Ein hochskalierbarer paralleler direkter Löser für Finite Elemente Diskretisierungen

Zur Erlangung des akademischen Grades eines

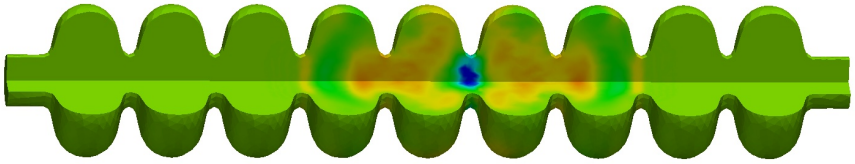
DOKTORS DER NATURWISSENSCHAFTEN

von der Fakultät für Mathematik des
Karlsruher Instituts für Technologie (KIT)
genehmigte

DISSERTATION

von

Dipl.-Math. techn. Daniel Maurer
aus Wiesbaden



Tag der mündlichen Prüfung: 22.05.2013

Referent:	Prof. Dr. Christian Wieners, Karlsruher Institut für Technologie
Korreferent:	PD Dr. Nicolas Neuß, Universität Erlangen

Danksagung

Die vorliegende Arbeit entstand in der Zeit von Oktober 2008 bis April 2013 im Rahmen meiner Tätigkeit als akademischer Angestellter in der Arbeitsgruppe *Wissenschaftliches Rechnen* am *Institut für Angewandte und Numerische Mathematik* des *Karlsruher Institut für Technologie (KIT)*. In der Zeit von Januar 2009 bis Juni 2012 war die Arbeit zusätzlich Teilprojekt des BMBF-Verbundprojekts *ASIL - Advanced Solvers Integrated Library - Schnelle, hochskalierbare Löser für komplexe Systeme* mit dem Förderkennzeichen 01IH08014F.

An erster Stelle geht mein Dank an Herrn Prof. Dr. Christian Wieners für die Aufnahme in seine Arbeitsgruppe und die Auswahl des spannenden Themas der vorliegenden Arbeit. Insbesondere sind die herausragenden Arbeitsbedingungen und die stets freundliche und engagierte Betreuung während meiner Promotionszeit hervorzuheben. Durch die von ihm maßgeblich entwickelte Software-Toolbox `M++` war es mir möglich, einen schnellen Zugang zur parallelen Programmierung einer Block-LR-Zerlegung für Finite Elemente Probleme zu erhalten.

Für die Übernahme des Korreferats danke ich Herrn PD Dr. Nicolas Neuß aus Erlangen. Zu Anfang meiner Promotionszeit war er noch in Karlsruhe tätig und hatte immer ein offenes Ohr für mich.

Herr Dr.-Ing. Wigand Koch von CST Darmstadt (Computer Simulation Technology) stellte mir zwei Geometrien zur Verfügung (ein Beschleunigungsresonator, welcher auch auf der Titelseite zu sehen ist, sowie ein Gehäuse mit Schlitz). Dafür, dass ich diese Geometrien für meine Berechnungen verwenden konnte, möchte ich ihm ebenfalls herzlich danken.

Weiterhin möchte ich mich bei allen Kolleginnen und Kollegen am Lehrstuhl für wissenschaftliches Rechnen für die freundliche und kooperative Arbeitsatmosphäre bedanken, vor allem die (nicht-) wissenschaftlichen Diskussionen in Raum 4C-10 waren eine tägliche Motivation. Insbesondere sind beziehungsweise waren dies Dr. Wolfgang Müller, Dr. Martin Sauter, Dr. Tudor Udrescu, Prof. Dr. Tobias Jahnke, Prof. Dr. Andreas Rieder, Tim Kreutzmann, Robert Winkler, Andreas Arnold, Michael Kreim, Andreas Schulz, Ekkachai Thawinan, sowie Nathalie Sonnefeld und Sonja Becker.

Schließlich danke ich meiner Familie für die Unterstützung jedweder Art, insbesondere meinen Eltern Heidi und Rüdiger Maurer und meinem Bruder Benjamin Maurer.

Inhaltsverzeichnis

Teil 2. Die parallele Block-LR-Zerlegung

Teil 3. Anwendung

Anhang

Motivation

Numerische Simulationen ermöglichen der Wissenschaft und Technik oftmals große Einsparungen an Geld und Zeit. Sie ersetzen in vielen Fällen den Bau von Prototypen, an denen ausgiebige Tests, wie zum Beispiel Belastungstests, Festigkeit und Crash, durchgeführt werden müssen. Die Prototypen werden am Computer modelliert, die Tests werden simuliert. Häufig eignen sich dazu Finite Elemente-Methoden [**Bra03**]. Um eine gute, möglichst genaue Simulation zu erhalten, müssen dabei lineare Gleichungssysteme der Art $Au = f$ mit mehreren Millionen Freiheitsgraden gelöst werden.

Die aus der Finiten Elemente-Methode entstehenden Gleichungssysteme sind in der Regel sehr schwach besetzt. Diese können sowohl direkt als auch iterativ gelöst werden [**Hac91**]. Bei einem iterativen Verfahren wird der Algorithmus mehrfach hintereinander angewandt, so dass sich die berechnete Lösung der Originallösung immer weiter annähert. Die Iteration wird abgebrochen, wenn bestimmte Voraussetzungen erfüllt sind, zum Beispiel, wenn das Residuum klein genug ist. Es kann jedoch geschehen, dass das Verfahren sehr langsam oder auch gar nicht konvergiert. Mit Hilfe eines Vorkonditionierers kann die Konvergenzgeschwindigkeit beschleunigt werden. Allerdings sind viele iterativen Verfahren auf ihre Anwendungsklasse beschränkt. Zum Beispiel kann ein cg-Verfahren nur auf symmetrisch positiv definite Systeme angewandt werden. Ein symmetrisch positive definites System erhält man bei einem Laplace-Problem oder in der linearen Elastizität. Viele Probleme führen allerdings auf unsymmetrische (zum Beispiel Konvektions-Diffusions-Probleme) und indefinite (Stokes- und Maxwell-Probleme [**Mon03**]) Gleichungssysteme. Hierfür können beispielsweise GMRES-Verfahren eingesetzt werden.

Als ein leistungsfähiger und effizienter Vorkonditionierer haben sich Mehrgitterverfahren herausgestellt [**Bra93**, **Hac03**]. Dort wird ein guter Grobgitterlöser benötigt, wobei sich ein direkter Löser hierbei dadurch eignet, da die gesuchte Lösung nach einer festen Anzahl von Schritten feststeht[1] und nicht iterativ gelöst werden muss. Bei einer LR-Zerlegung einer Matrix A muss die Zerlegung selbst nur einmal berechnet werden. Diese Zerlegung kann dann auf eine oder mehrere rechte Seiten angewandt werden, wodurch die Lösung nach nur einem Schritt feststeht (sofern die Anwendung der Zerlegung als ein Schritt aufgefasst wird). In der

[1] Das cg-Verfahren bzw. GMRES-Verfahren ist eigentlich auch ein direktes Verfahren, da es nach N Schritten theoretisch die exakte Lösung liefert (bei einer Matrix $A \in \mathbb{R}^{N \times N}$). Eine approximativ gute Lösung wird jedoch meist schon für deutlich weniger Schritte erreicht

Praxis muss jedoch beachtet werden, dass sich bei direkten Verfahren Rundungsfehler durch Gleitkommaberechnungen stärker auswirken können, wenn die Größe der Matrix A wächst. Eine Einführung in die LR-Zerlegung mit Fehleranalyse wird in Kapitel 1 dargestellt.

Nach dem Mooreschen Gesetz verdoppelt sich die Anzahl der Transistoren pro Flächeneinheit auf einem Computerchip alle ein bis zwei Jahre [M+98]. Damit steigert sich auch die Rechenleistung der Computer. Bis Anfang dieses Jahrtausends wurde die Verdoppelung meist noch innerhalb eines Hauptprozessors erreicht, gegenwärtig wird die Leistungsfähigkeit durch Multicore-Prozessoren gesteigert. Ein Programm kann jedoch ohne Parallelisierung nur von einem Prozessorkern abgearbeitet werden. Eine Steigerung der Effektivität kann heutzutage also nur durch eine parallele Verarbeitung erreicht werden. Somit müssen die Programme zum Lösen der Probleme parallelisiert werden.

Durch einen Zusammenschluss mehrerer Rechner zu einem Cluster lassen sich weiterhin große parallele Rechenmaschinen erstellen. Dies hat gleich mehrere Vorteile: Mehrere kleinere Rechner sind meist kostengünstiger als ein gleichwertiger "Superrechner", der in der Summe die gleiche Rechenleistung besitzt. Durch Zusammenfügen der kleineren Rechner wird gleichzeitig auch der verfügbare Speicher erhöht. Außerdem können zusätzliche Rechner meist ohne großen Aufwand hinzugefügt werden, um die Rechenleistung weiter zu steigern.

Inzwischen existieren hochparallele Rechner mit mehreren Hunderttausend Kernen, wie beispielsweise die Cray XE6 (Hermit) in Stuttgart [Her], auf der die meisten Tests in dieser Arbeit gerechnet wurden. Daher müssen parallele Algorithmen entwickelt werden, die selbst bei einer großen Prozessorzahl noch gut skalieren.

Im Bereich der Finite-Elemente-Verfahren wird zur Parallelisierung das Gebiet Ω auf alle zu verwendeten Prozessoren verteilt (Domain Decomposition). Eine optimale Aufteilung des Gebiets auf die Prozessoren ist ein graphentheoretisches Problem [KK98], eine einfache Aufteilung kann über die rekursive Koordinaten-Bisektion (RCB) [KK95] oder für kleinere Probleme und wenige Prozessoren auch über den Spektral-Bisektion-Algorithmus (RIB – recursive inertia bisection) [PSL90] erreicht werden. In unserem Fall verwenden wir neben dem RCB-Algorithmus einen angeschlossenen Graphpartitionierer von Christian Schulz und Peter Sanders namens `Kaffpa`[2] [SS11, OSS12], welcher am Karlsruher Institut für Technologie im Rahmen der Dissertation von Christian Schulz geschrieben wurde [Sch13]. Jeder Prozessor rechnet nun lokal auf dem ihm zugewiesenen Gebiet und kommuniziert über MPI-Routinen (vergleiche Anhang B) mit den anderen Prozessoren.

Viele numerische Verfahren können parallelisiert werden. Dies beginnt bei einer gewöhnlichen Matrixmultiplikation [Fro90, OG96] und geht über Parallelisierung von Jacobi- und Gauß-Seidel-Verfahren [ALO02] sowie Mehrgitterverfahren [Zum03]. Weiterhin existieren bereits eine Vielzahl von parallelen (Sparse)-Matrix-Lösern, insbesondere zu nennen sind hierbei MUMPS[3] [ADLK01], Pardiso[4][SG02, SGFS01, SWH07], SPIKE [PS07] für Bandmatrizen, PaStiX[5] [PPJ02] und

[2] **Ka**rlsruhe **F**ast **F**low **Par**titioner
[3] a **MU**ltifrontal **M**assively **P**arallel sparse direct **S**olver
[4] **PA**rallel **DI**rect **SO**lver
[5] **Pa**rallel **S**parse ma**triX** package

PETSc[6] [**BGMS97**]. Ein anderer Ansatz geht über die von Hackbusch eingeführten $\mathcal{H}$-Matrizen[7][**Hac09**] und als Erweiterung die $\mathcal{H}^2$-Matrizen[**Bö09**], wobei die Ursprungsmatrix durch Niedrigrangmatrizen approximiert wird. Somit können insbesondere Matrix-Matrix-Multiplikationen effizient berechnet werden.

Aufbau der Arbeit

In Teil 1 (Einführung) wird in Kapitel 1 zunächst eine allgemeine LR-Zerlegung einer Matrix A mit einer Fehleranalyse für Gleichkommasysteme angegeben. Das hier verwendete parallele Programmiermodell mit den parallelen Finiten Elemente wird in Kapitel 2 eingeführt. Dies ist die Grundlage der Implementierung der parallelen Block-LR-Zerlegung.

Teil 2 (Die parallele Block-LR-Zerlegung) beschäftigt sich insbesondere mit der Konstruktion des Algorithmus (Kapitel 3). Die Implementierung in das von uns verwendete Programm `M++` wird in Kapitel 4 vorgestellt. Den Abschluss dieses Teils bildet eine Komplexitätsanalyse für ein Laplace-Problem auf einem Einheitswürfel (Kapitel 5). Dort wird eine Vorhersage der Berechnungs- und Kommunikationszeit angegeben und mit der tatsächlichen Zeit verglichen.

Im abschließenden Teil 3 (Anwendung) formulieren wir zunächst mehrere Modellprobleme (Kapitel 6) (Poisson, Stokes, Elastizität, Maxwell) und untersuchen in Kapitel 7 die Effizienz des Algorithmus aus Kapitel 3.

[6] **P**ortable **E**xtensible **T**oolkit for **S**cientific **c**omputing
[7] Hierarchical Matrix

Bezeichnungen

Allgemein

A	zu zerlegende Matrix der Dimension N
A^p	lokale Matrix auf Prozessor p der Dimension N^p
A_{km}	Untermatrix von A bezogen auf π_k und π_m
A^p_{km}	lokale Untermatrix auf Prozessor p
f	rechte Seite
f_k	Teil der rechten Seite bezogen auf π_k
K	Anzahl der Interfaces
k	Zählvariable für die Interfaces
$K_{s,t}$	Index des letzten Interfaces des Clusters t in Schritt s
$\mathcal{K}^{s,t}$	tatsächliche Operationsmenge
$\mathcal{K}^{s,t}_{\mathrm{LR}}$	zu zerlegende Operationsmenge
$\mathcal{K}^{s,t}_{\Delta}$	Komplement zu $\mathcal{K}^{s,t}$
n, m	Zählvariablen für die Blockmatrizen
N	Anzahl der Unbekannten
N^p	Anzahl der lokalen Indizes auf Prozessor p
Ω	Gebiet im $\mathbb{R}^d$, $d = 2, 3$
π_k	Interface (Prozessor-Set)
Π	Set aller möglichen Interfaces
$\Pi_{\mathcal{I}}$	aktives Prozessor-Set ($\Pi_{\mathcal{I}} = \{\pi_1, \ldots, \pi_K\}$)
Π^s, $\Pi^{s,t}$	aktives Prozessor-Set im Schritt s (für Cluster t)
p, q	Zählvariablen für Prozessoren
P	Anzahl der beteiligten Prozessoren (meist $P = 2^S$)
$\mathcal{P}$	Prozessormenge, $\mathcal{P} = \{1, \ldots, P\}$
$\mathcal{P}^{s,t}$	kombinierte Prozessormenge (Prozessoren, die im Schritt s an Cluster t arbeiten)
s	Schrittnummer
S	Gesamtzahl der Schritte, als Matrix: Schur-Komplement
t	Clusternummer
T_s	Anzahl der Cluster im Schritt s
Z	die aktuell zu zerlegende Matrix

Parallele Finite Elemente

c	Zelle aus $\mathcal{C}$
$\mathcal{C}$	globale Menge der Zellen
$\mathcal{C}^p$	Menge der Zellen auf Prozessor p
$\mathcal{V}, \mathcal{E}, \mathcal{F}$	globale Menge der Ecken, Kanten, Seitenflächen
$\mathcal{V}_c, \mathcal{E}_c, \mathcal{F}_c$	Ecken, Kanten, Seitenflächen zu einer Zelle $c \in \mathcal{C}$
i, j	hier i.A.: Index aus $\mathcal{I}$
$\mathcal{I}$	globales Index-Set
$V_\mathcal{I}$	Finite Elemente Raum
ϕ_i	Basisfunktion von $V_\mathcal{I}$
$\mathcal{I}^p$	Index-Set auf Prozessor p
$\mathcal{I}_\Gamma$	Indexmenge auf dem Interface
$\mathcal{I}_\Omega^p$	lokale Indexmenge auf Prozessor p
$\mathcal{I}_\Omega$	Indexmenge ohne Interface
Ω^p	Teilgebiet von Ω auf Prozessor p
Γ	Interface aller Teilgebiete
Γ^{pq}	Interface der Teilgebiete von Prozessor p und q
$\pi(z), \pi(i)$	mengenwertige Abbildung auf die beteiligten Prozessoren
f	diskretes Funktional
$\underline{f}$	Gesamtheit der Koeffizientenvektoren f^p
$\underline{f}^p$	Koeffizientenvektor von f auf Prozessor p
v	Finite Elemente Funktion
$\underline{v}$	Gesamtheit der Koeffzientenvektoren von v
$\underline{v}^p$	lokale Einschränkung von v auf Prozessor p
z	Knotenpunkt, Schlüssel in einer Hashtabelle
$\mathcal{Z}$	Schlüsselmenge

Teil 1

Einführung

1

Eine LR-Zerlegung einer Matrix A

Um ein lineares Gleichungssystem $Au = f$ zu lösen, gibt es verschiedene Ansätze [**GL96**, **DH02**]. Ein Standardlösungsverfahren ist dabei das Gaußsche Eliminationsverfahren, mit dessen Hilfe eine LR-Zerlegung der Matrix A mit einer unteren Dreiecksmatrix L und einer oberen Dreiecksmatrix R berechnet werden kann. Als symmetrische Variante wird weiterhin die LDL^T-Zerlegung benutzt, wobei L eine untere Dreiecksmatrix ist und D eine Diagonalmatrix. Für die LR-Zerlegung gibt es unterschiedliche Möglichkeiten, diese Zerlegungen zu berechnen. Auch die Art der Einheitsdiagonale (ob in L oder R) ist prinzipiell frei wählbar. Wenn die Zerlegung blockweise betrachtet wird, können sogar innerhalb eines Algorithmus unterschiedliche Ansätze der Zerlegung gewählt werden.

In Kapitel 1.2 wird zunächst dargestellt, wie ein triangulares System gelöst wird, welches später auch in der Block-LR-Zerlegung benutzen werden wird. Im darauf folgenden Kapitel 1.3 ist die Triangulierung beschrieben [**GL96**, **Kan04**]. Die Fehleranalyse in Bezug auf Rundungsfehler, welche sich in Computersystemen generell ergeben [**Hig96**] ist in Kapitel 1.4 zu finden.

Allgemein gilt die Annahme, dass alle Operationen gültig sind und insbesondere bei einer Division nicht durch 0 geteilt wird. Dass dies in den betrachteten Matrizen aus dem Finite Elemente Verfahren gegeben ist, wird in Kapitel 3.9 gezeigt.

Nach der Einführung der Notation stellen wir in Kapitel 1.2 und 1.3 die Grundalgorithmen einer LR-Zerlegung vor, wobei wir weitgehend [**GL96**] folgen. Für die Parallelisierung wird später in Kapitel 3.2 eine Block-Variante betrachtet.

1.1. Notation

1.1.1. Matrixnotation.

Für eine Matrix $A \in \mathbb{R}^{M \times N}$ mit

$$A = (a_{mn}) = \begin{pmatrix} a_{11} & \cdots & a_{1N} \\ \vdots & & \vdots \\ a_{M1} & \cdots & a_{MN} \end{pmatrix}$$

bezeichnet

$$A[:,k] = \begin{pmatrix} a_{1k} \\ \vdots \\ a_{Mk} \end{pmatrix}$$

die k-te Spalte und

$$A[k, :] = \begin{pmatrix} a_{k1} & \cdots & a_{kN} \end{pmatrix}$$

die k-te Zeile der Matrix A. Mit

$$A[i:j, k:l] = (a_{mn})_{m=i,\ldots,j;n=k,\ldots,l} = \begin{pmatrix} a_{ik} & \cdots & a_{il} \\ \vdots & & \vdots \\ a_{jk} & \cdots & a_{jl} \end{pmatrix}$$

wird eine Untermatrix von A bezeichnet.

1.1.2. Algorithmennotation.
Die Kosten eines Algorithmus werden in flops angegeben. Dabei bezeichnen wir mit einem *flop* eine Gleitkomma-Operation im Rechner $+, -, \cdot, /$. Gewöhnlich beschränken wir uns auf die höchste Ordnung, das heißt, wenn ein Algorithmus mit $\frac{2}{3}N^3$ flops angegeben ist, so werden im Allgemeinen $\frac{2}{3}N^3 + O(N^2)$ flops benötigt.

1.2. Triangulare Systeme

In diesem Kapitel wird die Berechnung der Lösung eines Gleichungssystems mit einer triangularen Matrix betrachtet. Dabei muss zwischen einer unteren Dreiecksmatrix L und einer oberen Dreiecksmatrix R unterschieden werden. Erstere führt auf die sogenannte Vorwärts-Substitution (Kapitel 1.2.1), letztere auf die Rückwärts-Substitution (Kapitel 1.2.2). In Kapitel 3.2.1 wird auf eine Block-Variante eingegangen, hier exemplarisch für eine untere Block-Dreiecksmatrix.

1.2.1. Vorwärts-Substitution.
Gegeben sei ein reguläres lineares unteres triangulares Gleichungssystem $Lu = f$,

$$\begin{pmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ l_{N1} & l_{N2} & \cdots & l_{NN} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_N \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_N \end{pmatrix}$$

mit $l_{kk} \neq 0$, $k = 1, \ldots, N$. Mit Hilfe der ersten Zeile des Gleichungssystems ergibt sich direkt $u_1 = f_1/l_{11}$. Wir betrachten nun die k-te Zeile dieses Gleichungssystems

$$\sum_{m=1}^{k} l_{km} u_k = f_k \,.$$

Wenn die ersten $k-1$ Komponenten von u bereits berechnet wurden, ergibt sich für die k-te Komponente

$$u_k = \left(f_k - \sum_{m=1}^{k-1} l_{km} u_m \right) \Big/ \, l_{kk} \,.$$

Insgesamt können wir nach und nach die Komponenten von u berechnen, angefangen bei der ersten Komponente u_1 bis hin zur letzten Komponente u_N. Daher lautet der Name des Verfahrens auch Vorwärts-Substitution. Üblicherweise wird zur Berechnung von u die rechte Seite direkt überschrieben, da die f_k nur für den aktuellen Schritt benötigt werden. Anstatt die komplette Summe erst im aktuellen Schritt zu berechnen, kann diese auch in jedem Schritt für die späteren Komponenten berechnet werden.

Algorithmus 1.1 (Vorwärts-Substitution)**.**
Sei $L \in \mathbb{R}^{N\times N}$ eine reguläre untere Dreiecksmatrix und $f \in R^N$ die rechte Seite. Dann überschreibt der folgende Algorithmus f mit der Lösung des linearen Gleichungssystems $Lu = f$.

```
1  for k = 1, ..., N
2      f[k] := f[k]/L[k, k]
3      for m = k + 1, ..., N
4          f[m] := f[m] − L[m, k]f[k]
```

Dieser Algorithmus benötigt $O(N^2)$ flops.

1.2.2. Rückwärts-Substitution.

Ein analoger Algorithmus kann auch für eine obere Dreiecksmatrix angegeben werden. Sei dabei ein reguläres lineares oberes triangulares Gleichungssystem $Ru = f$,

$$\begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1N} \\ 0 & r_{22} & \cdots & r_{2N} \\ \vdots & \ddots & \ddots & \cdots \\ 0 & \cdots & 0 & r_{NN} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_N \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_N \end{pmatrix}$$

mit $r_{kk} \neq 0$, $k = 1, \ldots, N$. In diesem Fall ergibt die letzte Zeile des Gleichungssystems $u_N = f_N/r_{NN}$ und die k-te Zeile

$$\sum_{m=k}^{N} r_{km}u_k = f_k\,.$$

Wenn nun die $k+1$-te bis N-te Komponente von u schon berechnet wurden, kann die k-te Komponente berechnet werden mit

$$u_k = \left(f_k - \sum_{m=k+1}^{N} r_{km}u_k\right) \Big/ r_{kk}\,.$$

Damit können die Komponenten von u rückwärts berechnet werden, angefangen bei der letzten Komponente u_N bis hin zur ersten Komponente u_1. Daher auch der Name Rückwärts-Substitution. Auch hier kann die rechte Seite direkt überschrieben werden, womit sich folgender Algorithmus ergibt:

Algorithmus 1.2 (Rückwärts-Substitution)**.**
Sei $R \in \mathbb{R}^{N\times N}$ eine reguläre obere Dreiecksmatrix und $f \in \mathbb{R}^N$ die rechte Seite. Dann überschreibt der folgende Algorithmus f mit der Lösung des linearen Gleichungssystems $Ru = f$.

```
1  for k = N, ..., 1
2      f[k] := f[k]/R[k, k]
3      for m = k − 1, ..., 1
4          f[m] := f[m] − R[m, k]f[k]
```

Für die Rückwärts-Substitution wurde die Spalten-Variante gewählt (Algorithmus 3.1.4 in [**GL96**]). Das hat den Grund, da in der parallelen LR-Zerlegung in Kapitel 3 die Gesamtmatrix spaltenweise zerlegt wird und somit eine direkte Berechnung mit Hilfe der Spalten erfolgen kann.

Wie die Vorwärts-Substitution benötigt auch dieser Algorithmus $O(N^2)$ flops. Wenn $R[k, k] = 1$ (vergleiche Definition 1.3 im nächsten Kapitel 1.3), so kann der Schritt in Zeile 2 ignoriert werden.

1.3. Die LR-Zerlegung

Ein Gleichungssystem mit einer Dreiecksmatrix kann nach Kapitel 1.2.1 und 1.2.2 schnell und einfach gelöst werden. Wir betrachten nun eine Zerlegung einer Matrix $A \in \mathbb{R}^{N\times N}$ in zwei Dreiecksmatrizen $L \in \mathbb{R}^{N\times N}$ und $R \in \mathbb{R}^{N\times N}$ mit $A = LR$. Ein Gleichungssystem $Au = f$ kann nun in zwei Gleichungssysteme $Lv = f$, $Ru = v$ zerlegt werden, welche nacheinander mit Hilfe der Algorithmen 1.1 und 1.2 gelöst werden können. Wenn eine solche Zerlegung gefunden ist, kann der Aufwand zur Lösung dieses Gleichungssystems mit $O(N^2)$ angegeben werden.

Definition 1.3 (normierte Dreiecksmatrix)**.**
Eine obere normierte Dreiecksmatrix $R \in \mathbb{R}^{N\times N}$ ist definiert als eine obere Dreiecksmatrix mit Einsen auf der Diagonalen, d.h. $r_{kk} = 1$, $k = 1,\dots,N$, $r_{km} = 0$, $k > m$.
Eine obere normierte Block-Dreiecksmatrix ist definiert als eine obere Block-Dreiecksmatrix mit Einheitsmatrizen auf der Diagonalen, d.h. $R_{kk} = I_{N_k}$, $k = 1,\dots,N$, $R_{km} = 0$, $k > m$.

Eine untere normierte Dreiecksmatrix bzw. untere normierte Block- Dreiecksmatrix sind analog definiert.

Bemerkung 1.4.

a) Multiplikation zweier Dreiecksmatrizen gleicher Art ergibt wieder eine Dreiecksmatrix dieser Art.
b) Multiplikation zweier normierter Dreiecksmatrizen gleicher Art ergibt wieder eine normierte Dreiecksmatrix dieser Art.

Die LR-Zerlegung beruht auf dem Gaußschen Eliminationsverfahren. Es gibt verschiedene Varianten, eine LR-Zerlegung zu definieren. In dem hier vorgestellten Fall ist insbesondere diejenige Zerlegung interessant, bei der die Einheitsdiagonale in der R-Matrix zu finden ist, da dies später in der Block-LR-Zerlegung angewandt wird.

Definition 1.5 (LR-Zerlegung)**.**
Eine LR-Zerlegung einer Matrix $A \in \mathbb{R}^{N\times N}$ besteht aus einer unteren Dreiecksmatrix L und einer oberen normierten Dreiecksmatrix R, so dass $A = LR$ gilt.

Bemerkung 1.6.

a) In den meisten Beschreibungen einer LR-Zerlegung wird für gewöhnlich L als untere normierte Dreiecksmatrix gewählt, während hier R als obere normierte Dreiecksmatrix gewählt wird.
b) Selbst für eine reguläre Matrix A muss keine LR-Zerlegung existieren. Mit Hilfe von Pivotisierung kann jedoch immer eine PLR-Zerlegung angegeben werden, wobei P eine Permutationsmatrix beschreibt [**GL96**].
c) Für symmetrisch positiv definite Matrizen kann immer eine LR-Zerlegung angegeben werden (siehe auch Kapitel 3.9.1).

Eine LR-Zerlegung kann mit Hilfe von Gauß-Transformationen erreicht werden, wobei die Gauß-Transformation wie folgt definiert ist:

Definition 1.7 (Gauß-Transformation)**.**
Eine Gauß-Transformation zur Erstellung einer LR-Zerlegung im Sinne von 1.5 ist eine Matrix der Form $M_k = I - e_k\tau^T$ (mit e_k als k-tem Einheitsvektor), wobei die ersten k Komponenten von $\tau \in \mathbb{R}^N$ Null sind, das heißt

$$M_k = \begin{pmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & & 1 & -\tau_{k+1} & \cdots & -\tau_N \\ 0 & \cdots & 0 & 1 & & 0 \\ \vdots & & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & \cdots & 1 \end{pmatrix} .$$

Lemma 1.8. *Sei $\{M_k \colon M_k = I - e_k\tau^T,\ k = 1, \ldots, N-1\}$ eine geordnete Menge von Gauß-Transformationen.*

a) Für die Inverse einer Gauß-Transformation M_k gilt

$$M_k^{-1} = I + e_k\tau^T .$$

b) Für zwei Gauß-Transformationen M_k, M_l mit $k > l$ gilt

$$M_k M_l = I - e_k\tau_k^T - e_l\tau_l^T .$$

c) Sei mit $M_1 \cdots M_{N-1}$ ein Produkt von Gauß-Transformationen gegeben. Für die Inverse gilt

$$M_{N-1}^{-1} \cdots M_1^{-1} = (I + e_{N-1}\tau_{N-1}^T) \cdots (I + e_1\tau_1^T) = I + \sum_{k=1}^{N-1} e_k\tau_k^T .$$

Für weitere Erläuterungen siehe [GL96]. Die hier verwendeten Gauß-Transformationen sind transponiert im Vergleich zu Gauß-Transformationen aus der Literatur, da hier obere normierte Dreiecksmatrizen benutzt werden.

Anwendung einer Gauß-Transformation auf eine Matrix A von rechts ergibt

$$AM_k = A(I - e_k\tau^T) = A - (Ae_k)\tau^T = A - A[:,k]\tau^T ,$$

da hier die ersten k Einträge in τ gleich 0 sind, wird nur $A[:, k+1:N]$ geändert. Zur Elimination wird dabei in jedem Schritt $\tau[k+1:N] = A[k, k+1:N]/A[k,k]$ gesetzt. Mit Hintereinanderausführungen von Gauß-Transformationen kann damit eine Matrix A in eine untere Dreiecksmatrix transformiert werden.

Algorithmus 1.9 (Untere Triangulierung)**.**
Sei $A \in \mathbb{R}^{N\times N}$ eine gegebene reguläre Matrix. Dann erzeugt folgender Algorithmus eine untere Dreiecksmatrix.

```
1   for k = 1, ..., N-1
2       for m = k+1, ..., N
3           τ[m] := A[k,m]/A[k,k]
4           A[:,m] := A[:,m] - A[:,k]τ[m]
```

Hierbei wird angenommen, dass der betroffene Diagonaleintrag $A[k,k]$ in jedem Schritt k ungleich 0 ist[1].

[1] Dies ist für symmetrisch positiv definite Matrizen A immer erfüllt, vergleiche auch Kapitel 3.9.1

Im vorigen Algorithmus wird eine Hintereinanderausführung von Gauß-Transformationen M_k auf A angewandt, so dass

$$AM_1M_2\cdots M_{N-1} = L$$

eine untere Dreiecksmatrix ist. Mit

$$R = M_{N-1}^{-1}\cdots M_1^{-1} \tag{1.1}$$

gilt somit $A = LR$ und mit Bemerkung 1.4 und Lemma 1.8 folgt, dass R eine obere normierte Dreiecksmatrix ist.

Damit ist die LR-Zerlegung einer Matrix A beschrieben. Ein Vorteil ist, dass die Berechnung von R wegen Lemma 1.8.c) direkt gegeben ist und wir die benötigten Koeffizienten von τ_k direkt in der Matrix A speichern können, da durch die Transformation dort Nullen entstehen.

Somit kann die LR-Zerlegung einer Matrix A folgendermaßen beschrieben werden, wobei die Matrix A mit Koeffizienten aus L und R so überschrieben wird, dass $L[k,m] = A[k,m]$, $k \geq m$ und $R[k,m] = A[k,m]$, $k < m$.

Algorithmus 1.10 (LR-Zerlegung)**.**

```
1  for k = 1,...,N
2      for m = k+1,...,N
3          A[k,m] := A[k,m]/A[k,k]
4          for n = k+1,...,N
5              A[n,m] := A[n,m] - A[n,k]A[k,m]
```

Der Aufwand zur Berechnung der LR-Zerlegung ist in führender Ordnung $\frac{2}{3}N^3$ flops. Sei nun ein Gleichungssystem $Au = f$ gegeben, sowie eine LR-Zerlegung von A. Da $A = LR$, folgt mit $v = Ru$: $Au = f \Leftrightarrow LRu = f \Leftrightarrow Lv = f$. Um das Gleichungssystem $Au = f$ zu lösen, muss zuerst das Gleichungssystem $Lv = f$ mit der Vorwärts-Substitution 1.1 und danach das Gleichungssystem $Ru = v$ mit der Rückwärts-Substitution 1.2 gelöst werden. Da die Diagonaleinträge in R gleich 1 sind, ergibt sich insgesamt folgender Algorithmus:

Algorithmus 1.11 (Lösen mit Hilfe der LR-Zerlegung)**.**
Sei eine LR-Zerlegung der Matrix $A \in \mathbb{R}^{N\times N}$ durch Algorithmus 1.10, sowie eine rechte Seite $f \in \mathbb{R}^N$ gegeben. Dann überschreibt folgender Algorithmus f mit der Lösung u des Gleichungssystems $Au = f$.

```
1  for k = 1,...,N
2      f[k] := f[k]/A[k,k]
3      for m = k+1,...,N
4          f[m] := f[m] - A[m,k]f[k]
5  for k = N,...,2
6      for m = k-1,...,1
7          f[m] := f[m] - A[m,k]f[k]
```

1.4. Fehleranalyse

Für die Fehleranalyse folgen wir der Notation aus [Hig96]. Es gibt verschiedene Gründe für aufkommende Fehler in einer numerischen Rechnung: Rundungsfehler (da nur ein Teil der reellen Zahlen abgebildet werden kann), Datenunsicherheit (zum Beispiel in der rechten Seite f oder Anfangsbedingungen) und Diskretisierungsfehler (Wahl des Gitters bei den Finiten Elementen). Während die beiden letzteren insbesondere in der Finite Elemente-Theorie Beachtung finden, treten bei der Durchführung der Zerlegung zur Lösung der entstehenden linearen Gleichung in erster Linie Rundungsfehler auf.

1.4.1. Gleitkomma-System.

Ein Gleitkomma-System $F \subset \mathbb{R}$ ist eine Untermenge der reellen Zahlen, wobei die Elemente die Form

$$y = \pm m \cdot \beta^e$$

mit $m \in \mathbb{N}_0$, $\beta \in \mathbb{N}$ und $e \in \mathbb{Z}$ besitzen. Dieses System wird durch folgende Parameter bestimmt:

- die *Basis* β,
- die *Mantisse* m mit *Wortlänge* p, wobei m repräsentiert wird durch eine Zahlenkette $d_0 d_1 d_2 \dots d_{p-1}$ und $0 \leq d_i < \beta$, das heißt m ist eine ganze Zahl mit $0 \leq m \leq \beta^p - 1$,
- dem *Exponenten* e mit $e_{\min} \leq e \leq e_{\max}$.

Im IEEE-Standard [IEE08] gilt für das double-precision-Format

- $\beta = 2$,
- $p = 52$,
- $e_{\max} = 1023$, $e_{\min} = -1022$.

Hier wird weiterhin die Mantisse normalisiert. Dabei wird für die Gleitkommazahl $y = \pm m_n \cdot 2^e$ die normalisierte Mantisse $m_n = 1 + m/2^p$ verwendet. Einfacher ausgedrückt wird vor das Mantissenbitmuster m eine "1." angehängt. Für die normalisierte Mantisse gilt somit $1 \leq m_n < 2$. Die 1 in $m_n = 1.m$ muss nicht abgespeichert werden, so dass ein weiteres Bit Genauigkeit gewonnen werden kann[2]. In der IEEE-Konvention gilt weiterhin $e_{\min} = 1 - e_{\max}$. Der Exponent e wird mit Hilfe von 11 Bits abgespeichert. Damit lassen sich die Zahlen $E = 0$ bis $E = 2047$ darstellen. e wird mit Hilfe eines festen Biaswertes $B = 1023$ berechnet über $e = E - B$. Die Bitfolge $E_{\text{bit}} = 11 \dots 1$ und $E_{\text{bit}} = 00 \dots 0$ werden für Sonderfälle (not a number bzw. ∞ und Null) benötigt und werden somit nicht für die Darstellung eines Exponenten verwendet.

