Institut für Parallele und Verteilte Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3265

Cache-effiziente Block-Matrix-Löser für die Partition of Unity Methode

Patrick Gründer

Studiengang:	Informatik
Prüfer:	Prof. Dr. Marc Alexander Schweitzer
Betreuer:	Prof. Dr. Michael Bader Dr. Stefan Zimmer
begonnen am:	02. November 2011
beendet am:	03. Mai 2012
CR-Klassifikation:	G.1.3, F.2.1

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abstract

Die *Partition of Unity Methode* findet Anwendung in gitterlosen Diskretisierungsverfahren zum Lösen elliptischer partieller Differentialgleichungen. Die bei der Diskretisierung entstehenden Gleichungssysteme besitzen eine Blockstruktur, die sich mittels der Multilevel Partition of Unity Methode asymptotisch optimal lösen lassen. Ein alternatives Verfahren zum Lösen dieser Gleichungssysteme stellen die vorkonditionierten Krylow-Unterraumverfahren dar. In dieser Arbeit wird ein auf der ILU-Zerlegung basierenders CG-Verfahren für Block-Matrizen implementiert, das auf Cache-effizienten Algorithmen basiert. Der Ausgangspunkt stellt die Bibliothek *TifaMMy* dar. Die in den letzten Jahren entwickelte Bibliothek basiert auf inhärent Cache-effiziente Algorithmen für dicht- und dünnbesetzte Matrizen. Dabei wird eine neue Datenstruktur für Blockmatrizen (BCRS) und die nötigen Algorithmen implementiert. Die Leistung des Block-Matrix-Löser wird mit der Multilevel Partition of Unity Methode verglichen.

1. Einleitung

In den letzten Jahren wurde die C++ Bibliothek *TifaMMy*[1] basierend auf Cache-effizienten Algorithmen für dicht und dünnbesetzte Matrixoperationen entwickelt. Grundlage vieler Operationen bilden Block-rekursive Algorithmen die mittels der Peano-Kurve definiert sind. Die erste Arbeit auf diesem Gebiet beschreibt einen *cache oblivious* Algorithmus für die dichtbesetzte Matrix-Matrix-Multiplikation mit Hilfe der Peano-Kurve [BZ06]. Als cache oblivious werden dabei Algorithmen bezeichnet, die jede gegeben Cache-Architektur optimal nutzen, ohne dabei spezielle Parameter zu kennen. Die darauffolgende Entwicklungen einer cache-oblivous LU-Zerlegung [HB09] und die Erweiterung um dünnbesetzte Matrix-Formate (CRS) bilden die Grundlage für die Weiterentwicklung zu einer Implementierung von etwa BLAS oder LAPACK auf der Basis Cache-effizienter Algorithmen.

Bei der Diskretisierung von partiellen Differentialgleichungen entstehen typischerweise dünnbesetzte Gleichungsysteme mit einer Blockstruktur. Die Blockgröße ergibt sich dabei durch die Anzahl an Freiheitsgraden, die zu jedem Gitterpunkt oder Patch assoziiert sind. Die *Partition of Unity Methode* (PUM) [Sch03, Sch] gehört zu den gitterfreien Verfahren zur numerischen Lösung elliptischer partieller Differentialgleichungen. Das Verfahren verwendet dabei als Löser ein asymptotisch optimales Mehrgitterverfahren.

In dieser Arbeit wird auf Grundlage Cache-effizienten Algorithmen ein ILU-vorkonditioniertes CG-Verfahren entwickelt und mit der Multilevel Partition of Unity Methode verglichen. Dazu wird auf Grundlage der TifaMMy - Bibliothek eine Datenstruktur für BlockMatrizen (BCRS) und die notwendigen Algorithmen entwickelt und untersucht. Das Hauptaugenmerk liegt dabei auf den Cache-effizienten Implementierungen der Matrix-Vektor-Multiplikation und der ILU-Zerlegung. Insbesondere werden Techniken, wie die Autovektorisierung und Parallelisierung mittels OpenMP zur Anwendung kommen.

Das erste Kapitel dieser Arbeit behandelt die nötigen Grundlagen. Dazu gehört eine Beschreibung der Partition of Unity Methode sowie das CG-Verfahren und die unvollständige LU-Zerlegung. Die Abschnitte über die Peano-Kurve und Peano-Matrix sind dabei die elementare Grundlage für die weiteren Kapitel.

[1] http://tifammy.sf.net/

Im Kapitel-3 werden Datenstrukturen für dünnbesetzte Matrizen diskutiert. Die am Ende beschriebenen BCRS-Datenstruktur liefert die Grundlage für die Implementierung der Peano-Blockmatrizen.

Im Kapitel-4 wird aufbauend auf den Datenstrukturen und den Grundlagen die Algorithmen für die Matrix-Vektor-Multiplikation und die ILU-Zerlegung genauer beschrieben.

Das Kapitel-5 beschreibt die wesentlichen Details der Implementierung.

Das vorletzte Kapitel 6 beschäftigt sich mit Leistungsvergleichen der einzelnen Bausteine des Lösers und liefert Vergleichswerte zu anderen Implementierungen.

2. Grundlagen

2.1. *Patition of Unity* Methode

Die *Partition of Unity* - Methode ist ein gitterfreies Verfahren zur numerischen Lösung partieller Differentialgleichungen. Diese Verfahren besitzen besonders für zeitabhängige Probleme mit komplizierten Geometrien Vorteile gegenüber den gitterbasierten Verfahren, da die Konstruktion qualitativ guter Gitter einen Großteil des Rechenaufwandes in Anspruch nehmen kann. Partikelbasierte Verfahren stammen direkt aus physikalischen Anwendungen, bei denen das jeweilige Gebiet von einer Menge von Partikeln diskretisiert wird. Dabei besitzen die Partikel keine feste Verbindung zueinander, das heißt sie sind unabhängig von einem Gitter. Die partielle Differentialgleichung wird dabei, auf dem zu berechnenden Gebiet, in ein System gewöhnlicher Differentialgleichungen transformiert. Die Diskretisierung nach der Zeit führt dann in jedem Zeitschritt zu einer Partikelverteilung und definiert eine näherungsweise Lösung der partiellen Differentialgleichungen mittels einer Dichtefunktion ([Sch]).

Die allgemeine Partition of Unity Methode für eine gitterlose Diskretisierung einer elliptischen partieller Differentialgleichungen besteht grob aus den folgenden Schritten. Für eine Wahl von n unabhängigen Punkten $\{x_i\}$ wird ein Trail- und Test-Raum V^{PU} konstruiert. Dazu wird an jedem Punkt x_i ein Flächenstück oder Volumen $\omega_i \subset \mathbb{R}^d$ angefügt, sodass die Vereinigung $\overline{\Omega} \subset \bigcup\{\omega_i\}$ der Flächenstücke oder Volumen eine offene Überdeckung $C_\Omega = \{\omega_i\}$ des Gebietes Ω bilden. Mit Hilfe der Gewichtsfunktionen $W_i : \mathbb{R}^d \to \mathbb{R}$ mit Träger $\mathrm{supp}(W_i) = \overline{\omega_i}$ können die lokalen Formfunktionen ϕ_i mittels der Methode von Shepard konstruiert werden. Die Funktionen ϕ_i bilden eine Zerlegung der Eins (Partition of Unity). Dann wird jede Funktion ϕ_i mit einer Sequenz von lokalen Approximationsfunktionen ${\psi_i}^n$ multipliziert, um eine Formfunktion höherer Ordnung zusammenzusetzen. Diese Produktfunktionen $\phi_i\psi_i$ werden dann zuletzt in eine lose Form zusammengesetzt, um ein lineares Gleichungssystem mittels der Galerkin-Diskretisierung zu formen.

$$V^{PU} := \sum_i \phi_i V_i^{p_i} = \sum_i \phi_i \,\mathrm{span}\langle\{\psi_i^n\}\rangle = \mathrm{span}\langle\{\phi_i\psi_i^n\}\rangle$$

2.2. Das CG-Verfahren

Das CG-Verfahren (*conjugate gradients*) ist ein effizientes Lösungsverfahren für große, dünnbesetzte, lineare Gleichungsysteme der Form

$$Ax = b,$$

wobei A eine symmetrische positiv-definite Matrix ist. Das Verfahren löst dabei iterativ das zu $Ax = b$ äquivalente Minimierungsproblem:

$$F(x) = \frac{1}{2}x^T Ax - b^T x.$$

Dazu wird in jedem Schritt mittels des Residuums $r = b - A\,x$ eines zu Beginn beliebigen Punktes entlang der Richtung des steilsten Abstiegs ein neuer Näherungswert bestimmt. Dabei wird die Schrittweite mit $\alpha = \frac{r^T r}{d^T A\,d}$ bezeichnet. Der Faktor $\beta = \frac{r^T r}{r^T r}$ bewirkt, dass die Richtung des Residuums orthogonal zu allen Residuen davor ist. Die Residuen bilden zusammen ein Orthogonalsystem des $\mathbb{R}^n$. Damit terminiert dieses Verfahren nach spätestens n Schritten mit $r = 0$.

In Algorithmus-2.1 ist das klassische CG-Verfahren im Pseudocode angegeben.

Algorithmus 2.1 Klassisches CG-Verfahren

function CG(A, b, x_0, ϵ)
  $r \leftarrow b - A\,x_0$
  $d \leftarrow r$
  while $d^T\,d > \epsilon^2$ **do**
    $\alpha \leftarrow \frac{r^T\,r}{d^T\,A\,d}$
    $x \leftarrow x + \alpha\,d$
    $\widehat{r} \leftarrow r - \alpha\,A\,d$
    $\beta \leftarrow \frac{\widehat{r}^T\,\widehat{r}}{r^T\,r}$
    $d \leftarrow \widehat{r} + \beta\,d$
    $r \leftarrow \widehat{r}$
  end while
end function

Dieses Verfahren verwendet im wesentlichen nur Matrix-Vektor- und Skalarprodukte zur Berechnung der Lösung eines Gleichungsystems. Eine weiteres, für diese Arbeit wichtiges, Verfahren ist das vorkonditionierte CG-Verfahren. Dabei wird das Gleichungsystem

$$Ax = b$$

formal mit einem Konditionierer P multipliziert:

$$P^{-1}Ax = P^{-1}b.$$

Auf dieses neue System wird dann das CG-Verfahren angewendet.

Algorithmus 2.2 Vorkonditioniertes CG-Verfahren (PCG)

function PCG(A, P, b, x_0, ϵ)
  $r \leftarrow b - A\,x_0$
  $h \leftarrow P\,r$
  $d \leftarrow h$
  while $d^T\,d > \epsilon^2$ **do**
    $\alpha \leftarrow \frac{r^T\,h}{d^T\,A\,d}$
    $x \leftarrow x + \alpha\,d$
    $\widehat{r} \leftarrow r - \alpha\,A\,d$
    $\widehat{h} \leftarrow P\,\widehat{r}$
    $\beta \leftarrow \frac{\widehat{r}^T\,\widehat{h}}{r^T\,h}$
    $d \leftarrow \widehat{h} + \beta\,d$
    $r \leftarrow \widehat{r}$
    $h \leftarrow \widehat{h}$
  end while
end function

2.2.1. Unvollständige LU-Zerlegung

Das Lösen von großen linearen Gleichungssystemen $Ax = b$ mit iterativen Lösungsverfahren benötigt häufig eine Vielzahl von Iterationen, bedingt durch die Kondition der gegebenen Matrix. Um die iterativen Verfahren zu beschleunigen, ist es möglich, das Gleichungssystem äquivalent umzuformen, um dessen Kondition zu verbessern. Dazu wird die Inverse von M (*Vorkonditionierer*) formal auf beiden Seiten multipliziert. Auf das so entstandene vorkonditionierte System $M^{-1}Ax = M^{-1}b$ wird dann das eigentliche iterative Lösungsverfahren angewendet. Die Konvergenz des Verfahrens hängt dann von der Kondition $\kappa(M^{-1}A)$ des vorkonditionierten Systems ab.

Die unvollständige LU-Zerlegung (engl. *incomplete LU factorization*, ILU) ist besonders für große dünnbesetzte Gleichungssysteme als Vorkonditionierer interessant. Da im Allgemeinen die LU-Zerlegung eines solchen Gleichungsystems auf eine viel größere Besetzungsstruktur führt, besteht die Idee darin, nur Werte einer vorgegebenen Besetzungstruktur zu berechnen, also den sogenannten *fill-in* zu limitieren. Wird die Besetzungstruktur der gesamten Matrix zugrunde gelegt, erhält man die eigentliche LU-Zerlegung. Es gibt eine Reihe von Varianten des Verfahrens. Diese unterscheiden sich zum einen in den Strategien zur Einschränkung des *fill-in* und zum anderen darin, für bestimmte Probleme bessere Approximationseigenschaften zu erhalten.

Die Verwendung der ILU-Zerlegung als Vorkonditionierer für das CG-Verfahren erweist sich typischerweise beim Lösen von elliptischen partiellen Differentialgleichungen als nicht optimal. Der Grund dafür ist, dass die Konditionszahl des vorkonditionierten Gleichungssystems schneller mit fallender Gitterweite h wächst, als bei nicht-vorkonditionierten Gleichungssystemen. Dazu werden in [CV94] verschiedene Modifikation der ILU-Zerlegung erläutert. Unter anderem eine Variante, genannt MILU (*modified*-ILU), die erhebliche Geschwindigkeitsvorteile bewirken kann.

2.3. Cache-effiziente Algorithmen

Cache-effiziente Algorithmen zeichnen sich durch eine optimale Nutzung des Cache-Speichers der zugrundliegenden CPU aus. Dabei spielt der Begriff der Datenlokalität eine wesentliche Rolle. Es gibt zwei Arten der Datenlokalität bezüglich eines Cache-Speichers. Zum einen die räumliche (*spartial*) und zum anderen die temporäre (*temporal*) Datenlokalität. Erstere bezeichnet den Zusammenhang von Speicher, der hintereinander referenziert wird. Letztere beschreibt die Wiederverwendung zeitlich in kurzer Abfolge verwendeter Speicherzugriffe. Dabei kann die gesamte Speicher-Hierarchie, d.h. Register, Cache, Hauptspeicher und Festplattenspeicher, als Ebenen (*levels*) des Cache aufgefasst werden ([Kum03]). Werden Daten von einem Prozess benötigt, so bezeichnet ein *cache hit* (Treffer) das Vorhandensein der Information in einem bestimmten Level des Cache. Liegen die Informationen nicht vor, so wird dies als *cache miss* (Verfehlungen) bezeichnet. Cache misses führen in der Praxis zu langen Wartezeiten für das Nachladen der Daten. Die grundliegende Idee der Cache-Optimierung von Algorithmen liegt also in der Reduktion dieser Verfehlungen. Als Grundlage für die theoretische Analyse von Algorithmen bezüglich des Cache-Speichers, dient dabei das Modell des "idealisierten Cache" ([FLPR99]).

In den Arbeiten der letzten Jahre wurden eine Reihe von Algorithmen auf Basis der Peano-Kurve entwickelt. Selbstähnliche raumfüllenden Kurven offenbaren durch ihre rekursive Definition eine passende Eigenschaft für die Zerlegung von Problemen in Teilprobleme. So lassen sich cache-oblivious Algorithmen für die Multiplikation und LU-Zerlegung von dichtbesetzten Matrizen auf Basis der Peano-Kurve definieren ([BZ06, HB09]). Diese Algorithmen nutzen die Eigenschaften der Peano-Kurve, um eine asymptotisch optimale temporäre und räumliche Datenlokalität zu erhalten.

2.4. Die Peano-Kurve

Die Peano-Kurve[1] ist eine surjektive, stetige und selbstähnliche raumfüllende Kurve. Die rekursive Definition besteht aus den vier Muster P, Q, R und S und erzeugt eine Kurve für eine quadratische Fläche mit Kantenlänge 3^k, $k \in \mathbb{N}$. Fasst man die Muster entlang einer Peano-Kurve als Wort über dem Alphabet $\Sigma = \{P, Q, R, S\}$ auf, so sind die Muster wie folgt definiert:

$$
\begin{aligned}
P &\to P\ Q\ P\ R\ S\ R\ P\ Q\ P \\
Q &\to Q\ P\ Q\ S\ R\ S\ Q\ P\ Q \\
R &\to R\ S\ R\ P\ Q\ P\ R\ S\ R \\
S &\to S\ R\ S\ Q\ P\ Q\ S\ R\ S
\end{aligned}
$$

In jedem Ersetzungschritt zur Erzeugung der nächst größeren Kurve, müssen alle Symbole des Wortes simultan ersetzt werden. Betrachtet man lediglich die Wörter, die durch diese Vorschrift, beginnend mit dem P-Muster

$$
\begin{aligned}
\mathcal{L}(P) \quad = \{ \quad & P, \\
& PQPRSRPQP, \\
& \dots
\end{aligned}
$$

erzeugt werden, so erkennt man zum einen die Selbsähnlichkeit der Kurve und zum anderen, dass jedes kürzere Wort der Sprache ein Präfix eines Wortes größerer Länge ist. Definiert man nun für jedes Muster einen Richtungsvektor, so eignet sich diese Grammatik zur Konstruktion eines Kellerautomaten, der eine Transformation entlang der Kurve darstellt. In jedem Schritt wird dann der jeweilige Richtungsvektor auf die aktuelle Position aufaddiert. Diese Eigenschaft kann für die Traversierung, wie sie bei der Matrix-Vektor-Multiplikation vorkommt, verwendet werden.

