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

Lehrstuhl für Informatik 10 (Systemsimulation)

Codeoptimierung mit Expression Templates

Tobias Schleier

Bachelorarbeit

Codeoptimierung mit Expression Templates

Tobias Schleier

Bachelorarbeit

Aufgabensteller:	Prof. Dr. Christoph Pflaum
Betreuer:	Julian Hornich, M.Sc.
Bearbeitungszeitraum:	4.9.2017 – 4.2.2018

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 Bachelorarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 2. Februar 2018 .....

Der Quellcode der im Rahmen dieser Bachelorarbeit entstandenen Implementierungen liegt in Form einer CD bei und ist auf Anfrage am Lehrstuhl für Informatik 10 (Systemsimulation) der Friedrich–Alexander–Univerisät Erlangen–Nürnberg erhältlich.

Zusammenfassung

Naturwissenschaftliche Fachbereiche wie Chemie, Biologie und Physik sehen sich oft mit Problemstellungen konfrontiert, die sich in Form von partiellen Differenzialgleichungen formulieren lassen. Oftmals findet sich dann in der Analysis, einem Teilgebiet der Mathematik, eine Lösungsformel. Bei nicht trivialen Problemen ist die Formulierung einer analytischen Formel jedoch zu komplex. In einem solchen Fall knüpft das Teilgebiet der Numerik an, um Näherungslösungen zu bestimmen, die ein weiteres wissenschaftliches Verfolgen des Sachverhaltes ermöglichen. Somit finden sich Iterationsvorschriften, mit deren Hilfe die Lösung nachweislich schrittweise immer besser bis zur gewünschten Genauigkeit approximiert werden kann.

Implementierungen dieser iterativen Verfahren benötigen einen enormen Rechenaufwand, wodurch Optimierungen an Attraktivität gewinnen, und haben zugleich den Anspruch der numerischen Stabilität, weshalb Softwarefehler die Resultate gravierend verfälschen können.

Daher sind Bibliotheken für die Programmierung solcher Verfahren erwünscht. Die Technik der Expression Templates bietet die Möglichkeit Schnittstellen für diese zu entwerfen, sodass deren Benutzung der Formulierung mathematischer Formeln nahe kommt und die Laufzeiten in einem vergleichbaren Rahmen zu handoptimierten Implementierungen der Verfahren liegen.

Im Rahmen dieser Bachelorarbeit werden iterative Verfahren zur Lösung partieller Differenzialgleichungen mittels Expression Templates implementiert und deren Laufzeitverhalten mit alternativen Implementierungen verglichen. Dabei wurden insbesondere Messdaten der Laufzeiten, der Speicherzugriffe, des Cache–Verhaltens, sowie der Fließkommaoperationen und der Parallelisierbarkeit erhoben. Diese bestätigen die Hypothese, dass die Verwendung von Expression Templates nur marginal negative Auswirkungen auf die Performanz der numerischen Verfahren ausübt, dafür aber gute Eigenschaften in Hinsicht der Lesbarkeit und Schreibbarkeit aufweist.

Inhaltsverzeichnis

1 Motivation

Viele Probleme in den Fachbereichen der Chemie, Biologie und Physik stellen für die Wissenschaft eine Herausforderung dar, die sie in Form von partiellen Differentialgleichungen formulieren. Oft übersteigen die Anforderungen der Lösung den Bereich einzelner Disziplinen.

Daher kollaborieren sie mit Wissenschaftlern der Mathematik, um diese Probleme zu bewältigen. Dies ist in vielen Fällen ausreichend, um zu einem zufriedenstellenden Ergebnis zu gelangen. In relativ simplen Problemstellungen wie der Poissongleichung oder der Wärmeleitungsgleichung finden sich analytische Lösungen. Doch in manchen Fällen stößt die Mathematik an Grenzen, sodass bisweilen keine Lösung in analytischer Form gefunden werden kann.

Der Fachbereich der numerischen Mathematik kann jedoch in vielen Fällen zu einer Näherungslösung verhelfen. Erfordert das Problem nicht exakt gelöst zu werden, so kann diese Näherungslösung dem Anspruch genügen, der exakten Lösung um maximal einer angebbaren Toleranz abzuweichen.

Diese iterativen Näherungsverfahren gehen schrittweise vor, wobei die bisherige Lösung sich mit steigender Schrittzahl der exakten Lösung annähert. Dies wird als Konvergenz bezeichnet. Wurde der mathematische Beweis erbracht, dass ein Verfahren gegen die gesuchte Lösung konvergiert, stellt sich die Frage, wie schnell dies geschieht. Ein Maß hierfür ist die Konvergenzordnung, die eine obere Schranke für den verbleibenden Fehler darstellt. Es wird also der einzelne Schritt insoweit optimiert, dass der Fehler zur exakten Lösung nach diesem Schritt möglichst klein ist. Ein anderes und dazu gegenläufiges Ziel ist es, die Formeln dieser Verfahren in einem einfachen und realisierbarem Rahmen zu belassen um die Verwendbarkeit zu wahren.

Die Informatik greift diese Verfahren mit guter Konvergenzordnung auf und optimiert diese auf Rechengeschwindigkeit. Diese Symbiose ist unter dem Namen des Wissenschaftlichen Rechnens bekannt. Besonders im Dreidimensionalen ist der Anstieg an Rechenaufwand für die Verfahren enorm, da die Anzahl an Punkten in jede der drei Dimensionen und damit kubisch ansteigen kann. Selbst bei kleineren Lösungsgebieten sieht man sich schnell gezwungen, die Aktualisierung der Punkte zu parallelisieren, um das Ergebnis noch in einem praktikablen Zeitraum zu erhalten. Für große Gebiete ist daher der Einsatz von Hochleistungsrechnern unverzichtbar.

Der hohe Implementierungsaufwand ergibt sich aus der Tatsache, dass die Rechenvorschrift für einen Iterationsschritt sowohl von dem gewählten Verfahren, als auch von der zu lösenden Gleichung und dem Lösungsgebiet abhängt. Für jede Gleichung muss damit ein neuer Gleichungslöser programmiert oder eine vorhandene Implementierung im Kern modifiziert werden.

Dem wird vorgebeugt, indem das Verfahren in Form von Bibliotheken oder Frameworks bereitgestellt wird. Das sich hieraus ergebende neue Problem besteht jedoch in der korrekten Bedienung. Die Schnittstellen fordern oft, dass bestimmte Rahmenbedingungen erfüllt sind, und gestalten sich in ihrer Bedienung meist undurchsichtig. Mit manchen Programmierparadigmen ist dies jedoch nicht vermeidbar, ohne die Performanz der Verfahren zu beeinträchtigen.

Eine Möglichkeit für die Programmiersprache C++, dieses Problem zu lösen, besteht in der Verwendung von Expression Templates, einer Meta–Programmiertechnik. Sie ermöglichen die Implementierung des Problems in Form eines Ausdrucks, welcher mathematischer Notation sehr nahe kommt. Dieser Ausdruck kann daraufhin der Bibliothek übergeben werden oder mittels Zuweisungsoperator durch Überladen ganz in die C++–Syntax eingepflegt werden.Deshalb nutzen viele Meta–Bibliotheken wie *Blitz++* [Vel] oder ExPDE [Pfl01] Expression Templates. Diese Form von Schnittstelle bietet Benutzern den höchsten Komfort und gestaltet sich sehr intuitiv.

Im Rahmen dieser Arbeit soll nun erforscht werden, ob die Verwendung dieser Expression Templates einen negativen Einfluss in Bezug auf die Laufzeit darstellen. Hierfür wird eine Implementierung mittels Expression Templates mit einer von Hand optimierten Implementierung verglichen.

2 Grundlagen

2.1 Die Methode der Finiten Differenzen

Nun ist man mit der Herausforderung konfrontiert, die zu lösenden Gleichungen derart zu modellieren, dass sie auf diskrete Punkte eines Gitters angewandt werden können.

2.1.1 Herleitung der Differenzenquotienten

Leitet man die Formel des Vorwärtsdifferenzenquotienten durch Entwicklung einer Taylorreihe um den Punkt x_i her ehält man zugleich einen Fehlerterm für die Abschätzung der Güte der Approximation.[Stü05, S. 4-6]

h beschreibt dabei immer den Abstand zweier Punkte ($h = x_{i+1} - x_i$).

$$f(x) \approx T(x;a) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n \tag{1}$$

$$a \coloneqq x_i,\ x \coloneqq x_i + h \quad \rightarrow \quad (x-a)^n = h^n \tag{2}$$

$$\Rightarrow f(\underbrace{x_i + h}_{x_{i+1}}) = f(x_i) + hf'(x_i) + \frac{h^2}{2}f''(x_i) + \frac{h^3}{6}f'''(x_i) + \frac{h^4}{24}f^{(4)}(x_i) + \cdots \tag{3}$$

$$\frac{f(x_{i+1}) - f(x_i)}{h} = f'(x_i) \underbrace{+ \frac{h}{2}f''(x_i) + \frac{h^2}{6}f'''(x_i) + \frac{h^3}{24}f^{(4)}(x_i) + \cdots}_{\text{Fehlerterm in } \mathcal{O}(h)} \tag{4}$$

Die Herleitung des Rückwärtsdifferenzenquotienten und dessen Fehlerterm entstehen analog:

$$f(x) \approx T(x;a) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n \tag{5}$$

$$a \coloneqq x_i,\ x \coloneqq x_i - h \quad \rightarrow \quad (x-a)^n = (-h)^n \tag{6}$$

$$\Rightarrow f(\underbrace{x_i - h}_{x_{i-1}}) = f(x_i) - hf'(x_i) + \frac{h^2}{2}f''(x_i) - \frac{h^3}{6}f'''(x_i) + \frac{h^4}{24}f^{(4)}(x_i) + \cdots \tag{7}$$

$$\frac{f(x_{i-1}) - f(x_i)}{h} = -f'(x_i) + \frac{h}{2}f''(x_i) - \frac{h^2}{6}f'''(x_i) + \frac{h^3}{24}f^{(4)}(x_i) + \cdots \tag{8}$$

$$\frac{f(x_i) - f(x_{i-1})}{h} = f'(x_i) \underbrace{- \frac{h}{2}f''(x_i) + \frac{h^2}{6}f'''(x_i) - \frac{h^3}{24}f^{(4)}(x_i) + \cdots}_{\text{Fehlerterm in } \mathcal{O}(h)} \tag{9}$$

Kombiniert man nun die Gleichungen 3 und 7 erhält man die Formel sowie den Fehlerterm für zentrale Differenzenquotienten erster Ordnung:

$$f(x_i + h) - f(x_i - h) = 2hf'(x_i) + 2\frac{h^3}{6}f'''(x_i) + \cdots \tag{10}$$

$$\frac{f(x_i + h) - f(x_i - h)}{2h} = f'(x_i) \underbrace{+ \frac{h^2}{6}f'''(x_i) + \cdots}_{\text{Fehlerterm in } \mathcal{O}(h^2)} \tag{11}$$

Das Ergebnis der Herleitungen von Differenzenquotienten erster Ordnung:

Vorwärtsdifferenzenquotient:	$\frac{\partial u}{\partial x}(x,y,z) \approx \frac{u(x+h_x,y,z)-u(x,y,z)}{h_x}$	Fehlerterm in $\mathcal{O}(h_x)$
Rückwärtsdifferenzenquotient:	$\frac{\partial u}{\partial y}(x,y,z) \approx \frac{u(x,y,z)-u(x,y-h_y,z)}{h_y}$	Fehlerterm in $\mathcal{O}(h_y)$
zentraler Differenzenquotient:	$\frac{\partial u}{\partial z}(x,y,z) \approx \frac{u(x,y,z+h_z)-u(x,y,z-h_z)}{2h_z}$	Fehlerterm in $\mathcal{O}(h_z^2)$

Beim zentralen Differenzenquotienten liegt der Diskretisierungsfehler in $\mathcal{O}(h^2)$, womit der Fehler quadratisch abnimmt, unter der Annahme, dass die Gitterweite h kleiner als eins ist. Dies bedeutet, dass dieser Fehler mit feiner Gitterweite unterdrückt werden kann.

Abbildung 1 zeigt die graphische Darstellung dieses Prinzips. Lineare Funktionen werden von Finiten Differenzen erster Ordnung bereits exakt approximiert, da die Steigung einer Geraden unabhängig von Δx durch den Differenzenquotienten errechnet werden kann. Da es sich bei $f(x)$ um eine quadratische Funktion handelt entstehen für f aber bereits Approximationsfehler. Mit sinkender Distanz der Punkte $(h = x_{i+1} - x_i) \to 0$ nähert sich die Approximationen dem Differenzial ebenfalls an. Dies verdeutlicht, dass die Wahl der Gitterweite auch der Typ der zu approximierenden Funktion beachten muss.