Die Gleitkomma-Zahlen sind nicht gleichmäßig verteilt, es kann jedoch eine relative Genauigkeit angegeben werden. Diese wird als Maschinengenauigkeit $\mathfrak{u}$ bezeichnet. Eine reelle Zahl ist also für gewöhnlich nicht exakt darstellbar, so dass auf die nächste darstellbare Zahl gerundet werden muss. Mit fl bezeichnen wir die Rundung einer gegebenen Zahl auf Maschinenebene. Somit gilt nach [Hig96]

Lemma 1.12. *Sei $x \in \mathbb{R}$ innerhalb des Bereiches von F, dann gilt*

$$fl(x) = x(1 + \delta), \quad |\delta| < \mathfrak{u}\,.$$

Auch bei der Durchführung der arithmetischen Operationen muss zu jedem Zeitpunkt gerundet werden:

$$fl(x \text{ op } y) = (x \text{ op } y)(1 + \delta), \quad |\delta| \leq \mathfrak{u}\,.$$

[2]Daher wird für die Mantissenlänge oft auch $p = 53$ angegeben.

Dabei bezeichnet die Operation op $= +, -, \cdot, /$ eine der möglichen arithmetischen Operatoren. Im Fall der double-precision beträgt $\mathfrak{u}$ nach IEEE-Standard $\mathfrak{u} = 2^{-53} \approx 10^{-16}$.

1.4.2. Fehleranalyse der Triangular-Systeme und der LR-Zerlegung.

Im Folgenden wird eine Matrix mit einem Dach (z.B. $\hat{L}$) als eine Matrix bezeichnet, die im Gleitkomma-System berechnet wurde, das heißt jeder Berechnungsschritt muss auf eine Gleitkomma-Zahl gerundet werden. Im Allgemeinen sind die Eingangsdaten schon auf das Gleitkomma-System bezogen und erhalten kein zusätzliches Symbol. Mit ΔA wird die Differenz einer Berechnung zum ursprünglichen System bezeichnet.

Zuerst geben wir ein Lemma aus [**Hig96**, Theorem 8.5] an, welches zeigt, wie sich die Berechnung einer Lösung $\hat{x}$ über ein Triangularsystem aufgrund des Gleitkomma-Systems auswirkt.

Lemma 1.13. *Gegeben sei ein triangulares System $Tx = b$, mit $T \in \mathbb{R}^{N\times N}$ reguläre (obere oder untere) Dreiecksmatrix, welches über die Vorwärts- (Algorithmus 1.1) bzw. Rückwärts-Substitution (Algorithmus 1.2) gelöst wird. Dann gilt für die berechnete Lösung $\hat{x}$*

$$(T + \Delta T)\hat{x} = b, \quad |\Delta T| \leq \frac{N\mathfrak{u}}{1 - N\mathfrak{u}}|T| + O(\mathfrak{u}^2)$$

Hierbei ist der Betrag $|\cdot|$ einer Matrix komponentenweise zu verstehen. Das heißt, berechnet wird die Lösung eines schwach gestörten Systems, wobei die Störungsmatrix ΔT komponentenweise gegenüber der ursprünglichen Matrix T klein ist. Für ein System der Größe $N = 10^7$ ist der relative Fehler in den einzelnen Komponenten also kleiner als etwa 10^{-9}, wenn man von double-precision ausgeht.

Satz 1.14. *Sei $A \in \mathbb{R}^{N\times N}$ eine Matrix, dessen LR-Zerlegung existiert. Die berechneten Matrizen $\hat{L}$ und $\hat{R}$ genügen der Darstellung*

$$\hat{L}\hat{R} = A + \Delta A \tag{1.2}$$

$$|\Delta A| \leq 3(N-1)\mathfrak{u}\left(|A| + |\hat{L}||\hat{R}|\right) + O(\mathfrak{u}^2) \tag{1.3}$$

Beweis. Wir folgen dem Beweis nach [**GL96**, Theorem 3.3.1] über vollständige Induktion[3].

Der Satz gilt offensichtlich für $N = 1$ (da dort keine Rechenoperationen auftreten und nur $l_{11} = a_{11}, r_{11} = 1$ gesetzt wird). Wir nehmen an, dass er für alle $(N-1)\times(N-1)$-Matrizen gilt. Mit

$$A = \begin{pmatrix} \alpha & w^t \\ v & B \end{pmatrix},$$

$\alpha \in \mathbb{R}$, $B \in \mathbb{R}^{(N-1)\times(N-1)}$ folgt für den ersten Schritt der LR-Zerlegung

$$\tilde{A} = \begin{pmatrix} \alpha & \hat{z}^T \\ v & \hat{A}_1 \end{pmatrix},$$

wobei $\hat{z} = fl(w/\alpha)$ und $\hat{A}_1 = fl(B - v\hat{z}^T)$ berechnet werden. Dabei gilt:

$$\hat{z} = \frac{1}{\alpha}w + f, \quad |f| \leq \mathfrak{u}\frac{|w|}{|\alpha|}$$

und

$$\hat{A}_1 = B - v\hat{z}^T + F, \quad |F| \leq 2\mathfrak{u}(|B| + |v||\hat{z}|^T) + O(\mathfrak{u}^2)\,, \tag{1.4}$$

[3]Dort wird eine andere LR-Zerlegung verwendet. Während hier R normiert ist, wird dort mit einer normierten Dreiecksmatrix L gerechnet.

wobei die 2 in der Abschätzung für $|F|$ aus den beiden Operationen Multiplikation und Addition hervorgeht.

Da $\hat{A}_1 \in \mathbb{R}^{(N-1)\times(N-1)}$ können wir dafür nun die Induktionsvoraussetzung (1.2), (1.3) verwenden mit

$$\hat{L}_1\hat{R}_1 = \hat{A}_1 + \Delta A_1 \tag{1.5}$$

$$|\Delta A_1| \leq 3(N-2)\mathfrak{u}\left(|\hat{A}_1| + |\hat{L}_1||\hat{R}_1|\right) + O(\mathfrak{u}^2) \tag{1.6}$$

Damit gilt

$$\begin{aligned}\hat{L}\hat{R} &\equiv \begin{pmatrix}\alpha & 0\\ v & \hat{L}_1\end{pmatrix}\begin{pmatrix}1 & \hat{z}^t\\ 0 & \hat{R}_1\end{pmatrix}\\ &= A + \begin{pmatrix}0 & \alpha f\\ 0 & \Delta A_1 + F\end{pmatrix} \equiv A + \Delta A\,.\end{aligned}$$

Aus (1.4) folgt

$$|\hat{A}_1| \leq (1+2\mathfrak{u})(|B| + |v||\hat{z}|^T) + O(\mathfrak{u}^2) \tag{1.7}$$

und mit (1.5) und (1.6) gilt

$$|\Delta A_1 + F| \leq 3(N-1)\mathfrak{u}\left(|B| + |v||\hat{z}|^T + |\hat{L}_1||\hat{R}_1|\right) + O(\mathfrak{u}^2)\,.$$

Mit $|\alpha f| \leq \mathfrak{u}|w|$ kann man durch Ausmultiplizieren überprüfen, dass gilt:

$$|\Delta A| \leq 3(N-1)\mathfrak{u}\left[\begin{pmatrix}|\alpha| & |w|^T\\ |v| & |B|\end{pmatrix} + \begin{pmatrix}|\alpha| & 0\\ |v| & |\hat{L}_1|\end{pmatrix}\begin{pmatrix}1 & |\hat{z}|^T\\ 0 & |\hat{R}_1|\end{pmatrix}\right] + O(\mathfrak{u}^2)\,,$$

was genau der Induktionsbehauptung entspricht. □

2

Parallele Finite Elemente

Eine Einführung in Finite Elemente findet sich in [Bra03]. Für die Realisierung des parallelen direkten Lösers wird das parallele Finite Elemente Programm `M++` verwendet. Das im Programm verwendete Programmiermodell geht auf Arbeiten der UG-Gruppe [BBJ+97, BBJ+98, BJL+99, BJL+00, BJL+01] zurück. Um eine einheitliche Darstellung zu gewährleisten, werden in diesem Abschnitt die Definitionen aus [Wie10] übernommen und gegebenenfalls erweitert, vor allem mit einigen Ausführungen aus eigenen Veröffentlichungen [MW11, MW12]. Dabei wird insbesondere der parallelen additiven Repräsentation der Steifigkeitsmatrix Beachtung geschenkt, die Hauptbestandteil der parallelen Block-LR-Zerlegung ist. Aufgrund der Notation ist es übersichtlicher, im Folgenden anstatt von "Kernen" von "Prozessoren" zu reden. Wenn etwas "auf einem Prozessor" geschieht, so ist dies dabei ein Prozess in einer MPI-Routine.

2.1. Ein paralleles Gitter-Modell

Gegeben sei ein Gitter $(\mathcal{C}, \mathcal{V}, \mathcal{E}, \mathcal{F})$ mit Zellen $\mathcal{C}$, Ecken $\mathcal{V}$, Kanten $\mathcal{E}$ und Seitenflächen $\mathcal{F}$. Für eine Zelle $c \in \mathcal{C}$ werden mit $\mathcal{V}_c$, $\mathcal{E}_c$, $\mathcal{F}_c$ die jeweiligen Ecken, Kanten und Seitenflächen von c bezeichnet. Insgesamt wird durch das Set der Zellen $\mathcal{C}$ die Ecken $\mathcal{V} = \bigcup_{c\in\mathcal{C}} \mathcal{V}_c$, Kanten $\mathcal{E} = \bigcup_{c\in\mathcal{C}} \mathcal{E}_c$ und Seitenflächen $\mathcal{F} = \bigcup_{c\in\mathcal{C}} \mathcal{F}_c$ definiert. Jedes Element (Zelle, Ecke, Kante oder Seitenfläche) wird durch einen Punkt $z \in \mathbb{R}^3$ eindeutig definiert, wobei dieser Punkt als Schlüssel in einer Hashtabelle dient, worin diese Elemente gespeichert werden. Jede Kante $e \in \mathcal{E}_c$ wird über ein Paar $(x_e, y_e) \in \mathcal{V}_c \times \mathcal{V}_c$ und seinen Mittelpunkt $z_e \in \mathbb{R}^3$ beschrieben. Der Mittelpunkt der Kante ist dabei der zugehörige Schlüssel in der Hashtabelle. Genauso dienen die Zellmittelpunkte z_c einer Zelle $c \in \mathcal{C}$ und die Seitenflächenmittelpunkte z_f einer Seitenfläche $f \in \mathcal{F}_c$ als Schlüssel. Insgesamt ergibt sich für eine Zelle $c \in \mathcal{C}$ ein Schlüssel-Set

$$\mathcal{Z}_c = \mathcal{V}_c \cup \{z_e \colon e \in \mathcal{E}_c\} \cup \{z_f \colon f \in \mathcal{F}_c\} \cup \{z_c\}\,.$$

Für eine gegebene Prozessormenge $\mathcal{P} = \{1, \dots, P\}$ wird über eine Abbildung $\mathrm{dest} \colon \mathcal{C} \to \mathcal{P}$ eine Lastverteilung der Zellen $\mathcal{C}$ definiert. Mit

$$\mathcal{C}^p = \{c \in \mathcal{C} \colon \mathrm{dest}(c) = p\}$$

wird dabei die Menge der Zellen auf Prozessor p beschrieben, wobei wir insgesamt eine disjunkte Zerlegung

$$\mathcal{C} = \mathcal{C}^1 \,\dot\cup\, \cdots \,\dot\cup\, \mathcal{C}^P$$

haben. Ausgehend von dieser Zerlegung werden weiterhin die Mengen

$$\mathcal{V}^p = \bigcup_{c\in C^p} \mathcal{V}_c\,, \qquad \mathcal{E}^p = \bigcup_{c\in C^p} \mathcal{E}_c\,, \qquad \mathcal{F}^p = \bigcup_{c\in C^p} \mathcal{F}_c\,, \qquad \mathcal{Z}^p = \bigcup_{c\in C^p} \mathcal{Z}_c$$

beschrieben, wobei die Zerlegungen

$$\mathcal{V} = \mathcal{V}^1 \cup \cdots \cup \mathcal{V}^P\,, \qquad \mathcal{E} = \mathcal{E}^1 \cup \cdots \cup \mathcal{E}^P\,,$$
$$\mathcal{F} = \mathcal{F}^1 \cup \cdots \cup \mathcal{F}^P\,, \qquad \mathcal{Z} = \mathcal{Z}^1 \cup \cdots \cup \mathcal{Z}^P$$

im Allgemeinen nicht disjunkt sind. Für die Elemente der Hashschlüssel $\mathcal{Z}$ kann damit eine mengenwertige Abbildung

$$\pi\colon \mathcal{Z} \to 2^{\mathcal{P}}\,, \qquad \pi(z) = \{p \in \mathcal{P}\colon z \in \mathcal{Z}^p\} \tag{2.1}$$

definiert werden, die jedem Hashschlüssel eine Prozessormenge zuordnet.

2.2. Parallele Finite Elemente

2.2.1. Notation.

In diesem Kapitel wird eine lokale Notation eingeführt. So werden bis inklusive Kapitel 2.2.5 Vektoren ($\underline{v}, \underline{f}$) und Matrizen ($\underline{A}$), sowie deren zugehörigen Räume ($\underline{V}, \underline{V}'$) unterstrichen. Allgemeinere Räume und Operatoren werden "normal" (z.B. Finite Elemente Raum V) oder "fett" (z.B. Hilbertraum $\mathbf{V}$) dargestellt. Kalligraphische Buchstaben ($\mathcal{I}, \mathcal{P}, \dots$) bezeichnen wie bisher Mengen aus dem parallelen Gitter-Modell.

Aufgrund der Übersichtlichkeit wechseln wir ab Kapitel 2.4 für Matrizen wieder auf die "gewohnte" Darstellung (A), wie sie in Kapitel 1 üblich war.

2.2.2. Parallele Finite Elemente.

Ein Finite Elemente Gitter ist durch eine Zerlegung

$$\bar{\Omega} = \bigcup_{c\in\mathcal{C}} \bar{\Omega}_c$$

gegeben, wobei $\Omega_c \subset \mathbb{R}^3$ das Innere der Zelle c ist, mit $\bar{\Omega}_c = \operatorname{conv}\{\mathcal{V}_c\}$[1]. Die disjunkte Zerlegung der Zellen ergibt eine nichtüberlappende Gebietszerlegung

$$\Omega^1 \cup \cdots \cup \Omega^P, \qquad \Omega^p = \operatorname{int} \bigcup_{c\in\mathcal{C}^p} \bar{\Omega}_c \tag{2.2}$$

von Ω mit dem Interface

$$\Gamma = \bigcup_{p,q\in\mathcal{P}} \Gamma^{pq}, \qquad \Gamma^{pq} = \partial\Omega^p \cap \partial\Omega^q\,.$$

Sei $V_{\mathcal{I}} = \operatorname{span}\{\phi_i : i \in \mathcal{I}\}$ ein Finite Elemente Raum mit Basis ϕ_i, wobei mit $\mathcal{I}$ die zugehörige Indexmenge bezeichnet wird. Jeder Basisfunktion ϕ_i wird eine duale Basisfunktion $\phi' \in V_{\mathcal{I}}'$ zugeordnet, so dass $\langle \phi_i', \phi_j \rangle = \delta_{ij}$ gilt. Zu jedem Index $i \in \mathcal{I}$ existiert ein Knotenpunkt $z_i \in \bar{\Omega}$ und im Falle von mehrdimensionalen Systemen eine Komponente k_i. Aufgrund der Übersichtlichkeit setzen wir $k_i = 1$ und erlauben "mehrfache" Knoten, die über den Index k_i intern sortiert sind. Somit ergibt sich mit $\mathcal{N} = \{z_i\colon i \in \mathcal{I}\}$ die Menge der Knotenpunkte. In unserem Fall gelte $\mathcal{N} \subset \mathcal{Z}$, so dass die Abbildung π aus (2.1) auch direkt auf $\mathcal{I}$ definiert werden kann: Wir setzen $\pi(i) = \pi(z_i)$ und definieren mit

$$\mathcal{I}^p = \{i \in \mathcal{I}\colon z_i \in \overline{\Omega}^p\} = \{i \in \mathcal{I}\colon p \in \pi(i)\} \tag{2.3}$$

die lokalen Indizes auf Prozessor p.

[1] Mit $\operatorname{conv}\{\mathcal{V}_c\}$ wird die konvexe Hülle über alle Eckpunkte der Zelle c bezeichnet.

Somit ist über

$$\mathcal{I} = \mathcal{I}^1 \cup \cdots \cup \mathcal{I}^P$$

eine (überlappende) Zerlegung der Indizes gegeben. Weiterhin bezeichnen wir mit $N^p = \#\mathcal{I}^p$ die Anzahl der Indizes auf Prozessor p und mit $N = \#\mathcal{I} = \#\mathcal{N}$ die Anzahl der Unbekannten. Insgesamt wird hier keine globale Nummerierung der Indizes benötigt, da die Indizes über den Knotenpunkt z_i (beziehungsweise über das Paar (z_i, k_i)) eindeutig identifiziert sind. Somit kann eine *Interfacemenge*

$$\mathcal{I}_\Gamma = \{i \in \mathcal{I} \colon \#\pi(i) > 1\}$$

und die Menge der *inneren Indizes auf Prozessor p*

$$\mathcal{I}_\Omega^p = \{i \in \mathcal{I} \colon \pi(i) = p\}$$

definiert werden, wobei die Gesamtmenge der inneren Indizes mit

$$\mathcal{I}_\Omega = \bigcup_{p \in \mathcal{P}} \mathcal{I}_\Omega^p = \{i \in \mathcal{I} \colon \#\pi(i) = 1\}$$

gegeben ist.

Eine Finite Elemente Funktion $v = \sum_{i \in \mathcal{I}} v_i \phi_i \in V_\mathcal{I}$ wird eindeutig durch den Koeffizientenvektor $(v_i)_{i \in \mathcal{I}}$ mit $v_i = \langle \phi_i', v \rangle$ repräsentiert. Analog ist ein diskretes Funktional $f = \sum_{i \in \mathcal{I}} f_i \phi_i' \in V_\mathcal{I}'$ durch den Koeffizientenvektor $(f_i)_{i \in \mathcal{I}}$ mit $f_i = \langle f, \phi_i \rangle$ gegeben. Der Hauptgedanke der parallelen Finiten Elemente besteht in der unterschiedlichen parallelen Repräsentation von Finite Elemente Funktionen und diskreter Funktionale. Dazu werden zwei verschiedene Vektordarstellungen eingeführt: die konsistente Darstellung für die Finite Elemente Funktionen und die additive Darstellung für die diskreten Funktionale (und auch der Systemmatrizen).

2.2.3. Konsistente Darstellung von Vektoren.

Der Koeffizientenvektor $(v[i])_{i \in \mathcal{I}}$ einer Finite Elemente Funktion $v \in V_\mathcal{I}$ wird im parallelen Kontext durch seine lokalen Einschränkungen $\underline{v}^p = (v[i])_{i \in \mathcal{I}^p} \in \mathbb{R}^{N^p}$ repräsentiert, wodurch eine Abbildung

$$\begin{aligned} E \colon V_\mathcal{I} \to \underline{V}_\mathcal{I}^\mathcal{P} &:= \prod_{p \in \mathcal{P}} \mathbb{R}^{N^p} \\ v &\mapsto \underline{v} = (\underline{v}^p)_{p \in \mathcal{P}} \end{aligned} \tag{2.4}$$

definiert ist. Eine konsistente Darstellung eines Koeffizientenvektors $(v_i)_{i \in \mathcal{I}}$ ist gegeben, wenn alle Koeffizienten auf den Interfaces (das heißt für $i \in \mathcal{I}_\Gamma$) auf jedem Prozessor übereinstimmen. Wir definieren daher den Raum

$$\underline{V}_\mathcal{I} = \{\underline{v} \in \underline{V}_\mathcal{I}^\mathcal{P} \colon v^p[i] = v^q[j] \text{ für } q \in \pi^p(i)\}$$

wobei $\underline{V}_\mathcal{I} = E(V_\mathcal{I})$ gilt. Der Koeffizientenvektor einer Finiten Elemente Funktion ist konsistent.

2.2.4. Additive Darstellung von Vektoren.

Im Parallelen werden die diskreten Funktionale f gewöhnlich auf jedem Prozessor einzeln auf dem ihnen zugewiesenen Gebiet Ω^p aufgebaut (dies gilt auch für die Systemmatrix, siehe 2.2.5). Die Summe (für jeden Index betrachtet) der lokalen Darstellung $\underline{f}^p = (f^p[i])_{i \in \mathcal{I}^p} \in \mathbb{R}^{N^p}$ ergibt den vollen Koeffizientenvektor $(f[i])_{i \in \mathcal{I}}$ mit

$$f[i] = \sum_{p \in \pi(i)} f^p[i]\,. \tag{2.5}$$

Für die Darstellung von Funktionalen aus V' definieren wir den Quotientenraum

$$\underline{V}'_{\mathcal{I}} = \underline{V}^{\mathcal{P}}_{\mathcal{I}} / \underline{V}^{0}_{\mathcal{I}}$$

mit

$$\underline{V}^{0}_{\mathcal{I}} = \{\underline{f} \in \underline{V}^{\mathcal{P}}_{\mathcal{I}} \colon \sum_{p \in \pi(i)} f^p[i] = 0,\ i \in \mathcal{I}\}\,.$$

Aufgrund der konsistenten Darstellung von $\underline{v} \in \underline{V}_{\mathcal{I}}$ und der additiven Darstellung von $\underline{f} \in \underline{V}'_{\mathcal{I}}$ ist die duale Paarung

$$\underline{v} \cdot \underline{f} = \sum_{p \in \mathcal{P}} \underline{v}^p \cdot \underline{f}^p, \qquad \underline{v} \in \underline{V}_{\mathcal{I}}, \underline{f} \in \underline{V}'_{\mathcal{I}}$$

wohldefiniert auf $\underline{V}_{\mathcal{I}} \times \underline{V}'_{\mathcal{I}}$ und der adjungierte Operator bezüglich der Einbettung E aus (2.4) ergibt sich zu

$$\begin{aligned} E' \colon \underline{V}'_{\mathcal{I}} &\to V'_{\mathcal{I}} \\ \underline{f} &\mapsto f = \sum_{p \in \mathcal{P}} \sum_{i \in \mathcal{I}^p} f^p[i] \phi'_i\,. \end{aligned} \tag{2.6}$$

Die Abbildung (2.6) ist nicht injektiv. Um einen eindeutigen parallelen Repräsentanten $\underline{f} \in \underline{V}'_{\mathcal{I}}$ eines Funktionals $f \in V'_{\mathcal{I}}$ zu erhalten, setzen wir $p = \min \pi(i)$ als Masterprozessor eines Index i und definieren die Menge

$$\widetilde{\mathcal{I}}^p = \{i \in \mathcal{I}^p \colon p = \min \pi(i)\}\,,$$

wodurch sich eine nichtüberlappende Zerlegung der Indexmenge

$$\mathcal{I} = \widetilde{\mathcal{I}}^1 \cup \cdots \cup \widetilde{\mathcal{I}}^P \tag{2.7}$$

ergibt. Mit

$$\begin{aligned} \widetilde{\underline{V}}_{\mathcal{I}} &= \{\underline{f} = (\underline{f}^p)_{p \in \mathcal{P}} \in \underline{V}'_{\mathcal{I}} \colon f^p[i] = 0 \text{ für } p \neq \min \pi(i)\} \\ &= \{\underline{f} = (\underline{f}^p)_{p \in \mathcal{P}} \in \underline{V}'_{\mathcal{I}} \colon f^p[i] = 0 \text{ für } i \notin \widetilde{\mathcal{I}}^p\} \end{aligned}$$

ist die Einschränkung von E' auf $\widetilde{\underline{V}}_{\mathcal{I}}$ injektiv.

2.2.5. Additive Darstellung von Operatoren.

Sei $\mathbf{V}$ ein Hilbertraum und $a(\cdot,\cdot) \colon \mathbf{V} \times \mathbf{V} \to \mathbb{R}$ eine elliptische Bilinearform. Wir definieren den Operator $\mathbf{A} \colon \mathbf{V} \to \mathbf{V}'$ durch $\langle \mathbf{A}\mathbf{v}, \mathbf{w} \rangle = a(\mathbf{v}, \mathbf{w})$ für $\mathbf{v}, \mathbf{w} \in \mathbf{V}$. Für einen Finite Elemente Raum $V_{\mathcal{I}} \subset \mathbf{V}$ bezeichnen wir die Galerkin-Approximation mit $A \colon V_{\mathcal{I}} \to V'_{\mathcal{I}}$, definiert durch

$$\langle Av, w \rangle = a(v, w), \qquad v, w \in V_{\mathcal{I}}\,.$$

Im Zusammenhang mit Finite Elementen ist die Aufspaltung der Bilinearform über die Zellen additiv gegeben, so dass

$$a(\mathbf{v}, \mathbf{w}) = \sum_{c \in \mathcal{C}} a_c(\mathbf{v}, \mathbf{w}), \qquad \mathbf{v}, \mathbf{w} \in \mathbf{V}\,.$$

2.2.5.1. *Gebräuchliche konforme Finite Elemente.*
Im Fall gebräuchlicher konformer Finite Elemente[2] sind die Knotenpunkte z_i der zugehörigen Ansatzfunktionen ϕ_i entweder auf dem Rand einer Zelle c gegeben oder der Support ist auf die Zelle c beschränkt, das heißt für eine Zelle c gilt:

$$\text{supp } \phi_i \cap \Omega_c \neq \emptyset \Leftrightarrow z_i \in \bar{\Omega}_c\,. \tag{2.8}$$

[2] insbesondere Lagrange-Elemente, Taylor-Hood Serendipity, Nédélec-Elemente

Somit kann $a_c(\phi_i, \phi_j) \neq 0$ nur dann auftreten, wenn $c \in \mathcal{C}_i \cap \mathcal{C}_j$. Daher können wir lokale Steifigkeitsmatrizen $\underline{A}^p = (A^p[i,j])_{i,j\in\mathcal{I}^p}$ einführen mit

$$A^p[i,j] = \sum_{c\in\mathcal{C}^p} a_c(\phi_i, \phi_j)\,. \tag{2.9}$$

Die Matrixdarstellung $\underline{A}\colon \underline{V}_\mathcal{I} \to \underline{V}'_\mathcal{I}$ des Operators A mit $A = E' \circ \underline{A} \circ E$ ergibt sich somit zu $\underline{A} = (\underline{A}^p)_{p\in\mathcal{P}}$, das heißt, insgesamt erhalten wir

$$A\phi_i = \sum_{p\in\mathcal{P}} \sum_{j\in\mathcal{I}^p} A^p[i,j]\phi'_j\,. \tag{2.10}$$

2.2.5.2. *Discontinuous-Galerkin-Ansatz.*
Im Fall des Discontinuous-Galerkin-Ansatz (vergleiche Kapitel 6.6) sind die Basisfunktionen ϕ_i den Zellmittelpunkten z_c zugeordnet, das heißt für die Menge der Knotenpunkte gilt

$$\mathcal{N} = \{z_c\colon c \in \mathcal{C}\}\,.$$

In diesem Fall können wir somit eine Zelle c mit einer zugehörigen Basisfunktion ϕ_c – oder umgekehrt eine Basisfunktion ϕ_i mit einer Zelle c_i – charakterisieren. Ein Zellmittelpunkt befindet sich durch die disjunkte Zerlegung immer nur auf einem Prozessor, die Basisfunktionen besitzen allerdings einen Support in eine benachbarte Zelle, die sich auf einem anderen Prozessor befinden kann. Diejenigen Zellen auf Prozessor p, die eine benachbarte Zelle auf einem anderen Prozessor q besitzen, bezeichnen wir als eine Interface-Zelle.

Allgemein gilt hier für einen Matrixeintrag $a_c(\phi_i, \phi_j) \neq 0$ nur dann, wenn zur Zelle c eine weitere Zelle c' existiert, so dass $z_i \in \bar{\Omega}_c$, $z_j \in \bar{\Omega}_{c'}$ (oder umgekehrt) und die Zellen c, c' einen gemeinsamen Rand besitzen, das heißt

$$\begin{aligned} a_c(\phi_i, \phi_j) \neq 0 \Leftrightarrow \quad & (z_i \in \bar{\Omega}_c \wedge \exists c' \in \mathcal{C}, f \in \mathcal{F}_c\colon f \in \mathcal{F}_{c'}, z_j \in \bar{\Omega}_{c'}) \\ & \vee (z_j \in \bar{\Omega}_c \wedge \exists c' \in \mathcal{C}, f \in \mathcal{F}_c\colon f \in \mathcal{F}_{c'}, z_i \in \bar{\Omega}_{c'})\,. \end{aligned} \tag{2.11}$$

Für den späteren Algorithmus der parallelen Block-LR-Zerlegung müssen für diesen Fall die Prozessormengen $\pi(i)$ abgeändert werden zu

$$\pi(i) = \pi(z_i) = \sum_{f\in\mathcal{F}_{c_i}} \pi(z_f) \tag{2.12}$$

(vergleiche auch Kapitel 7.4.2). Somit erhalten alle Zellmittelpunkte die Information der Zugehörigkeit aller benachbarten Zellen. Für eine Interface-Zellen c gilt nun $|\pi(z_c)| > 1$, so dass diese Elemente nun zur Interfacemenge $\mathcal{I}_\Gamma$ gehören.

2.3. Nested Dissection

Die parallele LR-Zerlegung basiert in erster Linie darauf, wie das Gebiet Ω auf die Prozessoren verteilt wird. Bereits in den 70er-Jahren wurde von Alan George eine Methode vorgeschlagen, die "Nested Dissection" (geschachtelte Zerlegung) genannt wird [**Geo73**]. Aufgrund des lokalen Trägers der Basisfunktionen (vergleiche (2.8)) ist es möglich, das Gebiet für die Finiten Elemente so zu zerlegen, dass jeweils zwei Teilgebiete nur durch einen "Separator" miteinander verbunden sind, so dass die beiden Teilgebiete komplett voneinander getrennt betrachtet werden können. Dieser Separator ist eine Dimension kleiner als das Ursprungsgebiet (das heißt, wenn wir ein 3-dimensionales Gebiet betrachten, so ist der Separator - also, die Schnittfläche - 2-dimensional). Über einen "Schnitt" können wir uns die Teilung auch vorstellen: Das Gebiet wird in zwei Teilgebiete zerschnitten, wobei der Schnitt selbst der Separator ist und die entstehenden Teilgebiete voneinander unabhängig betrachtet werden können. Wenn wir das rekursiv fortsetzen, so können

wir jedes Teilgebiet erneut zerschneiden, so dass wir zum Schluss viele Separatoren (die jeweils eine Dimension kleiner als das Ursprungsgebiet sind) erhalten, die viele einzelne Teilgebiete miteinander verbinden – diese sind allerdings voneinander unabhängig und können somit parallel betrachtet werden. Damit wird jedes Teilgebiet einem Prozessor übergeben. Mit Hilfe der Separatoren können die Teilgebiete wieder zusammengefügt werden, so dass nach und nach das Gesamtgebiet wieder erlangt werden kann.

Letztendlich beschreiben die Separatoren nach der ersten Zerlegung ein Schur-Komplement, welches nach und nach parallel gelöst werden kann. Da für jeden "Schnitt" auch die Separatoren zerschnitten werden müssen, wird im nächsten Kapitel von "Interfaces" gesprochen, wobei der erste Separator (aus dem ersten Schnitt) zum Schluss aus mehreren Interfaces besteht.

2.4. Parallele Matrix- und Vektorblockstruktur

Wie in Kapitel 2.2.1 erwähnt, wird für Matrizen aufgrund der Übersichtlichkeit ab sofort wieder die "Normalschreibweise" verwendet. Des Weiteren wird auf die parallele Darstellung der Gesamtsteifigkeitsmatrix $A = (A^p)_{p\in\mathcal{P}}$ verzichtet und die globale Matrix A so definiert, als ob sie nur auf einem Prozessor erstellt werden würde. Mit (2.10) gilt, dass jeder Matrixeintrag von A als Summe von lokalen Matrixeinträgen von A^p angesehen werden kann (vergleiche auch Bemerkung 2.6).

2.4.1. Matrixblockstruktur über Interfaces.

Mit Hilfe einer weiteren Zerlegung der Indexmenge $\mathcal{I}$ führen wir eine Blockstruktur ein. Dazu sei $\Pi = 2^{\mathcal{P}}$ die Menge aller möglichen Prozessormengen. Mit

$$\Pi_{\mathcal{I}} = \{\pi(i) \in \Pi \colon i \in \mathcal{I}\}$$

bezeichnen wir die Menge der aktiven Prozessormengen in Bezug zum entsprechenden Matrixgraphen. Eine geeignete Aufzählung dieser Menge

$$\Pi_{\mathcal{I}} = \{\pi_1, \pi_2, \ldots, \pi_K\} \tag{2.13}$$

führt direkt zu einer nichtüberlappenden Zerlegung

$$\mathcal{I} = \mathcal{I}_1 \cup \cdots \cup \mathcal{I}_K \quad \text{mit} \quad \mathcal{I}_k = \{i \in \mathcal{I} \colon \pi(i) = \pi_k\}\,. \tag{2.14}$$

Es sollte beachtet werden, dass sich die Zerlegung (2.14) von der Zerlegung aus (2.7) unterscheidet. Die Zerlegung der Steifigkeitsmatrix wird komplett über die Zerlegung (2.14) erfolgen.

Bezeichnung 2.1.
Eine Indexmenge $\mathcal{I}_k$ bezeichnen wir als "*Interface*" mit zugehöriger Prozessormenge π_k. Wenn eine zugehörige Prozessormenge $\pi_k = \{p\}$ nur aus einem Element besteht, so bezeichnen wir die Indexmenge zusätzlich als das "*Innere*" zu Prozessor p, ansonsten als "*echtes Interface*".

Wir nehmen an, dass jeder Prozessor mindestens ein Element in seinem Inneren besitzt, das heißt, dass für jeden Prozessor p eine aktive Prozessormenge $\pi_k = \{p\}$ existiert.

Bemerkung 2.2.
Auch wenn das Innere im üblichen Sprachgebrauch eigentlich kein Interface darstellen kann, bezeichnen wir dies im Folgenden trotzdem als Interface, da für die meisten Definitionen kein Unterschied gemacht werden muss. Wenn eine Differenzierung zwischen einem "echten" Interface und dem Inneren gemacht werden muss, so wird dies explizit deutlich gemacht. Da π_k und $\mathcal{I}_k$ einander eindeutig zuzuordnen sind, wird auch vom Interface zu π_k gesprochen.

Bezeichnung 2.3.
Ein "echtes" Interface π_k wird zu einem "Inneren" einer Prozessormenge $\mathcal{P}^n$, wenn mehrere Prozessoren $p_1, \ldots, p_n$ zu einer Prozessormenge $\mathcal{P}^n = \{p_1, \ldots, p_n\}$ zusammengefasst werden, so dass $\pi_k \subset \mathcal{P}^n$.

Definition 2.4.
Mit $N_k = |\mathcal{I}_k|$ definieren wir die Anzahl der Unbekannten auf dem Interface $\mathcal{I}_k$.

Die Gesamtanzahl von Unbekannten ist somit $N = \sum_{k=1}^{K} N_k$. Es gilt, dass $N^p = \sum_{p \in \pi_k} N_k$ (vergleiche Kapitel 2.2.2).

Definition 2.5 (Matrixblock über Indexmenge)**.**
Der zu zwei Indexmengen $\mathcal{I}_k$ und $\mathcal{I}_m$ auf Prozessor p zugehörige Matrixblock wird definiert als

$$A^p_{km} = (A^p[i,j])_{i \in \mathcal{I}_k, j \in \mathcal{I}_m} \in \mathbb{R}^{N_k \times N_m} . \tag{2.15}$$

Bemerkung 2.6.
Aufgrund der Additivität gilt nach (2.10)

$$A_{km} = \sum_{p \in \mathcal{P}} A^p_{km} , \tag{2.16}$$

so dass die Steifigkeitsmatrix dargestellt werden kann als

$$A = (A_{km}) = \begin{pmatrix} A_{11} & \cdots & A_{1K} \\ \vdots & & \vdots \\ A_{K1} & \cdots & A_{KK} \end{pmatrix} . \tag{2.17}$$

Lemma 2.7. *Für $p \notin \pi_k \cap \pi_m$ gilt*

$$A^p_{km} = 0 , \tag{2.18}$$

und somit

$$A_{km} = \sum_{p \in \pi_k \cap \pi_m} A^p_{km} .$$

Bemerkung 2.8.
Für den Discontinuous-Galerkin-Ansatz (siehe Kapitel 2.2.5.2) müssen hier die erweiterten Prozessormengen π_k aus (2.12) verwendet werden.

Beweis. Mit (2.9) gilt

$$A^p_{km} = (A^p[i,j])_{i \in \mathcal{I}_k, j \in \mathcal{I}_m} = \left(\sum_{c \in \mathcal{C}^p} a_c(\phi_i, \phi_j) \right)_{i \in \mathcal{I}_k, j \in \mathcal{I}_m} .$$

Wir betrachten wir ein festes $i \in \mathcal{I}_k$ und $j \in \mathcal{I}_m$. Sei $p \notin \pi_k \cap \pi_m$ und $c \in \mathcal{C}^p$. Angenommen, $a_c(\phi_i, \phi_j) \neq 0$.

Für die gebräuchlichen konformen Finiten Elemente aus 2.2.5.1 bedeutet dies $c \in \mathcal{C}_i \cap \mathcal{C}_j$. Mit (2.8) gilt damit $z_i, z_j \in \bar{\Omega}_c$ und da $c \in \mathcal{C}^p$ gilt nach (2.2) auch $z_i, z_j \in \bar{\Omega}^p$. Mit (2.3) gilt damit $i, j \in \mathcal{I}^p$, bzw. $p \in \pi(i) = \pi_k$, $p \in \pi(j) = \pi_m$. Das ist widersprüchlich zu $p \notin \pi_k \cap \pi_m$.