Die Peano-Kurve besitzt gewisse Freiheitsgrade. So lässt sie sich ohne weitere Modifikationen anstelle der vertikalen Durchlaufrichtung horizontal durchlaufen. Weiterhin ist es möglich, die Richtung alternieren zu lassen. In jedem dieser Fälle bleiben die Eigenschaften der Kurve erhalten.

[1] Die Peano-Kurve wurde 1890 von G. Peano als eine surjektive, stetige Abbildung $f : [0,1] \times [0,1] \to [0,1]$ von einem Flächenstück auf eine Gerade beschrieben.

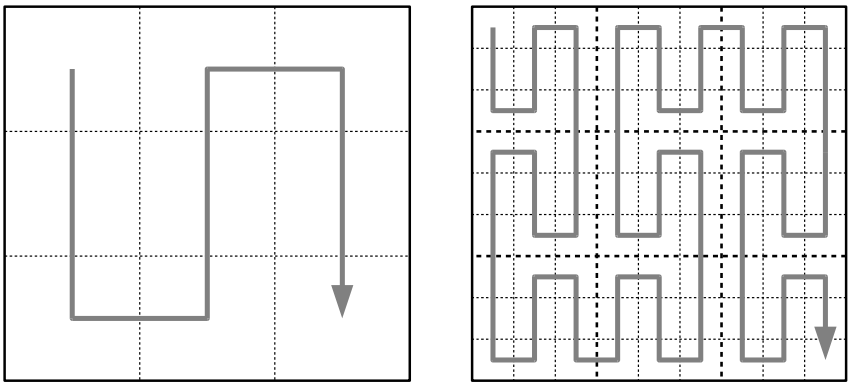

Abbildung 2.1.: Rekursive Erzeugung der Peano-Kurve

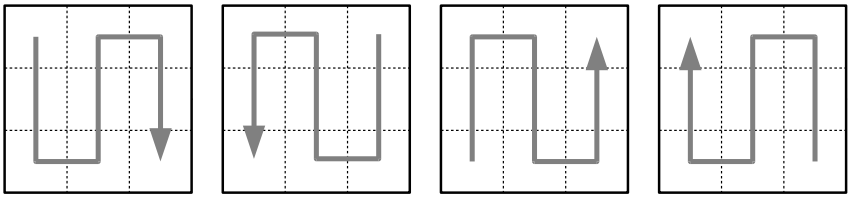

Abbildung 2.2.: Durchlaufrichtungen der Peano Kurve für die vier Muster P, Q, R und S.

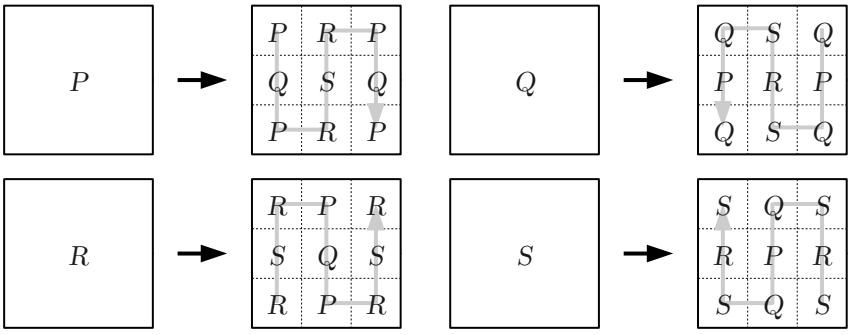

Abbildung 2.3.: Definition der Peanomuster zur Erzeugung der nächsten Rekursionstiefe.

2.5. Peano-Matrix

Eine Vielzahl der in dieser Arbeit verwendeten Algorithmen basieren auf der Peano-Kurve. In diesem Abschnitt wird näher auf die Transformation einer Matrix entlang

der Peano-Kurve, genannt *Peano-Nummerierung* oder *Peano-Reihenfolge*, eingegangen.

Sei $\mathbf{M}$ eine $n \times m$ - Matrix. Mit einer gegebenen, festen (Peano-)Blockgröße b_P wird nun ein Folge von Peano-Blöcken und ein Blockbelegungsbaum konstruiert.

Zuerst muss die kleinste Peano-Überdeckungsmatrix berechnet werden, die $\mathbf{M}$ räumlich enthält. Dazu bestimmt man zunächst die Anzahl der notwendigen Peano-Blöcke mittels

$$n_P := \left\lceil \frac{n}{b_P} \right\rceil \text{ sowie } m_P := \left\lceil \frac{m}{b_P} \right\rceil .$$

Damit kann die Rekursionsstiefe der Peano-Kurve durch

$$d_P := \lceil log_3 \left(\max(n_P, m_P) \right) \rceil$$

bestimmt werden. Die Peano-Überdeckungsmatrix besitzt dann die Dimension $3^{d_P} \times 3^{d_P}$. Da die Dimension von $\mathbf{M}$ im Allgemeinen keine Potenz von drei ist, bedarf es einer zusätzlichen Struktur, um die Information über die Belegung für jede Rekursionstiefe der Peano-Kurve zu speichern. Der so entstehende Baum besitzt pro Knoten jeweils neun Kinder für die nächste Rekursionstiefe. Für Blöcke, die sich außerhalb von $\mathbf{M}$ befinden, können dann die Unterbäume durch einen speziellen Wert als "ungenutzt" markiert werden.

Die Anzahl der Knoten in diesem Baum ergibt sich durch Aufsummieren der Belegungen für jede Rekursiontiefe der Peano-Matrix:

$$s_P := \sum_{k=0}^{d_P - 1} \left\lceil \frac{n_P}{3^k} \right\rceil \cdot \left\lceil \frac{m_P}{3^k} \right\rceil$$

Ist $n = m = 3^k$, mit $k > 0$, so ist

$$s = \sum_{k=0}^{d_P - 1} 3^k \cdot 3^k = \frac{1}{8} \left(9^{d_P} - 1 \right) =: s_{max}.$$

Zum speichern der vollbesetzten Matrix $\mathbf{M}$ in der Peano-Reihenfolge entsteht somit ein zusätzlicher Platzbedarf von $\mathcal{O}(s_{max})$ für den Blockbelegungsbaum. Eine genauere Abschätzung von s_{max} über

$$
\begin{aligned}
s_{max} &= \frac{1}{8}\left(9^{d_P} - 1\right) \\
&= \frac{1}{8}\left(9^{\lceil log_3(\max(n_P, m_P))\rceil} - 1\right) \\
&\leq \frac{1}{8}\left(9^{log_3(\max(n_P, m_P))+1} - 1\right) \\
&= \frac{9}{8} \cdot \left(\max(n_P^2, m_P^2) - 1\right) \\
&= \frac{9}{8} \cdot \left(\max(n_P, m_P)^2 - 1\right)
\end{aligned}
$$

liefert dann die Komplexität $\mathcal{O}(\max(n_P, m_P)^2)$. Da n_P und m_P bis auf den Teiler der Blockgröße die Matrixdimension sind, ergibt sich ein maximaler Speicherbedarf in der Größenordung der Matrix selbst.

In der Praxis wird der Blockbelegungsbaum effizient in einem Feld T aus ganzen Zahlen (Integer) organisiert. Die inneren Knoten enthalten dann neun Indizes zu ihren Kindern innerhalb von T. Einträge kleiner Null symbolisieren nicht gespeicherte Null-Blöcke oder Blöcke außerhalb der Matrix M. In der Ebene der Blätter werden dann die Indizes der jeweiligen Peano-Blöcke gespeichert, die sich in einem separaten Feld B befinden. Damit können auch dünnbesetzte Matrizen effizient repräsentiert werden. Bei der Beschreibung der Algorithmen auf dem Blockbelegungsbaum werden dann, durch zusätzliche Überprüfung der Belegung, Berechnungen auf Null-Blöcken nicht ausgeführt.

Das Auffinden eines bestimmten Blockes mit gegebenen Indizes erfordert einen Tiefendurchlauf durch den Blockbelegungsbaum der Matrix. Eine elegante iterative Variante für diese Aufgabe ist im Folgenden gegeben.

Algorithmus 2.3 Iterative Variante der Tiefensuche eines Peano-Block-Index durch den Blockbelegungsbaum.

function COORD2INDEX($row, column$)
  $dim \leftarrow 3^{d_P-1}$
  $i \leftarrow \left\lfloor \frac{row}{dim} \right\rfloor$
  $i \leftarrow \left\lfloor \frac{column}{dim} \right\rfloor$
  $pattern \leftarrow P$
  $index \leftarrow T[PeanoPattern[pattern][i][j]]$
  while $dim > 1$ **do**
    if $index \leq 0$ **then**
      return -1
    end if
    $row \leftarrow row - i * dim$
    $column \leftarrow column - j * dim$
    $dim \leftarrow \frac{dim}{3}$
    $i \leftarrow \left\lfloor \frac{row}{dim} \right\rfloor$
    $j \leftarrow \left\lfloor \frac{column}{dim} \right\rfloor$
    $index \leftarrow T[index + PeanoPattern[pattern][i][j]]$
  end while
  return $index$
end function

Der Algorithmus verwendet die (*lookup*-) Tabelle *PeanoPattern* zum Abruf des aktuellen Peano-Musters entsprechend der Abbildung-2.3.

2.5.1. Blockgrößen

Algorithmen, wie die in [BZ06] beschriebene Matrix-Multiplikation, basieren auf der Peano-Nummerierung und besitzen einen durch die Blockgröße b_P frei wählbaren Parameter. Im Folgenden wird beschrieben, welchen Effekt und welche Grenzen verschiedene Werte des Parameters haben.

Die Blöckgröße sollte so gewählt werden, dass für jede Operation alle Operanden zugleich in den L1-Cache der CPU passen. Aktuelle CPUs besitzen eine gleichmäßige Unterteilung des L1-Cache in Daten und Code. Zu betrachten ist jeweils nur die Größe für das Datensegment des Cache. Dieser Wert, sowie auch die Länge einer Cacheline, die später für die Ausrichtung der Peano-Blöcke im Speicher wichtig ist, können den Datenblättern entnommen werden. Die größe der Datentypen t sind in der nachstehenden Tabelle aufgeführt.

Datentyp	Größe in Bytes
single	4
double	8
complex	8
double complex	16

Abbildung 2.4.: Größe der vier BLAS-Datentypen

Um die maximalen Blockgrößen zu ermitteln, werden die Algorithmen einzeln betrachtet, da diese unter Umständen verschiedene Operanden haben.

Matrix - Multiplikation

Die Matrix-Multiplikation operiert, wegen $C = A \cdot B$, auf je drei Peano-Blöcken zugleich. Das liefert die einfache quadratische Gleichung

$$\mathrm{L1} = 3\,t\,{b_P}^2,$$

deren positive, nach unten abgerundet Lösung

$$b_P = \left\lfloor \sqrt{\frac{\mathrm{L1}}{3\,t}} \right\rfloor$$

die maximale Blockgröße für diese Operation liefert.

Matrix - Vektor - Multiplikation

Die Matrix - Vektor - Multiplikation $y = A \cdot x$ auf Peano-Blöcken benötigt pro Operation einen Peano-Matrixblock und zwei Vektoren der Länge b_P. Dies führt auf die Gleichung

$$\mathrm{L1} = 2\,b_P\,t + {b_P}^2\,t,$$

welche eine positive, nach unten abgerundet Lösung

$$b_P = \left\lfloor \frac{\sqrt{t^2 + l\,t} - t}{t} \right\rfloor$$

besitzt.

LU - Zerlegung

Der Algorithmus verwendet verschiedene Operationen auf den Peano-Blöcken. Diese Operationen sind im Abschnitt-4.2 genauer beschrieben und werden im Folgenden als bekannt vorausgesetzt. Die Operanden sind Peano-Blockmatrizen und es werden Operationen mit einem (*LUDecomposition*), zwei (*findLeft*) sowie (*findRight*), und drei Operanden (*mulSub*) verwendet. Die maximalen Blockgrößen ergeben sich zu

$$b_P = \left\lfloor \sqrt{\frac{\mathrm{L1}}{\omega \cdot t^2}} \right\rfloor$$

mit der Anzahl an Operanden $\omega = 1, 2, 3$ der jeweiligen Peano-Block-Operation.

Aktuelle CPUs von Intel und AMD besitzen sehr häufig einen L1-Daten-Cache der Größe L1 = 32768 Bytes. Für diesen speziellen Wert werden im Folgenden die Maximalwerte aufgeführt.

BLAS-Datentyp	$b_P(\omega = 1)$	$b_P(\omega = 2)$	$b_P(\omega = 3)$
single	90	64	52
double	64	45	36
complex	64	45	36
double complex	45	32	26

Tabelle 2.1.: Maximale Blockgrößen b_P für Peano-Matrixblöcke. (Matrix - Multiplikation und LU - Zerlegung)

BLAS-Datentyp	b_P
single	89
double	63
complex	44
double complex	11

Tabelle 2.2.: Maximal Blockgrößen b_P für die Matrix - Vektor - Multiplikation

Anmerkung: Die in Tabelle-2.1 und Tabelle-2.2 aufgeführten Werte sind theoretische Maximalwerte für dichtbesetzte Peano-Blöcke. Nicht berücksichtigt sind zusätzliche Daten, die etwa durch die Implementierung der Datenstruktur herrühren. Insbesondere muss für dünnbesetzte Datenstrukturen der Peano-Blöcke eine genauere Betrachtung angestrengt werden. In der Praxis führen größere Werte nicht unmittelbar zu Cache-Verfehlungen, was an der jeweiligen Cache-Architektur liegt.

Für die Optimierung der Datenstruktur ist ein Ausloten der Blockgröße b_P im Bezug auf das jeweilige System und der Problemstellung nötig.

3. Datenstrukturen für dünnbesetzte Matrizen

Zunächst soll angemerkt werden, dass es für den Begriff "dünnbesetzt" keine einheitliche Definition gibt. Einige Definitionen legen diesen über die Größenordung $\mathcal{O}(\max(n, m))$, oder einen bestimmten Prozentsatz der Nicht-Null-Belegung der zu betrachtenden $n \times m$ Matrix fest. Im Grunde genommen ist dieser Begriff aus algorithmischer Sicht zu betrachten, denn die Speicherung der Nulleinträge und die daraus resultierenden Nullprodukte können in verschiedenen Algorithmen zu unterschiedlichem Aufwand führen. Dazu wird folgende Auffassung zugrunde gelegt:

Eine Matrix wird als *dünnbesetzt* bezeichnet, wenn es sich "lohnt" nur die Nicht-Null Einträge zu speichern.

Dabei bezieht sich "lohnt" auf die Platz- und Zeitkomplexität der Algorithmen im Bezug auf die anzuwendenden Algorithmen selbst.

Im Allgemeinen gibt es zwei verschiedene Möglichkeiten der Speicherung von Matrixdaten, je nachdem, ob die Linearisierung der Matrix im Speicher reihen- (*row-major*) oder spaltenweise *column-major* vorliegt. Da diese Betrachtung in den meisten Fällen symmetrisch ist und es für viele der aufgeführten Datenstrukturen eine *column-major* - Variante existiert, wird in dieser Arbeit lediglich die *row-major* - Variante betrachtet.

Es gibt in der Literatur (vgl. [BBC+94]) eine ganze Reihe verschiedener Datenstrukturen für dünnbesetzte Matrizen. Je nach Eigenschaften der Matrix und der durchzuführenden Operationen, die durch das zu lösende Problem gegeben sind, existieren verschiedene Datenstrukturen. Allen gemein ist eine möglichst günstige Speicherung der Nicht-Null Einträge, verbunden mit der Repräsentation der Indizes, die Algorithmen effizient darauf umsetzbar macht.

3.1. Compressed Row Storage

Das *Compressed Row Storage* Format (CRS) einer $n \times m$ - Matrix mit nnz - Matrixeinträgen, die nicht Null sind, besteht aus drei Feldern. Eines für die Werte (val) und zwei für die Verwaltung von Zeilen- (col_ind) und Spaltenindizes (row_ptr). Die Spaltenindizes werden dabei direkt in das Feld eingetragen,

wohingegen das *row_ptr* - Feld den Index im *val*-Feld des ersten Elements der Zeile speichert. Darüberhinaus ist es für die Iteration über die Datenstruktur sinnvoll, dem *row_ptr* - Feld ein weiteres Element hinzuzufügen, speziell um das Ende der letzten nicht-leeren Zeile in der Matrix zu markieren ($row\_ptr[m] = nnz$). Werden Null-Zeilen zugelassen, so werden diese entsprechend mit dem Wert -1 markiert und zusätlich eine Funktion *next_row* definiert, die die nächste Nicht-Null-Zeile zurückgibt. Die Funktion lässt sich durch

function $next\_row(\ i\ )$
  repeat
    $i \leftarrow i + 1$
  until $row\_ptr[i] < 0$
  return i
end function

beschreiben. Diese Implementierung benötigt aufgrund des zusätzlichen Elements $row\_ptr[m]$ keine weitere Abbruchbedingung ($row < m$), um Überläufe zu vermeiden. Für alle Eingaben $0 \leq row < m$ führt spätestens der m-te Durchlauf mit $row\_ptr[m] = nnz \geq 0$ zum Abbruch der Schleife.

