FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

Generierung Domänenspezifischen Wissens für ExaStencils

B. Sc. Lorenz Haspel

Masterarbeit

Generierung Domänenspezifischen Wissens für ExaStencils

B. Sc. Lorenz Haspel

Masterarbeit

Aufgabensteller:	PD Dr.-Ing. habil. Harald Köstler
Betreuer:	M. Sc. Sebastian Kuckuk
Bearbeitungszeitraum:	23.6.2016 – 23.12.2016

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Masterarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 22. Dezember 2016 .....

Inhaltsverzeichnis

1 Abstract **5**

2 Einleitung **5**
2.1 Motivation 5
2.2 Eingliederung dieser Arbeit in das ExaStencils Projekt 5
2.3 Verwandte Arbeiten 6
2.4 Überblick 6

3 Der genetische Ansatz **7**
3.1 Exkurs: Genetische Algorithmen - Optimierungsprobleme und Evolution 7
3.2 Ablauf genetischer Algorithmen 8
3.3 Genetischer Algorithmus zum Optimieren von ExaSlang-Code 12
3.4 Auswertung des genetischen Algorithmus 18

4 Extraktion von domänenspezifischem Wissen **20**
4.1 Ausgangslage 20
4.2 Vergleich aller Konfigurationen 20
4.3 Finden von relevanten Parametern 22
4.4 Clustering 24
4.5 Mehrere Clustering Heuristiken 25
4.6 Ergebnis 25

5 Generierung von domänenspezifischem Wissen **27**
5.1 Ergebnisse der Analyse 27
5.2 Vergleich zweier Probleme 28
5.3 Vergleich mehrerer Probleme 29
5.4 Fazit 31

6 Zusammenfassung **32**

1 Abstract

Diese Arbeit betrachtet den Compilevorgang des ExaStencils Projekts als Optimierungsproblem. Dessen Ziel ist es, domänenspezifisches Wissen zu finden, das zu möglichst guter Performanz bei den erzeugten Programmen führt. Des Weiteren sollen Ähnlichkeiten zwischen Anwendungen automatisiert gefunden und erkannt werden, und domänenspezifisches Wissen erzeugt werden, das zukünftige Compilevorgänge optimiert.

2 Einleitung

2.1 Motivation

Domänenspezifische Sprachen (DSL: domain-specific language) sind ein Mittel, mit dem Experten einer Domäne (zum Beispiel Mathematiker) ein Problem beschreiben können, wobei sie auf Konzepte und Modelle zurückgreifen können, mit welchen sie vertraut sind. Die DSL kann anschließend von einem spezialisierten Compiler in eine Zielsprache, wie zum Beispiel C-Code umgewandelt werden. Hierdurch ist es für den Experten möglich, ohne Programmierkentnisse ein Programm zu erstellen, welches das von ihm spezifizierte Problem löst. Dieses Projekt verwendet als DSL den *ExaSlang* - eine Sprache zur Spezifikation von Programmen, die Gleichungssysteme, die aus der Diskretisierung partieller Differentialgleichungen stammen, numerisch mittels des Mehrgitterverfahren lösen. Der *ExaStencils* Compiler übersetzt den *ExaSlang* in C++ Code, wobei verschiedene Optimierungs- und Parallelisierungstechniken verwendet werden. Eines der Ziele des *ExaStencils* Projektes ist es, automatisiert optimale Performanz auf High Performance Computing (HPC) Clustern zu erhalten. Der *ExaStencils* Compiler kann hierbei auf domänenspezifisches Wissen zurückgreifen, welches ursprünglich von Domänenexperten zur Verfügung gestellt wird. Diese Arbeit nutzt aus, dass dieses Wissen automatisiert verändert und verbessert werden kann, so dass nach mehreren Compilevorgängen eine bessere Lösung mit höherer Performanz entsteht.

2.2 Eingliederung dieser Arbeit in das ExaStencils Projekt

ExaStencils ist ein Projekt, das sich mit Stencilcodes beschäftigt, also mit "rechenintensiven Algorithmen, in denen wiederholt Datenpunkte in einem Gitter aus einer Kombination von den Werten benachbarter Punkte bestimmt werden. Das verwendete Muster der benachbarten Punkte heißt Stempel, oder engl. Stencil. Stencilcodes finden zur Lösung von diskretisierten partiellen Differentialgleichungen und den daraus entstehenden linearen Systemen verbreitet Einsatz."(aus exastencils.org [3])

Das Ziel von ExaStencils ist es, „die bequeme, anwendungsnahe Formulierung von Problemlösungen zu ermöglichen, und deren Implementierung **möglichst automatisiert unter Verwendung von domänenspezifischem Wissen** in verschiedenen Schritten optimieren zu können, so dass portable Exascale-Leistung resultiert.“ [3]

ExaStencils geht in fünf Schritten vor:

1. Anpassung des mathematischen Problems
2. Erstellung eines Programms in einer domänenspezifischen Sprache
3. Domänenspezifische Optimierung anhand der Merkmale des Stencilcodes
4. Schleifenoptimierung im Polyedermodell
5. Plattformspezifische Nachschärfungen

Diese Arbeit stellt einen Teilbereich des dritten Schrittes des ExaStencil-Projektes dar. In diesem Schritt werden die "Ähnlichkeiten zwischen Stencilcodes für verschiedene Anwendungen" ausgenutzt, so dass die Implementierung des Stencilcodes dann automatisch erfolgen kann. Dies geschieht "als Komposition seiner Merkmale und unter Einsatz von **domänenspezifischen Optimierungen**, die auf die spezielle Natur von Stencilcodes sowie auf die Anforderungen der konkreten Anwendung abgestimmt sind." [3]

2.3 Verwandte Arbeiten

Kronawitter und Lengauer [5] beschreiben Codetransformationen und Optimierungen, die von ExaStencils Code Generator durchgeführt werden. Die vorliegende Arbeit erzeugt Konfigurationen für diesen Code Generator, welche die erwähnten Optimierungen auswählt, und versucht dadurch die Performanz zu optimieren.

Siegmund und andere [9] berechnen Performanz-Einfluss-Modelle, die beschreiben, wie Konfigurationsoptionen die Performanz eines Systems beeinflussen. Jene Arbeit zielt allgemein auf konfigurierbare Softwaresysteme ab, deren Qualität in irgendeiner Weise messbar sind (zum Beispiel Laufzeit). Die dort erwähnten Experimente beinhalten unter anderem errechnete Modelle für Compiler und Mehrgitterlöser; das heißt, in dieser Arbeit sind ähnliche Ergebnisse zu erwarten.

2.4 Überblick

„Die auf dem Papier entworfene Lösung [eines mathematischen Problems wird] in eine abstrakte aber ausführbare domänenspezifische Sprache überführt, in der die wesentlichen Merkmale der Lösung einfach benannt werden können." [3] Aus dieser Sprache kann der ExaStencils Compiler C++ Code generieren. Die Güte (bezüglich der Laufzeit, oder Ähnliches) des Codes hängt jedoch von weiteren Variablen ab, die dem Compiler übergeben werden. Bei diesen Variablen kann es sich zum Beispiel um den Grad der Parallelisierung oder um die Art der Optimierungsschritte handeln. Hierbei ist es für den Menschen unter Umständen schwierig, eine gute Variablenbelegung zu finden, die auf der Zielarchitektur zu guter Leistung führt, da die verschiedenen Variablen miteinander in Beziehung stehen beziehungsweise sich nicht direkt mit dem Problem in Beziehung setzen lassen. Ziel dieser Arbeit ist es, automatisiert gute Belegungen für diese Variablen zu finden, und hierbei die Güte des erzeugten Codes zu optimieren. Dies erfolgt grundlegend in drei Schritten:

1. Es wird ein **genetischer Algorithmus** (GA) ausgeführt, der für eine gegebene Problemstellung eine Variablenbelegung findet, die ein (lokales) Optimum bezüglich der Güte darstellt.
2. Die gefundenen Optima mehrerer solcher Optimierungsläufe werden verglichen, um für den Compiler **nutzbares Wissen** über das Problem **zu generieren**. Hiermit können weitere Optimierungsdurchgänge vereinfacht werden.
3. 3. Das **generierte Wissen** über mehrere ähnliche Probleme wird **verglichen**, um **zusätzliches Wissen** über weitere ähnliche Probleme **zu generieren**. Somit kann die Anzahl an nötigen Optimierungen reduziert werden.

Im folgenden Kapitel werden zunächst die Grundlagen genetischer Algorithmen aufgezeigt, um anschließend auf den zur Optimierung verwendeten genetischen Algorithmus einzugehen. In den darauffolgenden Kapiteln wird erläutert, wie daraus domänenspezifisches Wissen für den ExaStencils Compiler generiert werden kann.

3 Der genetische Ansatz

3.1 Exkurs: Genetische Algorithmen - Optimierungsprobleme und Evolution

In der Mathematik stoßen wir immer wieder auf **Optimierungsprobleme**, die sich nicht berechnen oder nicht exakt in vertretbarer Zeit berechnen lassen. Es gibt verschiedene heuristische Verfahren, mit deren Hilfe sich solche Optimierungsprobleme lösen lassen. Unter genetischen Algorithmen versteht man eine Metaheuristik, also ein heuristisches Verfahren, das auf beliebige Optimierungsprobleme anwendbar ist, ohne Hintergrundwissen über die Problemstellung zu haben. (nach Haspel [4])

Doch zunächst ein naiver Ansatz zur Lösung eines Optimierungsproblems:

Ein Optimierungsproblem lässt sich immer durch eine (mehrdimensionale) Funktion ausdrücken, deren Maximum bzw. Minimum man sucht. Man könnte also einen beliebigen Punkt im Definitionsbereich der Funktion auswählen und mittels Gradientenabstiegsverfahren den Punkt immer ein Stück in Richtung der größten Steigung verschieben, bis man bei einem Maximum angelangt ist. Das Problem bei diesem Vorgehen ist, dass es meist in einem lokalen Optimum konvergiert. Siehe Abbildung 1: Das Gradientenabstiegsverfahren von einem zufälligen Punkt aus führt nur selten zum globalen Maximum. Leicht können lokale Maxima gefunden werden. In Abbildung 1 ist eine Funktion zu sehen, die mehrere lokale Minima and Maxima aufweist und bei der das globale Maximum deutlich größer ist als die umgebenden lokalen Maxima.

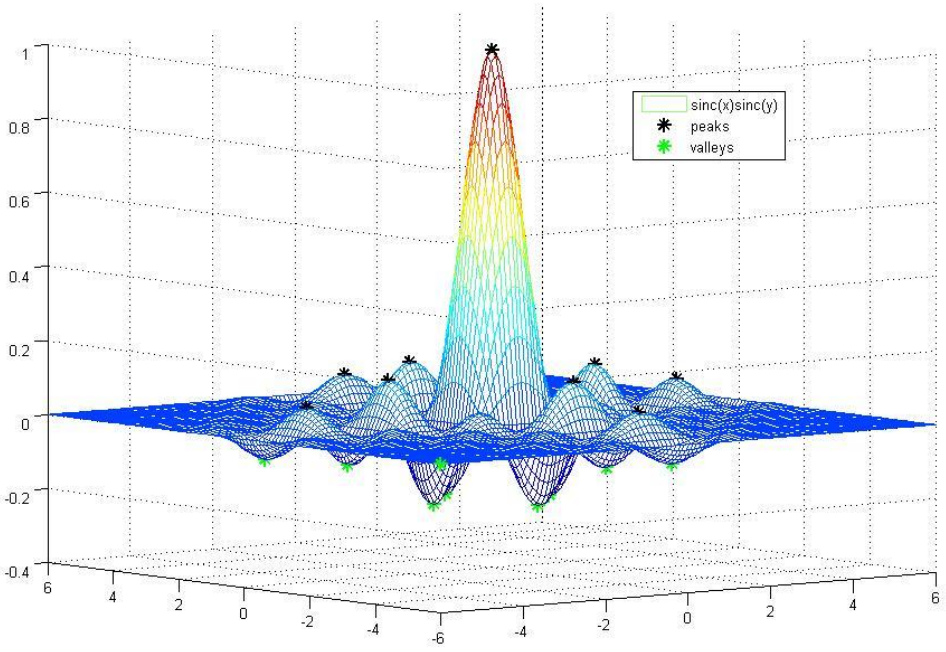

Abbildung 1: *[1] zwei-Dimensionale Funktion sinc(c)sinc(y). Markiert sind lokale Minima (grün) und Maxima (schwarz).*

Eine bessere Lösung liefert die **natürliche Evolution**:

Populationen von Lebewesen überleben in freier Wildbahn nur dann, wenn sie sich veränderlichen Umweltbedingungen möglichst optimal anpassen. Es gibt eine Vielzahl an unterschiedlichen *Individuen*, von denen die am besten angepassten überleben und sich stärker vermehren. Die Fähigkeit der Angepasstesten zu überleben wird an die Nachfolgegeneration weitervererbt, so dass sich die

gesamte *Population* über viele *Generationen* hinweg an die gegebenen Umweltbedingungen anpasst. Durch geringfügige *Mutationen* beim Weitergeben der Gene an die Nachfolgegeneration können sich neue Individuen entwickeln, die möglicherweise besser an die augenblicklichen Umweltbedingungen angepasst sind, als die Individuen der Elterngeneration.