Für den Discontinuous-Galerkin-Ansatz sei oBdA $z_i \in \bar{\Omega}_c$. Damit gibt es nach (2.11) eine Zelle c' so dass $z_j \in \bar{\Omega}_{c'}$ und es existiert ein $\tilde{f} \in \mathcal{F}_c \colon \tilde{f} \in \mathcal{F}_{c'}$. Da $c \in \mathcal{C}^p$ gilt $p \in \pi(i) = \pi_k$ und es gilt für alle Seitenflächen $f \in \mathcal{F}_c \colon p \in \pi(z_f)$, insbesondere also auch für $\tilde{f}$. Wegen $\tilde{f} \in \mathcal{F}_{c'}$ gilt nach (2.12) $p \in \pi(j) = \pi_m$ und somit $p \in \pi_k \cap \pi_m$, welches wieder einen Widerspruch darstellt. □

Korollar 2.9. *Für $\pi_k \cap \pi_m = \emptyset$ gilt $A_{km} = 0$.*

Definition 2.10. *(globale Matrizen)*
Eine Matrix A_{km} heißt "global vorhanden", wenn sie nicht mehr additiv verteilt ist, also komplett auf einem Prozessor vorhanden ist.

2.4.2. Vektorblockstruktur über Interfaces.

Für die rechte Seite f aus Kapitel 2.2.4 sollen dieselben Definitionen gelten wir für die Matrixblöcke. Damit erben sie auch die Eigenschaften dieser. Hier soll daher nur eine Übersicht für die rechte Seite f angegeben werden.

Definition 2.11. *(Vektorblöcke über Indexmenge)*
Der zu einer Indexmenge $\mathcal{I}_k$ auf Prozessor p zugehörige Vektorblock wird definiert als

$$f_k^p = (f^p[i])_{i \in \mathcal{I}_k} \in \mathbb{R}^{N_k}. \tag{2.19}$$

Damit gilt für die globale rechte Seite f wegen (2.6) und (2.5)

$$f = \begin{pmatrix} f_1 \\ \vdots \\ f_K \end{pmatrix} \quad \text{mit } f_k = \sum_{p \in \pi_k} f_k^p. \tag{2.20}$$

Definition 2.12. *(globale Vektoren)*
Ein Vektor f_k heißt "global vorhanden", wenn er nicht mehr additiv verteilt ist, also komplett auf einem Prozessor vorhanden ist.

2.4.3. Beispiel.

Wir betrachten als Beispiel ein zweidimensionales Gebiet, welches in 4 Teilgebiete zerlegt ist und jedes Teilgebiet einem Prozessor zugewiesen wurde (vergleiche Abbildung 1). Diese Zerlegung definiert eine aktive Prozessormenge $\Pi_{\mathcal{I}} = \{\pi_1, ..., \pi_9\}$ mit

$\pi_1 = \{1\}$, $\pi_2 = \{2\}$,
$\pi_3 = \{3\}$, $\pi_4 = \{4\}$,
$\pi_5 = \{1,2\}$, $\pi_6 = \{3,4\}$,
$\pi_7 = \{1,3\}$, $\pi_8 = \{2,4\}$,
$\pi_9 = \{1,2,3,4\}$.

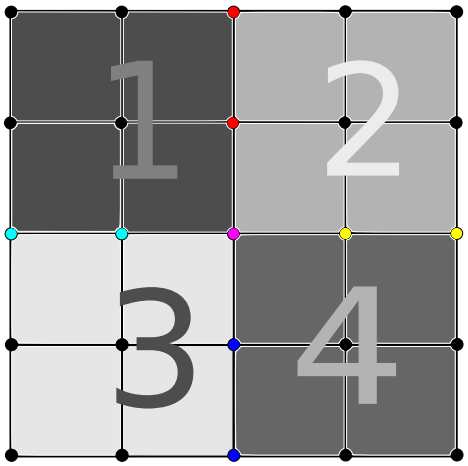


ABBILDUNG 1. Ein Quadrat mit 16 Zellen zerlegt auf 4 Prozessoren

Die Indexmenge $\mathcal{I}$ mit $|\mathcal{I}| = 25$ wird über die "großen Punkte" repräsentiert, während die unterschiedliche Farbgebung die verschiedenen aktiven Prozessormengen darstellen. In Kapitel 3.6.2 wird eine Sortierung der aktiven Prozessormenge angegeben. Jeder Prozessor besitzt somit $N^p = 9$ Unbekannte, wobei jeweils 4 Elemente zum Inneren zu jedem Prozessor gehören und 5 Elemente zum echten Interface.

In Kapitel 7.4.2 findet sich auch ein Beispiel zum Discontinuous-Galerkin-Ansatz, bei dem sich die Indizes in den Zellmittelpunkten befinden.

Teil 2

Die parallele Block-LR-Zerlegung

3

Von der seriellen zur parallelen Block-LR-Zerlegung

Analog zu Kapitel 1 definieren wir eine Block-LR-Zerlegung einer Matrix A, die nun in $K \times K$ Blöcke unterteilt ist.

Auch in diesem Kapitel nehmen wir an, dass alle Operationen gültig sind, also insbesondere, dass die in jedem Schritt zu zerlegenden Diagonal-Block-Matrizen regulär sind. Die Grundidee zur parallelen Block-LR-Zerlegung wurde in einer ersten Veröffentlichung [**MW11**, **MW12**] bereits beschrieben. In dieser Arbeit erweitern wir die Grundidee, so dass eine weitaus verbesserte Parallelität im Algorithmus vorhanden ist.

3.1. Notation

Eine $K \times K$ Block-Matrix $A \in \mathbb{R}^{N \times N}$ ist gegeben als

$$A = (A_{km}) = \begin{pmatrix} A_{11} & \cdots & A_{1K} \\ \vdots & & \vdots \\ A_{K1} & \cdots & A_{KK} \end{pmatrix}$$

mit $A_{km} \in \mathbb{R}^{N_k \times N_m}$ und $N = \sum_{k=1}^{K} N_k$.

Mit Kleinbuchstaben im Matrixeintrag wird wie in Kapitel 1 eine Nichtblockmatrix bezeichnet:

$$A = (a_{mn}) = \begin{pmatrix} a_{11} & \cdots & a_{1N} \\ \vdots & & \vdots \\ a_{M1} & \cdots & a_{MN} \end{pmatrix}.$$

Zusätzlich zu Kapitel 1 wird mit $A[k:m]$ eine Untermatrix von A bezeichnet, mit allen Spalten N und den Zeilen k bis m:

$$A = \left(\begin{array}{c} A[1:k-1] \\ \hline A[k:m] \\ \hline A[m+1:N] \end{array} \right).$$

Die weiteren Notationen aus Kapitel 1 bleiben unverändert.

3.2. Eine serielle Block-LR-Zerlegung

3.2.1. Eine Block-Variante der Vorwärts- bzw. Rückwärts-Substitution.

Gegeben sei eine untere $K \times K$-Block-Dreiecksmatrix $L \in \mathbb{R}^{N\times N}$ mit

$$\begin{pmatrix} L_{11} & 0 & \cdots & 0 \\ L_{21} & L_{22} & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ L_{K1} & L_{K2} & \cdots & L_{KK} \end{pmatrix}$$

Hierbei sind $L_{km} \in \mathbb{R}^{N_k\times N_m}$ Matrizen mit $\sum_{k=1}^{K} N_k = N$, wobei die Diagonalmatrizen L_{kk}, $k = 1, \ldots, K$ regulär sind. Eine Diagonalmatrix muss dabei keine untere Dreiecksmatrix darstellen, so dass das L selbst keine untere Dreiecksmatrix bilden muss.

Gelöst werden soll das System $Lu = f$ mit

$$\begin{pmatrix} L_{11} & 0 & \cdots & 0 \\ L_{21} & L_{22} & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ L_{K1} & L_{K2} & \cdots & L_{KK} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_K \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_K \end{pmatrix}.$$

Dabei ist die rechte Seite f und die gesuchte Lösung u in Vektoren unterteilt mit $u_k, f_k \in \mathbb{R}^{N_k}$.

Wir nehmen an, dass bereits eine Lösungsroutine SOLVE(L_{kk}, f_k) existiert, die eine gegebene rechte Seite f_k überschreibt in $f_k := L_{kk}^{-1} f_k$. Die Zerlegung von L_{kk} kann zum Beispiel wieder über eine LR-Zerlegung geschehen, wie sie in Kapitel 1 vorgestellt wurde. Insbesondere soll dort Pivotisierung möglich sein.

Der Block-Algorithmus für eine untere Block-Dreiecksmatrix sieht dann wie folgt aus.

Algorithmus 3.1 (Block-Vorwärts-Substitution)**.**
Sei $L \in \mathbb{R}^{N\times N}$ eine untere $K \times K$-Block-Dreiecksmatrix und blockweise unterteilt in $L_{km} \in \mathbb{R}^{N_k\times N_m}$ mit passender Unterteilung der rechten Seite $f \in \mathbb{R}^N$ mit $f_k \in \mathbb{R}^{N_k}$. Dann überschreibt der folgende Algorithmus f mit der Lösung des linearen Gleichungssystems $Lu = f$.

```
1  for k = 1, ..., K
2      f_k := SOLVE(L_kk, f_k)
3      for m = k + 1, ..., K
4          f_m := f_m − L_mk f_k
```

Der Algorithmus für die Rückwärts-Substitution wird analog zu Kapitel 1 aufgebaut.

Algorithmus 3.2 (Block-Rückwärts-Substitution)**.**
Sei $R \in \mathbb{R}^{N\times N}$ eine obere normierte $K \times K$-Block-Dreiecksmatrix und blockweise unterteilt in $R_{km} \in \mathbb{R}^{N_k\times N_m}$ mit passender Unterteilung der rechten Seite $f \in \mathbb{R}^N$ mit $f_k \in \mathbb{R}^{N_k}$. Dann überschreibt der folgende Algorithmus f mit der Lösung des linearen Gleichungssystems $Ru = f$.

```
1  for k = K, ..., 2
2      for m = k − 1, ..., 1
3          f_m := f_m − R_mk f_k
```

3.2.2. Block-LR-Zerlegung.

Die Matrix $A \in \mathbb{R}^{N\times N}$ sei zerlegt in $K \times K$ Teilblöcke, so dass $A = (A_{km})_{k,m=1,\dots,K}$ mit $A_{km} \in \mathbb{R}^{N_k \times N_m}$ und $\sum_{k=1}^{K} N_k = N$. Mit dieser Partitionierung lässt sich nun entsprechend der LR-Zerlegung (Algorithmus 1.10) eine Block-LR-Zerlegung angeben. Aufgrund der "Division" in Zeile 3, wird für die Diagonalblöcke A_{kk} eine LR-Zerlegung benutzt (hierbei ist Pivotisierung möglich!) und die Operation $A_{kk}^{-1}A_{km}$ daraufhin mit Hilfe der Vorwärts-/Rückwärts-Substitution gelöst.
Wir bezeichnen die Funktionen folgendermaßen:

- $A_{kk} := \mathrm{LR}(A_{kk})$
  Mit Algorithmus 1.10 wird eine LR-Zerlegung der Matrix A_{kk} erstellt, so dass die Zerlegung wieder in A_{kk} gespeichert wird.
- $A_{km} := \textsc{Solve}(A_{kk}, A_{km})$
  Mit Hilfe der vorigen Zerlegung und der Vorwärts-/Rückwärts-Substitution wird $A_{kk}^{-1}A_{km}$ berechnet und in A_{km} gespeichert.

Somit kann die Block-LR-Zerlegung wie folgt geschrieben werden:

Algorithmus 3.3 (Block-LR-Zerlegung)**.**

```
1  for k = 1, ..., K
2      A_kk := LR(A_kk)
3      for m = k + 1, ..., K
4          A_km = Solve(A_kk, A_km)
5          for n = k + 1, ..., K
6              A_nm = A_nm − A_nk A_km
```

3.3. Eine parallele Block-LR-Zerlegung mit verteilten Spalten

Algorithmus 3.3 soll nun parallelisiert werden, indem einzelne Block-Matrizen auf verschiedenen Prozessoren aufgeteilt werden. Im finalen Algorithmus ist insbesondere aufgrund von Kommunikation eine solche Aufteilung erst für mindestens vier Prozessoren sinnvoll, das heißt, eine solche Aufteilung für die zu zerlegende Matrix wird ab Schritt $s = 2$ stattfinden. Diese Matrix ist dort in jedem Fall vollbesetzt, so dass der kommende

Es seien P Prozessoren in einer Prozessormenge $\mathcal{P} = \{1, \dots, P\}$, sowie eine Matrix $Z \in \mathbb{R}^{N\times N}$ gegeben[1]. Diese Matrix wird nun spaltenweise gleichmäßig auf die Prozessoren verteilt. Sofern N durch P teilbar ist, erhalten alle Prozessoren genau N/P Spalten, andererseits erhalten die ersten p Prozessoren mit $p \le (N \bmod P)$ eine zusätzliche Spalte, die sich als Rest der Division ergibt. Somit werden mit

$$N_p = \begin{cases} \lfloor N/P \rfloor + 1 & p \le (N \mod P) \\ \lfloor N/P \rfloor & p > (N \mod P) \end{cases}$$

die Anzahl der Spalten auf Prozessor p definiert, so dass die Matrix Z so auf die Prozessoren verteilt ist, dass jeder Prozessor $p \in \mathcal{P}$ aufeinanderfolgende N_p Spalten der Matrix Z erhält, das heißt, Prozessor 1 erhält die ersten N_1 Spalten, Prozessor 2 erhält die nächsten N_2 Spalten und so weiter. Die lokalen Matrizen, welche auf Prozessor p zu finden sind, werden mit Z_p bezeichnet, somit kann die globale Matrix

[1] In der weiteren Parallelisierung werden Teilblockmatrizen Z der Gesamtmatrix A zerlegt, so dass sich dieses N auf die jeweils gegebene Größe bezieht

Z geschrieben werden als

$$Z = \left(\begin{array}{c|c|c|c} Z_1 & Z_2 & \cdots & Z_P \end{array} \right).$$

Um eine parallele Block-LR-Zerlegung von Algorithmus 3.3 angeben zu können, werden die Matrizen Z_p weiter unterteilt. Dazu definieren wir einen Zeilenblock $Z_p[i:j]$ so, dass die Zeilen i bis j von Z_p zusammengefasst werden, das heißt es gilt

$$Z_p = \left(\begin{array}{c} Z_p[1:i-1] \\ \hline Z_p[i:j] \\ \hline Z_p[j+1:N] \end{array} \right).$$

Weiterhin definieren wir $J_0 = 0$, $J_p = \sum_{n=1}^{p} N_n$ und unterteilen die Matrix Z_p auf Prozessor p folgendermaßen:

$$Z_p = \begin{pmatrix} Z_p[J_0+1:J_1] \\ \vdots \\ Z_p[J_{p-1}+1:J_p] \\ \vdots \\ Z_p[J_{P-1}+1:J_P] \end{pmatrix}.$$

Damit ist insbesondere $Z_p[J_{p-1}+1:J_p]$ eine $N_p \times N_p$ Block-Matrix. Insgesamt erhalten wir also die globale Block-Matrix

$$Z = \begin{pmatrix} Z_1[J_0+1:J_1] & \cdots & Z_p[J_0+1:J_1] & \cdots & Z_P[J_0+1:J_1] \\ \vdots & & \vdots & & \vdots \\ Z_1[J_{p-1}+1:J_p] & \cdots & Z_p[J_{p-1}+1:J_p] & \cdots & Z_P[J_{p-1}+1:J_p] \\ \vdots & & & & \vdots \\ Z_1[J_{P-1}+1:J_P] & \cdots & Z_p[J_{P-1}+1:J_P] & \cdots & Z_P[J_{P-1}+1:J_P] \end{pmatrix},$$

auf die nun Algorithmus 3.3 angewandt werden soll. Es muss nun beachtet werden, dass die Matrizen Z_p auf Prozessor p zu finden sind und für die Zerlegung Kommunikation zu den anderen Prozessoren benötigt wird. Daher werden für die Kommunikation zwischen zwei Prozessoren p und q zwei Routinen eingeführt:

- SENDE $[p \to q]\,(Z_p)$
  Die Matrix Z_p wird von Prozessor p zu Prozessor q gesendet.
- EMPFANGE $[q \leftarrow p]\,(Z_p)$
  Die Matrix Z_p wird auf Prozessor q von Prozessor p empfangen.

Wenn Prozessor q eine Matrix Z_p von Prozessor p erhält, so befindet sich auf Prozessor q eine Kopie der Matrix Z_p, das heißt, es können nun mit dieser Matrix Operationen durchgeführt werden. Für die parallele Block-LR-Zerlegung werden weiterhin folgende Matrix-Operationen benötigt (vergleiche die Funktionen aus Kapitel 3.2.2).

- $Z_p[i:j] := \mathrm{LR}(Z_p[i:j])$
  Mit Algorithmus 1.10 wird eine LR-Zerlegung der Matrix $Z_p[i:j]$ erstellt, so dass die Zerlegung wieder in $Z_p[i:j]$ gespeichert wird. Hierbei gilt $j - i + 1 = N_p$ gilt, so dass $Z_p[i:j]$ eine quadratische Matrix beschreibt.

- $Z_q[i:j] := \text{SOLVE}(Z_p[i:j], Z_q[i:j])$
  Mit Hilfe der Zerlegung von $Z_p[i:j]$ und der Vorwärts-/Rückwärtssubstitution wird $Z_p[i:j]^{-1} Z_q[i:j]$ berechnet und in $Z_q[i:j]$ gespeichert. Hierbei muss $q \neq p$ gelten.

Damit kann eine parallele Block-LR-Zerlegung für eine Matrix Z definiert werden. Hierbei muss in jedem Schritt bekannt sein, auf welchem Prozessor eine Aktion durchgeführt wird.

Algorithmus 3.4 (parallele Block-LR-Zerlegung mit verteilten Spalten)**.**

```
1  for p = 1, ..., P
2      setze  i = K_{p-1} + 1,   j = K_p
3      auf p:
4          Z_p[i:j] := LR(Z_p[i:j])
5          SENDE [p → q] (Z_p) für q ∈ P
6      auf q ∈ P: EMPFANGE [q ← p] (Z_p)
7      auf q ∈ P mit q > p (parallel):
8          Z_q[i:j] := SOLVE(Z_p[i:j], Z_q[i:j])
9          Z_q[j+1:N] := Z_q[j+1:N] − Z_p[j+1:N] Z_q[i:j]
```

Hierbei wurde die Schleife in Zeile 5 aus Algorithmus 3.3 mit Hilfe der Definition eines Zeilenblocks $Z_q[j+1:N]$ ersetzt.

3.4. Das Schur-Komplement für additive Matrizen

Als Vorbereitung für die Parallelisierung betrachten wir Algorithmus 3.3 für den Fall, dass die Matrizen A_{km} additiv auf P Prozessoren verteilt sind, so dass $A_{km} = \sum_{p=1}^{P} A_{km}^p$ gilt (vergleiche Bemerkung 2.6 und Lemma 2.7). Insbesondere sind wir am Schur-Komplement nach Schritt k interessiert. Zur Einführung des Schur-Komplements siehe auch [**Zha05**].

Wir betrachten den ersten Schritt der sequentiellen LR-Zerlegung aus 1.10 und zerlegen die Matrix $A \in \mathbb{R}^{N \times N}$ in

$$A = \begin{pmatrix} z & r^t \\ l & S \end{pmatrix},$$

wobei $z \in \mathbb{R}$, $r, l \in \mathbb{R}^{N-1}$ und $S \in \mathbb{R}^{N-1 \times N-1}$. Im ersten Schritt wird nun Folgendes berechnet:

$$A_1 = \begin{pmatrix} z & z^{-1} r^T \\ l & S - l z^{-1} r^T \end{pmatrix}.$$

Das Ganze kann auch blockweise betrachtet werden:

Definition 3.5 (Schur-Komplement)**.**
Gegeben sei die Matrix

$$A = \begin{pmatrix} Z & R \\ L & S \end{pmatrix},$$

mit $Z \in \mathbb{R}^{k \times k}$ regulär, $R \in \mathbb{R}^{k \times m}, L \in \mathbb{R}^{m \times k}, S \in \mathbb{R}^{m \times m}$. Die Matrix $S - LZ^{-1}R$ heißt das Schur-Komplement von A in Z beziehungsweise das Schur-Komplement von Z bezüglich A.

Bemerkung 3.6. Für das Schur-Komplement ist es irrelevant, wie die ersten k Schritte durchgeführt werden - ob einzeln wie in Algorithmus 1.10 oder blockweise wie in Algorithmus 3.3. Das Schur-Komplement selbst hat immer die gleiche Form.

Um dies zu zeigen betrachten wir zuerst ein Problem der Art

$$ZX = R \quad \text{mit } Z = \begin{pmatrix} Y & B \\ C & D \end{pmatrix}, R = \begin{pmatrix} R_1 \\ R_2 \end{pmatrix}, X = \begin{pmatrix} X_1 \\ X_2 \end{pmatrix}.$$

Dabei sei Y invertierbar, weitere benötigte Inversen sollen existieren. Gesucht ist eine Lösung $X = Z^{-1}R$. Das System $ZX = R$ ist äquivalent zu

$$YX_1 + BX_2 = R_1 \tag{3.1}$$

$$CX_1 + DX_2 = R_2\,. \tag{3.2}$$

Multiplikation von (3.1) mit $-CY^{-1}$ und Addition zu (3.2) ergibt

$$\begin{aligned} (D - CY^{-1}B)X_2 &= R_2 - CY^{-1}R_1 \\ \Rightarrow X_2 &= (D - CY^{-1}B)^{-1}(R_2 - CY^{-1}R_1)\,, \end{aligned} \tag{3.3}$$

und somit aus (3.1)

$$X_1 = Y^{-1}(R_1 - BX_2) = Y^{-1}(R_1 - B(D - CY^{-1}B)^{-1}(R_2 - CY^{-1}R_1))\,. \tag{3.4}$$

Sei nun A gegeben als

$$A = \begin{pmatrix} Z & R \\ L & S \end{pmatrix} \text{ mit } Z = \begin{pmatrix} Y & B \\ C & D \end{pmatrix}, R = \begin{pmatrix} R_1 \\ R_2 \end{pmatrix}, L = \begin{pmatrix} L_1 & L_2 \end{pmatrix}, \tag{3.5}$$

also

$$A = \begin{pmatrix} Y & B & R_1 \\ C & D & R_2 \\ L_1 & L_2 & S \end{pmatrix}. \tag{3.6}$$

Wir betrachten nun die beiden (Block)-LR-Zerlegungen der Matrix A, indem im ersten Fall die LR-Zerlegung auf die Matrix A aus (3.5) angewandt wird und im zweiten Fall auf die Matrix A aus (3.6). Interessiert sind wir am Schur-Komplement von A in Z.

(1) Zerlegung der Matrix $A = \begin{pmatrix} Z & R \\ L & S \end{pmatrix}$:
Algorithmus 3.3 liefert nach dem ersten Schritt

$$A_1 = \begin{pmatrix} \tilde{Z} & Z^{-1}R \\ L & S - LZ^{-1}R \end{pmatrix},$$

wobei $\tilde{Z}$ die LR-Zerlegung von Z ist. Für das Schur-Komplement $\hat{S}^{(1)} = S - LZ^{-1}R$ folgt also

$$\begin{aligned} \hat{S}^{(1)} &= S - \begin{pmatrix} L_1 & L_2 \end{pmatrix} \begin{pmatrix} Y & B \\ C & D \end{pmatrix}^{-1} \begin{pmatrix} R_1 \\ R_2 \end{pmatrix} \\ &= S - \begin{pmatrix} L_1 & L_2 \end{pmatrix} \begin{pmatrix} Y^{-1}(R_1 - B(D - CY^{-1}B)^{-1}(R_2 - CY^{-1}R_1)) \\ (D - CY^{-1}B)^{-1}(R_2 - CY^{-1}R_1) \end{pmatrix} \\ &= S - L_1Y^{-1}(R_1 - B(D - CY^{-1}B)^{-1}(R_2 - CY^{-1}R_1)) \\ &\quad - L_2(D - CY^{-1}B)^{-1}(R_2 - CY^{-1}R_1) \\ &= S - L_1Y^{-1}R_1 - (L_2 - L_1Y^{-1}B)(D - CY^{-1}B)^{-1}(R_2 - CY^{-1}R_1)\,. \end{aligned}$$

Die zweite Gleichung folgt dabei aus (3.4) und (3.3).

(2) Zerlegung der Matrix $A = \begin{pmatrix} Y & B & R_1 \\ C & D & R_2 \\ L_1 & L_2 & S \end{pmatrix}$:
In diesem Fall müssen zwei Schritte von Algorithmus 3.3 durchgeführt werden. Der erste Schritt liefert

$$A_1 = \begin{pmatrix} \tilde{Y} & Y^{-1}B & Y^{-1}R_1 \\ C & D - CY^{-1}B & R_2 - CY^{-1}R_1 \\ L_1 & L_2 - L_1Y^{-1}B & S - L_1Y^{-1}R_1 \end{pmatrix} .$$

Mit $D_1 = D - CY^{-1}B$ folgt weiter

$$A_2 = \begin{pmatrix} \tilde{Y} & Y^{-1}B & Y^{-1}R_1 \\ C & \tilde{D}_1 & D_1^{-1}(R_2 - CY^{-1}R_1) \\ L_1 & L_2 - L_1Y^{-1}B & \hat{S}^{(2)} \end{pmatrix} ,$$

mit dem Schur-Komplement

$$\begin{aligned} \hat{S}^{(2)} &= S - L_1Y^{-1}R_1 - (L_2 - L_1Y^{-1}B)D_1^{-1}(R_2 - CY^{-1}R_1) \\ &= S - L_1Y^{-1}R_1 - (L_2 - L_1Y^{-1}B)(D - CY^{-1}B)^{-1}(R_2 - CY^{-1}R_1) . \end{aligned}$$

Es gilt $\hat{S}^{(1)} = \hat{S}^{(2)}$ und per Induktion kann dies auf beliebig viele Block-Matrizen (bis zu 1×1-"Blöcken") erweitert werden. Beachte, dass nur das Schur-Komplement gleich ist, die anderen Einträge in der Matrix können aufgrund der unterschiedlichen Berechnungsweisen verschieden sein.

Wie in Definition 3.5 und in der obigen Ausführung der LR-Zerlegung zu sehen, ist das Schur-Komplement additiv zusammengesetzt aus der ursprünglichen Matrix S und einer Kombination der über die LR-Zerlegung schon zerlegten Matrizen. Wie die Addition über die zerlegten Matrizen zu Stande kommt, ist dabei nebensächlich.

Wir betrachten nun eine Matrix

$$A = \begin{pmatrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}$$

in der die Matrix A_{33} (im einfachsten Fall) auf zwei Prozessoren verteilt ist mit $A_{33} = A_{33}^1 + A_{33}^2$. Die oberen Indizes beschreiben dabei, auf welchen Prozessoren die Matrix zu finden ist. Aufgrund der Diagonalstruktur von A im oberen Teil, kann die LR-Zerlegung parallel geschehen, dazu sollen die Matrizen A_{11}, A_{13}, A_{31} auf Prozessor 1 zu finden sein und die Matrizen A_{22}, A_{23}, A_{32} auf Prozessor 2. Eine solche Situation ist gegeben, wenn die Nested Dissection mit einem Schnitt auf das Gebiet Ω angewandt wird und beide Gebiete je einem Prozessor zugeteilt werden. Somit ergibt sich eine Aufteilung der Matrix A zu

$$A = \begin{pmatrix} A_{11}^1 & 0 & A_{13}^1 \\ 0 & A_{22}^2 & A_{23}^2 \\ A_{31}^1 & A_{32}^2 & A_{33}^1 + A_{33}^2 \end{pmatrix} .$$

Lokal betrachtet besitzen die Prozessoren folgende Matrixstruktur, wobei die Nullzeilen und -spalten ignoriert werden:

$$A^1 = \begin{pmatrix} A_{11}^1 & A_{13}^1 \\ A_{31}^1 & A_{33}^1 \end{pmatrix} \quad \text{auf Prozessor 1}$$

$$A^2 = \begin{pmatrix} A_{22}^2 & A_{23}^2 \\ A_{32}^2 & A_{33}^2 \end{pmatrix} \quad \text{auf Prozessor 2} .$$

Das Schur-Komplement von A^1 in A^1_{11} lautet

$$S^1 = A^1_{33} - A^1_{31}(A^1_{11})^{-1}A^1_{13}$$

und für A^2 in A^2_{22}

$$S^2 = A^2_{33} - A^2_{32}(A^2_{22})^{-1}A^2_{23}\,.$$

Für das Schur-Komplement von A in $\begin{pmatrix} A_{11} & 0 \\ 0 & A_{22} \end{pmatrix}$ gilt

$$\begin{aligned} S &= A_{33} - \begin{pmatrix} A_{31} & A_{32} \end{pmatrix} \begin{pmatrix} A_{11} & 0 \\ 0 & A_{22} \end{pmatrix}^{-1} \begin{pmatrix} A_{13} \\ A_{23} \end{pmatrix} \\ &= A_{33} - \begin{pmatrix} A_{31} & A_{32} \end{pmatrix} \begin{pmatrix} A_{11}^{-1} & 0 \\ 0 & A_{22}^{-1} \end{pmatrix} \begin{pmatrix} A_{13} \\ A_{23} \end{pmatrix} \\ &= A_{33} - A_{31}(A_{11})^{-1}A_{13} - A_{32}(A_{22})^{-1}A_{23} \\ &= A^1_{33} + A^2_{33} - A_{31}(A_{11})^{-1}A_{13} - A_{32}(A_{22})^{-1}A_{23} \\ &= S^1 + S^2\,. \end{aligned}$$

Das Schur-Komplement kann also – sofern der linke obere Teil eine Diagonalstruktur aufweist – parallel berechnet werden und die lokalen Schur-Komplemente ergeben additiv das globale Schur-Komplement.

Bemerkung 3.7. Da hier die Berechnung über eine Block-LR-Zerlegung geschieht, müssen die zu zerlegenden Matrizen (im obigen Fall A_{11} und A_{22}) global vorhanden – also nicht mehr additiv verteilt – sein. Das (globale) Schur-Komplement kann jedoch noch additiv verteilt sein. Allgemein gilt: Zur Berechnung des Schur-Komplements einer Matrix

$$A = \left(\begin{array}{ccc|c} Z_1 & 0 & 0 & R_1 \\ 0 & \ddots & 0 & \vdots \\ 0 & 0 & Z_T & R_T \\ \hline L_1 & \dots & L_T & S \end{array}\right)$$

in $\operatorname{diag}(Z_1, \dots, Z_T)$ müssen die Matrizen Z_t, R_t und L_t, $t = 1, \dots, T$ global vorhanden sein und das Schur-Komplement $S - \sum_{t=1}^{T} L_t Z_t^{-1} R_t$ kann parallel berechnet werden, da $L_t Z_t^{-1} R_t$ für $t = 1, \dots, T$ unabhängig voneinander sind.

Das Schur-Komplement muss im Verlauf ebenso zerlegt werden. Daher müssen die additiv verteilten Matrizen zusammengefasst werden. Dies kann aufgrund der Struktur der Matrix, die aus der Nested Dissection folgt, nach und nach geschehen, so dass auch das Schur-Komplement in mehreren Schritten weiter zerlegt werden kann. Insgesamt erfordert dies allerdings Kommunikation zwischen den betroffenen Prozessoren.

Bisher wurde gezeigt, dass das Schur-Komplement parallel berechnet werden kann, wenn sich die betroffenen Matrizen auf verschiedenen Prozessoren befinden. Nun müssen die Teilmatrizen auf den Prozessoren jedoch nicht in der gleichen Reihenfolge gegeben sein, das heißt, die Spalten und Zeilen könnten permutiert sein[2]. Dass dies jedoch ebenso nur eine Permutation im Schur-Komplement verursacht, wird nun gezeigt.

[2]Hierbei soll eine beidseitige Permutation in Betracht gezogen werden, so dass Diagonalelemente wieder auf Diagonalelementen abgebildet werden.

Dazu sei eine Matrix

$$A = \begin{pmatrix} Z & R \\ L & S \end{pmatrix} \tag{3.7}$$

mit dem Schur-Komplement von A in Z mit

$$\hat{S} = S - LZ^{-1}R$$

gegeben.

Definition 3.8 (Permutationsmatrix)**.**
Eine Permutationsmatrix $P \in \mathbb{R}^{N\times N}$ besitzt in jeder Zeile und in jeder Spalte genau einen Eintrag 1*, ansonsten* 0*.*

Bemerkung 3.9.

a) Eine Permutationsmatrix $P \in \mathbb{R}^{N\times N}$ angewandt von links auf eine Matrix $A \in \mathbb{R}^{N\times M}$ (also PA) vertauscht die Zeilen von A.
b) Eine Permutationsmatrix $P \in \mathbb{R}^{M\times M}$ angewandt von rechts auf eine Matrix $A \in \mathbb{R}^{N\times M}$ (also AP) vertauscht die Spalten von A.
c) Eine Permutationsmatrix ist orthogonal, das heißt es gilt $P^{-1} = P^T$.
d) Sei $Z \in \mathbb{R}^{N\times N}$ invertierbar und $P \in R^{N\times N}$ eine Permutationsmatrix. Dann gilt $(PZP^T)^{-1} = PZ^{-1}P^T$, denn $(PZP^T)(PZ^{-1}P^T) = PZZ^{-1}P^T = PP^T = I$.

Sei P eine Permutationsmatrix der Größe von Z und Q eine Permutationsmatrix der Größe von S. Nun werden die Zeilen und Spalten der Matrix (3.7) so permutiert, dass die Diagonaleinträge von Z bzw. S wieder auf der Diagonalen stehen, das heißt

$$\tilde{A} = \begin{pmatrix} P & 0 \\ 0 & Q \end{pmatrix} \begin{pmatrix} Z & R \\ L & S \end{pmatrix} \begin{pmatrix} P^T & 0 \\ 0 & Q^T \end{pmatrix} = \begin{pmatrix} PZP^T & PRQ^T \\ QLP^T & QSQ^T \end{pmatrix} = \begin{pmatrix} \tilde{Z} & \tilde{R} \\ \tilde{L} & \tilde{S} \end{pmatrix}.$$

Für das Schur-Komplement von $\tilde{A}$ in $\tilde{Z}$ gilt nun

$$\begin{aligned} \hat{S}_{\text{perm}} &= \tilde{S} - \tilde{L}\tilde{Z}^{-1}\tilde{R} \\ &= QSQ^T - QLP^T(PZP^T)^{-1}PRQ^T \\ &= QSQ^T - QLP^TPZ^{-1}P^TPRQ^T \\ &= QSQ^T - QLZ^{-1}RQ^T \\ &= Q(S - LZ^{-1}R)Q^T \\ &= Q\hat{S}Q^T. \end{aligned}$$

Somit gilt für das Schur-Komplement einer permutierten Matrix, dass dieses eine Permutation des Schur-Komplements der ursprünglichen Matrix ist. Somit können die Zeilen und Spalten auf den Prozessoren unterschiedlich sortiert sein, bei der additiven Zusammenfassung zweier Schur-Komplemente muss daher nur eine eventuelle Permutation beachtet werden.

3.5. Motivation: Visuelle Darstellung der parallelen Block-LR-Zerlegung

Zur Darstellung des zu konstruierenden Algorithmus betrachten wir ein Gebiet Ω im Zweidimensionalen und zeigen, welche Elemente nach und nach eliminiert werden und wie die Matrizen auf den Prozessoren verteilt sind und aufgebaut werden. Im Verlauf genügt es für jeden Schritt jeweils eine Prozessormenge, welche jeweils einen Cluster bildet, zu betrachten, für die anderen Prozessormengen gelten entsprechende Aussagen.

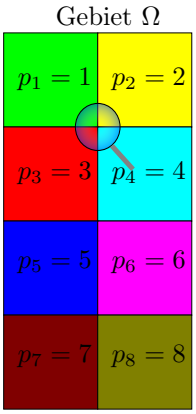


Abbildung 1. Gesamtgebiet Ω verteilt auf 8 Prozessoren

Gegeben sei ein Gebiet $\Omega \subset \mathbb{R}^2$, welches auf $P = 8$ Prozessoren verteilt ist (farblich markierte Gebiete).

Das Gebiet ist aufgeteilt in Zellen $c \in \mathcal{C}$, wobei sich eine Zelle auf jeweils einem Prozessor p befindet.

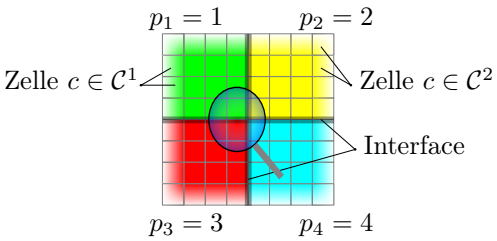


Abbildung 2. Zellenverteilung in der Nähe eines Interfaces

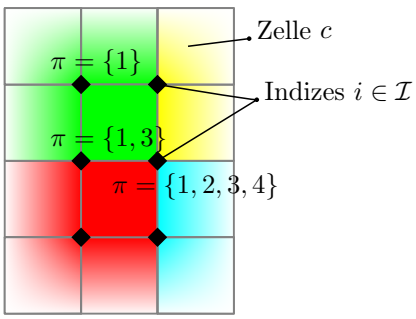


Abbildung 3. Knotenpunkte in der Nähe eines Interfaces

Jede Zelle besitzt Indizes für das Finite Elemente Verfahren. Diese befinden sich entweder auf nur einem Prozessor (zum Beispiel $\pi = \{1\}$) – das sogenannte Innere – oder auf einem echten Interface (zum Beispiel $\pi = \{1, 3\}$). Elemente mit einem gemeinsamen Interface werden zu einer Interfacemenge zusammengefasst. Insgesamt existieren 21 verschiedene Interfacemengen, davon 8 mit $|\pi| = 1$, 10 mit $|\pi| = 2$ und 3 mit $|\pi| = 4$ (vergleiche Tabelle 2).