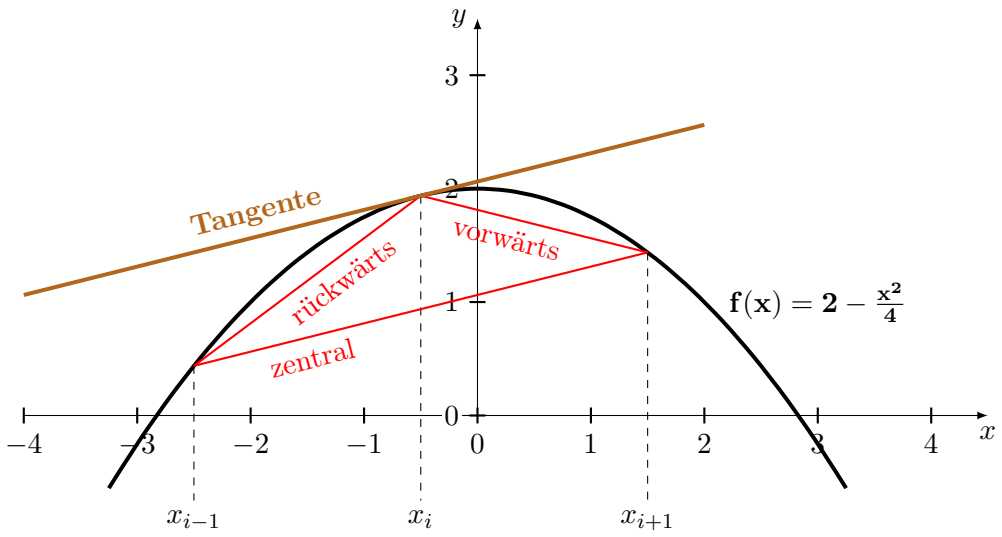


Abbildung 1: Grafische Darstellung der Differenzenquotiententen erster Ordnung

Durch analytische Differentiation erhält man durch Auswertung für den Punkt $x = -\frac{1}{2}$:

$$f'(x) = \left(2 - \frac{x^2}{4}\right)' = -\frac{x}{2} \Rightarrow f'\left(-\frac{1}{2}\right) = \frac{1}{4} \tag{12}$$

Dem Schaubild ist zu entnehmen, dass der zentrale Differenzenquotient als Einziger das Differenzial annähernd beschreibt.

Differenzenquotienten höherer Ordnung können durch Kaskadierung der Formeln erhalten werden. Der Fehlerterm entspricht den der verwendeten einfachen Quotienten. Um dies zu zeigen wird der zentrale Differenzenquotient zweiter Ordnung nun noch mittels Taylorreihenentwicklung gezeigt. Hierfür bedient man sich erneut der Gleichungen 3 und 7.

$$f(x_i + h) + f(x_i - h) = 2f(x_i) + h^2 f''(x) + \frac{h^4}{12} f^{(4)}(x) \cdots \tag{13}$$

$$f(x_i + h) - 2f(x_i) + f(x_i - h) = h^2 f''(x) + \frac{h^4}{12} f^{(4)}(x) \cdots \tag{14}$$

$$\frac{f(x_i + h) - 2f(x_i) + f(x_i - h)}{h^2} = f''(x) + \underbrace{\frac{h^2}{12} f^{(4)}(x) + \cdots}_{\text{Fehlerterm in } \mathcal{O}(h^2)} \tag{15}$$

Im mehrdimensionalen Fall hat die Formel des zentralen Differenzenquotienten zweiter Ordnung die folgende Form:

$$\frac{\partial^2 u}{\partial x^2}(x, y, z) \approx \frac{u(x - h_x, y, z) - 2u(x, y, z) + u(x + h_x, y, z)}{h_x^2}, \quad \text{Fehlerterm in } \mathcal{O}(h_x^2) \tag{16}$$

2.1.2 Das lineare Gleichungssystem

Diskretisiert man einen Differenzialoperator mittels Finiter Differenzen erhält man für jeden Punkt, an dem die Gleichung erfüllt sein soll, eine lineare Gleichung, die ihn repräsentiert. Dies führt für Gebiete mit mehreren Punkten zu einem linearen Gleichungssystem, dessen Lösungsvektor u die Punktwerte für die Lösung der partiellen Differenzialgleichung auf dem Gitter darstellen.

$$\underbrace{\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0m} \\ a_{10} & a_{11} & \cdots & a_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n0} & a_{n1} & \cdots & a_{nm} \end{bmatrix}}_{A} \cdot \underbrace{\begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_m \end{bmatrix}}_{u} = \underbrace{\begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_m \end{bmatrix}}_{f} \tag{17}$$

Diese ist jedoch sehr dünn besetzt, da im Verhältnis zur Anzahl an Punkten nur wenige Abhängigkeiten zu den Nachbarpunkten bestehen. Diese Abhängigkeiten zeichnen sich in der Systemmatrix A aufgrund der Homogenität des Gitters durch Einträge in Form von Diagonalen aus. Daher weisen auch alle Diagonalen mit Ausnahme der Hauptdiagonalen Nullelemente auf, da Punkte an Rändern des Gitters weniger Nachbarn besitzen.

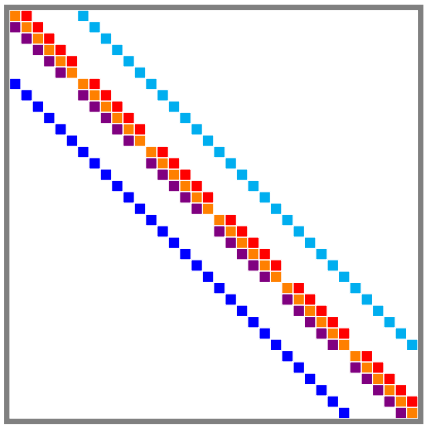


Abbildung 2: Form der Koeffizientenmatrix $A \in \mathbb{R}^{36\times 36}$ bei Diskretisierung des Laplace–Operators auf einem 6×6–Gitter in $\mathbb{R}^2$ zur Verdeutlichung der dünnen Besetzung. Die Quadrate repräsentieren Elemente, die von 0 verschieden sind.

Darum gibt man die Gewichtungen der Koeffizientenmatrix oft in Form eines Sternes an, der die mehrdimensionale Struktur des Gitters berücksichtigt und so die Nachbarschaftsbeziehungen der Punkte intuitiver veranschaulicht.

$$\frac{1}{h_2}\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{18}$$

2.2 Iterationsverfahren

Das Problem besteht jedoch in der Größe des linearen Gleichunssystems. Da die Gitterweite für den Nutzen der Lösung oft kleiner gewählt wird entsteht schnell eine große Anzahl an Unbekannten. Dadurch, dass direkte Löser eine höhere Komplexität aufweisen, ist es oft nicht ratsam, dieses lineare Gleichungssystem exakt zu lösen, zumal dort die Empfindlichkeit bezüglich Rundungsfehlern viel höher ist. Sind durch Rechengenauigkeit manche Unbekannte nicht direkt repräsentierbar, so entsteht eine Kaskade an Rundungsfehlern, die letztlich das Ergebnis verfälschen.

Eine Lösung bieten hier iterative Löser, die ausgehend von einem Startvektor sich der Lösung in Schritten annähern, bis ein Kriterium erfüllt ist und die Näherungslösung akzeptiert wird. Diese Kriterien gestalten sich oft in Form von Veränderungsmaßen zwischen zwei Iterationen, Abweichungen vom gewünschten und daher zwangsweise bereits bekannten Ergebnis, etwa für Messzwecke, oder einfacherweise einer Schrittanzahl. Abbildung 3 zeigt dies anhand einer Sinuskurve, die iterativ über mehrere Schritte angenähert wird.

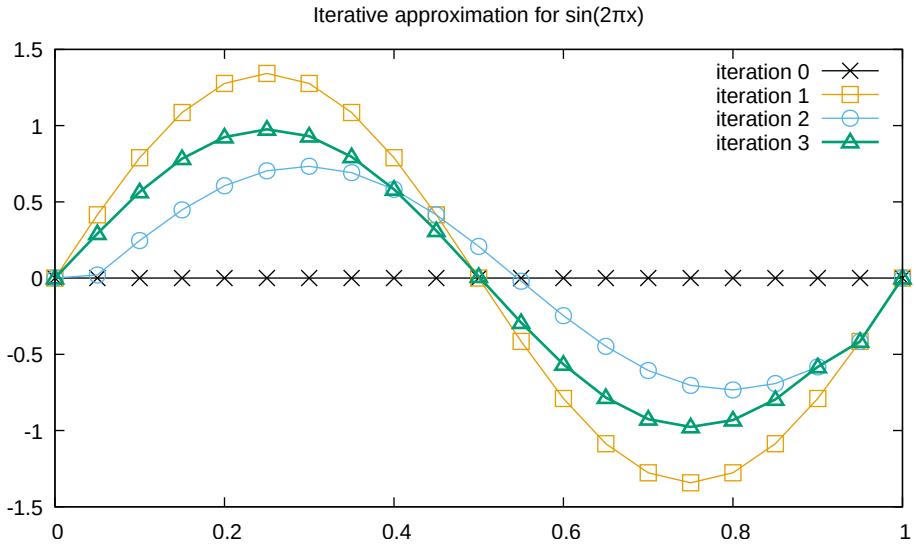


Abbildung 3: Veranschaulichung der Iterationen für die Approximation einer Sinuskurve

2.2.1 Jacobi–Verfahren

Das Jacobi–Verfahren oder Ganzschrittverfahren[BF, S. 450] löst das Lineare Gleichungssystem, indem es die Gleichungen nach der Variablen auf der Diagonalen umstellt und somit eine Aktualisierungsvorschrift findet, mit deren Hilfe die Unbekannten schrittweise zur Lösung konvergieren.

$$f_i = \sum_{j=0}^{n} a_{ij} \cdot u_j, \quad i = 1, \ldots, n \qquad \text{(Bedingung für Lösungsvektor } u\text{)} \tag{19}$$

$$= a_{ii} \cdot u_i + \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} \cdot u_j \tag{20}$$

$$u_i^{(m+1)} = \frac{1}{a_{ii}} \left(f_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} \cdot u_j^{(m)} \right), \quad i = 1, \ldots, n \tag{21}$$

Die Notation $u_i^{(m)}$ zeichnet den Wert des Vektors u an der Stelle i nach (m) Iterationsschritten aus und verdeutlicht damit, dass für die Berechnung der Werte im Schritt $(m+1)$ ausschließlich Werte des Schrittes (m) verwendet werden. Ebenfalls bemerkenswert ist die Division durch das Diagonalelement der Koeffizientenmatrix, das in dieser Formulierung von dem Wert null verschieden sein muss. Auf Konvergenzbedingungen soll an dieser Stelle lediglich verwiesen werden.[FP10, S. 112] Die ausschließlichen Abhängigkeiten zum vorherigen Iterationsschritt bedeuten für eine Implementierung, dass alte Werte, die noch benötigt werden, vorgehalten werden müssen, weshalb es sich oft durch einen erhöhten Speicheraufwand im Vergleich zu fortgeschritteneren Verfahren wie dem Gauß–Seidel–Verfahren auszeichnet.

2.2.2 Gauß–Seidel–Verfahren

Denn der geringere Speicheraufwand in Implementierungen des Gauß–Seidel–Verfahrens[BF, S. 454] gründet sich genau auf der Modifikation, nicht alle Werte der Unbekannten aus dem vorhergehenden Iterationsschritt zu entnehmen. Stattdessen fließen Unbekannte, deren Wert bereits bestimmt wurde, mit in die Gleichung ein.

$$u_i^{(m+1)} = \frac{1}{a_{ii}} \left(f_i - \sum_{j=1}^{i-1} a_{ij} \cdot u_i^{(m+1)} - \sum_{j=i+1}^{n} a_{ij} \cdot u_i^{(m)} \right), \quad i = 1, \ldots, n \tag{22}$$

Auch für dieses Verfahren gibt es Konvergenzuntersuchungen, die an dieser Stelle nicht näher beleuchtet werden.[FP10, S. 113] Es sei lediglich gesagt, dass im Falle der Konvergenz deren Ordnung mindestens der des Jacobi–Verfahrens entspricht. Für Implementierungen bietet dieses Verfahren den Vorteil, dass direkt auf den Lösungsvektor geschrieben werden kann und er somit in keiner Form dupliziert werden muss, was sich positiv auf Laufzeiten auswirkt. Vor allem bei Architekturen, deren Speicherzugriffszeiten den entscheidenden Faktor der Laufzeit bestimmen, stellt dies einen enormen Vorteil dar.

2.3 Programmiertechniken

Da nun die verwendeten numerischen Verfahren erläutert wurden soll über die grundlegenden Programmiertechniken, die bei der Implementierung dieser Verfahren zur Anwendung kamen, berichtet werden. Diese werden anhand kleiner Beispiele demonstriert oder graphisch verdeutlicht.

2.3.1 Überladen der Operatoren

Das Überladen von Operatoren in C++ bietet die Möglichkeit, die verwendeten Ausdrücke derart zu implementieren, sodass die Struktur der mathematischen Formeln zum Großteil erhalten bleibt. Ohne diese Semantik müssten alle Operationen durch Funktionsaufrufe modelliert werden, was legitim ist. Dennoch schränken kaskadierte Funktionsaufrufe die Lesbarkeit ein. Ein Beispiel hierfür ist der folgende Ausdruck:

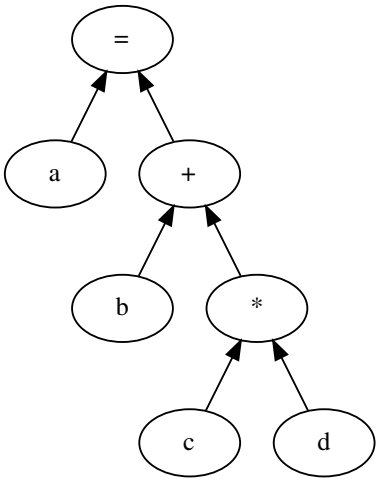


Abbildung 4: Beispiel eines Ausdruckbaumes

Dieser hat in C++ Syntax die folgende Form:

Listing 1: Ausdruckbaum aus Abbildung 4 in C++–Syntax

```
a = b + c * d;
```

```
// equivalent to
operator=(a, operator+(b,operator*(c,d)));
```

Da diese Operatoren jedoch von vielen unterschiedlichen Ausdrücken implementiert werden, dort jedoch die gleiche Signatur haben können, kann es vorkommen, dass diese vom Übersetzer nicht devirtualisiert werden, was eine längere Laufzeit zur Folge hat. Da diese Funktionalität aber die grundlegende zu lösende Gleichung beschreibt kommt sie im Kern des Programmes zur Anwendung, weshalb im Falle des mathematischen Lösers die Devirtualisierung essenziell für das Erreichen der gewünschten Performanz ist. Wie Devirtualisierung erzwungen werden kann behandelt der Abschnitt ‚Erzwungene Inline-Ersetzung'.

