Lösung dünnbesetzter Dreieckssysteme auf modernen Mehrkernprozessoren

Bachelorarbeit

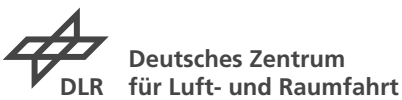
vorgelegt von **Christopher Zachow**
im Rahmen des Bachelor-Studiengangs Mathematik
am 27. Juni 2017

am Mathematischen Institut der Universität zu Köln

und

am Deutsches Zentrum für Luft- und Raumfahrt

Erstgutachter: Prof. Dr. Axel Klawonn
Zweitgutachter: Dr. Jonas Thies



Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Schon die Gelehrten im antiken China beschäftigten sich mit der Lösung von linearen Gleichungen mehrerer Unbekannter [1]. Auch über zweitausend Jahre später spielen sogenannte lineare Gleichungssysteme eine wichtige Rolle bei der Simulation komplexer Systeme in den Naturwissenschaften. Ein Beispiel hierfür ist die Lösung eines Minimierungsproblems mittels des Newton-Verfahrens [2, S. 83ff]. Innerhalb jedes Iterationsschrittes muss ein lineares Gleichungssystem gelöst werden. Während man früher die Gleichungen für wenige Unbekannte handschriftlich lösen konnte, benötigt man für die Lösung heutiger Probleme leistungsstarke Computersysteme.

Lineare Gleichungssysteme Allgemein beschreibt der Ausdruck lineares Gleichungssystem eine Menge von $N \in \mathbb{N}$ Gleichungen in $M \in \mathbb{N}$ Unbekannten. Innerhalb der Gleichungen werden die Abhängigkeiten der Unbekannten durch sogenannte Koeffizienten und die rechte Seite ausgedrückt. Die rechte Seite sind zusätzliche konstante, nicht von den Unbekannten abhängige, Terme. Die j-te Gleichung hat damit die Form:

$$a_{j1}x_1 + a_{j2}x_2 + \ldots + a_{jM-1}x_{M-1} + a_{jM}x_M = b_j.$$

Dabei haben wir die Koeffizienten in Gleichung j durch a_{jk}, die Unbekannten durch x_k und die rechte Seite durch b_j ausgedrückt. Der Einfachheit halber wollen wir uns auf reelle Gleichungssysteme beschränken, d.h. es gilt: $a_{jk} \in \mathbb{R}$ für $j = 1, ..., N$ bzw. $k = 1, ..., M; b_j \in \mathbb{R}$ für $j = 1, ..., N; x_k \in \mathbb{R}$ für $k = 1, ..., M$. Man nennt das Gleichungssystem homogen, falls für die rechte Seite $b_j = 0$ für $j = 1, ..., N$ gilt. Diese Notation motiviert das gesamte lineare Gleichungssystem mit Hilfe einer Matrix und zweier Vektoren auszudrücken:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}.$$

Hierbei haben wir die Koeffizienten a_{jk} zur Matrix $\boldsymbol{A} \in \mathbb{R}^{N \times M}$ mit $(\boldsymbol{A})_{jk} = a_{jk}$, die Unbekannten x_k zum Vektor $\boldsymbol{x} \in \mathbb{R}^M$ mit $(\boldsymbol{x})_k = x_k$ und die rechte Seite b_j zum Vektor $\boldsymbol{b} \in \mathbb{R}^N$ mit $(\boldsymbol{b})_j = b_j$ zusammengefasst. Wir wollen das Gleichungssystem bzw. die

Matrix dünnbesetzt (engl. sparse) nennen, wenn ein Großteil der Koeffizienten null ist. Entsprechend nennen wir die Matrix dichtbesetzt, falls es unökonomisch ist, die Anzahl an Null-Einträgen auszunutzen [3, S. 191].
Ein lineares Gleichungssystem ist lösbar mit der Lösung $\boldsymbol{x}^*$, falls die erweiterte Matrix $(\boldsymbol{A}, \boldsymbol{b})$ ($\boldsymbol{b}$ wird als weiterer Spaltenvektor der Matrix hinzugefügt) und die Matrix selbst, den gleichen Rang haben [4, S. 134].

$$rank\,(\boldsymbol{A}) = rank\,(\boldsymbol{A}, \boldsymbol{b})$$

Wir wollen im Folgenden nur quadratische ($M = N$), eindeutig lösbare lineare Gleichungssysteme betrachten, daher wählen wir die Matrizen passend.

Lösungsverfahren Wie in der Literatur üblich, wollen wir zwischen zwei Arten von Lösungsverfahren für lineare Gleichungssysteme unterscheiden. Zum Einen gibt es direkte und zum Anderen iterative Verfahren. Direkte Verfahren führen bei korrekter Arithmetik nach einer festen Anzahl Schritten zur exakten Lösung, wohingegen iterative Verfahren schrittweise eine näherungsweise Lösung verbessern.
Ein bekanntes Beispiel für ein direktes Verfahren ist der Gauß-Algorithmus, dieser basiert auf elementaren Umformungen der Zeilen. Leider ist der Algorithmus nur mit Pivotisierung numerisch stabil [5, S. 94ff] und schlecht parallelisierbar.
Ein iteratives Verfahren erzeugt eine Folge von Näherungslösungen $\boldsymbol{x}_k$, die gegen die exakte Lösung $\boldsymbol{x}^*$ konvergiert. Dabei wird die Näherungslösung des vorherigen Iterationsschritts als Startwert für den nächsten Iterationsschritt genutzt. Die Iteration wird abgebrochen, sobald die Lösung hinreichend genau ist, beispielsweise die Norm des Residuums $\|\boldsymbol{r_k}\| = \|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_k\|$ hinreichend klein ist. Innerhalb der Klasse der iterativen Verfahren wollen wir uns genauer mit der Gruppe der Krylov-Unterraum-Verfahren beschäftigen und dabei beispielhaft mit dem CG-Verfahren (vergleiche hierzu [2, Kapitel 13]). Es basiert auf der Idee, dass das Minimum der Funktion $E\,(\boldsymbol{x}) = \frac{1}{2}\langle \boldsymbol{A}\boldsymbol{x}, \boldsymbol{x}\rangle - \langle \boldsymbol{b}, \boldsymbol{x}\rangle$ auch Lösung des linearen Gleichungssystems ist. In jedem Iterationsschritt minimiert es die Funktion in eine Richtung $\boldsymbol{d}_k$, welche $\boldsymbol{A}$ konjugiert $\langle \boldsymbol{A}\boldsymbol{d}_k, \boldsymbol{d}_k\rangle = 0$ ist. Dazu muss in jedem Iterationsschritt eine Matrix-Vektor-Multiplikation durchgeführt werden. Diese ist für dünnbesetzte Matrizen effizient implementierbar und gut zu parallelisieren [6, Abschnitt 11.5]. Das Verfahren ist für symmetrisch, positiv definite Matrizen anwendbar und konvergiert, bei exakter Arithmetik, innerhalb von N Schritten zur exakten Lösung. Die Konvergenzrate des CG-Verfahrens ist von der Kondition der Matrix $\boldsymbol{A}$ abhängig und damit vom Verhältnis des größten und kleinsten Eigenwerts der Matrix. Dies be-

deutet, dass Matrizen mit einem sehr großen und einem sehr kleinen Eigenwert ein sehr schlechtes Konvergenzverhalten aufweisen. Die Lösung für dieses Problem liegt in einer sog. Vorkonditionierung [6, S. 275ff]. Dabei werden beide Seiten des Gleichungssystems mit einer weiteren Matrix $\boldsymbol{K}$ multipliziert.

$$\boldsymbol{KAx} = \boldsymbol{Kb}$$

Die Matrix $\boldsymbol{K}$ muss so gewählt werden, dass die Kondition der neuen Matrix $\boldsymbol{KA}$ kleiner ist als die Kondition der Matrix $\boldsymbol{A}$. Ebenso sollte der Vorkonditionierer leicht bestimmbar sein, damit sich durch die Verwendung eine Verbesserung der Rechenzeit ergibt. Entsprechend sollte für den Vorkonditionierer gelten: $\boldsymbol{K} \approx \boldsymbol{A}^{-1}$ und damit $\boldsymbol{KA} \approx \mathbb{1}_N$. Bereits der vergleichsweise einfache Jacobi-Vorkonditionierer $\boldsymbol{K} = \boldsymbol{D}^{-1}$ ($\boldsymbol{D} = diag\,(\boldsymbol{A})$) zeigt leichte Verbesserungen der Konvergenzgeschwindigkeit.

Vorkonditionierung durch ILU Wir wollen nun die Vorkonditionierung durch unvollständige LU-Zerlegung (kurz: ILU) betrachten. Dabei wird die Matrix $\boldsymbol{A}$ in eine untere Dreiecksmatrix $\boldsymbol{L}$ und eine obere Dreiecksmatrix $\boldsymbol{U}$ zerlegt. In den beiden Faktoren wollen wir nur Einträge an den Stellen zulassen, an denen $\boldsymbol{A}$ schon Einträge besitzt. Damit ist die Zerlegung unvollständig und es gilt $\boldsymbol{A} \approx \boldsymbol{LU}$. Zur Berechnung einer solchen Zerlegung gibt es eine Vielzahl von Algorithmen, die Meisten basieren auf dem Gauß-Algorithmus [6, S. 287ff]. Einen neuartigen Ansatz präsentierten Chow et al. im Jahre 2015 [7]. Ihr Algorithmus fasst die ILU-Zerlegung als Fixpunktiteration auf und berechnet daher die Zerlegung iterativ. Damit kann der Algorithmus parallelisiert werden und es konnte gezeigt werden, dass die Konvergenz der ILU-Zerlegung nicht von der Kondition der Matrix abhängig ist.

Anstatt nun die beiden Matrizen für Berechnungen der Form $\boldsymbol{U}^{-1}\boldsymbol{L}^{-1}\boldsymbol{b}' = \boldsymbol{x}$ zu invertieren, löst man zwei Gleichungssysteme der Form $\boldsymbol{Lb} = \boldsymbol{b}'$ und $\boldsymbol{Ux} = \boldsymbol{b}$. Deren Lösung wollen wir in der folgenden Arbeit behandeln. Hierzu gehen wir zunächst auf die Anforderungen ein, welche ein solcher Algorithmus erfüllen sollte. Anschließend beschreiben wir zwei Verfahren zur Lösung solcher Dreieckssysteme, einen etablierten Algorithmus, das Level-Scheduling, und einen vergleichsweise neuen Algorithmus, welcher ohne Synchronisation der Prozessoren auskommt. Die Hauptarbeit liegt in der Implementierung des zweiten Algorithmus für das Softwarepaket „`GHOST`“ [8]. Dabei vergleichen wir unsere Implementierung mit der anderer Autoren und einer Standardreferenz, der Intel `MKL`. Abschließend diskutieren wir die gewonnen Ergebnisse und formulieren ein Fazit.