3.2 Ablauf genetischer Algorithmen

Genetische Algorithmen bedienen sich nun der Methoden der natürlichen Evolution, um Optimierungsprobleme zu lösen. Es werden zufällig viele Punkte (*Individuen*) im Suchraum des gegebenen Problems ausgewählt, welche die *Startpopulation* bilden. Aus dieser werden durch Paarung und Vererbung (*Rekombination*) der Informationen (*Gene*) von Individuen und zufälliger Veränderung (*Mutation*) neue Kindindividuen gebildet: Für die Rekombination werden Individuen aus der Population ausgewählt, welche Kinder erzeugen. Ein Kind erbt Informationen der Eltern, das heißt, es wird aus den in den Eltern enthaltenen Informationen aufgebaut. Bei der Mutation wird anschließend ein zufälliger Teil des Kindes verändert. Schließlich werden alle Individuen der Population nach ihrem Gütewert bezüglich der Optimierung (*Fitness*) bewertet (*Evaluation*). Die besten Individuen werden für eine neue Population ausgewählt (*Selektion*). Dies wird mit der neu entstandenen Population wiederholt, solange bis ein *Abbruchkriterium* (maximale Generationenzahl oder erreichtes Optimum) erfüllt wird.

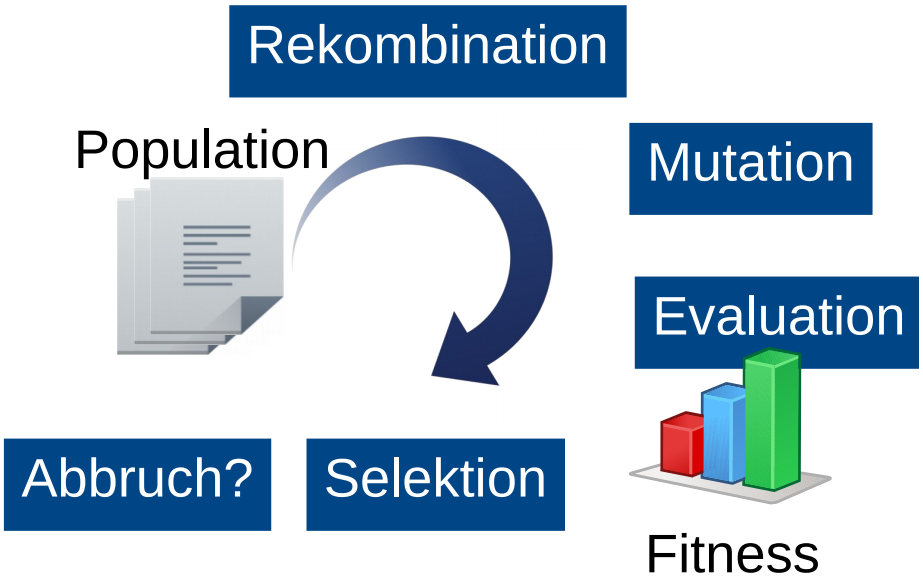

Abbildung 2: *Allgemeiner Ablauf genetischer Algorithmen. Operationen sind blau hinterlegt.*

Im Folgenden soll auf die einzelnen Operatoren und deren Zusammenhänge eingegangen werden.

Konvergenz und Erkundung

Das Ziel des Algorithmus soll es sein, das globale Optimum zu finden. Zu Beginn sind die Individuen zufällig über den Suchraum verteilt. Das Ziel ist es also, sowohl für Konvergenz hin zu Optima, als auch für die Erkundung des Suchraums zu sorgen. Hierfür stehen Selektions-, Rekombinations- und Mutationsoperatoren zur Verfügung: Durch die Rekombination wird eine Verständigung zwischen den Individuen hergestellt, mit dem Zweck ein Kind zu erzeugen, das die Vorteile der Eltern in sich vereint. Somit werden vorhandene Informationen dazu genutzt, weitere Punkte im Suchraum zu erzeugen. Die Mutation sorgt für eine zufällige Veränderung, und somit für die Erkundung des Suchraums. Die Selektion sorgt schließlich für eine Konvergenz der Gesamtpopulation, da schlechte Individuen nicht in die nachfolgende Generation übernommen werden.

Durch die Vielzahl an auszuwählenden Parametern (Populationsgröße, Bewertungsfunktion, Selektions-, Rekombinations-, Mutationsoperatoren) ist eine hohe Anpassbarkeit des Algorithmus auf das gegebene Problem möglich, jedoch durch die Abhängigkeit zwischen den einzelnen Parametern und fehlendes Wissens über die Auswirkungen der Parameterwahl schwierig.

Beispiele für mit genetischen Algorithmen gut zu lösende Optimierungsprobleme sind das Traveling Salesperson Problem, die Positionierung von Überwachungskameras in einem Kaufhaus, die Bestimmung von Rohrdurchmessern in einem Abwassersystem, etc.

Aufbau der Individuen

Bei der Beschreibung eines Individuums wird zwischen *Genotyp* und *Phänotyp* unterschieden. Der Genotyp beschreibt in der Evolutionslehre die Gene eines Individuums, bei den genetischen Algorithmen versteht man darunter die Parameter des Individuums, welche im Lauf des Algorithmus beeinflusst werden. Der Phänotyp beschreibt das beobachtbare Verhalten des Individuums, also hier den Gütewert, den die Bewertungsfunktion liefert, wenn die Parameter des Genotyps eingesetzt werden.

Beispiel für den Aufbau eines Individuums und Anwendung einer Bewertungsfunktion. Die erste Zeite gibt den Genotyp von A an, die zweite den Phänotyp:

$$A = (42, 13, 37)$$
$$fitness(A) = 2$$

Rekombination

Der Rekombinationsoperator erschafft aus zwei (oder mehr) Elternindividuen ein oder mehrere Kinder. Hierbei bestimmt der Aufbau der Eltern auf unterschiedliche Art und Weise den Aufbau des Kindes. *Kombinierende Operatoren* setzen den Genotyp der Eltern neu zusammen, um den Genotyp des Kindes zu bestimmen. Hierdurch können nur Punkte im Suchraum erzeugt werden, in deren Gebiet sich bereits Individuen befinden. Der Erfolg der Rekombination hängt hier also stark von der Diversität der Population ab.

Eine andere Form der Rekombination sind *expandierende Operatoren*, welche den Gütewert der Eltern verwenden, um eine möglichst günstige Suchrichtung zu bestimmen (*Extrapolation*). Hierbei muss natürlich Wissen über die Problemstellung mit in den Operator eingehen, um bessere Erfolge als bei kombinierenden Operatoren zu erzielen. Insgesamt sind expandierende Operatoren jedoch eher unüblich.

Als drittes existieren noch *kontrahierende Operatoren*, die sich auf Populationen mit wenig Diversität konzentrieren und somit für eine schnelle Konvergenz sorgen. Um die Konvergenz zu lokalen Optima zu verhindern, muss in einem solchen Fall der Mutationsoperator für möglichst große Diversität sorgen.

Beispiel für eine Crossover-Rekombination der Individuen A und B. Die ersten beiden Parameter stammen von B, der dritte von A:

$$A = (42, 13, 37)$$
$$B = (28, 16, 55)$$
$$A \times B = (28, 16, 37)$$

Mutation

Die Mutation verändert den Genotyp eines Individuums, woraus sich auch eine Veränderung des Phänotyps ergibt. Ist die Veränderung am Phänotyp klein, sorgt die Mutation für Feinabstimmung, das heißt ein Individuum kann sich einem Optimum in der Umgebung nähern. Ist die Veränderung am Phänotyp groß, so dass das neue Individuum große Sprünge im Suchraum macht, spricht man von *Makromutation*. Makromutationen können die Diversität bei bereits stark konvergierten Populationen erhöhen. Ist die Mutationsstärke groß gewählt, erhöht dies die Diversität und dient der Erkundung des Suchraums, jedoch muss die Feinabstimmung durch den Rekombinationsoperator erfolgen. Die Existenz lokaler Optima hängt von der Stärke der Mutation ab. An jeder Stelle im Suchraum, von der aus ein Individuum mit nur einem Mutationsschritt keinen besseren Gütewert erreichen kann, befindet sich ein lokales Optimum. Die Anzahl lokaler Optima hängt also nicht nur von der Beschaffenheit des Problems, sondern auch von der Mutationsstärke ab.

Wie viel Einfluss eine Veränderung des Genotyps am Phänotyp hat, hängt von der Codierung der Individuen und dem Aufbau des Problems ab.

Beispiel für eine Mutation von +5 im zweiten Parameter des Individuums A:

$A = (42, 13, 37)$
$A' = (42, 18, 37)$

Bewertung
Anders als bei der natürlichen Evolution verfügen die meisten Optimierungsprobleme über ein eindeutiges Bewertungskriterium, welches die Güte/Fitness eines Individuums angibt. Eine eindeutige Bewertung von Individuen ist essentiell, da später die besten Individuen ausgewählt werden sollen. Im Falle einer Maximumssuche bei einer Funktion entspricht die Güte dem Funktionswert, wenn die Werte des Individuums eingesetzt werden. Bei einigen Problemen, wie zum Beispiel Spielstrategien, müssen die Individuen durch Simulation bewertet oder per Turnierverfahren untereinander verglichen werden, was unter Umständen sehr zeitaufwändig sein kann. Da die Bewertung einmal für jedes Individuum erfolgen muss, sollte in einem solchen Fall die Menge an neu erzeugten Individuen pro Generation eher klein gewählt werden, wenn überhaupt mit genetischen Algorithmen vorgegangen werden soll.

Selektion
Die Selektion sorgt als kontrahierender Operator dafür, die Suche nach dem Optimum auf relevante Bereiche des Suchraums einzugrenzen. Es können unterschiedliche Selektionsmechanismen gewählt werden, ein Hauptkriterium bildet die Frage nach dem Zufallseinfluss. Werden nur die besten Individuen ausgewählt, spricht man von *deterministischer Selektion*, wird jedem Individuum auf Basis seines Gütewertes eine Wahrscheinlichkeit, ausgewählt zu werden, zugeordnet, spricht man von *probabilistischer Selektion*. Bei probabilistischer Selektion können also auch schlechtere Individuen ausgewählt werden, was eine größere Vielfalt (*Diversität*) der Population gewährleisten kann. Ein weiteres Kriterium ist die *Duplikatfreiheit*. Durch das Zulassen von Duplikaten bei probabilistischer Selektion können bessere Individuen mehrfach gewählt werden und erhalten somit eine höhere Wahrscheinlichkeit sich fortzupflanzen und die nächste Generation zu überleben. Durch das Verbieten von Duplikaten bleibt eine gewisse Diversität erhalten, da die Population nicht von einigen wenigen Individuen übernommen werden kann. Deterministische Selektionsverfahren sind in der Regel immer duplikatfrei.

Im Algorithmus erfolgen zwei Selektionen: Die Elternselektion, die bestimmt, welche Individuen sich fortpflanzen, und die Umweltselektion, die bestimmt welche Individuen in die Nachfolgepopulation aufgenommen werden. Bei der Umweltselektion stellt sich die Frage, ob nur die erzeugten Kinder oder auch deren Eltern für die Selektion in Frage kommen. Werden die Eltern mit einbezogen, spricht man von *überlappender Selektion*. Hierdurch wird ein Informationsverlust verhindert, da die besten Individuen der Elterngeneration auf diese Weise nicht verloren gehen. Überlappende Selektion kann auf zwei Arten realisiert werden: Man kann Eltern und Kinder gleichwertig aufgrund ihrer Güte bewerten (häufig bei deterministischer Umweltselektion) oder man kann einen Überlappungsgrad angeben, der besagt, wie viele Individuen aus der Elterngeneration und wie viele aus der Kindgeneration übernommen werden sollen.

Der *Selektionsdruck* gibt an, wie viele der existierenden Individuen übernommen werden. Ist der Selektionsdruck niedrig, werden auch schlechtere Individuen übernommen, was der Erkundung des Suchraums dient. Ist der Selektionsdruck hoch, werden wenige Individuen übernommen, was der Feinabstimmung, also der Suche nach Individuen dient, die Optima nahe stehen. In jedem Fall verringert die Selektion die Diversität der Population.

Beispiel für verschiedene Umweltselektionen auf Population P: Eine mögliche Nachfolgegeneration $P_{25\%}$ besteht aus den besten 25% der Individuen der alten Population P. Eine alternative Nachfolgegeneration mit niedrigerem Selektionsdruck würde zum Beispiel $P_{75\%}$ darstellen.

$$
\begin{aligned}
P &= \{A, B, C, D\} \\
fitness(A) &= 2 \\
fitness(B) &= 7 \\
fitness(C) &= 4 \\
fitness(D) &= 1 \\
P_{25\%} &= \{B\} \\
P_{75\%} &= \{A, B, C\}
\end{aligned}
$$

3.3 Genetischer Algorithmus zum Optimieren von ExaSlang-Code

In diesem Abschitt wird ein konkreter genetischer Algorithmus vorgestellt, der implementiert wurde, um Parameterbelegungen für den ExaStencils Compiler zu optimieren.