Nun werden kombinierte Prozessormengen $\mathcal{P}^{s,t}$ erstellt (nach Definition 3.10): In Schritt $s = 0$ existieren insgesamt 8 Prozessormengen mit je einem Prozessor $\mathcal{P}^{0,t} = \{p_t\} = \{t\}$, $t = 1, \ldots, 8$. In jedem folgenden Schritt werden jeweils zwei Prozessormengen zusammengefasst. Jede kombinierte Prozessormenge beschreibt dabei einen Cluster t. Für die Schritte gilt somit:

Schritt s	Anzahl Cluster t	Prozessormengen $\mathcal{P}^{s,t}$
0	8	$\{1\}$ $\{2\}$ $\{3\}$ $\{4\}$ $\{5\}$ $\{6\}$ $\{7\}$ $\{8\}$
1	4	$\{1,2\}$ $\{3,4\}$ $\{5,6\}$ $\{7,8\}$
2	2	$\{1,2,3,4\}$ $\{5,6,7,8\}$
3	1	$\{1,2,3,4,5,6,7,8\}$

TABELLE 1. kombinierte Prozessormengen

Jeder Interfacemenge wird nun seine zugehörige Schrittnummer s (nach (3.12)) und der jeweilige Cluster t (nach (3.15)) zugewiesen. Als sortierte Interfaceliste ergibt sich

Interface	Inhalt	Schritt s	Cluster t	Interface	Inhalt	Schritt s	Cluster t
π_1	1	0	1	π_{13}	1,3	2	1
π_2	2	0	2	π_{14}	2,4	2	1
π_3	3	0	3	π_{15}	1,2,3,4	2	1
π_4	4	0	4	π_{16}	5,7	2	2
π_5	5	0	5	π_{17}	6,8	2	2
π_6	6	0	6	π_{18}	5,6,7,8	2	2
π_7	7	0	7				
π_8	8	0	8	π_{19}	3,5	3	1
				π_{20}	4,6	3	1
π_9	1,2	1	1	π_{21}	3,4,5,6	3	1
π_{10}	3,4	1	2				
π_{11}	5,6	1	3				
π_{12}	7,8	1	4				

TABELLE 2. sortierte Interfaceliste

Der Tabelle 2 entnehmen wir, dass in Schritt $s = 0$ alle Elemente von π_1 bis π_8 auf 8 Clustern eliminiert werden. In Schritt $s = 2$ werden die Elemente von π_{13} bis π_{18} auf 2 Clustern eliminiert.

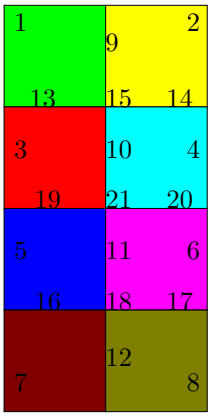


ABBILDUNG 4. Position und Nummern der Interfaces

In Abbildung 4 sind Positionen k der Interfaces π_k beschrieben. Aus Tabelle 2 können die zugehörigen Schritte und Cluster eines Interfaces π_k entnommen werden.
Zum Beispiel gilt für $\pi_1 = \{1\}$:
$s(\pi_1) = 0$, $t(\pi_1) = 1$
beziehungsweise für $\pi_{14} = \{2, 4\}$:
$s(\pi_{14}) = 2$, $t(\pi_{14}) = 1$,
das heißt, das Element π_{14} wird in Schritt $s = 2$ innerhalb des Clusters mit der Nummer 1 (dies entspricht der Prozessormenge $\mathcal{P}^{2,1} = \{1, 2, 3, 4\}$) eliminiert.

Auf jedem Prozessor p wird nun eine lokale Matrix über alle Indizes $i \in \mathcal{I}$ mit $p \in \pi(i)$ aufgebaut. Die Indizes auf dem Interface beschreiben das Schur-Komplement, während die Indizes mit $p = \pi(i)$ diejenigen Elemente beschreiben, die im ersten Schritt eliminiert werden. Dies wird über eine zu zerlegende Operationsmenge $\mathcal{K}_{\mathrm{LR}}^{s,t}$ und der zugehörigen Schur-Komplement-Menge $\mathcal{K}_{\mathrm{Sch}}^{s,t}$ (nach Definition 3.18) beschrieben.

Allgemein wird eine lokale Matrix $A_{\mathrm{loc}}^{(s),t}$ aufgeteilt in

$$\begin{pmatrix} Z^{(s),t} & R_{\mathrm{loc}}^{(s),t} \\ L_{\mathrm{loc}}^{(s),t} & S_{\mathrm{loc}}^{(s),t} \end{pmatrix}$$

(vergleiche (3.18)), wobei jede einzelne Matrix zusätzlich auf die vorhandenen Prozessoren im jeweiligen Cluster verteilt werden (nach (3.24)).

Wir betrachten ab jetzt den zufällig ausgewählten Prozessor $p_4 = 4$ und die jeweils zugehörige Prozessormenge. Die zu zerlegende Operationsmenge im ersten Schritt ist hierbei $\mathcal{K}_{\mathrm{LR}}^{0,4} = \{4\}$ und die zugehörige Schur-Komplement-Menge $\mathcal{K}_{\mathrm{Sch}}^{0,4} = \{10, 14, 15, 20, 21\}$. Das heißt, diejenigen Elemente von π_4 werden auf Prozessor 4 in diesem Schritt eliminiert, während sich die Elemente $\pi_{10}, \pi_{14}, \pi_{15}, \pi_{20}, \pi_{21}$ im Schur-Komplement befinden.

Die Prozessormenge $\mathcal{P}^{0,4} = \{4\}$ besteht aus nur einem Prozessor und die gesamte lokale Matrix $A_{\mathrm{loc}}^{(0),4}$ ist einem Prozessor zu finden. In Schritt $s = 0$ gilt somit

$$A_{\mathrm{loc}}^{(0),4} = \begin{pmatrix} Z^{(0),4} & R_{\mathrm{loc}}^{(0),4} \\ L_{\mathrm{loc}}^{(0),4} & S_{\mathrm{loc}}^{(0),4} \end{pmatrix} = \begin{pmatrix} Z_1^{(0),4} & R_1^{(0),4} \\ L_1^{(0),4} & S_1^{(0),4} \end{pmatrix},$$

wobei $Z^{(0),4}$ aus den Elementen von π_4 aufgestellt wird und $S_{\mathrm{loc}}^{(0),4}$ aus den Elementen von π_k mit $k \in \{10, 14, 15, 20, 21\}$ (vergleiche Abbildung 5 und 6). Alle Elemente in π_4 werden auf dem Prozessor p_4 eliminiert[3] und das zugehörige Schur-Komplement gebildet. Hierbei ist $Z^{(0),4}$ im Sparse-Format gegeben, so dass dafür ein Sparse-Löser genutzt werden kann.

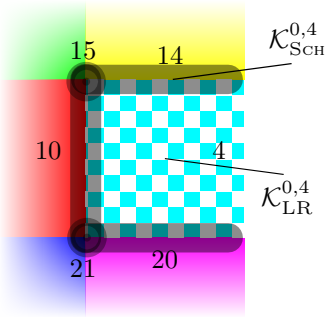


Abbildung 5. Elemente der Matrix $A_{\mathrm{loc}}^{(0),4}$

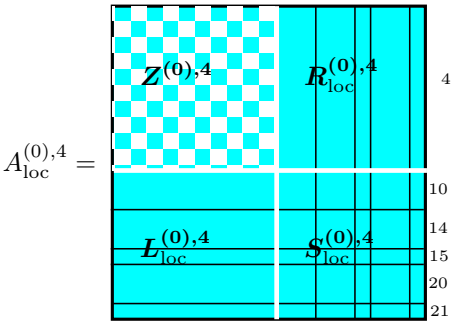


Abbildung 6. Die Matrix $A_{\mathrm{loc}}^{(0),4}$ aufgeteilt auf die Prozessormenge $\mathcal{P}^{0,4}$

[3] genauer: Auf der Prozessormenge $\mathcal{P}^{0,4} = \{4\}$

In Schritt $s = 1$ befindet sich Prozessor p_4 im Cluster $t = 2$ mit der Prozessormenge $\mathcal{P}^{1,2} = \{3, 4\}$. Hier wird eine lokale Matrix

$$A_{\text{loc}}^{(1),2} = \begin{pmatrix} Z^{(1),2} & R_{\text{loc}}^{(1),2} \\ L_{\text{loc}}^{(1),2} & S_{\text{loc}}^{(1),2} \end{pmatrix} = \begin{pmatrix} Z_1^{(1),2} & R_1^{(1),2} & R_2^{(1),2} \\ L_1^{(1),2} & S_1^{(1),2} & S_2^{(1),2} \end{pmatrix},$$

aus den beiden Schur-Komplementen $S_{\text{loc}}^{(0),4}$ und $S_{\text{loc}}^{(0),3}$ gebildet und auf die Prozessoren p_3 und p_4 verteilt (vergleiche Abbildung 8). In Schritt $s = 1$ ist dabei die zu zerlegende Matrix Z auf nur einem Prozessor gegeben, da bei einer Verteilung auf zwei Prozessoren die Zerlegung von $Z^{(1),2}$ sequentiell ablaufen würde und zusätzlich Kommunikation betrieben werden müsste, so dass sich eine Verteilung eher nachteilig auswirkt[4].

Die zugehörigen Operationsmengen sind hierbei

$$\mathcal{K}_{\text{LR}}^{1,2} = \{10\}$$
$$\mathcal{K}_{\text{Sch}}^{1,2} = \{13, 14, 15, 19, 20, 21\},$$

das heißt, in diesem Schritt werden die "inneren" Elemente mit der Interfacemenge $\pi_{10} = \{3, 4\}$ zerlegt und das Schur-Komplement auf dem "Rand" der Prozessormenge $\mathcal{P}^{1,2}$ gebildet.

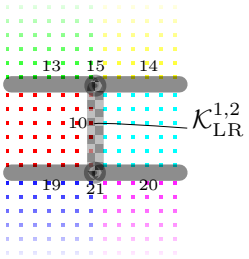


Abbildung 7. Elemente der Matrix $A_{\text{loc}}^{(1),2}$

Die Matrix $Z^{(1),2}$ und $L_{\text{loc}}^{(1),2}$ befindet sich dabei komplett auf Prozessor p_3, die Matrizen $R_{\text{loc}}^{(1),2}$ und $S_{\text{loc}}^{(1),2}$ sind jeweils zur Hälfte aufgeteilt auf die Prozessoren p_3 und p_4.

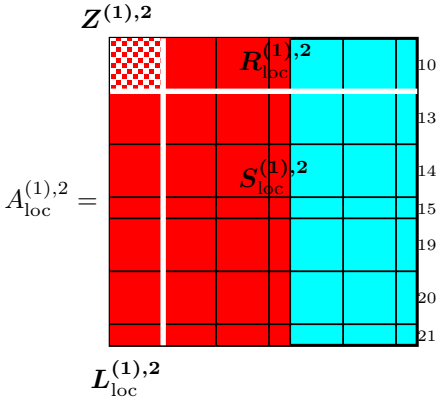


Abbildung 8. Die Matrix $A_{\text{loc}}^{(1),2}$ aufgeteilt auf die Prozessormenge $\mathcal{P}^{1,2}$

Auf Prozessor p_3 wird nun die komplette Matrix $Z^{(1),2}$ zerlegt und an Prozessor p_4 geschickt. Daraufhin berechnen beide Prozessoren ihren jeweiligen Teil des Schur-Komplements $S_{\text{loc}}^{(1),2}$.

Im nächsten Schritt $s = 2$ befinden sich 4 Prozessoren in der Prozessormenge $\mathcal{P}^{2,1}$. Somit lohnt sich auch eine Verteilung der zu zerlegenden Matrix $Z^{(2),1}$ auf alle Prozessoren. Für die zu zerlegende Operationsmenge gilt $\mathcal{K}_{\text{LR}}^{2,1} = \{13, 14, 15\}$ und für die Schur-Komplement-Menge $\mathcal{K}_{\text{Sch}}^{2,1} = \{19, 20, 21\}$. Die Matrix $A_{\text{loc}}^{(2),1}$ wird wieder aus den beiden Schur-Komplementen $S_{\text{loc}}^{(1),2}$ und $S_{\text{loc}}^{(1),1}$ aufgebaut.

[4]Dies wird im späteren Algorithmus nicht ausdrücklich verlangt und soll hier als Zusatzinformation angegeben werden

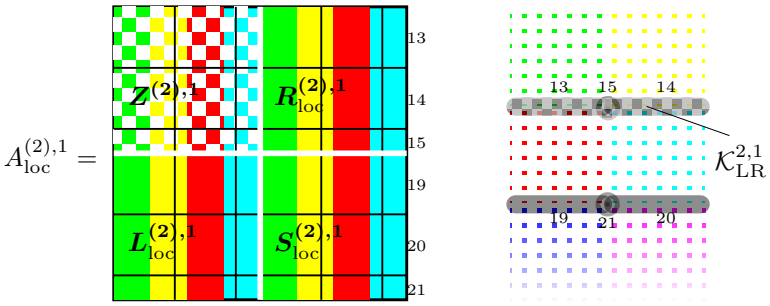


ABBILDUNG 9. Die Matrix $A_{\text{loc}}^{(2),1}$ aufgeteilt auf die Prozessormenge $\mathcal{P}^{2,1}$ mit zugehörigen Elementen

Hierbei gilt $Z^{(2),1} = \begin{pmatrix} Z_1^{(2),1} & Z_2^{(2),1} & Z_3^{(2),1} & Z_4^{(2),1} \end{pmatrix}$, wobei sich $Z_1^{(2),1}$ auf Prozessor p_1 befindet etc. Für die anderen Matrizen gilt eine analoge Verteilung. In dieser Prozessormenge zerlegt nun Prozessor p_1 seinen oberen Teil der Matrix $Z_1^{(2),1}$ und schickt die gesamte Matrix $Z_1^{(2),1}$ und $L_1^{(2),1}$ an alle anderen Prozessoren in der Prozessormenge $\mathcal{P}^{2,1}$, das heißt an p_2, p_3 und p_4. Diese aktualisieren ihr Schur-Komplement, so dass danach Prozessor p_2 seinen Teil der Matrix $Z_2^{(2),1}$ zerlegen und wieder an die restlichen Prozessoren verschicken kann, und so weiter. Am Ende des Schrittes gilt für das aktualisierte Schur-Komplement $\tilde{S}_{\text{loc}}^{(2),1} = S_{\text{loc}}^{(2),1} - L_{\text{loc}}^{(2),1}(Z^{(2),1})^{-1}R_{\text{loc}}^{(2),1}$, welches auf 4 Prozessoren verteilt ist.

Im letzten Schritt $s = 3$ muss kein Schur-Komplement mehr berechnet werden, so dass nur eine zu zerlegende Operationsmenge $\mathcal{K}_{\text{LR}}^{3,1} = \{19, 20, 21\}$ existiert (diese ist im letzten Schritt immer gleich der Schur-Komplement-Menge des vorletzten Schritts) und die Schur-Komplement-Menge $\mathcal{K}_{\text{SCH}}^{3,1}$ leer ist. Somit existiert die Matrix $A_{\text{loc}}^{(3),1}$ auch nur aus der Matrix $Z^{(3),1}$, welche nun auf der gesamten Prozessormenge $\mathcal{P}^{3,1}$ bestehend aus allen 8 Prozessoren p_1 bis p_8 verteilt ist.

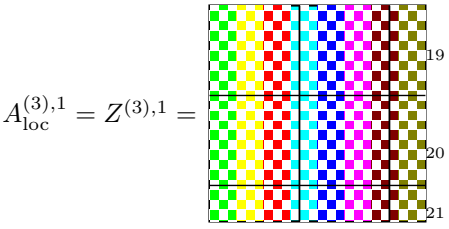


ABBILDUNG 10. Die Matrix $A_{\text{loc}}^{(3),1}$ aufgeteilt auf die Prozessormenge $\mathcal{P}^{3,1} = \mathcal{P}$ mit zugehörigen Elementen

Hierauf wird eine parallele Block-LR-Zerlegung nach Algorithmus 3.4 ohne zusätzliche Schur-Komplement-Berechnung angewandt.

3.6. Die parallele Block-LR-Zerlegung für $P = 2^S$ Prozessoren

Die parallele Block-LR-Zerlegung basiert in erster Linie auf dem Ansatz der Nested Dissection (geschachtelte Zerlegung), welche in Kapitel 2.3 beschrieben wurde. Mit insgesamt P Prozessoren werden dabei einzeln (also komplett parallel) das jeweilige Innere der zugeteilten Gebiete gelöst und das Schur-Komplement auf dem Rand erstellt. Daraufhin werden jeweils zwei Prozessoren zu einer *kombinierten Prozessormenge* zusammengefasst. Dabei ergibt sich auf einem Teil des Separators ein neues "Inneres", welches wiederum gelöst werden kann, da dieses – wie wir zeigen

werden – unabhängig ist von den anderen "Inneren" der anderen Prozessormengen. Rekursiv werden die neuen Prozessormengen zusammengefasst, bis zum Schluss nur noch eine kombinierte Prozessormenge übrig bleibt, auf der der letzte Separator gelöst werden kann. Dies beschreibt jedoch nur einen Teil der Parallelisierung. Die zweite Parallelisierung erfolgt aus der Verteilung des jeweiligen "Inneren" auf alle Prozessoren in der jeweiligen kombinierten Prozessormenge.

3.6.1. Reduzierung der Block-LR-Zerlegung.

Zur Vereinfachung wird angenommen, dass wir eine Zweierpotenz an Prozessoren zur Verfügung haben (also $P = 2^S$), auf denen das Gebiet Ω für die Finite Elemente Diskretisierung und damit die Matrizen aus Kapitel 2.4 verteilt sind. Dies ist für die Praxis auch sinnvoll, wie wir später sehen werden. Allgemein kann aber eine beliebige Anzahl an Prozessoren gewählt werden.

Wir starten mit einer geeigneten Nummerierung der Prozessoren in der Prozessormenge

$$\mathcal{P} = \{p_1, p_2, \dots, p_{2^S}\}\,. \tag{3.8}$$

Die Reduzierung der Block-LR-Zerlegung basiert nun auf der rekursiven Definition von "kombinierten Prozessormengen" $\mathcal{P}^{s,t}$. Dabei beschreibt s die aktuelle Schrittnummer (beginnend von $s = 0$) und $t \in \mathcal{T}_s$ eine Clusternummer. Innerhalb eines Schrittes werden unterschiedliche Cluster jeweils parallel arbeiten können. In jedem Schritt s gilt dabei $\bigcup_{t\in\mathcal{T}_s} \mathcal{P}^{s,t} = \mathcal{P}$, wobei die Zerlegungen in jedem Schritt disjunkt sind. Der erste Schritt $s = 0$ beginnt mit

$$\mathcal{P}^{0,t} = \{p_t\}, \quad t \in \mathcal{T}_0 = \{1, \dots 2^S\}\,. \tag{3.9}$$

Rekursiv werden kombinierte Prozessormengen für die nächsten Schritte definiert:

$$\mathcal{P}^{s,t} = \mathcal{P}^{s-1,2t-1} \cup \mathcal{P}^{s-1,2t}, \quad s = 1, \dots, S, \quad t \in \mathcal{T}_s = \{1, \dots, 2^{S-s}\}\,. \tag{3.10}$$

Das heißt, für Schritt s werden je zwei Prozessormengen aus Schritt $s-1$ zusammengefasst. Da wir eine Zweierpotenz an Prozessoren zur Verfügung haben, sind alle Mengen nichtleer und es gilt in jedem Schritt $\bigcup_{t\in\mathcal{T}_s} \mathcal{P}^{s,t} = \mathcal{P}$. Aufgrund der Ordnung in der Nummerierung der Prozessoren können wir die Prozessormengen auch direkt definieren.

Definition 3.10 (kombinierte Prozessormenge)**.**
Gegeben seien $P = 2^S$ Prozessoren in einer Prozessormenge $\mathcal{P} = \{p_1, \dots, p_P\}$. Für den Schritt $s = 0, \dots, S$ definieren wir $T_s = 2^{S-s}$ Cluster in einer Clustermenge $\mathcal{T}_s = \{1, \dots, T_s\}$ mit

$$\mathcal{P}^{s,t} = \{p_j : 2^s(t-1) < j \le 2^s t\}, \quad t = 1, \dots T_s\,. \tag{3.11}$$

$\mathcal{P}^{s,t}$ heißt kombinierte Prozessormenge *in Schritt s zum Cluster t.*

Bemerkung 3.11.
Definition 3.10 setzt eine sinnvolle Nummerierung der Prozessoren voraus. Insbesondere sollten bei der Zusammenfassung zweier kombinierter Prozessormengen diese jeweils ein gemeinsames Interface besitzen. Alle weiteren Definitionen und der entwickelte Algorithmus gelten jedoch auch, wenn kein gemeinsames Interface vorhanden ist. Dadurch geht allerdings etwas Parallelisierungsarbeit verloren.

Über die aktive Prozessormenge $\Pi_{\mathcal{I}}$ aus (2.13) wird nun bestimmt, welche Elemente zu welchem Zeitpunkt eliminiert werden können. Dazu wird für jedes $\pi \in \Pi_{\mathcal{I}}$ eine zugehörige Schrittnummer über

$$s(\pi) = \min\{s \mid \exists t \in \{1, \dots, T_s\} \colon \pi \subset \mathcal{P}^{s,t}\} \tag{3.12}$$

definiert. Somit erhalten wir eine Teilmenge der aktiven Prozessormenge für jeden Schritt s

$$\Pi^s = \{\pi \in \Pi_{\mathcal{I}} \colon s(\pi) = s\} \tag{3.13}$$

und eine weitere Unterteilung für jeden Cluster

$$\Pi^{s,t} = \{\pi \in \Pi^s \colon \pi \subset \mathcal{P}^{s,t}\}\,. \tag{3.14}$$

Bemerkung 3.12.
Für $s = 0, \dots, S$ gilt $\bigcup_{t=1}^{T_s} \Pi^{s,t} = \Pi^s$, sowie $\Pi^{s,t} \cap \Pi^{s,\tilde{t}} = \emptyset$ für $t \neq \tilde{t}$.

Dies ergibt eine disjunkte Zerlegung der aktiven Prozessormenge

$$\Pi_{\mathcal{I}} = \Pi^0 \cup \dots \cup \Pi^S = \bigcup_{s=0}^{S} \bigcup_{t=1}^{T_s} \Pi^{s,t}\,.$$

Für ein Element π der aktiven Prozessormenge wird weiterhin der zugehörige Eliminierungscluster t definiert über

$$t(\pi) = t \quad \text{mit} \quad \pi \subset \Pi^{s(\pi),t}\,. \tag{3.15}$$

Da die Zerlegung der aktiven Prozessormenge disjunkt ist, ist dieses t eindeutig.

Im Folgenden geben wir eine Sortierung der Elemente in der aktiven Prozessormenge über eine Totalordnung an. Mit Hilfe dieser Sortierung wird dann die globale Matrix A erstellt, auf der die Block-LR-Zerlegung angewandt wird.

Definition 3.13 (Totalordnung der aktiven Prozessormenge)**.**
Für zwei Elemente $\pi_k, \pi_l \in \Pi_{\mathcal{I}}$ mit $\pi_k \neq \pi_l$ gelte folgende Totalordnung:

$$\pi_k < \pi_l \quad \Leftrightarrow \quad \begin{array}{l} \textit{(1)}\ s(\pi_k) < s(\pi_l)\ \textit{(aus (3.12))} \\ \textit{(2)}\ t(\pi_k) < t(\pi_l)\ \textit{(aus (3.15))} \\ \textit{(3)}\ |\pi_k| < |\pi_l| \\ \textit{(4)}\ \min\{p \colon p \in \pi_k \setminus \pi_l\} < \min\{q \colon q \in \pi_l \setminus \pi_k\} \end{array}$$

Dabei wird (2) bis (4) nur dann angewandt, wenn die jeweils vorige Ungleichung mit einem Gleichheitszeichen gilt.

Die $K = |\Pi_{\mathcal{I}}|$ Elemente der aktiven Prozessormenge erhalten somit eine eindeutige Nummerierung $\Pi_{\mathcal{I}} = \{\pi_1, \dots, \pi_K\}$. Da wir annehmen, dass für jeden Prozessor innere Elemente existieren, gilt für die ersten π_k und insbesondere im Schritt $s = 0$: $\pi_k = \{k\}$ und $\Pi^{0,k} = \{\pi_k\}$, $k = 1, \dots, P$.

Für den Algorithmus der ersten Block-LR-Zerlegung werden weiterhin Zählvariablen mit Hilfe der Clusterunterteilung $\Pi^{s,t}$ definiert. Dabei wird die Sortierung der $\pi \in \Pi_{\mathcal{I}}$ berücksichtigt.

Definition 3.14 (Zählvariablen)**.**
Für jeden Schritt $s = 0, \dots, S$ werden Zählvariablen K_s definiert, so dass

$$\Pi^s = \{\pi_k \colon k = K_{s-1} + 1, \dots, K_s\} \quad (s = 0, \dots, S)$$

gilt. Weiterhin werden für jeden Cluster $t = 1, \dots, T_s$ zusätzliche Zählvariablen $K_{s,t}$ definiert, so dass

$$\Pi^{s,t} = \{\pi_k \colon k = K_{s,t-1} + 1, \dots, K_{s,t}\} \quad (s = 0, \dots, S, \quad t = 1, \dots, T_s)$$

gilt.

Diese Definition ist wohldefiniert aufgrund der disjunkten Zerlegung der aktiven Prozessormenge und der Nummerierung der Elemente von π_1 bis π_K.

Bemerkung 3.15.
Wegen (3.13) und (3.14) gilt folgende Eigenschaft der Zählvariablen K_s und $K_{s,t}$ aus Definition 3.14:

$$\begin{aligned} K_{-1} &= 0 & K_{0,0} &= 0 \\ K_S &= K & K_{S,1} &= K \\ K_s &= K_{s,T_s} & K_{s,0} &= K_{s-1,T_{s-1}} . \end{aligned}$$

Bemerkung 3.16.
Die Elemente $\pi \in \Pi_{\mathcal{I}}$ können umsortiert werden, indem eine andere Sortierung der Prozessoren in der Prozessormenge $\mathcal{P} = \{p_1, \dots, p_{2^S}\}$ angewandt wird.

Durch die Sortierung der aktiven Prozessormenge kann die globale Steifigkeitsmatrix nun wie in (2.17) dargestellt werden als

$$A = (A_{km}) = \begin{pmatrix} A_{11} & \cdots & A_{1K} \\ \vdots & & \vdots \\ A_{K1} & \cdots & A_{KK} \end{pmatrix} , \tag{3.16}$$

wobei die Matrizen A_{km} definiert sind über (2.16).

Mit den obigen Definitionen wenden wir nun die Block-LR-Zerlegung aus Algorithmus 3.3 auf die Matrix (3.16) an.

Algorithmus 3.17 (erweiterte Block-LR-Zerlegung)**.**

```
1  for s = 0, ..., S
2      for t = 1, ..., T_s
3          for k = K_{s,t-1} + 1, ..., K_{s,t}
4              A_kk := LR(A_kk)
5              for m = k + 1, ..., K
6                  A_km = SOLVE(A_kk, A_km)
7                  for n = k + 1, ..., K
8                      A_nm = A_nm − A_nk A_km
```

Die Zeilen 1 bis 3 stimmen aufgrund der Definitionen von s, t und $K_{s,t}$ mit **for** $k = 1, \dots, K$ überein, somit entspricht dieser Algorithmus genau dem Algorithmus 3.3, bis auf die Zählweise über s, t und k.

Nach Lemma 2.7 und Korollar 2.9 sind viele Matrizen leer (also $A_{km} = 0$). Viele dieser Matrizen bleiben auch nach der Durchführung des Algorithmus 3.17 leer, so dass nur der Teil berechnet werden muss, der sich auch ändert. Im Folgenden wird untersucht, auf welche Berechnungen wir uns beschränken können.

Definition 3.18 (Operationsmengen)**.**
Die tatsächliche Operationsmenge für einen Cluster $t = 1, \dots, T_s$ in einem Schritt $s = 0, \dots, S$ wird definiert über

$$\mathcal{K}^{s,t} = \left\{ k \in \{K_{s-1} + 1, \dots, K\} \colon \pi_k \cap \mathcal{P}^{s,t} \neq \emptyset \right\} .$$

Weiterhin sei

$$\mathcal{K}^{s,t}_{\mathrm{LR}} = \{K_{s,t-1} + 1, \dots, K_{s,t}\}$$

die zu zerlegende Operationsmenge,

$$\mathcal{K}_{\text{SCH}}^{s,t} = \left\{k \in \{K_s + 1, \dots, K\} \colon \pi_k \cap \mathcal{P}^{s,t} \neq \emptyset\right\}$$

die zugehörige Schur-Komplement-Menge und

$$\mathcal{K}_{\Delta}^{s,t} = \left\{k \in \{K_{s-1} + 1, \dots, K\} \colon \pi_k \cap \mathcal{P}^{s,t} = \emptyset\right\}$$

das Komplement zu $\mathcal{K}^{s,t}$.
Mit $\mathcal{K} = \{1, \dots, K\}$ *wird die gesamte Operationsmenge bezeichnet.*

Lemma 3.19.
Für eine kombinierte Prozessormenge $\mathcal{P}^{s,t}$ *in Schritt* s*, welche sich aus den beiden kombinierten Prozessormengen* $\mathcal{P}^{s-1,t_1}$ *und* $\mathcal{P}^{s-1,t_2}$ *wie in (3.10) ergibt (das heißt es gilt* $\mathcal{P}^{s,t} = \mathcal{P}^{s-1,t_1} \cup \mathcal{P}^{s-1,t_2}$*) gilt für die Operationsmenge*

$$\mathcal{K}^{s,t} = \mathcal{K}_{\text{SCH}}^{s-1,t_1} \cup \mathcal{K}_{\text{SCH}}^{s-1,t_2}\,.$$

BEWEIS. Wir betrachten die Definitionen der Mengen:

$$\begin{aligned}
\mathcal{K}_{\text{SCH}}^{s-1,t_1} &= \{k \in \{K_{s-1} + 1, \dots, K\} \colon \pi_k \cap \mathcal{P}^{s-1,t_1} \neq \emptyset\} \\
\mathcal{K}_{\text{SCH}}^{s-1,t_2} &= \{k \in \{K_{s-1} + 1, \dots, K\} \colon \pi_k \cap \mathcal{P}^{s-1,t_2} \neq \emptyset\} \\
\mathcal{K}^{s,t} &= \{k \in \{K_{s-1} + 1, \dots, K\} \colon \pi_k \cap \mathcal{P}^{s,t} \neq \emptyset\}\,.
\end{aligned}$$

Somit gilt

$$\begin{aligned}
\mathcal{K}_{\text{SCH}}^{s-1,t_1} \cup \mathcal{K}_{\text{SCH}}^{s-1,t_2} &= \{k \in \{K_{s-1} + 1, \dots, K\} \colon \pi_k \cap \mathcal{P}^{s-1,t_1} \neq \emptyset\} \\
&\qquad \cup \{k \in \{K_{s-1} + 1, \dots, K\} \colon \pi_k \cap \mathcal{P}^{s-1,t_2} \neq \emptyset\} \\
&= \{k \in \{K_{s-1} + 1, \dots, K\} \colon \pi_k \cap \left(\mathcal{P}^{s-1,t_1} \cup \mathcal{P}^{s-1,t_2}\right) \neq \emptyset\} \\
&= \{k \in \{K_{s-1} + 1, \dots, K\} \colon \pi_k \cap \mathcal{P}^{s,t} \neq \emptyset\} = \mathcal{K}^{s,t}\,.
\end{aligned}$$

□

Bemerkung 3.20.
Es gilt $\Pi^{s,t} = \{\pi_k \colon k \in \mathcal{K}_{\text{LR}}^{s,t}\}$ (vergleiche Definition 3.14) und somit $\pi_k \subset \mathcal{P}^{s,t}$ für $k \in \mathcal{K}_{\text{LR}}^{s,t}$. Insbesondere gilt $\mathcal{K}_{\text{LR}}^{s,t} \subset \mathcal{K}^{s,t}$. Nach Bemerkung 3.12 gilt $\Pi^{s,t} \cap \Pi^{s,\tilde{t}} = \emptyset$ für $t \neq \tilde{t}$, daher kann $\mathcal{K}^{s,t}$ auch definiert werden über

$$\begin{aligned}
\mathcal{K}^{s,t} &= \left\{k \in \{K_{s,t-1} + 1, \dots, K\} \colon \pi_k \cap \mathcal{P}^{s,t} \neq \emptyset\right\} \\
&= \mathcal{K}_{\text{LR}}^{s,t} \cup \mathcal{K}_{\text{SCH}}^{s,t}\,.
\end{aligned}$$

Damit gilt für jeden Schritt $s = 0, \dots, S$

$$\mathcal{K}_{\text{LR}}^{s,t} \cap \mathcal{K}_{\text{LR}}^{s,\tilde{t}} = \emptyset \quad \text{für } t \neq \tilde{t}\,.$$

Wir können $\mathcal{K}_{\text{LR}}^{s,t}$ und $\Pi^{s,t}$ also miteinander identifizieren, somit ergibt sich auch eine aufsteigende Nummerierung der Elemente in $\mathcal{K}_{\text{LR}}^{s,t}$, für größer werdende t beziehungsweise s.

Die Hauptaussage, welche Matrizen in der Block-LR-Zerlegung berechnet werden müssen, folgt nun aus folgendem Satz 3.21.

Satz 3.21.
In Algorithmus 3.17 gilt $A_{km} = 0$ *und* $A_{mk} = 0$ *während der Durchführung von Schritt* s *für* $k \in \mathcal{K}_{LR}^{s,t}$, $m \in \mathcal{K}_{\Delta}^{s,t}$. *Weiterhin werden die Matrizen* A_{nm}*, für* $n \in \mathcal{K}_{\Delta}^{s,t}$ *oder* $m \in \mathcal{K}_{\Delta}^{s,t}$ *in Schritt* s *für* $k \in \mathcal{K}_{LR}^{s,t}$ *nicht geändert.*

BEWEIS. Wir zeigen zuerst die zweite Behauptung unter Annahme der ersten Behauptung, dass also A_{nm} im Schritt s nicht geändert wird, wenn $n \in \mathcal{K}_{\Delta}^{s,t}$ oder $m \in \mathcal{K}_{\Delta}^{s,t}$ und $k \in \mathcal{K}_{\text{LR}}^{s,t}$. Die erste Behauptung besagt, dass $A_{km} = 0$ und $A_{mk} = 0$ für $m \in \mathcal{K}_{\Delta}^{s,t}$.

In Algorithmus 3.17 existieren drei Operationen zur Änderung von Matrizen:

- In Zeile 4 wird eine LR-Zerlegung einer Diagonalmatrix berechnet ($A_{kk} := \mathrm{LR}(A_{kk})$), es gilt allerdings $k \in \mathcal{K}_{\mathrm{LR}}^{s,t} \subset \mathcal{K}^{s,t}$, so dass die Voraussetzung $k \in \mathcal{K}_{\Delta}^{s,t}$ nicht gegeben ist.
- Die zweite Operation lautet $A_{km} = \textsc{Solve}(A_{kk}, A_{km})$ (Zeile 6), wobei die Lösungsroutine über die eben betrachtete LR-Zerlegung von A_{kk} geschieht. Wenn allerdings $m \in \mathcal{K}_{\Delta}^{s,t}$, so ist $A_{km} = 0$ und für die Operation gilt ebenso $A_{kk}^{-1} A_{km} = 0$. Diese Nullmatrix wird also nicht geändert.
- Die dritte Operation findet sich in Zeile 8 mit $A_{nm} := A_{nm} - A_{nk} A_{km}$. Um die Matrix A_{nm} zu ändern, müssen sowohl $A_{nk} \neq 0$, als auch $A_{km} \neq 0$ sein. Falls nun jedoch $n \in \mathcal{K}_{\Delta}^{s,t}$, so gilt $A_{nk} = 0$ (bzw. falls $m \in \mathcal{K}_{\Delta}^{s,t}$, so gilt $A_{km} = 0$), somit ändert eine Durchführung dieser Operation die Matrix A_{nm} nicht.

Insgesamt werden Matrizen A_{nm} für $n, m \in \mathcal{K}_{\Delta}^{s,t}$ nicht geändert, wenn $A_{km} = 0$ und $A_{mk} = 0$ für $m \in \mathcal{K}_{\Delta}^{s,t}$.

Für diese erste Behauptung zeigen wir per Induktion, dass $A_{km} = 0$ und $A_{mk} = 0$ zu Beginn jeden Schrittes s für $k \in \mathcal{K}_{\mathrm{LR}}^{s,t}$, $m \in \mathcal{K}_{\Delta}^{s,t}$. Es ist offensichtlich, dass sich diese Matrizen nicht mehr ändern, nachdem der $k-$te Schritt im Algorithmus durchgeführt wurde, da alle folgenden Operationen für Matrizen A_{nm} mit $n, m > k$ gelten.