Mit dieser Funktion werden dann, falls $\mathrm{val}[k] = a_{i,j}$ ist, für $0 \leq k < nnz$, folgende Bedingungen erfüllt:

$$\mathrm{col\_ind}[k] = j$$

$$\mathrm{row\_ptr}[i] \leq k < \mathrm{row\_ptr}[\mathrm{next\_row}(i)].$$

Zusammenfassend besteht die Datenstruktur aus den drei genannten Feldern

$$\begin{aligned} val &= \{0, \ldots, nnz - 1\} \\ col\_ind &= \{0, \ldots, nnz - 1\} \\ row\_ptr &= \{0, \ldots, n\} \end{aligned}$$

und benötigt somit $\mathcal{O}(2nnz + n + 1) = \mathcal{O}(nnz)$ Speicherplatz zur Repräsentation der Matrix $\mathbf{A}$. Ein Beispiel für eine 9×9 - Matrix im CRS - Format ist in Abbildung-3.2 aufgeführt.

Der Aufbau der Datenstruktur auf diese Weise birgt einige Vorteile. Vor allem, weil man, wie in Abbildung-3.1 zu sehen ist, die Iteration über die Matrix analog zur Schleifenformulierung auf zweidimensionalen Feldern beschreiben kann.

Algorithmus 3.1 Schleifenformulierungen der Matrix-Vektor-Multiplikation für zweidimensionale Felder im Vergleich zur CRS-Datenstruktur

```
procedure MVP( x, y )
    for i = 0 → n − 1 do
        for j = 0 → n − 1 do
            y[i] ← a[i][j] · x[j]
        end for
    end for
end procedure

procedure MVP_CRS( x, y )
    for i = 0 → n − 1 do
        for k = row_ptr[i] → row_ptr[next_row(i)] − 1 do
            y[i] ← val[k] · x[col_ind[k]]
        end for
    end for
end procedure
```

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 6 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 & 0 & 5 & 0 & 0 \\ 0 & -1 & 0 & 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 3 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
val	1	3	2	6	7	5	-1	4	3	1	1	3	8	3	2

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
col_ind	0	5	1	4	5	7	1	3	1	4	8	0	7	3	4

index	0	1	2	3	4	5	6	7	8	9	10
row_ptr	0	2	4	6	-1	8	-1	11	13	-1	15

Abbildung 3.1.: Beispiel für eine im CRS-Format gespeicherte dünnbesetzte 10×10 - Matrix.

3.2. Datenstrukturen für Block-Matrizen

Blockmatrizen entstehen typischerweise bei der Diskretisierung von partiellen Differentialgleichungen bei denen jeder Gitterpunkt eine bestimmte Anzahl von Freiheitsgraden d besitzt. Die Matrix lässt sich dann in Blöcke der Größe d partitionieren. Jeder Nicht-Null Block der Partition wird dann innerhalb der Datenstruktur vollständig gespeichert. Dabei wird in Kauf genommen, dass in den Blöcken enthaltenen Nullen dennoch gespeichert werden. Da die Blockgröße d für die gesamte Matrix fest ist, bietet es sich an die in 3.1 beschriebene CRS-Datenstruktur auf Blöcke zu erweitern.

3.2.1. Block Compressed Row Storage

Das *Block Compressed Row Storage* - Format (BCRS) ist eine auf uniforme Blöcke der Größe b verallgemeinerte Variante des CSR. Für eine Matrix M mit n_b Block-Reihen und m_b Block-Spalten, werden die $nnzb$ Nicht-Null-Blöcke analog zum CRS-Format in einem Feld val zusammenhängend gespeichert. Die Felder row_ptr und col_ind werden analog verwaltet.

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 6 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 & 0 & 5 & 0 & 0 \\ 0 & -1 & 0 & 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 3 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

index	0	1	2	3	4	5
val	1 0 0 2	0 3 6 0	0 0 0 -1	0 0 0 4	0 7 0 0	0 5 0 0

6	7	8	9	10	11	12
0 0 0 3	0 0 1 0	0 0 1 0	3 0 0 0	0 0 0 8	0 3 0 0	2 0 0 0

index	0	1	2	3	4	5	6	7	8	9	10	11	12
col_ind	0	2	0	1	2	3	0	2	4	0	3	1	2

index	0	1	2	3	4	5
row_ptr	0	2	6	9	11	13

Abbildung 3.2.: Beispiel für eine im BCRS-Format gespeicherte dünnbesetzte 10×10 - Matrix mit Blockgröße $b = 2$.

3.2.2. Beliebige Blockgrößen

In vielen Fällen bilden benachbarte Blöcke innerhalb von Blockmatrizen Blöcke größerer Ordnung. Ein anderer Ansatz besteht nun darin, möglichst große Blöcke zu erfassen. Eine solche Datenstruktur stellt aus algorithmischer Sicht ein größeres Problem dar. Dazu betrachtet man ein Matrixprodukt zweier so generierter Datenstrukturen. Offensichtlich führen die Skalarprodukte des Algorithmus zu Zugriffsproblemen, da Blöcke nicht, wie bei dem BCRS-Format, über ihre Indizes assoziiert sind. Das bedeutet, dass korrespondierende Einträge zu Suchproblemen innerhalb der Datenstruktur führen. Allerdings werden derartige Zugriffe nicht in jedem Algorithmus benötigt. Die Matrix-Vektor-Multiplikation benötigt lediglich die Position und die Größe eines Blockes, um mittels der korrespondierenden Vektor-Einträge das Skalarprodukt berechnen zu können.

3.3. Blockpartition

Eine Blockpartition $\mathcal{B}$ einer Matrix M, ist eine Menge von Matrixblöcken, die sich nicht überschneiden und deren Vereinigung wieder M ergibt. Als Elemente der Blockpartition sind beliebige Matrixblöcke B mit positiver Dimension zugelassen. Für die weitere Verwendung wird lediglich die Teilmenge von $\mathcal{B}$ betrachtet, deren Elemente mindestens einen von Null verschiedenen Eintrag besitzen. Eine so definierte Blockpartition lässt sich nun auf natürliche Weise in eine Datenstruktur übersetzen. Die Elemente von $\mathcal{B}$ werden als Liste oder Feld von Blöcken mit ihren Werten, der Blockdimension und Position gespeichert.

Die einfachste mögliche Datenstruktur für das Matrix-Vektor-Produkt speichert schlicht die Blöcke der Matrix in einer Liste, mit ihrer Größe und Position.

3.4. Optimierung

Dieser Abschnitt beschreibt die in dieser Arbeit verwendeten Konzepte und Techniken für die Optimierung.

3.4.1. Parallelisierung mittels Autovektorisierung

Eine Reihe von Compilern, wie beispielsweise der Compiler von Intel oder auch der GCC, besitzen neben vielen anderen Programmtransformationen auch Optimierungen für die Generierung von Vektoren aus sequenziellen Daten. Operationen auf den Vektoren können dann mittels SIMD-Befehlen in der CPU parallelisiert werden. SIMD steht für *single instruction, multiple data*, womit die Verarbeitung von mehreren Daten mit einem Befehl gemeint ist. Im Gegensatz zur üblichen Verarbeitung von Daten in der CPU, der sogenannten SISD (*single instruction, single data*), ist es mit SIMD-Befehlen möglich, ganze Vektoren zu verarbeiten. Die Länge der Vektoren sind dabei auf eine gewisse Länge beschränkt. Befehlssätze wie MMX, 3DNow!, SSE [1] und das neuere AVX bieten eine Reihe von Befehlen für typische Probleme in der Verarbeitung von Fließkommaberechnungen. SSE-Befehle besitzen zwei Operanden mit je 128 Bit breiten Registern. Damit können vier singel-, zwei double-, zwei complex- oder vier complex-double- Werte zugleich verarbeitet werden. Die jüngste Befehlssatzerweiterung (AVX) bringt eine Verbreiterung der Register auf 256 Bit mit sich. Zudem kommt noch ein neues Befehlsformat im drei Operanden-Format hinzu.

[1] Es gibt eine Reihe von SSE-Befehlssätzen, diese sind u.a. SSE2, SSE3, SSSE3, SSE4, SSE4a und SSE5.

Die Optimierung mittels dieser Techniken ist alles in allem sehr speziell und führt beim Aufkommen neuerer Plattformen häufig zu kompletten Neuentwicklungen. Der hier gewählte Ansatz verwendet die Fähigkeiten moderner Compiler, vektorisierten Code generieren zu können. Für viele Probleme werden heute bereits Vektorisierungen automatisch durch den Compiler durchgeführt. Allerdings müssen in gewissen Fällen durch den Programmierer spezielle Hinweise annotiert werden, um den Compiler anzuweisen die Optimierung durchzuführen. Solche Fälle treten zum Beispiel dann auf, wenn der Compiler die Abhängigkeiten von Zeigern nicht herleiten kann.

Die Verarbeitung von Blöcken gegenüber einzelner Elementen birgt das Potential einer Reihe von Optimierungen. So ist es möglich, die Blockoperationen mittels SIMD-Befehle zu vektorisieren. Die Vektorlängen sind, bedingt durch die Registergrößen, auf Zweierpotenzlänge limitiert. Für die Fälle $d = 2k$, $k > 2$ ist dieses Vorgehen ohne Weiteres für alle BLAS-Datentypen möglich. Probleme entstehen durch die Fälle $d = 2k + 1$, $k > 1$, bei denen Überlappungen entstehen und Speicheradressen nicht die nötige Ausrichtung für schnelle Ladebefehle besitzen können. Dies betrifft insbesondere auch den in der Praxis häufig anzutreffenden Fall $d = 3$.

3.4.2. Parallelisierung mittels OpenMP

OpenMP (Open Multi-Processing) ist eine Programmierschnittstelle, die seit einigen Jahren in Kooperation vieler Compiler- und Hardwareherstellern entwickelt wurde. OpenMP ist für Shared-Memory-Systeme mit symmetrischen Multiprozessoren (SMP) gedacht. Solche Systeme besitzen viele gleichartige Prozessoren, die einen gemeinsamen Speicher verwenden. Die Verwendung findet über Compiler-Direktiven statt, die direkt in den Quelltext eingebettet werden.

Für die Parallelisierung von rekursiven Funktionen eignet sich insbesondere das OpenMP Task-Konzept.

4. Algorithmen

In den Arbeiten [May06, HB09] wurden cache-oblivious Algorithmen für die Matrix-Multiplikation und die LU-Zerlegung unter Verwendung der Peano-Kurve beschrieben. Diese Ansätze werden im Folgenden dahingehend angepasst, dass sie Block-Matritzen genügen. Die in Abschnitt-2.5 eingeführte Peano-Matrix, ist auch die Grundlage der Implementierungen dieser Algorithmen in *TifaMMy*.

Führt man für dünnbesetzte Matrizen die in Abschnitt-2.5 beschriebene Kompression von Null-Blöcken ein, so werden diese nicht gespeichert. Bei der Durchführung einer Matrix-Multiplikation $C = AB$ werden dann für Null-Blöcke in C keine Werte berechnet. Aus diesem Grund muss dafür Sorge getragen werden, dass die Matrix C für diese Operation stets vollbesetzt ist. Allerdings wird diese Eigenschaft für die Berechnung der unvollständigen LU-Zerlegung benötigt.

Im Folgenden wird aber zunächst der Algorithmus für die cache-oblivious Matrix-Vektor-Multiplikation beschrieben.

4.1. Matrix - Vektor - Multiplikation

Die Beschreibung für einen *cache-oblivious* Matrix-Vektor-Multiplikation auf Basis der Peano-Kurve und Peano-Nummerierung erfolgt entlang der Peano-Kurve selbst. Die folgende Beschreibung verwendet Peano-Vektoren, die analog zur Peano-Matrix oder genauer als $3^k \times 1$-Matrix definiert werden können. Ausgehend vom Muster P der rekursiven Definition (vgl. Abbildung-2.3), gibt es lediglich zwei Typen von Peano-Vektoren:

$$P \to P\,Q\,P$$
$$Q \to Q\,P\,Q.$$

Für das Startmuster $P{+}{=}P \cdot P$ wird dazu die rekursive Definition (Abbildung-2.3) eingesetzt. Die entstehenden Tripel für Muster und Index sind in Abbildung-A.1 für dieses Schema vollständig gelistet.

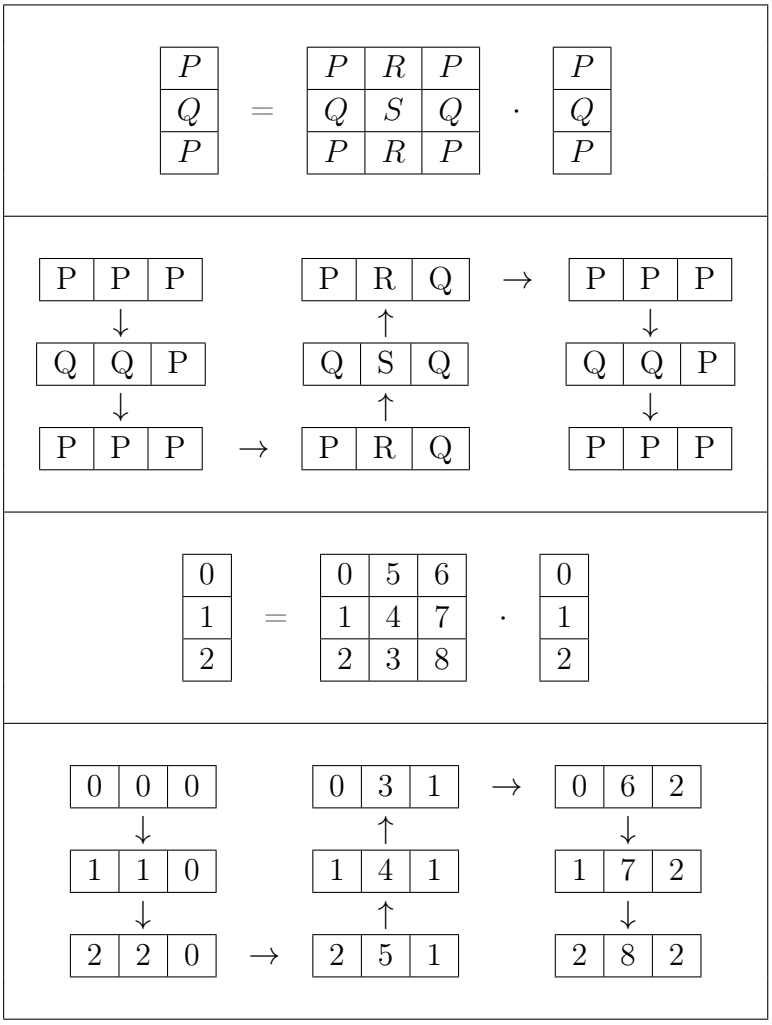

Abbildung 4.1.: Peano Matrix-Vektor-Multiplikationsschema $P \mathrel{+}= P \cdot P$

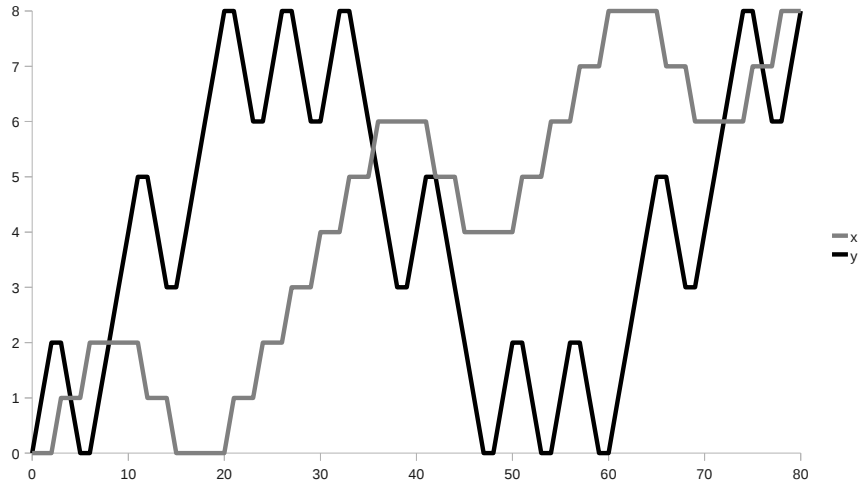

Abbildung 4.2.: Speicherzugriff auf die Vektoren x und y einer 9×9 - Matrix-Vektor-Multiplikation. Der Zugriff auf A erfolgt entlang der Peano-Kurve und wurde nicht eingezeichnet.

Die Algorithmen lassen sich ohne weiteres mittels Lookup-Tabellen implementieren.

Listing 4.1 BCRS-Block-Matrix Produkt

```
void PeanoMatrixVectorMultiplication(
    Type p,
    int index_a,
    int index_b,
    int index_c,
    unsigned int dim
) {
    if( dim > 3 ) {
        for( int i = 0; i < 9; i++ ) {
            if( sima[ index_a + mv_pattern_lut[ p ][ i ][ 1 ] ] > -1 ) {
                // Recursive call
                PeanoMatrixVectorMultiplication(
                    mv_pattern_lut[ p ][ i ][ 0 ],
                    sima[ index_a + mv_pattern_lut[ p ][ i ][ 1 ] ],
                    index_b + mv_pattern_lut[ p ][ i ][ 2 ] * dim,
                    index_c + mv_pattern_lut[ p ][ i ][ 3 ] * dim,
                    dim / 3
                );
            }
        }
    } else {
        for( int i = 0; i < 9; i++ ) {
            if( sima[ index_a + i ] > -1 ) {
                // Peano-Block Matrix-Vector-Multiplication
                y[ index_c + mv_lut[ p ][ i ][ 1 ] ] +=
                    bima[ sima[ index_a + i ] ] * x[ index_b + mv_lut[ p ][ i ][ 0 ] ];
            }
        }
    }
}
```

4.2. ILU-Zerlegung für BCRS

Als Grundlage für die Implementierung der ILU-Zerlegung wird die Cache-effiziente LU-Zerlegung verwendet. Der in [May06, HB09] beschriebene cache-oblivious Algorithmus berechnet die LU-Zerlegung für eine gegebene Matrix ohne zusätzlichen Speicherbedarf (in-situ). Das Verfahren ist für vollbesetzte Matrizen beschrieben worden, lässt sich aber durch die in Abschnitt-2.5 beschriebenen Markierungen der Null-Blöcke ohne weitere Anpassungen in eine ILU-Zerlegung für dünnbesetzte Block-Matrizen mit Blockgröße b_P überführen.