Eingabe des ExaStencil Compilers

Der ExaStencils Compiler generiert C++ Code aus einer abstrakten Spezifikation, die das Problem beschreibt. Diese Spezifikation liegt als ExaSlang Code vor, einer domänenspezifischen Sprache, die zur Specifikation von numerischen Mehrgitterlösern verwendet wird. Eine Programmspezifikation in ExaSlang wird durch die Zusammenarbeit von Ingenieuren, Naturwissenschaftlern, Mathematikern und Informatikern erstellt und durchläuft deshalb im Entstehungsprozess mehrere Abstraktionsschichten (nach Schmitt et al. [8]):

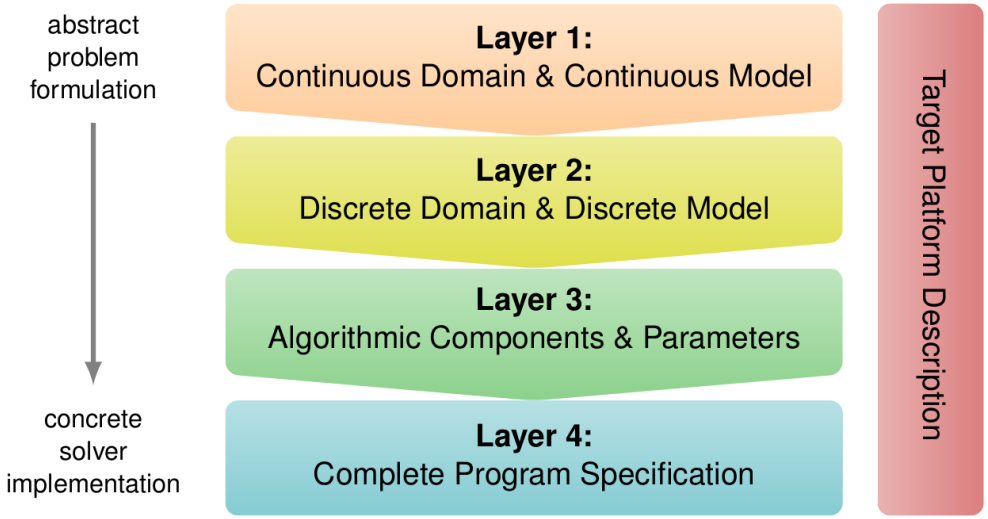

Abbildung 3: *Die verschiedenen Schichten der ExaSlang DSL*

In der abstraktesten Schicht (Layer 1) wird das Problem als zu minimierendes Energiefunktional oder als zu lösende partielle Differentialgleichung definiert. Das Problem ist auf dieser Schicht als kontinuierliches Problem beschrieben. Diese Schicht wendet sich an Naturwissenschaftler und Ingenieure, die über keine Programmiererfahrung verfügen.

Die zweite Schicht ist weniger abstrakt, so dass das Problem als diskretisiert spezifiziert werden kann. Diese Schicht ist sowohl für erfahrenere Naturwissenschaftler und Ingenieure, als auch für Mathematiker gedacht.

Auf der dritten Schicht werden algorithmische Komponenten, Einstellungen und Parameterwerte modelliert. Dies ist die erste Schicht, in der das Mehrgitterverfahren, auf welchem das ExaStencils Projekt aufgebaut ist, zu erkennen ist. Diese Schicht wendet sich hauptsächlich an Mathematiker und Informatiker.

Die konkreteste Schicht ist Schicht 4, auf der Parallelisierungstechniken und Datenstrukturen spezifiziert werden. Diese Schicht wird ausschließlich von Informatikern verwendet.

Eine fertige Programmspezifikation liegt am Ende in Form von Parameterzuweisungen vor, und kann wie folgt aussehen:

Auszug aus der Spezifikation 2D_ConstCoeff.knowledge. Der ExaStencils Compiler erzeugt daraus ein Programm, das eine finite Differenzen Diskretisierung der Poissiongleichung mit Dirichlet Randbedingungen löst:

```
comm_strategyFragment              = 6
comm_useFragmentLoopsForEachOp     = true
data_alignFieldPointers            = false
dimensionality                     = 2
domain_numBlocks                   = 4
...
```

Den Parametern des Compilers (linke Spalte) werden auf diese Weise Werte als Argumente (rechte Spalte) übergeben. Hierbei definieren einige Parameter das gegebene Problem und sind durch die Problemspezifikation festgelegt (wie zum Beispiel `dimensionality`), und dürfen durch Optimierungen nicht verändert werden.

Andere Parameter stellen Compileroptionen dar (zum Beispiel `comm_strategyFragment`) und können verändert werden, um effizienteren Code zu erzeugen. Hierunter fallen Techniken, die das Zielprogramm verwendet (Wahl des Glättungsverfahrens), Art und Grad der Parallelisierung (openMP, MPI, Anzahl der jeweils verwendeten Threads) und Optimierungsschritte auf Ebene des C++ Compilers (loop-unrolling).

Der Wertebereich für eine gültige Wahl der Parameteroptionen ist sehr groß, so dass eine vollständige Durchsuchung des Suchraums nach dem globalen Optimum nicht in akzeptabler Zeit durchgeführt werden kann. Um ein Optimum zu finden muss also eine Heuristik verwendet werden. In dieser Arbeit wurde ein genetischer Algorithmus zur Optimumssuche verwendet.

Der Compilevorgang

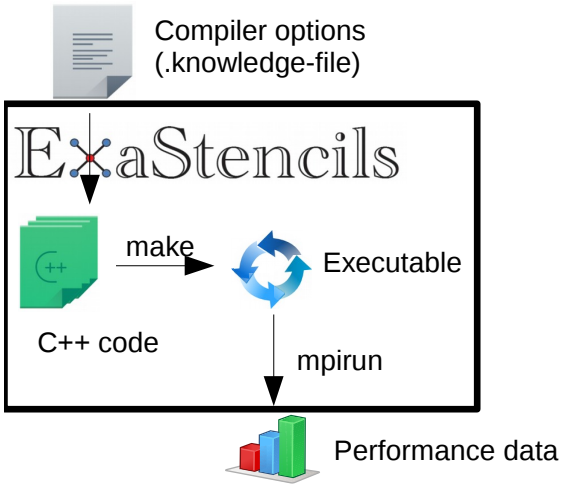

Abbildung 4: *Der Compilierungsvorgang ausgehend von der Problemspezifikation in domänenspezifischen Sprache (Compiler options) über C++ Code bis zur Ausführbaren Datei (Executable), die Performanzdaten ausgibt.*

Typischerweise besteht ein Compilevorgang aus dem Generieren des C++ Codes aus der Domänenspezifischen Sprache durch den ExaStencils Compiler und anschließend dem Übersetzen des C++ Codes in Maschinensprache durch einen C++ Compiler. Anschließend kann das erzeugte Programm ausgeführt werden, wobei das Programm seine Laufzeit zurück gibt (vgl. Abbildung 4).

Jetzt kann ein genetischer Algorithmus verwendet werden, dessen Ziel es ist effiziente Compileroptionen für ein gegebenes Problem zu finden. Hierbei kann der Generier-, Compile-, und Ausführungsvorgang als Black-Box-Vorgang betrachtet werden, der als Eingabe Compileroptionen bekommt und die daraus resultierende Laufzeit ausgibt.

Genetischer Algorithmus - Operationen und Parameter

Ein **Individuum** des genetischen Algorithmus stellt eine Konfiguration von Parameteroptionen dar, also etwas, was dem ExaStencil Compiler übergeben werden kann, woraufhin ein Programm erzeugt wird. Die Fitness des Individuums entspricht hierbei dem Inversen der Laufzeit (je schneller das fertige Programm zu seinem Ergebnis kommt, desto besser ist die Konfiguration, aus der es erzeugt wurde) und ist mit der Güte der Konfiguration gleichbedeutend.

Auf Codeebene besteht ein Individuum aus zwei Arten von Werten: normalen Compileroptionen, die in die Konfiguration übernommen werden können, und *virtuelle* Optionen, die eine Abstraktion von einem oder mehreren Compileroptionen darstellen. Diese *virtuellen* Optionen können verwendet werden, um Einschränkungen und Zusammenhänge zwischen mehreren Compileroptionen zu realisieren.

Beispiel: Es gilt folgende Einschränkung: Wenn openMP mit mehr als einem Thread verwendet wird, muss genau eine der beiden Compileroptionen `omp_parallelizeLoopOverFragments` und `omp_parallelizeLoopOverDimensions` den Wert `true` annehmen.

Da der genetische Algorithmus beide Parameter unabhängig voneinander verändern würde, könnten ungültige Konfigurationen entstehen. Deshalb werden diese beiden Compileroptionen durch die *virtuelle* Option `ompParallelizeLoops` abstrahiert. Diese gibt an, welche der beiden Compileroptionen auf `true` gesetzt wird. Die *virtuelle* Option ist somit Teil des Genotyps des Individuums und wird vom genetischen Algorithmus manipuliert, während die beiden ursprünglichen Optionen für den genetischen Algorithmus nicht sichtbar sind.

Bei der Erzeugung der Konfiguration für den ExaStencil Compiler treten dann die für den genetischen Algorithmus nicht sichtbaren Compileroptionen an Stelle der *virtuellen* Optionen.

Implementierungsdetails zu Parametern:

```
Param = {
    'minLevel':                         Parameter(range(0,12)),
    'comm_strategyFragment':            Parameter([6,26]),
    'comm_useFragmentLoopsForEachOp':   Parameter([True,False]),
    'ompParallelizeLoops':              Parameter([ True, False ],
                                                  virtual=True),
    ...
    }
```

Alle zu optimierenden Parameter werden in einem Python `dict` angelegt. Jeder Paramter wird durch ein `Parameter` Objekt modelliert, dem beim Anlegen ein Wertebereich übergeben werden muss. Der Wertebereich kann als Liste beliebiger Datentypen oder als `range` Objekt vorliegen (das einen Integer-Wertebereich definiert). [1] Soll der Parameter eine *virtuelle* Option darstellen, wird `virtual=True` übergeben. Das sorgt dafür, dass der Parameter bei der Evaluation nicht in die Konfigurationsdatei geschrieben wird.

Implementierungsdetails zu Individuen (vereinfacht):

```
class Individuum:
    def __init__(self, pset=Param):
        for p in pset:
            self.values[p] = random.choice(Param[p]._rng)
```

Die Individuen der **Startpopulation** werden mit zufälligen Werten für alle Optionen initialisiert. Die zufälligen Werte stammen in den meisten Fällen gleichverteilt aus dem Wertebereich des entsprechenden Parameters. Bei einer Gleichverteilung würden bei einem sehr großen Wertebereich kleine (ein- oder zweistellige) Zahlen sehr selten auftreten. In einigen Fällen ist dieses

[1] Bei Python3 `range` Objekten ist das angegebene Ende nicht mehr im Wertebereich enthalten. Damit jedoch der Code zum Anlegen der Parameter leichter aus dem Scala-Code des ExaStencils Compilers erzeugt werden kann, wird der Wertebereich der `Parameter` Objekte beim Übergeben von `range` Objekten um einen Wert erweitert. Somit gilt beim Anlegen eines neuen Parameter mit einem `range` Objekt als Wertebereich, dass das Ende, wie beim Scala-Code, mit im Wertebereich enthalten ist.

Verhalten jedoch unerwünscht. Deshalb wird bei Parametern mit sehr großem oder nach oben hin unbeschränkten Wertebereich der Startwert paretoverteilt ermittelt.

Paretoverteilung [7] [6] (Beispiel in Abbildung 5):
Diese Verteilung beschreibt ein Verhalten, bei dem kleinere Werte häufiger auftreten, sehr große Werte hingegen sehr selten vorkommen. Die Verteilung ist nach oben hin unbeschränkt, so dass sie für Variablen mit unbeschränkten Wertebereich verwendet werden kann.

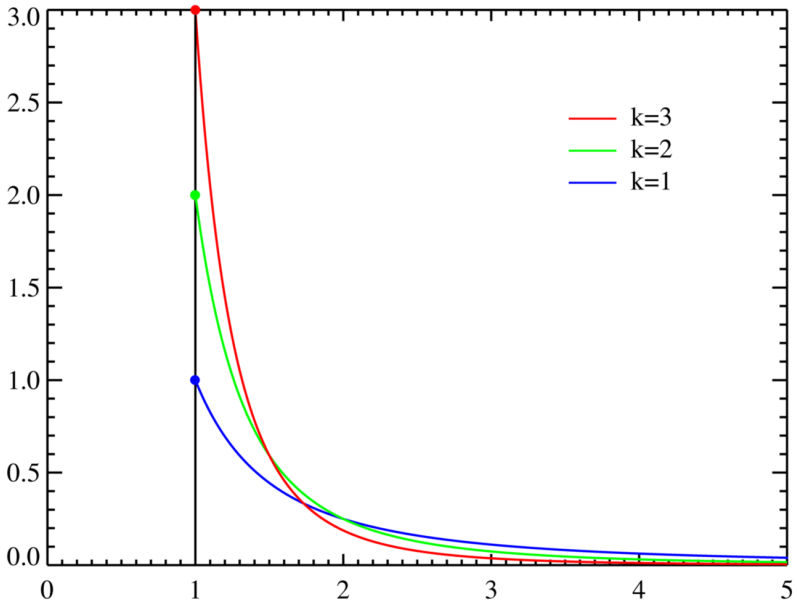

Abbildung 5: *[2] Pareto-Wahrscheinlichkeitsdichte f(x) mit (xmin=1)*

Implementierungsbeispiel:
Python-code, der für den implementierten genetischen Algorithmus das Intervall für die Compileroption ***poly_tileSize_x*** *definiert. Als Verteilung wird eine Paretoverteilung verwendet:*

```
'poly_tileSize_x': Parameter(range(112,1000000000 ,32),
                    distr = lambda: random.paretovariate(.2) -1 )
```

Der Wertebereich von `poly_tileSize_x` reicht von `112` bis `1000000000`, wobei in `32`er Schritten inkrementiert wird. Zur Bestimmung eines Startwertes wird eine $Par(0.2,\ 1)$ paretoverteilte Zufallszahl ermittelt, die als Index auf dem Wertebereich fungiert. (wird beispielsweise die Zahl 4 ausgewürfelt, wird das vierte Element des Wertebereichs zurückgegeben. In diesem Fall die 208.)[2]

Bei der **Rekombinationsselektion** werden zwei Elternteile zufällig aus der Population ausgewählt, wobei eine Gewichtung nach der Fitness vorgenommen wird. Es gilt:

$$P(\text{"}X\text{wird für diesen Crossover-Schritt ausgewählt"}) = \frac{fitness(X)}{\sum_{i \in Population} fitness(i)}$$

Dadurch haben Individuen mit höherer Fitness eine höhere Chance sich fortzupflanzen. Dies sorgt für eine schnellere Konvergenz als eine Selektion ohne Gewichtung.
Um aus den beiden gefundenen Elternteilen ein neues Individuum zu erschaffen, wird ein **uniformes Crossover**-verfahren verwendet. Das bedeutet, für jedes Gen des Individuums (also jede

[2] `random.paretovariate(.2) -1`: Vom Ergebnis der Zufallsfunktion wird 1 subtahiert, so dass index = 0 erreicht werden. Ohne diese Verschiebung beginnt eine Paretoverteilung bei 1. Außerdem ist die Paretoverteilung eine stetige Verteilung, so dass der stetige Wert auf einen ganzzahligen Wert abgerundet werden muss.