2.3.2 Curiously recurring template pattern

Die Basis der Parameterübergabe bildet die Expression–Klasse, die das curiously recurring template pattern[Här07, S. 13] implementiert. Alle Klassen, die von ‚Expression' erben,

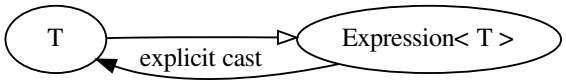


Abbildung 5: Graphische Darstellung des Curiously recurring template pattern

besitzen einen Operator zur Typumwandlung. Dadurch ist es der Klasse nicht nur möglich zur Expression upgecastet zu werden. Somit ist es beispielsweise möglich den Zuweisungsoperator mit einer Expression als Parameter zu überladen, sodass dieser Operator alle Klassen akzeptiert, die von der Expression–Klasse erben.

```
1  template <class E>
2  Grid& Grid::operator=(const Expression<E>& we) {
3      // unwrap expression
4      const E& e = E(we);
5      // update Gridpoints
6      for (int k = 0; k < size(2); ++k)
7          for (int j = 0; j < size(1); ++j)
8              for (int i = 0; i < size(0); ++i)
9                  (*this)(i,j,k) = e(i,j,k);
10 }
```

Listing 2: Beschreiben der Punkte eines Gitters mit einer Expression

Listing 2 zeigt die Konvertierung von Expression<E> zu E. Der Template–Mechanismus erzeugt den operator=() für die übergebene Expression, wodurch das Gitter durch eine Zuweisung eines Ausdrucks beschrieben werden kann. Da der Operator speziell für diesen Ausdruck erzeugt wird erkennt der Übersetzer, falls der verwendete Zugriffsoperator für den Ausdruck nicht definiert wurde. Somit bliebt dieses Vorgehen typsicher.

2.3.3 Traits

Wichtig für Expression–Templates sind sogenannte Traits.[Här07, S. 12] Diese Datenstrukturen erhalten durch über den Template–Mechanismus definierte Bedingungen gewisse Eigenschaften, die es ermöglichen, dass eine Entscheidung bereits zur Übersetzungszeit erzwungen wird und daher nicht mehr in den Programmablauf verlagert werden kann. Dies ermöglicht es von Typinformationen abhängige Bedingungen auszuwerten, und diese für Entscheidungsfindungen des Übersetzers zu verwenden. Ihre Verwendung und der damit verbundene Nutzen der Traits wird im Abschnitt „Substitution failure is not an error" erläutert.

```
1  // is_expression::value determines if T is derived from Expression.
2  template <typename T>
3  struct is_expression
4      : public std::integral_constant < bool ,
5                  std::is_base_of< Expression<T> , T >::value >
6  {};
```

Listing 3: Trait, dessen boolsche Membervariable value anzeigt, ob der Template–Parameter T von der Expression–Klasse erbt.

Listing 3 zeigt die Konstruktion eines neuen Traits aus bereits definierten Traits, die durch den Header <type_traits> ab C++11 eingebunden werden können. Falls dies unerwünscht ist, bietet die Bibliothek boost [Boo] eine Alternative, deren Lizenz es erlaubt ihren Quellcode als Grundlage für eine eigene Implementierung der Traits zu nutzen [BoL].

```
1  //enable_expression_type::type only exists if type E is an expression.
2  template <typename T, typename E>
3  struct enable_expression_type
4      : public std::enable_if<is_expression<E>::value, T >
5  {};
```

Listing 4: Trait, dessen Membervariable type vom Typ T nur dann existiert, wenn E eine Expression ist.

Der in Listing 3 definierte Trait kommt nun in Listing 4 zur Anwendung, wodurch eine Membervariable des Traits nur dann deklariert ist, falls der übergebene Typ T von Expression erbt. Dieser Trait–Mechanismus kann über diese Beispiele hinaus erweitert werden, um gewünschte Bedingungen an Deklarationen zu knüpfen.

Weil dieses Vorgehen es ermöglicht, Berechnungen in den Übersetzungsvorgang zu verschieben, kann hierdurch die Übersetzungsdauer enorm beeinträchtigt werden. Wird diese Technik zu massiv beansprucht entartet der Übersetzer selbst zum Löser und das eigentliche Programm dient letztlich nur zur Ausgabe der errechneten Resultate. Daher sollte man Traits konservativ nur für Konzepte verwenden, die Resultate von Berechnungen während der Übersetzung erfordern.

2.3.4 Substitution failure is not an error

Eine der verwendeten Techniken für die Implementierung der Binären Operatoren ist „Substitution failure is not an error“ (SFINAE). Hierbei wird die Tatsache ausgenutzt, dass der Übersetzer keine Fehler erzeugt, falls die Substitution von Template–Parametern fehlschlägt. Stattdessen wird diese Deklaration verworfen.

Auf diese Weise können mehrere Implementierungen mit gleicher Signatur bereitgestellt werden, sofern die Template–Parameter derart strukturiert sind, dass jeweils nur eine gültige Implementierung nach der Substitution erhalten bleibt. In der Implementierung kann so beispielsweise sichergestellt werden, dass nicht versucht wird, Methoden einer Expression bei primitiven Datentypen aufzurufen.

```
1  // Expression + Expression
2  template <typename EL, typename ER>
3  auto operator[]( int i ) const
4  -> typename enable_et_types_fit<EL,ER,true,true,double>::type
5  {
6      return m_l[i] + m_r[i];
7  }
8  // Expression + constant
9  template <typename EL, typename CR>
10 auto operator[]( int i ) const
11 -> typename enable_et_types_fit<EL,CR,true,false,double>::type
12 {
13     return m_l[i] + m_r;
```

```
14 }
15 // constant + Expression
16 template <typename CL, typename ER>
17 auto operator[]( int i ) const
18 -> typename enable_et_types_fit<CL,ER, false ,true ,double >::type
19 {
20     return m_l + m_r[i];
21 }
```

Listing 5: SFINAE zur Vermeidung des Zugriffsoperators auf Konstanten bei Addition.

Listing 5 beschreibt die Verwendung mehrerer Definitionen für den operator[] einer Addition. Der Trait enable_et_types_fit besitzt den Member type vom Typ double nur, falls die Auswertung von enable_expression_type der beiden ersten Template–Parametern den Wahrheitswerten den darauf folgenden beiden Template–Parametern entspricht. Da sich die Bedingungen gegenseitig ausschließen kommt es nicht zu vielfachen Definitionen mit gleicher Signatur. Der vierte und fehlende Fall zweier Konstanten wird nicht implementiert, da diese vom Übersetzer vorher gefaltet werden können.

3 Problemstellung

Das Ziel war es eine Bibliothek bereitzustellen, die mit möglichst benutzerfreundlicher Schnittstelle partielle Differenzialgleichungen performant löst. Das Hauptaugenmerk lag hierbei auf der Programmlaufzeit, diese hängt jedoch von der Zahl der aktualisierten Punkte pro Zeiteinheit ab. Ein generischer Löser von partiellen Differenzialgleichungen ist jedoch nur bedingt performant realisierbar, da die Voraussetzungen vieler Optimierungen wie beispielsweise das Speicherlayout nicht sichergestellt werden können. Um die Laufzeit der Implementierung zu steigern wurden so Einschränkungen eingeführt, die die Art der Gleichung sowie das Lösungsgebiet betreffen.

3.1 Quaderförmiges Lösungsgebiet im dreidimensionalen Raum

Um die Aktualisierungsvorschrift homogener zu gestalten wird ein quaderförmiges Lösungsgebiet angenommen.

$$\Omega \coloneqq \left\{(x,y,z) \in \mathbb{R}^3 \;\middle|\; n_{min} \leq n \leq n_{max},\ n \in \{x,y,z\}\right\} \tag{23}$$

Zunächst wird die Anzahl an Punkten in jeder Dimension (m,n,p) festgelegt. Daraus lässt sich aufgrund der Äquidistanz bereits die Gitterweite berechnen.

$$(h_x,h_y,h_z) \coloneqq \left(\frac{x_{max}-x_{min}}{m-1}, \frac{y_{max}-y_{min}}{n-1}, \frac{z_{max}-z_{min}}{p-1}\right) \tag{24}$$

Die Subtraktion eines Punktes ist nötig, da m lineare Punkte im Raum $m-1$ Intervalle zwischen ihnen eingrenzen, in die das Gebiet aufgeteilt wird. Anders ausgedrückt entsteht erst durch die Positionsbestimmung des zweiten Punktes das erste Intervall.

Um dieses zu modellieren wird es nun diskretisiert, also auf Punkte beschränkt, die Vielfache der Gitterweite h in den jeweiligen Dimensionen repräsentieren:

$$\mathbb{G} \coloneqq \{(i,j,k) \in \mathbb{Z} \mid \quad (x_{min}+ih_x,\ y_{min}+jh_y,\ z_{min}+kh_z) =: (x_i,y_j,z_k) \in \Omega\} \tag{25}$$

Die Information der Gitterweite muss bei der praktischen Anwendung nicht dem Gitter selbst bekannt sein, zumal diese voneinander unabhängig sind. Lediglich die verwendeten Sterne enthalten meist in die Gleichung eingehende Distanzen zwischen Punkten. Von jedem inneren Punkt wird jedoch angenommen, dass er genau 26 benachbarte Punkte hat. Ist dies im zu lösenden Problem nicht der Fall muss diese Situation durch geeignete nicht vorhandene Nachbarschaften modeliert werden, die dann jedoch mit der Gewichtung null in die Gleichung eingehen.

Unter der Annahme, dass ein Eckpunkt des Gitters im Ursprung zu finden ist, $x_{min} = y_{min} = z_{min} = 0$ ergibt sich das folgende Beispiel:

$$i=0, j=0, k=0 \quad \Rightarrow \quad (x_i,y_j,z_k) = (0,0,0) \quad \Rightarrow \quad (i,j,k) \in \mathbb{G} \tag{26}$$

$$i=1, j=0, k=2 \quad \Rightarrow \quad (x_i,y_j,z_k) = (h_x,0,2h_z) \quad \Rightarrow \quad (i,j,k) \in \mathbb{G} \tag{27}$$

$$\left(\frac{h_x}{2}, 3.2 \cdot h_y, 5\right) \Rightarrow \quad (i,j,k) = (\frac{1}{2}, \frac{3.2}{h_y}, \frac{5}{h_z}) \notin \mathbb{Z}^3 \quad \Rightarrow \quad (i,j,k) \notin \mathbb{G} \tag{28}$$

Bisher stillschweigend angenommen wurde, dass die Kanten des Quaders parallel zu den Koordinatenachsen verlaufen. Weicht das Gebiet von dieser Beschreibung ab, ist es nötig, dieses durch Transformationen auf die geforderte Form zu bringen. Ist dies nicht möglich muss das Problem mittels eines anderen Ansatzes gelöst werden.

3.2 Partielle Differenzialgleichungen

Nun wird erneut die Sternform der Iterationvorschrift aus Gleichung 17 aufgegriffen. Eine weitere Forderung lautet, dass die partielle Differenzialgleichung für jeden Punkt in Form eines 27–Punkt Sternes angebbar sein muss. Dies ermöglicht gewichtete Abhängigkeiten

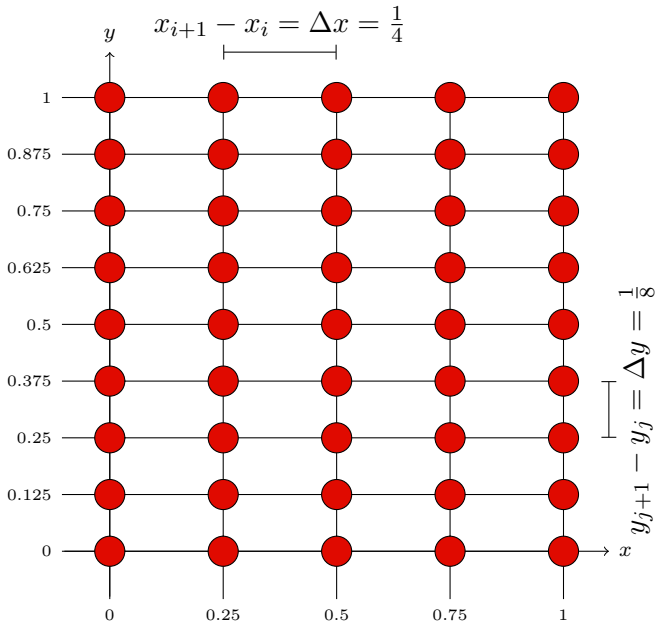


Abbildung 6: Beispiel eines Gitters in $\mathbb{R}^2$

direkter Nachbarn einschließlich diagonaler Nachbarn. Für Diskretisierungen mittels Finiter Differenzen genügen oft direkte Nachbarn, wird jedoch eine Diskretisierung mittels Finiter Elemente gewählt tritt diese Form des 27–Punkt Sternes auf. Auf die Testfälle, die diese Sterne erzeugen, wird im Abschnitt Testfälle genauer eingegangen. Ist die partielle Differenzialgleichung nicht in dieser Sternform angebbar oder werden Konvergenzbedingungen hierfür erfüllt, muss auf die Verwendung dieser Implementierung ebenfalls verzichtet werden. Sollten nicht alle 27 Punkte benötigt werden, können diejenigen ohne Beitrag jedoch problemlos mit null belegt werden.

