotzdem setzte sie sich aber aufgrund der Parcousfokussierung weiterhin gegen potenzielle gute Konfigurationen durch. Die Einzelläufe wiesen keine so starke Parcoursfokussierung auf und dementsprechend fielen ihre Ergebnisse deutlich

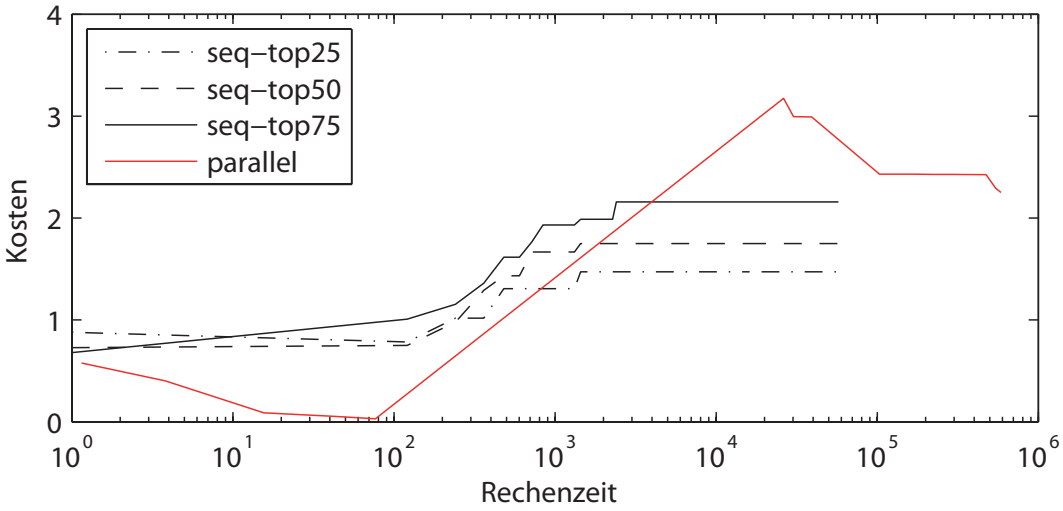

Abbildung 5.5: Captain Jack - 7sat60-Train: Konfiguration durch Parallel FocusedILS

besser aus. Auf den Testinstanzen zeichnete sich allerdings ein ähnliches Bild wie bei Parallel BasicILS ab. Zwar kamen die Kosten einiger Einzelläufe den Kosten der Startkonfiguration sehr nahe, aber verbessert werden konnten sie nicht. Die Auslastung lag bei 60%, bzw. 43%. Mit Roar konnte sogar eine Auslastung von 55% in den Einzelläufen erzielt werden. Trotz der höheren Auslastung ähneln die Ergebnisse denen von Parallel FocusedILS ohne Roar allerdings stark. Zwar werden auf den Trainingsinstanzen durchweg bessere Parameterkonfiguration gefunden (Abbildung 5.6), auf den Testinstanzen hingegen konnte nur ein Einzellauf eine minimale Verbesserung von 1% erzielen.

Insgesamt erzielte Parallel BasicILS zum zweiten Mal ein besseres Ergebnis auf den Trainingsinstanzen. Die Differenz zu den Ergebnissen auf den Testinstanzen ist allerdings höher als bei Parallel FocusedILS. Ein Grund dafür könnte die größere Anzahl an Probleminstanzen sein auf denen die Konfigurationen bei Parallel FocusedILS evaluiert werden.

5.3 Captain Jack - 3sat1k

Im letzten Experiment mit dem Sat-Solver Captain Jack, wurde auf den 3sat1k-Probleminstanzen mit 1000 Variablen und 4260 Klauseln, eine Cutoff-Zeit von 30 Sekunden zur Verfügung gestellt. Die Tendenz aus den ersten beiden Experimenten, dass aus einer größeren