Option für den Compiler) wird zufällig gleichverteilt ermittelt, von welchem Elternteil es kopiert wird. Somit kann es auch geschehen, das ein Großteil des Genoms von nur einem der beiden Eltern stammt.

Implementierungsdetails zur Rekombination zweier Individuen:

```
def crossover(self, other):
    child = Individuum(other=self)
    father = random.sample(other.values.keys(),
                random.randint(1, int(len(other.values)/2)))
    for p in father:
        child.values[p] = other.values[p]
    return child
```

Nach der Rekombination findet mit dem neu entstandenen Individuum mit einer gewissen Wahrscheinlichkeit[3] eine **Mutation** statt. Hierbei wird ein einzelnes Gen des Individuums zufällig verändert, indem es auf einen anderen zufälligen Wert gesetzt wird. Bei Optionen mit einem großen Wertebereich wird ein Mutationsschritt verwendet, der nicht völlig zufällig aus dem Wertebereich wählt, sondern der ein Element bestimmt, das zum aktuellen Element benachbart im Wertebereich liegt. Hierbei wir eine maximale Schrittgröße von zwei verwendet: Der neue Wert darf nicht mehr als zwei Elemente vom ursprünglichen Wert entfernt sein. Dies wirkt sich zugunsten der *Exploitation* (der Suche in der Nähe des bisherigen lokalen Optimums) aus, aber auch zulasten der *Exploration* (der Suche in noch nicht erforschten Gebieten des Suchraums) auswirken. Deshalb wird mit 20% Wahrscheinlichkeit auch bei großen Wertebereichen rein zufällig gewählt. Bei kleinen Wertebereichen gibt es wenig Unterschiede im Verhalten des genetischen Algorithmus, wenn statt einer zufälligen Mutation eine maximale Schrittgröße bestimmt wird. Deshalb wird in diesem Fall weiterhin ein zufälliger neuer Wert aus dem gesamten Wertebereich bestimmt.

Implementierungsdetails zur Mutaion:

```
#Individuum
def mutate(self):
    target = random.choice(list(self.values.keys()))
    self.values[target] = Param[target].mutate(self.values[target])

#Parameter
def mutate(self, value):
    if type(value) == bool:
        return not value
    elif len(self._rng) > 8: #large range, use mustation steps
        strategy = random.randint(-2,+2)
        if strategy == 0:
            return self.random_init()
        else:
            new_val = value + self._rng.step * strategy
            new_val = max(new_val, self._rng.start)
            new_val = min(new_val, self._rng.stop-self._rng.step)
            return new_val
    else:
        new_val = value
        while(value == new_val):
            new_val = self.random_init()
        return new_val
```

[3]Für die Implementierung wurde eine Wahrscheinlichkeit von 0.2 gewählt

Die **Evaluation** ist der aufwändigste Schritt. Hier müssen mehrere Schritte durchlaufen werden, bis am Ende die Fitness jedes Individuums ermittelt werden kann. Zuerst werden aus den virtuellen Optionen des Individuums tatsächliche Optionen berechnet, die zusammen mit den normalen (nicht virtuellen) Optionen des Individuums eine fertige Konfiguration von Compileroptionen ergeben. In diesem Schritt werden auch Einschränkungen der Optionen realisiert, die über eine Beschränkung auf einen festen Wertebereich hinausgehen. Dies ist außerdem der einzige Schritt, bei dem der genetische Algorithmus Informationen über Implementierungsdetails des ExaStencil Compilers benötigt, wie zum Beispiel die tatsächliche Realisierung von Abhängigkeiten zwischen Optionen. Wenn sich im Laufe der Entwicklung des ExaStencil Compilers entsprechende Details ändern, ist dies der einzige Schritt, der in der Implementierung des genetischen Algorithmus ebenfalls angepasst werden muss.

In diesem Schritt kann es sein, dass einzelne Individuen bereits verworfen werden, falls durch Einschränkungen bereits klar wird, dass keine gültige Konfiguration für den Compiler vorliegt. Dies stellt jedoch einen Ausnahmefall dar, der durch ein gutes Design von virtuellen Optionen normalerweise verhindert werden kann.[4]

Die so entstandene Konfiguration wird zusammen mit vorgegebenen festen Werten (die die Problemstellung angeben) in Form einer .knowledge-Datei dem ExaStencils Compiler übergeben. Dem Compiler ist es möglich, einzelne Optionen zu verändern, um ein gültiges Programm zu erzeugen. Diesen Vorgang teilt der Compiler in Form von Warnungen mit, so dass das Individuum entsprechend angepasst werden kann. Somit können eigentlich ungültige Individuen während der Evaluation vom Compiler repariert werden, ohne dass der Compilevorgang erneut gestartet werden muss.[5]

Der ExaStencil Compiler erzeugt C++ Code, der anschließend automatisch gebaut und ausgeführt werden kann. Das ausgeführte Programm liefert einige Metriken zurück, die als Optimierungsziele für den genetischen Algoritmus verwendet werden können. Im Folgenden wird aussschließlich die Gesamtlaufzeit des ausführbaren Proramms als Maß für die Güte verwendet. Die Fitness jedes Individuums berechnet sich somit aus dem Inversen der durchschnittlichen gemessenen Laufzeit (zum Beispiel $\frac{1}{0.27ms}$).

Es kann gelegentlich vorkommen, dass Individuen erzeugt werden, die eine ungültige Konfiguration repräsentieren. Durch die virtuellen Parameter wird versucht die Entstehung solcher Individuen soweit möglich zu verhindern; Durch die Überprüfung von Beschränkungen können ungültige Individuen vor dem Compilevorgang repariert oder verworfen werden. Es kann vorkommen, dass trotz dieser Methoden ungültige Individuen beim Compiler ankommen. In diesem Fall wird der Compile- oder der Ausführvorgang mit einem Fehler abbrechen, woraufhin das Individuum verworfen wird.

Der Evaluationsvorgang ist der Schritt im genetischen Algorithmus, der am meisten zeitintensiv ist, da für jedes Individuum der Population ein kompletter Compilevorgang durchgeführt werden muss.

Am Ende folgt noch eine **Elite-Selektion**: Von der Elterngeneration werden die besten fünf Prozent der Individuen in die Nachfolgegeneration übernommen. Den Rest der Nachfolgegeneration stellen die (gültigen) Kinder dar. Dadurch wird sichergestellt, dass die Gene sehr guter Individuen nicht aus der Population verschwinden, wenn die erzeugten Kinder nicht so gute Fitness aufweisen. Gleichzeitig wird dafür gesorgt, dass eine Population nicht stagniert, da 95% der Elterngeneration verworfen und durch ihre Kinder ersetzt werden, auch wenn die Fitness der schlechteren Kinder nicht so gut ist, wie die der schlechteren Eltern. Es wird also erlaubt, dass sich ein Teil der Population wieder verschlechtert, wodurch lokale Optima wieder verlassen werden können.

Abbruchkriterium: Nach spätestens einer vorgegebenen Anzahl an Iterationen wird der Algorithmus beendet. (Eine Iteration ist die einmalige Durchführung aller Schritte an allen Individuen

[4] In Tests zeigt sich, dass dadurch bei Startpopulationen etwa 32% der Individuen nicht evaluiert werden können. Bei Individuen, die durch Rekombination entstehen, ist diese Quote geringer

[5] Es gibt drei Fälle die eintreten können: 1. die übergebene Konfiguration ist gültig und wird compiliert. 2. die übergebene Konfiguration ist ungültig und der Compiler beendet sich mit einer Fehlermeldung. 3. die übergebene Konfiguration ist ungültig, aber der Compiler kann sie anpassen, so dass er mit einer gültigen Konfiguration weiterarbeiten kann. In diesem Fall weiß der genetische Algorithmus aber noch nicht, dass die Konfiguration angepasst wurde. Deshalb muss er die Warnungen des Compilers einlesen und das Individuum so anpassen, dass die Konfiguration, mit der der Compiler arbeitet, der Konfiguration entspricht, die das Individuum repräsentiert

der Population.) Der Algorithmus kann sich auch vorzeitig beenden, wenn sich über mehrere Generationen[6] hinweg die Fitness des jeweils besten Individuums nicht verbessert. Der Algorithmus kann auf diese Weise feststellen, dass die Population in einem lokalen Optimum stagniert, und eine Verbesserung in nächster Zeit unwahrscheinlich ist. In diesem Fall ist es für die Optimierung zielführender, mit einer neuen zufällig gewählten Population erneut zu beginnen, als zu versuchen, die bestehende Population dazu zu bringen das gefundene lokale Optimum wieder zu verlassen.

Es können verschiedene **Probleme bei der Durchführung** auftreten. Einige davon werden so gut wie möglich umgangen; ihr Auftreten kann jedoch nicht komplett verhindert werden.

Wenn ein sehr großer Teil des Suchraums ungültige Lösungen darstellt, kann es passieren, dass während den ersten Generationen komplette Populationen generiert werden, die ungültig sind. Solange es nicht mindestens zwei gültige Individuen gibt, kann keine Rekombination stattfinden; der Algorithmus verkommt zu einer zufälligen Suche, die einige Zeit brauchen kann, bis genug Individuen für eine Rekombination gefunden wurden.

Wenn die Wahrscheinlichkeit hoch ist, dass die Rekombination ein ungültiges Individuum erzeugt, fällt es einer kleinen Population sehr schwer zu wachsen.

Unter Umständen existieren nur sehr wenige Individuen, aus denen keine weiteren Individuen erzeugt werden können, die eine gültige Konfiguration repräsentieren. In diesem Fall, der aus dem ersten Problem hervorgehen kann, wird sich die Population nicht verbessern können. Der genetische Algorithmus stagniert bei einigen zufälligen Punkten im Suchraum.

Beide Probleme lassen sich vermeiden, indem sichergestellt wird, dass möglichst wenig ungültige Individuen erzeugt werden können. Wie bereits erwähnt, wird dies durch virtuelle Optionen realisiert, die Einschränkungen des Suchraums umsetzen. Im Augenblick wird durch diese Technik die Anzahl der ungültigen Individuen soweit reduziert, dass der genetische Algorithmus nicht von Anfang an stagniert, sondern nach einiger Zeit eine gute Lösung findet.

Ein weiteres Problem stellen Begrenzungen durch die Hardware dar. Wenn die Lösungen mit steigendem Speicherverbrauch besser werden, wird irgendwann eine Grenze erreicht, die den kompletten Speicher der Hardware auslastet. In diesem Fall versucht der genetische Algorithmus sich dieser Grenze so dicht wie möglich anzunähern, was je nach Konfiguration des Betriebssystems zum Einfrieren der Rechner führen kann.[7]

Die gefundenen Optima stellen somit eine Lösung dar, die stark plattformabhängig ist.

Dies ist jedoch sowieso der Fall, da einige der zu optimierenden Compileroptionen ohnehin von der verwendeten Hardware abhängen können (zum Beispiel der Grad der Parallelisierung von der Anzahl der Prozessorkerne). Als Fazit lässt sich festhalten, dass die Optimierung in jedem Fall auf der Zielarchitektur durchgeführt werden muss.

3.4 Auswertung des genetischen Algorithmus

Dieser genetische Algorithmus wurde implementiert und auf vier Problemspezifikationen angewendet. Das Ziel war es, automatisiert eine Konfiguration zu finden, die Programme mit möglichst geringer Laufzeit erzeugt. Es ist zu beobachten, dass nach spätestens 400 Compileraufrufen auch bei schlechten Startpopulationen eine gute Konfiguration gefunden werden kann, die sich wesentlich performanter verhält, als eine von einem Menschen willkürlich festgelegte Ausgangskonfiguration. In einigen Fällen sind nach weiteren Durchgängen weitere geringfügige Verbesserungen zu bemerken.

Fazit

Durch diese Experimente (siehe Abbildung 6) zeigt sich, dass der genetische Algorithmus geeignet ist, die Parameterliste für den ExaStencil Compiler hinsichtlich eines gewählten Optimierungsziels (Laufzeit, Fehler, etc...) zu optimieren. Es kann in der Regel ein Optimum gefunden werden, dessen Performanz besser ist als die einer von Menschen willkürlich gewählten Parameterbelegung.