4 Implementierung

4.1 Struktur der Expression Templates

Die Implementierung der Ausdrucksbäume erfolgte durch Ausdrucksschablonen Diese ermöglichen es, Mechanismen zu etablieren, die in der Lage sind, Typsicherheit zu gewährleisten.

Die verwendeten Datenstrukturen müssen von der *Expression*–Klasse erben und somit die Eigenschaften besitzen, selbst Ausdrücke zu sein. Durch die Möglichkeit, die Expression–Klasse als Wrapper anzusehen und diesen durch Konvertieren zu entpacken, sind einheitliche Schnittstellen möglich. [Vel95] Durch Traits kann eine Template–Implementierung außerdem insoweit eingeschränkt werden, dass sie nur mit Ausdrücken kompatibel ist, die über die nötigen Methoden verfügen. Hierdurch wird die Typsicherheit gewährleistet. Die nun folgenden Abschnitte werden beispielhaft an dem Listing 6 veranschaulicht.

```
a = W(b + c) / E( 3 * d );
```

Listing 6: Beispiel zur Veranschaulichung von Expression Templates, in Quellcodeform

Die Verschiebungsoperatoren W() und E() manipuliren den Index des Zugriffs x zu $x + 1$ beziehungsweise $x - 1$. Die Variablen a bis d können sowohl als primitive Datentypen, als auch als komplexe Datenstrukturen mit überladenen Operatoren verstanden werden. Der Ausdrucksbaum bleibt von diesen Interpretationen vorerst unberührt und kann bereits angegeben werden:

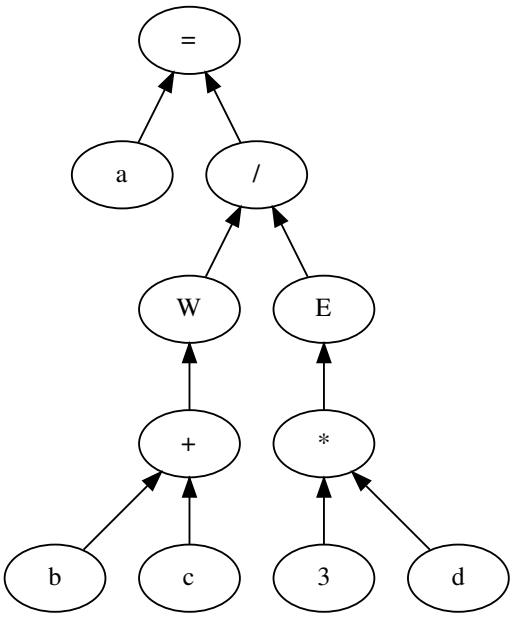


Abbildung 7: Ausdrucksbaum des Listings 6

4.1.1 Gemeinsame Methoden

Damit ein Ausdruck, der von der Expression–Klasse erbt, die selbst keine Methoden bereitstellt, für die Erstellung eines Ausdrucksbaumes verwendet werden kann, muss er folgende Methoden definieren:

- Eindimensionaler Zugriff
- Dreidimensionaler Zugriff

- Größe der Speicherstruktur
- Maximaler und minimaler Index (lesend)

Dies gilt beispielsweise auch für alle Knoten des Ausdrucksbaumes aus Abbildung 7 die keine Konstanten repräsentieren, einschließlich den Variablen a bis d, den Verschiebungsoperatoren und den arithmetischen Operationen. Ausgenommen ist der Wurzelknoten, der immer eine Zuweisung repräsentiert, da er als Ausdruck in der Implementierung nicht vorhanden ist. Stattdessen wird von allen Ausdrücken, die Speicher verwalten, gefordert, diesen Operator zu implementieren. Weitere Besonderheiten dieser Knoten sind im Abschnitt Speicherrepräsentanten zu lesen. Da der Operator jedoch einen dieser speicherverwaltenden Knoten als Rückgabetyp definiert, die wiederum diese Methoden implementieren, wird er in den Abbildungen als vollwertiger Ausdrucksknoten dargestellt. Unter den Voraussetzungen, dass der Rückgabetyp des Zugriffsoperators mit dem Typ der zugewiesenen Variable übereinstimmt und der Index gültig ist, beschreibt Listing 7 eine syntaktisch korrekte Zuweisung.

```
int i = ( W( b + c ) / E( 3 * d ) )[15];
```

Listing 7: Beispiel für direkten Zugriff auf einen Ausdruck an Index 15

Er ist äquivalent zu:

```
int i = (b[14] + c[14]) / (3 * d[16]) ;
```

Die notwendige Unterscheidung zwischen Größe und erlaubten Indices ergibt sich aus der Tatsache, dass der Ausdruck auch Indexverschiebungen enthalten kann, die den erlaubten Bereich verschieben oder sogar verkleinern können, falls mehrere Verschiebungen in entgegengesetzten Richtungen im selben Ausdruck Verwendung finden. Im Beispielausdruck von Listing 7 ist der Zugriff auf den Index $i = 0$ nicht gültig, da dieser durch den W()–Operator auf $W(0) = -1$ verschoben wird. Daher wurden gesonderte Größenangaben eingeführt, die es ermöglichen, das Intervall der gültigen Indices in Erfahrung zu bringen.

4.1.2 Speicherrepräsentanten

Da ein Zugriff auf einen Operationsknoten im Ausdrucksbaum diesen Zugriff auf die darunter liegenden Operanden nur weiterreicht und die Ergebnisse modifiziert, endet diese Kaskade an den Blattknoten. Diese können entweder aus Konstanten oder aus Ausdrücken, die Speicher belegen, bestehen. Daher müssen Knoten, die zu dieser Art von Ausdrücken gezählt werden, Speicher allokieren, während die inneren Knoten lediglich Referenzen verwalten. Diese Referenzen werden im späteren Verlauf der Übersetzung herausoptimiert, sodass nur die reinen Speicherzugriffe auf die Speicherstruktur der Blattknoten verbleiben. An dieser Stelle sei betont, dass der Zweck der Expression Templates darin liegt, diesen Ausdrucksbaum aufzustellen, diesen dann aber durch Substitution zu einer Liste umzuwandeln, wodurch weitere Optimierungen möglich werden. Bleibt der Ausdrucksbaum erhalten, weil beim Übersetzungsvorgang Instanzvariablen nicht entfernt werden können, so ist der einzige Effekt der Expression Templates Zusatzaufwand für das Programm zur Laufzeit. Innere Knoten dürfen also keinesfalls diese Eigenschaften aufweisen, Speicher zu belegen.

Des Weiteren müssen Speicherrepräsentanten neben den lesenden Zugriffoperatoren auch den schreibenden Zugriffsoperator bereitstellen, um Modifikationen zu erlauben. Innere Knoten hingegen finden nur lesende Verwendung. Von Vorteil, jedoch nicht zwingend notwendig, ist die vorhandene Implementierung des Zuweisungsoperators, der für alle gültig lesbaren Indices des Parameterausdrucks die Werte des Speicherrepräsentanten aktualisiert. Aus dieser Designentscheidung folgt ebenfalls, dass bei der Anwendung eines 27–Punkte Sterns die äußeren Ränder des Gebietes nicht beschrieben werden.

Point3D. Die Point3D–Klasse dient zur Repräsentation eines Punktes im dreidimensionalen Raum. Die Koordinaten werden in einem statischen Feld der Größe drei gespeichert. Daneben bietet sie einen eindimensionalen Zugriffsoperator, Vergleichsoperatoren, sowie

Methoden, um Summe und Produkt der Koordinatenwerte zu errechnen. Sie erbt von der Ausdruck–Klasse, um Summen oder Vielfache eines Punktes ohne weiteren Implementierungsaufwand errechnen zu können.

Array3D hingegen repräsentiert den dreidimensionalen Raum in Quaderform. Sie implementiert alle Methoden und Operatoren der Ausdrücke. Der benötigte Speicher wird in Form eines dynamisch allokierten Feldes realisiert. Hierdurch werden Programmabläufe auf kleinen Feldern verlangsamt, der Zusatzaufwand sinkt im Verhältnis zur Berechnung mit steigender Gittergröße, bis er nach kurzer Zeit vernachlässigbar ist, da es sich um einmalige Allokation handelt.

Stencil27. Eine Mischung aus Point3D und Array3D stellt Stencil27 dar. Zu Anfang als Array3D mit fester Größe implementiert zeigten sich nach einiger Zeit zyklische Abhängigkeiten, woraufhin es als eigene Ausdrucksklasse realisiert wurde. Daher konnte nun auch der benötigte Speicher in Form eines statischen Feldes der Größe 27 umgesetzt werden und der Zugriff mittels Verschiebungsoperatoren auf zentrierte und damit intuitivere Indices erweitert werden.

4.1.3 Bedienbarkeit und Benutzerfreundlichkeit

Um die Benutzerfreundlichkeit von Expression Templates zu verdeutlichen werden Implementierungen der Relaxation eines Fünf–Punkte Sterns aus der Diskretisierung des Laplace–Operators in $\mathbb{R}^2$ an Codebeispielen demonstriert.

```
u = ( S(u)*S(s) + W(u)*W(s) + E(u)*E(s) + N(u)*N(s) ) / s[0];
```

Listing 8: Relaxation eines Fünfpunktsterns mit Expression Templates, die ermöglichen die Mitte des Sterns durch Verschiebungen mit dem Index 0 zu belegen

Die Implementierung in C++11 ohne Expression Templates im Vergleich:

```
1  for (int k = 0; k < u.size(2); ++k)
2      for (int j = 0; j < u.size(1); ++j)
3          for (int i = 0; i < u.size(0); ++i) {
4              u(i,j,k) = (   u(i  ,j-1) * s( 1, 0 )
5                           + u(i-1,j  ) * s( 0, 1 )
6                           + u(i+1,j  ) * s( 2, 1 )
7                           + u(i  ,j+1) * s( 1, 2 )
8                         ) / s(1,1,1);
9          }
```

Listing 9: Relaxation eines Fünfpunktsterns ohne Expression Templates

Anzumerken ist hier, dass es ohne Expression Templates nur schwerlich möglich ist, die Schleifen von Listing 9 in die Zuweisung zu verlegen, da dies die Erzeugung eines zweiten Gitters erfordert. Das Problem, dass für jede Operation ein temporäres Gitter erzeugt wurde, welches nach seiner Verwendung wieder verworfen wird, wurde von C++ mittels der Move–Semantik gelöst. Dies erlaubt, mit einem temporären Objekt auszukommen. In dieser Hinsicht ist der Vorteil der Expression Templates gegenüber herkömmlichen Implementierungen in C++ geschwunden, wobei dieses eine temporäre Objekt noch eingespart wird. Außerdem erweisen sich Indexverschiebungen der Gitterpunkte in Kombination mit konstanten Indices des Sterns als unleserlich, was Listing 9 ebenfalls verdeutlicht.

4.2 Operatoren

Bisher wurden Operatoren nur in Hinsicht der Ausdrucksbäume betrachtet, ohne näher auf ihre Implementierung einzugehen. Um Ausdrucksklassen auf intuitive Weise verwenden zu können werden Operatoren definiert, die als Wrapper–Klasse agieren und den Zugriff entsprechend verändern.

4.2.1 Unäre Operatoren

Die Negation modifiziert den Elementzugriff derart, dass das von dem Zugriff zurückgegebene Element noch zusätzlich negiert wird. Um dies generisch zu gestalten wird der unäre Operator operator-() auf das Resultat angewandt. Somit erhalten alle Ausdrücke eine Überladung des operator-(), wobei dessen Definition in einem Ausdruck gekapselt ist. Die Übersetzung schlägt fehl, falls der Rückgabetyp des Zugriffsoperators den operator-() nicht definiert.

Die Shift–Operation verändert die für den Zugriff verwendeten Indizes um die als Template–Argumente ganzzahlig angegebenen Verschiebungen.
Durch die Design–Entscheidung, dass diese Verschiebungen als Template–Parameter übergeben werden, müssen diese zur Laufzeit bekannt sein, was jedoch für die wenigsten Anwendungsfälle eine Einschränkung darstellen sollte. Hierdurch kann eine dreidimensionale Verschiebung bei konstanter Gittergröße in eine konstante linearisierte Verschiebung vereinfacht werden, wodurch die Laufzeit drastisch sinkt. Desweiteren wurden Generatorfunktionen definiert, um einfache Verschiebungen zu realisieren. Beispielsweise kann der Zugriff mittels W–Wrapper so modifiziert werden, dass statt der angegebenen x-Koordinate $(x - 1)$ verwendet wird.

Dies verändert jedoch den gültigen Wertebereich der Indices. Verwendet man gegensätzliche Verschiebungen in einer Gleichung, so verkleinert sich der Wertebereich. Hierdurch kann der Rand des Gitters nicht mehr beschrieben werden, weshalb dieser bei einer Iteration des Lösers unverändert bleibt. Wird dieser jedoch benötigt muss derzeit eine Lösung mittels Ghost–Layern gefunden werden.