2 Hauptteil

2.1 Grundlagen

2.1.1 Moderne Computerarchitekturen

Um Berechnungen zahlreicher, genauer und schneller durchführen zu können wurden bereits früh Rechenmaschinen, Vorläufer der heutigen Computer, erfunden. Während sie zunächst nur mechanisches Hilfsmittel des Menschen waren, wurden sie im Zuge der Entdeckung der Elektrizität zu Maschinen weiterentwickelt. Mit der Entdeckung des Transistors begann dann die fortlaufende Miniaturisierung der Bauteile und die Entwicklung des heutigen Computers.
Moderne Computersysteme zur rechnerischen (numerischen) Lösung komplexer Probleme werden grob in zwei verschiedene Architekturen eingeteilt. Man unterscheidet Distributed-Memory- (verteilter Speicher) [9, S. 103ff] und Shared-Memory-Systeme (gemeinsamer Speicher) [9, S. 77ff]. Tatsächlich sind die meisten Systeme als hybride Architektur aufgebaut. Dies bedeutet, dass der Computer unterteilt ist in sogenannte Knoten oder Nodes, welche untereinander mit einem Netzwerk zu einem Cluster verbunden sind. Will man Informationen zwischen den Nodes austauschen, so müssen diese durch das Netzwerk transportiert werden (distributed memory). Innerhalb einer Node gibt es mehrere Recheneinheiten, die gemeinsam auf einen Speicherpool zugreifen können (shared memory). Dabei kann es jedoch zu unterschiedlichen Zugriffsgeschwindigkeiten kommen, da manche Speicherbereiche weiter von einer Recheneinheit entfernt sind als Andere. Dies wird als „Non-uniform memory access“ (NUMA) bezeichnet. Eine solche Recheneinheit wird Prozessor oder CPU (Central Processing Unit) genannt.
Um die Jahrtausendwende wurde klar, dass eine fundamentale Änderung der Prozessorarchitektur nötig ist, um eine weitere Steigerung der Performance zu ermöglichen [10]. Bisher bestanden Prozessoren immer aus einem einzigen zentralen Rechenwerk (Core). Durch die stetig steigende Leistungsaufnahme der Prozessoren war auch deren Leistungsdichte gestiegen und machte es bald unmöglich, diese weiter unter Kontrolle zu halten.

Ebenso wurde es immer komplizierter genügend Daten bereitzustellen, da die Speicherarchitektur nicht in gleichem Maße mit gewachsen war.
Die Lösung dieses Problems bestand in der Entwicklung der sogenannten Multi-Core- und später Many-Core-Archtiktur, zu deutsch Mehr-Kern- und Viel-Kern-Architektur. Mehrere gleiche Prozessorkerne werden hierbei kombiniert. Dabei teilen sie sich die verschiedenen anderen Komponenten des Prozessors und greifen gemeinsam auf sie zu. Der Nachteil dieser Architektur ist, dass es neuer Programme bedarf, um die Leistungsfähigkeit des Prozessors nutzen zu können.
Zur etwa gleichen Zeit wurden parallele Instruktionen, sogenannte `SIMD` Instruktionen (single instruction multiple data) entwickelt. Diese greifen das Prinzip der Vektorrechner aus den 80er und 90er Jahren wieder auf und ermöglichen eine weitere Verbesserung der Performance. Während eine normale Instruktion nur auf eine Dateneinheit angewendet werden kann, wird bei einer Vektorinstruktion die gleiche Instruktion auf einen Block (Vektor) von Dateneinheiten angewendet. Zu beachten ist jedoch, dass es keine Datenabhängigkeit innerhalb des Vektors geben darf. Ebenso müssen Zugriffe auf den Speicher immer in ganzzahligen Vielfachen der Vektorlänge erfolgen (alignment). Dies kann eine Änderung des Datenlayouts zur Folge haben.
Wir wollen im Folgenden nur die Shared-Memory-Architektur betrachten, d.h. wir beschränken uns auf einen Knoten des Clusters. Entsprechend suchen wir Algorithmen, die beide Arten von Parallelität innerhalb des Knotens nutzen. Zum Einen sollte der Algorithmus die Arbeit gleichmäßig auf die vorhandenen Kerne verteilen und zum Anderen sollte er die Vektorinstruktionen des Prozessors nutzen können. Im Folgenden verwenden wir die Begriffe Prozessor und Prozessorkerne synonym.

2.1.2 Messgrößen für Computerprogramme

Um Computerprogramme vergleichen zu können, benötigt man Messgrößen anhand derer der Vergleich vorgenommen werden kann. Beispiele für Messgrößen sind die Anzahl an Speicherzugriffen, die durchgeführten Operationen pro Zeiteinheit oder die benötigte Energiemenge. Dabei unterscheidet man zwischen direkten und indirekten Größen. Direkte Messgrößen lassen sich messtechnisch ermitteln bzw. ablesen, zu ihnen gehört bspw. die Laufzeit des Programms. Dies ist die vergangene Zeit zwischen Programmstart und Programmabschluss. Aus dieser direkten Größe lässt sich die mittlere Laufzeit ableiten, eine indirekte Messgröße. Die mittlere Laufzeit ist das arithmetische Mittel der Laufzeiten mehrerer Programmdurchläufe. Die Zeitdauer eines einzelnen Laufes kann aufgrund von anderen Programmen, welche zur gleichen Zeit ausgeführt werden, zum

Teil stark variieren. Bei einer hinreichend großen Zahl an Wiederholungen (bspw. 100) sollte die Standardabweichung klein genug sein, um eine vergleichbare Größe zu haben. Ist man am Zeitgewinn durch Parallelisierung auf einer bestimmten Anzahl von Prozessoren interessiert, so ist der sog. Speedup die zu betrachtende Größe. Dieser ist das Verhältnis aus der Laufzeit des parallelen und des bestmöglichen seriellen Programms.

$$S_p = \frac{T_{seriell}}{T_{parallel}} \tag{2.1}$$

Für verschiedene Probleme ist nicht unbedingt bewiesen, dass ein gegebener serieller Algorithmus der Beste ist, sodass hier meist eine bestimmte Variante gewählt werden muss. Ebenso hängt der Speedup von der verwendeten Anzahl Prozessoren ab. Vergleicht man zwei parallele Programme, so lässt sich auch zwischen diesen ein Speedup definieren. Er ist gegeben als der Quotient der jeweiligen Speedups und damit als Quotient der beiden Laufzeiten.
Um zu untersuchen, wie effizient ein Algorithmus den Prozessor nutzt, ist die Performance des Programms wichtig. Diese wird angegeben in Operationen pro Zeiteinheit. Eine Operation ist eine elementare Berechnung mit einer einzelnen Dateneinheit, für die der Prozessor im Idealfall genau einen Befehl benötigt. Zu diesen Operationen gehören die Addition und Multiplikation. Die Effizienz des Programms lässt sich dann als Verhältnis von theoretischer Performance des Prozessors und tatsächlicher Performance des Programms definieren.

2.1.3 Speicherung dünnbesetzter Matrizen

Wie wir in der Einleitung bereits gesehen haben, sollten dünnbesetzte Matrizen nicht so behandelt werden, wie Dichtbesetzte. In diesem Abschnitt wollen wir uns mit der effizienten Speicherung solcher Matrizen beschäftigen. Die Idee aller dünnbesetzter Matrixformate ist, dass hauptsächlich Nicht-Null-Einträge gespeichert werden. Deren Anzahl wollen wir mit *NNZ* bezeichnen.

CSR Ein weit verbreitetes Format ist das CSR-Format, dessen Name sich aus dem Englischen „compressed sparse row storage“ ableitet. Die Matrix wird in drei Vektoren unterschiedlicher Dimension gespeichert.

$$\boldsymbol{A} \in \mathbb{R}^{N \times N} \rightarrow \boldsymbol{r} \in \mathbb{N}^{N+1}, \boldsymbol{c} \in \mathbb{N}^{NNZ}, \boldsymbol{v} \in \mathbb{R}^{NNZ} \tag{2.2}$$

Der Vektor $\boldsymbol{v}$ enthält alle Nicht-Null-Einträge der Matrix, wobei die Einträge zeilenweise hintereinander abgespeichert sind, d.h erst alle Einträge der ersten Zeile, dann alle der Zweiten usw. Die Einträge einer Zeile sind aufsteigend nach ihren Spalten sortiert. Die Information über die zugehörige Spalte wird dann im Vektor $\boldsymbol{c}$ abgespeichert. Der Beginn jeder Zeile wird im Vektor $\boldsymbol{r}$ gespeichert. Damit ist der erste Eintrag des Vektors $\boldsymbol{r}_1 = 0$ und der Letzte $\boldsymbol{r}_{N+1} = NNZ$, wobei der letzte Eintrag eine Konvention ist. In Abbildung 2.1 ist die Speicherfolge für das CSR-Format beispielhaft dargestellt. Dieses Speicherformat ist besonders gut geeignet, wenn man gleichzeitig auf alle Einträge einer Zeile zugreifen will, da diese hintereinander gespeichert sind.

CSC Das CSC-Format, englisch für „compressed sparse column storage“ ist dem CSR-Format sehr ähnlich. Es basiert auf der gleichen Idee, ist jedoch spaltenbasiert.

$$\boldsymbol{A} \in \mathbb{R}^{N \times N} \rightarrow \boldsymbol{c} \in \mathbb{N}^{N+1}, \boldsymbol{r} \in \mathbb{N}^{NNZ}, \boldsymbol{v} \in \mathbb{R}^{NNZ} \tag{2.3}$$

Entsprechend werden alle Elemente einer Spalte hintereinander im Vektor $\boldsymbol{v}$ gespeichert und deren entsprechende Zeile im Vektor $\boldsymbol{r}$. Der Vektor $\boldsymbol{c}$ enthält dann die Information über den Index, ab dem eine bestimmte Spalte beginnt.

ELLPACK Das Format ELLPACK stellt eine Idee da, eine dünnbesetzte Matrix in zwei dichtbesetzte Matrizen zu überführen.

$$\boldsymbol{A} \in \mathbb{R}^{N \times N} \rightarrow \boldsymbol{C} \in \mathbb{N}^{N \times NR}, \boldsymbol{V} \in \mathbb{R}^{N \times NR} \tag{2.4}$$

Die Dimensionen der Matrizen sind gegeben als die Anzahl an Spalten N und die maximale Anzahl an Zeileneinträgen NR. Da nicht jede Zeile die gleiche Anzahl an Nicht-Null-Einträgen hat, werden Zeilen mit weniger Einträgen mit Nullen aufgefüllt. Diesen Prozess nennt man Padding. In der Matrix $\boldsymbol{V}$ werden die Elemente der Matrix gespeichert, während der Zeilenindex unverändert bleibt, gibt der Spaltenindex an, um das wievielte Nicht-Null-Element es sich handelt. Der korrekte Spaltenindex wird ähnlich dem Element selbst in der Matrix $\boldsymbol{C}$ abgespeichert. Speichert man die beiden Matrizen auch als Vektoren der Dimension $N \cdot NR$ ab, so wählt man das Speichermuster, welches in Grafik 2.1 zu sehen ist. Weicht die Anzahl an Nicht-Null-Einträgen pro Zeile stark voneinander ab, so müssen viele zusätzliche explizite Null-Einträge gespeichert werden.

SELL-C-1 Eine Mischung der beiden Formate stellt das SELL-C-1-Format (Sliced ELLPACK) dar. Dieses unterteilt die Zeilen der Matrix in Blöcke sog. Chunks der Länge C. Sollte die Blockgröße nicht zur Dimension der Matrix passen, so werden Leer-Zeilen hinzugefügt. Innerhalb eines Blocks wird das ELLPACK-Format verwendet und oberhalb der Blöcke ein Art CSR-Format. Dies verringert die Menge an expliziten Null-Einträgen aufgrund von Padding erheblich, vergleiche dazu auch Grafik 2.1 und [11]. Entsprechend wird die Matrix in vier Vektoren gespeichert.

$$\boldsymbol{A} \in \mathbb{R}^{N \times N} \rightarrow \boldsymbol{s} \in \mathbb{N}^{NC}, \boldsymbol{l} \in \mathbb{N}^{NC}, \boldsymbol{c} \in \mathbb{N}^{NNZ}, \boldsymbol{v} \in \mathbb{R}^{NNZ} \tag{2.5}$$

Während die Vektoren $\boldsymbol{c}$ und $\boldsymbol{v}$ die gleichen Werte, jedoch in anderer Reihenfolge wie beim CSR-Format, enthalten, gibt es zwei neue Vektoren. Zum Einen den Vektor $\boldsymbol{s}$ der Länge NC (Anzahl der Chunks), welcher die Anfänge aller Chunks speichert. Zum Anderen den Vektor $\boldsymbol{l}$ gleicher Länge, welcher die Länge des aktuellen Chunks speichert.
Dieses Format wurde entwickelt, um für verschiedene Architekturen ein gemeinsames Format bereitzustellen. Dies ist insbesondere unter dem Aspekt der Nutzung hybrider Computersysteme, welche aus gewöhnlichen Prozessoren und zusätzlichen Beschleunigern (Grafikprozessoren bzw. Vektorprozessoren) bestehen, wichtig. Der Parameter C sollte zum System passend gewählt werden und der Größe der Vektoreinheit entsprechen (bspw. $C = 4$ für CPUs mit `AVX` und Berechnungen in doppelter Genauigkeit).
Auf dem Sell-C-1 Format aufbauend, wurde das SELL-C-σ Format entwickelt. Der Parameter σ gibt einen Bereich an, in dem eine zusätzliche Sortierung vorgenommen wird. Daher wird der Parameter auch „sorting scope“ genannt. Es werden große Blöcke der Länge σ gebildet. Innerhalb dieser Blöcke werden die Zeilen absteigend ihrer Anzahl an Nicht-Null-Einträgen sortiert. Die sortierten Blöcke werden dann wieder entsprechend der gewählten Chunk-Größe unterteilt. Der Vorteil dieser zusätzlichen Sortierung liegt in einer weiteren Reduzierung des Paddings.