[6] 10% der maximalen Iterationen

[7] Im besten Fall wird bei kritischem Speicherverbrauch das ausgeführte Programm terminiert; damit erhält der genetische Algorithmus die Rückmeldung über den Absturz, so dass das entsprechende Individuum als ungültig behandelt wird. Im schlimmsten Fall wird der Auslagerungsspeicher hinzugenommen, was das ausgeführte Programm stark verlangsamt. Es kann sogar passieren, dass hierdurch der komplette Rechner unbedienbar wird, wodurch der genetische Algorithmus nicht fortfahren kann.

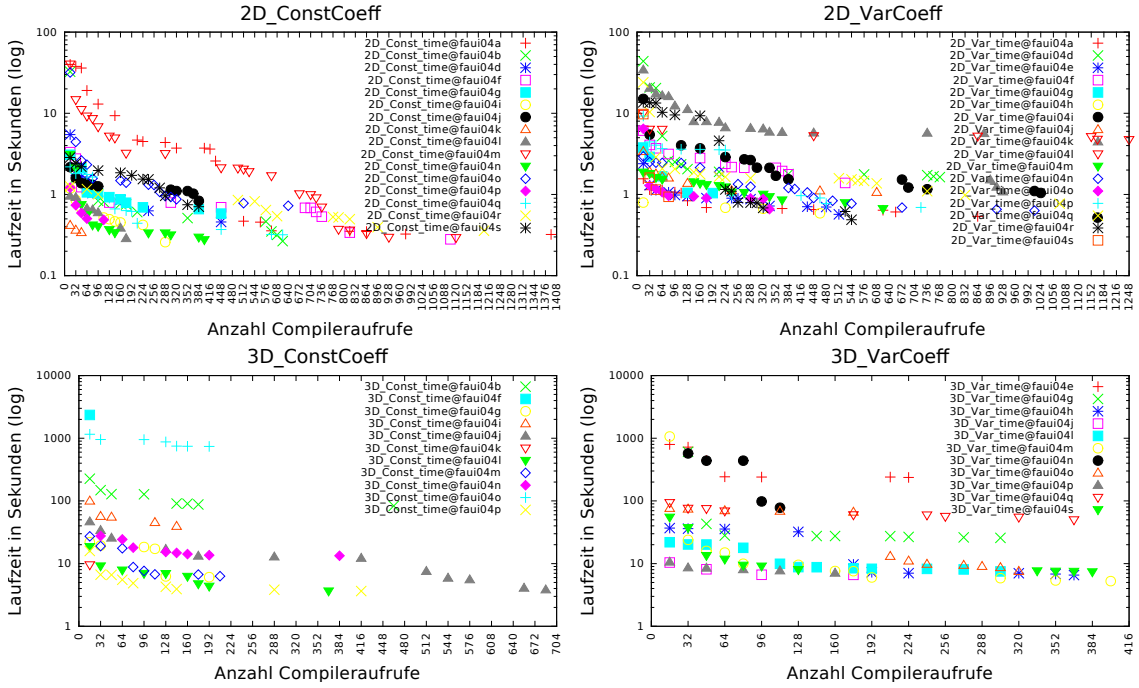

Abbildung 6: *Der genetische Algorithmus wird auf vier Probleme angewendet. Die eingezeicheten Punkte kennzeichnen die Laufzeit des jeweils besten Individuums einer Population zu dem Zeitpunkt, zu dem es gefunden wurde (angegeben in Compileraufrufen, die insgesamt bisher stattgefunden haben). Eine geringe Laufzeit entspricht einer hohen Fitness. Der genetische Algorithmus wurde jeweils mehrfach parallel durchgeführt, so dass die besten Individuen mehrerer unabhängiger Populationen zu sehen sind (gleiche Farbe bedeutet gleiche Population). Man sieht, dass der genetische Algorithmus im Lauf der Zeit immer bessere Individuen findet, bis er irgendwann in einem lokalen Optimum stagniert. Die Startpopulation hat anfangs großen Einfluss auf die Suche, jedoch kann in den meisten Fällen auch bei einer schlechten Startpopulation ein gutes Optimum gefunden werden.*

Dieser Vorgang findet voll automatisiert statt, so dass auch neue Probleme automatisch optimiert werden können.

Die Laufzeit des genetischen Algortihmus variiert stark, da vor allem in den zufälligen Startpopulationen gelegentlich sehr hohe Laufzeiten der generierten Programme auftreten.[8] Zudem wird der Algorithmus frühzeitig abgebrochen, sobald mehrere Generationen lang keine Verbesserung der Population beobachtet werden kann. Die Gesamtlaufzeit des genetischen Algorithmus (bei 50 Iterationen mit einer Populationsgröße von 16, also maximal 800 Compileraufrufen) lag bei den Experimenten zwischen 32 Minuten und 7 Stunden und 52 Minuten, wobei sehr hohe Gesamtlaufzeit selten vorkommen.

Es wird bei jedem Optimierungsvorgang immer nur ein einzelnes lokales Optimum gefunden; es wird aber keine Aussage darüber getroffen, wie gut das gefundene Optimum wirklich ist. Da das globale Optimum nur sehr schwer gefunden werden kann, ist im Allgemeinen keine Aussage über die absolute Güte des lokalen Optimums möglich. Des Weiteren geht aus dem gefundenen Optimum nicht hervor, welche Parameter tatsächlich für die Güte relevant sind, oder welche der Parameter problemabhängig bzw. hardwareabhängig sind. Auch Beziehungen und Abhängigkeiten der Parameter untereinander sind nicht erkennbar.

Einige dieser Fragen sollen im folgenden Kapitel beantwortet werden.

[8] Die höchste gemessene Laufzeit beträgt etwa 7800 Sekunden.

4 Extraktion von domänenspezifischem Wissen

Es zeigt sich, dass zwar ein lokales Optimum mittels des genetischen Algorithmus gefunden werden kann, jedoch kann noch keine Aussage über die Güte des Optimums getroffen werden. Bei mehreren Durchläufen des genetischen Algorithmus können zwar mehrere Optima gefunden werden, so dass man deren Güte untereinander vergleichen kann. Jedoch kann man daraus noch nicht direkt erkennen, *warum* die gefundenen Optima besser oder schlechter sind. Die Parameter der gefundenen Optima sollen nun analysiert und untereinander verglichen werden, um herauszufinden, welche Parameter zur Güte beitragen und welche für dieses Problem irrelevant sind.

Der folgende Teil der Arbeit beschreibt einen Versuch, aus der Analyse der vom genetischen Algorithmus gefundenen Optima einer Problemspezifikation Wissen zu generieren, was unter anderem dazu verwendet werden kann, zukünftige Optimierungsvorgänge zu vereinfachen.

4.1 Ausgangslage

Wir gehen hier davon aus, dass mit Hilfe des genetischen Algorithmus eine Vielzahl an Konfigurationen gefunden wurden, die, wenn sie dem ExaStencils Compiler übergeben werden, in Programmen resultieren, die über eine mehr oder weniger gute Laufzeit (oder ähnliche Gütemerkmale) verfügen. Die Datenmenge beinhaltet nicht nur die lokalen Optima, sondern auch Konfigurationen, die vom genetischen Algorithmus im Laufe der Optimierung erzeugt wurden. Daraus ergibt sich eine Verteilung, die viele Konfigurationen enthält, die sich in der Nähe von lokalen Optima befinden, aber auch einige, die weit von den lokalen Optima entfernt sind (zum Beispiel schlechte Individuen der Startpopulationen).

Diese Konfigurationen stellen Punkte im Suchraum dar, der durch die Wertebereiche der Konfigurationsparameter aufgespannt wird. Diese Konfigurationen unterscheiden sich untereinander in der gemessenen Güte (Laufzeit, Fitness) und in der Belegung ihrer Parameter.

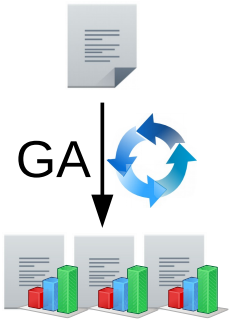

Abbildung 7: *schematischer Ablauf des Vorgangs: aus einer Ausgangskonfiguration werden viele weitere Konfigurationen und deren Fitnesswerte generiert.*

Zielsetzung

Es soll herausgefunden werden, welche der Parameter einer Konfiguration bei einem gegebenen Problem zur Güte beitragen, und ob es Parameter gibt, die für das Problem irrelevant sind. Später soll darüber hinaus ermittelt werden, ob es Parameter gibt, deren Belegung abhängig von der verwendeten Hardware zu unterschiedlicher Güte führt und ob es Parameter gibt, die rein problemabhängig sind, so dass die verwendete Hard- und Softwarearchitektur außer Acht gelassen werden kann.

4.2 Vergleich aller Konfigurationen

Als Lösungsansatz werden zunächst viele Konfigurationen erzeugt, indem der genetische Algorithmus mehrfach auf dem gleichen Problem angewendet wird. Es werden dadurch mehrere lokale Optima gefunden, aber auch viele Konfigurationen in der Nähe dieser Optima (die also über eine ähnliche Laufzeit verfügen). Es reicht nicht, das beste gefundene Optimum zu betrachten: man erhält zwar eine sehr gute Konfiguration, allerdings kein Wissen darüber, welche der Parameter wie relevant sind. Wenn alle lokalen Optima untereinander verglichen werden, lassen sich zwar einige Zusammenhänge erahnen, jedoch ist die Datenmenge relativ klein, wodurch eine statistische Analyse erschwert wird. Bei den durchgeführten Experimenten ist zudem eine Messungenauigkeit von bis zu 10% bei der Laufzeit zu beobachten. Die Unterschiede in der Laufzeit einiger der gefundenen Optima sind kleiner als der beobachtete Messfehler, wodurch eine sinnvolle Gewichtung der Optima erschwert wird.

Für die Analyse werden also sämtliche Konfigurationen, die von den Instanzen des genetischen Algorithmus erzeugt wurden, zu Rate gezogen. Dies sorgt für eine ausreichend große Datenmenge in

der Nähe der lokalen Optima und die einzelnen Konfigurationen können basierend auf ihrer Laufzeit gewichtet werden.

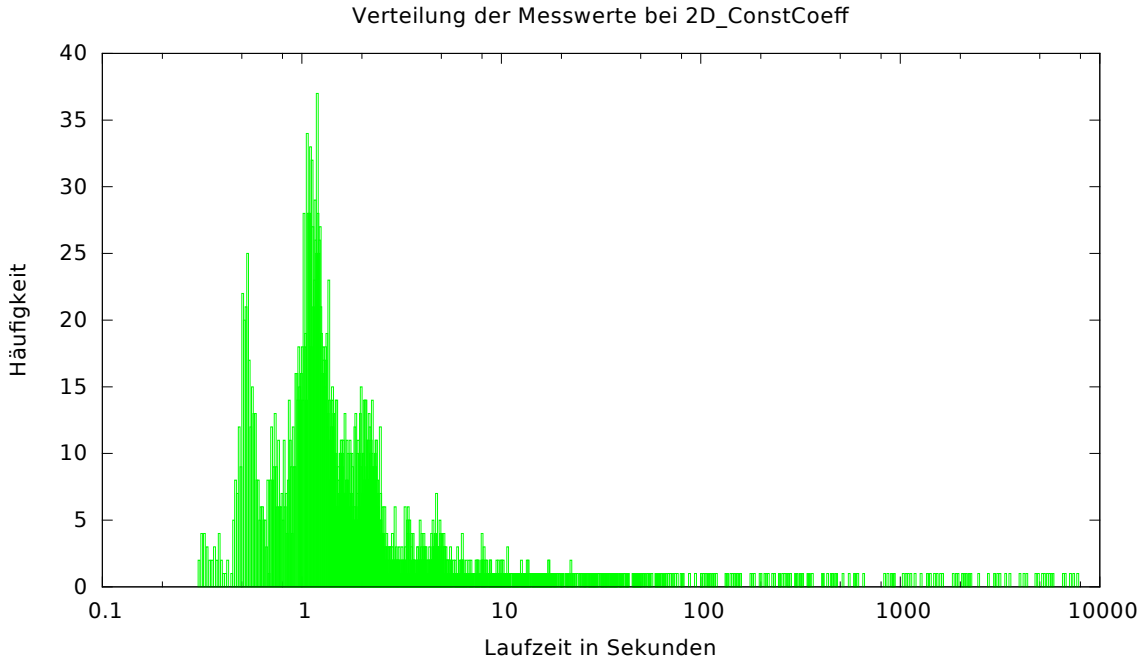

Abbildung 8: *Abgebildet ist, wie häufig welche Laufzeiten bei allen gültigen generierten Konfigurationen für das Problem 2D_ ConstCoeff gefunden wurden. Man kann eine Häufung bei lokalen Optima beobachten. Sehr hohe Laufzeiten bei geringer Häufigkeit deuten daraf hin, dass die entsprechende Konfiguration eher bei Beginn des genetischen Algorithmus erzeugt wurde, während eine große Häufung bei geringer Laufzeit darauf hindeutet, dass der genetische Algorithmus in der Nähe des von ihm gefundenen lokalen Optimums sucht.*

Der Aufbau der Messdaten (wie zum Beispiel in Abbildung 8) mach eine statistische Analyse schwierig:

Die einzelnen Messwerte sind stark voneinander abhängig, da jede Konfiguration, die nicht aus der zufälligen Startpopulation stammt, aus einer anderen in den Messdaten enthaltenen Konfiguration hervorgeht. Auch sind die Messwerte nicht normalverteilt.