Der Laplace–Operator kann nun mittels dieser Verschiebungen implementiert werden. Da er jedoch für lineare Gleichungslöser verwendet wird trägt die Diagonale des LGS nichts zu dem Ergebnis der Operation bei. Dieser Diagonaleintrag kann mittels eines eigenen unären Operators in die Gleichung eingehen, sofern dieser für das Verfahren erforderlich ist. Die Herleitung der Faktoren des Sternes wird im Abschnitt der Laplace–Gleichung näher beleuchtet.

Listing 10 zeigt die Implementierung des Zugriffs. Das hier verwendete Trait reduce_type wird verwendet, um den Datentyp der Rückgabe zu bestimmen, indem es den Ausdruck T entpackt, bis es zu einem Typ gelangt, das kein Ausdruck ist. Die Bestimmung des Rückgabetyps von binären Operatoren wird im Abschnitt Binäre Operatoren beschrieben.

```
1  template <typename T>
2  inline auto operator[]( int i ) consta
3  -> typename reduce_type<T>::type
4  {
5      return -(  ( W(u) + E(u) ) / (hx*hx)
6               + ( S(u) + N(u) ) / (hy*hy)
7               + ( D(u) + U(u) ) / (hz*hz)
8              )[i];
9
10 }
```

Listing 10: Implementierung des Zugriffoperators für den Laplace–Operator mit Expression Templates

```
1
2  /*
3   * Obtain the reduced type from an arithmetic or expression type.
4   */
5
6  // void form (no type).
7  template <typename T, class enable = void >
```

```
 8  struct reduce_type
 9  {};
10
11  // Expression form (entry type).
12  template <typename T>
13  struct reduce_type< T,
14      typename std::enable_if<
15          is_expression< T >::value
16      >::type
17  >
18  {
19      using type = typename T::entry_type;
20  };
21
22  // Non-Expression form (non-reduced T).
23  template <typename T>
24  struct reduce_type< T,
25      typename std::enable_if<
26          !(is_expression< T >::value)
27      >::type
28  >
29  {
30      using type = T;
31  };
```

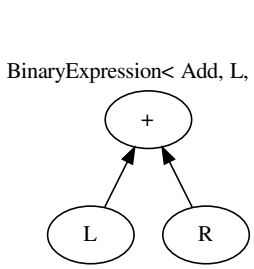
Listing 11: Definition des Traits reduce_type zur Identifikation des Datentyps der Einträge im darunterliegenden Speicherfeld

4.2.2 Binäre Operatoren

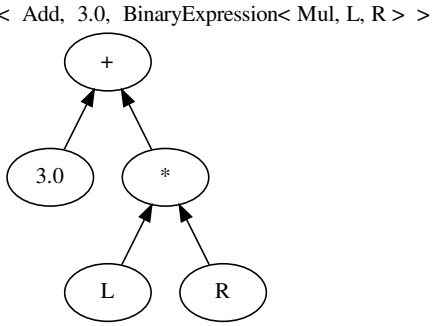
Der binäre Operator wurde in allgemeiner Form implementiert. Der erste Template–Parameter besteht aus einer Struktur, die den eigentlichen Operator über eine Funktion *apply(L l,R r)* bereitstellt. Diese Argumente l und r sind die verbleibenden zwei der drei Template–Argumente. Sie können aus Konstanten bestehen oder aus weiteren Ausdrücken mit Zugriffsfunktion, wodurch Ausdrucksbäume entstehen.

```
1   //Sum of l and r.
2   struct Addition {
3       template <class L, class R>
4       static inline auto apply(const L& l, const R& r)
5       -> decltype( l + r )
6       {
7           return l + r;
8       }
9   };
10
11  //operator+ for Expressions.
12  template <class A, class B>
13  inline
14  BinaryOperator< Addition, A, B >
15  operator+(const A& a, const B& b)
16  {
17      return BinaryOperator<Addition,A,B>(a,b);
18  }
```

Listing 12: Die Addition–Klasse zur Nutzung als Template–Parameter der BinaryExpression–Klasse, sowie die allgemeine Überladung des operator+() für Ausdrücke



(a) Binäre Addition



(b) Ausdrucksbaum

Abbildung 8: Binäre Ausdrücke als Baum und als Expression Templates

Die binären Operatoren sind in vielen Hinsichten Kernpunkte der Implementierung. Zunächst sollte die Frage des Rückgabetyps gelöst werden. Die Lösung liegt im Trait namens get_binop_result_type, das auf dem in Listing 11 definierten reduce_type aufbaut und in Listing 13 beschrieben wird.

```
1  // Get type of result of binary operator.  Uses reduce_type.
2  template <class Op, typename L, typename R>
3  struct get_binop_result_type
4  {
5      private:
6          // reduced types of arguments
7          using RL = typename reduce_type<L>::type;
8          using RR = typename reduce_type<R>::type;
9      public:
10         // the result type
11         using type = decltype( Op::apply(RL(),RR()) );
12 };
```

Listing 13: Traits zur Bestimmung des Rückgabetyps einer binären Operation

4.2.3 Stern–Operatoren

Zur Lösung einer allgemeinen partiellen Differenzialgleichung ist es erforderlich, dass diese dem iterativen Löser in Form eines 27–Punkte Sterns übergeben wird.

Relaxation. Für diese Schnittstellen wurde der 27–Punkte Stern als Klasse Stencil27 definiert und Schnittstellen bereitgestellt. Seine Einträge beschreiben die Vorfaktoren, die bei der Applikation des Sternes auf einen Punkt im Gitter zur Anwendung kommen.

```
1  // linear coordinate access
2  template <typename T>
3  inline T StencilRelaxation<T>::operator[]( int ii ) const
4  {
5      return ( shift<-1,-1,-1>(u)[ii] * stencil(-1,-1,-1)
6             + shift< 0,-1,-1>(u)[ii] * stencil( 0,-1,-1)
7             + shift< 1,-1,-1>(u)[ii] * stencil( 1,-1,-1)
8             + shift<-1, 0,-1>(u)[ii] * stencil(-1, 0,-1)
9             + shift< 0, 0,-1>(u)[ii] * stencil( 0, 0,-1)
10            + shift< 1, 0,-1>(u)[ii] * stencil( 1, 0,-1)
11            + shift<-1, 1,-1>(u)[ii] * stencil(-1, 1,-1)
12            + shift< 0, 1,-1>(u)[ii] * stencil( 0, 1,-1)
13            + shift< 1, 1,-1>(u)[ii] * stencil( 1, 1,-1)
14            //
```

```
15              + shift<-1,-1, 0>(u)[ii] * stencil(-1,-1, 0)
16              + shift< 0,-1, 0>(u)[ii] * stencil( 0,-1, 0)
17              + shift< 1,-1, 0>(u)[ii] * stencil( 1,-1, 0)
18              + shift<-1, 0, 0>(u)[ii] * stencil(-1, 0, 0)
19              // NOT (!) mid point
20              + shift< 1, 0, 0>(u)[ii] * stencil( 1, 0, 0)
21              + shift<-1, 1, 0>(u)[ii] * stencil(-1, 1, 0)
22              + shift< 0, 1, 0>(u)[ii] * stencil( 0, 1, 0)
23              + shift< 1, 1, 0>(u)[ii] * stencil( 1, 1, 0)
24              //
25              + shift<-1,-1, 1>(u)[ii] * stencil(-1,-1, 1)
26              + shift< 0,-1, 1>(u)[ii] * stencil( 0,-1, 1)
27              + shift< 1,-1, 1>(u)[ii] * stencil( 1,-1, 1)
28              + shift<-1, 0, 1>(u)[ii] * stencil(-1, 0, 1)
29              + shift< 0, 0, 1>(u)[ii] * stencil( 0, 0, 1)
30              + shift< 1, 0, 1>(u)[ii] * stencil( 1, 0, 1)
31              + shift<-1, 1, 1>(u)[ii] * stencil(-1, 1, 1)
32              + shift< 0, 1, 1>(u)[ii] * stencil( 0, 1, 1)
33              + shift< 1, 1, 1>(u)[ii] * stencil( 1, 1, 1)
34              ) / stencil[0] ;
35  }
```

Listing 14: Relaxation eines 27–Punkte Sterns

Hier wurde keine Verwendung des reduce_type Traits gemacht, da kein sinnbehafteter Problemfall gefunden wurde, bei dem diese ohnehin schon schwergewichtige Operation auf einen komplexeren Ausdruck ausgeführt wird. Eine bessere Herangehensweise ist es, das Ergebnis zwischenzuspeichern, um vielfachen Zugriff, der zu vielfacher Auswertung führt, zu vermeiden. Sollte es notwendig sein, diese Expression in einem Ausdrucksbaum zu verwenden, kann die Implementierung ohne viel mehraufwand analog zu anderen Expressions erweitert werden.

Lokale Relaxation. Die Lokale Relaxation geht analog zur Relaxation vor, mit dem Unterschied, dass der entsprechende Stern erst aus einem Array3D ausgelesen werden muss, welches der Operation als Argument übergeben wird. Hierdurch wird ermöglicht jedem Gitterpunkt einen individuellen Stern zuzuweisen, sodass auch inhomogene Verhalte berücksichtigt werden. Die in Listing 14 dargestellte Operation wird hierfür um einen Zugriff erweitert, der den entsprechenden Stern aus einem Array3D< Stencil27<T> > ausliest. Dennoch bedeutet dies einen erhöhten Speicherbedarf, wodurch sich zur Laufzeit andere Bedingungen an die Archtektur ergeben können.

5 Optimierungen

Um den Performanzgewinn der implementierten Laufzeitverbesserungen vergleichbar zu gestalten wurde deren Nutzung steuerbar mithilfe von Übersetzer–Flags konzipiert. Die Optimierungen behandelten die Anpassung der Algorithmen an parallele Ausführung wie das Rot–Schwarz Gauß–Seidel–Verfahren, Veränderungen der Reihenfolgen von Speicherzugriffen durch Loop unrolling oder Cache Blocking und eindimensionale Indizierung, um die Rechenkapazitäten für die Berechnung des Ergebnisses voll ausschöpfen zu können, ohne dass diese in einem unnötigen Maß für die Ermittlung der Speicheraddressen von Operatoren beansprucht wird.

5.1 Loop unrolling

Bei einer Schleife muss nach jeder Iteration geprüft werden, ob die Schleifenbedingung noch erfüllt ist, um herauszufinden, an welcher Stelle die Befehlsausführung fortgesetzt werden soll. Dieser Zusatzaufwand kann vermindert werden, wenn bekannt ist, dass die Anzahl der Schleifendurchläufe eine Vielzahl einer Ganzzahl i darstellt.[Här07, S. 17] Denn in diesem Fall kann durch Substitution jeweils i Durchläufe zu einem Verschmolzen werden. In vielen Fällen kann und sollte diese Optimierung vom Übersetzer durchgeführt werden, da gebräuchliche Übersetzer architekturspezifische Eigenschaften optimal ausnutzen können und der Programmcode portabel bleibt. Dennoch ist zu prüfen, ob dies im vorliegenden Anwendungsfall zutrifft. Denn diese Maßnahme ermöglicht es dem Übersetzer Vektoroperationen anzuwenden, ohne über die Grenzen der Schleifeniterationen hinweg optimieren zu müssen. Falls die Prozessorarchitektur Vektorinstruktionen bereitstellt bieten diese für numerische Löser einen großen Vorteil und sollten genutzt werden.

```
1     int array[256];
2     for (int i = 0; i < 16; ++i) {
3         array[i] = i;
4     }
5     // is equivalent to
6     int array[256];
7     for (int i = 0; i < 16; i += 4) {
8         array[i] = i;
9         array[i+1] = i+1;
10        array[i+2] = i+2;
11        array[i+3] = i+3;
12    }
```

Listing 15: Beispiel für Loop unrolling

5.2 Cache Blocking

Caches

Da der Zugriff auf Hauptspeicher im Vergleich zur Zykluszeit eines Prozessors mindestens um den Faktor 30 langsamer ist wurden Caches, zu deutsch ‚Verstecke' entwickelt.[Pfl, S. 8] Diese Speicher stellen eine schnellere Form des Zugriffs bereit, sind jedoch kostenaufwendiger in der Herstellung. Daher wird dieses Konzept kaskadiert angewandt. Mit jeder weiteren Stufe wird der Cache schneller, jedoch kleiner hinsichtlich der Speichergröße. Daraus resultiert eine Hardwarearchitektur innerhalb des Prozessors, der sich gut in Form eines Dreieckes beschreiben lässt. Die gängigen Prozessorarchitekturen beinhalten meist drei Stufen der Caches. Die zugrunde liegenden Prinzipien sind sowohl die zeitliche, als auch räumliche Lokalität.

Die zeitliche Lokalität beschreibt den Sachverhalt, dass bei einem Zugriff auf eine Speicheradresse genau auf Dieselbe mit hoher Wahrscheinlichkeit in naher Zukunft ein

erneuter Zugriff erfolgt. Ein triviales Beispiel hierfür ist die Zählschleife. Die Zählvariable wird nach jeder Iteration erneut benötigt. Ist es nicht möglich, diese Zählvariable in einem Register vorzuhalten, muss sie im Hauptspeicher abgelegt werden. Ohne die Funktionalität des Caches, der es ermöglicht, dieses Datum dennoch schnell zugreifbar vorzuhalten, bedeutet dies massive Performanzeinbußen, deren Faktor 30 nicht selten übersteigt.