2.1.4 Lösung von Dreieckssystemen

Wie bei der Lösung eines linearen Gleichungssystems bieten sich auch hier zwei Arten von Lösungsverfahren an. Nämlich die direkte Lösung oder die iterative Lösung des Dreieckssystems. Zumeist zeichnen sich iterative Verfahren durch eine gute Parallelisierbarkeit aus, jedoch ist bei vielen Verfahren das Konvergenzverhalten von der Kondition der Matrix abhängig. Da das betrachtete Dreieckssystem aus der ILU-Zerlegung einer schlecht konditionierten Matrix entstanden ist, kann das System auch schlecht konditio-

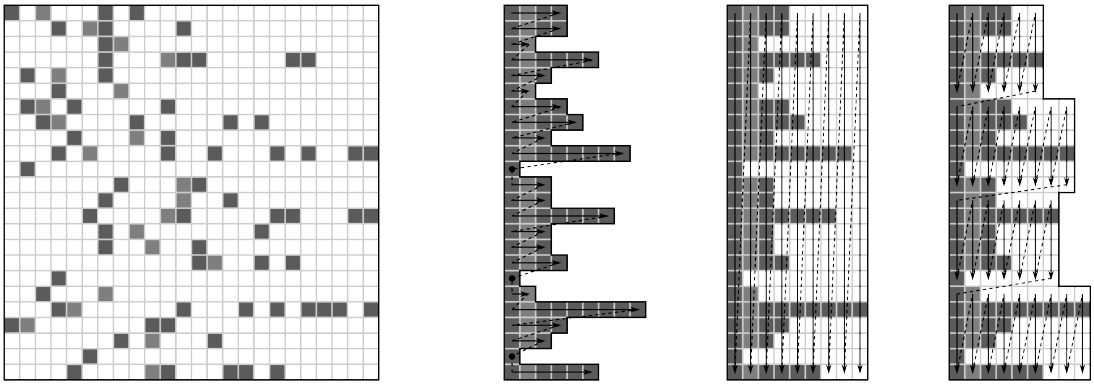


Abbildung 2.1: Vergleich der drei Speicherformate CSR, ELLPACK und SELL-6-1 mit der Verteilung der Einträge für eine Beispielmatrix [11]

niert sein. Damit sind iterative Verfahren, deren Konvergenz von der Kondition abhängig ist, für diesen Problemfall eher ungeeignet. Unter gewissen Bedingungen haben Chow et al. die Verwendbarkeit von iterativen Methoden, wie der Jacobi-Iteration, gezeigt [12]. Um keine Einschränkungen an die Matrix stellen zu müssen, wollen wir uns mit direkten Verfahren zur Lösung des Dreieckssystems beschäftigen und dabei zunächst auf ein serielles Verfahren eingehen, bevor wir uns danach mit der Parallelisierung beschäftigen. Wir betrachten dabei immer eine obere Dreiecksmatrix, da sich alle Algorithmen auch auf Systeme mit unterer Dreiecksmatrix übertragen lassen. Man unterscheidet zwischen vorwärts und rückwärts gerichteten Verfahren. Bei vorwärts gerichteten Verfahren berechnet man nach dem Lösen der aktuellen Zeile das Produkt aller Einträge mit gleichem Spaltenindex wie der aktuellen Zeile mit dem aktuellen Element des Lösungsvektors und speichert diese ab. Damit hat man bereits Anteile berechnet, die man später beim Lösen der entsprechenden Zeilen benötigt. Die rückwärts gerichteten Verfahren führen eine solche Vorberechnung nicht durch, sondern berechnen das Produkt aus aktueller Zeile und anteiligem Lösungsvektor. Je nach Verfahrensart bietet sich die Wahl eines der vorgestellten dünnbesetzten Formate zur Speicherung der Matrix an. Dabei lässt sich feststellen, dass für rückwärts gerichtete Verfahren zeilenbasierte Formate besser sind und umgekehrt für vorwärts gerichtete Verfahren Spaltenbasierte [13]. Die Wahl des passenden Formats behandeln wir im Abschnitt Implementierung.

Serieller Ansatz

Rückwärts-Substition Der einfachste Algorithmus zur Lösung eines oberen Dreiecks-systems ist die Rückwärts-Substitution. Für gewöhnlich wird sie als Teil des Gauß-Algorithmus angesehen. Da in der untersten Zeile des Gleichungssystems nur ein Eintrag ist, lässt sich die Unbekannte x_N sofort berechnen. Da Zeilen maximal von allen Zeilen mit höherer Zeilennummer abhängig sind, kann anschließend die vorletzte Zeile durch Einsetzen von x_N gelöst werden. Dadurch können wir Schritt für Schritt alle Zeilen aufsteigend nacheinander berechnen. Aufgrund des Einsetzens bereits berechneter Unbekannter von unten nach oben, nennt man das Verfahren Rückwärts-Substitution. Der Algorithmus ist in Listing 1 dargestellt.

Algorithmus 1 Rückwärts-Substitution (rückwärts gerichtet)

Input: $\boldsymbol{U} \in \mathbb{R}^{N \times N}$; $\boldsymbol{b} \in \mathbb{R}^N$
Output: $\boldsymbol{x} \in \mathbb{R}^N$
for $k = N$ **to** 0 **do**
    $x[k] \leftarrow \left(b[k] - \sum_{m=k+1}^{N} U[k,m] \cdot x[m]\right) / U[k,k]$
end for

Paralleler Ansatz

Während bei dichtbesetzten Matrizen jede Zeile von allen Zeilen mit höherer Zeilennummer abhängt, gibt es bei dünnbesetzten Matrizen Zeilen, die unabhängig voneinander sind. Beispielsweise kann es Zeilen geben, die nur einen einzigen Eintrag besitzen und daher sofort gelöst werden können. Auch Zeilen mit wenigen Einträgen und Abhängigkeit von Zeilen nahe der Untersten, können baldmöglichst bearbeitet werden. Diese Tatsache bildet die Grundlage aller Ideen zur Parallelisierung dünnbesetzter Dreieckssysteme. Alle parallelen Algorithmen werden daher in zwei Phasen unterteilt, eine Analyse-Phase und eine Lösungs-Phase. Während der Analyse-Phase wird die Struktur und die Abhängigkeiten der Einträge der Matrix untersucht. In der darauf folgenden Lösungs-Phase wird dann die gefundene Struktur genutzt, um das Dreieckssystem zu lösen.

Level-Scheduling Der Algorithmus des Level-Scheduling wurde zunächst für Matrizen entwickelt, welche durch Finite-Differenzen-Diskretisierung von Differentialgleichungen entstanden sind. Je nach Art der Diskretisierung gibt es voneinander unabhängige Einträge innerhalb einer Nebendiagonalen der Matrix, die gleichzeitig berechnet werden

können. Dieser Ansatz wurde dann auf allgemeine Matrizen erweitert [6, S. 370ff]. Während der Analyse-Phase müssen Ebenen bzw. Level gefunden werden, denen die Elemente der Matrix zugeordnet werden können. Innerhalb eines Levels können die Berechnungen parallel ablaufen, die Level müssen jedoch sequentiell abgearbeitet werden. Dem ersten Level werden alle Einträge zugeordnet, die keinerlei Abhängigkeit besitzen. Die Einträge des zweiten Levels sind dementsprechend nur abhängig von Einträgen des ersten Levels. Auf diese Weise wird sukzessive jedem Element der Matrix ein passendes Level zugeordnet. Nachdem die Level bekannt sind, kann das Gleichungssystem gelöst werden. In Algorithmus 2 sind beide Phasen dargestellt. Die Aufteilung der Matrix in die verschiedenen Level ist in Grafik 2.2 zu sehen. Je nach Format der Matrix lässt sich das dünnbesetzte Skalarprodukt in der innersten Schleife durch `SIMD` Instruktionen effizient parallelisieren.

Level	1	2	3	4	5
Zeilen	5 & 8	4 & 7	3 & 6	2	1

Abbildung 2.2: Schema der Level-Scheduling-Methode

Synchronization-Free Ein Nachteil der Level-Scheduling-Methode ist der Zwang, nach jedem Level alle Prozessoren zu synchronisieren. Dies bedeutet, dass Prozessoren, welche bereits fertig sind, warten müssen. Ein Lösungsvorschlag wurde im Jahre 2016 auf der Euro-Par unter dem Namen „A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves“ von Liu et al. vorgestellt [14].
In der, im Vergleich zur Level-Scheduling-Methode, kurzen Analyse-Phase bestimmt der Algorithmus die Anzahl an Nicht-Null-Einträgen pro Zeile. Diese wird in der darauf folgenden Lösungs-Phase schrittweise verringert und gibt damit in der Analyse-Phase die Anzahl der noch verbleibenden Elemente an. In der Lösungs-Phase wird das Gleichungssystem parallel gelöst. Unter der Annahme, dass P Prozessoren zur Verfügung stehen, werden zunächst die untersten P Zeilen auf die Prozessoren verteilt. Falls die Zeile bereits lösbar ist, beginnt der Prozessor mit der Arbeit, andernfalls wartet er. Nach erfolgrei-

Algorithmus 2 Level-Scheduling-Methode (rückwärts gerichtet)

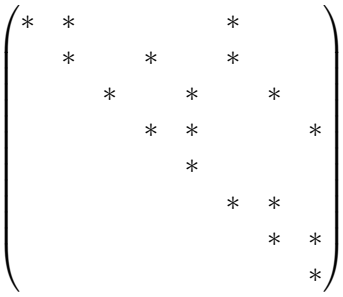
Input: $\boldsymbol{U} \in \mathbb{R}^{N \times N}$; $\boldsymbol{b} \in \mathbb{R}^N$
Output: $\boldsymbol{x} \in \mathbb{R}^N$
$NL \leftarrow 0$
levelList ← Leere Liste der Länge N für Level
procedure ANALYSE-PHASE
    statusList ← Leere Liste der Länge N für Status
    for $j \leftarrow 1$ **to** N **in parallel do**
        statusList[j] ← Anzahl an Einträgen in Zeile j
    end for
    $NL \leftarrow 1$
    do
        for $j \leftarrow 1$ **to** N **in parallel do**
            if *statusList*[j] == 1 **then** ▷ Zeile hängt nur noch von sich selbst ab
                levelList[j] = NL
                statusList[j] ← 0 ▷ Vermeide doppelte Bearbeitung
            end if
        end for
        for $j = 1$ **to** N **in parallel do**
            if *levelList*[j] == NL **then**
                for $k \in$ Spalten der Einträge in Zeile j **do**
                    statusList[k] ← *statusList*[k] − 1
                end for
            end if
        end for
        $NL \leftarrow NL + 1$
    while *statusList*[j] $= 0 \, \forall \, j = 1, ..., N$
    Konvertiere *levelList* in eine Liste die jedem Level die passenden Knoten zuordnet
end procedure
procedure LÖSUNGS-PHASE
    for $j \leftarrow 1$ **to** NL **do**
        for $k \in$ Zeilen in Level j aus *levelList* **in parallel do**
            $x[k] \leftarrow \left(b[k] - \sum_{m=k+1}^{N} U[k,m] \cdot x[m]\right) / U[k,k]$
        end for
        Synchronisation aller Prozessoren
    end for
end procedure

chem Abschluss der Arbeit informiert der Prozessor alle Zeilen, die Einträge besitzen, deren Spaltenindex gleich der aktuellen Zeile ist und damit von der aktuellen Zeile abhängen. Dazu verringert er die Anzahl an verbleibenden Nicht-Null-Elementen dieser Zeile. Anschließend beginnt er mit der Lösung der höchsten nicht gelösten Zeile, siehe dazu Algorithmus 3 und Grafik 2.3. Eine weitere Parallelisierung wird durch die Vorberechnung des dünnbesetzten Skalarproduktes erreicht (vorwärts gerichtetes Verfahren). Sobald eine Zeile gelöst ist, können die Nicht-Null-Einträge der entsprechenden Spalte mit dem Element des Lösungsvektors multipliziert und je Zeile gespeichert werden. Aufgrund dieser Vorberechnung eignet sich ein spaltenbasiertes Format für die Speicherung der Matrix.