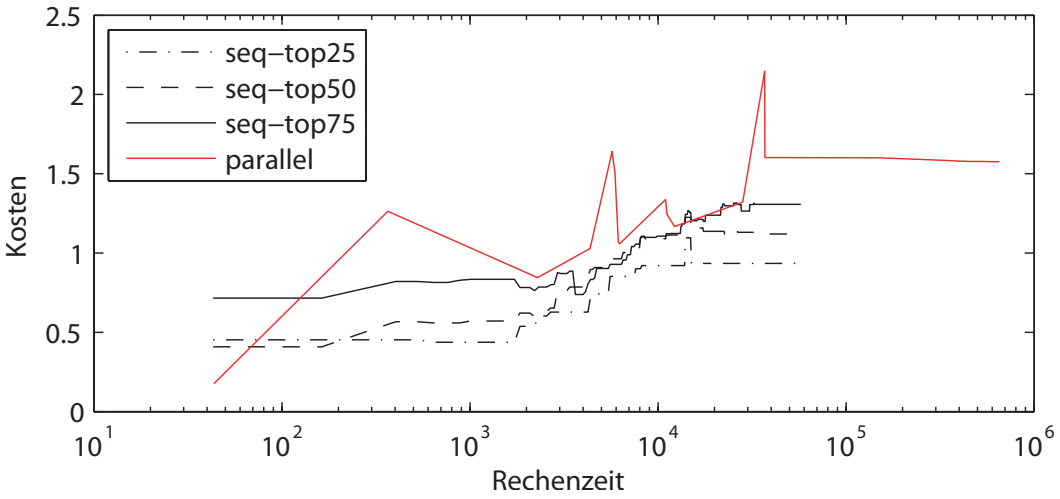

Abbildung 5.6: Captain Jack - 7sat60-Train: Konfiguration durch Parallel FocusedILS mit ROAR

	parallel		seq-top25		seq-top50		seq-top75	
	Train	Test	Train	Test	Train	Test	Train	Test
pBasic	1,21	2,894	1,19	2,843	1,238	2,7	1,281	2,655
pFocused	2,249	2,943	1,471	2,543	1,748	2,695	2,157	2,767
pFocused mit ROAR	1,576	2,371	1,108	2,82	1,222	2,639	1,376	2,637

	Train	Test
default	2,153	2,198

Tabelle 5.2: Captain Jack - 7sat60: Trainings- und Testergebnisse

Cutoff-Zeit eine höhere Auslastung folgt, konnte dabei auch hier beobachtet werden. Der Grund dafür liegt allerdings viel mehr in der durchschnittlichen Rechenzeit, die ein Lauf benötigt, um eine Lösung zu finden, als in der Cutoff-Zeit selber.

Parallel BasicILS erreichte eine Auslastung von 66% im 384h-Lauf und durchschnittlich 54% im Einzellauf. Die gefundenen Parameterkonfiguration aus dem 384h-Lauf wies mit 11,02 niedrigere Kosten auf, als die Startkonfiguration mit 24,211 und sämtliche Einzelläufe, sowohl auf den Trainings-, als auch auf den Testinstanzen, siehe Tabelle 5.3. Die Anzahl der gefundenen Inkumbenten sank allerdings erheblich im Vergleich zu den 5sat100 und 7sat60 Läufen. Durch den höheren Aufwand eine Parameterkonfiguration zu evaluieren, sinkt die Anzahl der betrachteten Konfigurationen. Damit allerdings auch der Zufallsfaktor. Entsprechend setzt sich ein Lauf mit größerer Rechenzeit mit höherer Wahrscheinlichkeit