Die räumliche Lokalität dagegen beschreibt die Tatsache, dass bei einem Zugriff auf eine Speicheradresse deren Nachbaradressen mit hoher Wahrscheinlichkeit ebenfalls besucht werden. Dies geschieht vorrangig bei der Iteration über Felder. Durch das Laden von *Cache–Blöcken* wird dieser enorme Aufwand, das Datum aus dem Hauptspeicher zu laden, zumindest für die restlichen Daten, die im selben Cache–Block angesiedelt sind, umgangen. Wenn auch hier in ungünstigen Datenstrukturen Einbußen hinsichtlich der Laufzeit auftreten können, so ist dies doch eine gewinnbringende sowie praktikable Maßnahme.

Da die Zugriffe auf Caches in SRAM Technologie realisiert sind bieten sie in Bezug auf die Zugriffszeit einen so immensen Vorteil gegenüber dem Hauptspeicher, welcher mittels DRAM implementiert ist, sodass ohne Veränderung des Instruktionsstromes bereits Performanzgewinn verzeichnet werden kann. Möchte man jedoch das gesamte Potenzial der Architektur nutzen, so sollten für diese angepasste Algorithmen und die zugehörigen Datenstrukturen genutzt werden. Diese werden im Kapitel Optimierungen im Detail erläutert.

Nicht näher eingegangen wird an dieser Stelle auf die unterschiedlichen Satzaufteilungen, die ein Cache implementieren kann.[Pfl, S. 9] Weil dies jedoch von der Hardwarearchitektur festgelegt wird und somit nicht zu den beeinflussbaren Faktoren zählt, wird die Thematik an dieser Stelle nicht genauer behandelt.

Oft verschlechtern auch Caching-Effekte die Performanz von Numerischen Lösern spürbar, indem später noch benötigte Daten aus dem Cache verdrängt werden, nur um später erneut vom Hauptspeicher geladen zu werden. Da die räumliche Lokalität durch das Lösungsgebiet und die zu lösende Differenzialgleichung weitestgehend invariant ist verbleibt die zeitliche Lokalität und damit der Lösungsalgorithmus als Raum für Optimierungen.

Man zerteilt, sofern anwendbar, das Gebiet in Blöcke, sodass die Blöcke der Operanden erst dann aus dem Cache verdrängt werden, wenn sie nicht mehr im Verlauf dieser Operation verwendet werden. Dabei ist die Blockgröße derart zu wählen, dass alle Blöcke, die an der Aktualisierung des Ergebnisblocks beteiligt sind, gleichzeitig in einer Stufe des Caches vorgehalten werden können. So können die Werte des Gebiets blockweise berechnet werden und Speicherzugriffe zu einem großen Prozentsatz vom Hauptspeicher auf den schnelleren Cache verlagert werden. Oft wird hierfür der Level2–Cache verwendet, da die Blockgröße in der obersten Stufe zu klein gewählt werden müsste.[Stü05, S. 56]

5.3 Erzwungene Inline-Ersetzung

Eine Besonderheit des ‚inline' Schlüsselwortes in C++ ist, dass es für den Übersetzer nicht verpflichtend ist. Der Übersetzer darf dieses Schlüsselwort ignorieren oder diese Eigenschaft selbst Funktionen zusprechen, sofern diese gekapselt sind und er es als vorteilhaft betrachtet. Diese Entscheidung wird oft aufgrund von Heuristiken der Codegröße getroffen und ist im durchschnittlichen Anwendungsfall eine gute Wahl. Ist die Inline–Substitution eines Funktionsaufrufes ausdrücklich erwünscht, um funktionsübergreifende Optimierungen zu ermöglichen, so muss dies über das Setzen bestimmter Attribute der Funktionen erfolgen. Auf diese Weise kann ebenso erzwungen werden, dass überladene Operatoren nicht als ausführbare Funktion im Programm erstellt werden, was das Problem der benötigten Devirtualisierung löst. Den Nachteil dieser Optimierung stellen längere Übersetzungszeiten und größere Programm–Binärdateien dar.

Unglücklicherweise handhaben Übersetzer diese Attribute nicht einheitlich, weshalb diese Optimierung zu übersetzerspezifischem Code führt. In der vorliegenden Implementierung wurde daher versucht, dieses Manko durch Makro–Substitution für die gängigsten Übersetzer zu umgehen. Wird erkannt, dass ein Übersetzer verwendet wird, der diese Form der Attributierung unterstützt, wird das Makro ‚INLINE' zu diesem substituiert. Sollte

der Übersetzer eine solche Semantik nicht bereitstellen wird auf das im C++–Standard spezifizierte Schlüsselwort ‚inline' zurückgegriffen.

In allen vorhergehenden Codebeispielen wurde dieses Makro durch dieses Schlüsselwort ‚inline' ersetzt, um Verwirrung vorzubeugen.

5.4 Linearisierte Indizierung

Dieser nun erzwungenermaßen direkt aufeinander folgende Code birgt jedoch noch eine weitere Hürde. Durch die Anwendung der gleichen Operation auf die Gitterpunkte entsteht ein Muster, sodass die Offsets der Indices für die jeweiligen Zugriffe konstant sind. Der Übersetzer kann dies bei dreidimensionaler Indizierung jedoch nur in trivialen Fällen erkennen und erzeugt somit zur Laufzeit viel Mehraufwand. Mit eindimensinalen Indices, für die diese Offsets bei konstanter Gittergröße explizit angebbar sind, können diese nun leichter erkennbaren Verschiebungen schneller identifiziert und weiter optimiert werden. Das Gitter zur Repräsentation des dreidimensionalen quaderförmigen Lösungsgebiets wird in Form eines kompakten Feldes abgespeichert. Dies erfordert jedoch, dass die Berechnungsvorschrift für diesen Index im eindimensionalen aus den drei 3D–Indices Bijektivität aufweisen muss. Daher sollte ein Inkrementieren in einer höheren Dimension einer Addition der Größen aller darunterliegenden Dimensionen entsprechen. Die folgende Berechnungsvorschrift stellt dies sicher:

$$i_{1D} = i + j \cdot s_i + k \cdot s_i \cdot s_j, \quad s_n : \text{ Größe des Gitters in Dimension } n \tag{29}$$

Dies erleichtert nicht nur common–subexpression–elimination, sondern ermöglicht dem Übersetzer außerdem auch feinere Unterschiede zu verzeichnen, welche Parameter als konstant betrachtet werden können. Da diese sich oft auf die Größe des Gitters bezieht bietet sich hier ein großes Optimierungspotential.

Für den Fall, dass der eindimensionale Zugriff nicht in dieser Form optimiert werden kann, etwa weil die Gittergrößen nicht als konstant betrachtet werden können, wird dieser vom Übersetzer zumeist noch durch Anwendung des Horner–Schemas optimiert:

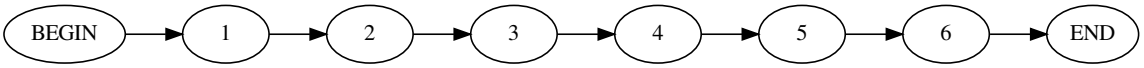
$$i + j \cdot s_i + k \cdot s_i \cdot s_j = i + s_i \cdot (j + (k \cdot s_j)) \tag{30}$$

Viele Prozessoren stellen MulAdd–Instruktionen bereit, durch die diese Form von Ausdruck weniger Taktzyklen benötigt und der Zusatzaufwand der Indexberechnung sinkt. Ist diese Linearisierung also problembedingt nicht möglich, gibt es andere Ansätze, um die Laufzeit zu verkürzen. Die Linearisierung ist jedoch das Mittel der Wahl.

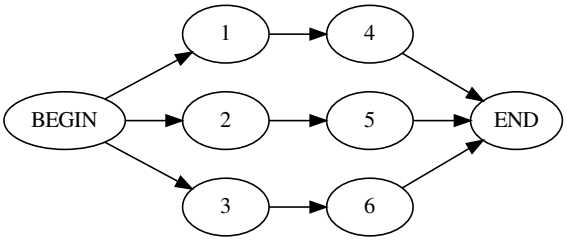
5.5 Parallelisierung auf Thread–Ebene

Da die Zielarchitekturen oftmals mehrere Rechenkerne zur Verfügung stellen ist es wünschenswert, diese Rechenleistung auch für die Berechnung des Ergebnisses auszuschöpfen, zumal die Anwendung ein hohes Maß an Parallelität vorweist. Hierzu wird der Prozess auf Thread–Ebene parallelisiert. Das Lösungsgebiet wird aufgeteilt, sodass der Rechenaufwand der Teile möglichst gleich ist. Dann kann jeweils ein Ausführungsstrang den ihm zugeteilten Teil des Lösungsgebiets aktualisieren. Abbildung 9 zeigt eine Schleife, die durch OpenMP parallelisiert wurde. Weil ihre Iterationen echt parallel ablaufen können ist die Reihenfolge der Ziffern 0 bis 9 in der Ausgabe nicht vorhersagbar.

Dabei ist es wichtig, sicherzustellen, dass sich die Fäden nach einer Iteration synchronisieren, um die Datenabhängigkeiten zu berücksichtigen. Daher ist es für eine Verbesserung der Performanz wichtig, dass die Rechenlast auf den Fäden möglichst gleichverteilt ist. Im Falle der iterativen Löser ist der Rechenaufwand für die Aktualisierung eines Punktes unabhängig von dessen Lage im Raum oder Gitter. Ein intuitiver Ansatz ist daher, das Gebiet in Intervalle gleicher Größe aufzuteilen und jedem Faden einen solchen Bereich des Lösungsgebietes zuzuteilen. Dieses Vorgehen ist beim Jacobi–Verfahren sinnvoll und führt zu dem gewünschten Ergebnis. Im Falle des Gauß–Seidel–Verfahrens führt es aber zu einer Verlangsamung im Vergleich zur sequentiellen Laufzeit, da die Formulierung der Berechnung eine strenge Sequentialität erfordert. Infolge dessen laufen die Fäden



(a) Sequentielle Abarbeitung der Instruktionen



(b) Parallele Abarbeitung der Instruktionen in drei Fäden

Abbildung 9: Veranschaulichung der Vorteile echter Parallelität auf die Laufzeit

nicht parallel, sondern sequentiell nacheinander und erzeugen somit keine Parallelität zur Laufzeit, nehmen aber zusätzlichen Rechenaufwand für ihre Erzeugung, Zerstörung und Synchronisation in Anspruch.

5.5.1 Rot–Schwarz Gauß–Seidel–Verfahren

Um die Fäden auch echt parallel ausführbar zu gestalten wird die Formel für das Gauß–Seidel–Verfahren in der folgenden Art modifiziert. Diese Modifikation des Gauß–Seidel–Verfahrens besteht im „Einfärben“ der Punkte des Lösungsgebietes, sodass die Berechnungsvorschrift einen Punktes des Gitters von keinem anderen Punkt derselben Farbe abhängt.[Stü05, S. 7] Als Folge daraus kann für einen Iterationsschritt jeweils der Unterraum, den eine Farbe repräsentiert, echt parallel berechnet werden. Die Farb–Unterräume selbst müssen jedoch sequentiell aktualisiert werden, um die Datenabhängigkeiten zu wahren. Dadurch ist es nötig für jede Farbe über das Gitter zu iterieren, um einen Schritt des Verfahrens zu berechnen.

Der Namenszusatz Rot–Schwarz gründet sich auf der Erkenntnis, dass im zweidimensionalen für die Anwendung eines Fünf-Punkte-Sternes zwei Farben ausreichend sind. Abbildung 10 veranschaulicht dies.

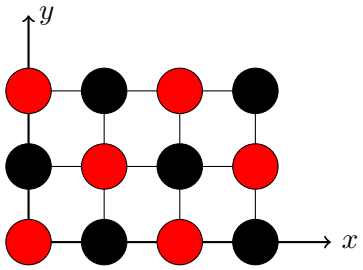


Abbildung 10: Rot–Schwarz Farbunterräume eines Gitters in $\mathbb{R}^2$

Resultiert im zweidimensionalen Raum eine diskrete Approximation der partiellen Differenzialgleichung zu einem 9–Punkte Stern, so dürfen ebenfalls diagonale Nachbarpunkte nicht die gleiche Farbe aufweisen. Somit werden bereits vier Farben benötigt, was beispielhaft in Abbildung 11 dargestellt wurde.

Da die Implementierung jedoch 27–Punkte Sterne auf dreidimensionalen Gittern unterstützen soll, ist es nötig das Gebiet in acht Farben einzufärben.

In vielen Fällen ist abzuwägen, ob diese Einbußen benötigt werden, da in der Praxis auch ohne Beachtung dieser Abhängigkeiten brauchbare Ergebnisse in viel kürzerer Zeit erzielt werden können. Aus akademischer Sicht ist dies jedoch nötig, um den Beweis der

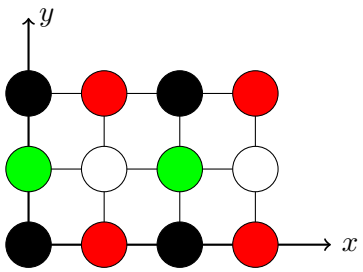


Abbildung 11: Vier Farbunterräume eines Gitters in $\mathbb{R}^2$ für Neun-Punkte-Sterne

Konvergenz des Verfahrens erbringen zu können und damit die Schätzbarkeit der Qualität des Ergebnisses zu gewährleisten.