Als erstes können **Parameterbelegungen identifiziert** werden, die bei dem gegebenen Problem **konstant** sind, falls solche existieren. Beim Problembeispiel 2D_ConstCoeff lässt sich feststellen, dass bei jeder gültigen Konfiguration folgende Parameter gleich sind:

```
domain_fragmentLength_z:          1    [3496]
domain_rect_numBlocks_z:          1    [3496]
domain_rect_numFragsPerBlock_z:   1    [3496]
special_numOMP_z:                 1    [3496]
```

Zu sehen ist die Ausgabe des ersten Analyseschrittes. Linke Spalte: Parametername, Mittlere Spalte: gefundene Belegung, Rechte Spalte: Anzahl der Konfigurationen in den Messdaten, bei welchen dem Parameter der entsprechende Wert zugewiesen wurde. In diesem Beispiel wurden insgesamt 3496 Messergebnisse analysiert, wovon alle in den vier genannten Parameterbelegungen übereinstimmen.

Das liegt in diesem Fall daran, dass der Compiler diese Werte auf 1 setzt. Dieses Wissen lässt sich für zukünftige Optimierungsvorgänge verwenden: Die Parameter können fest auf diese Belegungen eingestellt werden und es müssen keine Ressourcen darauf verwendet werden, andere Belegungen auszuprobieren. Im Falle des genetischen Algorithmus wird dies noch recht wenig nützen, da der Compiler diese Werte immer automatisch auf 1 setzt. Da der genetische Algorithmus damit umgehen kann, wird der Wert bei jeder Rekombination bereits richtig gesetzt sein. Bislang laufen lediglich einige Mutationen ins Leere, die versuchen diese Werte zu ändern. Werden sie vorher festgelegt, werden von der Mutation nur noch Parameter verändert, die auch tatsächlich relevant sind. Somit kann der genetische Algorithmus unter Umständen ein paar Iterationen weniger Zeit brauchen, ein Optimum zu finden.

4.3 Finden von relevanten Parametern

Als nächstes gilt es Parameter zu finden, die für die Performaz relevant sind.

Um solche Parameterbelegungen zu finden, die für eine gute Performanz sorgen, werden die Messwerte ausgewählt, die über sehr niedrige Laufzeit verfügen, um sie auf Gemeinsamkeiten hin zu untersuchen. Hierzu kann ein *greedy Hillclimbing* Algorithmus verwendet werden:

Es wird solange immer die Konfiguration mit der schlechtesten Laufzeit entfernt, bis einer der Parameter nur noch mit einer eindeutigen der möglichen Belegungen auftritt. Im Beispiel:

```
domain_rect_numFragsPerBlock_x:      1       [736]
domain_rect_numFragsPerBlock_y:      1       [736]
omp_parallelizeLoopOverDimensions:   true    [736]
omp_parallelizeLoopOverFragments:    false   [736]
fitness of 736th entry:              1.0679167
```

Der *greedy Hillclimbing* Algorithmus findet heraus, dass **die besten** 736 (von insgesamt 3496) Messwerte in der Parameterbelegung der vier genannten Parameter **übereinstimmen**. Dass alle vier dieser Parameter gleichzeitig (erst wenn alle bis auf die besten 736 verworfen wurden) über eine eindeutige Belegung verfügen, lässt darauf schließen, dass knapp oberhalb (Nr. 737) ein lokales Optimum liegt, das nicht in allen vier Parametern die gleichen Belegungen aufweist. In Abbildung 9 ist dies eingezeichnet.

Durch dieses Vorgehen werden nicht nur die **Parameter gefunden**, die **sehr großen Einfluss auf die Performanz** haben, sondern es wird auch eine **Schwelle** definiert, ab der die **Fitness** einer Konfiguration als **gut** erachtet werden kann. Das unterstützt die Automatisierung der Analyse und künftiger Optimierungsvorgänge, da diese Schwelle automatisiert gefunden werden kann, ohne dass ein Mensch irgendeinen festen Schwellenwert festlegen muss. Auch dieses Wissen lässt sich für künftige Optimierungsvorgänge verwenden: Die gefundenen Parameterbelegungen können im Rahmen einer spätere Optimierung als konstant angenommen werden. Hierdurch wird der Suchraum verkleinert, während sich das Risiko, das globale Optimum auszuschließen, als gering angenommen werden kann. Gleichzeitig wird ein lokales Optimum (das direkt oberhalb der Schwelle liegt) komplett ausgeschlossen. Wird der genetische Algorithmus erneut auf alle anderen Parameter angewendet, werden einige Konfigurationen mit sehr hoher Laufzeit vermieden, so dass die Gesamtlaufzeit des genetischen Algorithmus sinkt. Die Optimierung würde ohne das Festlegen der Parameter die ersten Generationen darauf verwenden, diese wie angegeben zu belegen. Somit kann die Optimierung am Angang beschleunigt werden. Im späten Verlauf der Optimierung werden einige unnötige Mutationen vermieden, was aber nur noch eine eher geringe Verbesserung bringt.

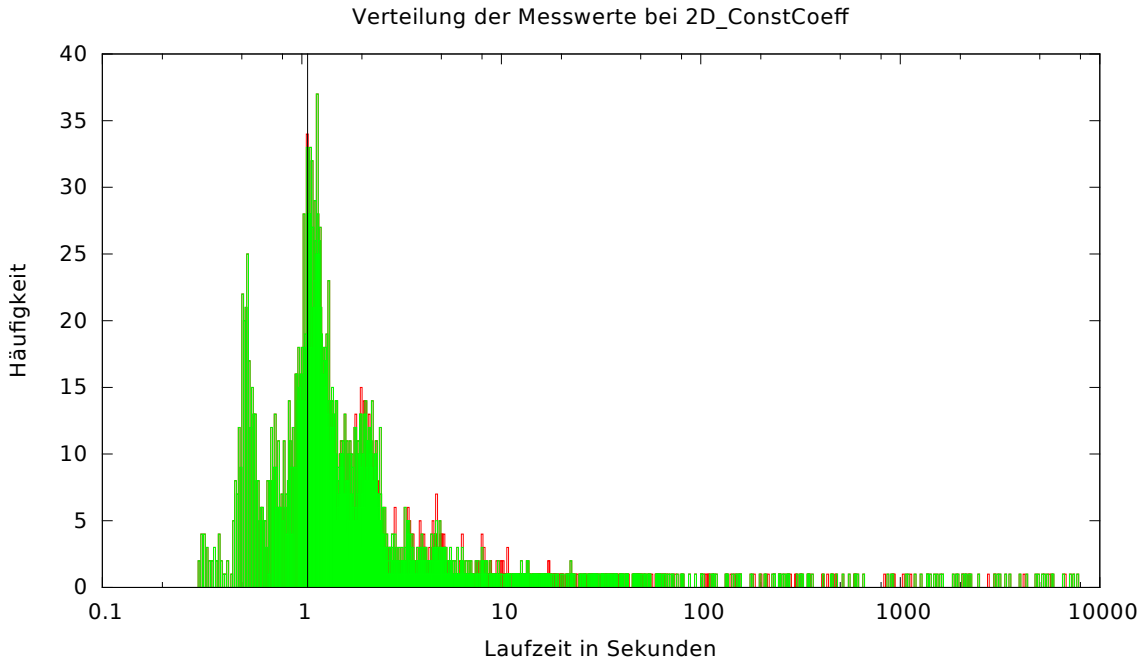

Abbildung 9: *Als schwarze Linie eingezeichnet ist die Laufzeit des 736. Messwertes. (Die Laufzeit des 736. Messwertes liegt bei 1.07 Sekunden.) Alle Konfigurationen links dieser Markierung stimmen in den vier oben genannten Parameterbelegungen überein. Grün eingezeichnet sind alle Konfigurationen, die mit den genannten Parameterbelegungen übereinstimmen, rot sind die Konfigurationen, die von den genannten Parameterbelegungen abweichen*

Die Parameterbelegungen werden jedoch nicht zwangsweise gleichzeitig gefunden. Siehe folgendes Beispiel (das aus einer Optimierung von 3D_VarCoeff hervorgeht):

```
maxLevel:                          4       [1102]
poly_tileSize_x:                   32      [1102]
domain_rect_numFragsPerBlock_y:    1       [951]
domain_rect_numFragsPerBlock_z:    1       [942]
omp_parallelizeLoopOverFragments:  false   [904]
domain_rect_numFragsPerBlock_x:    1       [904]
special_numNodes:                  2       [132]
...
```

Bei 3D_VarCoeff werden nur zwei Parameterbelegungen bei 1102 gefunden, kurz darauf hintereinander weitere vier. Dieses Verhalten kann den Ablauf der Optimierung widerspiegeln: einzelne Parameter wurden verändert, bis eine bessere Belegung erreicht wurde. Wenn im späteren Verlauf eine weitere Optimierung mit eingeschränktem Suchraum durchgeführt werden soll, besteht in diesem Fall zwar die Möglichkeit nur *maxLevel=4* und *poly_tileSize_x=32* festzulegen, es kann jedoch auch zielführender sein, zusätzlich weitere Parameter festzulegen. Es macht dann jedoch keinen Sinn, stupide den nächst besseren zu nehmen (*domain_rect_numFragsPerBlock_y=1 [951]*), sondern es sollten alle bis einschließlich *domain_rect_numFragsPerBlock_x=1 [904]* als Gruppe hinzugenommen werden. Im Beispiel sieht man, dass die 951 besten Konfigurationen die Parameterbelegung *domain_rect_numFragsPerBlock_y=1* aufwesien, von denen wiederum die besten 904 zusäzlich *domain_rect_numFragsPerBlock_z, omp_parallelizeLoopOverFragments,*
domain_rect_numFragsPerBlock_x festlegen.

4.4 Clustering

Um dies zu realisieren, kann ein einfacher **Clustering**-Ansatz wie zum Beispiel *single link clustering* verwendet werden: Die Daten werden zu Gruppen zusammengefasst, in dem solange immer die beiden Untergruppen, deren Randelemente den geringsten Abstand zueinander haben, verbunden werden, bis eine gewünschte Anzahl an Gruppen erzeugt wurde.

Die für die Analyse optimale Anzahl an Gruppen kann sich von Problem zu Problem unterscheiden, deshalb wird die gewünschte Gruppenzahl dem Analyseprogamm als Parameter übergeben.

Im oberen Beispiel ergibt sich:

```
['maxLevel', 'poly_tileSize_x']
['domain_rect_numFragsPerBlock_x', 'domain_rect_numFragsPerBlock_y',
 'domain_rect_numFragsPerBlock_z', 'omp_parallelizeLoopOverFragments']
[ 'special_numNodes', ...]
...
```

Würde eine geringere Anzahl an Gruppen erzeugt werden, würden auch die ersten beiden Gruppen zu einer verschmelzen.

Das hier gefundenen Wissen **teilt Parameter in Gruppen** ein, deren Elemente ähnlich relevant für das Problem sind. Die Gruppen selbst sind **nach Relevanz sortiert**, so dass es für zukünftige Optimierungsvorgänge möglich ist, eine oder mehrere dieser Gruppen auszuwählen, deren Parameterbelegungen festzusetzen und für die übrigen Parameter weiter zu suchen. Somit kann der Suchraum für zukünftige Optimierungen stark eingeschränkt werden.

Die konkreten Wertzuweisungen können zusammen mit ihrer Relevanz als typisches Merkmal des zu optimierenden Problems gesehen werden. Dies soll später den Vergleich zwischen verschiedenen Problemen ermöglichen.

Es muss beachtet werden, dass die zweite Gruppe von Parameterbelegungen von der ersten Gruppe abhängt. Siehe folgendes Beispiel (2D_ConstCoeff):

```
omp_parallelizeLoopOverDimensions:   true    [736]
domain_rect_numFragsPerBlock_x:      1       [736]
omp_parallelizeLoopOverFragments:    false   [736]
domain_rect_numFragsPerBlock_y:      1       [736]
domain_rect_numBlocks_x:             1       [304]
special_numOMP_x:                    2       [304]
domain_fragmentLength_x:             2       [304]
```

Die besten 304 Konfigurationen weisen obige Parameterbelegung auf. Die Aussage, dass *domain_rect_numBlocks_x, special_numOMP_x, domain_fragmentLength_x* jedoch die zweitrelevanteste Parametergruppe (*nach omp_parallelizeLoopOverDimensions, domain_rect_numFragsPerBlock_x, omp_parallelizeLoopOverFragments, domain_rect_numFragsPerBlock_y*) ist, kann jedoch nicht getroffen werden, da eine Abhängigkeit besteht. Es kann lediglich ausgesagt werden, dass die zweite Gruppe relevant ist, unter der Bedingung, dass die Parameterbelegung der ersten Gruppe bereits erfolgt ist.