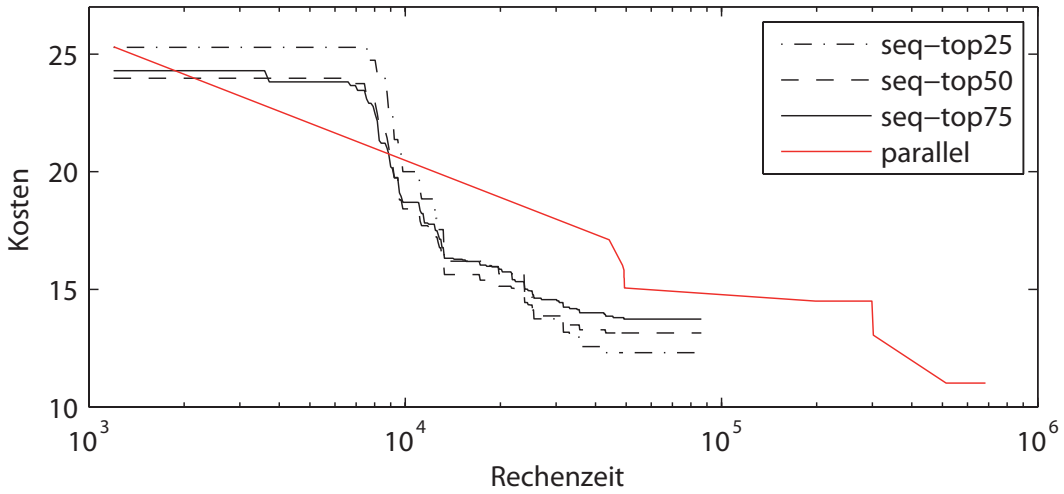

Abbildung 5.7: Captain Jack - 3sat1k-Train: Konfiguration durch Parallel BasicILS

gegenüber vielen Einzelläufen mit geringeren Rechenkapazitäten durch. Trotzdem erzielten selbst die Einzelläufe auf den Trainingsinstanzen sehr gute Ergebnisse, wie in Abbildung 5.7 dargestellt. Zwei besaßen sogar auf den Testinstanzen geringere Kosten als die Startkonfiguration. Die Ergebnisse von Parallel FocusedILS unterlagen hingegen wieder

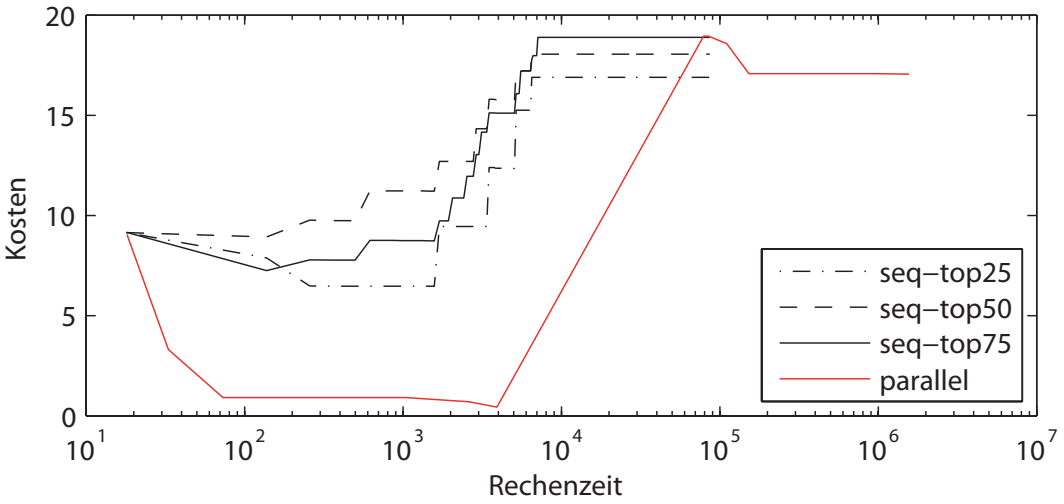

Abbildung 5.8: Captain Jack - 3sat1k-Train: Konfiguration durch Parallel FocusedILS

der Parcoursfokussierung. In Abbildung 5.8 wird deutlich, dass der 384h-Lauf um ca. 4000 Sekunden sehr gute Lösungen für die ersten beiden Parcoursinstanzen ermittelte. Nach 998 Bonusläufen wurde die obere Parcoursgrenze erreicht und die Kosten derselben Konf-

guration verdoppelten sich. Die Rechenkapazitäten konnten dabei zu 85% genutzt werden. Wie bereits in den ersten beiden Experimenten, erzielte Parallel FocusedILS mit ROAR leicht bessere Ergebnisse. Zwar scheinen die Ergebnisse des 384h-Laufes aus Abbildung 5.9 im Vergleich zu den Einzelläufen nicht allzu gut zu sein, aber aufgrund der geringen Evaluationsläufe der gefundenen Parameterkonfigurationen der Einzelläufe, erzielt die beste Konfiguration vom 384h-Lauf auf den Trainingsinstanzen geringere Kosten. Es wird sogar mit 25,494 knapp die Startkonfiguration mit 26,143 geschlagen. Die Auslastung liegt bei 86%.