Sei $s = 0$. Nach (3.9) gilt $\mathcal{P}^{0,t} = \{p_t\}$, somit ergeben sich für $t = 1, \ldots, T_0$

$$\begin{aligned}
\mathcal{K}_{\mathrm{LR}}^{0,t} &= \{t\} \\
\mathcal{K}^{0,t} &= \{k \geq t \colon \pi_k \cap \{t\} \neq \emptyset\} \\
\mathcal{K}_{\Delta}^{0,t} &= \{k \colon \pi_k \cap \{t\} = \emptyset\}\,.
\end{aligned}$$

Da nur ein Element in $\mathcal{K}_{\mathrm{LR}}^{0,t}$ existiert, können wir dieses Element $k \in \mathcal{K}_{\mathrm{LR}}^{0,t}$ mit $k = t$ identifizieren. Sei $m \in \mathcal{K}_{\Delta}^{s,t}$, also insbesondere $\pi_m \cap \{t\} = \emptyset$. Wegen $\pi_k = \{t\}$ gilt nach Korollar 2.9, dass $A_{km} = 0$ und $A_{mk} = 0$.

Für den Induktionsschritt betrachten wir nun $s > 0$. Sei $k \in \mathcal{K}_{\mathrm{LR}}^{s,t}, m \in \mathcal{K}_{\Delta}^{s,t}$. Damit gilt $\pi_k \subset \mathcal{P}^{s,t}$ und $\pi_m \cap \mathcal{P}^{s,t} = \emptyset$, also insbesondere auch $\pi_m \cap \pi_k = \emptyset$. Zur Übersichtlichkeit betrachten wir hier nur die Matrix A_{km}, für die Matrix A_{mk} gelten die gleichen Argumente. Es gilt $A_{km} = 0$ am Anfang des Algorithmus nach Korollar 2.9. Wir zeigen, dass A_{km} im bisherigen Verlauf bis zum Anfang von Schritt s nicht geändert wurde. Dazu reicht es, die Matrixoperation aus Zeile 8 zu betrachten, die sich hier beschreiben lässt als $A_{km} = A_{km} - A_{kl} A_{lm}$ für $l < k$. Die beiden anderen Operationen können nur Matrizen in der jeweiligen Zeile $l < k$ geändert haben. Eine Änderung tritt nicht auf, wenn $A_{kl} = 0$ oder $A_{lm} = 0$, daher zeigen wir, dass aus $A_{kl} \neq 0$ folgt, dass $A_{lm} = 0$ für $l < k$.
Sei dazu $l \in \mathcal{K}_{\mathrm{LR}}^{\sigma,\tau}$ mit $\sigma < s$ und $\tau \in \{1, \ldots, T_\sigma\}$ so gewählt, dass $A_{kl} \neq 0$ (andererseits findet keine Änderung statt). Damit gilt $k \in \mathcal{K}^{\sigma,\tau}$ nach Induktionsannahme und somit $\pi_k \cap \mathcal{P}^{\sigma,\tau} \neq \emptyset$. Es soll $A_{lm} = 0$ gezeigt werden. Wir zeigen dies über Widerspruchsannahme. Sei also $A_{lm} \neq 0$. Dann gilt nach Induktionsannahme $m \in \mathcal{K}^{\sigma,\tau}$ und somit $\pi_m \cap \mathcal{P}^{\sigma,\tau} \neq \emptyset$. Nach (3.10) gilt für die Definition der kombinierten Prozessormengen $\mathcal{P}^{s,t} = \mathcal{P}^{s-1,\tau_1} \cup \mathcal{P}^{s-1,\tau_2}$ für alle $s > 0$ mit geeigneten $\tau_1, \tau_2 \in \{1, \ldots, T_{s-1}\}$. Da weiterhin $\mathcal{P}^{s,t_1} \cap \mathcal{P}^{s,t_2} = \emptyset$ (das heißt, zwei Prozessormengen innerhalb eines Schrittes sind disjunkt) gilt insbesondere $\mathcal{P}^{\sigma,\tau} \subset \mathcal{P}^{s,t}$, da $\pi_k \cap \mathcal{P}^{\sigma,\tau} \neq \emptyset$ und $\pi_k \subset \mathcal{P}^{s,t}$. Damit gilt $\pi_m \cap \mathcal{P}^{s,t} \neq \emptyset$, dies ist ein Widerspruch zu $m \in \mathcal{K}_{\Delta}^{s,t}$. □

Somit können für $k \in \mathcal{K}_{\mathrm{LR}}^{s,t}$ alle Operationen für $m \in \mathcal{K}_{\Delta}^{s,t}$ ignoriert werden. Insgesamt kann Algorithmus 3.17 vereinfacht werden zu

Algorithmus 3.22 (reduzierte Block-LR-Zerlegung)**.**

```
1  for s = 0, ..., S
2      for t = 1, ..., T_s
3          for k ∈ K_LR^{s,t}
4              A_kk := LR(A_kk)
5              for m ∈ K^{s,t}, m > k
6                  A_km = SOLVE(A_kk, A_km)
7                  for n ∈ K^{s,t}, n > k
8                      A_nm = A_nm − A_nk A_km
```

3.6.2. Erste Parallelisierung über Nested Dissection.

Im Folgenden setzen wir $p_t = t$ (und zählen dabei von $p = 1, \ldots, P$), da die eigentliche Nummerierung nebensächlich ist und somit ein Index zur Übersichtlichkeit gespart werden kann.

In Schritt s gilt für verschiedene Cluster $t \neq \tilde{t}$, dass $\mathcal{K}_{\mathrm{LR}}^{s,t} \subset \mathcal{K}_{\Delta}^{s,\tilde{t}}$. Wir bezeichnen mit $A_{nm}^{(s)}$ eine Blockmatrix in A zu Beginn von Schritt s. Die Blockmatrizen

$$Z^{(s),t} = \left(A_{nm}^{(s)}\right)_{n,m \in \mathcal{K}_{\mathrm{LR}}^{s,t}}$$

bilden in Schritt s somit eine Diagonalstruktur im oberen linken Teil. Zur Verdeutlichung betrachten wir die Matrix A zu Beginn in Schritt s. Dabei ignorieren wir die schon zerlegten Blockmatrizen, so dass die restliche Matrix (also das Schur-Komplement nach Schritt $s-1$) dargestellt werden kann als

$$A^{(s)} = \left(\begin{array}{ccc|c} Z^{(s),1} & 0 & 0 & R^{(s),1} \\ 0 & \ddots & 0 & \vdots \\ 0 & 0 & Z^{(s),T_s} & R^{(s),T_s} \\ \hline L^{(s),1} & \ldots & L^{(s),T_s} & S^{(s)} \end{array}\right), \tag{3.17}$$

mit

$$\begin{aligned} R^{(s),t} &= (A_{nm})_{n \in \mathcal{K}_{\mathrm{LR}}^{s,t},\ m = K_s+1,\ldots,K} \\ L^{(s),t} &= (A_{nm})_{n = K_s+1,\ldots,K,\ m \in \mathcal{K}_{\mathrm{LR}}^{s,t}} \\ S^{(s),t} &= (A_{nm})_{n,m = K_s+1,\ldots,K}\,. \end{aligned}$$

Für $s = 0$ ergibt obige Definition die globale Matrix A vor Beginn der Zerlegung. In Schritt s werden nun die Elemente $k \in \mathcal{K}_{\mathrm{LR}}^{s,t}$ für $t = 0, \ldots, T_s$ zerlegt. Diese Zerlegungen können komplett parallel durchgeführt werden (vergleiche Kapitel 3.4) und wir erhalten ein additives Schur-Komplement.

Bisher wurde die Block-LR-Zerlegung global betrachtet, in dem Sinne, dass alle Blockmatrizen komplett gegeben sind. In unserem Fall ist die globale Matrix jedoch additiv verteilt, so dass während der Zerlegung additive Matrizen zusammengefasst werden müssen. Dies geschieht später über Kommunikationsroutinen, wenn jeweils zwei Prozessormengen zusammengefasst werden. Die additiven Matrizen befinden sich dabei im jeweiligen Schur-Komplement $S^{(s)}$.

Weiterhin können die Matrizen $R^{(s),t}$ und $L^{(s),t}$ reduziert werden, indem die Nullblöcke gestrichen werden. Dafür definieren wir Restriktionsmatrizen $P^{(s),t}$, die über die Elemente in $\mathcal{K}_{\mathrm{SCH}}^{s,t}$ gegeben sind.

Definition 3.23 (Restriktionsmatrix)**.**
Die Mengen

$$\mathcal{M} = \{m_i \colon m_i \in \mathcal{K}_{\text{SCH}}^{s,t}\},$$
$$\mathcal{N} = \{n_i \colon n_i \in \{K_s + 1, \ldots, K\}\}$$

seien aufsteigend nummeriert (das heißt $m_i > m_j$ für $i > j$) und es gelte $M = |\mathcal{M}|$, $N = |\mathcal{N}|$. Die $M \times N$ Blockmatrix

$$P^{(s),t} = \left(P_{ij}^{(s),t}\right)_{m_i \in \mathcal{M}, n_j \in \mathcal{N}}$$

ist definiert über

$$P_{ij}^{(s),t} = \begin{cases} I & m_i = n_j \\ 0 & \text{sonst} \end{cases}$$

wobei I die Einheitsmatrix der entsprechenden Dimension $N_{m_i} \times N_{n_j}$ (aus Definition 2.4) bezeichnet und 0 die Nullmatrix.

Beispiel 3.24. Für $\mathcal{M} = \{5, 7, 10\}$ und $\mathcal{N} = \{5, 6, 7, 8, 9, 10, 11\}$ hat die Restriktionsmatrix P die Form

$$P = \begin{pmatrix} I & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & I & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & I & 0 \end{pmatrix}.$$

Mit Hilfe dieser Restriktionen definieren wir die lokalen Blockmatrizen

$$R_{\text{loc}}^{(s),t} = R^{(s),t}(P^{(s),t})^T = (A_{nm}^{(s)})_{n \in \mathcal{K}_{\text{LR}}^{s,t}, m \in \mathcal{K}_{\text{SCH}}^{s,t}}$$
$$L_{\text{loc}}^{(s),t} = P^{(s),t}L^{(s),t} = (A_{nm}^{(s)})_{n \in \mathcal{K}_{\text{SCH}}^{s,t}, m \in \mathcal{K}_{\text{LR}}^{s,t}}.$$

Da durch diese Restriktionen nur Nullspalten (in $R^{(s),t}$) bzw. Nullzeilen (in $L^{(s),t}$) gestrichen werden, gilt außerdem

$$R^{(s),t} = R_{\text{loc}}^{(s),t}P^{(s),t}$$
$$L^{(s),t} = (P^{(s),t})^T L_{\text{loc}}^{(s),t}.$$

Die globalen Matrixblöcke A_{km} (und damit die globale Matrix A selbst) sei nun auf die Prozessoren verteilt wie in Definition 2.5. Wir betrachten nun für Schritt s insbesondere die Blockdiagonalmatrizen $Z^{(s),t}$, sowie die zugehörigen Matrizen $R^{(s),t}$ und $L^{(s),t}$. Nach Bemerkung 3.7 müssen diese Matrizen global auf einem Prozessor bzw. Cluster vorhanden sein, während das zu berechnende Schur-Komplement additiv sein darf. Dann kann das Schur-Komplement

$$\begin{aligned} A^{(s+1)} &= S^{(s)} - \sum_{t=1}^{T_s} L^{(s),t}(Z^{(s),t})^{-1}R^{(s),t} \\ &= S^{(s)} - \sum_{t=1}^{T_s} (P^{(s),t})^T L_{\text{loc}}^{(s),t}(Z^{(s),t})^{-1}R_{\text{loc}}^{(s),t}P^{(s),t} \end{aligned}$$

parallel auf T_s Prozessoren bzw. Clustern berechnet werden. Damit $L^{(s),t}$, $Z^{(s),t}$ und $R^{(s),t}$ auf dem jeweiligen Cluster t in Schritt s global vorhanden sind, treffen wir folgende Vereinbarung.

Vereinbarung 3.25. Bei der Zusammenführung zweier Prozessor-Sets $\mathcal{P}^{s-1,t_1}$, $\mathcal{P}^{s-1,t_2}$ zu $\mathcal{P}^{s,t} = \mathcal{P}^{s-1,t_1} \cup \mathcal{P}^{s-1,t_2}$ (nach (3.10)) werden alle Matrizen additiv über

$$A_{nm}^{(s),t} = A_{nm}^{(s-1),t_1} + A_{nm}^{(s-1),t_2}$$

zusammengefasst. Dies geschieht am Anfang jedes Schrittes s. Für Schritt $s = 0$ setzen wir

$$A_{nm}^{(0),p} = A_{nm}^{p} \,.$$

Somit können wir lokale Matrizen im Schritt s auf einem Cluster t definieren über

$$A_{\text{loc}}^{(s),t} = \begin{pmatrix} Z^{(s),t} & R_{\text{loc}}^{(s),t} \\ L_{\text{loc}}^{(s),t} & S_{\text{loc}}^{(s),t} \end{pmatrix} = \left(A_{nm}^{(s),t}\right)_{n,m\in\mathcal{K}^{s,t}} , \tag{3.18}$$

wobei

$$\begin{aligned} Z^{(s),t} &= (A_{nm}^{(s),t})_{n,m\in\mathcal{K}_{\text{LR}}^{s,t}} \\ R_{\text{loc}}^{(s),t} &= (A_{nm}^{(s),t})_{n\in\mathcal{K}_{\text{LR}}^{s,t},m\in\mathcal{K}_{\text{SCH}}^{s,t}} \\ L_{\text{loc}}^{(s),t} &= (A_{nm}^{(s),t})_{n\in\mathcal{K}_{\text{SCH}}^{s,t},m\in\mathcal{K}_{\text{LR}}^{s,t}} \\ S_{\text{loc}}^{(s),t} &= (A_{nm}^{(s),t})_{n,m\in\mathcal{K}_{\text{SCH}}^{s,t}} \,. \end{aligned} \tag{3.19}$$

Dabei ist $S_{\text{loc}}^{(s),t}$ das lokale Schur-Komplement, so dass das globale Schur-Komplement über

$$S^{(s)} = \sum_{t=1}^{T_s} S^{(s),t} \quad \text{mit} \quad S^{(s),t} = (P^{(s),t})^T S_{\text{loc}}^{(s),t} P^{(s),t}$$

berechnet werden kann (vergleiche (3.17)).

Lemma 3.26. *Die lokalen Matrizen $Z^{(s),t}$, $R_{loc}^{(s),t}$, $L_{loc}^{(s),t}$ sind global vorhanden.*

Beweis. Wir betrachten zuerst $s = 0$. Hier kann ein Cluster t mit einem Prozessor p identifiziert werden. Wegen Lemma 2.7 sind alle Zeilen- und Spalten-Blöcke A_{km} von $A^{(0),p}$ mit $p \neq \pi_k$ oder $p \neq \pi_m$ nicht besetzt, somit betrachten wir nur diejenigen lokalen Blöcke, in denen Einträge vorhanden sind. Da $\mathcal{P}^{0,t} = \{t\}$, können die lokalen Matrizen auf Prozessor p beschrieben werden als

$$A^p = (A_{nm}^p)_{n,m\in\mathcal{K}^{0,p}} \,.$$

Weiterhin gilt $\mathcal{K}_{\text{LR}}^{0,p} = \{p\}$ (und für alle anderen $q \in \mathcal{K}^{0,p}$ gilt sogar $q > P$, da $\mathcal{K}_{\text{LR}}^{0,p} \subset \mathcal{K}_{\Delta}^{0,t}$ für $p \neq t$, $t \in \{1,\ldots,P\}$), somit wird die jeweils linke obere Blockmatrix auf Prozessor p über A_{pp}^p beschrieben. Diese Blockmatrizen sind auf keinem anderen Prozessor zu finden (das heißt $A_{pp}^q = 0$ für $p \neq q$), somit sind alle Blockmatrizen A_{pp} $(p = 1,\ldots,P)$ nicht-additiv (also global) vorhanden. Global betrachtet hat die Matrix die Form aus (3.17) für $s = 0$ und $Z^{(0),p} = A_{pp}$, also

$$A^{(0)} = \left(\begin{array}{ccc|c} A_{11} & 0 & 0 & R^{(0),1} \\ 0 & \ddots & 0 & \vdots \\ 0 & 0 & A_{PP} & R^{(0),P} \\ \hline L^{(0),1} & \ldots & L^{(0),P} & S^{(0)} \end{array}\right) .$$

Zusätzlich sind alle Matrizen $L^{(0),p}$ und $R^{(0),p}$ global auf Prozessor p vorhanden, da für $k \in \mathcal{K}_{\text{SCH}}^{0,p}$ und $p \in \mathcal{K}_{\text{LR}}^{0,p}$ gilt $\pi_k \cap \pi_p = \{p\}$. Damit ist nur $S^{(0)}$ additiv verteilt

mit

$$S^{(0)} = \left(A_{nm}\right)_{n,m\in\{P+1,\dots,K\}} = \sum_{p=1}^{P} \left(A^p_{nm}\right)_{n,m\in\{P+1\dots,K\}} .$$

Mit Hilfe der Restriktionen $P^{(0),p}$ gilt für eine lokale Matrix auf Prozessor p vor Schritt $s = 0$ somit

$$A^{(0),p}_{\text{loc}} = \left(A^p_{nm}\right)_{p\in\pi_n\cap\pi_m} = \begin{pmatrix} A_{pp} & R^{(0),p}_{\text{loc}} \\ L^{(0),p}_{\text{loc}} & S^{(0),p}_{\text{loc}} \end{pmatrix} ,$$

mit $R^{(0),p}_{\text{loc}} = R^{(0),p}(P^{(0),p})^T$ und $L^{(0),p}_{\text{loc}} = P^{(0),p}L^{(0),p}$. Für $S^{(0),p}_{\text{loc}}$ gilt

$$S^{(0),p}_{\text{loc}} = \left(A^p_{nm}\right)_{n,m\in\mathcal{K}^{0,p}_{\text{SCH}}} .$$

Alle anderen hier nicht betrachteten Matrizen sind auf Prozessor p nicht besetzt, (das heißt $A^p_{nm} = 0$) da diese über A^p_{nm} mit $p \notin \pi_n \cap \pi_m$ beschrieben werden.

Das lokale Schur-Komplement auf Prozessor p kann berechnet werden über

$$\hat{S}^{(0),p}_{\text{loc}} = S^{(0),p}_{\text{loc}} - L^{(0),p}_{\text{loc}} A^{-1}_{pp} R^{(0),p}_{\text{loc}} ,$$

was zum globalen Schur-Komplement

$$\begin{aligned} A^{(1)} = \hat{S}^{(0)} &= S^{(0)} - \sum_{t=1}^{T_0} L^{(0),p}(Z^{(0),p})^{-1}R^{(0),p} \\ &= S^{(0)} - \sum_{p=1}^{P} (P^{(0),p})^T L^{(0),p}_{\text{loc}} A^{-1}_{pp} R^{(0),p}_{\text{loc}} P^{(0),p} \\ &= \sum_{p=1}^{P} (P^{(0),p})^T \hat{S}^{(0),p}_{\text{loc}} P^{(0),p} \end{aligned} \tag{3.20}$$

führt.

Induktiv folgt nun, dass $Z^{(s),t}$, $R^{(s),t}_{\text{loc}}$ und $L^{(s),t}_{\text{loc}}$ global vorhanden sind, da auf allen anderen Clustern $\tilde{t} \neq t$ die zugehörigen Matrizen nicht belegt sein können:

Der Induktionsanfang für $s = 0$ ist oben gezeigt.

Wir betrachten die Prozessormenge $\mathcal{P}^{s,t}$ in Schritt $s > 0$ auf einem Cluster t. Diese wird aus zwei Prozessormengen $\mathcal{P}^{s-1,t_1}$ und $\mathcal{P}^{s-1,t_2}$ erstellt. Wegen Lemma 3.19 und der Vereinbarung 3.25 wird die lokale Matrix $A^{(s),t}_{\text{loc}}$ aus (3.18) genau aus den beiden Schur-Matrizen $S^{(s-1),t_1}_{\text{loc}}$ und $S^{(s-1),t_2}_{\text{loc}}$ additiv erstellt. Für alle Matrizen $A^{(s),t}_{nm}$ aus der lokalen Matrix $A^{(s),t}_{\text{loc}}$ gilt $\pi_n \cap \mathcal{P}^{s,t} \neq \emptyset \neq \pi_m \cap \mathcal{P}^{s,t}$. Andererseits gilt $A^{(s),t}_{nm} = 0$ für $\pi_n \cap \mathcal{P}^{s,t} = \emptyset$ oder $\pi_m \cap \mathcal{P}^{s,t} = \emptyset$. Wegen Bemerkung 3.20 gilt $\pi_k \subset \mathcal{P}^{s,t}$ für $k \in \mathcal{K}^{s,t}_{\text{LR}}$. Da die Blockmatrizen $A^{(s),t}_{nm}$ in $Z^{(s),t}$, $R^{(s),t}_{\text{loc}}$ und $L^{(s),t}_{\text{loc}}$ in jedem Fall zu einem Index $k \in \mathcal{K}^{s,t}_{\text{LR}}$ gehören, sind diese wegen $\pi_k \cap \mathcal{P}^{s,\tilde{t}} = \emptyset$ für $\tilde{t} \neq t$ global vorhanden. □

Somit lässt sich Algorithmus 3.22 mit Hilfe der additiven Zerlegung der Teilmatrizen wie folgt parallelisieren, wobei die Matrizen jetzt clusterweise betrachtet werden und $A^{(0),p}_{nm} = A^p_{nm}$ $(p = 1, \dots, P = T_0)$ nach Vereinbarung 3.25 gesetzt wird:

Algorithmus 3.27 (Block-LR-Zerlegung mit Nested Dissection (parallel))**.**

1 **for** $s = 0, \dots, S$
2 **for** $t = 1, \dots, T_s$ **(parallel)**
3 **if** $(s > 0)$
4 **setze** $t_1, t_2 \in \{1, \dots, T_{s-1}\}$ **so dass** $\mathcal{P}^{s,t} = \mathcal{P}^{s-1,t_1} \cup \mathcal{P}^{s-1,t_2}$
5 **for** $n, m \in \mathcal{K}^{s,t}$
6 $A_{nm}^{(s),t} = A_{nm}^{(s-1),t_1} + A_{nm}^{(s-1),t_2}$
7 **for** $k \in \mathcal{K}_{\mathrm{LR}}^{s,t}$
8 $A_{kk}^{(s),t} := \mathrm{LR}(A_{kk}^{(s),t})$
9 **for** $m \in \mathcal{K}^{s,t}, m > k$
10 $A_{km}^{(s),t} = \textsc{Solve}(A_{kk}^{(s),t}, A_{km}^{(s),t})$
11 **for** $n \in \mathcal{K}^{s,t}, n > k$
12 $A_{nm}^{(s),t} = A_{nm}^{(s),t} - A_{nk}^{(s),t} A_{km}^{(s),t}$

Die globalen Matrizen sind in diesem Fall auf verschiedenen Clustern gegeben.

3.6.3. Zweite Parallelisierung über parallele Matrixverteilung. Wie wir in (3.18) gesehen haben, können in jedem Schritt s für jeden Cluster t lokale Matrizen

$$A_{\mathrm{loc}}^{(s),t} = \begin{pmatrix} Z^{(s),t} & R_{\mathrm{loc}}^{(s),t} \\ L_{\mathrm{loc}}^{(s),t} & S_{\mathrm{loc}}^{(s),t} \end{pmatrix}$$

definiert werden, auf denen lokal eine Block-LR-Zerlegung zur Berechnung des Schur-Komplements von $Z^{(s),t}$ angewandt werden kann (Schritte 7 bis 12 in Algorithmus 3.27). Die Definition erfolgt dabei aus den Interfaces π_k, wobei sich die Matrix $Z^{(s),t}$ über $k \in \mathcal{K}_{\mathrm{LR}}^{s,t}$ definiert. Im Folgenden betrachten wir alle Matrizen aus (3.19) im Ganzen, das heißt sie sollen nicht mehr blockweise über $k \in \mathcal{K}_{\mathrm{LR}}^{s,t}$ bzw. $k \in \mathcal{K}_{\mathrm{Sch}}^{s,t}$ unterteilt sein. Dafür erhalten sie eine neue Zerlegung auf die Prozessoren. Zur Berechnung des Schur-Komplements $S_{\mathrm{loc}}^{(s),t} - L_{\mathrm{loc}}^{(s),t}(Z^{(s),t})^{-1} R_{\mathrm{loc}}^{(s),t}$ wird nun eine Block-Zerlegung in Abhängigkeit der Anzahl vorhandener Prozessoren im Cluster t erstellt, wie sie in Kapitel 3.3 eingeführt wurde.

Dazu betrachten wir die Dimensionen der Matrizen:

$$\begin{aligned} (A_{nm}^{(s),t})_{n,m \in \mathcal{K}_{\mathrm{LR}}^{s,t}} &= Z^{(s),t} \in \mathbb{R}^{N^{s,t} \times N^{s,t}} \\ (A_{nm}^{(s),t})_{n \in \mathcal{K}_{\mathrm{LR}}^{s,t}, m \in \mathcal{K}_{\mathrm{Sch}}^{s,t}} &= R_{\mathrm{loc}}^{(s),t} \in \mathbb{R}^{N^{s,t} \times M^{s,t}} \\ (A_{nm}^{(s),t})_{n \in \mathcal{K}_{\mathrm{Sch}}^{s,t}, m \in \mathcal{K}_{\mathrm{LR}}^{s,t}} &= L_{\mathrm{loc}}^{(s),t} \in \mathbb{R}^{M^{s,t} \times N^{s,t}} \\ (A_{nm}^{(s),t})_{n,m \in \mathcal{K}_{\mathrm{Sch}}^{s,t}} &= S_{\mathrm{loc}}^{(s),t} \in \mathbb{R}^{M^{s,t} \times M^{s,t}} \end{aligned} \tag{3.21}$$

mit $N^{s,t} = \sum_{k \in \mathcal{K}_{\mathrm{LR}}^{s,t}} |\mathcal{I}_k|$, $M^{s,t} = \sum_{k \in \mathcal{K}_{\mathrm{Sch}}^{s,t}} |\mathcal{I}_k|$, sowie der zugehörigen Prozessormenge

$$\mathcal{P}^{s,t} = \{p_1^{s,t}, \dots, p_{Q^{s,t}}^{s,t}\}$$

mit $Q^{s,t} = |\mathcal{P}^{s,t}|$. Die Matrizen werden nun spaltenweise auf die Prozessoren verteilt, so dass jeder Prozessor $p_q^{s,t} \in \mathcal{P}^{s,t}$

$$N_{p_q^{s,t}} = \begin{cases} \lfloor N^{s,t}/Q^{s,t} \rfloor + 1 & q \leq N^{s,t} \mod Q^{s,t} \\ \lfloor N^{s,t}/Q^{s,t} \rfloor & q > N^{s,t} \mod Q^{s,t} \end{cases} \tag{3.22}$$

Spalten der Matrizen $Z^{(s),t}$ und $L_{\text{loc}}^{(s),t}$, sowie

$$M_{p_q^{s,t}} = \begin{cases} \lfloor M^{s,t}/Q^{s,t} \rfloor + 1 & q \le M^{s,t} \mod Q^{s,t} \\ \lfloor M^{s,t}/Q^{s,t} \rfloor & q > M^{s,t} \mod Q^{s,t} \end{cases} \tag{3.23}$$

Spalten der Matrizen $R_{\text{loc}}^{(s),t}$ und $S_{\text{loc}}^{(s),t}$ besitzt. Für die lokale Matrix ergibt sich also folgendes Bild:

$$A_{\text{loc}}^{(s),t} = \left(\begin{array}{c|c|c|c|c} Z_1^{(s),t} & \cdots & Z_{Q^{s,t}}^{(s),t} & R_1^{(s),t} & \cdots & R_{Q^{s,t}}^{(s),t} \\ \hline L_1^{(s),t} & \cdots & L_{Q^{s,t}}^{(s),t} & S_1^{(s),t} & \cdots & S_{Q^{s,t}}^{(s),t} \end{array} \right). \tag{3.24}$$

Auf $Z^{(s),t}$ wird nun die parallele Block-LR-Zerlegung wie in Algorithmus 3.4 angewandt und das Schur-Komplement in $S_{\text{loc}}^{(s),t}$ berechnet. Dafür setzen wir wie in Kapitel 3.3 (dort wurden diese Variablen mit J bezeichnet)

$$\begin{aligned} N_0^{s,t} &= 0, \\ N_q^{s,t} &= \sum_{n=1}^{q} N_{p_n^{s,t}}, \qquad 1 \le q \le Q^{s,t} \end{aligned} \tag{3.25}$$

Damit ergibt sich als Erweiterung zu Algorithmus 3.4 der folgende Algorithmus.

Algorithmus 3.28 (lokale Block-LR-Zerlegung mit verteilten Spalten)**.**

1 **for** $q = 1, \ldots, Q^{s,t}$
2 **setze** $i = N_{q-1}^{s,t} + 1, \quad j = N_q^{s,t}$
3 **auf** $p_q^{s,t}$**:**
4 $Z_q^{(s),t}[i:j] := \text{LR}(Z_q^{(s),t}[i:j])$
5 SENDE $\left[p_q^{s,t} \to p_r^{s,t}\right] \left(Z_q^{(s),t}\right)$ **für** $p_r^{s,t} \in \mathcal{P}^{s,t}$
6 SENDE $\left[p_q^{s,t} \to p_r^{s,t}\right] \left(L_q^{(s),t}\right)$ **für** $p_r^{s,t} \in \mathcal{P}^{s,t}$
7 **auf** $p_r^{s,t} \in \mathcal{P}^{s,t}$**:** EMPFANGE $\left[p_r^{s,t} \leftarrow p_q^{s,t}\right] \left(Z_q^{(s),t}\right)$
8 **auf** $p_r^{s,t} \in \mathcal{P}^{s,t}$**:** EMPFANGE $\left[p_r^{s,t} \leftarrow p_q^{s,t}\right] \left(L_q^{(s),t}\right)$
9 **auf** $p_r^{s,t} \in \mathcal{P}^{s,t}$ **mit** $r > q$ **(parallel):**
10 $Z_r^{(s),t}[i:j] := \text{SOLVE}(Z_q^{(s),t}[i:j], Z_r^{(s),t}[i:j])$
11 $Z_r^{(s),t}[j+1{:}N^{s,t}] := Z_r^{(s),t}[j+1{:}N^{s,t}] - Z_q^{(s),t}[j+1{:}N^{s,t}] Z_r^{(s),t}[i{:}j]$
12 $L_r^{(s),t} := L_r^{(s),t} - L_q^{(s),t} Z_r^{s,t}[i:j]$
13 **auf** $p_r^{s,t} \in \mathcal{P}^{s,t}$ **(parallel):**
14 $R_r^{(s),t}[i:j] := \text{SOLVE}(Z_q^{(s),t}[i:j], R_r^{(s),t}[i:j])$
15 $R_r^{(s),t}[j+1{:}N^{s,t}] := R_r^{(s),t}[j+1{:}N^{s,t}] - Z_q^{(s),t}[m\,(+)\,1{:}N^{s,t}] R_r^{(s),t}[i{:}j]$
16 $S_r^{(s),t} := S_r^{(s),t} - L_q^{(s),t} R_r^{(s),t}[i:j]$

Im Vergleich zu Algorithmus 3.4 müssen hier die Matrizen $L_{\text{loc}}^{(s),t}$, $R_{\text{loc}}^{(s),t}$ und $S_{\text{loc}}^{(s),t}$ zusätzlich berücksichtigt werden, wobei die LR-Zerlegung nur für $Z^{(s),t}$ berechnet wird.

3.6.4. Zusammenführung der beiden Parallelisierungen.

Algorithmus 3.28 kann nun in Algorithmus 3.27 anstelle den Zeilen 7 bis 12 benutzt werden. Dabei werden die Matrixblöcke A_{nm} aufgegeben und als Gesamtmatrix wie in (3.24) betrachtet. Die Zusammenfassung der beiden Schur-Komplemente aus Schritt $s-1$ in den Zeilen 4 bis 6 geschieht im finalen Algorithmus im Prinzip genauso, es muss hier nur beachtet werden, dass die Matrixblöcke $A_{nm}^{(s-1),t}$ wegen der gleichmäßigen Spaltenzerlegung auf verschiedene Prozessoren verteilt sein können und keine Ähnlichkeit mit der Zerlegung der Matrixblöcke $A_{nm}^{(s),t}$ im folgenden Schritt haben müssen.

Dies ist jedoch nur eine programmiertechnische Einschränkung, letztendlich können aus der spaltenweise Zerlegung der Matrix $A_{\text{loc}}^{(s),t}$ die Blockmatrizen $A_{nm}^{(s),t}$ wieder zusammengesetzt werden und somit für den nächsten Schritt zusammengefasst werden. Im Programm selbst wird dies durch direkte Kommunikation zwischen den betroffenen Prozessoren geschehen. Dies ist hier recht einfach möglich, da in jedem Fall jeweils die gesamte Spalte auf einem Prozessor zu finden ist. Es genügt also, jede Spalte aus dem Schur-Komplement $S_{\text{loc}}^{(s-1),t_1}$ und $S_{\text{loc}}^{(s-1),t_2}$ einzeln zu betrachten und herauszufinden, auf welchem Prozessor diese Spalte in der neuen Matrix $A_{\text{loc}}^{(s),t}$ zu finden ist.

Insgesamt ergibt sich der finale Algorithmus 3.30 auf Seite **??** zur Berechnung der parallelen Block-LR-Zerlegung für Finite Elemente Matrizen.

Bemerkung 3.29.

(1) Algorithmus 3.30 entspricht Algorithmus 3.27 mit erweiterter Parallelisierung über die Zerlegung der einzelnen Matrizen nach (3.24).
(2) In Schritt $s=0$ ist in jedem Cluster t nur ein Prozessor vorhanden. Somit existiert keine parallele Zerlegung nach (3.24), insbesondere sind die Matrizen $Z^{(0),t}$ (welche der Matrix A_{tt} entspricht), $R_{\text{loc}}^{(0),t}$, $L_{\text{loc}}^{(0),t}$ und $S_{\text{loc}}^{(0),t}$ auf dem Prozessor t komplett vorhanden. Damit kann dort direkt Algorithmus 3.27 angewandt und auf die Kommunikationsroutinen verzichtet werden.
(3) In Schritt $s=0$ sind alle Matrizen dünnbesetzt, insbesondere $Z^{(0),t}$ und $L_{\text{loc}}^{(0),t}$. Für die LR-Zerlegung der Matrix $Z^{(0),t}$ in Zeile 11 kann somit ein Sparse-Löser verwendet werden[5]. Zur Berechnung des Schur-Komplements $S_{\text{loc}}^{(0),t} - L_{\text{loc}}^{(0),t}(Z^{(0),t})^{-1}R_{\text{loc}}^{(0),t}$ kann die Dünnbesetztheit von $L_{\text{loc}}^{(0),t}$ ausgenutzt werden. Dabei müssen nur diejenigen Elemente mit $L_{\text{loc}}^{(0),t}[i,j] \neq 0$ betrachtet werden.
(4) Die Matrix $A_{\text{loc}}^{(s),t}$ wird über die Blockmatrizen $A_{nm}^{(s),t}$ nach (3.21) definiert und kann somit über diese Blockmatrizen identifiziert werden[6]. Die Zusammenfassung des Schur-Komplements in Zeile 6 geschieht dabei spaltenweise. Für jede Spalte müssen nur jeweils zwei Prozessoren miteinander kommunizieren. Diese Zusammenfassung wird also über Eins-zu-Eins-Kommunikationen durchgeführt.
(5) Die Kommunikationsroutinen in Zeile 12 bis 15 finden über einen Broadcast (vergleiche Anhang A) statt.