Ein generelles Problem, das dieser *greedy-Hillclimbing*-Ansatz aufweist ist, dass die Menge an Messdaten, die für den jeweils nächsten Schritt verwendet werden kann, mit jedem Schritt stark schrumpfen kann.

```
...
domain_fragmentLength_x:    2        [304]
special_numNodes:           2        [291]
opt_useAddressPrecalc:      true     [279]
special_ranksPerNode:       4        [276]
domain_rect_numBlocks_y:    1        [276]
special_numOMP_y:           2        [276]
domain_fragmentLength_y:    2        [276]
mpi_numThreads:             1        [276]
l3tmp_smoother:             "RBGS"   [250]
special_secondDim:          0        [35]
...
```

Die 35 besten Konfigurationen aus den ursprünglich 3496 weisen die oben genannte Parameterbelegung auf. Je kleiner jedoch die betrachtete Datenmenge ist, desto schwieriger ist es, weitere Aussagen zu treffen. Im obigen Fall kann es zwar sinnvoll sein, für einen zukünftigen Optimierungslauf alle Parameterbelegungen bis einschließlich *l3tmp_smoother="RBGS"* festzulegen, um gezielt in einem vielversprechenden Teil des Suchraums zu suchen, sobald jedoch *special_secondDim=0* festgelegt wird, ist die Wahrscheinlichkeit groß, wieder zu dem bereits gefundenen lokalen Optimum zu konvergieren.

4.5 Mehrere Clustering Heuristiken

Um das auszugleichen, wird nun nicht alleine der vorgestellte *greedy-Hillclimbing*-Algorithmus verwendet, sondern er fungiert als eine Heuristik, die zusammen mit zwei anderen Heuristiken die Relevanz der einzelnen Parameterbelegungen anzugeben versucht.

Als zweite Heuristik wird ein ähnlicher *greedy-Hillclimbing*-Algorithmus verwendet, der jedoch nicht die Konfiguration mit der höchsten Fitness entfernt, sondern versucht, die Parameterbelegung zu streichen, die den Durchschnitt der Performanz-Werte am meisten negativ beeinflusst. Hierdurch sollen gezielt schlechte Parameterbelegungen entfernt werden, ohne dass vorher Konfigurationen mit eigentlich besseren Belegungen entfernt werden müssen.

Die dritte Heuristik betrachtet von allen Parametern denjenigen, deren zwei besten Belegungsmöglichkeiten den größten Unterschied im Durchschnitt der Performanz-Werte ausmachen und wählt die Belegung, die für einen besseren Durchschnitt sorgt. Diese Heuristik streicht keine Konfigurationen, sondern betrachtet immer alle Messwerte unterhalb einer festgelegten Schwelle (es wird die vorher gefundene Schwelle benutzt: die Konfiguration, bei der die erste Parameterbelegung eindeutig ist). Diese Heuristik betrachtet alle Parameter, als wären sie unabhängig, während die vorhergehenden die Abhängigkeiten in der Reihenfolge betrachten, in denen sie im jeweiligen Algorithmus aufgebaut werden.

Die drei **Heuristiken** liefern am Ende eine ähnliche Reihenfolge an **Relevanz** für die **Parameterbelegungen**. Zusammengenommen können sie für eine Bewertung der Relevanz benutzt werden, ohne dass die Schwächen einer einzelnen Heuristik zu stark zum Tragen kommen.

4.6 Ergebnis

Das implementierte Programm verwendet die genannten Analyse- und Clustertechniken, um für die übergebenen Messdaten eines Problems eine heuristische Bewertung zu finden. Jeder Parameterbelegung, die zu guter Laufzeit führt, wird ein Wert zugewiesen, der angibt, wie relevant für die Laufzeit diese Parameterbelegung geschätzt wird. Dieser Wert entspricht der Summe der Bewertung der verwendeten Heuristiken, die wiederum den Rang der Klasse angeben, in den die Parameterbelegung einsortiert wurde. Im folgengen Beispiel wurden von drei Cluster-Heuristiken

die Parameterbelegungen jeweils in drei Klassen unterteilt. Somit erhalten alle Parameterbelegungen, die in allen drei Heuristiken zur relevantesten Klasse gehören, eine Bewertung von 9. alle, die in den jeweils zweitbesten Klassen vorkommen, eine Bewertung von 6. Eine Bewertung von 4 resultiert daraus, dass eine der Heuristiken die Parameterbelegung höher bewertet als die anderen beiden. Je weniger ein Parameter relevant ist, desto häufiger kann dieses Verhalten auftreten: zwei der Heuristiken reduzieren den Datensatz bei jedem Schritt stark, während eine der Heuristiken nur den kompletten Datensatz betrachtet. Das führt zu unterschiedlicheren Bewertungen, je mehr Schritte die Heuristiken bereits durchgeführt haben.

```
#relevance
9:  special_numOMP_z:                 1
9:  domain_rect_numFragsPerBlock_y:   1
9:  omp_parallelizeLoopOverFragments: false
9:  domain_fragmentLength_z:          1
9:  omp_parallelizeLoopOverDimensions: true
9:  domain_rect_numFragsPerBlock_z:   1
9:  domain_rect_numBlocks_z:          1
9:  domain_rect_numFragsPerBlock_x:   1
6:  l3tmp_smoother:                   "RBGS"
6:  domain_rect_numBlocks_x:          1
6:  special_numOMP_y:                 2
6:  domain_fragmentLength_x:          2
6:  opt_useAddressPrecalc:            true
6:  special_numNodes:                 2
6:  special_numOMP_x:                 2
6:  special_ranksPerNode:             4
6:  domain_fragmentLength_y:          2
6:  domain_rect_numBlocks_y:          1
6:  mpi_numThreads:                   1
4:  special_secondDim:                0
4:  opt_vectorize:                    false
4:  data_alignFieldPointers:          false
4:  comm_useFragmentLoopsForEachOp:   true
4:  simd_avoidUnaligned:              false
4:  l3tmp_numPre:                     1
4:  l3tmp_numPost:                    2
...
```

Das Ergebnis ist eine Liste von Gruppen, die der nach Relevanz der in ihr enthaltenen Parameterbelegungen sortiert ist. Diese Liste ist einem konkreten Problem zugeordnet und die Bewertungen der Gruppen geben keine absolute, sondern nur eine relative Bewertung der Relevanz an. Somit ist die Bewertung mit den Bewertungen der Gruppen anderer Probleme vergleichbar. Das wird später bei der Generierung domänenspezifischen Wissens von Nöten sein.

Je nach Wahl der Anzahl der zu bildenden Klassen können die Ergebnisse variieren. Bei wenigen Klassen ist das Ergebniss relativ ungenau. Bei zu vielen Klassen kann es passieren, dass Parameterbelegungen, die nahe beieinander liegen, nicht gruppiert werden.

5 Generierung von domänenspezifischem Wissen

Im bisherigen Verlauf der Arbeit wurde ein genetischer Algorithmus vorgestellt, der für ein gegebenes Problem lokale Optima hinsichtlich der Laufzeit findet und ein Analyseprogramm, das die Ergebnisse des genetischen Algorithmus verwendet, um die Parameterbelegungen zu finden, die für das gegebene Problem am zuverlässigsten für gute Performanz sorgen. Den größte Rechenaufwand stellt hierbei das generieren und ausführen von möglichen Konfigurationen durch den genetischen Algorithmus dar, der für jedes Problem einzeln ausgeführt werden muss. Jedoch sind sich die zu optimierenden Probleme in einigen Punkten ähnlich, bzw. unterscheiden sich nur in einigen wenigen Merkmalen (zum Beispiel Dimensionalität), so dass es möglich ist, aus dem bereits generiertem Wissen über einige Probleme Rückschlüsse auf ähnliche Probleme zu ziehen. Im Folgenden sollen aus dem Vergleich verschiedener Probleme relevante Parameterbelegungen für bestimmte Eigenschaften der Probleme gefunden werden. Anschließend wird daraus Wissen für ein noch nicht optimiertes und analysiertes Problem generiert und überprüft, ob dieses Verfahren zu der gleichen Lösung wie eine Optimierung durch den genetischen Algorithmus führt.

5.1 Ergebnisse der Analyse

Im ExaStencils Projekt wird für viele verschiedene Probleme Code generiert. Viele dieser Probleme unterscheiden sich jedoch nur darin, dass sie ein bestimmtes Problem auf unterschiedliche Art und Weise bzw. mit unterschiedlichen Parametern lösen. Hier werden als Beispiel weiterhin vier Konfiguration betrachtet, die jeweils eine finite Differenzen Diskretisierung der Poissiongleichung mit Dirichlet Randbedingungen lösen. Die Unterschiede sind hierbei die Dimension (2D oder 3D) und die Stencil Koeffizienten (konstant oder variabel).

Die vier Probleme sind sich ähnlich, so dass zu erwarten ist, dass gute gefundene Parameterbelegungen aller vier Probleme Ähnlichkeiten aufweisen. Es wurden für jedes Problem 20 Instanzen des genetischen Algorithmus gestartet und die erzeugten Konfigurationen analysiert. Es wurden jeweils die Parameterbelegungen ermittelt, die den meisten Einfluss auf die Performanz haben, wie im vorhergehenden Kapitel beschrieben:

```
==> 2D_ConstCoeff_xeon.relevance <==
#relevance
9:  domain_fragmentLength_z:              1
9:  domain_rect_numFragsPerBlock_y:       1
9:  domain_rect_numBlocks_z:              1
9:  domain_rect_numFragsPerBlock_x:       1
9:  domain_rect_numFragsPerBlock_z:       1
9:  omp_parallelizeLoopOverDimensions:    true
9:  omp_parallelizeLoopOverFragments:     false
9:  special_numOMP_z:                     1
6:  special_numNodes:                     2
6:  special_numOMP_x:                     2
6:  domain_fragmentLength_x:              2
6:  domain_fragmentLength_y:              2
6:  domain_rect_numBlocks_x:              1
6:  domain_rect_numBlocks_y:              1
6:  l3tmp_smoother:                       "RBGS"
6:  mpi_numThreads:                       1
6:  opt_useAddressPrecalc:                true
6:  special_numOMP_y:                     2
6:  special_ranksPerNode:                 4

...
```

```
...

==> 2D_VarCoeff_xeon.relevance <==
#relevance
 9:   domain_fragmentLength_z:              1
 9:   domain_rect_numBlocks_z:              1
 9:   domain_rect_numFragsPerBlock_x:       1
 9:   domain_rect_numFragsPerBlock_y:       1
 9:   domain_rect_numFragsPerBlock_z:       1
 9:   omp_parallelizeLoopOverFragments:     false
 9:   special_numOMP_z:                     1
 7:   omp_parallelizeLoopOverDimensions:    true

==> 3D_ConstCoeff_xeon.relevance <==
#relevance
 9:   domain_rect_numFragsPerBlock_x:       1
 6:   maxLevel:                             4
 6:   poly_tileSize_x:                      32

==> 3D_VarCoeff_xeon.relevance <==
#relevance
 9:   maxLevel:                             4
 9:   poly_tileSize_x:                      32
 7:   domain_rect_numFragsPerBlock_z:       1
 7:   domain_rect_numFragsPerBlock_y:       1
 7:   domain_rect_numFragsPerBlock_x:       1
 7:   omp_parallelizeLoopOverFragments:     false
```

Die Zahl in der ersten Spalte gibt einen heuristischen Wert für die Relevanz der Parameterbelegung an, die zweite Spalte gibt den Parameter an und die dritte Spalte weißt dem Parameter einen Wert zu, der zu guter Performanz führt. Es wurden nur Parameterbelegungen aufgenommen, die hinreichend wichtig sind: die Parameterbelegungen wurden von drei einzelnen Heuristiken in jeweils drei Klassen unterteilt (sehr wichtig (3), mittel wichtig (2), wenig wichtig (1)); eine Bewertung von neun Punkten bedeutet, dass die Parameterbelegung von allen Heuristiken als sehr wichtig für die Performanz eingestuft wird. Eine Bewertung von sechs Punkten bedeutet, dass die entsprechende Parameterbelegung von allen drei Heuristken als mittel wichtig für die Performanz eingestuft wird. (Es kann zwar vorkommen, dass eine Heuristik drei, eine zwei und die dritte Heuristik einen Punkt verteilt, dies kommt in der Praxis jedoch nicht vor, da die Heuristiken ähnliche Ergebnisse liefern).

5.2 Vergleich zweier Probleme

Nun können diese Ergebnisse verglichen werden: Wenn zwei Probleme sich nur in einem Aspekt, wie der Dimension oder dem Stencil Koeffizienten unterscheiden, sind die Unterschiede typisch für diesen Aspekt. Die verschiedenen vom genetischen Algorithmus erzeugten Konfigurationen und ihre Fitnesswerte können verglichen werden. Werden die Probleme 2D_ConstCoeff und 3D_ConstCoeff verglichen, ergibt sich Folgendes:

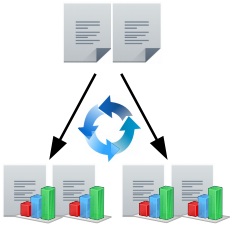

Abbildung 10: *schematischer Ablauf des Vorgangs.*

```
['2D']
domain_fragmentLength_z:              1       [1]
domain_rect_numBlocks_z:              1       [1]
domain_rect_numFragsPerBlock_y:       1       [1]
domain_rect_numFragsPerBlock_z:       1       [1]
omp_parallelizeLoopOverDimensions:    true    [1]
omp_parallelizeLoopOverFragments:     false   [1]
```

```
special_numOMP_z:                   1          [1]
domain_fragmentLength_x:            2          [2]
domain_fragmentLength_y:            2          [2]
domain_rect_numBlocks_x:            1          [2]
domain_rect_numBlocks_y:            1          [2]
l3tmp_smoother:                     "RBGS"     [2]
mpi_numThreads:                     1          [2]
opt_useAddressPrecalc:              true       [2]
special_numNodes:                   2          [2]
special_numOMP_x:                   2          [2]
special_numOMP_y:                   2          [2]
special_ranksPerNode:               4          [2]

['3D']
maxLevel:                           4          [2]
poly_tileSize_x:                    32         [2]
```

Hier wurden eine Reihe von Parameterbelegungen gefunden, die nur im zweidimensionalen Fall für gute Performanz sorgen, sowie zwei Parameterbelegungen, die nur im dreidimensionalen Fall ausschlaggebend sind. In eckigen Klammern ist der Rang der Klasse angegeben, die aus dem Clustering hervorgeht. '1' ist die Klasse der Parameterbelegungen, die am meisten Einfluss auf die Performanz haben, '2' hat am zweitmeisten. '3' wäre in diesem Fall die Klasse an Parametern, die am wenigsten relevant sind.