Algorithmus 3 Synchronization-Free-Methode (vorwärts gerichtet)

Input: $\boldsymbol{U} \in \mathbb{R}^{N \times N}$; $\boldsymbol{b} \in \mathbb{R}^N$
Output: $\boldsymbol{x} \in \mathbb{R}^N$
sumList ← Leere Liste der Länge N für Vorberechnung
statusList ← Leere Liste der Länge N für den Grad der Zeile
procedure ANALYSE-PHASE
  for $j \leftarrow 1$ **to** N **in parallel do**
    $statusList[j] \leftarrow$ Anzahl an Einträgen in Zeile j
  end for
end procedure
procedure LÖSUNGS-PHASE
  for $k = N$ **to** 1 **in parallel do**
    while $statusList[k] \neq 1$ **do** ▷ Warte bis alle Abhängigkeiten beseitigt sind
    end while
    $x[k] \leftarrow (b[k] - sumList[k])\,/U[k,k]$
    for $m \in$ Zeilenindizes in Spalte k **in parallel do**
      $sumList[m] \leftarrow sumList[m] + U[m,k] \cdot x[k]$)
      $statusList[m] \leftarrow statusList[m] - 1$)
    end for
  end for
end procedure

Grenzen der Parallelisierbarkeit Wir haben somit zwei verschiedene Ansätze zur Lösung der Dreieckssysteme kennengelernt. Die Gemeinsamkeit beider Ansätze ist, dass sie nur die intrinsische Parallelität der Matrix nutzen. Sie werten die Struktur der Matrix aus, um anschließend mehrere Einträge gleichzeitig lösen zu können. Dies hat zur Folge, dass Matrizen mit einer Doppel-Diagonal-Struktur, wie in Abbildung 2.4, eine rein serielle Abarbeitung zur Folge haben. Zumeist wird für die ILU-Zerlegung eine Sortierung



Zeile	1	2	3	4	5	6	7	8
Schritt	6	5	4	4	3	3	2	1

Abbildung 2.3: Schema der Synchronization-Free-Methode mit 2 Prozessoren

der Matrix vorgenommen. Um die Genauigkeit einer iterativen Zerlegung zu verbessern, kann beispielsweise eine Reduzierung der Bandbreite zur Verbesserung der Speicherzugriffe vorgenommen werden. Damit ist die Matrix in der Nähe der Diagonalen eher dichtbesetzt, die gesamte Matrix jedoch weiterhin dünnbesetzt. Diese Sortierung verbessert ebenfalls die Güte der Approximation einer iterativen Methode [15]. Für die beiden beschriebenen Methoden zur Lösung des Dreieckssystems ist sie jedoch unvorteilhaft, da sie die Matrix in eine ähnliche Struktur, wie die Doppel-Diagonal-Struktur bringt. Bei einer Zerlegung mit einer direkten Methode wird eine Sortierung gewählt, welche eine Verbesserung der Parallelität erlaubt. Beispiele hierfür sind Mulit-Coloring oder die Partitionierung der Matrix in Arrowhead-Struktur.

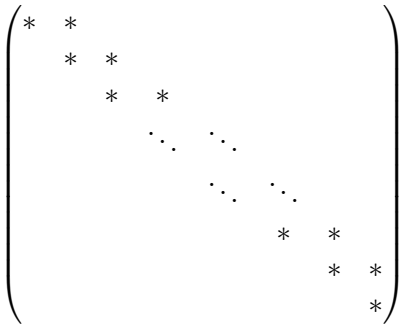


Abbildung 2.4: Matrix mit einer Doppel-Diagonal-Struktur

Arrowhead-Methode Eine solche Partitionierung teilt die Matrix in mehrere Blöcke und einen Separator auf. Diese sogenannte Arrowhead-Form, siehe dazu Grafik 2.5, hat den Vorteil, dass nach serieller Abarbeitung des Separators alle Blöcke parallel abgearbei-

tet werden können. Der entsprechende Algorithmus für die Lösung des Dreieckssystems ist in Alg. 4 zu sehen. Die Anzahl der Blöcke sollte ein Vielfaches der zur Verfügung stehenden Prozessorkerne innerhalb eines Rechenknotens sein. Unter dieser Bedingung dürfte die Aufteilung bestmöglich sein. Ein Nachteil ist jedoch, dass die Partitionierung in eine Arrowhead-Form für andere Algorithmen ungeeignet sein kann. Wie in Algorithmus 4 zu erkennen ist, entfällt die Analyse-Phase, da die Informationen über die Partitionierung direkt übergeben werden können.

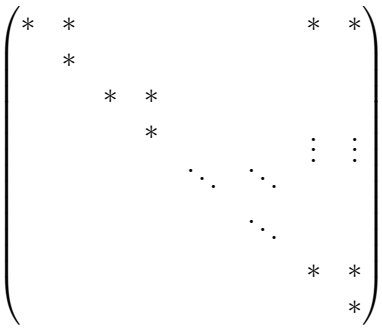


Abbildung 2.5: Matrix mit einer Arrowhead-Struktur

Algorithmus 4 Arrowhead-Matrix-Methode (rückwärts gerichtet)

Input: $\boldsymbol{U} \in \mathbb{R}^{N \times N}$ in Arrowhead-Form; $\boldsymbol{b} \in \mathbb{R}^N$; *blockPtr* ← Liste mit Beginn der Blöcke bzw. Separator; $NB \leftarrow$ Anzahl der Blöcke
Output: $\boldsymbol{x} \in \mathbb{R}^N$
procedure LÖSUNGS-PHASE
  for $k = \text{blockPtr}[NB+2]$ **to** $blockPtr[NB+1]$ **do**
    $x[k] \leftarrow \left(b[k] - \sum_{m=k+1}^{N} U[k,m] \cdot x[m]\right) / U[k,k]$
  end for
  for $j = 1$ **to** NB **in parallel do**
    for $k = \text{blockPtr}[j+1]$ **to** $blockPtr[j]$ **do**
      $x[k] \leftarrow \left(b[k] - \sum_{m=k+1}^{N} U[k,m] \cdot x[m]\right) / U[k,k]$
    end for
  end for
end procedure

Blockinversion Verwendet man das SELL-C-σ Format zur Speicherung der Matrix, so hat man automatisch eine Unterteilung der Matrix in abhängige Blöcke $\boldsymbol{U_j} \in \mathbb{R}^{C \times (N - Cj)}$ erzeugt. Die rechte Seite lässt sich ebenfalls so unterteilen $\boldsymbol{b_j} \in \mathbb{R}^C$. Jeder Block der

Matrix lässt sich nun in zwei Anteile zerlegen, einen Diagonalanteil $\boldsymbol{D_j} \in \mathbb{R}^{C\times C}$ und den Rest $\tilde{\boldsymbol{U_j}} \in \mathbb{R}^{C\times(N-(C+1)j)}$. Invertiert man nun den Diagonalanteil, so lässt sich das System wie folgt schreiben.

$$\boldsymbol{U_j x} = [\boldsymbol{D_j}, \tilde{\boldsymbol{U_j}}]\boldsymbol{x} = \boldsymbol{b_j} \tag{2.6}$$

$$\Rightarrow [\mathbb{1}_C, \boldsymbol{D_j}^{-1}\tilde{\boldsymbol{U_j}}]\boldsymbol{x} = \boldsymbol{D_j}^{-1}\boldsymbol{b_j} \tag{2.7}$$

Hierbei beschreibt $[\boldsymbol{D_j}, \tilde{\boldsymbol{U_j}}]$ das Aneinanderfügen der beiden Matrizen auf Spaltenebene. Die Zeilen innerhalb eines Chunks können nun parallel mit `SIMD` Befehlen gelöst werden. Das Produkt aus inversem Diagonalanteil und Rest muss nicht explizit ausgerechnet werden, sondern kann als zweifaches Matrix-Vektor-Produkt implementiert werden.
Mit dieser Vorgehensweise kann eine parallele Abarbeitung der Zeilen auf Chunk-Ebene erreicht werden. Damit diese Methode jedoch für Mehrkernprozessoren geeignet ist, muss sie zusätzlich Parallelität zwischen den Chunks aufweisen. Dies kann durch eine geschickte Sortierung erreicht werden. Sie sollte Zeilen ähnlicher Abhängigkeiten einem Chunk zuordnen. Andernfalls kann es passieren, dass für die Abarbeitung des nächsten Chunks alle Vorherigen gelöst sein müssen. Eine solche Aufteilung ist beispielsweise durch die Arrowhead-Sortierung gegeben. Ebenso muss es einfach möglich sein, einen Chunk in seine zwei Anteile zu zerlegen. Diese Methode wollen wir an dieser Stelle nur skizzieren, da eine Implementierung zusätzliche Änderungen des Matrix-Formats benötigt.

2.2 Implementierung

In diesem Abschnitt wollen wir eine kurze Einführung in die zugrundeliegenden Bibliotheken der Programme geben und dabei auch alternative Implementierungen ähnlicher Algorithmen beschreiben. Der Einfachheit halber haben wir alle Algorithmen nur in doppelter Genauigkeit und immer für eine obere Dreiecksmatrix implementiert.

2.2.1 `MKL`

Wir wollen unsere Implementierung mit der „Math Kernel Library“ (kurz `MKL`) von Intel vergleichen. Diese gilt als die Standardreferenz im Bereich wissenschaftlicher Berechnungen, da sie eine Vielzahl von Algorithmen für verschiedene Probleme in optimierter Form implementiert. Wir verwenden die `MKL` Version 2017.0. Zur Lösung des dünnbesetzten oberen Dreieckssystems haben wir die Methode `mkl_sparse_d_trsv` und das

CSR-Format verwendet. Zusätzlich wurde noch eine nicht weiter spezifizierte Optimierung mittels der Methoden `mkl_sparse_optimize` und `mkl_sparse_set_sv_hint` als Analyse-Phase durchgeführt. Da es sich um eine proprietäre Software handelt, kann nicht festgestellt werden, welcher Ansatz zur Parallelisierung verwendet wird. Wir gehen jedoch davon aus, dass die Methode des Level-Scheduling verwendet wird.