[5] In unserem Fall verwenden wir `SuperLU`
[6] Die Blockmatrizen $A_{nm}^{(s),t}$ können sich nun auf verschiedenen Prozessoren befinden

Algorithmus 3.30 (parallele Block-LR-Zerlegung)**.**

1 **for** $s = 0, \ldots, S$
2 **for** $t = 1, \ldots, T_s$ **(parallel)**
3 **if** $(s > 0)$
4 **setze** $t_1, t_2 \in \{1, \ldots, T_{s-1}\}$ **so dass** $\mathcal{P}^{s,t} = \mathcal{P}^{s-1,t_1} \cup \mathcal{P}^{s-1,t_2}$
5 **for** $n, m \in \mathcal{K}^{s,t}$
6 $A_{nm}^{(s),t} = A_{nm}^{(s-1),t_1} + A_{nm}^{(s-1),t_2}$
7 **Konstruiere** $A_{\mathbf{loc}}^{(s),t}$ **mit** $Z_q^{(s),t}$**,** $R_q^{(s),t}$**,** $L_q^{(s),t}$**,** $S_q^{(s),t}$ **nach** (3.24)
8 **for** $q = 1, \ldots, Q^{s,t}$
9 **setze** $i = N_{q-1}^{s,t} + 1, \quad j = N_q^{s,t}$
10 **auf** $p_q^{s,t}$**:**
11 $Z_q^{(s),t}[i:j] := \mathrm{LR}(Z_q^{(s),t}[i:j])$
12 SENDE $\left[p_q^{s,t} \to p_r^{s,t}\right] \left(Z_q^{(s),t}\right)$ **für** $p_r^{s,t} \in \mathcal{P}^{s,t}$
13 SENDE $\left[p_q^{s,t} \to p_r^{s,t}\right] \left(L_q^{(s),t}\right)$ **für** $p_r^{s,t} \in \mathcal{P}^{s,t}$
14 **auf** $p_r^{s,t} \in \mathcal{P}^{s,t}$**:** EMPFANGE $\left[p_r^{s,t} \leftarrow p_q^{s,t}\right] \left(Z_q^{(s),t}\right)$
15 **auf** $p_r^{s,t} \in \mathcal{P}^{s,t}$**:** EMPFANGE $\left[p_r^{s,t} \leftarrow p_q^{s,t}\right] \left(L_q^{(s),t}\right)$
16 **auf** $p_r^{s,t} \in \mathcal{P}^{s,t}$ **mit** $r > q$ **(parallel):**
17 $Z_r^{(s),t}[i:j] := \mathrm{SOLVE}(Z_q^{(s),t}[i:j], Z_r^{(s),t}[i:j])$
18 $Z_r^{(s),t}[j+1:N^{s,t}] := Z_r^{(s),t}[j+1:N^{s,t}] - Z_q^{(s),t}[j+1:N^{s,t}] Z_r^{(s),t}[i:j]$
19 $L_r^{(s),t} := L_r^{(s),t} - L_q^{(s),t} Z_r^{(s),t}[i:j]$
20 **auf** $p_r^{s,t} \in \mathcal{P}^{s,t}$ **(parallel):**
21 $R_r^{(s),t}[i:j] := \mathrm{SOLVE}(Z_q^{(s),t}[i:j], R_r^{(s),t}[i:j])$
22 $R_r^{(s),t}[j+1:N^{s,t}] := R_r^{(s),t}[j+1:N^{s,t}] - Z_q^{(s),t}[j+1:N^{s,t}] R_r^{(s),t}[i:j]$
23 $S_r^{(s),t} := S_r^{(s),t} - L_q^{(s),t} R_r^{(s),t}[i:j]$

3.7. Bemerkungen zur parallelen Block-LR-Zerlegung für $P \neq 2^S$ Prozessoren

Algorithmus 3.30 kann in genau dieser Form auch für $P \neq 2^S$ Prozessoren angewandt werden. Es ändern sich letztendlich nur die kombinierten Prozessormengen aus Definition 3.10. Für die Anzahl der benötigten Schritte gilt

$$S = \lceil \log_2(P) \rceil \qquad \text{(aufgerundeter Zweierlogarithmus)}\,.$$

Der erste Schritt $s = 0$ beginnt wie (3.9) mit

$$\mathcal{P}^{0,t} = \{p_t\}, \quad t \in \mathcal{T}_0 = \{1, \ldots, T_0\} \tag{3.26}$$

mit $T_0 = P$. In den folgenden Schritten werden nun soweit möglich kombinierte Prozessormengen wie in (3.10) erstellt. Insgesamt sind in Schritt $s > 0$

$$T_s = \left\lceil \frac{T_{s-1}}{2} \right\rceil$$

kombinierte Prozessormengen vorhanden.

Aus je zwei kombinierten Prozessormengen $\mathcal{P}^{s-1,t_0}$ und $\mathcal{P}^{s-1,t_1}$ aus Schritt $s-1$ entsteht eine neue kombinierte Prozessormenge

$$\mathcal{P}^{s,t} = \mathcal{P}^{s-1,t_0} \cup \mathcal{P}^{s-1,t_1}, \quad s = 1, \ldots, S, \quad t = 1, \ldots, \lfloor T_{s-1}/2 \rfloor . \tag{3.27}$$

Wenn T_{s-1} ungerade ist, bleibt eine kombinierte Prozessormenge $\mathcal{P}^{s-1,t^\star}$ übrig, die für den nächsten Schritt einfach übernommen wird, das heißt, es wird

$$\mathcal{P}^{s,T_s} = \mathcal{P}^{s-1,t^\star}$$

gesetzt. In diesem Fall gilt

$$\mathcal{K}_{\mathrm{LR}}^{s,T_s} = \emptyset ,$$

da $\Pi^{s,t} = \emptyset$ (aus 3.14): Angenommen, es gäbe ein $\pi \in \Pi^{s,t}$ mit $\pi \in \Pi^s$ und $\pi \in \mathcal{P}^{s,T_s}$, dann gilt aber auch $\pi \in \mathcal{P}^{s-1,t^\star}$, das heißt $s(\pi) \leq s-1$ und somit $\pi \notin \Pi^s$. Damit ist $Z^{(s),T_s}$ nicht besetzt und das Schur-Komplement bleibt aus dem vorigen Schritt erhalten. Das heißt, auf diesem Cluster wird keine Zerlegung in diesem Schritt stattfinden, dieser Cluster wird in die Parallelisierungsarbeit nicht eingebunden. Um eine möglichst parallele Ausführung zu erhalten ist es daher sinnvoll, eine Zweierpotenz an Prozessoren zu benutzen.

Weiterhin ist es auch möglich, mehr als zwei Prozessormengen zusammenzufügen. Das heißt, es werden insgesamt weniger Schritte benötigt. Damit vergrößert sich jedoch der Arbeitsaufwand für den einzelnen Cluster und die Parallelisierung über die Idee der Nested Dissection geht zum Teil verloren.

3.8. Lösen mit Hilfe der parallelen Block-LR-Zerlegung

Mit Hilfe von Algorithmus 3.30 wurde eine parallele LR-Zerlegung einer Matrix $A \in \mathbb{R}^{N\times N}$ erstellt. Dabei ist A eine Matrix eines Finite Elemente Problems und additiv auf P Prozessoren verteilt. Die LR-Zerlegung selbst ist wiederum auf diese Prozessoren verteilt. Die Lösungsroutine ergibt sich aus der Block-Vorwärts-Substitution (Algorithmus 3.1) und der entsprechenden Block-Rückwärts-Substitution. Dabei müssen die Block-Matrizen zugehörig zu Algorithmus 3.30 angepasst werden.

Die rechte Seite $f \in \mathbb{R}^N$ aus (2.5) ist ebenso wie die Matrizen additiv gegeben. Der Lösungsvektor u zum zugehörigen Finite Elemente Problem $Au = f$ soll konsistent sein, das heißt die Einträge sollen auf den Interfaces übereinstimmen (siehe Kapitel 2.2.3).

In der Parallelisierung über die Nested Dissection werden nach Vereinbarung 3.25 die additiven Matrizen nach und nach zusammengefasst, so dass sie letztendlich global vorhanden sind. Dieses Prinzip wird auch beim Lösungsverfahren angewandt, so dass auch der Lösungsvektor am Ende global vorhanden ist und durch Kopieren auf die entsprechenden Prozessoren somit konsistent ist.

Zur besseren Übersicht und zum vereinfachten Nachvollziehen wird das Lösungsverfahren entsprechend der konstruierten Algorithmen aufgebaut. Dabei beginnen wir mit dem Lösungsverfahren zur reduzierten Block-LR-Zerlegung (Algorithmus 3.22). Daraufhin wird das Lösungsverfahren zu Algorithmus 3.27 (Block-LR-Zerlegung mit Nested Dissection) betrachtet, bevor das Verfahren zur parallelen Block-LR-Zerlegung (Algorithmus 3.30) beschrieben wird.

Da sich Vorwärts- und Rückwärts-Substitution im Verlauf teilweise deutlich unterscheiden, wird jeder Algorithmus aufgeteilt in beide Verfahren. Das allgemeine Lösungsverfahren lautet

Algorithmus 3.31. (allgemeines Lösungsverfahren)

1 **Vorwärts-Substitution**

2 **Rückwärts-Substitution**

Als Eingabeparameter dient dabei die rechte Seite f. Diese wird mit Hilfe der beiden Substitutionen überschrieben zum Lösungsvektor u des linearen Gleichungssystems $Au = f$.

3.8.1. Lösungsverfahren zur reduzierten Block-LR-Zerlegung. Wir betrachten Algorithmus 3.22 und geben dazu das Lösungsverfahren zur Lösung von $Au = f$ an. In diesem Fall sind die Matrixblöcke über A_{nm} gegeben und global definiert. Die rechte Seite f soll blockweise wie in (2.20) unterteilt und ebenso global vorhanden sein. Die Vorwärts- und Rückwärts-Substitution ergibt sich damit direkt aus den Standard-Block-Substitutionen 3.1 und 3.2, wobei nur die Operationsmengen $\mathcal{K}^{s,t}$ betrachtet werden müssen. Da hier die Reihenfolge wichtig ist, wird für die Menge $\mathcal{K}^{s,t}_{\mathrm{LR}} = \{K_{s,t-1}+1, \ldots, K_{s,t}\}$ (vergleiche Definition 3.18) zusätzlich angegeben, ob die Elemente aufsteigend $(K_{s,t-1}+1, \ldots, K_{s,t})$ oder absteigend $(K_{s,t}, \ldots, K_{s,t-1}+1)$ betrachtet werden.

Algorithmus 3.32. (Vorwärts-Substitution zu Algorithmus 3.22)

```
1  for s = 0, ..., S
2      for t = 1, ..., T_s
3          for k ∈ K^{s,t}_LR (aufsteigend)
4              f_k = SOLVE(A_kk, f_k)
5              for m ∈ K^{s,t}, m > k
6                  f_m := f_m − A_mk f_k
```

Bei der Rückwärts-Substitution muss von $k = K, \ldots, 1$ gezählt werden, wobei wir hier über die Schritte $s = S, \ldots, 0$ und die Clusternummer $t = T_s, \ldots, 1$ zählen. Insbesondere müssen die Elemente der Menge $\mathcal{K}^{s,t}_{\mathrm{LR}}$ absteigend betrachtet werden.

Algorithmus 3.33. (Rückwärts-Substitution zu Algorithmus 3.22)

```
1  for s = S, ..., 0
2      for t = T_s, ..., 1
3          for k ∈ K^{s,t}_LR (absteigend)
4              for m ∈ K^{s,t}, m < k
5                  f_m := f_m − A_mk f_k
```

3.8.2. Lösungsverfahren zur Block-LR-Zerlegung mit Nested Dissection. Für den ersten parallelen Algorithmus 3.27 mit Hilfe der Nested Dissection werden die Teilvektoren wie in Definition 2.11 Prozessorweise betrachtet, das heißt die globale Seite

$$f = \begin{pmatrix} f_1 \\ \vdots \\ f_K \end{pmatrix}$$

ist aufgeteilt in

$$f_k = \sum_{p \in \pi_k} f_k^p \,.$$

Wie für die Matrizen werden auch die Vektoren schrittweise betrachtet. Mit $f_k^{(s)}$ wird ein Teilvektor in f zu Beginn von Schritt s bezeichnet.

Damit die Vektoren nach und nach zusammengefasst werden, treffen wir für die Vorwärts-Substitution die gleiche Vereinbarung wie für Matrizen:

Vereinbarung 3.34. Bei der Zusammenführung zweier Prozessor-Sets $\mathcal{P}^{s-1,t_1}$, $\mathcal{P}^{s-1,t_2}$ zu $\mathcal{P}^{s,t} = \mathcal{P}^{s-1,t_1} \cup \mathcal{P}^{s-1,t_2}$ werden die Vektoren additiv über

$$f_k^{(s),t} = f_k^{(s-1),t_1} + f_k^{(s-1),t_2}$$

zusammengefasst. Für Schritt $s = 0$ setzen wir

$$f_k^{(s),p} = f_k^p \,.$$

Damit ist ein Teilvektor f_k im Schritt s global vorhanden, wenn $\pi_k \subset \mathcal{P}^s$. Wenn sich ein globaler Teilvektor nicht mehr ändert, wird er für die Rückwärts-Substitution auf f_k gesetzt (Zeile 11 in Algorithmus 3.35). Dieser Vektor ist dann auf allen Prozessoren in der jeweiligen Clustermenge vorhanden und die Definition zeigt das globale Vorhandensein auf allen entsprechenden Prozessoren. Die Vorwärts-Substitution lautet dann

Algorithmus 3.35 (Vorwärts-Substitution zu Algorithmus 3.27)**.**

```
1   for s = 0,...,S
2     for t = 1,...,T_s (parallel)
3       if (s > 0)
4         setze t_1, t_2 ∈ {1,...,T_{s-1}} so dass P^{s,t} = P^{s-1,t_1} ∪ P^{s-1,t_2}
5         for k ∈ K^{s,t}
6           f_k^{(s),t} = f_k^{(s-1),t_1} + f_k^{(s-1),t_2}
7       for k ∈ K_LR^{s,t} (aufsteigend)
8         f_k^{(s),t} = SOLVE(A_kk^{(s),t}, f_k^{(s),t})
9         for m ∈ K^{s,t}, m > k
10          f_m^{(s),t} = f_m^{(s),t} − A_mk^{(s),t} f_k^{(s),t}
11        f_k = f_k^{(s),t}
```

Am Ende der Vorwärts-Substitution sind alle Teilvektoren global vorhanden. Somit müssen bei der Rückwärts-Substitution keine Vektoren mehr zusammengefasst werden. Daher kann hier auf die Notation der Teilvektoren mit Zusatzinformation des Schrittes s und des Clusters t verzichtet werden. In der Vorwärts-Substitution wurde dies in Zeile 11 berücksichtigt, so dass die Rückwärts-Substitution auf den Vektoren f_k (anstatt $f_k^{(s),t}$) arbeiten kann. Die Konsistenz der Vektoren ergibt sich hier automatisch, da in diesem Algorithmus angenommen wird, dass jeder Teilvektor auf der gesamten Clustermenge vorhanden ist.

Algorithmus 3.36 (Rückwärts-Substitution zu Algorithmus 3.27)**.**

```
1   for s = S,...,0
2     for t = 1,...,T_s (parallel)
3       for k ∈ K_LR^{s,t} (absteigend)
4         for m ∈ K^{s,t}, m < k
5           f_m = f_m − A_mk^{(s),t} f_k
```

3.8.3. Finales Lösungsverfahren zur parallelen Block-LR-Zerlegung. Für die finale parallele Block-LR-Zerlegung (Algorithmus 3.30) müssen die Teilvektoren f_k neu aufgestellt werden, so dass sie zu der lokalen Matrix $A_{\text{loc}}^{(s),t}$ aus (3.24) passen.

Dazu werden im ersten Schritt analog zu den Matrizen die lokalen rechten Seiten zusammengefasst zu

$$f_{\text{loc}}^{(s),t} = \begin{pmatrix} f_{\text{LR}}^{(s),t} \\ f_{\text{Sch}}^{(s),t} \end{pmatrix}$$

mit

$$\begin{aligned} f_{\text{LR}}^{(s),t} &= (f_k^{(s),t})_{k \in \mathcal{K}_{\text{LR}}^{s,t}} \in \mathbb{R}^{N^{s,t}} \\ f_{\text{Sch}}^{(s),t} &= (f_k^{(s),t})_{k \in \mathcal{K}_{\text{Sch}}^{s,t}} \in \mathbb{R}^{M^{s,t}} \end{aligned} \tag{3.28}$$

mit $N^{s,t}, M^{s,t}$ wie in (3.21). Der zusammengefasste Vektor $f_{\text{LR}}^{(s),t}$ ist theoretisch auf dem Cluster t global vorhanden. Wir betrachten zunächst die globale Situation auf dem Cluster t in Schritt s: Wir erhalten das zu lösende Gleichungssystem

$$\begin{pmatrix} Z^{(s),t} & R_{\text{loc}}^{(s),t} \\ L_{\text{loc}}^{(s),t} & S_{\text{loc}}^{(s),t} \end{pmatrix} \begin{pmatrix} u_{\text{LR}}^{(s),t} \\ u_{\text{Sch}}^{(s),t} \end{pmatrix} = \begin{pmatrix} f_{\text{LR}}^{(s),t} \\ f_{\text{Sch}}^{(s),t} \end{pmatrix}.$$

Dabei beschreibt die Matrix $Z^{(s),t}$ eine Block-LR-Zerlegung. Weiterhin beschreibt $S_{\text{loc}}^{(s),t}$ einen Teil der zugehörigen Zerlegung in Schritt $s+1$ und wird somit in Schritt s nicht beachtet. In der Vorwärts-Substitution in Schritt s werden nur die beiden Matrizen $Z^{(s),t}$ (davon nur der linke untere Block-Teil inklusive Diagonale) und $L_{\text{loc}}^{(s),t}$ benötigt. Die Matrix $R_{\text{loc}}^{(s),t}$ und der rechte obere Block-Teil von $Z^{(s),t}$ ohne Diagonalanteil wird in der Rückwärts-Substitution benötigt.

Hier wird jedoch nicht mehr der Cluster als Ganzes gesehen, sondern jeder Prozessor auf dem Cluster wird für sich betrachtet. Auf den Prozessoren sind die lokalen Matrizen $A_{\text{loc}}^{(s),t}$ nichtüberlappend in $Z_p^{(s),t}$, $R_p^{(s),t}$, $L_p^{(s),t}$ und $S_p^{(s),t}$ unterteilt. Die Rechnungen mit diesen Matrizen (oder einem Teil davon) können immer nur auf dem Prozessor stattfinden, auf dem diese Matrix auch vorhanden ist. Die anderen Prozessoren erhalten das Ergebnis durch Kommunikation. Dies muss im Lösungsalgorithmus berücksichtigt werden.

Algorithmus 3.37 (Vorwärts-Substitution zu Algorithmus 3.30)**.**

1 **for** $s = 0, \dots, S$
2 **for** $t = 1, \dots, T_s$ **(parallel)**
3 **if** $(s > 0)$
4 **setze** $t_1, t_2 \in \{1, \dots, T_{s-1}\}$ **so dass** $\mathcal{P}^{s,t} = \mathcal{P}^{s-1,t_1} \cup \mathcal{P}^{s-1,t_2}$
5 **for** $k \in \mathcal{K}^{s,t}$
6 $f_k^{(s),t} = f_k^{(s-1),t_1} + f_k^{(s-1),t_2}$
7 **konstruiere** $f_{\textbf{loc}}^{(s),t} = \begin{pmatrix} f_{\text{LR}}^{(s),t} \\ f_{\text{Sch}}^{(s),t} \end{pmatrix}$ **nach** (3.28)
8 **for** $q = 1, \dots, Q^{s,t}$
9 **setze** $i = N_{q-1}^{s,t} + 1, \quad j = N_q^{s,t}$
10 **auf** $p_q^{s,t}$**:**
11 $f_{\text{LR}}^{(s),t}[i:j] := \textsc{Solve}(Z_q^{(s),t}[i:j], f_{\text{LR}}^{(s),t}[i:j])$

12 $\quad f_{\mathrm{LR}}^{(s),t}[j+1:N^{s,t}] :=$
$\qquad f_{\mathrm{LR}}^{(s),t}[j+1:N^{s,t}] - Z_q^{(s),t}[j+1:N^{s,t}] f_{\mathrm{LR}}^{(s),t}[i:j]$

13 $\quad f_{\mathrm{Sch}}^{(s),t} := f_{\mathrm{Sch}}^{(s),t} - L_q^{(s),t} f_{\mathrm{LR}}^{(s),t}[i:j]$

14 **if** $q < Q^{s,t}$

15 $\quad$ **auf** $p_q^{s,t}$**:** Sende $\left[p_q^{s,t} \to p_{q+1}^{s,t}\right] \left(f_{\mathbf{loc}}^{(s),t}\right)$

16 $\quad$ **auf** $p_{q+1}^{s,t}$**:** Empfange $\left[p_{q+1}^{s,t} \leftarrow p_q^{s,t}\right] \left(f_{\mathbf{loc}}^{(s),t}\right)$

17 **else**

18 $\quad$ **auf** $p_q^{s,t}$**:** Sende $\left[p_q^{s,t} \to p_r^{s,t}\right] \left(f_{\mathbf{loc}}^{(s),t}\right)$ **für** $p_r^{s,t} \in \mathcal{P}^{s,t}$

19 $\quad$ **auf** $p_r^{s,t} \in \mathcal{P}^{s,t}$**:** Empfange $\left[p_r^{s,t} \leftarrow p_q^{s,t}\right] \left(f_{\mathbf{loc}}^{(s),t}\right)$

Anstatt in der Vorwärts-Substitution in jedem Schritt den kompletten Vektor auf alle anderen Prozessoren zu kopieren, wird der Vektor immer nur zum nächsten bearbeitenden Prozessor geschickt (Zeile 14 bis 16) und erst am Ende an alle Prozessoren verteilt (Zeile 17 bis 19). Dabei genügt es, nur die Prozessoren in der jeweiligen Prozessormenge $\mathcal{P}^{s,t}$ zu berücksichtigen, da die anderen Prozessoren die Information des Teilvektors nicht benötigt. Die Addition der Teilvektoren f_k aus dem vorigen Schritt in Zeile 3 bis 6 ist wieder nur ein programmiertechnisches Problem und soll hier nicht weiter betrachtet werden. Die Zuweisung in Zeile 7 entspricht einer Bijektion zu den Teilvektoren $f_k^{(s),t}$, das heißt, aus $f_{\mathrm{LR}}^{(s),t}$ und $f_{\mathrm{Sch}}^{(s),t}$ können die entsprechenden Teilvektoren $f_k^{(s),t}$ wieder entnommen werden, so dass die Operationen in Zeile 11 bis 13 eigentlich auf Teilen der Vektoren $f_k^{(s),t}$ arbeiten.

Für die Rückwärts-Substitution benötigen wir für die Anwendung der Matrix $R_{\mathrm{loc}}^{(s),t}$, welche auf die Prozessoren verteilt ist, die Information, welche Elemente von $f_{\mathrm{Sch}}^{(s),t}$ zu diesen Spalten gehören. Für $Z^{(s),t}$ wurden die zugehörigen Elemente von $f_{\mathrm{LR}}^{(s),t}$ mit $N_k^{s,t}$ in (3.25) definiert. Dementsprechend werden die Spaltennummern für den rechten Matrixteil von $A_{\mathrm{loc}}^{(s),t}$ definiert:

$$\begin{aligned} M_0^{s,t} &= 0, \\ M_q^{s,t} &= \sum_{m=1}^{q} M_{p_m^{s,t}}, \qquad 1 \le q \le Q^{s,t}. \end{aligned} \tag{3.29}$$

Die Rückwärts-Substitution zur parallelen Block-LR-Zerlegung teilt sich im Gegensatz zu den vorigen Rückwärts-Substitutionen in zwei Teile auf. In Schritt s wird zuerst die Matrix $R_{\mathrm{loc}}^{(s),t}$ in der Lösungsroutine angewandt, danach der obere Teil von $Z^{(s),t}$. In $f_{\mathrm{Sch}}^{(s),t}$ ist schon die exakte Lösung vorhanden (diese wurde in den Schritten $s+1, \ldots, S$ erstellt). Der erste Schritt in dieser Substitution (Zeile 3 bis 7 in Algorithmus 3.38) liefert global auf dem Cluster gesehen

$$f_{\mathrm{LR}}^{(s),t} = f_{\mathrm{LR}}^{(s),t} - R_{\mathrm{loc}}^{(s),t} f_{\mathrm{Sch}}^{(s),t}.$$

Das heißt, es werden nur Einträge in $f_{\mathrm{LR}}^{(s),t}$ geändert. Da die Matrix $R_{\mathrm{loc}}^{(s),t}$ spaltenweise verteilt ist, kann diese Formulierung auch wie folgt geschrieben werden:

$$f_{\mathrm{LR}}^{(s),t} = f_{\mathrm{LR}}^{(s),t} - \sum_{q=1}^{Q^{s,t}} R_q^{(s),t} f_{\mathrm{Sch}}^{(s),t}[M_{q-1}^{s,t}+1 : M_q^{s,t}].$$

Die Summanden sind unabhängig voneinander, diese können also parallel berechnet werden und müssen daraufhin nur noch zu $f_{\mathrm{LR}}^{(s),t}$ aufaddiert werden (Zeile 6 bis 7). Dazu dient die Funktion

- SUMMIERE $[\mathcal{P}](h)$
  Der Vektor h wird mit Hilfe eines Allreduce (siehe Anhang B) von allen Prozessoren in der Prozessormenge $\mathcal{P}$ aufsummiert und in h gespeichert.

Algorithmus 3.38 (Rückwärts-Substitution zu Algorithmus 3.30)**.**

```
1   for s = S, ..., 0
2     for t = 1, ..., T_s (parallel)
3       for q = 1, ..., Q^{s,t} (parallel)
4         setze i = M_{q-1}^{s,t} + 1,   j = M_q^{s,t}
5         auf p_q^{s,t}:
6           h = R_q^{(s),t} f_Sch^{(s),t}[i : j]
7           f_LR^{(s),t} = f_LR^{(s),t} - SUMMIERE[P^{s,t}](h)
8       for q = Q^{s,t}, ..., 2
9         setze i = M_{q-1}^{s,t} + 1,   j = M_q^{s,t}
10        auf p_q^{s,t}:
11          f_LR^{(s),t}[1 : i-1] := f_LR^{(s),t}[1 : i-1] - Z_q^{(s),t}[1 : i-1] f_LR^{(s),t}[i : j]
12        if (q > 2)
13          auf p_q^{s,t}: SENDE [p_q^{s,t} -> p_{q-1}^{s,t}] (f_LR^{(s),t})
14          auf p_{q-1}^{s,t}: EMPFANGE [p_{q-1}^{s,t} <- p_q^{s,t}] (f_LR^{(s),t})
15        else if (q == 2)
16          auf p_q^{s,t}: SENDE [p_q^{s,t} -> p_r^{s,t}] (f_LR^{(s),t}) für p_r^{s,t} ∈ P^{s,t}
17          auf p_r^{s,t} ∈ P^{s,t}: EMPFANGE [p_r^{s,t} <- p_q^{s,t}] (f_LR^{(s),t})
18      if (s > 0)
19        setze t_1, t_2 ∈ {1, ..., T_{s-1}} so dass P^{s,t} = P^{s-1,t_1} ∪ P^{s-1,t_2}
20        for c = 1, 2
21          for k ∈ K^{s-1,t_c}
22            f_k^{(s-1),t_c} = f_k^{(s),t}
23      for k ∈ K_LR^{s,t}
24        f_k = f_k^{(s),t}
```

Der zweite Teil (Zeile 8 bis 17) ist ähnlich zur Vorwärts-Substitution aufgebaut, da hier wieder die Matrix $Z^{(s),t}$ betrachtet werden muss. Dieser Teil erfolgt wieder sequentiell, da die Prozessoren $p_1, \ldots, p_q$ das Ergebnis der Prozessoren $p_{q+1}, \ldots, p_{Q^{s,t}}$ benötigen. Weiterhin muss auf Prozessor p_1 keine Berechnung erfolgen, da dort kein oberer Dreiecksanteil der Zerlegung von $Z_{\mathrm{loc}}^{(s),t}$ zu finden ist (Zeile 8 bis 11). Daraufhin wird wie in der Vorwärts-Substitution der aktualisierte Vektor $f_{\mathrm{LR}}^{(s),t}$ zum nächsten Prozessor – beziehungsweise nach Vollendung der Schleife in 8 an alle Prozessoren – geschickt (Zeile 12 bis 17). Zum Schluss werden in Zeile 18 bis 22 der Vektor $f_{\mathrm{loc}}^{(s-1),t_c}$ $(c = 1, 2)$ für den nächsten Schritt $s - 1$ auf den Clustern t_1 und t_2 gesetzt, sowie kenntlich gemacht, welche Vektoren f_k ihr globales Endergebnis erreicht haben (Zeile 23 bis 24).

3.9. Wohldefiniertheit der parallelen Block-LR-Zerlegung

3.9.1. Symmetrisch positiv definite Matrizen.

In diesem Kapitel zeigen wir die Wohldefiniertheit der parallelen Block-LR-Zerlegung für positiv definite Matrizen. Dafür zitieren wir insbesondere aus [**Pla04**, **DR08**, **DH02**], ändern dabei aber die Notation auf die Form unserer LR-Zerlegung ab[7].

Definition 3.39 (symmetrisch positiv definite Matrix)**.**
Eine Matrix $A \in \mathbb{R}^{N\times N}$ heißt symmetrisch positiv definit (spd), falls

$$A^T = A \qquad \textit{(Symmetrie)}$$

und

$$x^T Ax > 0 \qquad \textit{(positiv definit)}$$

für alle $x \in \mathbb{R}^N$, $x \neq 0$ gilt.

Symmetrisch positiv definite Matrizen entstehen beispielsweise bei der Finite Elemente Diskretisierung eines Laplace- oder Elastizität-Problems. Wir zeigen, dass die parallele Block-LR-Zerlegung für positiv definite Matrizen ohne Pivotisierung durchführbar ist.

Satz 3.40. *Sei $A \in \mathbb{R}^{N\times N}$ symmetrisch positiv definit. Dann gilt*

(i) *A ist invertierbar.*
(ii) *$A[k,k] > 0$ für $k = 1,\dots,N$.*
(iii) *Jede Hauptuntermatrix $A[k:l,k:l]$ $(k,l = 1,\dots,N)$ von A ist symmetrisch positiv definit.*
(iv) *Sei $k \in \{1,\dots,N\}$. Das Schur-Komplement*

$$A[k+1:N,k+1:N] - A[k+1:N,1:k]A[1:k,1:k]^{-1}A[1:k,k+1:N]$$

ist positiv definit.

Satz 3.40(iv) besagt, dass bei der Durchführung der LR-Zerlegung ohne Pivotisierung jede Restmatrix (also das Schur-Komplement) wieder symmetrisch positiv definit ist.

Beweis. Die Invertierbarkeit von A folgt direkt aus der Eigenschaft $x^T Ax > 0$ für alle $x \in \mathbb{R}^N$, $x \neq 0$ (wenn A singulär wäre, gäbe es ein $x \in \mathbb{R}^N$, $x \neq 0$, so dass $Ax = 0$). (ii) folgt, indem $x = e_k$ (k-ter Einheitsvektor) gewählt wird:

$$0 < e_k^T Ae_k = A[k,k]\,.$$

Für (iii) wähle $z = (z[j])_{j=k}^{l} \in \mathbb{R}^{l-k+1}$ mit $z \neq 0$. Setze

$$x = \begin{cases} x[j] = z[j] & j = k,\dots,l \\ x[j] = 0 & \text{sonst}\,. \end{cases}$$

Dann gilt

$$z^T A[k:l,k:l]z = \sum_{i,j=k}^{l} a_{ij}z_i z_j = \sum_{i,j=1}^{N} a_{ij}x_i x_j = x^T Ax > 0\,.$$

[7]Erinnerung: In unserem Fall ist die Matrix R eine obere normierte Dreiecksmatrix.

Zum Beweis von (iv) betrachten wir die Matrix

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{12}^T & A_{22} \end{pmatrix}$$

und das zugehörige Schur-Komplement

$$S = A_{22} - A_{12}^T A_{11}^{-1} A_{12} \,.$$

Das Schur-Komplement ist wohldefiniert, da A_{11} nach (iii) symmetrisch positiv definit und damit nach (i) invertierbar ist. Wir setzen

$$T = \begin{pmatrix} I & 0 \\ A_{12}^T A_{11}^{-1} & I \end{pmatrix} .$$

Damit gilt

$$A = T \begin{pmatrix} A_{11} & 0 \\ 0 & S \end{pmatrix} T^T .$$

T ist eine untere normierte Dreiecksmatrix und somit invertierbar. Insbesondere ist S symmetrisch und für $y \neq 0$ setze $x = T^{-T} \begin{pmatrix} 0 \\ y \end{pmatrix} \neq 0$ und somit gilt

$$x^T S y = x^T A x > 0 \,.$$

□

Für eine symmetrisch positiv definite Matrix gilt also insbesondere für das erste Diagonalelement $A[1,1] > 0$. Somit ist auch der erste Diagonaleintrag in jedem Schur-Komplement $\tilde{A}[k,k] > 0$ und die gesamte LR-Zerlegung ist ohne Pivotisierung durchführbar. Weiterhin ist jede Hauptuntermatrix positiv definit.

In Algorithmus 3.30 ist das Schur-Komplement additiv auf den Prozessoren verteilt, für die jeweilige LR-Zerlegung in Schritt 11 wird aber nur die Matrix $Z^{(s),t}$ benötigt. Diese ist nach Lemma 3.26 global auf den Prozessoren vorhanden, außerdem eine Hauptuntermatrix und somit positiv definit. Die Zerlegung jeder dieser Blockmatrizen ist also ohne Pivotisierung durchführbar. Für diese Matrix wird wiederum eine Block-LR-Zerlegung auf p Prozessoren durchgeführt, jede einzelne Zerlegung findet aber wieder auf einer Hauptuntermatrix statt. Somit ist der gesamte Algorithmus wohldefiniert.

3.9.2. Erweiterung auf eine nichtsymmetrische Variante.

Wir erweitern die Aussage der Wohldefiniertheit auf reguläre Matrizen mit invertierbaren Hauptuntermatrizen.

Satz 3.41. *Sei $A \in \mathbb{R}^{N\times N}$ mit regulären Hauptuntermatrizen gegeben, das heißt $A[1:k,1:k]$ ist regulär. Dann existiert eine Block-LR-Zerlegung. Insbesondere ist das Schur-Komplement*

$$S = A[k+1:N,k+1:N] - A[k+1:N,1:k]A[1:k,1:k]^{-1}A[1:k,k+1:N]$$

regulär und S besitzt reguläre Hauptuntermatrizen.

Beweis. Für die Existenz der Block-LR-Zerlegung folgen wir der Beweisidee aus [**Pla04**]. Es genügt, die Regularität des Schur-Komplements zu zeigen. Dazu sei

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} A[1:k,1:k] & A[1:k,k+1:N] \\ A[k+1:N,1:k] & A[k+1:N,k+1:N] \end{pmatrix} .$$

A_{11} ist nach Voraussetzung regulär und das Schur-Komplement lässt sich damit beschreiben als

$$A_{22} - A_{21}A_{11}^{-1}A_{12} \,. \tag{3.30}$$

Wir verwenden nun den Ansatz

$$L = \left(\begin{array}{c|c} L_k & 0 \\ \hline X^T & S \end{array}\right), \qquad R = \left(\begin{array}{c|c} R_k & Y \\ \hline 0 & I \end{array}\right),$$

wobei $L_k R_k$ eine LR-Zerlegung von A_{11} beschreibt und bestimmen $X, Y \in \mathbb{R}^{k \times N-k}$ und $S \in \mathbb{R}^{N-k \times N-k}$. Wir erhalten die Gleichungen

$$L_k Y = A_{12} \qquad X^T R_k = A_{21} \tag{3.31}$$

$$X^T Y + S = A_{22}\,. \tag{3.32}$$

Aus (3.31) erhalten wir $Y = L_k^{-1} A_{12}$ und $X^T = A_{21} R_k^{-1}$ und mit (3.32) folgt

$$S = A_{22} - X^T Y = A_{22} - A_{21} R_k^{-1} L_k^{-1} A_{12} = A_{22} - A_{21} A_{11}^{-1} A_{12}\,,$$

welches gerade dem Schur-Komplement (3.30) entspricht. Die Regularität folgt nun aus

$$\begin{aligned} \det(A) &= \det \left(\begin{array}{c|c} L_k & 0 \\ \hline X^T & S \end{array}\right) \det \left(\begin{array}{c|c} R_k & Y \\ \hline 0 & I \end{array}\right) \\ &= \det(L_k)\det(S)\det(R_k) = \det(A_{11})\det(S)\,. \end{aligned}$$

Wegen $\det(A) \neq 0$ und $\det(A_{11}) \neq 0$ folgt schließlich $\det(S) \neq 0$ und somit S regulär.

Zum Beweis der regulären Hauptuntermatrizen von S setze

$$\tilde{A} = \begin{pmatrix} \tilde{A}_{11} & \tilde{A}_{12} \\ \tilde{A}_{21} & \tilde{A}_{22} \end{pmatrix} = \begin{pmatrix} A[1:k, 1:k] & A[1:k, k+1:M] \\ A[k+1:M, 1:k] & A[k+1:M, k+1:M] \end{pmatrix}$$

für $M = k+1, \ldots, N$. Damit beschreibt $\tilde{A}_{22} - \tilde{A}_{21}\tilde{A}_{11}^{-1}\tilde{A}_{12}$ die $(M-k)$-te Hauptuntermatrix von S, welche nach dem vorigen Beweisabschnitt regulär ist, da $\tilde{A}$ als Hauptuntermatrix von A regulär ist und zusätzlich reguläre Hauptuntermatrizen aufweist. □

Die Matrix $Z^{(s),t}$ in der parallelen Block-LR-Zerlegung ist hierbei eine Hauptuntermatrix des globalen Schur-Komplements aus Schritt $s-1$. Diese ist somit regulär mit regulären Hauptuntermatrizen, so dass hiervon eine LR-Zerlegung berechnet werden kann (Zeile 11 in Algorithmus 3.30).

3.9.3. Bezug zu Finite Elemente und zugehörige Matrizen.

Wir zitieren in diesem Kapitel aus [**Bra03**] und zeigen, dass im Bereich der Finiten Elemente Matrizen entstehen, die eine der Eigenschaften aus den vorigen beiden Kapitel 3.9.1 und 3.9.2 besitzen, so dass die Block-LR-Zerlegung dort wohldefiniert ist. Wir verzichten hier auf die zugehörigen Beweise, diese sind in [**Bra03**] zu finden.