Ferner werden während der Berechnung die folgenden vier Operationen auf den Peano-Böcken ausgeführt:

- LUDecomposition: Berechnung der LU-Zerlegung $LU = A$ eines Peano-Blocks der Diagonalen.
- findLeft: Lösen des Gleichungssystems $LU = A$ mit einer gegebenen oberen Dreiecksmatrix U für den vollbesetzten Peano-Block L.
- findRight: Lösen des Gleichungssystems $LU = A$ mit einer gegeben unteren Dreiecksmatrix L für den vollbesetzten Peano-Block U.
- mulSub: Subtraktion des Peano-Block-Produktes der vollbesetzten Blöcke L und U von A. Kurz: $A := A - LU$.

Die Algorithmen der genannten vier Operationen sind in Algorithmus-A.1, Algorithmus-A.2, Algorithmus-A.3 und Algorithmus-A.4 aufgeführt.

Die Routinen *findLeft* und *findRight* entsprechen jeweils den beiden inneren Schleifen der *LUDecomposition*-Routine.

Die Verallgemeinerung der Algorithmen auf Blockmatrizen mit einer festen Blockgröße $b \in \mathbb{N}$, und $b|b_P$, ist ohne Weiteres möglich. Zunächst fasst man die Matrixeinträge $A_{i,j}$ als Matrixblöcke auf und definiert die Routinen *LUDecomposition*, *findLeft*, *findRight* sowie *mulSub* unverändert auf den Blöcken $A_{i,j}$.

Blocked_LUDecomposition

Algorithmus 4.1 Blockbasierte LU-Zerlegung

```
function BLOCKED_LUDECOMPOSITION( A )
    for i = 0 ... n_P/b − 1 do                                   // j < i
        for j = 0 ... i − 1 do
            for k = 0 ... j − 1 do
                mulSub( A_{i,j}, A_{i,k}, A_{k,j} )
            end for
            findLeft( A_{i,j}, A_{j,j} )
        end for                                                  // j = i
        for k = 0 ... i − 1 do
            mulSub( A_{i,i}, A_{i,k}, A_{k,i} )
        end for
        LUDecomposition( A_{i,i} )                               // j > i
        for j = i + 1 ... n_P/b − 1 do
            for k = 0 ... i − 1 do
                mulSub( A_{i,j}, A_{i,k}, A_{k,j} )
            end for
            findRight( A_{i,j}, A_{i,i} )
        end for
    end for
end function
```

Die Algorithmus lässt sich ebenfalls mit nur einer k - Schleife formulieren.

Algorithmus 4.2 Blockbasierte LU-Zerlegung (alternative Formulierung)

```
function BLOCKED_LUDECOMPOSITION( A )
    for i = 0 ... n_P/b − 1 do
        for j = 0 ... n_P/b − 1 do
            for k = 0 ... min(i, j) − 1 do
                mulSub( A_{i,j}, A_{i,k}, A_{k,j} )
            end for
            if j < i then
                findLeft( A_{i,j}, A_{j,j} )
            else if i = j then
                LUDecomposition( A_{i,i} )
            else
                findRight( A_{i,j}, A_{i,i} )
            end if
        end for
    end for
end function
```

Blocked_findLeft

Algorithmus 4.3 findLeft-Routine für die vollbesetzte Blockmatritzen A und B

```
function BLOCKED_FINDLEFT( A, B )
    for i = 0 ... n_P/b − 1 do
        for j = 0 ... n_P/b − 1 do
            for k = 0 ... j − 1 do
                mulSub( A_{i,j}, A_{i,k}, A_{k,j} )
            end for
            findLeft( A_{i,j}, A_{j,j} )
        end for
    end for
end function
```

Blocked_findRight

Algorithmus 4.4 findRight-Routine für die vollbesetzte Peano-Blöcke A und B

function BLOCKEDFINDRIGHT(A, B)
  for $i = 1 \ldots n_P - 1$ **do**
    for $j = 0 \ldots n_P - 1$ **do**
      for $k = 0 \ldots i - 1$ **do**
        $B_{i,j} \leftarrow B_{i,j} - A_{i,k} \cdot B_{k,j}$
      end for
    end for
  end for
end function

Blocked_mulSub

Algorithmus 4.5 mulSub-Routine für die vollbesetzte Peano-Blöcke A und B

function BLOCKED_MULSUB(C, A, B)
  for $i = 1 \ldots n_P - 1$ **do**
    for $j = 0 \ldots n_P - 1$ **do**
      for $k = 0 \ldots n_P - 1$ **do**
        $C_{i,j} \leftarrow C_{i,j} - A_{i,k} \cdot B_{k,j}$
      end for
    end for
  end for
end function

5. Implementierung

Die Implementierung der BCRS-Datenstruktur und der Algorithmen verwendet den Quelltext der TifaMMy[1]-Bibliothek der Version 2.3.1.

TifaMMy ist eine C++ - Bibliothek die Konzepte der Template-Metaprogrammierung nutzt, um eine Trennung von Datenstrukturen und Algorithmen zu ermöglichen. Die Bibliothek besitzt eine logische Unterteilung in verschiedene Kategorien von Funktionalitäten, die in verschiedenen Verzeichnissen abgelegt sind. Die Kategorien sind

- Algorithmen (*algorithm*) Beinhaltet Algorithmen, die auf der Peano-Kurve definiert sind.
- API[2] (*api*) Beinhaltet die Schnittstelle für den Programmierer. Die Klassen sind typischerweise Wrapper der eigentlichen Peano-Matrix-Datenstruktur und besitzen keine Template-Parameter mehr.
- Allgemein (*common*) Klassen-Definition wie Matrix, PeanoMatrix u.ä
- Datentypen (*datatype*) Beinhaltet die unterschiedlichen Datentypen. D.h. Dense-, Hybrid-, bzw. BCRSMatrix
- Datenstrukturen (*datastruct*) Hier werden die Datenstrukturen der Peano-Blöcke abgelegt.
- Speicherverwaltung (*memmgmt*) Die Speicherverwaltung ist ebenfalls modularisiert.

Die Aufteilung wird durch eine Technik, genannt *Curiously recurring template pattern* (CRTP), ermöglicht. Diese Technik ermöglicht statischen Polymorphismus durch rekursive Template-Muster. Dabei ist die zentrale (Template-)Klasse die Peano-Matrix, die durch den Basis-, Blockmatrix- und Matrixdatentyp parametrisiert werden kann. Die Implementierung der BCRS-Datenstruktur in dieser Arbeit erfolgt anhand dieser Aufteilung.

[1] http://sourceforge.net/projects/tifammy/
[2] Application Programming Interface

Ein wichtiges Merkmal der Implementierung ist die statisch festgelegte Blockgröße. Diese lässt sich in einer Headerdatei frei einstellen, erfordert aber die Neuübersetzung des Programms. Der Grund für diese Entscheidung liegt in der zu nutzenden Autovektorisierung. Moderne Compiler können für Felder mit statisch bekannter Größe besonders gute Optimierungen vornehmen.

5.1. Implementierung der Peano-Matrix

Die Peano-Matrix der TifaMMy-Bibliothek verwendet ein einfaches Feld für die Organisation der Peano-Blöcke. Für die Implementierung der BCRS-Matrix ist diese Organisation unzureichend. Aus diesem Grund wurde eine modifizierte Version als Grundlage implementiert, die eine beliebige Peano-Blockgröße zulässt.

5.2. Aufbau dünnbesetzter Peano-Matrizen

Die Verwaltung der Peano-Matrix in einem Feld von Blöcken und einem Blockbelegungsbaum führt zu Problemen beim Hinzufügen neuer Elemente. Dies betrifft insbesondere den Aufbau von dünnbesetzten Matrizen. Es ist zunächst unklar, wie beispielsweise aus eingelesenen Matrixelementen der Form (Reihe, Zeile, Wert) aus einer Datei eine Peano-Matrix konstruiert werden kann. Die Folge der Matrixelemente muss zunächst in die Peano-Nummerierung transformiert werden. Dazu wird die Matrix-Dimension benötigt, aus der die Größen b_P, n_P und m_P, wie in Abschnitt-2.5 beschrieben, gewonnen werden können. Mit der Kenntnis dieser Werte lässt sich mittels Tiefensuche (vgl. Algorithmus-2.3) ein Baum konstruieren, der in seinen Blättern die zugehörigen Elemente enthält. Dieser Zwischenschritt ermöglicht es, die nötigen Informationen für weitere Schritte, wie die noch folgende Blockgenerierung, bereitzustellen. Denn die Datenstrukturen in den Peano-Blöcken benötigen, abhängig von der Anzahl und Anordnung ihrer Elemente, unterschiedlichen Speicherbedarf. Diese Informationen müssen bereitstehen, um die Größe des Feldes der Peano-Blöcke allokieren zu können.

5.2.1. Speicherorganisation

Im Folgenden werden Möglichkeiten beschrieben, die sich für die Implementierung einer Peano-Matrix für dünnbesetzte Block-Matrizen eignen. Grund dafür sind die verschieden großen Peano-Blöcke und die daraus resultierende Problematik, einen direkten (Feld-) Zugriff auf die Blöcke zu ermöglichen. Für dichtbesetzte Blöcke bietet sich auf natürliche Weise ein Feld an, da die Blöcke uniforme Größe besitzen.

P_0	P_1	P_2	...	P_{N-1}	P_N

Im Fall der BCRS-Datenstruktur hängt die größe des Peano-Blocks unter anderem von der Anzahl der Nicht-Null Blöcke ab. Dies führt zu verschiedenen Größen und lässt sich nicht direkt mit einem Feld realisieren. Dieses Problem lässt sich typischerweise mit den folgenden Varianten lösen.

1. **Indirekte Addressierung**: Dabei werden Zeiger auf die jeweiligen Blöcke in einem zusätzlichen Feld verwaltet. Dieses Feld und die Blockdaten werden in einem Datenstom allokiert. Der Zugriff auf einen beliebigen Block ist damit durch zwei Dereferenzierungen möglich, also in $\mathcal{O}(1)$.
2. **Verkettete Liste**: Jeder Block besitzt Zeiger für Vorgänger und/oder Nachfolger innerhalb des Datenstroms. Der Zugriff auf einen beliebigen Block erfordert die Iteration über das Blockarray, benötigt also $\mathcal{O}(N)$. In vielen Algorithmen werden die Blöcke sequenziell verarbeitet, dabei kann der Zugriff wiederum in $\mathcal{O}(1)$ geschehen.

Aufgrund der Allgemeinheit der Variante 1, wurde diese für die Implementierung gewählt.

5.2.2. Generierung von Blockpartitionen

Die Generierung von Blockpartitionen ist nötig, falls die Blockmatrix aus einer Folge von Elementen, beispielsweise aus einer Datei, erzeugt werden soll. Im Fall der BCRS-Datenstruktur geschieht dies mittels einer uniformen Blockpartition mit der gegeben Blockgröße. Sind beliebige Blockgrößen zugelassen, so ist es nötig, eine möglichst optimale Blockstruktur aus der Besetzungstruktur zu gewinnen. Optimal bedeutet hier, dass einerseits möglichst wenige Null-Elemente gespeichert und andererseits möglichst wenige große Blöcke erzeugt werden sollen.

Die als Blockmatrix implementierte BCRS-Datenstruktur wurde bereits in 3.2.1 genauer beschrieben. Für die Implementierung war die genaue Bestimmung des benötigten Speichers von besonderer Bedeutung.

6. Experimentelle Ergebnisse

In diesem Kapitel werden die verschiedenen Algorithmen bezüglich ihrer Leistung miteinander verglichen. Dabei werden Laufzeitmessungen der implementierten Algorithmen einzeln, Matrix-Vektor-Multiplikation und ILU-Zerlegung, sowie im Kontext des Lösers durchgeführt. Die Ergebnisse werden unter anderem mit der *Math Kernel Library 10.3* (MKL) von Intel und dem bereits bestehenden, auf der CRS-Datenstruktur basierenden TifaMMyHybrid, verglichen.

Alle Messungen werden exemplarisch mit dem Datentyp *double* durchgeführt. Wenn die Anzahl der verwendeten Threads nicht ausgezeichnet ist, wurde implizit nur ein Thread verwendet.

Die MKL unterstützt verschieden Varianten der BCRS-Datenstruktur (vgl. Abschnitt-3.2.1). Diese spalten sich in zwei Arten auf, zum einen die *row-major* und zum andern die *column-major* Variante. Um möglichst gleiche Bedingungen zu schaffen, wurde analog zu allen anderen Datenstrukturen die *row-major* Variante vorgezogen. Unabhängig von dieser Entscheidung, gibt es zwei weitere Varianten der Verwaltung der Zeilenindizes. Die MKL unterstützt eine Variante mit drei Feldern, analog zur BCRS-Datenstruktur, sowie eine Variante mit vier Feldern, wobei in dem zusätzlichen Feld die Zeilenenden gespeichert werden. Für die Tests wurde die drei-Felder Variante gewählt. Die dazugehörige Methode für den Datentyp *double* ist die Bibliotheksfunktion `mkl_cspblas_dbsrgemv`.

Testdatensätze

Als Datensatz werden zwei, aus der Diskretisierung von Poisson-Problemen entstandene und mittels der Partition of Unity Methode diskretisierte, Gleichungssysteme verwendet. Details zu den einzelnen Datensätzen sind in Tabelle-6.1 und Tabelle-6.2 aufgeführt.

Level	Matrix Dimension	# Blöcke	# Elemente	MVP FLOP
1	12 x 12	16	144	288
2	48 x 48	100	900	1800
3	192 x 192	484	4356	7992
4	768 x 768	2116	19044	32922
5	3072 x 3072	8836	79518	132786
6	12288 x 12288	36100	324758	532728
7	49152 x 49152	145924	1312570	2133288
8	196608 x 196608	586756	5278624	8537310
9	786432 x 786432	2353156	21169184	34156782
10	3145728 x 3145728	9424900	84800560	136641636

Tabelle 6.1.: PUM Testdatensatz mit 3×3 - Blockgröße

Level	Matrix Dimension	# Blöcke	# Elemente	MVP FLOP
1	16 x 16	16	254	512
2	64 x 64	100	1600	3200
3	256 x 256	484	7732	14176
4	1024 x 1024	2116	33762	57184
5	4096 x 4096	8836	141198	229792
6	16384 x 16384	36100	577076	921376
7	65536 x 65536	145924	2332764	3690016
8	262144 x 262144	586756	9383526	14769184
9	1048576 x 1048576	2353156	37635424	59095072
10	4194304 x 4194304	9424900	150731872	236417056

Tabelle 6.2.: PUM Testdatensatz mit 4×4 - Blockgröße

Testsysteme

Die verwendeten Testsysteme sind im folgenden gelistet

1. Intel Core-i7 2600K, 8 MB Cache, 3.8 GHz, 4 Kerne, 16 GB RAM
2. Intel Xeon E7540, 18 MB Cache, 2.00 GHz, 24 Kerne, 6.40 GT/s Intel QPI, 512 GB RAM

Parameter

Die Tests wurden ausschließlich auf Linux-basierten System durchgeführt. Als Compiler wurde der Intel Compiler (Version 12.1.3) verwendet. Die verwendeten Compiler-Optionen waren bei allen Test

```
-O3 -ip -ipo -fno-builtin -fp-model fast -funroll-all-loops
-unroll-aggressive -opt-prefetch
```

Das Core-i7 2600K System unterstützt die AVX-Erweiterung. Für die Tests auf diesem System wurden die zusätzlichen Optionen `-xAVX -mavx` gewählt, um die Erweiterung einzuschalten.

6.1. Matrix-Vektor-Multiplikation

In diesem Abschnitt wird die Matrix-Vektor-Multiplikation im einzelnen betrachtet. Bei den Tests werden jeweils die Gleichungssysteme der beiden Testdatensätze verwendet.

6.1.1. Bestimmung der Peano-Blockgrößen

Die entscheidende Größe für Leistungsfähigkeit der Matrix-Vektor-Multiplikation ist die Peano-Blockgröße. Im Abschnitt-2.5.1 wurden bereits theoretische Blockgrößen, auf Grundlage von dichtbesetzten Peano-Blöcken, diskutiert. Die hier verwendeten dünnbesetzten Matrizen zeigen, dass Werte, die um ein Vielfaches größer sind als die Größe des L1-Daten-Cache, hier entscheidende Leistungsvorteile liefern. Ein Grund dafür ist die sehr verschiedene Besetzung der Peano-Blöcke mit Nicht-Null Blöcken. Viele dünnbesetzte Peano-Blöcke führen zu einer größeren Laufzeit des Algorithmus, wohingehen bei Inkaufnahme von Blockgrößen oberhalb der Größe des Cache, zusätzliche Cache-Verfehlungen für dichtbesetzte Peano-Blöcke auftreten können. Ferner kann dadurch auch die temporäre Lokalität des Algorithmus vermindert werden. Die Frage nach der "optimalen" Blockgröße ist letzlich ein Ausgleich, der über Messungen der benötigten Rechenzeit bestimmt werden muss.