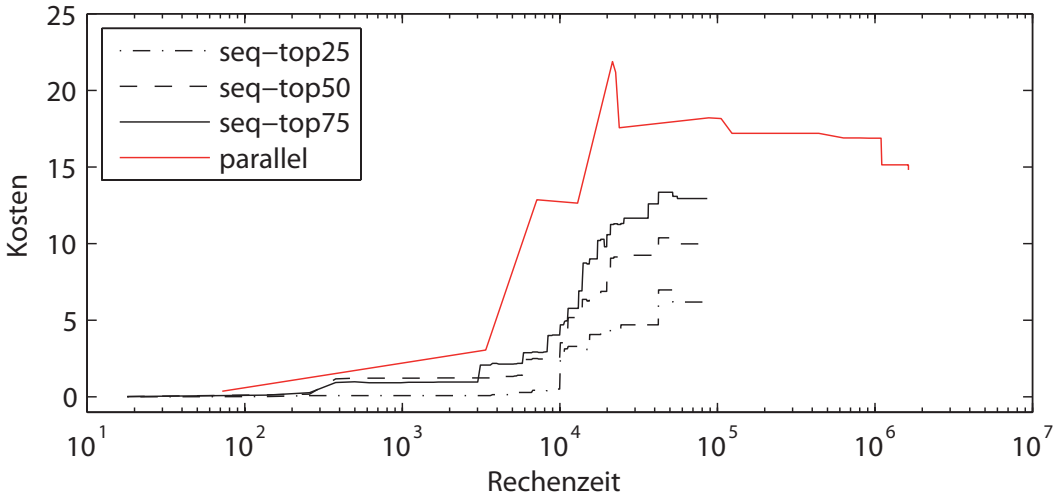

Abbildung 5.9: Captain Jack - 3sat1k-Train: Konfiguration durch Parallel FocusedILS mit ROAR

Alles in allem scheint Parallel BasicILS die besten Ergebnisse für Captain Jack zu liefern, Parallel FocusedILS hingegen die Stabilsten. Der Mechanismus der Bonusläufe bei Parallel FocusedILS erschwert den Wechsel des Inkumbenten unnötig, da bereits in sehr geringer Zeit eine Konfiguration, die auf die ersten beiden Parcourinstanzen fokussiert ist, die maximale Parcoursgrenze erreicht. Die Auslastung ist in allen Experimenten situationsbedingt annehmbar. Da nicht nur die konfigurierbaren Algorithmen, sondern der Konfigurator selber auch noch eine Vielzahl von variablen Parametern hat, gibt es auch in diesem Fall ein hohes Optimierungspotenzial. In diesem Bereich wird in EDACC bereits an der Möglichkeit gearbeitet, einen Konfigurator durch einen Konfigurator konfigurieren zu lassen.

	parallel		seq-top25		seq-top50		seq-top75	
	Train	Test	Train	Test	Train	Test	Train	Test
pBasic	11,02	25,43	12,31	29,2	13,15	28,38	13,74	27,98
pFocused	17,05	28,13	16,9	28,24	18,05	28,43	18,88	28,59
pFocused mit ROAR	14,81	25,49	4,72	28,24	9,88	27,58	13,13	27,57

	Train	Test
default	24,211	26,143

Tabelle 5.3: Captain Jack - 3sat1k: Trainings- und Testergebnisse

5.4 Sparrow - 3sat10k

Das letzte Experiment bildet mit dem Sparrow auf 3sat10k-Instanzen, mit 10000 Variablen und 42000 Klauseln, auch das Schwierigste. Die Cutoff-Zeit von 75 Sekunden und die durchschnittliche benötigte Zeit von 128,73 Sekunden für die Lösung einer Probleminstanz mit der Startkonfiguration, verdeutlichen dies. Wie direkt in der Abbildung 5.10 ersichtlich ist, findet keiner der Einzelläufe mit Parallel BasicILS auch nur eine etwas bessere Parameterkonfiguration. Selbst der 384h-Lauf findet nur einen neuen Inkumbenten. Dieser