Wenn 2D_VarCoeff und 3D_VarCoeff miteinander verglichen werden, ergibt sich fast das gleiche Ergebnis:

```
['2D']
domain_fragmentLength_z:            1          [1]
domain_rect_numBlocks_z:            1          [1]
domain_rect_numFragsPerBlock_x:     1          [1]
domain_rect_numFragsPerBlock_y:     1          [1]
domain_rect_numFragsPerBlock_z:     1          [1]
omp_parallelizeLoopOverFragments:   false      [1]
special_numOMP_z:                   1          [1]
omp_parallelizeLoopOverDimensions:  true       [2]

['3D']
maxLevel:                           4          [1]
poly_tileSize_x:                    32         [1]
```

Hier fallen die Unterschiede geringer aus. Das hat zur Folge, dass es mehr Parameter gibt, die sowohl im zweidimensionalen als auch im dreidimensionalen Fall ähnlich großen Einfluss haben, während der Einfluss im Fall mit Konstantem Koeffizienten große Unterschiede aufweißt.

5.3 Vergleich mehrerer Probleme

Dieser Vorgang kann nun auf mehrere Probleme angewendet werden, um paarweise Unterschiede zu finden. Bei zwei Problemen, die sich nur in einem Aspekt unterscheiden, wird nach Unterschieden in den Parameterbelegungen gesucht, um den einzelnen Aspekten Parameterbelegungen zuzuordnen. Wenn die Probleme nur in einem Aspekt übereinstimmen, wird nach Gemeinsamkeiten gesucht, um eine ähnliche Zuordnung vorzunehmen. Wenn durch mehrere Unterschiede und Gemeinsamkeiten dem gleichen Aspekt unterschiedliche Parameterbelegungen zugewießen werden, werden nur die Parameterbelegungen ausgewählt, die übereinstimmen. Somit wird aus ähnlichen Ausgangskonfiguration und den auf ihnen basierenden Konfigurationen und Fitnesswerte Wissen über weitere Konfigurationen erzeugt (schematisch abgebildet in Abbildung 11). In folgendem Beispiel wurden die drei Konfigurationen 2D_ConstCoeff, 2D_VarCoeff und 3D_ConstCoeff verglichen:

```
['2D']
domain_fragmentLength_z:                1       [1]
domain_rect_numBlocks_z:                1       [1]
domain_rect_numFragsPerBlock_y:         1       [1]
domain_rect_numFragsPerBlock_z:         1       [1]
omp_parallelizeLoopOverDimensions:      true    [1]
omp_parallelizeLoopOverFragments:       false   [1]
special_numOMP_z:                       1       [1]
domain_fragmentLength_x:                2       [2]
domain_fragmentLength_y:                2       [2]
domain_rect_numBlocks_x:                1       [2]
domain_rect_numBlocks_y:                1       [2]
l3tmp_smoother:                         "RBGS"  [2]
mpi_numThreads:                         1       [2]
opt_useAddressPrecalc:                  true    [2]
special_numNodes:                       2       [2]
special_numOMP_x:                       2       [2]
special_numOMP_y:                       2       [2]
special_ranksPerNode:                   4       [2]

['3D']
maxLevel:                               4       [2]
poly_tileSize_x:                        32      [2]

['ConstCoeff']
omp_parallelizeLoopOverDimensions:      true    [1]
domain_fragmentLength_x:                2       [2]
domain_fragmentLength_y:                2       [2]
domain_rect_numBlocks_x:                1       [2]
domain_rect_numBlocks_y:                1       [2]
l3tmp_smoother:                         "RBGS"  [2]
mpi_numThreads:                         1       [2]
opt_useAddressPrecalc:                  true    [2]
special_numNodes:                       2       [2]
special_numOMP_x:                       2       [2]
special_numOMP_y:                       2       [2]
special_ranksPerNode:                   4       [2]

['VarCoeff']

['xeon']
domain_rect_numFragsPerBlock_x:         1       [?]
```

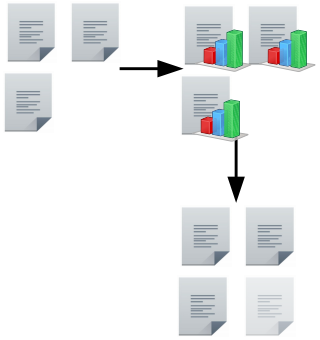

Abbildung 11: *schematischer Ablauf des Vorgangs*

Die Unterschiede zwischen 2D_ConstCoeff und 2D_VarCoeff gehen in 'ConstCoeff' und 'VarCoeff' ein. Hier fällt auf, dass bei konstantem Koeffizienten mehr Parameter als sehr relevant gelten, als bei variablem Koeffizienten und dass alle Zuordnungen, die bei variablem Koeffizienten getroffen werden, auch für das Problem mit konstantem Koeffizienten getroffen werden können. Somit gilt keiner der Parameter als typisch für 'VarCoeff'. Zwischen allen Problemen gibt es eine Gemeinsamkeit: `domain_rect_numFragsPerBlock_x: 1` wird in allen drei Fällen gefunden. Es gilt somit als typisch für alle Probleme. Da nur eine Hardwareplatform getestet wurde, wird diese Parameterbelegung in diesem Fall als Plattformabhängig angesehen (x86_64 Intel Xeon CPU mit 16 Prozessorkernen). Das '?' bei `domain_rect_numFragsPerBlock_x` gibt an, dass diese Belegung in den analysierten Fällen nicht in den gleichen oder benachbarten Klassen zu finden war. Sie ist somit weniger zuverlässig, als die anderen Belegungen.

Aus den so gewonnenen Informationen kann Wissen für das in diesem Fall nicht analysierte Problem generiert werden, indem die Zuweisungen von Parameterbelegungen den Aspekten zugeordnet werden und dann zum neuen Problem zusammengesetzt werden.

```
['3D', 'VarCoeff', 'xeon']
maxLevel:                          4       [2]
poly_tileSize_x:                   32      [2]
domain_rect_numFragsPerBlock_x:    1       [?]
```

Verglichen mit den Analyse von 3D_VarCoeff zeigt sich, dass das generierte Wissen eine Teilmenge des Wissens aus Optimierung und Analyse ist. Es entstehen hier also keine falschen Informationen, sondern nur Informationen, die die durchgeführte Optimierung tatsächlich produziert. Dieses Wissen kann also dazu verwendet werden, eine Optimierung zu beschleunigen, indem einige Parameter von vorne herein festgelegt werden. Es kann jedoch auch sein, dass mehr Informationen durch erzeugt werden, als eine tatsächliche Optimierung ergeben würde:

```
['3D', 'ConstCoeff', 'xeon']
maxLevel:                          4       [1]
poly_tileSize_x:                   32      [1]
omp_parallelizeLoopOverDimensions: true    [1]
domain_fragmentLength_x:           2       [2]
domain_fragmentLength_y:           2       [2]
domain_rect_numBlocks_x:           1       [2]
domain_rect_numBlocks_y:           1       [2]
l3tmp_smoother:                    "RBGS"  [2]
mpi_numThreads:                    1       [2]
opt_useAddressPrecalc:             true    [2]
sisc2015_numNodes:                 2       [2]
sisc2015_numOMP_x:                 2       [2]
sisc2015_numOMP_y:                 2       [2]
sisc2015_ranksPerNode:             4       [2]
domain_rect_numFragsPerBlock_x:    1       [?]
domain_rect_numFragsPerBlock_y:    1       [?]
domain_rect_numFragsPerBlock_z:    1       [?]
omp_parallelizeLoopOverFragments:  false   [?]
```

Hier entsteht mehr Wissen, als durch die Analyse in diesem Beispiel bestätigt werden kann. In diesem Fall muss für weitere Optimierungsvorgänge die Reihenfolge der Parameter beachtet werden. Wenn eine Optimierung mit eingeschränktem Suchraum vorgenommen werden soll, ist es ratsam, zunächst die Parameter mit hoher Priorität festzulegen, um nicht durch die Wahl der anderen Parameter möglicherweise eingeschränkt zu werden.

5.4 Fazit

Das implementierte Programm macht es möglich, aus den Ergebnissen vorhergehender Analysen Wissen für noch nicht analysierte Probleme zu erzeugen. Am Beispiel wurde gezeigt, dass das so erzeugte Wissen eine unterschiedliche Näherung zu dem darstellt, was eine zeitaufwändige Optimierung und Analyse ergeben würde. Die gefundenen Parameterbelegungen sind nach Relevanz für die Performanz des Zielprogramms sortiert, wobei weniger relevante Belegungen mit geringerer Sicherheit als optimal angesehen werden könnnen. Der Ansatz erweist sich als geeignet, wenn genug Probleme analysiert wurden, dass für jeden Aspekt eine typische Parameterbelegung ermittelt werden kann. Mit dem Ansatz können nicht nur Unterschiede zwischen den Problemen, sondern auch hardware- und architekturspezifische Unterschiede entdeckt werden, in dem eine identische Problemkonfiguration auf unterschiedlichen Architekturen analysiert wird.

Mit dem gefundenen Wissen können zukünftige Optimierungen verbessert werden, da durch Festlegung der gefundenen optimalen Parameter der Suchraum der gegebenen Probleme verkleinert wird.

6 Zusammenfassung

Es wurde ein genetischer Algorithmus vorgestellt, der für ein gegebenes Problem ein lokales Optimum finden kann. Wird dieser Algorithmus mehrfach auf das gleiche Problem angewandt, werden mehrere unterschiedlich gute lokale Optima gefunden und viele verschiedene Konfigurationen mit Laufzeitinformationen für das Problem erzeugt. Anschießen wurde ein Programm zur Analyse der erzeugten Konfigurationen vorgestellt. Das Ergebnis der Analyse ist eine Liste von von Parameterbelegungen, aus denen die Konfigurationen bestehen, die zu sehr guter Laufzeit führen. Diese Parameterbelegungen liegen in der Reihenfolge ihrer Relevanz vor. Es ist ersichtlich, welche Zuweisungen zu welchen Parametern den größten Einfluss auf die Performanz des Zielprogramms haben und welche eher im sehr späten Verlauf der Optimierung relevant wurden. Es wurde eine Schwelle gefunden, ab der eine Konfiguration als gut hinsichtlich der Performanz gelten kann und es wurden die Parameterbelegungen identifiziert, die zu guter Performanz führen. Als drittes wurden die Ergebnisse eines Programmes gezeigt, das die Reihenfolge der Parameterbelegungen, die aus der Analyse eines Problems hervorgehen, mit denen vergleicht, die aus der Analyse anderer Probleme stammen. Hierdurch ist es möglich Wissen über verwandte Probleme zu generieren, die bislang noch nicht optimiert und analysiert wurden. Somit wird eine spätere Optimierung dieser Probleme vereinfacht. In diesem Schritt wird schließlich Wissen über die Zusammenhänge zwischen einzelnen Aspekten der Probleme und Parameterbelegungen generiert, so dass im Nachhinein ein besseres Verständnis der analysierten Probleme für Menschen möglich ist.

Literatur

[1] https://de.mathworks.com/matlabcentral/mlc-downloads/downloads/submissions/14498/versions/1/screenshot.jpg. aufgerufen am 13.10.2016, 13:59.

[2] https://upload.wikimedia.org/wikipedia/commons/d/d9/pareto_distributionpdf.png. aufgerufen am 13.10.2016, 17:73.

[3] http://www.exastencils.org/de. aufgerufen am 2.8.2016, 16:43.

[4] Lorenz Haspel. Evolutionäre algorithmen. In *Seminar Machine Learning*. Friedrich-Alexander Universiät Erlangen-Nürnberg, 2013.

[5] Stefan Kronawitter and Christian Lengauer. Optimizations applied by the exastencils code generator. In *Technical Report MIP-1502*. Faculty of Computer Science and Mathematics, University of Passau, January 2015.

[6] Friedrich Mosler, Karl C. und Schmid. *Wahrscheinlichkeitsrechnung und schließende Statistik*. Berlin [u.a.], Springer, 2006.

[7] Rainer Schlittgen. *Einführung in die Statistik - Analyse und Modellierung von Daten*. München, Oldenbourg Wissenschaftsverlag GmbH, 2012.

[8] Christian Schmitt, Sebastian Kuckuk, Frank Hannig, Harald Köstler, and Jürgen Teich. Exaslang: A domain-specific language for highly scalable multigrid solvers. In *Proc. of the 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 42–51. IEEE Press, 2014.

[9] Norbert Siegmund und Alexunder Grebhahn und Sven Apel und Christian Kästner. Performance-influence models. In *Proceedings of Software Engineering – Fachtagung des GI-Fachbereichs Softwaretechnik, GI-Edition – Lecture Notes in Informatics*, page 29–31. Gesellschaft für Informatik, Februar 2016.