2.2.2 `CUDA` bzw. `OpenCL`

Wie bereits beschrieben, wurde die Synchronization-Free-Methode zuerst von Liu et al. in [14] vorgestellt. Sie implementierten die Methode als vorwärts gerichtetes Verfahren in `CUDA` bzw. `OpenCL` für Grafikprozessoren (GPUs). Dabei verwenden sie das CSC-Format zur Speicherung der Matrix, um direkten Zugriff auf die Spalten der Matrix zu haben. Ebenso ändern sie Algorithmus 3 ab, indem sie eine Aufteilung zwischen den Werten nahe und abseits der Diagonalen durchführen. Daher werden die bisher verwendeten Listen in jeweils zwei Neue aufgeteilt. Die Idee dieser Aufteilung liegt in der Optimierung der Lese- und Schreibzugriffe.
Aufgrund der gleichzeitigen Bearbeitung von Datenelementen, kann es zu Konflikten im Datenzugriff der einzelnen Prozessoren kommen. Daher müssen sogenannte `ATOMIC` Befehle verwendet werden. Diese Befehle erzwingen eine Blockierung des zu bearbeitenden Datenbereichs bis zum Abschluss des Befehls. Andere Berechnungen, die nicht diesen spezifischen Datenbereich benötigen, können jedoch ungehindert weitergeführt werden.
Ein Vorteil der Grafikprozessoren ist, dass sie zwei Level von parallelen Recheneinheiten besitzen. So können Aufgaben erst einem Block von Einheiten, Warp genannt, zugeordnet werden und dann innerhalb dieses Warps parallel ausgeführt werden. Dies ähnelt den `SIMD` Befehlen bei CPUs, ist jedoch umfangreicher, da bei CPUs keine `ATOMIC` Befehle auf `SIMD` Ebene durchführbar sind.

2.2.3 `GHOST`

Die Abkürzung `GHOST` steht für „general, hybrid and optimized sparse toolkit“ und bildet ein grundlegendes Gerüst für Berechnungen mit großen, dünnbesetzen Matrizen [8]. Es wird innerhalb des Projekts ESSEX im Programm SPPEXA der Deutschen Forschungsgemeinschaft [16] entwickelt. Dieses Softwarepaket ist in der Programmiersprache `C/C++` geschrieben und erlaubt die Benutzung verschiedener Hardware. Daher basiert es auf dem Programmieransatz `MPI+X`, wobei `X` durch verschiedene hardwarespezifische Programmieransätze innerhalb der Knoten gegeben ist. Beispiele sind `CUDA` für Grafikkarten und

`OpenMP` für Mehrkernprozessoren. Da wir eine Implementierung für ein Shared-Memory-System verfolgen, verwenden wir den Ansatz `OpenMP`. Das Softwarepaket `GHOST` enthält eine Implementierung des SELL-C-σ-Format zur Speicherung dünnbesetzter Matrizen. Dieses Format wurde gewählt, um Vektorisierung der Berechnungen und optimale Nutzung verschiedener Computerkomponenten zu ermöglichen. Entsprechend enthält das Paket optimierte Funktionen zur Berechnung von dünn- und dichtbesetzten Skalar- und Matrix-Vektor-Produkten. Im Zuge seiner weiteren Entwicklung sollen auch verschiedene Vorkonditionierer, wie die unvollständige LU-Zerlegung, implementiert werden.
Wird diese beispielsweise in Form der iterativen Methode von Chow et. al. [7] implementiert, so empfiehlt sich die Speicherung der Matrizen $\boldsymbol{L}$ in einem zeilenorientierten und $\boldsymbol{U}$ in einem spaltenorientierten Format. Damit kann $\boldsymbol{L}$ direkt im SELL-C-σ Format gespeichert werden, von $\boldsymbol{U}$ sollte jedoch die Transponierte gespeichert werden. Für das System $\boldsymbol{Lb} = \boldsymbol{b}'$ kann dann ein rückwärts gerichtetes Verfahren und ein vorwärts gerichtetes Verfahren zur Lösung von $\boldsymbol{Ux} = \boldsymbol{b}$ genutzt werden. Wir haben daher die Synchronization-Free-Methode für Prozessoren angepasst und sie in beiden Varianten implementiert. Für die vorwärts gerichtete Implementierung wurde der Algorithmus 3 ohne große Änderungen verwendet. Der einzige Unterschied liegt in der Aufteilung der zentralen Schleife in der Lösungs-Phase. Diese wurde in Blöcke der Größe C zerlegt, sodass jeder Prozessor immer einen Chunk löst. Diese Vorgehensweise haben wir gewählt, um einen besseren Speicherzugriff zu ermöglichen. Für das rückwärts gerichtete Verfahren musste eine neue Möglichkeit gefunden werden, abhängige Zeilen über den Status der gerade gelösten Zeile zu informieren. Das zuvor “passive“ Verhalten haben wir nun zu einem “aktiven“ Verhalten abgeändert. Entsprechend überprüft jede Zeile selbst, ob die Zeilen von denen sie abhängt, fertig sind. Diese Vorgehensweise hat den Vorteil, dass keine `ATOMIC` Befehle mehr nötig sind, da keine konkurrierenden Schreibprozesse stattfinden.
Wir haben ebenfalls die spezialisierte Methode zur Lösung eines Dreieckssystems in Arrowhead-Struktur implementiert. Dazu haben wir Algorithmus 4 wie beschrieben verwendet.

2.3 Ergebnisse

Im folgenden Abschnitt listen wir die Ergebnisse der Veröffentlichung durch Liu et al. und unsere Ergebnisse auf. Innerhalb der Tabellen und Grafiken haben wir Abkürzungen verwendet, die wir an dieser Stelle erläutern wollen. Die Abkürzungen sind Akronyme für die verwendeten Methoden (LSM = Level-Scheduling-Methode und SFM = Synchronization-Free-Methode). Ebenso haben wir für die Arrowhead-Methode die Abkürzung AR benutzt und für vorwärts bzw. rückwärts Ausrichtung des Verfahrens V bzw. R.

2.3.1 Laufzeituntersuchungen durch Liu et al.

Wir beziehen uns hier auf die Publikation durch Liu et al. [14]. Sofern nicht anders angegeben, entstammen die Angaben dieser Publikation. Dort wird im Gegensatz zu unserem Fall die Lösung des Systems $\boldsymbol{Lx} = \boldsymbol{b}$ behandelt. Die Algorithmen lassen sich jedoch entsprechend umformulieren und die Ergebnisse sind vergleichbar. Wir haben der Publikation nur die Ergebnisse der Messungen in doppelter Genauigkeit entnommen.

Hardware Die Autoren haben für ihre Messungen drei verschiedene Grafikprozessoren von zwei unterschiedlichen Herstellern verwendet. Zwei der drei GPUs stammen von Nvidia und werden daher in `CUDA` programmiert. Die dritte Karte stammt von AMD und wird daher in `OpenCL` programmiert. Die GPUs unterscheiden sich in Bezug auf Ihre Speichermenge, Bandbreite und Anzahl an Rechenkernen. Die genauen Eigenschaften lassen sich dem Absatz 4 der Publikation entnehmen.

Matrizen Die verwendeten Matrizen stammen aus der „University of Florida Sparse Matrix Collection“ [17] und sollen eine Auswahl repräsentativer Matrizen darstellen. In Tabelle 2.3 sind die von beiden Gruppen verwendeten Matrizen aufgelistet. Die Autoren haben diese Matrizen nach ihren Eigenschaften für die Level-Scheduling-Methode ausgewählt. So besitzen die Matrizen unterschiedlich viele Level-Sets und zeigen daher jeweils eine andere Charakteristik. Um aus den Matrizen eine untere Dreiecksmatrix zu erzeugen, haben die Autoren den unteren Dreiecksanteil der Matrizen verwendet und zusätzlich die Diagonale gegen eine vollbesetzte Diagonale getauscht. Bemerkenswert ist, dass die Anzahl an Nicht-Null-Einträgen der Matrizen in der Publikation nicht mit denen der Quelle [17] übereinstimmt.

Analyse-Phase Die Autoren haben die Laufzeitdauer der Analyse-Phase ihres Algorithmus explizit angegeben. In Tabelle 2.1 sind die Zeiten in Millisekunden für die gemeinsam verwendeten Matrizen aufgelistet. Leider finden wir in der Publikation keine Angabe zur Anzahl der Wiederholungen der Laufzeitmessungen. Die Autoren vergleichen ihre Implementierung mit der Level-Scheduling-Methode, welche in der `cuSparse` Bibiothek von Nvidia implementiert ist.

GPU [ms]	K40c		Titan X		Fury X
Methode	LSM	SFM	LSM	SFM	SFM
cantilever	8.92	0.10	8.28	0.16	0.07
ship_003	6.41	0.19	4.34	0.26	0.13
webbase-1M	8.53	0.19	5.48	0.13	0.11
nlpkkt160	40.58	7.27	19.99	8.91	5.58

Tabelle 2.1: Dauer der Analyse-Phase der Synchronization-Free-Methode (SFM) und der Level-Scheduling-Methode (LSM) bei Liu et al.

Performance beider Phasen In der Publikation finden wir keine genauen Angaben zur Dauer der Lösungs-Phase des Algorithmus. Dafür haben die Autoren die Performance beider Phasen in einem Balkendiagramm dargestellt. Die Performance wird hierbei in der Einheit GFlops/s gemessen. Wir haben die Werte der Grafik entnommen und in Tabelle 2.2 zusammengefasst.

GPU GFlops/s	K40c		Titan X		Fury X
Methode	LSM	SFM	LSM	SFM	SFM
cantilever	0.18	0.38	0.20	0.72	0.36
ship_003	0.54	0.41	0.52	0.91	0.54
webbase-1M	0.48	0.48	0.54	0.68	0.60
nlpkkt160	3.00	1.86	4.00	4.00	4.00

Tabelle 2.2: Performance der Synchronization-Free-Methode (SFM) und der Level-Scheduling-Methode (LSM) bei Liu et al.

2.3.2 Eigene Laufzeituntersuchungen

Hardware Wir verwenden für unsere Messungen eine Workstation mit zwei Prozessoren, welche auf der Shared-Memory-Architektur basiert. Jeder dieser Prozessoren (Intel

Xeon E5-2658v3) hat zwölf Kerne, wobei jeder Kern wiederum zwei virtuelle Kerne bereitstellen kann. Damit stehen insgesamt 48 Prozessorkerne für das Betriebssystem zur Verfügung. Je Prozessor sind 64 Gigabyte Speicher verbaut, sodass insgesamt 128 Gigabyte Speicher vorhanden sind. Für unsere Messungen wurde nur ein Prozessor benutzt und keiner der virtuellen Kerne verwendet. Dies bedeutet, dass der Prozessor nur auf Daten des nächstgelegenen Speichers zugreifen sollte (Vermeidung von NUMA-Effekten). Als Betriebssystem wurde SUSE Linux Enterprise in der Version 12.1 verwendet.