Die folgenden Abbildungen zeigen die Laufzeit für eine Matrix-Vektor-Multiplikation als prozentuale Differenz der Laufzeit der MKL. Dabei wurden für die beiden Datensätze nur jeweils die Matrizen größerer Dimension, Level 7 bis Level 10, auf dem Intel i7-System betrachtet.

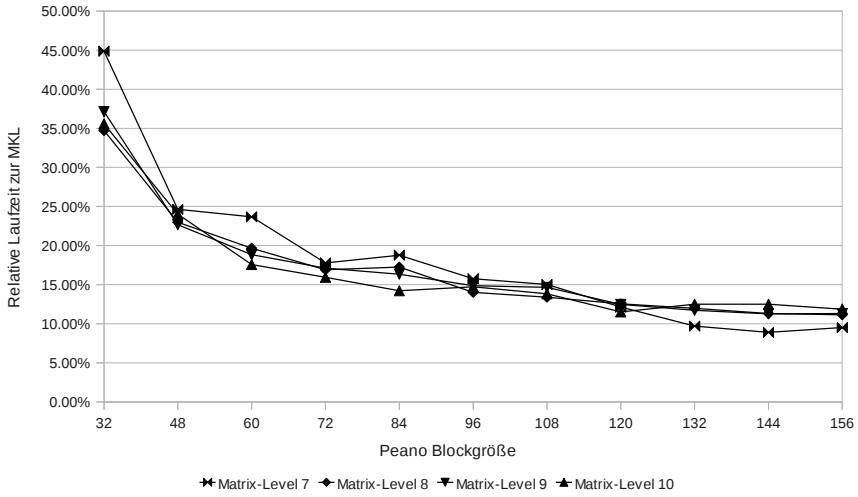

Abbildung 6.1.: Relatives Laufzeitverhalten zur MKL (Intel Code-i7 2600K) mit 3×3 - Blöcke, Matrix Level7 - Level10

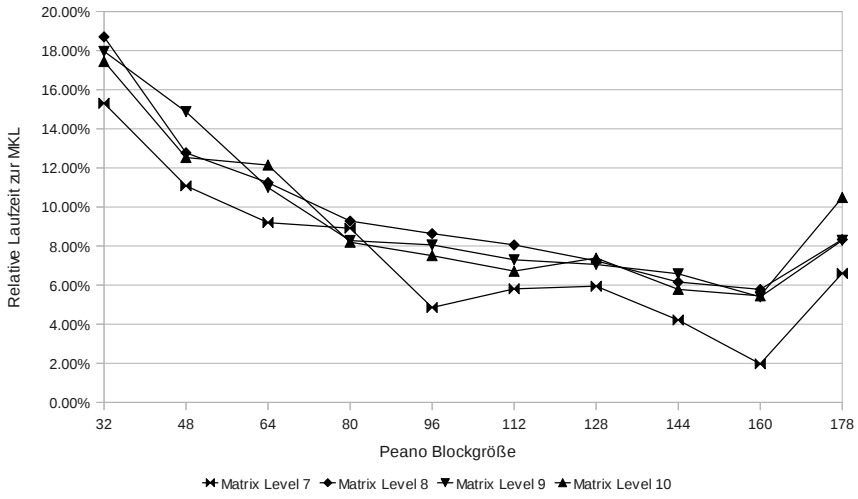

Abbildung 6.2.: Relatives Laufzeitverhalten zur MKL (Intel Code-i7 2600K) mit 4×4 - Blöcke, Matrix Level7 - Level10

Abbildung-6.1 bezieht sich auf die 3×3-Blockmatrizen. Man kann deutlich erkennen, dass Blockgrößen $b_P > 144$ zu keinen weiteren Verbesserungen führen. Im Fall der 4×4-Blockmatrizen ist ein anderes Verhalten zu erkennen. Für Blockgrößen $b_P > 160$ werden die Laufzeiten hingegen schlechter.

Für den weiteren Verlauf dieses Kapitels werden die Werte $b_P = 144$ (Datensatz 1) und $b_P = 160$ (Datensatz 2) festgehalten.

6.1.2. Leistungsvergleich

Mit den zuvor festgestellten Blockgrößen werden im Folgenden die Leistung der unterschiedlichen Datenstrukturen miteinander verglichen. Um die Leistungsfähigkeit der entwickelten Datenstrukturen zu ermitteln, werden diese mit der MKL und der CRS-basierten Implementierung (TifaMMyHybrid) verglichen. Insbesondere werden hier die dynamische und statische Variante der BCRS-Datenstruktur verwendet. Als Besetzungsfaktor der dynamischen Blockdatenstruktur wurde für diesen Test 0.5 und 0.75 gewählt.

Die Ergebnisse in Abbildung-6.4 zeigen, dass keine BCRS-Variante an die Leistungsfähigkeit der MKL heranreicht. Die hier verwendeten Matrizen bestehen aus 3×3-Blöcken, für die aufgrund der Speicherausrichtung, in der bestehenden Implementierung keine Autovektorisierung vorgenommen wurde. Das Leistungspektrum der BCRS-Varianten ist zwischen $50\% - 100\%$ über dem der CRS-Variante. Auffällig sind ebenfalls die Leistungseinbrüche aller Verfahren bei Matrizen ab der Größenordnung 65536×65536 (Level 7).

Im Vergleich der beiden Datensätze zeichnet sich das Potential der Autovektorisierung deutlich ab. Die statische BCRS-Variante profitiert von der 4×4-Blockgröße, die der Compiler offensichtlich optimal vektorisieren konnte. Die Ergebnisse der dynamische Variante zeigen allerdings diesen Zuwachs nicht auf. Es muss also deutlich mehr Aufwand betrieben werden, die dynamische auf das Leistungsniveau der statischen Block-Datenstruktur zu bringen. Dazu müssen für Spezialfälle optimierte Block-Routinen (Kernel) entwickelt werden, die während der Laufzeit anhand der vorliegende Blockgröße dynamisch ausgewählt werden.

Ein besonders schönes Ergebnis liefert der Vergleich der MKL mit der statischen BCRS-Datenstruktur auf dem Intel Xeon E7540 System. Die hier vorgestellte Implementierung zeigt in Abbildung-6.6 unter Verwendung eines Kerns, auf jedem Matrix-Level eine bessere Lesitung als die MKL.

Die nachfolgenden Abbildungen zeigen die erziehlten Leistungen gemessen in FLOP/s [1].

[1] FLOP steht für *floating-point operation*. Die Messung der Leistungsfähigkeit von Systemen wird häufig in der Form FLOP pro Sekunde (FLOP/s oder FLOPs) angegeben.

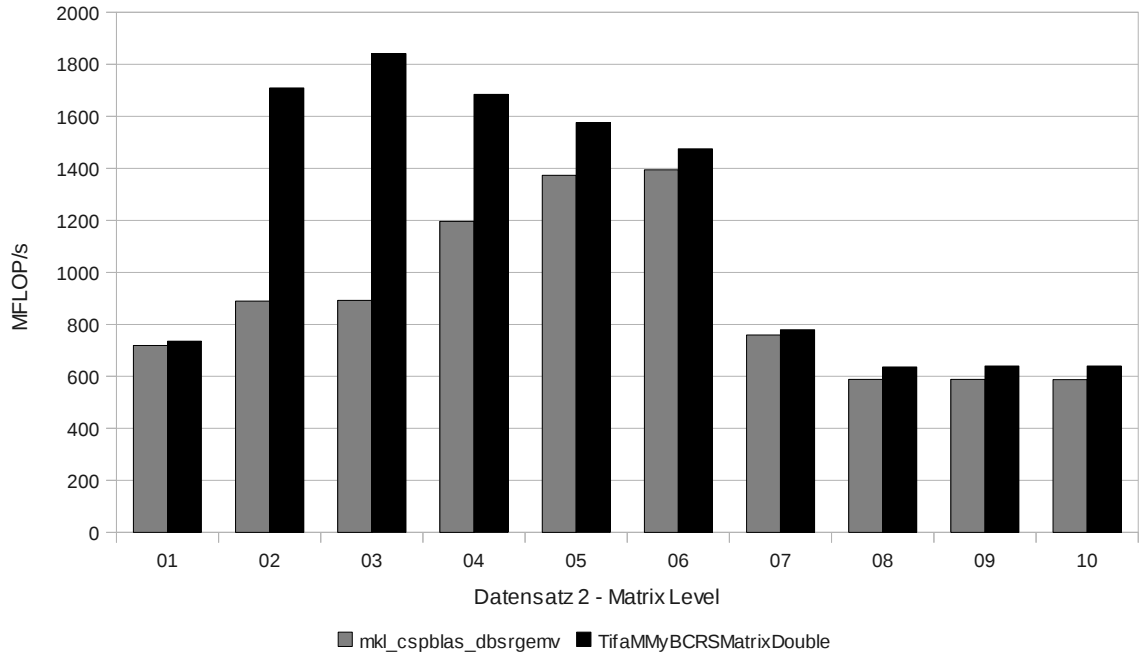

Abbildung 6.3.: Matrix-Vektor-Multiplikation im Vergleich: statische BCRS-Datenstruktur und Intel MKL, Intel Xeon E7540, 1 Thread

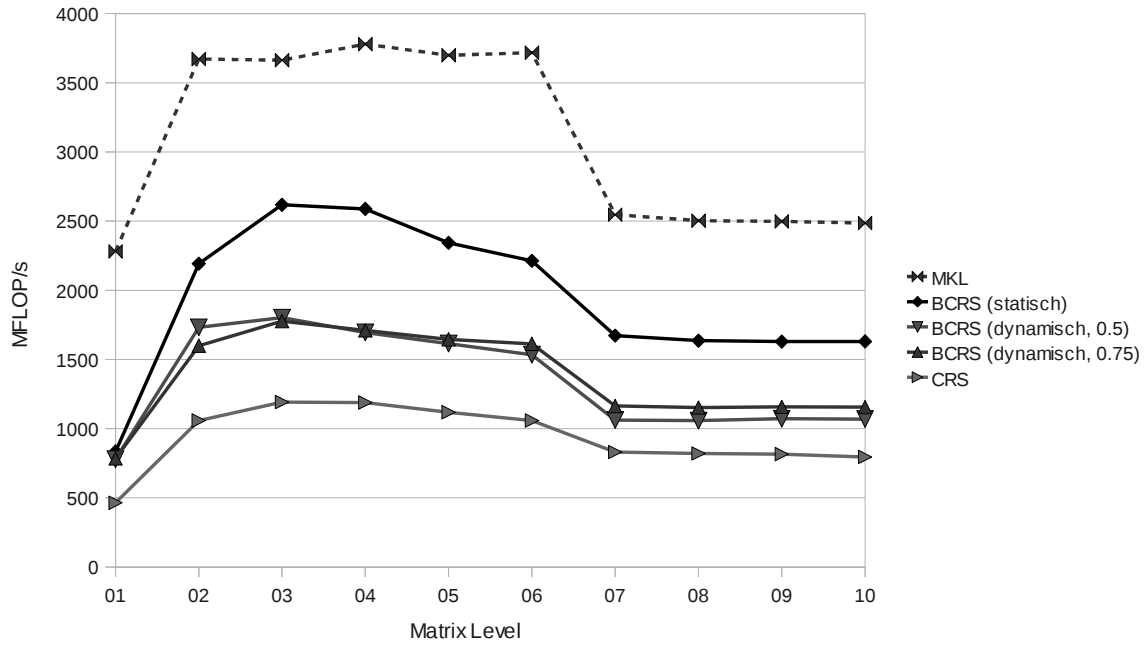

Abbildung 6.4.: Matrix-Vektor-Multiplikation im Vergleich: BCRS-Datenstrukturen, MKL und TifaMMyHybrid; Intel Core-i7 2600K mit Datensatz 1

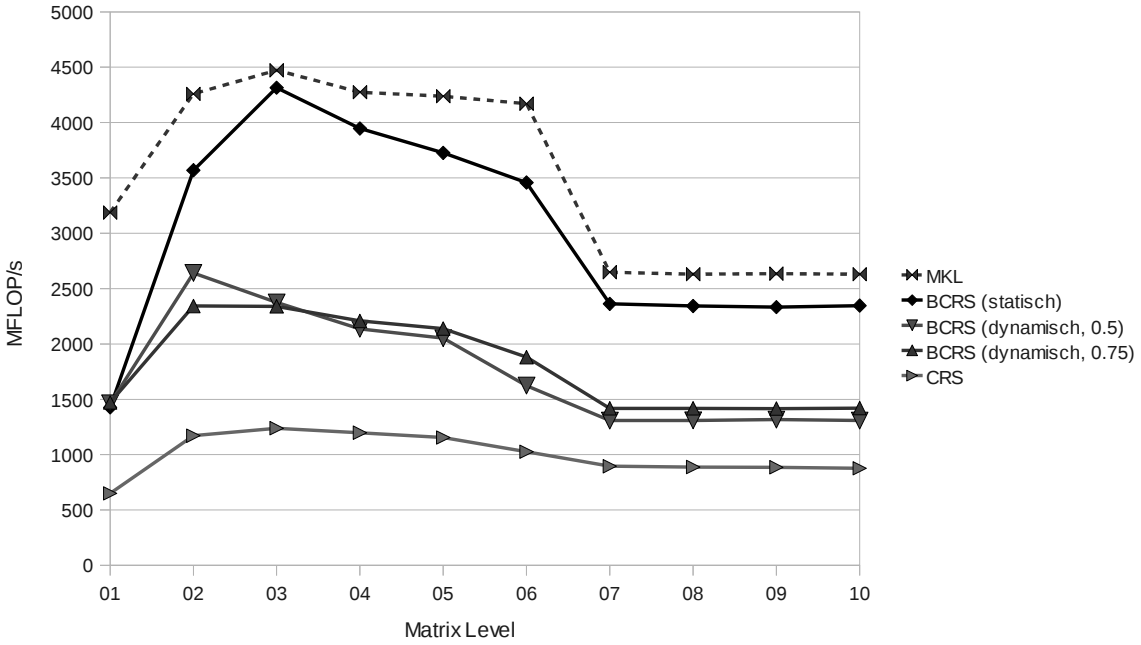

Abbildung 6.5.: Matrix-Vektor-Multiplikation im Vergleich: BCRS-Datenstrukturen, MKL und TifaMMyHybrid; Intel Core-i7 2600K mit Datensatz 2

Die dynamische Block-Datenstruktur erwies sich als schlechter optimiert als sein statisches Pendant. Die verwendeten Compiler-Direktiven bewirken für dynamische Blockgrößen nicht die gleiche Optimierung, obwohl die jeweiligen Block-Partitionen auf Mindestblockgrößen der statischen Variante zurückgreifen. Das heißt genauer, dass die generierte Blockpartition im schlechtesten Fall mit der uniformen Blockstruktur übereinstimmt. Damit müssten zumindest ähnliche Werte entstehen können, die aber nicht von den Messungen bestätigt werden.

6.1.3. Parallelität

Im Folgenden wird die Parallelität der statischen BCRS-Datenstruktur bezüglich der Matrix-Vektor-Multiplikation verglichen. Die Messungen wurden auf dem Intel Xeon E7540 System mit beiden Datensätzen durchgeführt. Dabei wurden 2 bis 16 Threads verwendet. Der Vergleich findet nur zwischen der MKL und der statischen BCRS-Datenstruktur statt.

Die Ergbenisse des Vergleichs zeigen eine deutliche bessere Leistung der MKL unter Verwendung von mehreren Threads, wohingegen für einen Thread die BCRS-Datensturktur leistungsfähiger war. Allerdings gelten für zunehmend größere Matrizen und unter Verwendung mehrerer Threads, eine Leistungsdifferenz der BCRS-Datenstruktur von etwa 10% gegenüber der MKL.