Dies und eine intuitivere Aufteilung der Programmfäden erzielt man ebenso mittels des Jacobi–Verfahrens, dessen Abhängigkeiten von Beginn an festlegen, dass hierauf nicht geachtet werden muss. Jedoch zu dem Preis, dass das Jacobi–Verfahren den doppelten Speicherbedarf besitzt. Dieser ist nicht nur kritisch in Hinsicht der Realisierbarkeit für große Gitter, sondern zieht damit verbundene Effekte in Bezug auf Caches und Speicherbandbreite mit sich. Aus diesen Gründen wurde das Rot–Schwarz Gauß–Seidel–Verfahren mittels acht Farben implementiert, selbst wenn hierdurch ein Teil des Laufzeitgewinnes reinvestiert werden muss, um die enorme Speicherplatzersparnis zu nutzen.

Nach der Festlegung des Verfahrens trotz seiner Komplexität bezüglich der Parallelisierung stellt sich die Frage der Realisierung.

5.5.2 OpenMP

Diese Aufteilung in Fäden geschieht nicht durch händische Implementierung. Dies wäre nicht nur aufwendig, sondern ebenso fehleranfällig. Die Implementierung der Erstellung, Verwaltung und Synchronisation der Fäden ist komplex und oft unübersichtlich, die Fehlersuche sehr zeitinteniv, zumal die Testbarkeit durch echte Parallelität eingeschränkt wird. Außerdem stellt es eine weitere Herausforderung dar, den Performanzgewinn zu maximieren. Denn um Parallelität zu ermöglichen bedarf es zusätzlichen Verwaltungsaufwand zur Laufzeit, sodass erst nach dessen Kompensation eine Verkürzung der benötigten Rechenzeit zu verzeichnen ist. Abhilfe schafft OpenMP [CJP08], eine API, die mithilfe von Übersetzer–Direktiven alle Aufgaben, die eine Multithread–Implementierung zu verwalten hat, abstrahiert und nutzerfreundlich programmierbar anbietet. Um beispielsweise die Initialisierungsschleife aus Listing 15 zu parallelisieren genüg es, zwischen den Zeilen 6 und 7 ein Pragma, eine Steueranweisung für den Übersetzer einfügen, damit die Iterationen dieser Schleife parallel laufen. Dieses hat die folgende Form:

```
#pragma omp parallel for
```

Während es aber mittels OpenMP ein Leichtes ist, Parallelität zu erzeugen, erfordert es für komplexere Anwendungen oder Speicherstrukturen fundiertes Hintergrundwissen und fortgeschrittenere Pragmas, um das volle Potenzial der Nebenläufigkeit auszuschöpfen.

6 Testfälle

6.1 Laplace–Gleichung

Die Eignung der Laplace–Gleichung besteht vor allem in der Eigenschaft, dass analytische Lösungen gefunden werden können. Dies erlaubt eine Bewertung der Ausgabe und damit der Identifikation falscher Berechnungen. Ein weiteres Kriterium, das für diese Gleichung anzuführen ist, stellt ihre verhältnismäßig geringe Komplexität dar. Dies erleichtert die Fehlersuche im Falle unkorrekter Ergebnisse. Die Gleichung gestaltet sich wie folgt:

$$\Delta u = 0 \tag{31}$$

Δu stellt hierbei den Laplace-Operator dar.

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \tag{32}$$

Durch eine Diskretisierung mittels der Finite Differenzen Methode , die in Abschnitt 2.1.1 hergeleitet wurde, erhält man für partielle Ableitungen zweiter Ordnung in $\mathbb{R}^3$:

$$\frac{\partial^2 u}{\partial x^2}(x,y,z) = \frac{u(x-h_x,y,z) - u(x,y,z) + u(x+h_x,y,z)}{h_x^2}, \text{ mit } \quad h_x \text{ Gitterweite in Dimension x} \tag{33}$$

Daraus ergibt sich die Diskretisierung der Laplace–Gleichung mittels Finiter Differenzen.[Stü05, S. 6]

$$\begin{aligned} \Delta u(x,y,z) = \quad & \frac{u(x-h_x,y,z) - 2\cdot u(x,y,z) + u(x+h_x,y,z)}{h_x^2} \\ & + \frac{u(x,y-h_y,z) - 2\cdot u(x,y,z) + u(x,y+h_y,z)}{h_y^2} \\ & + \frac{u(x,y,z-h_z) - 2\cdot u(x,y,z) + u(x,y,z+h_z)}{h_z^2} \end{aligned} \tag{34}$$

Und damit den folgenden 27–Punkte Stern:

$$z=-1 \begin{bmatrix} & \frac{1}{h_z^2} & \\ & & \end{bmatrix}, \qquad z=0: \begin{bmatrix} & \frac{1}{h_y^2} & \\ \frac{1}{h_x^2} & -\left(\frac{2}{h_x^2}+\frac{2}{h_y^2}+\frac{2}{h_z^2}\right) & \frac{1}{h_x^2} \\ & \frac{1}{h_y^2} & \end{bmatrix}, \qquad z=1: \begin{bmatrix} & \frac{1}{h_z^2} & \\ & & \end{bmatrix} \tag{35}$$

Nullwerte sind hierbei nicht visualisiert.

6.2 Voller 27–Punkt Stern

Der verwendete Stern, um die Korrektheit bei Sternen mit 27 Punkten zu testen, hatte folgende Form:

$$z=-1: \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \qquad z=0: \begin{bmatrix} 1 & 1 & 1 \\ 1 & -26 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \qquad z=1: \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{36}$$

Dieser Stern ist zwar trivial, jedoch erfüllt er die nötigen Voraussetzungen, um Performanzmessungen der Bibliothek durchzuführen. Im Gegensatz zu den Sternen, die sich aus Finiten Differenzen ergeben, nutzt er alle 27 Einträge.

Alle Einträge erwiesenermaßen zu nutzen ist für die Testbarkeit der Implementierung und damit die Fehlerbereinigung unerlässlich.

7 Verwendete Rechnersysteme

Die folgenden Systeme wurden zur Implementierung sowie für die Erfassung der Messergebnisse verwendet:

Heim–PC

- Intel(R) Core(TM) i7-6700HQ CPU; 2.60GHz
- 8 GB Hauptspeicher (7.22 GiB usable)
- Linux 4.12.12-gentoo #6 SMP x86_64a GNU/Linux
- g++ (Gentoo 5.4.0-r3 p1.4, pie-0.6.5) 5.4.0

Rechnerräume am Lehrstuhl für Informatik 10 (Systemsimulation) FAU

- Intel(R) Xeon(R) CPU E3-1275 v5; 3.60GHz
- 64 GB Hauptspeicher (62.8 GiB)
- Linux 4.4.0-98-generic #121-Ubuntu SMP GNU/Linux
- g++ (Ubuntu 5.4.0-6ubuntu1 16.04.5) 5.4.0 20160609

CIP-Pool Informatik FAU

- Intel(R) Xeon(R) CPU E5620; 2.40GHz
- 24 GB Hauptspeicher (23.5GiB)
- Linux 4.9.78-1-cip-amd64 #1 SMP x86_64 GNU/Linux
- g++ (Debian 6.3.0-18) 6.3.0 20170516

8 Resultate

8.1 Vergleich der Unoptimierten Implementierungen

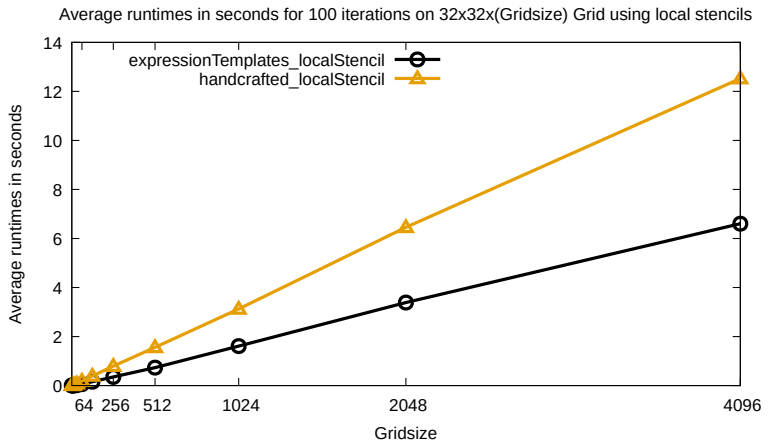


Abbildung 12: Durchschnittliche Laufzeiten mit lokalen Sternen ohne Optimierungen

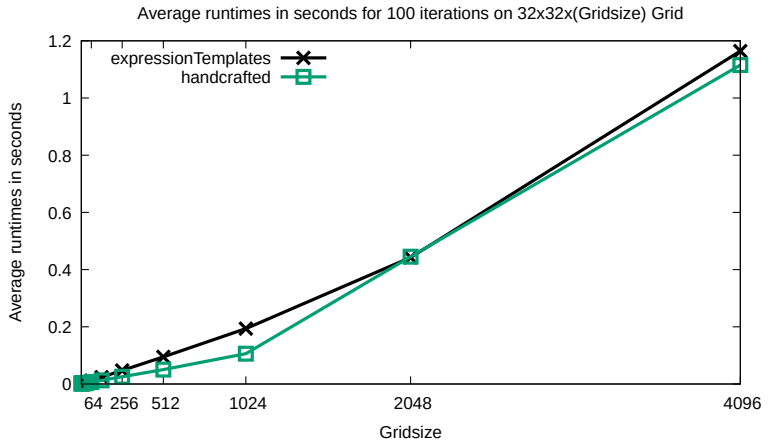


Abbildung 13: Durchschnittliche Laufzeiten mit globalen Sternen ohne Optimierungen

Abbildung 13 zeigt auf, dass für den Fall von globalen Sternen, bei denen der Prozessor den performanztechnischen Flaschenhals bildet, Expression Templates um nur einen geringfügigen Faktor langsamer laufen als handoptimierter Code. Liegt der Fokus dagegen auf speicherintensiven Operationen, wie es bei lokalen Sternen der Fall ist, zeigt sich eine Diskrepanz auf, die in Abbildung 12 graphisch veranschaulicht wird. Dies ist auf Aliasing zurückzuführen. Es wurden bereits weiterführende Konzepte wie Fast Expression Templates[Här07, S. 45] entwickelt, um diese Laufzeiten anzugleichen.

Es ist aus Abbildung 15 ebenfalls ersichtlich, dass die Operation zumindest im Falle lokaler Sterne eine speicherintensive Berechnung darstellt. Die höhere Zahl an Megaflops pro Sekunde für globale Sterne beweist zwar nicht, dass diese Berechnung rechenintensiv ist, jedoch lässt sich die Aussage der speicherintensiven Berechnung über die Implementierung mit lokalen Sternen treffen. Messungen der L2 Cache Bandbreite stärken diese Hypothesen aber weiter. Abbildung 16 zeigt, dass für lokale Sterne eine höhere Bandbreite ausgeschöpft wird.

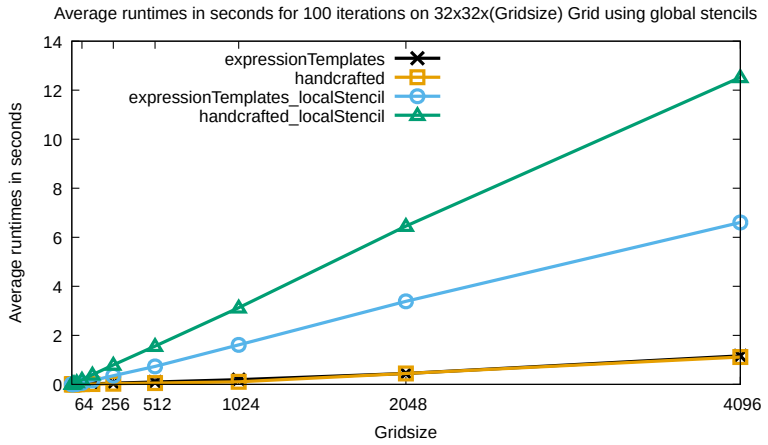


Abbildung 14: Durchschnittliche Laufzeiten ohne Optimierungen

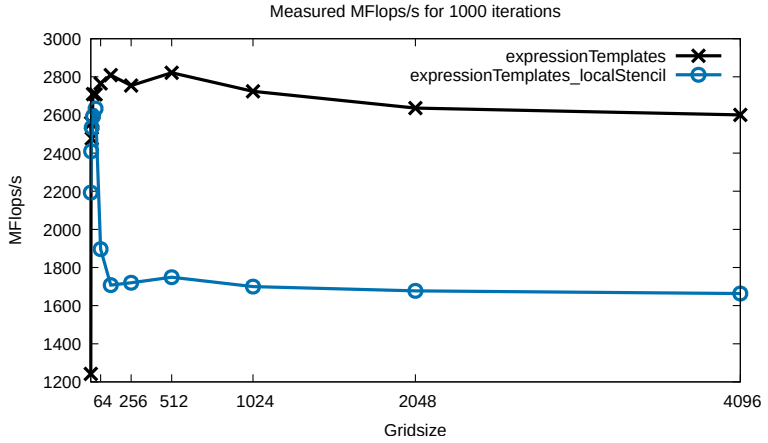


Abbildung 15: MFlops/s der Expression Template Implementierung

Berücksichtigt man nun noch das Cache–Hit/Miss Verhalten, das in Abbildungen 17 und 18 zu sehen sind, ist die Behauptung begründet, dass die Entscheidung zu lokalen oder globalen Sternen die Anforderungen an die zugrunde liegende Architektur bestimmt. Denn während die Miss–Ratio annähernd bei $\frac{1}{3}$ liegt, was bedeutet, dass die erforderten Daten oft im Cache zu finden sind, weisen die Miss-Raten eine weite Diskrepanz auf, weshalb gefolgert werden kann, dass durch die lokalen Sterne viel mehr Speicherzugriffe auf Addressen erfolgen, die nicht im Cache vorgehalten werden. Beides in Kombination lässt erahnen, dass die lokalen Sterne durch die erhöhten Speicheranforderungen die Berechnung speicherintensiver gestalten.