Matrizen In der Tabelle 2.3 sind die von uns untersuchten Matrizen aufgelistet. Diese stammen aus der „University of Florida Sparse Matrix Collection“ [17]. Es werden nur quadratische Matrizen verwendet, sodass die Anzahl der Zeilen und Spalten gleich ist. Einige der Matrizen wurden bereits von anderen Autoren [14] verwendet, daher sollte es sich um repräsentative Matrizen handeln. Diese Matrizen wurden durch eine unvollständige LU-Zerlegung in eine obere und eine untere Dreiecksmatrix zerlegt. Damit das entstehende Gleichungssystem lösbar ist und der Algorithmus für die ILU-Zerlegung konvergiert, darf die Diagonale der Matrix keine Null-Einträge enthalten. Daher sind einige der Matrizen symmetrisch positiv definit (SPD), da diese die gewünschte Bedingung erfüllen. Für die Zerlegung haben wir die Methode `dcsrilu0` der Intel `MKL` verwendet.
Für die zusätzliche Partitionierung in Arrowhead-Form haben wir uns für das Paket `METIS` [18] entschieden und dessen Funktion `METIS_PartGraphKway` genutzt. Es wurden 12 Partitionen erstellt, da dies der maximal verfügbaren Anzahl an Prozessoren entspricht.
Für die rechte Seite des zu lösenden Gleichungssystems wurde ein Vektor passender Dimension verwendet, dessen Einträge das Inverse der Matrixdimension sind. Damit konnte dann ein Gleichungssystem der Form $\boldsymbol{U}\boldsymbol{x} = \boldsymbol{b}$ aufgestellt und mit den bereits vorgestellten Algorithmen eine Lösung $\boldsymbol{x}$ ermittelt werden.
Aufgrund der verwendeten Hardware wurde die Matrix im SELL-4-1 Format gespeichert. Je Methode wurden hundert Läufe der Analyse-Phase und hundert Läufe der Lösungs-Phase durchgeführt. Dabei wurde zunächst noch ein extra Lauf beider Phasen zur Initialisierung durchgeführt.

Analyse-Phase In der folgenden Tabelle 2.4 haben wir die Dauer der Analyse-Phase für die betrachteten Implementierungen gesammelt. Da die serielle Implementierung und die Implementierung für die Arrowhead-Struktur keine Analyse-Phase besitzen, sind sie hier nicht zu finden. Die tabellierten Laufzeiten wurden bei der Verwendung unserer

Name	Zeilen/Spalten	*NNZ*	SPD	Liu et al.
cantilever	62451	4007383	N	Y
ship_003	121728	3777036	Y	Y
offshore	259789	4242673	Y	N
inline_1	503712	36816170	Y	N
webbase-1M	1000005	3105536	N	Y
nlpkkt160	8345600	225422112	N	Y

Tabelle 2.3: Verwendete Matrizen mit deren Größe, Anzahl an Nicht-Null-Einträgen und Eigenschaften beider Gruppen

maximalen Anzahl an Prozessoren $P = 12$ gemessen. Die Messungen mit kleinerer Prozessoranzahl sind der großen Tabelle auf Seite 33 im Anhang zu entnehmen.
In der Grafik 2.7 ist der Speedup der Synchronization-Free-Methode im Vergleich zur Intel `MKL` zu sehen. Dabei wurden die Laufzeiten der beiden Programme bei Nutzung der gleichen Prozessoranzahl ins Verhältnis gesetzt. Da die Messwerte der Analyse-Phase der `MKL` bei Nutzung der Arrowhead-Struktur nicht stark von der Analyse-Phase ohne Arrowhead-Struktur abweichen, haben wir darauf verzichtet hier einen Speedup zu bestimmen.

Methode [ms]	`MKL`	`MKL` AR	SFM R	SFM V
cantilever	0.27	0.18	0.07	0.07
ship_003	0.35	0.47	0.09	0.09
offshore	0.46	0.45	0.13	0.13
inline_1	3.90	2.08	0.26	0.25
webbase-1M	0.65	0.36	0.46	0.49
nlpkkt160	18.46	12.08	3.18	3.24

Tabelle 2.4: Dauer der Analyse-Phase der `MKL` und der implementierten Synchronization-Free-Methode (SFM) auf 12 Prozessoren

Lösungs-Phase Ähnlich der Analyse-Phase haben wir in der Tabelle 2.5 die Laufzeit der Lösungs-Phasen für die Algorithmen aufgelistet. Die Zeiten für die kleineren Anzahlen an Prozessoren sind ebenfalls in der großen Tabelle auf Seite 33 abgebildet. In der Grafik 2.7 ist der Speedup gegenüber der seriellen Implementierung für drei Matrizen dargestellt. Diese Matrizen wurden bereits bei der Darstellung des Speedups der Analyse-Phase in Grafik 2.6 verwendet.

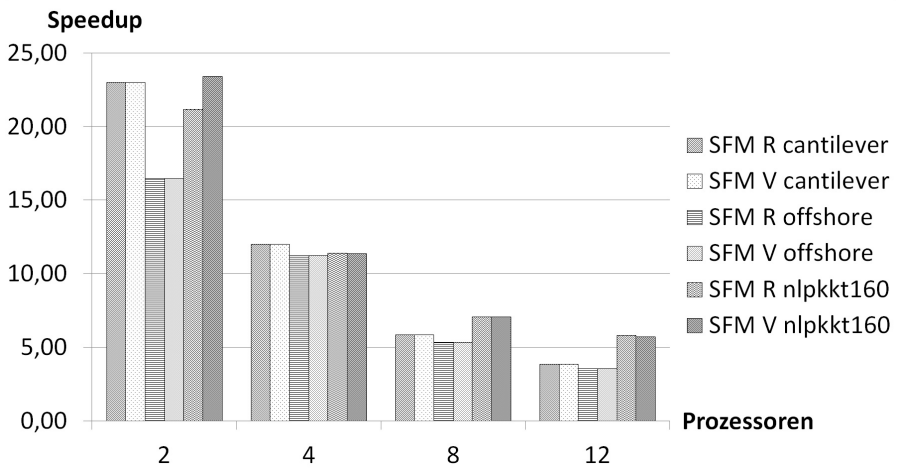


Abbildung 2.6: Speedup der Analyse-Phase der SFM im Vergleich zur `MKL` für drei Matrizen

Methode [ms]	Ser.	`MKL`	`MKL` AR	SFM R	SFM V	AR
cantilever	3.13	2.58	2.51	6.58	41.21	0.56
ship_003	10.24	7.47	6.92	17.46	90.42	2.02
offshore	6.44	6.16	5.75	14.89	56.56	1.11
inline_1	45.06	36.32	37.85	79.14	396.96	47.84
webbase-1M	8.30	6.54	5.73	13.74	43.15	2.36
nlpkkt160	259.08	210.16	208.69	89.31	571.39	50.48

Tabelle 2.5: Dauer der Lösungs-Phasen im Vergleich auf 12 Prozessoren

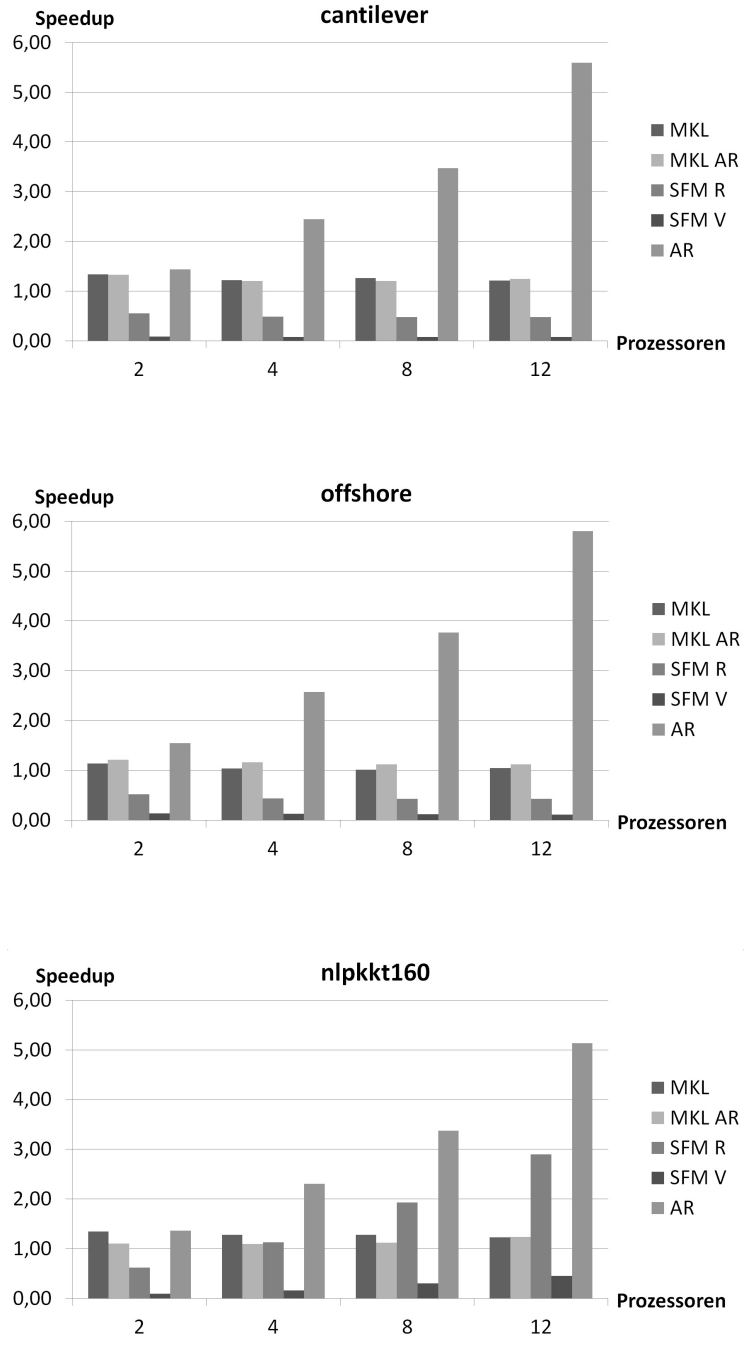


Abbildung 2.7: Speedup der Lösungs-Phase für drei Matrizen und verschiedene Implementierungen im Vergleich zur Seriellen

2.4 Vergleich

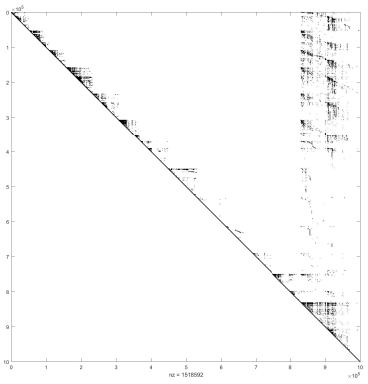
Nachdem wir die Ergebnisse von Liu et al. und unsere Ergebnisse aufgelistet haben, wollen wir diese nun miteinander vergleichen. Dazu wollen wir zunächst die beiden Phasen einzeln betrachten und anschließend die gesamte Implementierung.

Analyse-Phase Wie sowohl der Tabelle in der Publikation als auch Tabelle 2.1 entnommen werden kann, hat die Analyse-Phase der Synchronization-Free-Methode auf GPUs eine kürzere Laufzeit im Vergleich zu jener der Level-Scheduling-Methode. Dabei verzeichnen die Autoren einen mittleren Speedup von $S = 43.7 - 58.2$ bei einem Maximum von $S = 70.5 - 89.2$. Dabei stellen wir fest, dass, in der von den Autoren verwendeten Version, die Analyse-Phase für jeden Aufruf der gesamten Methode erneut durchgeführt werden muss, sodass es sich nicht um eine Analyse-Phase im eigentlichen Sinne handelt. In unserer Implementierung konnten wir die Analyse-Phase vereinfachen, da das SELL-C-σ Format die Anzahl an Einträgen in den Zeilen bzw. Spalten der Matrix abspeichert. Damit haben wir es sowohl für das vorwärts gerichtete als auch für das rückwärts gerichtete Verfahren geschafft, die Analyse-Phase zu optimieren. Vergleicht man unsere Implementierung mit der Analyse-Phase der `MKL` bei gleicher Anzahl an Prozessoren, siehe Grafik 2.6, so lässt sich ein Speedup $S = 17 - 23$ für zwei Prozessoren und $S = 3 - 6$ für zwölf Prozessoren feststellen. Der Speedup gegenüber der Analyse-Phase der `MKL` sinkt also mit zunehmender Anzahl der Prozessoren. Die Erklärung hierfür ist, dass die Berechnungen immer schlechter aufteilbar sind und somit der Overhead (Aufwand für die Aufteilung) für die Parallelisierung zu hoch wird. Im Vergleich zur Publikation fällt der Speedup in unseren Messungen geringer aus. Dies ist auf die unterschiedliche Hardware zurückzuführen. Die Analyse-Phase der Synchronization-Free-Methode benötigt wenige Berechnungen, dafür aber viel Interaktion mit dem Speicher. In diesem Bereich ist der Grafikprozessor ($\approx 300\,\mathrm{GB/s} - 500\,\mathrm{GB/s}$) dem gewöhnlichen Prozessor ($\approx 50\,\mathrm{GB/s} - 100\,\mathrm{GB/s}$) deutlich überlegen.
Die Verwendung von mehr Prozessoren führt jedoch bei der `MKL` zu einer Verbesserung der Laufzeit. Betrachtet man dazu die Tabelle auf Seite 33, so erkennt man eine ungefähre Halbierung der Laufzeit je Verdoppelung der Anzahl an Prozessoren. Dies deutet darauf hin, dass es sich um eine parallele Implementierung handelt, welche zusätzliche Prozessoren gut nutzen kann (Skalierung). Wie bereits beschrieben, ist in Tabelle 2.4 zu erkennen, dass die Analyse-Phase der `MKL` für die Matrizen mit und ohne Arrowhead-Struktur ungefähr gleich lange dauert. Damit generiert diese zusätzliche Sortierung hier keinerlei Speedup.