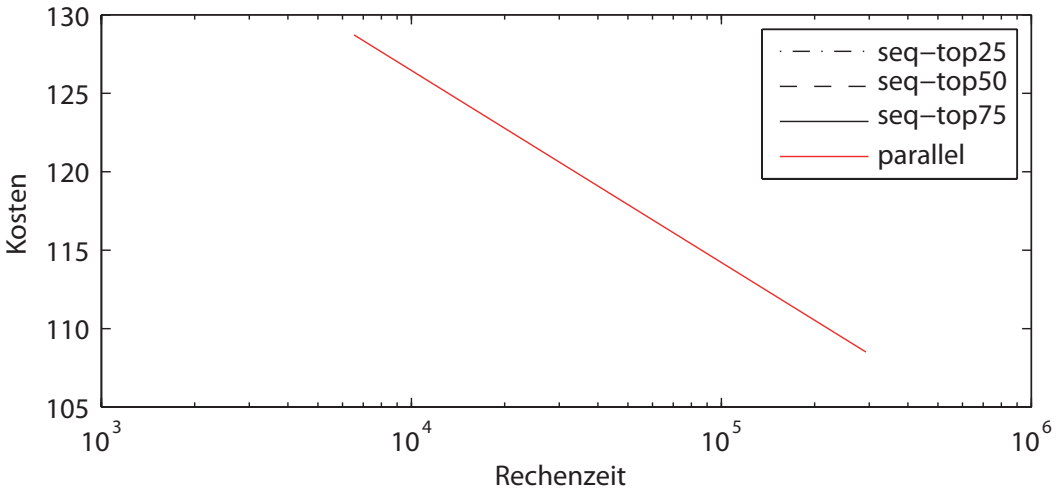

Abbildung 5.10: Sparrow - 3sat10k-Train: Konfiguration durch Parallel BasicILS

kann sich allerdings auch auf den Trainingsinstanzen gegen die Startkonfiguration behaupten. Die Auslastung liegt bei geringen 57%. Parallel FocusedILS erreicht eine Auslastung von 75%, mit ROAR sogar von 96%. Von den 24 Einzelläufen von Parallel FocusedILS fan-

den nur vier eine bessere Konfiguration. Drei davon hatten sogar geringere Kosten als der

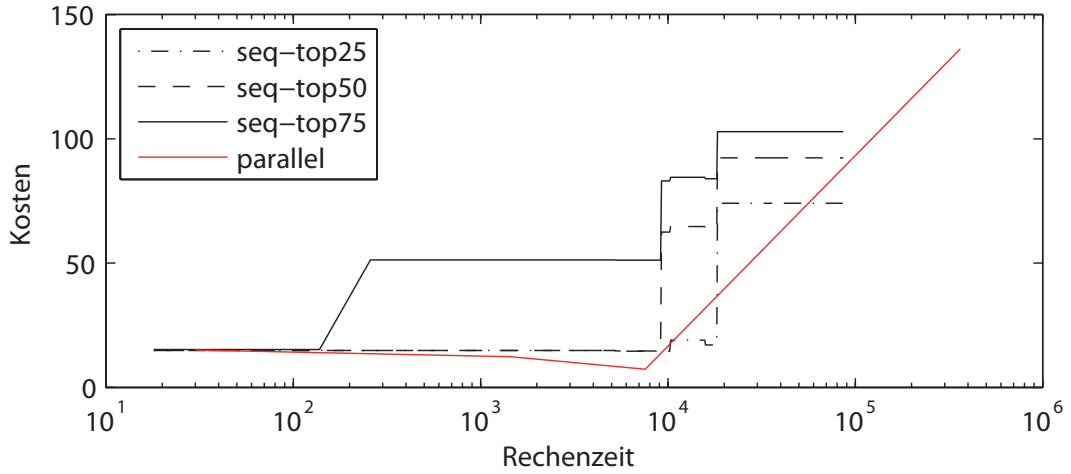

Abbildung 5.11: Sparrow - 3sat10k-Train: Konfiguration durch Parallel FocusedILS

Inkumbent von Parallel BasicILS, vgl. Abbildung 5.10. Noch bessere Ergebnisse liefern die Konfiguratorenläufe mit ROAR-Prozedur. Sie können eine 12%ige Verbesserung der Parameterkonfiguration vorweisen und schlagen deutlich, sowohl auf den Trainings-, als auch auf den Testinstanzen die Ergebnisse von Parallel BasicILS, siehe Tabelle 5.4. Im Graph