Für symmetrische Bilinearformen a ist der Satz von Lax-Milgram bedeutend. Zuerst benötigen wir die Definition einer elliptischen Bilinearform.

Definition 3.42 (elliptische Bilinearform)**.**
Sei H ein Hilbert-Raum. Eine Bilinearform $a\colon H \times H \to \mathbb{R}$ heißt stetig*, wenn mit einem $C > 0$*

$$|a(u,v)| \leq C\|u\| \cdot \|v\| \quad \textit{für alle } u, v \in H$$

gilt. Eine symmetrische, stetige Bilinearform a heißt elliptisch*, wenn ein $\alpha > 0$ existiert, so dass*

$$a(v,v) \geq \alpha \|v\|^2 \quad \textit{für alle } v \in H$$

gilt.

Satz 3.43 (Satz von Lax-Milgram für konvexe Mengen)**.**
Sei V eine abgeschlossene, konvexe Menge in einem Hilbert-Raum H und $a\colon H \times H \to \mathbb{R}$ eine elliptische Bilinearform. Für jedes $\ell \in H'$ hat das Variationsproblem

$$J(v) := \frac{1}{2}a(v,v) - \langle \ell, v\rangle \to \min! \tag{3.33}$$

genau eine Lösung in V.

Sei nun a eine elliptische Bilinearform und $V_h \subset V$ ein Finite Elemente Raum mit $V_h = \operatorname{span}\{\phi_1, \ldots, \phi_N\}$. Das Variationsproblem (3.33) eingeschränkt auf V_h kann umformuliert werden zu einem linearen Gleichungssystem

$$Au = f$$

mit $A[i,j] = a(\phi_i, \phi_j)$ und $f_i = \langle \ell, \phi_i\rangle$. Die Matrix A ist positiv definit.

Eine elliptische Bilinearform ist beim Poisson-Problem (vergleiche Kapitel 6.1) und Elastizitäts-Problem (vergleiche Kapitel 6.3) gegeben. Für diese kann das Ergebnis aus Kapitel 3.9.1 angewandt werden.

Für die nichtelliptischen Probleme betrachten wir die Hilberträume U und V, sowie eine (nicht notwendig symmetrische) Bilinearform $a\colon U \times V \to \mathbb{R}$. Dieser wird ein linearer Operator $L\colon U \to V'$ durch

$$\langle Lu, v\rangle = a(u,v) \quad \text{für } v \in V$$

zugeordnet. Es gilt folgender Satz:

Satz 3.44. *Seien U und V Hilbert-Räume. Eine lineare Abbildung $L\colon U \to V'$ ist genau dann ein Isomorphismus, wenn die zugehörige Bilinearform $a\colon U \times V \to \mathbb{R}$ folgende Bedingungen erfüllt:*

(1) Stetigkeit: Es existiert ein $C > 0$ mit

$$|a(u,v)| \leq C\|u\|_U \|v\|_V .$$

(2) inf-sup-Bedingung: Es existiert ein $\alpha > 0$, so dass

$$\inf_{u\in U} \sup_{v\in V} \frac{a(u,v)}{\|u\|_U \|v\|_V} \geq \alpha . \tag{3.34}$$

(3) Zu jedem $v \in V$, $v \neq 0$ gibt es ein $u \in U$ mit

$$a(u,v) \neq 0 .$$

Für das Stokes-Problem aus Kapitel 6.2 betrachten wir die Hilberträume

$$X = H_0^1(\Omega)^2 \quad \text{und} \quad M = L_0^2(\Omega) = \{q \in L_2(\Omega) \colon \int_\Omega q \, \mathrm{d}x = 0\}$$

und die Bilinearformen

$$a \colon X \times X \to \mathbb{R}, \qquad b \colon X \times M \to \mathbb{R}\,.$$

Für die Bilinearform b gelte nun folgende inf-sup-Bedingung[8]: Es existiert ein $\beta > 0$ mit

$$\inf_{q \in M} \sup_{v \in X} \frac{b(v,q)}{\|v\|\|q\|} \geq \beta\,. \tag{3.35}$$

Das Stokes-Problem wird als Sattelpunktproblem

$$\begin{aligned} &\text{Suche } (u,p) \in X \times M, \text{ so dass} \\ &a(u,v) + b(v,p) = l(v) \qquad \forall v \in X \\ &b(u,q) = 0 \qquad \forall q \in M \end{aligned} \tag{3.36}$$

beschrieben. Hierdurch wird eine lineare Abbildung

$$\begin{aligned} L \colon X \times M &\to X' \times M' \\ (v,q) &\mapsto (f,g) \end{aligned} \tag{3.37}$$

definiert. Der folgende Satz garantiert die Invertierbarkeit der zugehörigen Matrix $\begin{pmatrix} A & B^T \\ B & 0 \end{pmatrix}$ (vergleiche Kapitel 6.2). Hierbei definieren wir mit

$$X_0 := \{v \in X \colon b(v,q) = 0 \qquad \text{für } q \in M\}$$

den Kern von B als abgeschlossenen Unterraum von X.

Satz 3.45. *Durch das Sattelpunktproblem (3.36) wird mit (3.37) genau dann ein Isomorphismus $L \colon X \times M \to X' \times M'$ erklärt, wenn die beiden folgenden Bedingungen erfüllt sind:*

(1) Die Bilinearform a ist X_0-elliptisch.
(2) Die Bilinearform b erfüllt die inf-sup-Bedingung (3.35).

Nach [**Bra03**, §6] sind diese Bedingungen für das Stokes-Problem erfüllt.

Wir nehmen nun an, dass die Bedingungen aus Satz 3.44 für ein Finites Elemente Problem mit zugehörigen Hilberträumen $U = V \subset L_2(\Omega)$ auf einem Gebiet $\Omega \subset \mathbb{R}^3$ gelten. Weiterhin seien die diskreten Räume U_h, V_h so gewählt, dass die diskrete inf-sup-Bedingung gleichmäßig in h gilt, das heißt

$$\inf_{u_h \in U_h} \sup_{v_h \in V_h} \frac{a(u_h, v_h)}{\|u_h\|_U \|v_h\|_V} \geq \alpha_h \geq \bar{\alpha} > 0\,.$$

Satz 3.46. *Seien $U_h = V_h$ Finite Elemente Räume mit*

$$V_h = \operatorname{span}\{\phi_1, \ldots, \phi_N\}\,.$$

Für die Bilinearform $a \colon V_h \times V_h \to \mathbb{R}$ seien die Voraussetzungen aus Satz 3.44 für alle $\bar{\omega} \subset \Omega$ und

$$V_h(\omega) := \{v \in V_h \colon v = 0 \text{ auf } \Omega \setminus \omega\} \tag{3.38}$$

erfüllt. Dann sind die Block-Hauptuntermatrizen

$$\tilde{A} = (A_{lm})_{l,m=1,\ldots,k}$$

[8] In diesem Zusammenhang wird die inf-sup-Bedingung auch als *Brezzi-Bedingung* bezeichnet

der globalen Steifigkeitsmatrix A (vergleiche (3.16) aus Kapitel 3 auf Seite 39) für $k = 1, \dots, K$ regulär.

BEWEIS. Sei $N_k = |\mathcal{I}_k|$ (vergleiche Definition 2.4). Damit hat der Matrixblock A_{lm} die Größe $N_l \times N_m$. Mit $\hat{N}_k = \sum_{n=1}^{n} N_k$ hat die Block-Hauptuntermatrix $\tilde{A}$ die Größe $\hat{N}_k \times \hat{N}_k$. Sei

$$V_{h,k} = \operatorname{span}\{\phi_1, \dots, \phi_{\hat{N}_k}\} = \operatorname{span}\{\phi_i \colon i \in \bigcup_{i=1}^{k} \mathcal{I}_i\}\,.$$

Sei weiterhin

$$\hat{\omega}_k = \bigcup_{n=1}^{\hat{N}_k} \operatorname{int}(\operatorname{supp}\, \phi_k)\,.$$

das Innere vom zu betrachtendem Gebiet und

$$\omega_k = \hat{\omega}_k \cup (\partial\hat{\omega}_k \cap \partial\Omega)$$

das Gesamtgebiet. Die Definition über zwei Schritte ist notwendig, um eventuelle Randbedingungen von Ω nicht zu vernachlässigen (diese Elemente befinden sich innerhalb des Gebietes ω_k) und gleichzeitig diejenigen Interfaces, welche nicht betrachtet werden sollen, herauszunehmen (diese Elemente befinden sich nicht im Gebiet ω_k, aber eventuell auf dem Rand von ω_k).

Wir betrachten das Finite Elemente Problem eingeschränkt auf $\bar{\omega}_k$ mit Dirichlet-Nullrandbedingungen auf $\partial\omega_k \setminus \partial\Omega$. Das heißt, die Nullrandbedingungen sind definiert auf den Interfaces, welche nicht zu ω_k gehören, aber am Rand von ω_k liegen. Somit gilt insbesondere $V_h(\omega_k) = V_{h,k}$ und als Steifigkeitsmatrix erhalten wir

$$\hat{A} = a(\phi_i, \phi_j)_{\phi_i, \phi_j \in V_{h,k}}\,,$$

was genau der Block-Hauptuntermatrix $\tilde{A}$ entspricht (vergleiche Definition 2.5 und Bemerkung 2.6) Nach Voraussetzung ist diese Matrix regulär. □

Mit Anwendung von Satz 3.41 ist damit jede Matrix $Z^{(s),t}$ im Schur-Komplement invertierbar und die Block-LR-Zerlegung wohldefiniert.

Bemerkung 3.47.
Für das Stokes-Problem mit $U = X \times M$ verwenden wir zur Diskretisierung inf-sup-stabile Taylor-Hood-Serendipity Q2/Q1-Elemente. Die Voraussetzungen für (3.38) sind nach [**BF91**, Kapitel IV] gegeben, wenn jedes Dreieckselement höchstens eine Kante mit Dirichlet-Randbedingungen besitzt.

4 Implementierung der parallelen Block-LR-Zerlegung

Für die Implementierung der parallelen Block-LR-Zerlegung (Algorithmus 3.30) in M++[1] betrachten wir zuerst die gegebene Programmstruktur, insbesondere wie die Gitterstruktur und die zugehörigen Matrizen aufgebaut sind. Das Programm ist soweit undokumentiert, so dass keine allgemeine Referenzen zur Programmstruktur angegeben werden können. In einigen Arbeiten (unter anderem in [**NCMW07**, **Wie07**, **Wie08**]) und Dissertationen [**Mü09**, **Sau10**] wurde M++ als Bearbeitungsprogramm eingesetzt, so dass dort teilweise auch einige Zeilen Code und Beschreibungen zu finden sind. Allgemein beruht M++ auf dem parallelen Programmiermodell aus [**Wie10**], welches in Kapitel 2 vorgestellt wurde.

4.1. Knoten und Matrixerstellung

Die Implementierung des parallelen Programmiermodells in M++ ist in C++ geschrieben und benutzt daher insbesondere objektorientierte Programmiertechniken. Die zugrunde liegende Geometrie, sowie die zugehörigen Freiheitsgrade werden in einem `MatrixGraph` abgespeichert, wobei alle Konstrukte (zum Beispiel Zellen, Seitenflächen) über Gitterpunkte zugeordnet sind. Allgemein ist für Gitterpunkte eine Hash-Funktion gegeben, so dass alle Konstrukte, welche die Gitterpunkte verwenden, über eine Hash-Map ihre Informationen speichern.

Der Programmablauf gestaltet sich wie folgt: Zuerst werden auf einem Prozessor aus einer `.geo`-Datei alle benötigten Informationen des Gitters eingelesen. Die Ecken $\mathcal{V}$ werden zuerst beschrieben, welche in einer Liste abgespeichert werden, so dass eine Nummerierung dieser vorhanden ist. Diese sind ebenso als erste Gitterpunkte gegeben. Die Zellen und Seitenflächen werden daraufhin mit Hilfe der gegebenen Nummerierung der Ecken $\mathcal{V}$ angegeben, wobei die entsprechenden Mittelpunkte dieser Konstrukte als neue Gitterpunkte definiert werden. Für die Seitenflächen können zusätzlich noch eventuelle Randbedingungen über einen Index angegeben werden. Eine entsprechende `.geo`-Datei hat dabei zum Beispiel folgenden Inhalt:

```
POINTS
0.00    0.00    0.00          //  0
1.00    0.00    0.00          //  1
1.00    1.00    0.00          //  2
0.00    1.00    0.00          //  3
0.00    0.00    1.00          //  4
```

[1] Meshes, Multigrid and More

```
1.00   0.00   1.00          //  5
1.00   1.00   1.00          //  6
0.00   1.00   1.00          //  7
CELLS
8 0 0 1 2 3 4 5 6 7
FACES
4 1 4 5 6 7  // oben
4 1 1 2 6 5  // rechts
4 1 0 1 5 4  // vorn
4 1 3 0 4 7  // links
4 1 2 3 7 6  // hinten
4 1 0 1 2 3  // unten
```

Nach dem Schlüsselwort `POINTS` werden nacheinander die Gitterpunkte definiert. Eine Kombination der Gitterpunkte ergeben die Zellen (`CELLS`), wobei die erste Zahl die Anzahl der Gitterpunkte der einzelnen Zelle angibt und die zweite Zahl einen Index[2] zur Zelle anbietet. Die darauf folgenden Zahlen entsprechen den Eckpunkten der Zelle, welche über die Liste der `POINTS` definiert wurden. Die Seitenflächen und Kanten ergeben sich aus der Reihenfolge der angegebenen Gitterpunkte. Zum Schluss werden die entsprechenden Seitenflächen an den Außenrändern (`FACES`) definiert, wobei die erste Zahl hier wieder die Anzahl der benötigten Gitterpunkte angibt und die zweite Zahl eventuelle Randbedingungen beschreibt. In diesem speziellen Fall wird ein Einheitswürfel mit Dirichlet-Randbedingungen (`1` als zweite Zahl bei den Seitenflächen) erzeugt. Da hier insgesamt nur ein Würfel gegeben ist, existieren sechs Seitenflächen, die jeweils gleichzeitig auch Außenflächen sind.

Dieses Gitter wird nun über eine Konfigurations-Variable `plevel = k` insgesamt k-mal verfeinert und entsprechende Gitterpunkte, Zellen, Kanten und Seitenflächen erzeugt. Daraufhin werden mit Hilfe einer angegebenen `distribution` alle Zellen auf die vorhandenen P Prozessoren verteilt[3]. Dabei wird jedem Gitterpunkt z, welcher sich nun auf mehreren Prozessoren befinden kann, eine Prozessormenge über eine Abbildung $\pi(z) \subset \mathcal{P}$ in einer Klasse `ProcSet` zugeordnet (vergleiche Kapitel 2.1). Über die Konfiguration `level = l` wird nun auf jedem Prozessor weiter verfeinert und entsprechende Gitterpunkte mit zugehörigen Prozessormengen erzeugt, bis insgesamt l-mal verfeinert wurde.

Die Diskretisierung mit zugehöriger Quadratur läuft über ausgewählte Gitterpunkte, die als `row` bezeichnet werden. Dabei kann jede `row` eine bestimmte Anzahl an Freiheitsgraden haben. Ein Eintrag in der Matrix ist definiert über jeweils zwei `row`s mit zugehörigem Freiheitsgrad. Eine Zeile in der Matrix A entspricht damit jeweils einer zugehörigen `row` und einem Freiheitsgrad. Somit ist jede Zeile der Matrix direkt einem Gitterpunkt zugeordnet, das heißt, eine Sortierung der Matrixzeilen bzw. -spalten kann über eine Sortierung der Gitterpunkte geschehen. Auf jedem Prozessor wird die lokale Matrix A^p erstellt, welche in der Summe die globale Matrix A ergibt.

Die komplette Struktur, das heißt, insbesondere die Gitterpunkte der `row`s, die zugehörige Prozessormenge und die Anzahl der Freiheitsgrade wird in einer Klasse `MatrixGraph` abgespeichert, die für alle weiteren Konstrukte, wie Vektoren (`Vector`) und Matrizen (`Matrix`) als Oberklasse dient. In der Klasse `Matrix` und `Vector` werden die Matrix- bzw. Vektoreinträge in einem `BasicVector` abgespeichert, wobei die Indizierung über die Oberklassen `MatrixGraph` und insbesondere

[2] zum Beispiel zur Definition verschiedener Materialien

[3] Hierbei benutzen wir i.A. entweder den RCB-Algorithmus [**KK95**], eine Bisektions-Algorithmus [**PSL90**] oder eine Verteilung über einen angeschlossenen Graphpartitionierer `Kaffpa` [**SS11**]

`row` definiert wird. Somit können die Einträge der Matrix (bzw. der Vektoren für die rechte Seite) direkt über die `row` angesprochen und geändert werden.

Um mit einer Matrix A Rechnungen durchzuführen, kann diese in das CRS-Format[4] transformiert werden, welche in der Klasse `BasicSparseMatrix` gegeben ist. Diese Klasse benutzt folgende Speichereinheiten:

```
class BasicSparseMatrix {
    Scalar* a;
    int* d;
    int* col;
    int n;
 ...}
```

Die Nicht-Null-Einträge der Matrix A mit Gesamtgröße `n` werden dabei zeilenweise und von links nach rechts im Vektor `a` abgespeichert[5], während `col` die zugehörige Spalte zum passenden Element in `a` beschreibt. An der k-ten Stelle von `d` wird die Anzahl der Nicht-Null-Elemente vor der k-ten Zeile abgespeichert, was dem Beginn der Matrixzeile k in `a` entspricht. In Zeile k der Matrix A befinden sich dementsprechend `d[k+1]-d[k]` Nicht-Null-Einträge.

Beispiel 4.1. Gegeben sei die Matrix

$$A = \begin{pmatrix} 11 & 12 & 0 & 0 \\ 21 & 22 & 0 & 24 \\ 0 & 0 & 33 & 34 \\ 41 & 0 & 0 & 44 \end{pmatrix}.$$

Die Matrix A hat im CRS-Format folgende Form (wobei die Zählweise der Vektoren ab 0 beginnt):

```
a   = [11, 12, 22, 21, 24, 33, 34, 44, 41]    // Inhalt
col = [ 0,  1,  1,  0,  3,  2,  3,  3,  0]    // zug. Spalte
d   = [ 0,  2,  5,  7,  9]
```

Zusammenfassung:
Auf jedem Prozessor steht ein lokaler `MatrixGraph` und eine lokale Matrix A^p zur Verfügung. Jeder Zeile bzw. Spalte einer Matrix ist einer `row` und einem geordneten Freiheitsgrad zugeordnet. Eine `row` entspricht gleichzeitig einem Gitterpunkt `Point`. Weiterhin besitzt jeder Gitterpunkt x eine zugehörige Prozessormenge $\pi(x)$ (welche hier in der Klasse `ProcSet` gespeichert wird).

4.2. Grundstruktur der parallelen Block-LR-Zerlegung

Der parallele Löser ist sowohl für reelle als auch für komplexe Daten implementiert. Dafür wurde der Datentyp `Scalar` gewählt, welcher je nach Fall ein `double` oder ein `complex<double>` beschreiben kann.

Für die Konstruktion des parallelen Block-LR-Lösers werden – zusätzlich zur lokalen Matrix A^p im Sparse-Format – folgende Daten benötigt:

```
vector<double>* coordinate
vector<short>*  procsets
int*            ind
short*          dof
int             numR
int             p
int             NP
```

[4] Compressed Row Storage

[5] In unserem Fall wird meist das Diagonalelement als erstes Element in der jeweiligen Zeile gesetzt, während der Rest von links nach rechts folgt

In allen `*`-Deklarationen sind dabei zusammengehörige Daten sortiert gegeben, das heißt, das erste Element in `coordinate` gehört zum ersten Element in `procsets` und so weiter. In `coordinate` sind dabei die Koordinaten der Gitterpunkte gegeben. Die zugehörigen Prozessormengen sind in `procsets` zu finden. In `ind` ist die zugehörige Zeile in der Matrix A^p definiert, wobei eventuell die jeweiligen Freiheitsgrade `dof` additiv hinzukommen (das heißt, wenn `ind[0]=4` und `dof[0]=3`, so sind den Zeilen `4`, `5` und `6` der Gitterpunkt `coordinate[0]` zugeordnet, welcher auf den Prozessoren `procsets[0]` zu finden ist). Mit `numR` werden die Anzahl der gegebenen Gitterpunkte definiert. `p` und `NP` bezeichnen hier die eigene Prozessornummer beziehungsweise die Gesamtzahl an Prozessoren.

Zur Erstellung der Matrixblöcke mit gleichen Prozessormengen (vergleiche Definition 2.5) werden alle Elemente zuerst über die `procsets` als Einheit sortiert, zusammengefasst und daraufhin nach Koordinaten geordnet. Dafür sind insbesondere folgende Klassen zuständig:

```
class ProcSet: public vector<short> {
...}

class vecps {
    ProcSet ps;
    int             local_size;
    int             global_size;
    vector<int>     invIND;
    int             position;
...};

class vecProcSet {
    vector<vecps*> vecPS;
...};

class vecprocset {
    vecProcSet* vps;
...};
```

Die Klasse `vecprocset` dient nur als Referenz auf die Klasse `vecProcSet`. Dort wird eine Liste aller aktiven Prozessormengen $\Pi_{\mathcal{I}}$ (`vecPS`) (vergleiche (2.13)) erstellt. Eine Prozessormenge $\pi_i \in \Pi_{\mathcal{I}}$ wird dabei in `ProcSet` definiert. Für jede aktive Prozessormenge π_i existiert eine lokale Größe (`local_size`), sofern die eigene Prozessornummer p in π_i vorhanden ist, sowie eine globale Größe (`global_size`). Da die Elemente eine eigene Sortierung erhalten haben, wird in einem Vektor `invIND` die jeweilige Position (sozusagen die Zeile) in der lokalen Matrix A^p abgespeichert, so dass hierüber die Blockmatrizen A^p_{km} definiert werden können. In `position` wird letztendlich die globale Position von π_k in der (fiktiven) globalen Matrix A definiert, welches der Sortierung nach Definition 3.13 entspricht.

Da jeder Prozessor p nur seine eigenen Elemente und damit nur diejenigen aktiven Prozessormengen π_i mit $p \in \pi_i$ kennt, müssen die restlichen aktiven Prozessormengen über alle Prozessoren kommuniziert werden. Dies geschieht über ein `MPI_Allgatherv` (vergleiche Anhang B).

Nachdem die aktive Prozessormenge $\Pi_{\mathcal{I}} = \{\pi_i\}$ auf jedem Prozessor bekannt ist, können die Schritte s mit zugehörigen Operationsmengen $\mathcal{K}^{s,t}$ für jede Prozessormenge $\pi \in \Pi_{\mathcal{I}}$ nach (3.12) definiert werden. Dafür werden die kombinierten Prozessormengen nach Definition 3.10 erzeugt. Wir betrachten dazu folgende Klassen:

```
class ParSol_cluster_step {
    vecprocset& vps;                          // reference
    ProcSet combined_PS;                      // combined ProcSet
    ParComm* CommunicationModule;
    vector<int> Sol;                          // elements to solve
    vector<int> Schur;                        // elements in Schur
...};
class ParSol_one_step {
    vecprocset& vps;                          // reference
    int _t;                                   // own cluster number
    vector<ParSol_cluster_step* > cluster; // all clusters
...};
class ParSol_all_steps {
    vecprocset* vps;
    vector<ParSol_one_step* > s;
...};
```

Die Hauptklasse ist in diesem Fall `ParSol_all_steps`, in denen alle Schritte in einem Vektor `s` zusammengefasst werden. Hier wird auch die Klasse `vecprocset` erzeugt, auf denen die jeweiligen Unterklassen eine Referenz besitzen. Für jeden Schritt werden auf jedem Prozessor alle Cluster erzeugt und im Vektor `cluster` abgespeichert. Jeder Prozessor besitzt in jedem Schritt nur einen zugehörigen Cluster, welche durch die Variable `_t` beschrieben wird. Ein einzelner Cluster wird innerhalb der Klasse `ParSol_cluster_step` beschrieben und definiert die jeweilige kombinierte Prozessormenge `combined_PS` ($\mathcal{P}^{s,t}$ nach (3.11)), das zugehörige Kommunikationsmodul `CommunicationModule` für die `MPI`-Routinen, sowie die zu zerlegende Operationsmenge $\mathcal{K}_{\mathrm{LR}}^{s,t}$ in `Sol` und die zugehörige Schur-Komplement-Menge $\mathcal{K}_{\mathrm{Sch}}^{s,t}$ in `Schur` (vergleiche Definition 3.18).

Bez.	Variablenname	Typ	in Klasse
$\Pi_{\mathcal{I}}$	`vecPS`	`vector<vecps*>`	`vecProcSet`
π_k	`ps`	`ProcSet`	`vecps`
N_k	`global_size`	`int`	`vecps`
	`local_size`	`int`	`vecps`
t	`_t`	`int`	`ParSol_one_step`
$\mathcal{P}^{s,t}$	`combined_PS`	`ProcSet`	`ParSol_cluster_step`
$\mathcal{K}_{\mathrm{LR}}^{s,t}$	`Sol`	`vector<int>`	`ParSol_cluster_step`
$\mathcal{K}_{\mathrm{Sch}}^{s,t}$	`Schur`	`vector<int>`	`ParSol_cluster_step`

TABELLE 1. Bezeichnungen aus Kapitel 2 und 3 und entsprechende Variablennamen im Programm für die Grundstruktur

4.3. Matrixklassen

Somit sind alle Vorbereitungen zur Definition der Schritte getroffen und die jeweiligen Matrizen können erzeugt werden. In diesem Fall heißt das insbesondere, dass der benötigte Speicherplatz für alle lokalen Matrizen reserviert wird. Die eigentliche Besetzung wird dann in jedem Schritt vorgenommen. Dazu betrachten wir die lokale Matrix $A_{\mathrm{loc}}^{(s),t}$, welche wie in (3.18) in Teilmatrizen $Z^{(s),t}$, $R_{\mathrm{loc}}^{(s),t}$, $L_{\mathrm{loc}}^{(s),t}$ und $S_{\mathrm{loc}}^{(s),t}$ aufgeteilt ist. Die Größen der Teilmatrizen ergeben sich direkt aus den Einzelgrößen der aktiven Prozessormengen, welche in `Sol` beziehungsweise `Schur` in der Klasse `ParSol_cluster_step` gegeben sind. Diese Teilmatrizen müssen nun nach (3.24) weiter aufgeteilt werden. Dafür ist die Klasse `ParallelMatrix` zuständig:

```
struct pardec {
    int size;
    int proc;
    int shift;
};

class ParallelMatrix {
    ParComm& C;         // reference
    Scalar* a;          // local matrix
    int* IPIV;          // pivoting for Lapack
    int local_col;      // sum of all local_cols = col
    int local_row;      // could be 0, if no matrix is defined here
    int row;
    int col;
    vector <pardec> dec; // distribution

    ParComm* C_loc;

    void Create_Decomposition(int MINSIZE, int maxP);
...};
```

In `row` bzw. `col` wird die globale Größe der Zeilen bzw. Spalten der parallel verteilten Teilmatrix abgespeichert. Die Routine `Create_Decomposition` berechnet daraufhin die Anzahl der Spalten `local_col` auf den jeweiligen Prozessoren, so dass insgesamt ein Speicherbedarf von `row`×`local_col` für die lokale Teilmatrix `a` benötigt wird. Dabei kann zusätzlich eine Mindestgröße der Teilmatrizen (`MINSIZE`) beziehungsweise eine maximale Menge der beteiligten Prozessoren (`maxP`) angegeben werden. Der Vektor `dec` (definiert über `pardec`) dient dabei einer Aufzählung aller Teilmatrizen innerhalb der Prozessormenge $\mathcal{P}^{s,t}$. Ein Prozessor `proc` ($p_q^{s,t}$) besitzt in diesem Fall `size` Spalten (vergleiche (3.22) bzw. (3.23)) der lokalen Matrix, welche ab `shift` (dies entspricht $N_q^{s,t}$ aus (3.25)) beginnen. In der Klasse `ParallelMatrix` können auch einzelne Einträge von `a` gelesen oder geschrieben werden.

Um die lokale Matrix $A_{\text{loc}}^{(s),t}$ mit ihren Untermatrizen zu beschreiben, bedienen wir uns der Klasse `ParSol_Matrix`, mit folgendem Inhalt:

```
class ParSol_Matrix {    // Variablendeklaration
protected:
    const ParSol_cluster_step& cluster; // reference
    const ParSol_one_step& step;        // reference
    const vecprocset& vps;              // reference
    ParComm& C;                         // reference

    ParComm* C_loc;

    ParallelMatrix* A;       // matrix Z
    ParallelMatrix* Right;   // matrix R
    ParallelMatrix* Left;    // matrix L
    ParallelMatrix* Schur;   // matrix S

    Scalar* rhs;
    int size_rhs;
    int nrhs;
...};
```

In `cluster`, `step` und `C` sind Referenzen zum aktuellen Cluster, Schritt bzw. zur parallelen Schnittstelle zur Kommunikation für den aktuellen Schritt gegeben. Die einzelnen Matrizen von $A_{\text{loc}}^{(s),t}$ (das heißt $Z^{(s),t}$, $L_{\text{loc}}^{(s),t}$, $R_{\text{loc}}^{(s),t}$ und $S_{\text{loc}}^{(s),t}$) sind definiert über die vorher beschriebene Klasse `ParallelMatrix`. In `rhs` wird im Verlauf die zugehörige rechte Seite des Gleichungssystems gespeichert und mit der Lösung überschrieben. Dabei ist es möglich, direkt mehrere (`nrhs`) rechten Seiten zu lösen.

Für Schritt $s = 0$ wird zusätzlich ein Sparse-Matrix-Format zur Verfügung gestellt, so dass $Z_{\text{loc}}^{(0),t}$ und $L_{\text{loc}}^{(0),t}$ im Sparse-Format abgespeichert werden können:

```
class ParSol_Matrix_Sparse: public ParSol_Matrix {
    BasicSparseMatrix* SparseA;
    SparseSolver* S;
    BasicSparseMatrix* SparseLeft;
...};
```

Für den `SparseSolver` wird eine externe Routine benutzt, hier im Allgemeinen `SuperLU` oder `MUMPS`.

Bez.	Variablenname	Typ	in Klasse
$p_q^{s,t}$	`proc`	`int`	`pardec`
$N_{p_q^{s,t}}$	`size`	`int`	`pardec`
$N_q^{s,t}$	`shift`	`int`	`pardec`
$Z^{(s),t}$	`A`	`ParallelMatrix*`	`ParSol_Matrix`
$R_{\text{loc}}^{(s),t}$	`Right`	`ParallelMatrix*`	`ParSol_Matrix`
$L_{\text{loc}}^{(s),t}$	`Left`	`ParallelMatrix*`	`ParSol_Matrix`
$S_{\text{loc}}^{(s),t}$	`Schur`	`ParallelMatrix*`	`ParSol_Matrix`
f	`rhs`	`Scalar*`	`ParSol_Matrix`

TABELLE 2. Bezeichnungen aus Kapitel 2 und 3 und entsprechende Variablennamen im Programm für die Matrixklassen

4.4. Routinen der Matrixklasse

Die Hauptarbeit des Algorithmus erfolgt in der Klasse `ParSol_Matrix`, wobei hier einige Routinen näher beschrieben werden. Dazu betrachten wir zuerst die öffentlichen (`public`) Routinen, die zur Erstellung der parallelen Block-LR-Zerlegung notwendig sind:

```
class ParSol_Matrix {    // Zerlegungsroutinen
public:
    virtual void Set(const BasicSparseMatrix&);
    virtual void makeLU();
    void SetNext_Matrix(ParSol_Matrix&);
...};
```

Die Routine `Set(const BasicSparseMatrix&)` ist nur für den ersten Schritt $s = 0$ notwendig. Dort wird aus der vorhandenen lokalen Matrix A^p auf jedem Prozessor die zugehörige Matrix $A_{\text{loc}}^{(0),p}$ mit Hilfe der Klasse `VecProcSet` erstellt. In dem Fall, dass ein externer Sparse-Löser angeschlossen ist, werden `A` und `Left` aus der Oberklasse `ParSol_Matrix` nicht besetzt, sondern direkt die Matrizen `SparseA` und `SparseLeft` im Sparse-Format erstellt.

In den beiden anderen Routinen `makeLU` und `SetNext_Matrix` wird die grundlegende Zerlegung geleistet, so dass sich hier eine genauere Betrachtung lohnt. Im

Fall $s = 0$ und dem Anschluss eines Sparse-Lösers wird die Matrix `SparseA` mit Hilfe des `SparseSolver` zerlegt und auf `Right` angewandt. Daraufhin wird das lokale Schur-Komplement `Schur` aktualisiert.

Für alle anderen Schritte $s > 0$ oder wenn kein zusätzlicher Sparse-Löser vorhanden ist, wird folgende Routine benutzt, welche Algorithmus 3.28 beschreibt:

```
void ParSol_Matrix::makeLU() {
    int decsize = A->get_dec_size();
    for (int i=0; i<decsize; ++i) {
        A->makeLU(i);
        if (C_loc->proc() == C.proc())
            A->Communicate_Column_intern(i,C_loc);
        pardec dec = A->get_dec(i);

        if (a && C.proc() > i) {
            A->Solve_intern(i);
            A->MMM_intern(i);
        }
        if (Schur->rows() != 0) {
            if (C_loc->proc() == C.proc())
                Left->Communicate_Column_intern(i, C_loc, false);
            Scalar* Left_column = Left->get_other_column();
            if (left && C.proc() > i)
                Left->MMM_left(a+dec.shift,A->rows(),dec);
            if (right) {
                A->Solve_right(Right->ref(),Right->cols_loc(),dec);
                Right->MMM_right(A->get_other_column(),dec);
                Schur->MMM_schur(Left->get_other_column(),
                                 Right->ref(),dec.shift);
            }
        }
    }
}
```

Verglichen mit Algorithmus 3.28 beschreibt hierbei `decsize` die Größe $Q^{s,t}$. Die Kommunikation geschieht über die Funktion `Communicate_Column_intern`, sowohl für `A` ($Z_q^{(s),t}$) als auch `Left` ($L_q^{(s),t}$). In den Funktionen `MMM` wird eine Matrix-Matrix-Multiplikation mit zugehöriger Subtraktion analog zu Algorithmus 3.28 durchgeführt.