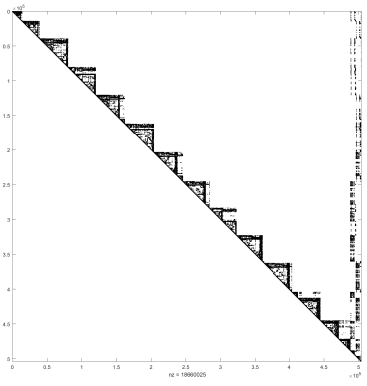
Lösungs-Phase Für die Lösungs-Phase der Synchronization-Free-Methode verzeichnen Liu et al. ebenfalls eine Verbesserung der Laufzeit im Vergleich zur Level-Scheduling-Methode. Der Speedup liegt im Mittel bei $S = 2.14$, bei einem Maximum von $S = 3.65$. Diese Ergebnisse können wir auf CPUs nicht erzielen, dort sind die Ergebnisse sowohl von der Struktur der Matrix als auch von der Ausrichtung des Verfahrens abhängig. Das vorwärts gerichtete Verfahren, welches der Implementierung der Autoren sehr ähnlich ist, zeigt einen Speedup von $S = 0.08 - 0.30$ im Vergleich zur seriellen Implementierung und ist damit langsamer als diese. Die rückwärts gerichtete Implementierung zeigt dagegen für die Matrix nlpkkt160 einen Speedup von maximal $S = 2.90$ bei der Verwendung von zwölf Prozessoren. Den Autoren nach lässt sich diese Matrix jedoch auch in genau zwei Level zerlegen und ist damit gut für die Level-Scheduling-Methode geeignet. Für die anderen Matrizen zeigt die Methode jedoch nur einen maximalen Speedup von $S = 0.48 - 0.69$. Damit ist auch diese Implementierung für die meisten Matrizen langsamer als der serielle Algorithmus. Wie man der Tabelle 2.5 entnehmen kann, ist das rückwärts gerichtete schneller als das vorwärts gerichtete Verfahren. Dies lässt sich durch den Verzicht auf `ATOMIC` Befehle im rückwärtigen Verfahren erklären.
Trotzdem zeigen beide Versionen im Vergleich zur Publikation einen geringeren Speedup. Erklärungen hierfür liegen sowohl im Matrixformat als auch in der jeweils genutzten Hardware. Da das SELL-C-σ Format auf Vektorisierung ausgelegt ist, befinden sich die Werte gleicher Zeile bzw. Spalte immer C Werte voneinander entfernt, in unserem Fall also immer vier Werte. Aufgrund der Blockstruktur des Speichers müssen, bei Schreib- und Lesevorgängen, immer drei dieser Werte übersprungen werden und können nicht genutzt werden. Dieses Problem kann mit der beschriebenen Methode der Blockinversion umgangen werden. Dazu muss sichergestellt sein, dass alle verbleibenden Abhängigkeiten innerhalb des Chunks liegen. Ebenso sind `ATOMIC` Befehle auf der GPU schneller durchzuführen als auf der CPU. Die GPU zeigt zudem ein höheres Level an Parallelität, da hier wesentlich mehr, jedoch auch einfachere, Recheneinheiten vorhanden sind.
Vergleicht man die Laufzeiten der Lösungs-Phase der `MKL` bei der Verwendung verschieden vieler Prozessoren, so stellt man fest, dass die Laufzeiten ungefähr gleich sind. Während sich die Laufzeit der Analyse-Phase je Verdopplung der Anzahl an Prozessoren ungefähr halbierte, ist sie hier konstant. Auch der Speedup von 1.01-1.35 fällt im Vergleich zu unserer einfachen, seriellen Implementierung gering aus. Dies gilt auch für die Matrix `nlpkkt160`, welche mit der Level-Scheduling-Methode besonders schnell zu lösen sein sollte. Damit stellen wir fest, dass es sich bei der Methode `mkl_sparse_d_trsv`

wahrscheinlich nicht um eine Level-Scheduling-Methode handelt. Die Lösungs-Phase ist also entweder ein schlechter paralleler Algorithmus oder eine rein serielle Implementierung. Gegebenenfalls sollte man einen weiteren Vergleich mit anderen Implementierungen durchführen. Eine Möglichkeit ist die Implementierung in `Trilinos` durch Heroux et al. [19].
Die zusätzlich durchgeführte Sortierung in unabhängige Blöcke (Arrowhead-Struktur) zeigt eine deutliche Verbesserung der Laufzeit des angepassten Algorithmus. Hier erreichen wir einen maximalen Speedup von $S = 5.8$ für die Matrix `offshore`. Desweiteren zeigt diese Methode ein wünschenswertes Verhalten bei der Erhöhung der Anzahl an Prozessoren (Skalierung). Hier erhöht sich der Speedup um etwas weniger als den Faktor zwei je Verdopplung der Prozessoren. Dies lässt sich durch den nicht parallel lösbaren Separator erklären. Ein Nachteil dieser Methode ist, dass die Größe des Separators mit der Anzahl der Blöcke zunimmt. Das bedeutet, je mehr Prozessoren man nutzen möchte, desto mehr Blöcke benötigt man und desto größer wird der serielle Anteil.
Eine Besonderheit bildet die Matrix `inline_1` mit circa einer halben Million Zeilen und 37 Millionen Einträgen. Die Implementierung zur Lösung des Dreieckssystems in Arrowhead-Struktur zeigt hier eine ähnliche Laufzeit, wie der serielle Algorithmus. In Grafik 2.8 ist das Besetztheitsstruktur dieser Matrix zu sehen. Vergleicht man es mit dem Muster der ebenfalls abgebildeten Matrix `webbase-1M` ($N \approx 1$ Mio. und $NNZ \approx 3$ Mio.) so erkennt man, dass bei der Matrix `inline_1` die Blöcke dichtbesetzer sind. Obwohl der Separator vergleichsweise schmal ausfällt, ist dieser recht dicht. Dadurch kommt es zu sehr vielen Lesezugriffen im Bereich hoher Indizes des Lösungsvektors. Ebenso müssen beim Lösen eines einzelnen Blocks viele Werte des Lösungsvektors der entsprechenden vorherigen Zeilen des gleichen Blocks verwendet werden. Wir vermuten, dass dies die Gründe für die hohe Laufzeit und geringen Nutzen der zusätzlichen Sortierung sind. Das bedeutet, dass die Matrix im Sinne der Lösung des Dreieckssystems zu viele Nicht-Null-Einträge im Vergleich zur Anzahl an Zeilen besitzt. Eine weitergehende Analyse, insbesondere der Auslastung des Speichers, kann hier weitere Erkenntnisse liefern.

Kombination beider Phasen Typischerweise wird die Lösung eines dünnbesetzten Dreieckssystems innerhalb eines Vorkonditionierers für ein iteratives Verfahren genutzt. Daher wird die Methode mehrfach aufgerufen und es ist durchaus akzeptabel eine zeitintensive Analyse-Phase zu verwenden, wenn dadurch gewährleistet wird, dass insgesamt ein Zeitgewinn entsteht.
Die Autoren von [14] geben zum Vergleich der Kombination beider Phasen nur die Performance der Verfahren an. Da nicht eindeutig erkennbar ist, wie diese gemessen wird,



webbase-1M



inline_1

Abbildung 2.8: Vergleich der Besetztheitsstruktur zweier Matrizen

haben die Werte keine große Aussagekraft. Das Problem ist, dass die verschiedenen Algorithmen unterschiedlich viele Rechenoperationen benötigen. Damit besitzt auch ein Algorithmus der viel Zeit benötigt, dabei aber viele Rechenoperationen durchführt, eine hohe Performance, obwohl dieser besonders lange benötigt, um zur Lösung zu gelangen. Diese Größe kann jedoch dabei helfen, zu beurteilen wie gut ein Algorithmus die gegebene Hardware nutzen kann. Eine bessere Aussage zur Geschwindigkeit des Algorithmus bietet die Zeitdauer bis zur Lösung (time to solution). Diese ist für einen einzelnen Lauf gegeben als die Summe aus den Laufzeiten für die Analyse-Phase und die Lösungs-Phase. Der Tabelle auf Seite 33 können alle diese summierten Laufzeiten entnommen werden. Wie schon für die Lösungs-Phase ergibt sich auch hier im Mittel nur ein Speedup für die Methode mit Ausnutzung der Arrowhead-Struktur.

3 Zusammenfassung

In der vorliegenden Arbeit haben wir die Lösung dünnbesetzter oberer Dreieckssysteme der Form $\boldsymbol{Ux} = \boldsymbol{b}$ auf modernen Mehrkernprozessoren betrachtet. Dazu haben wir zunächst eine Einführung in die Architektur moderner Computer gegeben, die Speicherung dünnbesetzter Matrizen behandelt und Algorithmen zur Lösung des Dreieckssystems verglichen. Im Anschluss haben wir mehrere Implementierungen, sowohl eigene als auch fremde, erläutert und schließlich eine Laufzeitmessung durchgeführt. Zuletzt wurden die Implementierungen und deren Eigenschaften verglichen.

Wir konnten die Ergebnisse unserer zentralen Referenz zum Teil auf gewöhnliche Prozessoren übertragen. So konnten wir zeigen, dass sich die Analyse-Phase der Synchronization-Free-Methode bei Nutzung eines fortschrittlichen Matrixformats weiter vereinfachen lässt. Die Portierung der Lösungs-Phase der Synchronization-Free-Methode konnte nicht erfolgreich durchgeführt werden. Im Rahmen der Arbeit war es nicht möglich, eine Implementierung mit Vektorisierung zu erreichen, jedoch konnten wir die Form des zu verwendenden Algorithmus erarbeiten und skizzieren. Eine optimierte Implementierung dieses Algorithmus würde auch die Probleme bei den Speicherzugriffen beseitigen und sollte daher im Vergleich eine bessere Performance aufweisen.
Bei der Verwendung einer zusätzlichen Sortierung der Matrix in eine Arrowhead-Struktur konnte eine Verkürzung der Laufzeit gegenüber der seriellen Implementierung, bei Verwendung eines spezialisierten Algorithmus, erreicht werden. Diese Implementierung zeigt zudem eine Verbesserung der Laufzeit bei zunehmender Anzahl an Prozessoren.
Zusammenfassend stellen wir fest, dass für alle betrachteten parallelen Algorithmen eine zusätzliche Sortierung der Matrix unerlässlich ist. Sowohl zur Reduzierung der Bandbreite, als auch zur Erhöhung der Parallelität, ist diese nötig. Ebenso wird eine solche Sortierung meist schon für die ILU-Zerlegung benötigt, daher sollte hier versucht werden, eine Partitionierung zu finden, welche beiden Algorithmen gerecht wird. Nachteilig ist, dass je nach Struktur der Matrix auch eine solche Sortierung keinen weiteren Vorteil generiert.