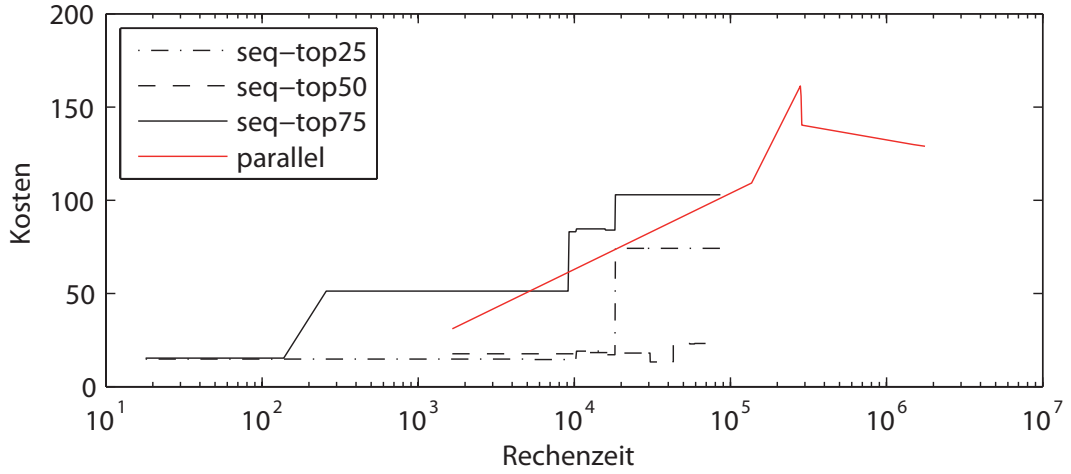

Abbildung 5.12: Sparrow - 3sat10k-Train: Konfiguration durch Parallel FocusedILS mit ROAR

(Abbildung 5.12) erzielen zwar die Einzelläufe bessere Ergebnisse als der 384h-Lauf, aber dies findet seinen Grund wieder in der geringen Anzahl von Evaluationsläufen.

Insgesamt bildet das vierte Experiment das einzige Experiment, indem Parallel FocusedILS die Ergebnisse von Parallel BasicILS schlägt. Dies liegt besonders an der Effizienz der Capping-Prozedur. Das sofortige Evaluieren auf allen Parcoursinstanzen kostet unnötige Rechenzeit und reduziert neben den Suchradius auch die Suchtiefe dramatisch. Parallel FocusedILS erzielt mit ROAR durchgehend bessere Ergebnisse und sollte standardmäßig integriert werden.

	parallel		seq-top25		seq-top50		seq-top75	
	Train	Test	Train	Test	Train	Test	Train	Test
pBasic	108,51	148,49	-	-	-	-	-	-
pFocused	126,74	140,35	74,11	146	92,32	144,92	102,92	142,63
pFocused mit ROAR	128,98	125,64	14,19	147,26	31,92	142,18	78	141,9

	Train	Test
default	128,732	150,762

Tabelle 5.4: Sparrow - 3sat10k: Trainings- und Testergebnisse

6 Ausblick

Auch wenn bereits zufriedenstellende Ergebnisse mit den hier vorgestellten parallelisierten Algorithmen-Konfiguratoren erzielt werden konnten, so gibt es doch eine Menge Optimierungspotenzial. Zum Einen muss für die Frage nach der richtigen Konfiguration eines Konfigurators eine Antwort gefunden werden, zum Anderen sollten die Such- und Racing-Prozeduren stärker gegeneinander abgewägt werden. Zum Beispiel gibt es Konfiguratorenvarianten, die wie die Racing-Konfiguratoren stochastische Tests verwenden, um Parameterkonfiguratoren zu vergleichen, eingebunden in die iterative lokale Suche. Dabei stellt die iterative lokale Suche auch nur eine von vielen vorstellbaren Such-Prozeduren dar. Die Möglichkeiten bei der Entwicklung eines Algorithmen-Konfigurators sind so gut wie unbegrenzt.

Eine Tendenz in Richtung Parallelität zeichnet sich dabei heute schon ab. So existieren bereits weitere parallele Varianten von BasicILS und FocusedILS und beinahe für jede Konfiguratorenart werden zumindest schon theoretisch, parallele Ansätze behandelt. Aufgrund des hohen Nutzens, wird vermutlich diese Nischentechnologie auch weiterhin immer stärker an Bedeutung gewinnen. Da es um riesige Rechenkapazitäten geht und damit um wertvolle Zeit, sollte der einmalige Aufwand einer Algorithmen-Konfiguration grundsätzlich immer auf sich genommen werden. Zeit, die nicht in das Konfigurieren von Algorithmen fließt, kann so in die Algorithmen selber investiert werden.