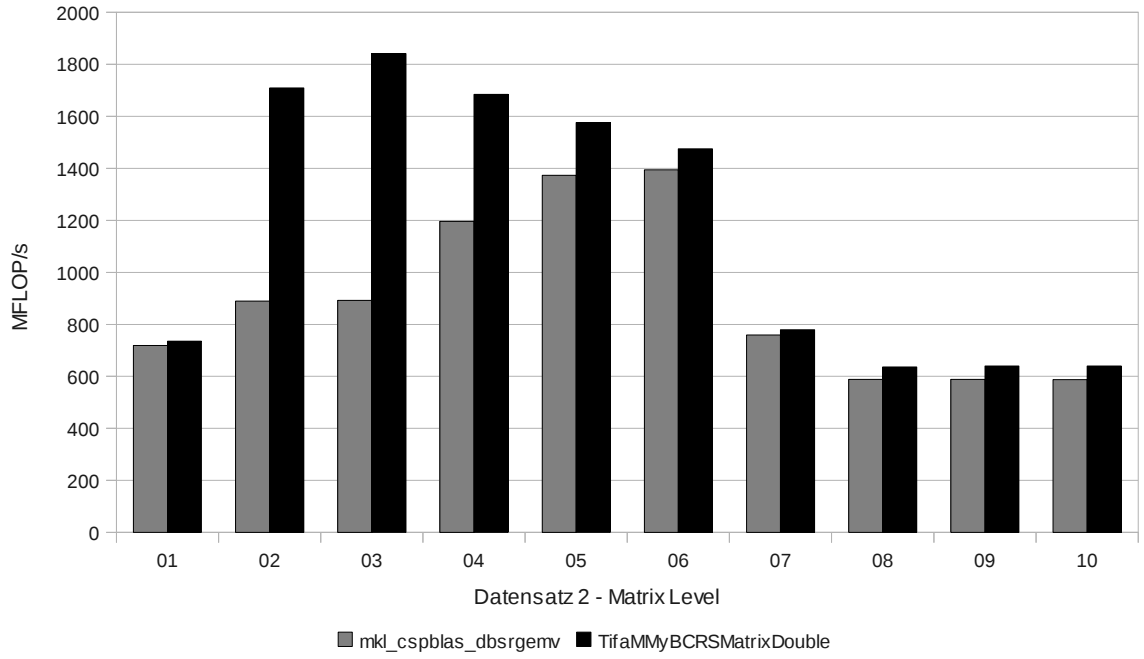

Abbildung 6.6.: Parallelität im Vergleich zur MKL, Intel Xeon E7540, 2 Threads

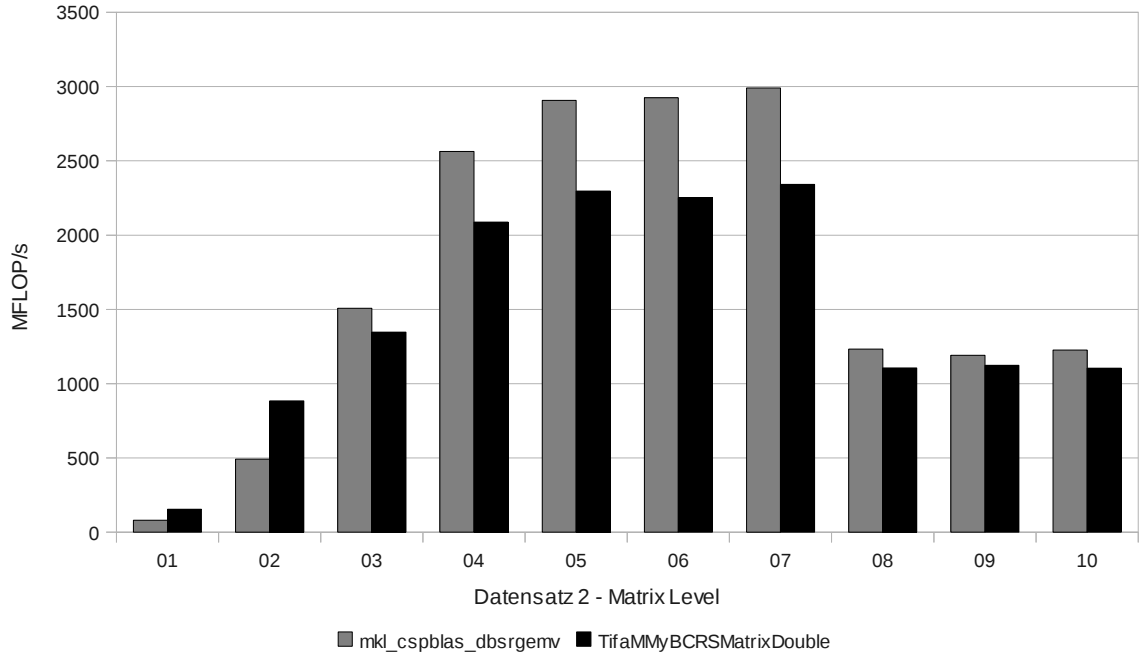

Abbildung 6.7.: Parallelität im Vergleich zur MKL, Intel Xeon E7540, 2 Threads

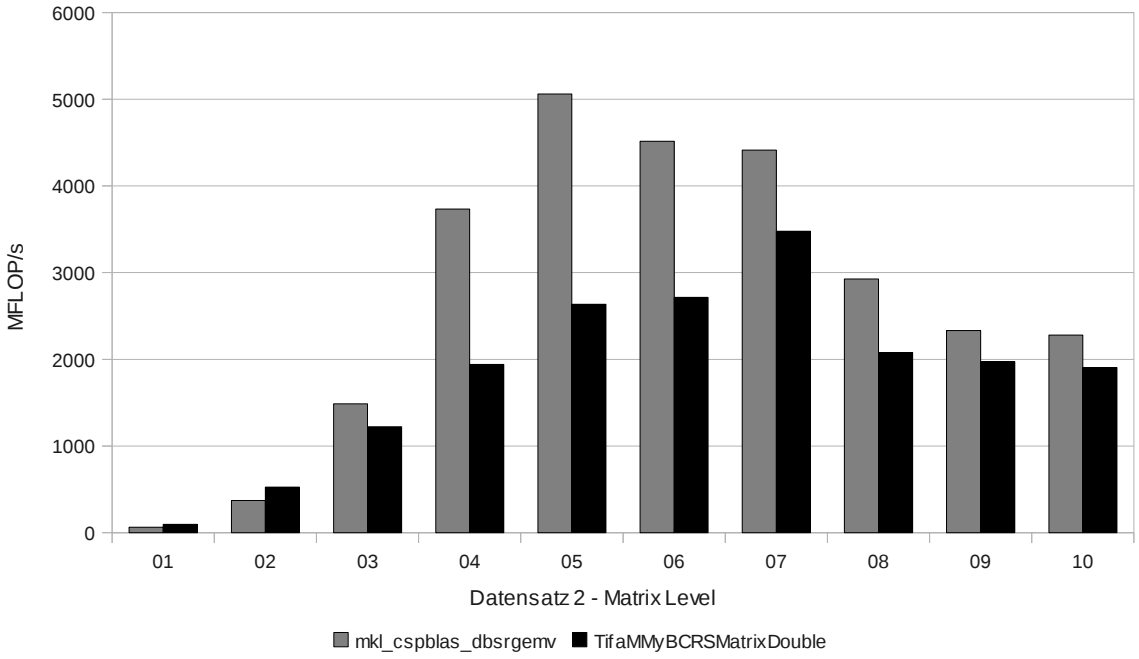

Abbildung 6.8.: Parallelität im Vergleich zur MKL, Intel Xeon E7540, 4 Threads

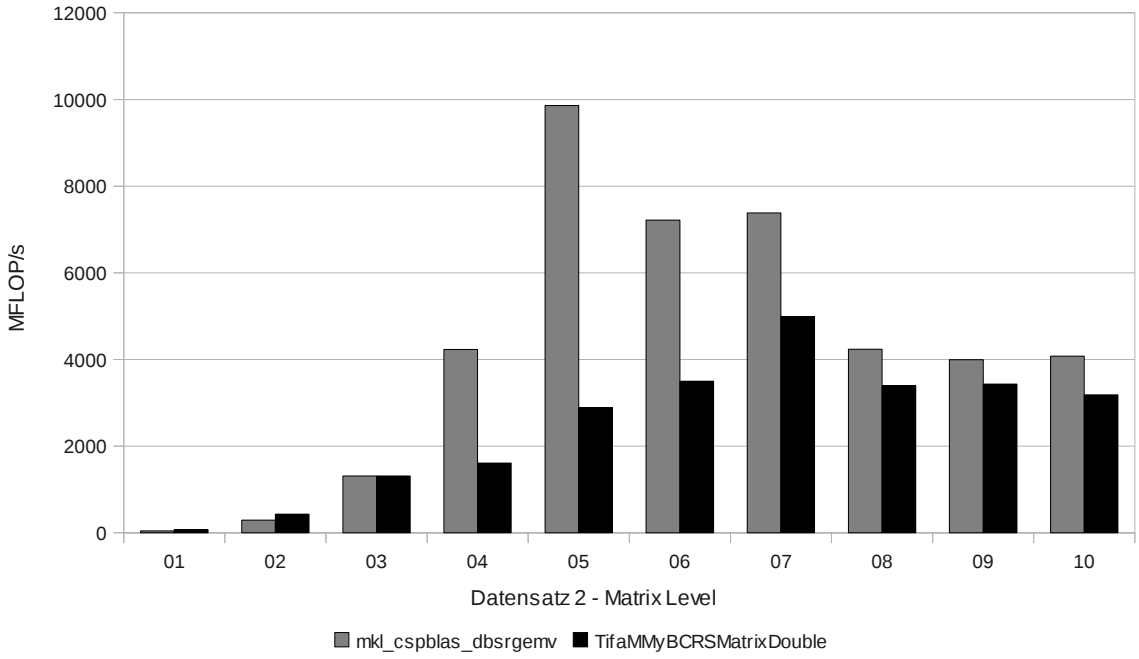

Abbildung 6.9.: Parallelität im Vergleich zur MKL, Intel Xeon E7540, 8 Threads

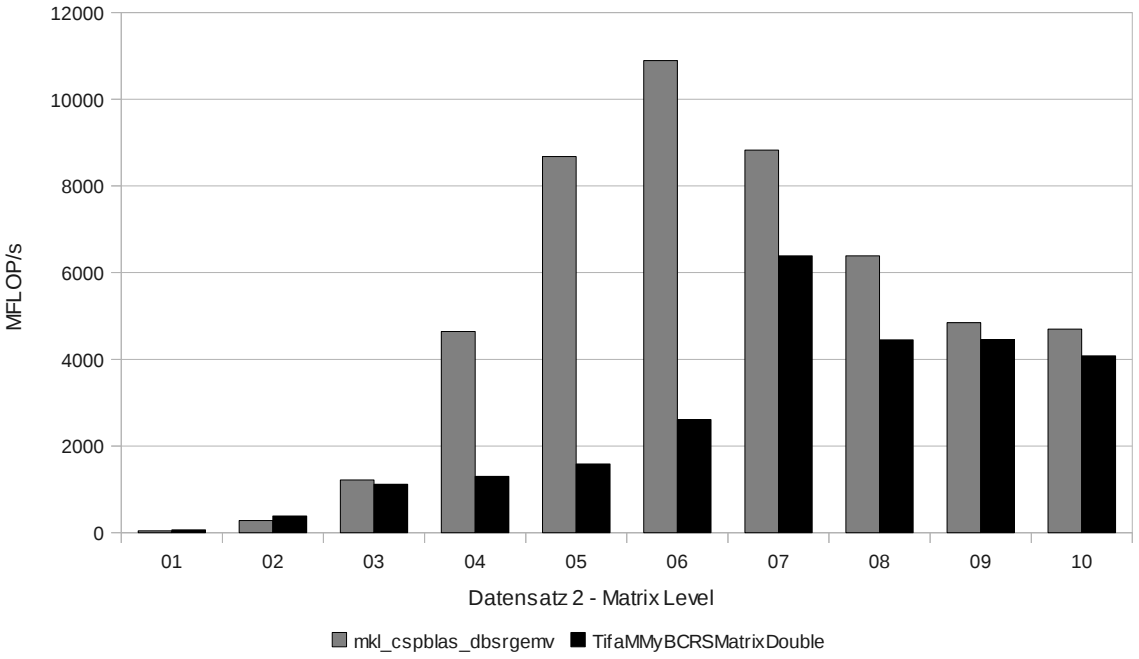

Abbildung 6.10.: Parallelität im Vergleich zur MKL, Intel Xeon E7540, 16 Threads

6.2. ILU-Zerlegung

In diesem Abschnitt wird die in dieser Arbeit vorgestellte ILU-Zerlegung (BCRS), mit der bereits in TifaMMy implementierten (CRS) Datenstruktur verglichen (TifaMMyHybrid). Damit dieser Vergleich sinnvoll ist, wurde der Test für die BCRS-Datenstruktur zusätzlich mit Blockgröße $b = 1$ durchgeführt.

6.2.1. Leistungsvergleich

In den Tabellen 6.3 und 6.4 sind die Laufzeiten für die Berechnung einer ILU-Zerlegung aller Operatormatrizen der beiden Datensätze gelistet.

	TifaMMyBCRS($b = 3$)	TifaMMyBCRS($b = 1$)	TifaMMyHybrid
Level 1	< 1 ms	< 1 ms	< 1 ms
Level 2	< 1 ms	< 1 ms	< 1 ms
Level 3	< 1 ms	4 ms	3 ms
Level 4	< 1 ms	12 ms	10 ms
Level 5	3 ms	31 ms	47 ms
Level 6	6 ms	128 ms	203 ms
Level 7	21 ms	520 ms	840 ms
Level 8	84 ms	2095 ms	3426 ms
Level 9	339 ms	8402 ms	13806 ms
Level 10	1364 ms	33672 ms	55183 ms

Tabelle 6.3.: Laufzeit für die Berechung einer ILU-Zerlegung der Operatoren von Datensatz 1

	TifaMMyBCRS($b = 4$)	TifaMMyBCRS($b = 1$)	TifaMMyHybrid
Level 1	< 1 ms	< 1 ms	< 1 ms
Level 2	< 1 ms	1 ms	1 ms
Level 3	< 1 ms	9 ms	7 ms
Level 4	1 ms	19 ms	34 ms
Level 5	4 ms	82 ms	152 ms
Level 6	6 ms	337 ms	656 ms
Level 7	27 ms	1362 ms	2692 ms
Level 8	111 ms	5491 ms	10925 ms
Level 9	447 ms	22000 ms	43962 ms
Level 10	1800 ms	97463 ms	175455 ms

Tabelle 6.4.: Laufzeit für die Berechung einer ILU-Zerlegung der Operatoren von Datensatz 2

Die Ergebnisse in 6.3 und 6.4 zeigen, dass sich die ILU-Zerlegung für Block-Matrizen effizient mit der hier vorgestellten Implementierung berechnen lassen. Verwunderlich erscheint die Tatsache, dass die bestehende Implementierung ca. 75% langsamer ist als die BCRS-Datenstruktur mit Blockgröße $b = 1$.

6.2.2. Parallelität

Zuletzt werden noch die Laufzeitergebnisse zur Parallelisierung der ILU-Zerlegung vorgestellt. Die folgenden Abbildungen zeigen ein negatives Verhalten auf die Laufzeit unter Verwendung von OpenMP Tasks zur Parallelisierung des rekursiven

Algorithmus. Dieser Effekt geht auf die Lastverteilung der Threads zurück. Die Entwicklung einer Lastverteilung für die ILU-Zerlegung ist im Allgemeinen keine leichte Aufgabe.

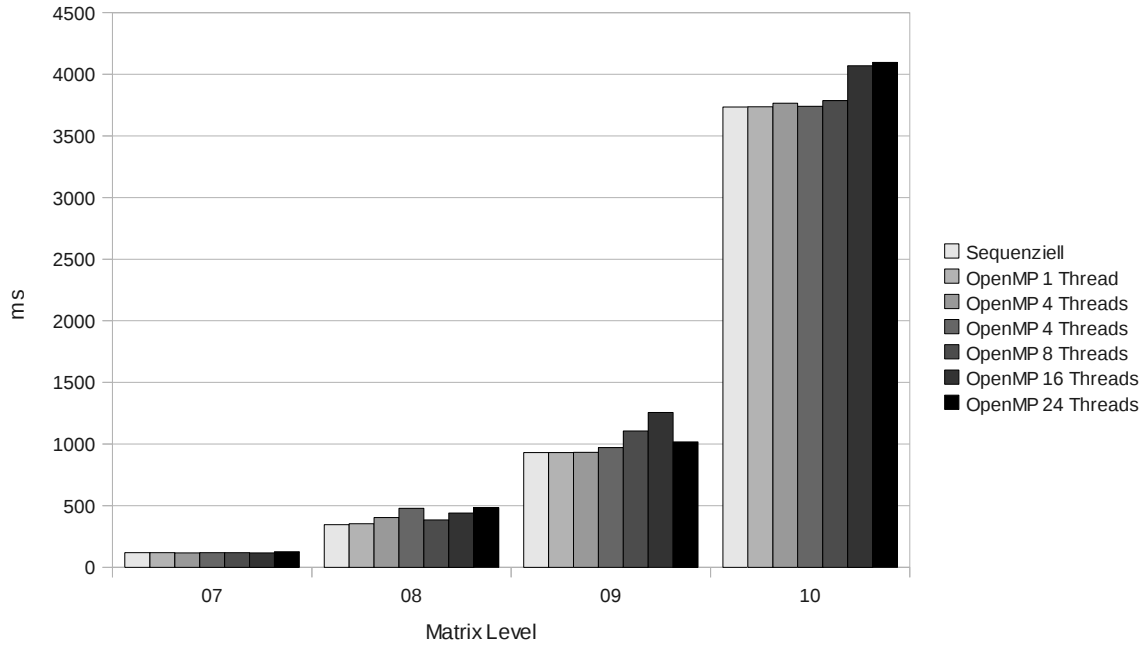

Abbildung 6.11.: Parallelität der ILU-Zerlegung im Vergleich zur MKL, Intel Xeon E7540, 1 - 24 Threads