Um auf Synchronisation beim Abschluss eines Levels innerhalb der Level-Scheduling-Methode verzichten zu können, könnte es von Interesse sein, diese mit der Synchronization-Free-Methode zu kombinieren. Legt man durch die Level eine Sortierung der Matrix fest und verwendet die Lösungs-Phase der Synchronization-Free-Methode, so sollte sich diese auch für Prozessoren eignen, da innerhalb der Warte-Schleifen seltener verharrt werden muss. Eine Möglichkeit Vektorisierung nutzen zu können, haben wir mit der Blockinversion-Methode beschrieben. Hier muss ebenfalls eine Sortierung der Zeilen vorgenommen werden, um ähnlich abhängige Zeilen zu gruppieren.
Bei der Beurteilung der Verfahren zur Nutzung als Vorkonditionierer sollte das System als Ganzes betrachtet werden. Hierbei sollte immer eine Abwägung zwischen weiterer Reduzierung der Laufzeit des Lösers und der Komplexität bzw. Robustheit des Vorkonditionierers vorgenommen werden. In vollem Maße iterative Algorithmen, wie Referenz [7, 12], bieten zwar ein hohes Maß an Parallelität, erzeugen jedoch nur eine schlechte Approximierung. Ebenso ist das Einsatzfeld aufgrund der Anforderungen an die Matrix beschränkt. Direkte Algorithmen zeichnen sich durch ihre höhere Genauigkeit, aber auch geringere Parallelisierung aus. Bei der Nutzung zusätzlicher Beschleuniger zeigen sich durchaus bessere Ergebnisse als auf gewöhnlichen Prozessoren.

T [ms]	T Ana.	T Loes.	T Sum.	T Ana.	T Loes.	T Sum.	T Ana.	T Loes.	T Sum.	T Ana.	T Loes.	T Sum.	T Ana.	T Loes.	T Sum.	T Ana.	T Loes.	T Sum.
Seriell																		
Prozessoren	cantilever			ship_003			offshore			inline_1			webbase-1M			nlpkkt160		
1	0,00	3,13	3,13	0,00	10,24	10,24	0,00	6,44	6,44	0,00	45,06	45,06	0,00	8,30	8,30	0,00	259,08	259,08
MKL																		
Prozessoren	cantilever			ship_003			offshore			inline_1			webbase-1M			nlpkkt160		
2	0,92	2,34	3,26	1,68	6,53	8,21	1,81	5,65	7,46	9,05	35,99	45,04	1,29	6,41	7,70	80,43	192,25	272,68
4	0,48	2,56	3,04	0,96	6,93	7,89	1,12	6,22	7,34	6,57	37,29	43,86	1,01	6,54	7,55	47,72	202,93	250,65
8	0,35	2,48	2,83	0,53	6,85	7,38	0,64	6,37	7,01	5,17	37,36	42,53	0,80	6,37	7,17	23,86	202,56	226,42
12	0,27	2,58	2,85	0,35	7,47	7,82	0,46	6,16	6,62	3,90	36,32	40,22	0,65	6,54	7,19	18,46	210,16	228,62
MKL Arrowhead-Struktur																		
Prozessoren	cantilever			ship_003			offshore			inline_1			webbase-1M			nlpkkt160		
2	0,94	2,35	3,29	1,77	6,54	8,31	1,87	5,31	7,18	9,05	36,00	45,05	1,04	6,58	7,62	54,64	234,80	289,44
4	0,48	2,59	3,07	1,01	7,27	8,28	1,05	5,55	6,60	5,31	37,58	42,89	0,75	5,97	6,72	32,58	236,38	268,96
8	0,35	2,59	2,94	0,82	7,42	8,24	0,85	5,74	6,59	3,70	39,82	43,52	0,51	6,29	6,80	21,51	230,44	251,95
12	0,18	2,51	2,69	0,47	6,92	7,39	0,45	5,75	6,20	2,08	37,85	39,93	0,36	5,73	6,09	12,08	208,69	220,77
Synchronization-Free-Methode Rückwärts																		
Prozessoren	cantilever			ship_003			offshore			inline_1			webbase-1M			nlpkkt160		
2	0,04	5,68	5,72	0,06	16,18	16,24	0,11	12,33	12,44	0,35	69,95	70,30	0,68	11,95	12,63	3,80	418,30	422,10
4	0,04	6,39	6,43	0,06	17,17	17,23	0,10	14,82	14,92	0,28	73,13	73,41	0,57	12,97	13,54	4,19	229,46	233,65
8	0,06	6,56	6,62	0,07	17,48	17,55	0,12	14,96	15,08	0,26	79,38	79,64	0,52	13,26	13,78	3,38	134,02	137,40
12	0,07	6,58	6,65	0,09	17,46	17,55	0,13	14,89	15,02	0,26	79,14	79,40	0,49	13,74	14,23	3,18	89,31	92,49
Synchronization-Free-Methode Vorwärts																		
Prozessoren	cantilever			ship_003			offshore			inline_1			webbase-1M			nlpkkt160		
2	0,04	35,58	35,62	0,06	75,50	75,56	0,11	46,24	46,35	0,35	348,97	349,32	0,68	37,69	38,37	3,44	2785,99	2789,43
4	0,04	37,57	37,61	0,06	80,96	81,02	0,10	49,31	49,41	0,28	370,26	370,54	0,57	39,30	39,87	4,21	1622,44	1626,65
8	0,06	40,30	40,36	0,08	86,41	86,49	0,12	54,41	54,53	0,26	395,05	395,31	0,53	41,75	42,28	3,39	855,88	859,27
12	0,07	41,21	41,28	0,09	90,42	90,51	0,13	56,56	56,69	0,25	396,96	397,21	0,49	43,15	43,64	3,24	571,39	574,63
Methode direkt für Arrowhead-Struktur																		
Prozessoren	cantilever			ship_003			offshore			inline_1			webbase-1M			nlpkkt160		
2	0,00	2,18	2,18	0,00	5,75	5,75	0,00	4,16	4,16	0,00	57,58	57,58	0,00	5,26	5,26	0,00	189,98	189,98
4	0,00	1,28	1,28	0,00	4,09	4,09	0,00	2,50	2,50	0,00	46,28	46,28	0,00	3,80	3,80	0,00	112,52	112,52
8	0,00	0,90	0,90	0,00	2,77	2,77	0,00	1,71	1,71	0,00	42,21	42,21	0,00	3,03	3,03	0,00	76,74	76,74
12	0,00	0,56	0,56	0,00	2,02	2,02	0,00	1,11	1,11	0,00	47,84	47,84	0,00	2,36	2,36	0,00	50,48	50,48

Tabelle 4.1: Zusammenfassung aller Laufzeitmessungen

Literaturverzeichnis

[1] Lam Lay Yong, *Jiu zhang suanshu (nine chapters on the mathematical art): An overview*, Archive for History of Exact Sciences **47** (1994), no. 1, 1–51.

[2] PD Dr. Christian Kanzow (auth.) Professor Dr. Carl Geiger, *Numerische Verfahren zur Lösung unrestringierter Optimierungsaufgaben*, 1st ed., Springer-Lehrbuch, Springer-Verlag Berlin Heidelberg, 1999.

[3] Prof. Dr. F. L. Bauer (eds.) Dr. J. H. Wilkinson F.R.S. Dr. C. Reinsch (auth.), *Linear algebra*, 1st ed., Handbook for Automatic Computation 2, Springer-Verlag Berlin Heidelberg, 1971.

[4] Prof. Dr. Gerd Fischer (auth.), *Lineare Algebra*, 14, durchges. Aufl., Vieweg Studium, Vieweg+Teubner Verlag, 2003.

[5] James H. Wilkinson, *Rounding errors in algebraic processes* (1994).

[6] Y. Saad, *Iterative methods for sparse linear systems*, 2nd ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[7] Edmond Chow and Aftab Patel, *Fine-grained parallel incomplete LU factorization*, SIAM Journal on Scientific Computing **37** (2015), no. 2, C169–C193, available at `https://doi.org/10.1137/140968896`.

[8] Moritz Kreutzer, Jonas Thies, Melven Röhrig-Zöllner, Andreas Pieper, Faisal Shahzad, Martin Galgon, Achim Basermann, Holger Fehske, Georg Hager, and Gerhard Wellein, *GHOST: Building blocks for high performance sparse linear algebra on heterogeneous systems*, International Journal of Parallel Programming (2016), 1–27.

[9] Hesham El-Rewini and Mostafa Abd-El-Barr, *Advanced computer architecture and parallel processing*, Wiley-Interscience, 2005.

[10] P. P. Gelsinger, *Microprocessors for the new millennium: Challenges, opportunities, and new frontiers*, 2001 IEEE International Solid-State Circuits Conference. Digest of Technical Papers. ISSCC (Cat. No.01CH37177) (2001).

[11] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, *A unified sparse matrix data format for efficient general sparse matrix-vector multiply on modern processors with wide simd units*, ArXiv e-prints (July 2013), available at `1307.6209`.

[12] Hartwig Anzt, Edmond Chow, and Jack Dongarra, *Iterative sparse triangular solves for preconditioning*, Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings (2015), 650–661.

[13] Michael M. Wolf, Michael A. Heroux, and Erik G. Boman, *Factors impacting performance of multithreaded sparse triangular solve*, High Performance Computing for Computational Science – VECPAR 2010: 9th International conference,Berkeley, CA, USA, June 22-25, 2010, Revised Selected Papers (2011), 32–44.

[14] Weifeng Liu, Ang Li, Jonathan Hogg, Iain S. Duff, and Brian Vinter, *A synchronization-free algorithm for parallel sparse triangular solves*, Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings (2016), 617–630.

[15] Iain S. Duff and Gérard A. Meurant, *The effect of ordering on preconditioned conjugate gradients*, BIT Numerical Mathematics **29** (1989), no. 4, 635–657.

[16] Deutsche Forschungsgemeinschaft DFG, *German Priority Programme 1648, SPPEXA Software for Exascale Computing*, 2017.

[17] Timothy A. Davis and Yifan Hu, *The University of Florida Sparse Matrix Collection*, ACM Trans. Math. Softw. **38** (December 2011), no. 1, 1:1–1:25.

[18] George Karypis and Vipin Kumar, *A fast and highly quality multilevel scheme for partitioning irregular graphs*, SIAM Journal on Scientific Computing **20** (1999), no. 1, 359—392.

[19] Michael Heroux, Roscoe Bartlett, Vicki Howle Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams, *An overview of trilinos* **SAND2003-2927** (2003).

[20] Maxim Naumov, *Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU*, Nvidia Technical Reports (2011).

Danksagung

Ich danke der Abteilung SC-HPC des DLR unter der Leitung von Herrn Dr. Achim Basermann herzlich für die gute Betreuung. Insbesondere möchte ich mich an dieser Stelle bei Herrn Dr. Jonas Thies für die Hilfe und Anleitung beim Erstellen dieser wissenschaftlichen Arbeit und der Lösung zahlreicher programmiertechnischer Probleme bedanken. Zugleich gilt mein Dank Herrn Prof. Dr. Axel Klawonn für die Unterstützung, meine Bachelorarbeit in Kooperation der Universität zu Köln und dem DLR zu verfassen.

Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne die Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden. Ich versichere, dass die eingereichte elektronische Fassung der eingereichten Druckfassung vollständig entspricht.

Ort, Datum Unterschrift