8.2 Linearisierung des Indexzugriffs

Abbildung 19 beschreibt die Messergebnisse zur Optimierung der Linearisierung. Dort ist ersichtlich, dass diese eine marginale Verschlechterung der Laufzeiten zur Folge hat. Dies kann möglicherweise auf die im hohen Grad erzwungen geinlineten Zugriffsoperatoren zurückzuführen sein, was zur Folge hat, dass dem Übersetzer durch diese Vorschrift Freiheitsgrade entzogen werden, mit deren Hilfe er durch situationsbedingte Entscheidungen schnelleren Binärcode erzeugen kann als mit der Vorschrift der Linearisierung. Jedoch ist alles über die Feststellung, dass die Linearisierung in diesem Fall nicht als Optimierung

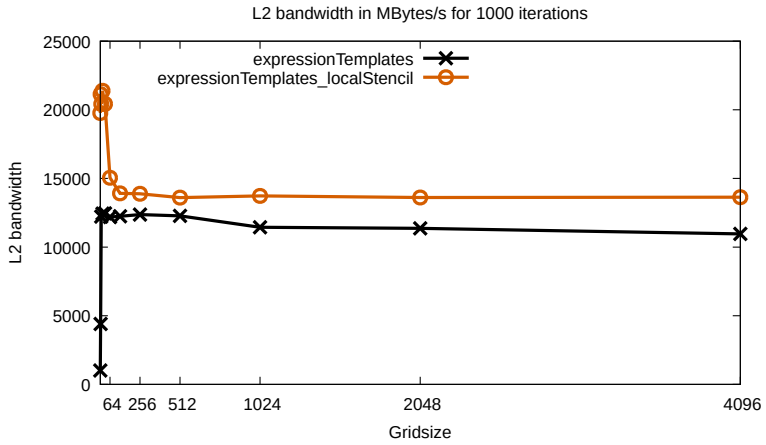


Abbildung 16: L2 Bandbreite in MBytes/s der Expression Template Implementierung

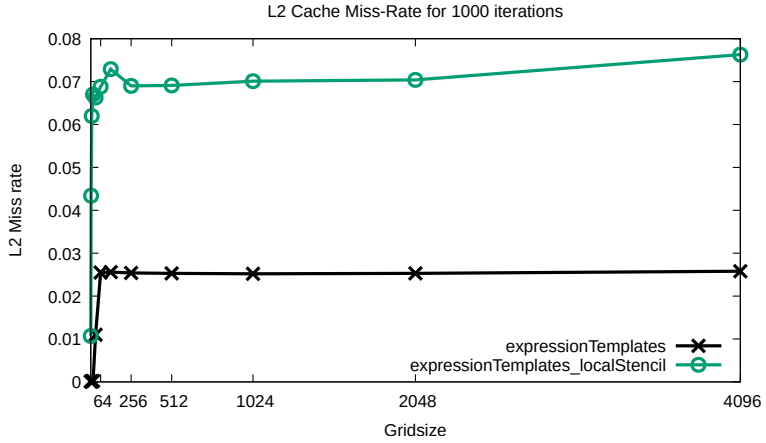


Abbildung 17: L2 Miss-Rate der Expression Template Implementierung

betrachtet werden kann, rein spekulativ und nicht gesichert.

8.3 Cache Blocking

Auf Grundlage von Abbildung 20 ist anzunehmen, dass der fehlende Laufzeitgewinn in dieser Hinsicht dem zuzuschreiben ist, dass der Flaschenhals der Applikation nicht im Speicherzugriff angesiedelt ist, sondern die Leistung des Prozessors den entscheidenden Faktor darstellt. Demnach ist es irrelevant, Speicher zu optimieren, wenn er für das Programm schnell genug zur Verfügung steht und vor allem die Berechnungen im Prozessor die langsamste Komponente des Ablaufs darstellt. Ein weiterer Grund für dieses Ergebnis ist die Tatsache, dass das Rot–Schwarz Gauß–Seidel–Verfahren wenig Raum für Cache Blocking bietet. Es werden für die Aktualisierung einer Farbe immer alle Punkte des Gitters benötigt, wobei nur jeder Siebenundzwanzigste beschrieben wird. Somit bietet dieses Verfahren keine guten Voraussetzungen für den Erfolg dieser Optimierung, weshalb eine marginale Verschlechterung dem erwarteten Ergebnis entspricht.

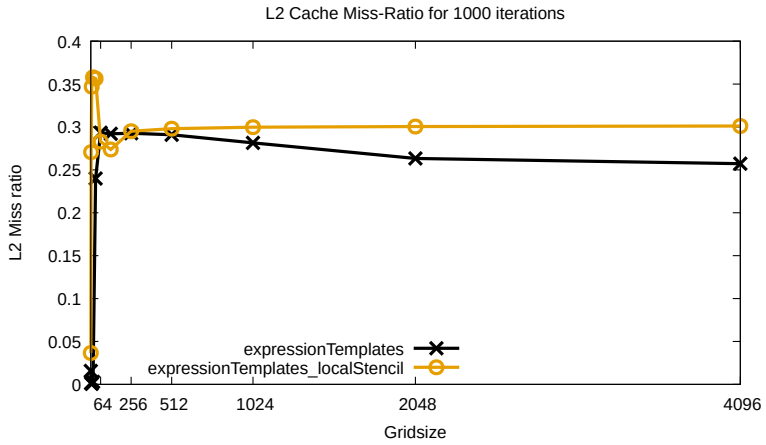


Abbildung 18: L2 Miss-Ratio der Expression Template Implementierung

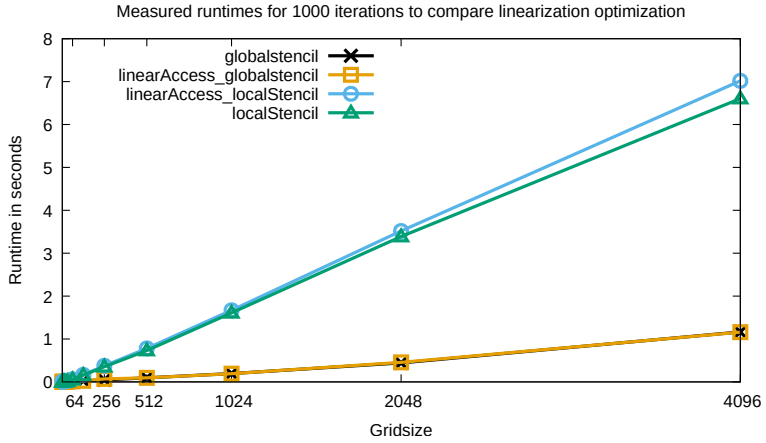


Abbildung 19: Vergleich der Laufzeiten zu Linearisierung in der Expression Template Implementierung

8.4 OpenMP

Als abschließendes Resultat ist noch der gemessene Speedup der Parallelisierung durch OpenMP zu betrachten. Abbildung 21 zeigt, dass die Implementierung bei der Verwendung von globalen Sternen das gewünschte Verhalten aufweist und mittels OpenMP für vier Kerne einen Speedup von vier zeigt. Der ebenfalls dort zu findende Speedup von zwei bezieht sich auf die Verwendung von lokalen Sternen. Die Anwendung selbst zeichnet sich durch einen höheren Grad an möglicher Parallelität aus, weshalb dieses Messergebnis die Hypothese der speicherintensiven Berechnung für lokale Sterne weiter stützt.

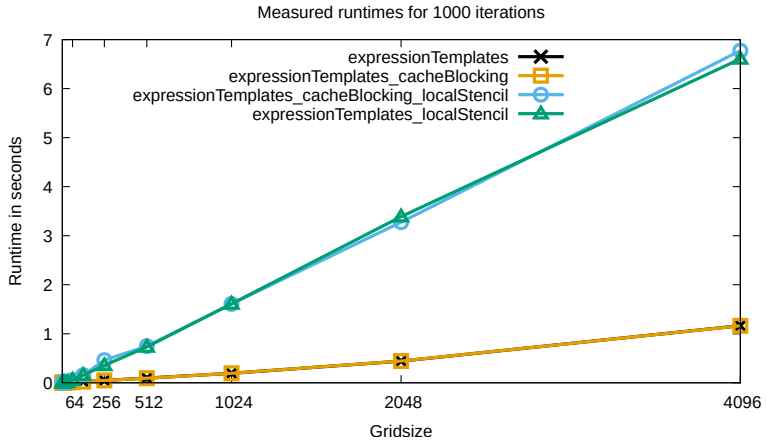


Abbildung 20: Vergleich der Laufzeiten der Cache Blocking Optimierung in der Expression Template Implementierung

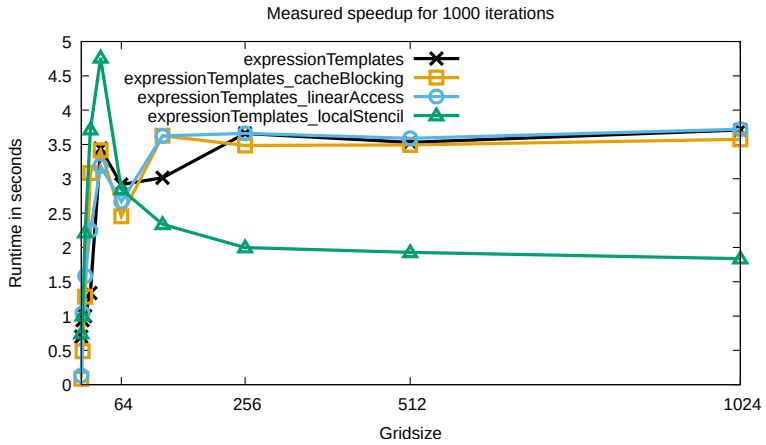


Abbildung 21: Gemessener Speedup der Implementierung bei vier physikalischen Rechenkernen

9 Fazit

Expression Templates ermöglichen es, eine benutzerfreundliche Schnittstelle bereitzustellen, ohne markante Abstriche hinsichtlich der Performanz verzeichnen zu müssen. Ebenso ist der Anwendungsfall iterativer Verfahren zur Lösung von partiellen Differenzialgleichungen, die sich im diskreten Raum als Sterne modellieren lassen, prädestiniert für die Anwendung dieser Technik. Die Benutzerfreundlichkeit für die Formulierung der diskretisierten Differenziale als Expression ist enorm komfortabel.

Die Messungen zeigen im parallelen Fall sowohl eine volle Auslastung aller Rechenkerne, als auch nahezu das damit theoretische Maß an Rechengeschwindigkeitsgewinn.

Somit stellen Expression Templates eine Optimierung im Entwicklungsprozess numerischer Software dar, mit deren Hilfe es möglich ist Entwicklungszeiten zu verkürzen. Die intuitive Verwendung, die vergleichbar guten Laufzeiten und die leichtere Wartung der Software durch Kapselung in einer Bibliothek sprechen durchaus für die weitere Verfolgung dieses Forschungsgebietes, das für komplexere Problemstellungen bereits weiter fortgeschritten ist.

Literatur

[BF] Richard L. Burden und J.Douglas Faires. *Numerical Analysis*. 9. Aufl.

[BoL] *Boost–Sofwarelizenz*. Softwarelizenz. URL: `https://www.boost.org/LICENSE_1_0.txt` (besucht am 27. 01. 2018).

[Boo] *Boost*. Softwarebibliothek. URL: `https://www.boost.org` (besucht am 27. 01. 2018).

[CJP08] Barbara Chapman, Gabriele Jost und Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. 2008.

[FP10] Hermann Friedrich und Frank Pietschmann. *Numerische Methoden*. 2010.

[Här07] Jochen Härdtlein. „Moderne Expression Templates Programmierung“. Dissertation. Friedrich-Alexander Universität Erlangen-Nürnberg, 2007.

[Pfl] Christoph Pflaum. *Simulation und wissenschaftliches Rechnen (SiWiR I) 2013/14*. URL: `https://www10.cs.fau.de/media/filer_public/22/71/2271ff4c-dab4-4e50-8432-4a469a74b877/skript_siwir.pdf` (besucht am 15. 01. 2018).

[Pfl01] Christoph Pflaum. „Expression templates for partial differential equations“. In: *Computing and Visualization in Science* 4.1 (Nov. 2001), S. 1–8.

[Stü05] Markus Stürmer. „Optimierung des Red–Black–Gauss–Seidel–Verfahrens auf ausgewählten x86–Prozessoren“. Studienarbeit. Friedrich-Alexander Universität Erlangen-Nürnberg, 2005.

[Vel] Todd Veldhuizen. *Blitz++*. Software-Bibliothek. URL: `https://github.com/blitzpp/blitz` (besucht am 19. 12. 2017).

[Vel95] Todd Veldhuizen. „Expression Templates“. In: *C++ Report* 7.5 (Juni 1995), S. 26–31.

Abbildungsverzeichnis

Listings