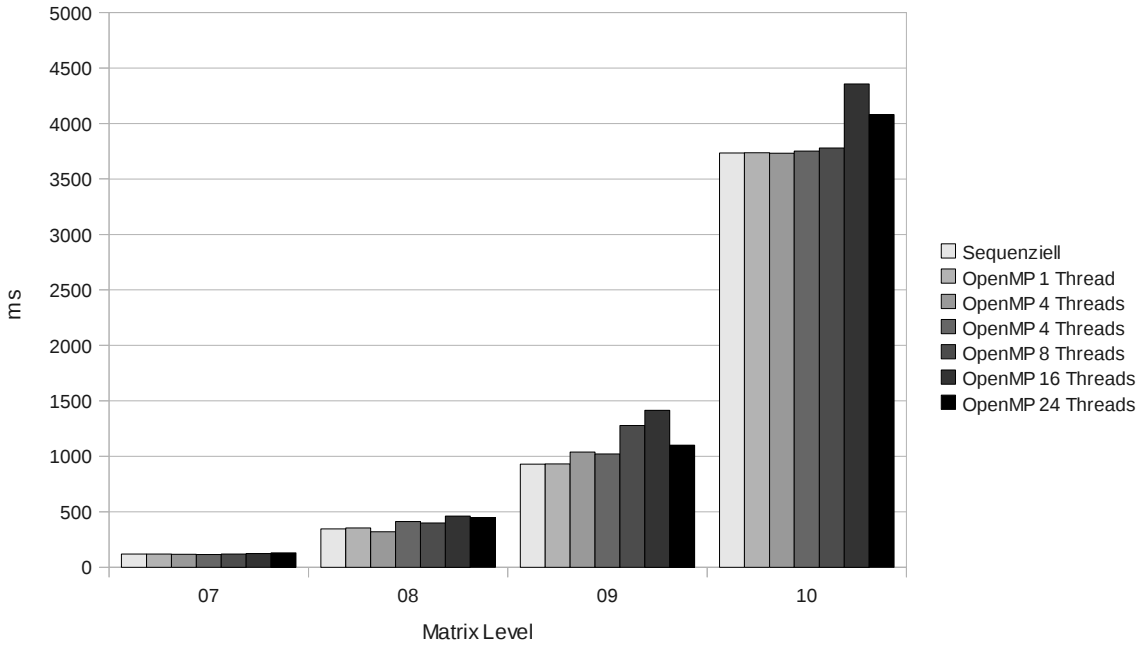

Abbildung 6.12.: Parallelität der ILU-Zerlegung im Vergleich zur MKL, Intel Xeon E7540, 1 - 24 Threads

6.3. Block-Matrix-Löser

Aufgrund der schlechten Parallelisierbarkeit der ILU-Zerlegung und der Vorwärts-Rückwärts-Substitution wurden jeweils die sequenziellen Varianten dieser Algorithmen im Löser verwendet. Das ILU-vorkonditionierte CG-Verfahren (ILUPreCondCG) mit parallellisierter Matrix-Vektor-Multiplikation wird in Konkurrenz mit der Multilevel Partition of Unity Methode verglichen. Um die Wirkung der Parallelelisierung im Kontrast betrachten zu können wird die Leistung mit einem Thread und 24 Threads durchgeführt.

	BCRS-ILUPreCondCG		PUM	
	Iterationen	Laufzeit	Iterationen	Laufzeit
Level 1	1	< 1 ms	1	< 1 ms
Level 2	8	< 1 ms	9	< 1 ms
Level 3	14	1 ms	11	3 ms
Level 4	24	6 ms	11	33 ms
Level 5	43	47 ms	12	72 ms
Level 6	81	350 ms	12	300 ms
Level 7	155	1987 ms	12	2888 ms
Level 8	290	18 s	12	11 s
Level 9	532	160 s	12	49 s
Level 10	1035	1152 s	12	270 s

Tabelle 6.5.: Vergleich der Iterationen und Laufzeiten der ILUPreCondCG (1 Thread) und der PUM

	BCRS-ILUPreCondCG		PUM	
	Iterationen	Laufzeit	Iterationen	Laufzeit
Level 1	1	< 1 ms	1	< 1 ms
Level 2	8	< 1 ms	9	< 1 ms
Level 3	14	1 ms	11	3 ms
Level 4	24	6 ms	11	33 ms
Level 5	43	38 ms	12	72 ms
Level 6	81	276 ms	12	300 ms
Level 7	155	1388 ms	12	2888 ms
Level 8	290	15 s	12	11 s
Level 9	532	118 s	12	49 s
Level 10	1035	931 s	12	270 s

Tabelle 6.6.: Vergleich der Iterationen und Laufzeiten der ILUPreCondCG (24 Threads) und der PUM

Die Ergebnisse diese Vergleichs zeigen, dass die ILU-Vorkonditionierung nicht mit den, asymptotisch optimalen, Mehrgitterverfahren der Multilevel Partition of Unity Methode konkurrieren kann. Die Parallelisierung der Matrix-Vektor-Vektor-Multiplikation unter Verwendung der 24 Threads liefert zwar eine Leistungssteigerung zwischen $20\% - 30\%$, aber dies steht nicht in Relation mit den verwendeten Ressourcen.

Ansatzpunkte für Verbesserungen bestehen in der Wahl des Vorkonditionierers, um die Anzahl der nötigen Iterationen zu minimieren. Die in [CV94] vorgeschlagene (modified)ILU könnte die Anzahl der Iterationen verringern, wobei das grund-

sätzliche Problem der Parallelisierung unangetastet bleibt. Eine Alternative zum Vorkonditionierer stellen die approximativen Inversen dar. Damit wäre es möglich die parallelisierte Matrix-Vektor-Multiplikation anstelle der "beinahe" sequentiellen Vorwärts-Rückwärts-Substitution zu verwenden.

7. Zusammenfassung und Ausblick

In dieser Arbeit wurde ein auf dem ILU-vorkonditioniert CG-Verfahren basierenden Block-Matrix-Löser entwickelt. Die wesentlichen Bestandteile des Verfahrens, die Matrix-Vektor-Multiplikation und ILU-Zerlegung, wurden dabei separat untersucht.

Die Analyse der Leistungsfähigkeit der vorgestellten Matrix-Vektor-Multiplikation wurde mit statischer und dynamischer Blockgröße implementiert und getestet. Die Ergebnisse zeigen, dass die Kombination aus statisch bekannter Blockgröße und Autovektorisierung zu erheblichen Leistungszuwächsen führen kann.

Auf der Intel Xeon E7540 Plattform konnte, unter Verwendung eines Prozessor-Kerns, eine bessere Leistung über den gesamten Testdatensatz, als die MKL gezeigt werden. Die verwendete OpenMP-Parallelisierung erwies sich im Vergleich aber als weniger leistungsfähig als die MKL. Dies ist auf die Lastverteilung der erzeugten Tasks zurückzuführen. Hier beseteht ein Bedarf an Optimierung, um die Parallelität zu verbessern.

Die dynamische Block-Datenstruktur konnte im Gegensatz zur statischen BCRS-Datenstruktur nicht von der verwendeten Autovektorisierung profitieren. Um diese Datenstruktur effizienter zu gestalten, müssen für bestimmte Blockgrößen optimierte Kernel implementiert werden. Während der Laufzeit kann dann anhand der Blockgröße eine Fallunterscheidung für die Auswahl der jeweiligen Routine sorgen. Im Fall von uniformen Blöcken ist es möglich, diese Fallunterscheidung für die gesamte Matrix durchzuführen. Damit würde der zusätzliche Aufwand an Überprüfungen für die Blöcke entfallen.

Das ILU-vorkonditioniert CG-Verfahren leidet allerdings unter der schwer parallelisierbaren ILU-Zerlegung sowie dem ebenfalls nur schwer zu parallelsierenden Vorwärts-Rückwärts-Substitition. Die Berechnung der ILU-Zerlegung findet zwar nur einmal zu Beginn des Verfahrens statt, aber die Vorkonditionierung der Residuen erfordert in jeder Iteration eine Vorwärts-Rückwärts-Substitution mit der ILU-Systemmatrix. Darüberhinaus zeigt sich, dass die ILU-Zerlegung als Vorkonditionierer für die hier verwendeten Gleichungssysteme deutlich zu viele Iterationen benötigt.

Zusammenfassend kann festgestellt werden, dass die Verwendung von Cache-effizienten Algorithmen auf Basis der Peano-Kurve zu einer durchaus Konkurrenzfähige Leistung zur MKL führt.

Danksagung

Ich danke Herrn Schweitzer und Herrn Bader für die Aufgabenstellung und ihre freundliche Unterstützung bei dieser Arbeit.

Desweiteren danke ich allen, die mich während meines Studiums unterstützt haben, insbesondere meinem Vater Andreas Gründer, meiner Freundin Katerine Basler und meiner lieben Mitbewohnerin Amelie Janetschek für die Unterstützung und überhaupt.

Meinen besonderen Dank geht an Simon Schmider für das Korrekturlesen der Ausarbeitung der Arbeit.

A. Anhang

A.1. Matrix-Vektor-Multiplikation Schemata

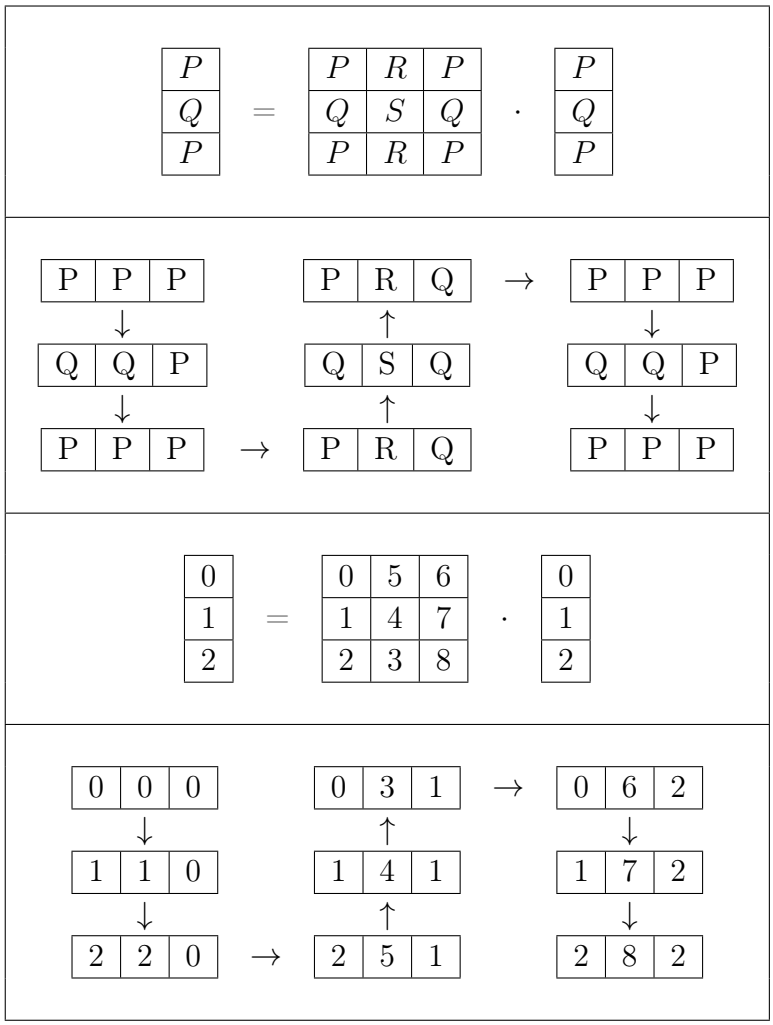

Abbildung A.1.: Peano Matrix-Vektor-Multiplikationsschema $P \mathrel{+}= P \cdot P$

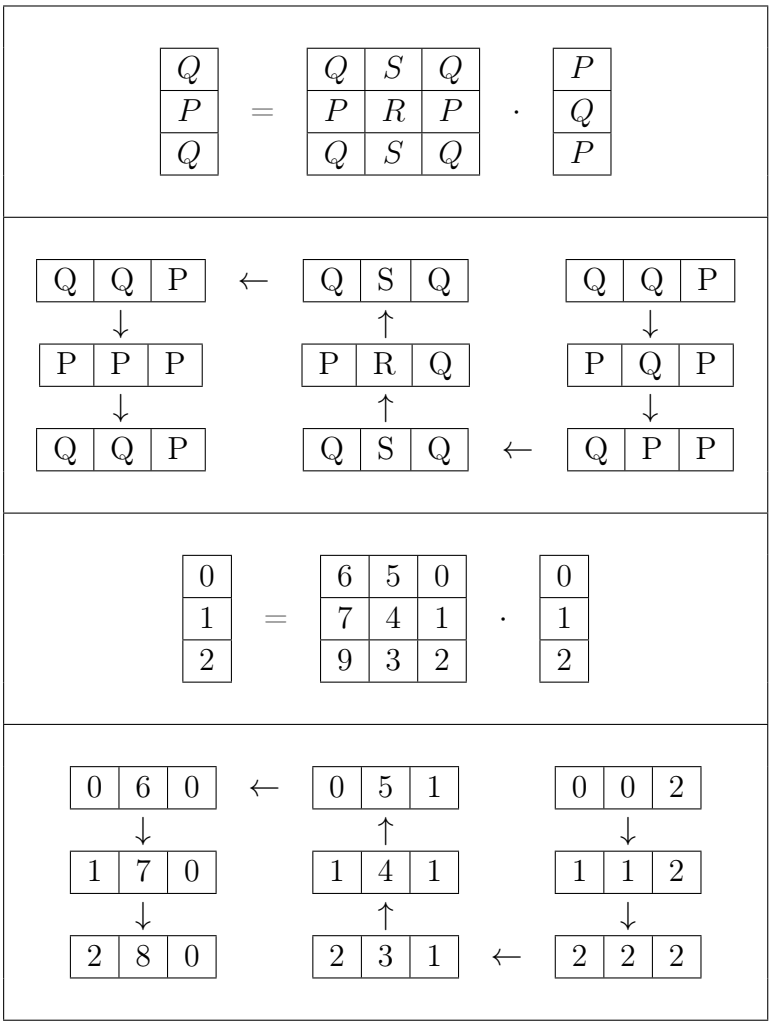

Abbildung A.2.: Peano Matrix-Vektor-Multiplikationsschema $Q \mathrel{+}= Q \cdot P$

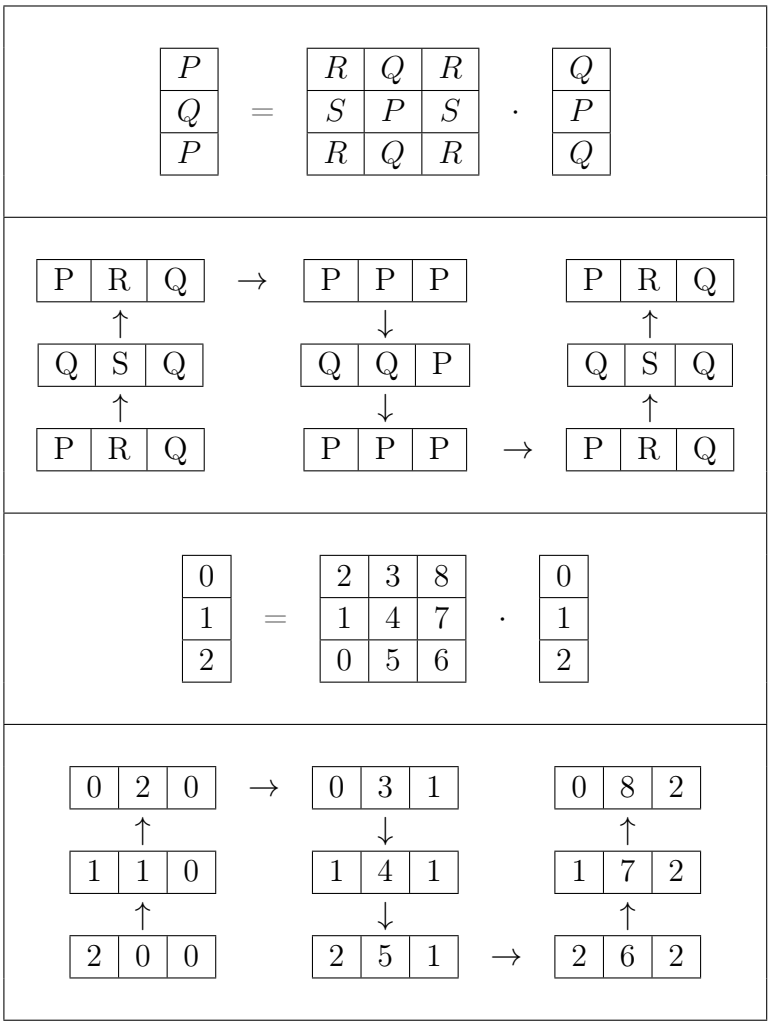

Abbildung A.3.: Peano Matrix-Vektor-Multiplikationsschema $P \mathrel{+}= R \cdot Q$