Alles in allem ist ein wachsendes Interesse, vor allem von wirtschaftlicher Seite, an dieser Thematik wahrzunehmen. Die Anzahl an neuen Konzepten und Ideen steigt kontinuierlich und setzt Konfiguratorenansätze unter immer höheren Wettbewerbsdruck. Entsprechend effizienter werden die Konfiguratoren und es kann gespannt auf noch folgende Konfiguratoren geblickt werden.

7 Zusammenfassung

Da das manuelle Konfigurieren von Algorithmen eine aufwendige Arbeit darstellt, setzten sich zahlreiche Konzepte für automatische Verfahren durch. Von Racing-Konfiguratoren über Konfiguratoren basierend auf genetischen Algorithmen, bis hin zu Konfiguratoren basiernd auf iterativer lokaler Suche, steht eine Vielzahl von Ansätzen zur Verfügung, die alle ihre Vor- und Nachteile haben. Die vielversprechendsten Konfiguratoren stellten dabei BasicILS und FocusedILS dar. Durch die verwendete Such-Prozedur konnten ermittelte Ergebnisse stetig verbessert werden und schnell zufriedenstellende Parameterkonfigurationen gefunden werden. Der Nachteil lag allerdings in der mangelnden Möglichkeit Berechnungen auf mehrere Prozessoren auszulagern. In dieser Arbeit wurden mehrere Mechanismen entworfen, um genau diese Problematik zu beseitigen und möglichst effiziente parallele Konfiguratorenvarianten zu implementieren. Durch eine Verteilung der Läufe auf freie Prozessoren, eine Reduzierung der gespeicherten Daten und einem dynamischen Job Buffer, wurde genau dies möglich gemacht. In vier Experimenten konnten zwei Sat-Solver mit größtenteils positiven Ergebnissen konfiguriert werden. Je nach Probleminstanzen konnte eine Auslastung von 7% bis zu 96% erzielt werden. Auf einfacheren Probleminstanzen setzte sich dabei Parallel BasicILS gegenüber den beiden Parallel FocusedILS Varianten durch, auf schwierigen Instanzen war das Gegenteil der Fall.

Literaturverzeichnis

[1] *bwGRiD (http://www.bw-grid.de): Mitglied der deutschen D-Grid Initiative*

[2] ANSÓTEGUI, Carlos ; SELLMANN, Meinolf ; TIERNEY, Kevin: A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms. In: *In Proc. of CP-09*, 2009, S. 142–157

[3] BALINT, Adrian ; DIEPOLD, Daniel ; GALL, Daniel ; GERBER, Simon ; KAPLER, Gregor ; RETZ, Robert: *EDACC - an advanced platform for the experiment design, administration and analysis of empirical algorithms*. 2011

[4] BALINT, Adrian ; FRÖHLICH, Andreas: Improving Stochastic Local Search for SAT with a New Probability Distribution SAT 2010, Springer, 2010

[5] BIRATTARI, Mauro ; STUTZLE, Thomas ; PAQUETE, Luis ; VARRENTRAPP, Klaus: A racing algorithm for configuring metaheuristics. In: *Proceedings of the Genetic and Evolutionary Computation Conference*, Morgan Kaufmann, 2002, S. 11–18

[6] HUTTER, Frank: *Automated Configuration of Algorithms for Solving Hard Computational Problems*. Vancouver, Kanada, Diss., 2009

[7] HUTTER, Frank ; HOOS, Holger H. ; LEYTON-BROWN, Kevin: *Tradeoffs in the Empirical Evaluation of Competing Algorithm Designs*. 2010

[8] HUTTER, Frank ; HOOS, Holger H. ; LEYTON-BROWN, Kevin: *Sequential Model-Based Optimization for General Algorithm Configuration (extended version)*. 2011

[9] TOMPKINS, Dave A. D. ; BALINT, Adrian ; HOOS, Holger H.: Captain Jack: New Variable Selection Heuristics in Local Search for SAT SAT 2011, Springer, 2011

Name: Juri Schulte Matrikelnummer: 677063

Erklärung

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

Ulm, den .....

Juri Schulte