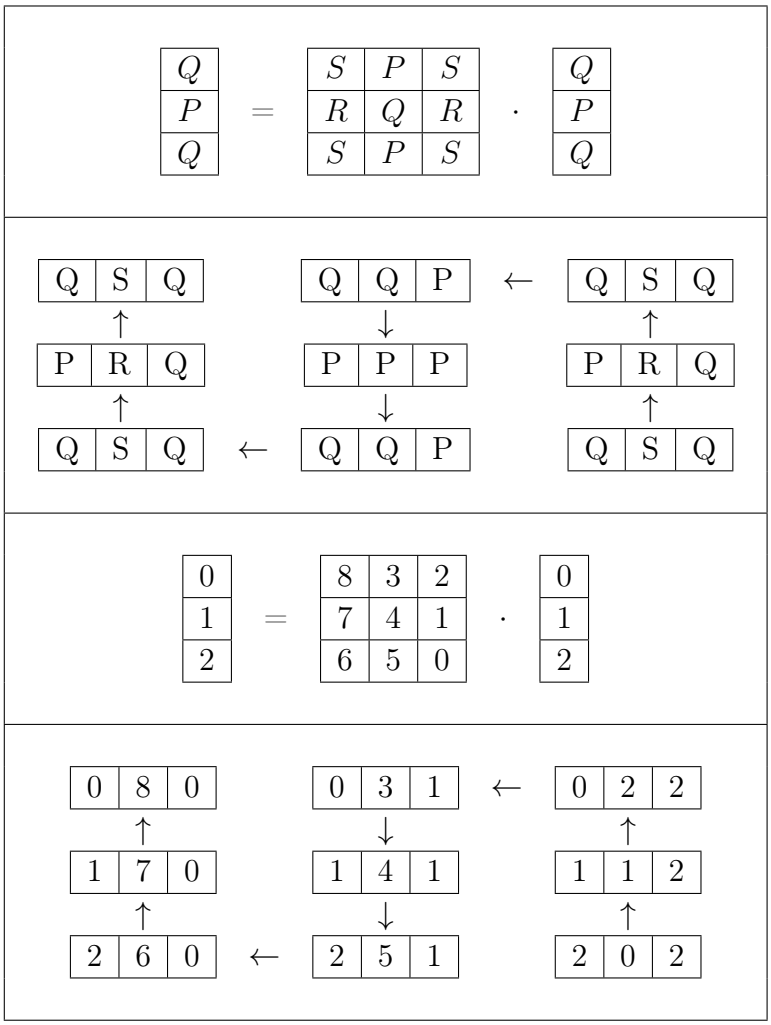

Abbildung A.4.: Peano Matrix-Vektor-Multiplikationsschema $Q \mathrel{+}= S \cdot Q$

A.2. Subroutinen der ILU-Zerlegung

Algorithmus A.1 LUDecomposition-Subroutine für den dichtbesetzten Peano-Block A

function LUDecomposition_Dense(A)
  for $i = 0 \ldots n_P - 1$ **do**
    for $j = 0 \ldots i - 1$ **do**
      for $k = 0 \ldots j - i$ **do**
        $A_{i,j} \leftarrow A_{i,j} - A_{i,k} \cdot A_{k,j}$
      end for
      $A_{i,j} \leftarrow \frac{A_{i,j}}{A_{j,j}}$
    end for
    for $j = i \ldots n_P - 1$ **do**
      for $k = 0 \ldots i - 1$ **do**
        $A_{i,j} \leftarrow A_{i,j} - A_{i,k} \cdot A_{k,j}$
      end for
    end for
  end for
end function

Algorithmus A.2 findLeft-Subroutine für die dichtbesetzte Peano-Blöcke A und B

function findLeft_Dense(A, B)
  for $i = 0 \ldots n_P - 1$ **do**
    for $j = 0 \ldots n_P - 1$ **do**
      for $k = 0 \ldots j - 1$ **do**
        $A_{i,j} \leftarrow A_{i,j} - A_{i,k} \cdot B_{k,j}$
      end for
      $A_{i,j} \leftarrow \frac{A_{i,j}}{B_{j,j}}$
    end for
  end for
end function

Algorithmus A.3 findRight-Subroutine für die dichtbesetzte Peano-Blöcke A und B

```
function findRight_Dense( A, B )
    for i = 1 ... n_P − 1 do
        for j = 0 ... n_P − 1 do
            for k = 0... i − 1 do
                B_{i,j} ← B_{i,j} − A_{i,k} · B_{k,j}
            end for
        end for
    end for
end function
```

Algorithmus A.4 findRight-Subroutine für die dichtbesetzte Peano-Blöcke A und B

```
function mulSub_Dense( C, A, B )
    for i = 1 ... n_P − 1 do
        for j = 0 ... n_P − 1 do
            for k = 0 ... n_P − 1 do
                B_{i,j} ← A_{i,k} · B_{i,k}
            end for
        end for
    end for
end function
```

A.3. Parallele Leistung der Matrix-Vektor-Multiplikation im Vergleich

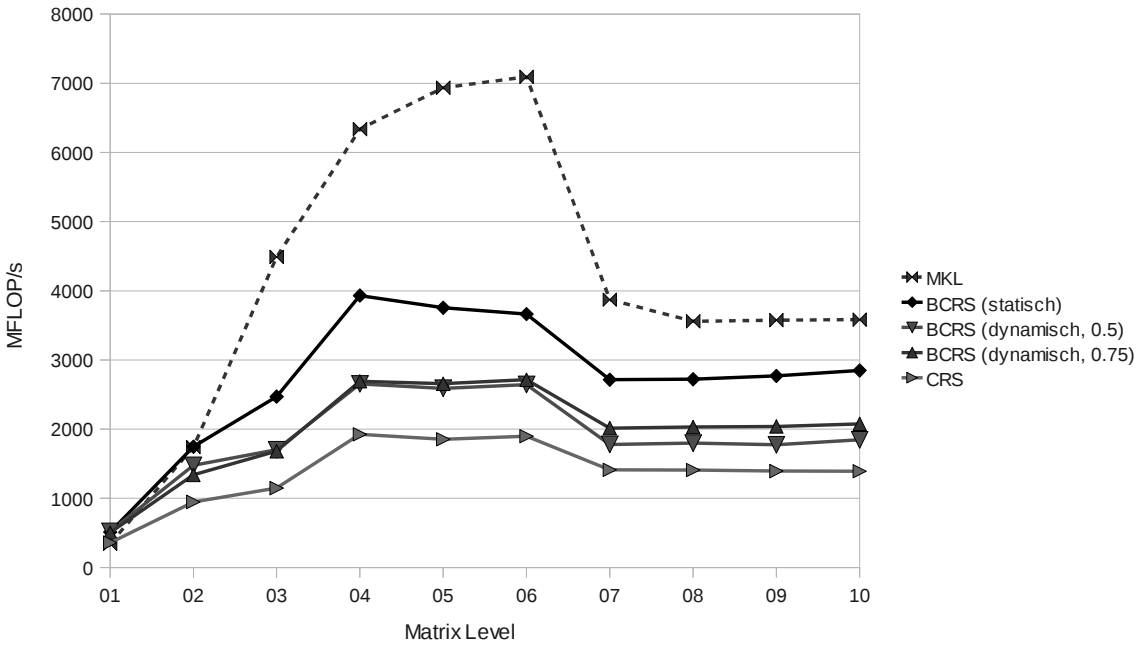

Abbildung A.5.: Matrix-Vektor-Multiplikation im Vergleich: BCRS-Datenstrukturen, MKL und TifaMMyHybrid; Intel Core-i7 2600K mit Datensatz 1 und 2 Threads

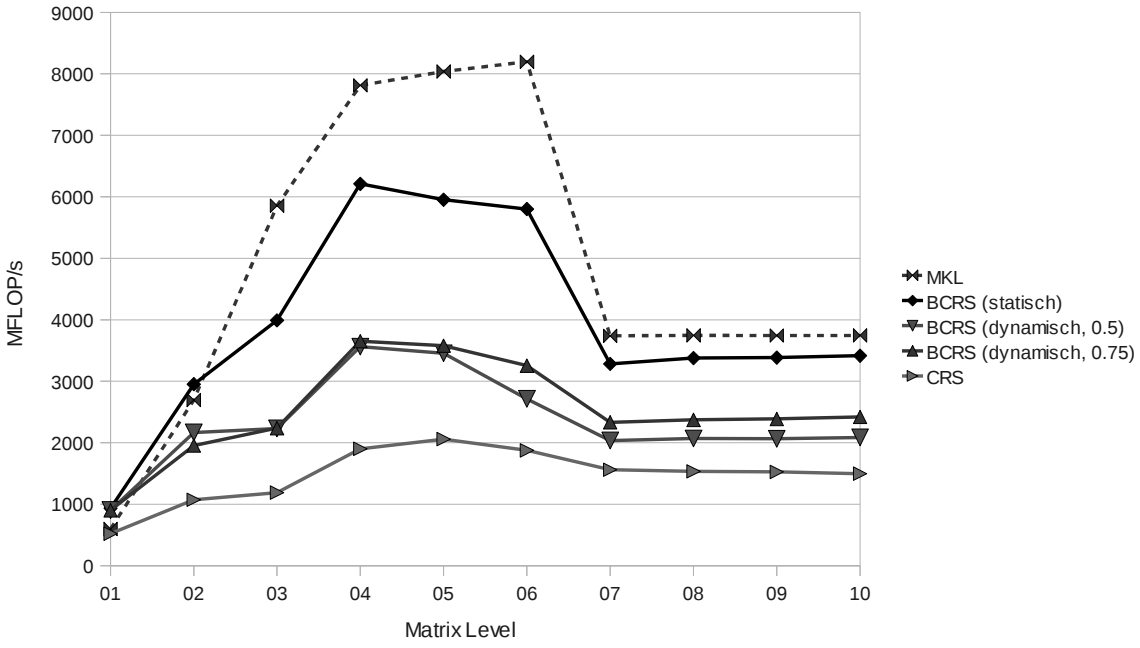

Abbildung A.6.: Matrix-Vektor-Multiplikation im Vergleich: BCRS-Datenstrukturen, MKL und TifaMMyHybrid; Intel Core-i7 2600K mit Datensatz 2 und 2 Threads

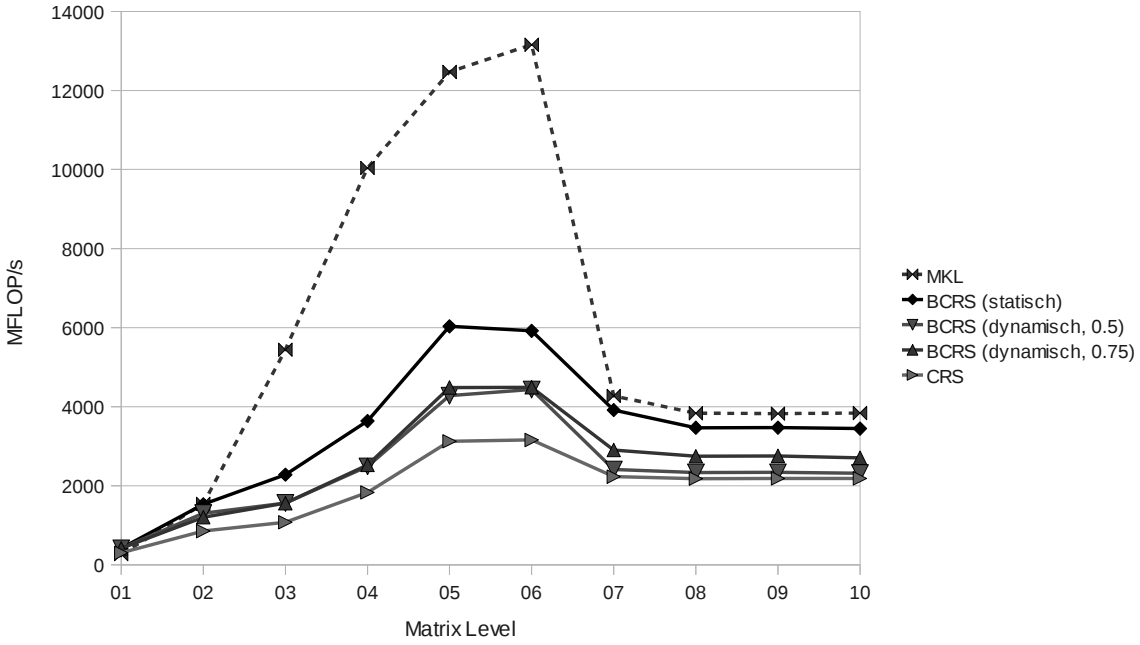

Abbildung A.7.: Matrix-Vektor-Multiplikation im Vergleich: BCRS-Datenstrukturen, MKL und TifaMMyHybrid; Intel Core-i7 2600K mit Datensatz 1 und 4 Threads

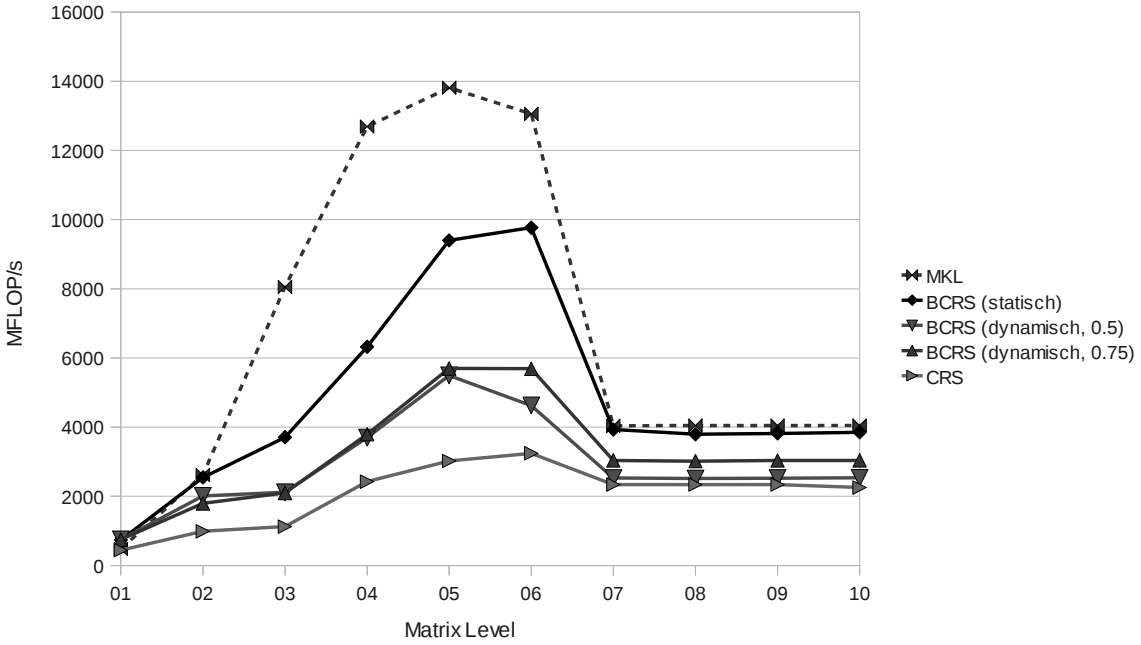

Abbildung A.8.: Matrix-Vektor-Multiplikation im Vergleich: BCRS-Datenstrukturen, MKL und TifaMMyHybrid; Intel Core-i7 2600K mit Datensatz 2 und 4 Threads

Literaturverzeichnis

[BBC+94] Barrett, R. ; Berry, M. ; Chan, T. F. ; Demmel, J. ; Donato, J. ; Dongarra, J. ; Eijkhout, V. ; Pozo, R. ; Romine, C. ; Vorst, H. V.: *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. Philadelphia, PA : SIAM, 1994 (Zitiert auf Seite 23)

[BZ06] Bader, M ; Zenger, C: Cache oblivious matrix multiplication using an element ordering based on a Peano curve. In: *Linear Algebra and its Applications* 417 (2006), Nr. 2-3, S. 301–313 (Zitiert auf den Seiten 9, 14 und 19)

[CV94] Chan, Tony F. ; Vorst, Henk A. d.: Approximate And Incomplete Factorizations. In: *ICASE/LARC INTERDISCIPLINARY SERIES IN SCIENCE AND ENGINEERING*, 1994, S. 167–202 (Zitiert auf den Seiten 14 und 56)

[FLPR99] Frigo, Matteo ; Leiserson, Charles E. ; Prokop, Harald ; Ramachandran, Sridhar: Cache-Oblivious Algorithms. In: *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*. Washington, DC, USA : IEEE Computer Society, 1999 (FOCS '99). – ISBN 0–7695–0409–4, 285– (Zitiert auf Seite 14)

[HB09] Heinecke, Alexander ; Bader, Michael: Towards many-core implementation of LU decomposition using Peano Curves. In: *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*. New York, NY, USA : ACM, 2009 (UCHPC-MAW '09). – ISBN 978–1–60558–557–4, 21–30 (Zitiert auf den Seiten 9, 14, 31 und 34)

[Kum03] Kumar, Piyush: Cache oblivious algorithms. In: *Algorithms for Memory Hierarchies, LNCS 2625*, Springer-Verlag, 2003, S. 193–212 (Zitiert auf Seite 14)

[LRW91] Lam, Monica S. ; Rothberg, Edward E. ; Wolf, Michael E.: The Cache Performance and Optimizations of Blocked Algorithms. In: *In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991, 63–74

[May06] Mayer, Christian: *Cache oblivious matrix operations using Peano curves*, Technische Universität München, Diplomarbeit, 2006 (Zitiert auf den Seiten 31 und 34)

[Sch] Schweitzer, Marc A.: Efficient Implementation and Parallelization of Meshfree and Particle Methods - The Parallel Multilevel Partition of Unity Method. (Zitiert auf den Seiten 9 und 11)

[Sch03] Schweitzer, M.A.: *A Parallel Multilevel Partition of Unity Method for Elliptic Partial Differential Equations*. Springer, 2003 (Lecture Notes in Computational Science and Engineering). – ISBN 9783540003519 (Zitiert auf Seite 9)

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Patrick Gründer)