Die in Kapitel 3.6.4 angekündigte Betrachtung der Besetzung der Matrix $A_{\text{loc}}^{(s),t}$ durch die Addition zweier Matrizen aus dem vorigen Schritt $s-1$ findet in der Funktion `SetNext_Matrix` statt. Hier werden nacheinander die zwei Schur-Matrizen $S_{\text{loc}}^{(s-1),t}$ der beiden Cluster aus Schritt $s-1$ in die lokale Matrix $A_{\text{loc}}^{(s),t}$ zusammengefasst. Welcher Cluster seine Matrix kopieren muss wird durch `current_cluster` beschrieben. Die Variable wechselt im Verlauf die Referenz vom ersten zum zweiten Cluster. Da jeder Prozessor nur Kenntnisse über die Zerlegung der Matrix $S_{\text{loc}}^{(s-1),t}$ in seinem eigenen Cluster hat, werden mit Hilfe von `Communicate_other_dec` die nötigen Informationen zwischen den Prozessoren der beiden Cluster ausgetauscht. In der Funktion `SetNext_Matrix_LEFT_CLUSTERED` werden die beiden Matrizen $Z^{(s),t}$ und $L_{\text{loc}}^{(s),t}$ und in der Funktion `SetNext_Matrix_RIGHT_CLUSTERED` werden die Matrizen $R_{\text{loc}}^{(s),t}$ und $S_{\text{loc}}^{(s),t}$ besetzt.

```
void ParSol_Matrix::SetNext_Matrix(ParSol_Matrix& next) {
...
    vector <pardec> other_dec;
    Communicate_other_dec(next,other_dec,shift_own_proc,
                          shift_other_proc,other_cluster_number);

// set current_cluster #1  {...}

    SetNext_Matrix_LEFT_CLUSTERED(next, current_cluster,
                                  current_dec, current_proc_shift);
    SetNext_Matrix_RIGHT_CLUSTERED(next, current_cluster,
                                   current_dec, current_proc_shift);

// set current_cluster #2  {...}

    SetNext_Matrix_LEFT_CLUSTERED(next, current_cluster,
                                  current_dec, current_proc_shift);
    SetNext_Matrix_RIGHT_CLUSTERED(next, current_cluster,
                                   current_dec, current_proc_shift);
    Schur->Clear();        // delete Schurmatrix
}
```

Die Funktion `SetNext_Matrix_LEFT_CLUSTERED` und die zugehörigen Folgefunktionen gehen nun folgendermaßen vor: Belegt werden sollen die Matrizen $Z^{s,t}$ und $L^{s,t}_{\text{loc}}$ aus dem Schur-Komplement $S^{s-1,t}_{\text{loc}}$ (vergleiche dazu auch Lemma 3.19 und Bemerkung 3.20). Jede komplette Spalte der einzelnen Matrizen ist auf genau einem Prozessor zu finden – verschiedene Spalten können aber auf verschiedenen Prozessoren gegeben sein. Weiterhin befindet sich die i-te Spalte der Matrix $Z^{s,t}$ auf dem gleichen Prozessor wie die i-te Spalte der Matrix $L^{s,t}_{\text{loc}}$. Somit kann jeweils eine komplette Spalte von $S^{s-1,t}_{\text{loc}}$ zum zugehörigen Prozessor von Schritt s kommuniziert werden. Die Spalten definieren sich über $\mathcal{K}^{s,t}_{\text{LR}}$. Die Elemente $k \in \mathcal{K}^{s,t}_{\text{LR}}$ sind in allen Mengen aufsteigend sortiert, außerdem sind die Elemente der Indexmenge $\mathcal{I}_k$, welche durch π_k definiert sind und jeweils einer Spalte der Gesamtmatrix zugehörig sind, ebenso sortiert gegeben. Weiterhin sind alle Spalten einer Matrix, welche zu einem Prozessor gehören, aufeinanderfolgend gegeben. Somit können mehrere aufeinanderfolgende Spalten zusammengefasst und versendet werden, sofern sie sich jeweils auf dem gleichen Prozessor befinden. Damit reduziert sich der Kommunikationsaufwand erheblich.

```
void ParSol_Matrix::SetNext_Matrix_LEFT_CLUSTERED(
                ParSol_Matrix& next,
                const ParSol_cluster_step* current_cluster,
                const vector<pardec>* current_dec,
                int& current_proc_shift) {

// find consecutive active processor sets  {...}
    set_sr_dec_LEFT(next, current_cluster, current_dec,
                    current_proc_shift, shift_send, size,
                    shift_receive);
}
```

Hierbei werden zuerst aktive Prozessormengen π_k gesucht, die jeweils zusammenhängend in $\mathcal{K}^{s-1,t}_{\text{SCH}}$ und $\mathcal{K}^{s,t}_{\text{LR}}$ zu finden sind. Diese Information erhält die Funktion `set_sr_dec_LEFT`, welche nun zusammenhängende Spalten sucht, für die nur

eine Kommunikation nötig ist. Die Informationen für alle benötigten Kommunikationen werden im Vektor `sending_dec` für die sendenden Prozessoren und im Vektor `receiving_dec` für die empfangenden Prozessoren gespeichert. Falls der Sender und Empfänger gleich sind, muss keine "echte" Kommunikation stattfinden, so dass die entsprechenden Daten einfach über eine `COPY`-Funktion kopiert werden können. Ansonsten werden die betreffenden Daten über eine `SEND`-Funktion versendet und über eine `RECEIVE`-Funktion empfangen. Daraufhin müssen die Daten noch richtig einsortiert werden. Dafür ist die Funktion `Set_LEFT` zuständig. Dort werden die Zeilenpositionen der Spaltenelemente in den neuen Matrizen $Z^{s,t}$ (`next.A`) beziehungsweise $L_{\text{loc}}^{s,t}$ (`next.Left`) bestimmt und addiert.

```
void ParSol_Matrix::set_sr_dec_LEFT(
            ParSol_Matrix& next,
            const ParSol_cluster_step* current_cluster,
            const vector<pardec>* current_dec,
            int& proc_shift,
            int& shift_send,
            int& size,
            int& shift_receive) {

  vector<pardec> sending_dec;
  vector<pardec> receiving_dec;

// find consecutive columns for sending and receiving processors
// save the information in sending_dec and receiving_dec. {...}

  for (int i = 0; i < sr_size; ++i) {
    if (sending_dec[i].proc == receiving_dec[i].proc &&
        sending_dec[i].proc == next.C.proc()) {
      COPY(Schur->ref()+rows*sending_dec[i].shift,
           sending_dec[i].size,rows,tmp);
      Set_LEFT(next, current_cluster, tmp, rows,
               receiving_dec[i].size, receiving_dec[i].shift);
    }
    else if (next.C.proc() == sending_dec[i].proc)
      SEND(next.C,Schur->ref()+Schur->rows()*sending_dec[i].shift,
           sending_dec[i].size,rows,receiving_dec[i].proc);
    else if (next.C.proc() == receiving_dec[i].proc) {
      RECEIVE(next.C,tmp,sending_dec[i].size,rows,
              sending_dec[i].proc);
      Set_LEFT(next, current_cluster, tmp, rows,
               receiving_dec[i].size, receiving_dec[i].shift);
    }
  }
}
```

Für die Funktion `SetNext_Matrix_RIGHT_CLUSTERED` gilt Analoges, nur dass für die Matrizen $R_{\text{loc}}^{s,t}$ und $S_{\text{loc}}^{s,t}$ nun die Elemente von $\mathcal{K}_{\text{SCH}}^{s,t}$ betrachtet werden.

Das Schurkomplement $S_{\text{loc}}^{s-1,t}$ wird im Folgenden nicht mehr benötigt – alle Informationen sind in der Matrix $A_{\text{loc}}^{s,t}$ zu finden – und kann somit gelöscht werden, um Speicherplatz zu sparen.

4.5. Lösungsroutinen

Für die Implementierung der Lösungsroutinen betrachten wir zunächst wieder die zugehörigen Funktionen in der Klasse `ParSol_Matrix`.

Im Gegensatz zur Matrix-Verteilung sind die rechten Seiten nach jedem Schritt innerhalb einer Prozessormenge konsistent, das heißt auf allen Prozessoren gleich.

```
class ParSol_Matrix {    // Lösungsroutinen
public:
    void Set_rhs(const Scalar* u);
    void Set_rhs(const vector<const Scalar*> u);

    void Write_rhs(Scalar* u);
    void Write_rhs(vector<Scalar*> u);

    virtual void SolveL();
    virtual void SolveU();

    void SetNext_rhs_LEFT(ParSol_Matrix& next);
    void SetNext_rhs_RIGHT(ParSol_Matrix& next);
...}
```

Über `Set_rhs` wird die rechte Seite in diesem Löser aufgebaut. Dies geschieht ähnlich zur Besetzung der Matrix $A_{\text{loc}}^{(0),p}$ mit Hilfe der Klasse `VecProcSet`. Mit `Write_rhs` wird die gelöste rechte Seite wieder in das ursprüngliche Format umsortiert. Hierbei können auch mehrere rechte Seiten gleichzeitig gesetzt und geschrieben werden (`vector<Scalar*>`).

`SolveL` beschreibt die Vorwärts-Substitution auf dem jeweiligen Cluster (Algorithmus 3.37) ohne Besetzung der Teilvektoren $f_{\text{loc}}^{(s),t}$ in Zeile 3-7.

```
void ParSol_Matrix::SolveL() {
    for (int i = 0; i < decsize; ++i) {
        if (C.proc() == i) {
            A->Solve(i,rhs,nrhs,size_rhs);
            if (i != decsize-1)
                A->MV_rhs(i, rhs, nrhs,size_rhs)
            Left->MV_rhs_left(i, A->rows(), rhs, nrhs,size_rhs);
            if (i < decsize-1)
                SEND(C,rhs,nrhs,size_rhs,i+1);
        }
        if (i < decsize-1 && C.proc() == i+1)
            RECEIVE(C,rhs,nrhs,size_rhs,i);
    }
    if (C.proc() == C_loc->proc())
        C_loc->Broadcast(rhs,size_rhs*nrhs,SCALAR,decsize-1);
}
```

In den `MV_rhs`-Routinen werden dabei die Matrix-Vektor-Multiplikation mit anschließender Subtraktion durchgeführt (vergleiche Zeile 12,13 im Algorithmus der Vorwärts-Substitution).

In `SolveU` wird dementsprechend die Rückwärts-Substitution (Algorithmus 3.38) durchgeführt, wiederum ohne die Besetzung der Teilvektoren im nächsten Schritt (Zeile 18-24). Um die Summation aus Zeile 7 zu verwirklichen, wird dabei auf jedem bis auf einem Prozessor die rechte Seite gleich 0 gesetzt. Somit ist die aktuelle rechte Seite auf genau einem Prozessor vorhanden. Daraufhin wird die Matrix-Vektor-Multiplikation mit anschließender Subtraktion durchgeführt und alle Daten

mit Hilfe einer MPI-Routine auf allen Prozessoren aufsummiert (`C.AllSum(rhs,row);`)

```
void ParSol_Matrix::SolveU() {
    int decsize = Right->get_dec_size();
    if (decsize > 0) {
        if (C.proc() != decsize-1)
            for (int K=0; K<nrhs; ++K)
                for (int i=0; i< row; ++i)
                    rhs[K*size_rhs+i] = 0;
        Right->MV_rhs_right(C.proc(),rhs,nrhs)
        for (int K=0; K<nrhs; ++K)
            C.AllSum(rhs+K*size_rhs,row);
    }

    decsize = A->get_dec_size();
    if (decsize<=1) {
        C.Broadcast(rhs,size_rhs*nrhs,LIB_PS_SCALAR,0);
        return;
    }
    for (int i = decsize-1; i >= 1; --i) {
        if (C.proc() == i) {
          A->MV_rhs_Z(i,rhs,nrhs);
          if (i > 1)
              SEND(C,rhs,nrhs,size_rhs,i-1);
        }
        if (i > 1 && C.proc() == i-1)
            RECEIVE(C,rhs,nrhs,size_rhs,i);
    }
    if (decsize > 1)
        C.Broadcast(rhs,size_rhs*nrhs,SCALAR,1);
}
```

Die Besetzung der Teilvektoren findet in `SetNext_rhs_LEFT` für die Vorwärts-Substitution beziehungsweise `SetNext_rhs_RIGHT` für die Rückwärts-Substitution statt und wird ähnlich wie in `SetNext_Matrix` durchgeführt. Dabei müssen in `SetNext_rhs_LEFT` wieder zwei Vektoren von verschiedenen Prozessormengen zusammengefasst werden. Bei der Rückwärts-Substitution befinden sich die Einträge von $f_k^{(s),t}$ (Zeile 22 in Algorithmus 3.38) bereits auf allen beteiligten Prozessoren, so dass diese für den nächsten Schritt nur richtig einsortiert werden müssen und daher keine zusätzliche Kommunikation benötigt wird.

Es existieren noch eine Vielzahl weiterer Klassen und Routinen[6], die hier beschriebenen verrichten jedoch die Hauptarbeit der Zerlegung beziehungsweise der Anwendung auf die rechten Seiten.

[6]Die Endversion der parallelen Block-LR-Zerlegung besitzt mehr als 3000 Zeilen Code

5 Komplexitätsanalyse für ein Laplace-Problem

Im Allgemeinen benötigt eine LR-Zerlegung einer $N \times N$-Matrix $O(N^3)$ Operationen [GL96, Kapitel 3.2]. Im Finite Elemente-Fall ist die zugehörige Matrix dünnbesetzt (sogenannte Sparse-Matrix), so dass hier – je nach Art des Problems – weitaus weniger Operationen nötig sind. In diesem Fall wird ein Sparse-Löser wie zum Beispiel `SuperLU` eingesetzt, der die Dünnbesetztheit beachtet [DEG+99, GL81]. Für eine Zerlegung einer $N \times N$-Matrix mit Bandbreite b werden dabei $O(Nb^2)$ Operationen benötigt. Weiterhin stehen hier $P = 2^S$ Prozessoren zur Verfügung, auf denen das Gesamtproblem aufgeteilt wird, so dass mit Hilfe von Parallelisierungen viele Operationen gleichzeitig durchgeführt werden, so dass insgesamt weniger Zeit zur Berechnung benötigt wird.

Für eine Komplexitätsanalyse von Algorithmus 3.30 betrachten wir ein einfaches Modell im Dreidimensionalen. Das Gebiet soll beschrieben werden durch einen Würfel $\Omega = (0,1)^3$, der in gleichmäßige Hexaeder unterteilt ist. Diese sollen gleichmäßig auf $P = 2^S$ Prozessoren verteilt sein. Auf diesem Würfel soll ein Laplace-Problem

$$-\Delta u = f \quad \text{in } \Omega = (0,1)^3$$

mit homogenen Dirichlet-Randbedingungen auf $\partial\Omega$ gelöst werden (siehe auch Kapitel 6.1). Dies geschieht mit linearen Ansatzfunktionen. Die Knotenpunkte befinden sich dabei an den Ecken der Hexaeder. Für die Zerlegung selbst ist die rechte Seite f nicht relevant. Im Prinzip ist auch die Art des Problems nicht von Bedeutung, im Falle des Laplace-Problems erhalten wir jedoch eine symmetrisch positiv definite Matrix. Somit kann der gesamte Algorithmus ohne Pivotisierung durchgeführt werden und die Existenz einer LR-Zerlegung ist gesichert (vergleiche auch Kapitel 3.9.1).

5.1. Übersicht der Variablen und Aufwandsabschätzungen

Da in diesem Kapitel viele Variablen eingeführt werden, stellen wir eine Kurzbeschreibung für eine übersichtliche Darstellung hier vor. Die Seitenangabe bezieht sich hierbei auf das erstmalige Erscheinen der jeweiligen Variable. Dort findet sich gewöhnlich auch eine kurze Erklärung im Text zur Bedeutung.

l — Seite 77
Verfeinerungslevel; Anzahl der Verfeinerungen des Startgebiets

l_0 — Seite 79
Verfeinerungslevel für den ersten Schritt

$N = (2^l + 1)^3$ — Seite 77
Anzahl der Unbekannten im Gesamtgebiet; Matrixgröße von A

N_0 — Seite 79
Anzahl der Unbekannten in Schritt $s = 0$ auf jedem Cluster

$n = \sqrt[3]{N} = 2^l + 1$ — Seite 78
Anzahl der Unbekannten des Gesamtgebiets auf einer Kante

$n_l = 2^l + 1$ — Seite 78
Anzahl der Unbekannten auf einer Kante auf Level l oder nach einem Schnitt (dabei ist l als variabel anzusehen)

n_{l_0} — Seite 79
Anzahl der Unbekannten auf einer Kante auf Level l_0; Anzahl der Unbekannten auf einer Kante in Schritt $s = 0$ auf jedem Cluster

N^s_{flops} — Seite 87
Anzahl der flops für die Berechnungen in Schritt s

$T_s = 2^{S-s}$ — Seite 78
Anzahl der Gebiete bzw. Cluster in Schritt s

$T_{\text{ber},s}$ — Seite 87
Abschätzung der Berechnungszeit für Schritt s

$T_{\text{bc},s}$ — Seite 87
Abschätzung der Broadcast-Zeit für Schritt s

$T^o_{\text{ber},s}$ — Seite 87
gemessene/tatsächliche Berechnungszeit in Schritt s

$T^o_{\text{bc},s}$ — Seite 87
gemessene/tatsächliche Broadcast-Zeit in Schritt s

z_s — Seite 77
Größe der lokalen Matrix $Z^{(s),t}$; Anzahl der in diesem Schritt zu lösenden Unbekannten innerhalb eines Clusters

i_s — Seite 77
Größe der lokalen Matrix $S^{(s),t}_{\text{loc}}$; Anzahl der in diesem Schritt auftretenden Schurelemente innerhalb eines Clusters

$P_s = 2^s$ — Seite 81
Anzahl der Prozessoren je Cluster in Schritt s

$m_s = \frac{z_s}{P_s}$ — Seite 81
Anzahl der Spalten von $Z^{(s),t}$ und $L^{(s),t}_{\text{loc}}$ auf einem Prozessor in Schritt s

$r_s = \frac{i_s}{P_s}$ — Seite 82
Anzahl der Spalten von $R^{(s),t}_{\text{loc}}$ und $S^{(s),t}_{\text{loc}}$ auf einem Prozessor in Schritt s

Der Aufwand der Operationen wird in der O-Notation angegeben. Hierbei werden die folgende Abschätzungen eingeführt. Die Zeilenzahl bezieht sich dabei auf die Angabe der Zeile in Algorithmus 3.30.

C^0_{LR} (Zeile 11) — Seite 81
LR-Zerlegung mit Hilfe eines Sparse-Lösers der Matrix $Z^{(0),t}$

$C^0_{\text{Solve:}R}$ (Zeile 21) — Seite 81
Anwendung der mit Hilfe des Sparse-Lösers zerlegten Matrix $Z^{(0),t}$ auf die Matrix $R^{(0),t}_{\text{loc}}$

$C_{\text{MM:}S}$ (Zeile 23) — Seite 81
Matrix-Matrix-Multiplikation mit Subtraktion zur Berechnung des Schur-Komplements

C^s_{LR} (Zeile 11) — Seite 81
LR-Zerlegung eines Diagonalblocks von $Z^{(s),t}$ für $s > 0$

C^s_{Solve} (Zeile 17) — Seite 81
Anwendung der LR-Zerlegung des Diagonalblocks auf $Z^{(s),t}$

$C^s_{\text{Solve:}R}$ (Zeile 21) — Seite 82
Anwendung der LR-Zerlegung des Diagonalblocks auf $R^{(s),t}_{\text{loc}}$

C^s_{MM} (Zeile 18) — Seite 82
Matrix-Matrix-Multiplikation mit Subtraktion zur Aktualisierung von $Z^{(s),t}$

$C^s_{\text{MM:}S}$ (Zeile 23) — Seite 82
Matrix-Matrix-Multiplikation mit Subtraktion zur Aktualisierung von $R^{(s),t}_{\text{loc}}$

$C^s_{\text{MM:}R}$ (Zeile 22) — Seite 82
Matrix-Matrix-Multiplikation mit Subtraktion zur Aktualisierung von $S^{(s),t}_{\text{loc}}$

C_{ber} — Seite 83
asymptotische Gesamtkomplexität des Rechenaufwands

C_{bc} — Seite 84
asymptotische Gesamtkomplexität der Broadcast-Kommunikation

5.2. Diskretisierung und Verteilung auf die Prozessoren

Zur Bestimmung der Gesamtfreiheitsgrade N betrachten wir die Diskretisierung des Einheitswürfels $\Omega = (0,1)^3$ in Hexaeder. Der Feinheitsgrad der Diskretisierung wird dabei über ein Verfeinerungslevel l angegeben, wobei für $l = 0$ das Ursprungsproblem und damit der Einheitswürfel selbst als Starthexaeder gegeben ist. Eine Verfeinerung eines Hexaeders ergibt 8 neue Hexaeder, wobei die Knotenpunkte auf den Ecken der Hexaeder gegeben sind. Somit ergibt sich folgende Anzahl an Hexaedern auf Level l mit gesamten Knotenpunkten N.

Level l	#Hexaeder	N
	8^l	$(2^l+1)^3$
0	1	8
1	8	27
2	64	125
3	512	729
4	4 096	4 913
5	32 768	35 937
6	262 144	274 625
7	2 097 152	2 146 689

TABELLE 1. Anzahl an Hexaedern und Freiheitsgraden N für einen Würfel auf Verfeinerungslevel l

Um die Hexaeder auf die Prozessoren zu verteilen, verwenden wir den RCB-Algorithmus (recursive coordinate bisection) [**Wil91**]. Dabei wird das Gebiet nacheinander in x-, y- und z-Richtung wiederholt halbiert und jeweils auf die Prozessoren verteilt (siehe auch Nested Dissection in Kapitel 2.3). Eine Halbierung in x-Richtung definiert also einen Schnitt in der y/z-Ebene. Da eine Verfeinerung des Hexaeders ebenso in allen drei Dimensionen halbiert, ergibt sich somit eine gleichmäßige Verteilung auf die Prozessoren. Wenn zudem die Anzahl der Prozessoren eine Potenz von 8 ist, erhält jeder Prozessor einen (kleinen) Würfel des Gesamtgebiets.

5.3. Bestimmung der Matrixgrößen

Für die Komplexitätsanalyse ist es wichtig zu wissen, wie viele innere Elemente z_s (dies definiert die Größe von $Z^{(s),t}$) und Interface-Elemente i_s (dies definiert die Größe von $S^{(s),t}_{\text{loc}}$) in jedem Schritt auf den Prozessoren auftreten. Dazu betrachten wir die auftretenden Mengen durch die Schnitte des RCB-Algorithmus. Während

der Würfel dreidimensional ist, wird durch einen Schnitt ein zweidimensionales Gebilde erzeugt. Das heißt, alle Interfaces sind eine Dimension niedriger als die Ursprungsdimension. Es bietet sich an, die Anzahl der Unbekannten auf einer Kante des Würfels mit $n = n_l = \sqrt[3]{N} = 2^l + 1$ zu bezeichnen. Ein Schnitt einer Kante der Länge $n_l = 2^l + 1$ ergibt $n_{l-1} = 2^{l-1} + 1$ Unbekannte für jeden Teilschnitt, wobei ein Element als Interface auf beiden Teilschnitten zu finden ist.

Wir betrachten den Algorithmus aus zwei Richtungen. Die erste Richtung von Schritt $s = S$ abwärts zeigt die Zerlegung des Ursprungswürfels in Teilgebiete durch Schnitte des RCB-Algorithmus. Die zweite Richtung beginnt von $s = 0$ und zeigt die Zusammenführung der Teilgebiete in größere Gebiete für den nächsten Schritt $s + 1$. Die verschiedenen Betrachtungsweisen haben den Sinn, da am Ende des Algorithmus der LR-Zerlegung keine Interface-Elemente am Rand des Würfels existieren, während es am Anfang Teilgebiete gibt, die am gesamten Rand auch nach mehrfachem Zusammenführen Interface-Elemente besitzen.

Wir betrachten zuerst die letzten drei Schritte des Algorithmus von $s = S$ abwärts. Dies entspricht der Darstellung des RCB-Algorithmus, da die LR-Zerlegung den umgekehrten Weg nimmt. Mit $T_s = 2^{S-s}$ bezeichnen wir die Anzahl der Gebiete vor jedem Schnitt, das entspricht der Anzahl der Cluster in Schritt s. Weiterhin wird mit $N_s = n_x \times n_y \times n_z$ die Anzahl der Unbekannten auf einem Cluster in x-, y- und z-Richtung angegeben.

Zu Beginn existieren $T_S = 1$ Cluster mit

$$N_S = n_l \times n_l \times n_l$$

Unbekannten. Die erste Halbierung durch die Mitte des Würfels in x-Richtung ergibt einen Schnitt durch $1 \times n_l \times n_l = n^2$ Unbekannte. Diese werden im letzten Schritt S gelöst und beschreiben somit die Anzahl der Unbekannten der Matrix $Z^{(S),0}$. Da im letzten Schritt keine Interface-Elemente auftreten ist die zugehörige Matrix $S_{\text{loc}}^{(S),0}$ nicht besetzt. Somit gilt in Schritt S

$$\begin{aligned} z_S &= n^2 \\ i_S &= 0\,. \end{aligned}$$

Durch diesen Schnitt entstehen $T_{S-1} = 2$ Cluster mit

$$N_{S-1} = n_{l-1} \times n_l \times n_l = n^3/2 + O(n^2)$$

Unbekannten. Im folgenden Schritt wird das gesamte Gebiet und somit jedes der zwei Teilgebiete in y-Richtung halbiert. Somit wird der Schnitt durch die x/z-Ebene beschrieben Die Unbekannten von Schritt S finden sich nun in den Interface-Elementen wieder, das heißt

$$i_{S-1} = n^2\,.$$

Durch den Schnitt entstehen

$$z_{S-1} = (n_{l-1} - 1) \times 1 \times n_l = n^2/2 + O(n)$$

"neue" innere Elemente. Die -1 im Ausdruck der x-Richtung folgt daher, dass ein Teil dieser Elemente schon in den Interface-Elementen berücksichtigt wurden. Damit ergeben sich

$$\begin{aligned} T_{S-2} &= 2^2 \\ N_{S-2} &= n_{l-1} \times n_{l-1} \times n_l = n^3/4 + O(n^2) \end{aligned}$$

Cluster bzw. Unbekannte für Schritt $S-2$. Eine Halbierung in z-Richtung ergibt die inneren Elemente bzw. Interfaces

$$z_{S-2} = (n_{l-1}-1) \times (n_{l-1}-1) \times 1 = n^2/4 + O(n)$$
$$i_{S-2} = n^2\,,$$

so dass sich

$$T_{S-3} = 2^3$$
$$N_{S-3} = n_{l-1} \times n_{l-1} \times n_{l-1} = n^3/8 + O(n^2)$$

Cluster bzw. Unbekannte pro Cluster in Schritt $S-3$ befinden. Rekursiv kann das Verfahren fortgeführt werden, wobei nun wieder in x-Richtung geteilt wird.

Im von uns später betrachteten Fall auf Level $l = 7$ erhalten wir somit für die letzten drei Schritte die in Tabelle 2 angegebenen Größen der Matrizen $Z^{(s),t}$ (z_s) beziehungsweise $S_{\text{loc}}^{(s),t}$ (i_s).

Level $l = 7$		$n = 2^l + 1 = 129$	
Schritt s	z_s	i_s	T_s
$s = S$	16641	0	1
$s = S-1$	8256	16641	2
$s = S-2$	4096	16641	4

TABELLE 2. Matrixgrößen (z_s bzw. i_s) und Anzahl der Cluster (T_s) der letzten drei Schritte auf Level 7

Bisher ergab jeder Schnitt eine gleichmäßige Verteilung für jeden Cluster, da jeder Cluster jeweils eine Ecke des Ursprungswürfels besitzt. Eine weiterer Schnitt ergibt eine unterschiedliche Anzahl an inneren und Interface-Elementen, wobei letztendlich nur die Maximalzahl jedes Clusters wichtig ist. Aus dem Grund betrachten wir nun die ersten Schritte des Algorithmus, beginnend von Schritt $s = 0$. Der Einfachheit halber nehmen wir an, dass wir eine Potenz von 8 an Prozessoren zur Verfügung haben (das heißt S ist durch 3 teilbar und es gilt $P = 8^{S/3}$). Insbesondere hat dann jeder Prozessor einen Würfel als Gebiet zugeteilt bekommen. Im ersten Schritt ist die Anzahl der Cluster gleich der Anzahl der Prozessoren, also $T_0 = 2^S$. Jeder Cluster besitzt somit

$$N_0 = n_{l_0} \times n_{l_0} \times n_{l_0} = n^3/P + O(n^2)$$

Unbekannte mit $l_0 = l - S/3$[1]. Davon sind

$$z_0 = (n_{l_0}-2) \times (n_{l_0}-2) \times (n_{l_0}-2) = n^3/P + O(n^2)$$

innere Elemente und

$$i_0 = N_0 - z_0 = 6 \cdot (n^3/P)^{2/3} + O(n)$$

Interface-Elemente, da wir davon ausgehen, dass die gesamte Oberfläche aus Interface-Elementen besteht. Dabei beschreibt der Term $(n^3/P)^{1/3}$ eine Kante des Teilwürfels. Weiterhin beschreiben die inneren Elemente im ersten Schritt – im Gegensatz zu allen weiteren Schritten – kein echtes Interface, so dass dieser Fall

[1]Dies entspricht der Anzahl der Verfeinerungen, bevor das Gebiet auf die Prozessoren verteilt wird, vergleiche Kapitel 4, dort lautet die zugehörige Konfigurationsvariable `plevel`

gesondert betrachtet werden muss (Kapitel 5.4). Mit der Zusammenführung zweier in x-Richtung[2] nebeneinander liegenden Cluster, entsteht ein doppelter Würfel mit

$$\begin{aligned} N_1 &= n_{l_1} \times n_{l_0} \times n_{l_0} = 2n^3/P + O(n^2) \\ z_1 &= (n^3/P)^{2/3} + O(n) \\ i_1 &= 10 \cdot (n^3/P)^{2/3} + O(n)\,, \end{aligned}$$

wobei $l_1 = l_0 + 1$. Eine Erweiterung in y-Richtung ergibt

$$\begin{aligned} N_2 &= n_{l_1} \times n_{l_1} \times n_{l_0} = 4n^3/P + O(n^2) \\ z_2 &= 2 \cdot (n^3/P)^{2/3} + O(n) \\ i_2 &= 16 \cdot (n^3/P)^{2/3} + O(n) \end{aligned}$$

und eine letzte Erweiterung in z-Richtung

$$\begin{aligned} N_3 &= n_{l_1} \times n_{l_1} \times n_{l_1} = 8n^3/P + O(n^2) \\ z_3 &= 4 \cdot (n^3/P)^{2/3} + O(n) \\ i_3 &= 24 \cdot (n^3/P)^{2/3} + O(n)\,. \end{aligned}$$

Für die ersten drei Schritte erhalten wir mit $l_0 = 4$ und $l = 7$ (dies entspricht der Aufteilung von Ω mit Verfeinerungslevel 7 auf 512 Prozessoren)[3] die Angaben in Tabelle 3. Hierbei geben wir die gerundeten Berechnungen mit n in der am höchsten vorkommenden Ordnung an (in diesem Fall also meist $(n^3/P)^{2/3}$ mit dem jeweiligen Faktor).

$N = 2\,146\,689$
$P = 512 \quad n = 129$
$(n^3/P)^{2/3} \approx 260$

Schritt s	z_s	i_s
$s = 0$	4193	1560
$s = 1$	260	2600
$s = 2$	520	4160
$s = 3$	1040	6240

TABELLE 3. Matrixgrößen (z_s bzw. i_s) der ersten vier Schritte auf Level 7 auf 512 Prozessoren (nach der Abschätzung)

5.4. Analyse für Schritt $s = 0$

Der erste Schritt $s = 0$ muss gesondert betrachtet werden, da hier noch keine Zerlegung erfolgt ist und somit die gesamte Matrix in diesem Fall dünnbesetzt ist. Insbesondere gilt dies für die Matrix $Z^{(0),p}$ und $L^{(0),p}_{\text{loc}}$ in der lokalen Matrix (nach (3.18))

$$A^{(0),p}_{\text{loc}} = \begin{pmatrix} Z^{(0),p} & R^{(0),p}_{\text{loc}} \\ L^{(0),p}_{\text{loc}} & S^{(0),p}_{\text{loc}} \end{pmatrix}.$$

Zur Berechnung des Schur-Komplements

$$\hat{S}^{(0),p}_{\text{loc}} = S^{(0),p}_{\text{loc}} - L^{(0),p}_{\text{loc}} (Z^{(0),p})^{-1} R^{(0),p}_{\text{loc}}$$

[2]Es kann natürlich auch jede beliebige andere Richtung zuerst gewählt werden
[3]Die Angaben in Tabelle 8 unterscheiden sich geringfügig, da dort viele Würfel auch am Rand liegen

wird die Matrix $Z^{(0),p}$ mit Hilfe eines Sparse-Lösers zerlegt. Die Sparse-Matrix $Z^{(0),p}$ hat Bandbreite $O\left((n^3/P)^{2/3}\right)$ und die Gesamtgröße $N_0 = n^3/P$. Somit werden wegen $N = n^3$

$$C^0_{\mathrm{LR}} = O\left(\left(\frac{N}{P}\right)^{7/3}\right) \tag{5.1}$$

Operationen zur Zerlegung dieser Matrix benötigt. Die Zerlegung von $Z^{(0),p}$ wird daraufhin auf $R^{(0),p}_{\mathrm{loc}}$ angewandt. Die Matrix $R^{(0),p}_{\mathrm{loc}}$ besitzt die Größe

$$z_0 \times i_0 = N/P \times 6 \cdot (N/P)^{2/3}\,.$$

Es müssen also $6 \cdot (N/P)^{2/3}$ rechte Seiten mit Hilfe des Sparse-Lösers gelöst werden. Dies benötigt ebenso

$$C^0_{\textsc{Solve}:R} = O\left(\left(\frac{N}{P}\right)^{7/3}\right) \tag{5.2}$$

Operationen. Um das Schur-Komplement zu berechnen, muss nun nur noch die Matrix $L^{(0),p}_{\mathrm{loc}}$ mit der geänderten Matrix $R^{(0),p}_{\mathrm{loc}}$ multipliziert werden und von $S^{(0),p}_{\mathrm{loc}}$ abgezogen werden. Da $L^{(0),p}_{\mathrm{loc}}$ ebenso eine Sparse-Matrix der Größe

$$i_0 \times z_0 = 6 \cdot (N/P)^{2/3} \times N/P$$

mit Bandbreite $(N/P)^{2/3}$ ist, folgt für die Berechnung des Schur-Komplements ein Aufwand von

$$C^0_{\mathrm{MM}:S} = O\left(\left(\frac{N}{P}\right)^{2/3}\left(\frac{N}{P}\right)^{4/3}\right) = O\left(\left(\frac{N}{P}\right)^{2}\right)\,. \tag{5.3}$$

Damit ist Schritt $s = 0$ abgeschlossen und die Gesamtoperationen aus (5.1), (5.2), (5.3) betragen

$$C^0_{\mathrm{LR}} + C^0_{\textsc{Solve}:R} + C^0_{\mathrm{MM}} = O\left(\left(\frac{N}{P}\right)^{7/3}\right) + O\left(\left(\frac{N}{P}\right)^{2}\right)\,. \tag{5.4}$$

5.5. Analyse zur Zerlegung der Matrix $Z^{(s),t}$

Für $s > 1$ sind alle Matrizen vollbesetzt, so dass ein Sparse-Löser wie in Schritt $s = 0$ ausgeschlossen werden kann. Allerdings stehen hier P_s Prozessoren zur Verfügung, die die Zerlegung parallel bearbeiten. Wir betrachten zunächst Algorithmus 3.4 (parallele Block-LR-Zerlegung mit verteilten Spalten), welcher einen Teil von Algorithmus 3.30 beschreibt. Die Matrix $Z^{(s),t}$ der Größe $z_s \times z_s$ ist spaltenweise auf P_s Prozessoren verteilt, somit besitzt jeder Prozessor

$$m_s := \frac{z_s}{P_s}$$

Spalten. Der Algorithmus teilt sich auf in einen sequentiellen Teil (Zeile 1) und einen parallelen Teil (Zeile 7). Durch den sequentiellen Teil erhalten wir in jeder Abschätzung einen Faktor P_s. In Zeile 4 wird eine LR-Zerlegung einer $m_s \times m_s$ Matrix berechnet. Dies benötigt

$$C^s_{\mathrm{LR}} = O\left(P_s(m_s)^3\right) = O\left(\frac{z_s^3}{P_s^2}\right) \tag{5.5}$$

Operationen. Diese Zerlegung wird – bis auf den letzten Schritt – in Zeile 8 parallel auf mehreren Prozessoren auf eine $m_s \times m_s$ Matrix angewandt, so dass hier ebenso

$$C^s_{\textsc{Solve}} = O\left(P_s(m_s)^3\right) = O\left(\frac{z_s^3}{P_s^2}\right) \tag{5.6}$$

Operationen notwendig sind. In Zeile 9 wird nun noch eine Matrix-Matrix-Multiplikation zur Aktualisierung des Schur-Komplements durchgeführt. Dabei berechnet jeder Prozessor $q > p$ einen Teil der Schur-Komplements der Größe

$$(P_s - p - 1) \cdot m_s \times m_s \tag{5.7}$$

(im Schnitt also $(P_s/2) \cdot m_s \times m_s$), so dass hier insgesamt

$$C^s_{\text{MM}} = O\left(P_s^2 (m_s)^3\right) = O\left(\frac{z_s^3}{P_s}\right) \tag{5.8}$$

Operationen durchgeführt werden.

5.6. Analyse der weiteren Operationen

Für die weiteren Operationen betrachten wir wieder Algorithmus 3.30. Es fehlen noch die Operationen für die Matrix $L^{(s),t}_{\text{loc}}$ (Zeile 19), die Anwendung der LR-Zerlegung auf $R^{(s),t}_{\text{loc}}$ (Zeile 21), sowie die Aktualisierungen des Schur-Komplements in $R^{(s),t}_{\text{loc}}$ (Zeile 22) und in $S^{(s),t}_{\text{loc}}$ (Zeile 23). Alle Matrizen sind spaltenweise auf P_s Prozessoren verteilt, insbesondere finden sich m_s Spalten von $L^{(s),t}_{\text{loc}}$ und

$$r_s := \frac{i_s}{P_s}$$

Spalten von $R^{(s),t}_{\text{loc}}$ und $S^{(s),t}_{\text{loc}}$ auf den Prozessoren. Auch hier erhalten wir durch den sequentiellen Teil in Zeile 8 einen Faktor P_s für jede Abschätzung. Die Matrixoperation in $L^{(s),t}_{\text{loc}}$ aus Zeile 19 benötigt dabei

$$C^s_{\text{MM:}L} = O\left(P_s \left(i_s m_s^2\right)\right) = O\left(\frac{i_s z_s^2}{P_s}\right) \tag{5.9}$$

Operationen, da mindestens ein Prozessor existiert, der seinen kompletten Anteil von $L^{(s),t}_{\text{loc}}$ mit Größe $i_s \times z_s$ aktualisieren muss. Für die Anwendung der zerlegten Matrix in $Z^{(s),t}$ auf $R^{(s),t}_{\text{loc}}$ in Zeile 21 werden

$$C^s_{\text{Solve:}R} = O\left(P_s \left(m_s^2 r_s\right)\right) = O\left(\frac{i_s z_s^2}{P_s^2}\right) \tag{5.10}$$

Operationen benötigt. Zur Aktualisierung des Schur-Komplements in $R^{(s),t}_{\text{loc}}$ muss entsprechend (5.7) im Schnitt eine Matrix der Größe $P_s/2 \cdot m_s \times r_s$ aktualisiert werden, so dass hierfür

$$C^s_{\text{MM:}R} = O\left(P_s \left(P_s m_s^2 r_s\right)\right) = O\left(\frac{i_s z_s^2}{P_s}\right) \tag{5.11}$$

Operationen nötig sind. Die letzte Operation aus Zeile 23 entspricht einer Matrix-Matrix-Multiplikation einer $i_s \times m_s$-Matrix mit einer $m_s \times r_s$-Matrix, so dass hier

$$C^s_{\text{MM:}S} = O\left(P_s \left(i_s m_s r_s\right)\right) = O\left(\frac{i_s^2 z_s}{P_s}\right) \tag{5.12}$$

Operationen benötigt werden.

5.7. Asymptotisches Verhalten der Rechenoperationen

Zur genaueren Analyse wurden bisher die Teiloperationen einzeln betrachtet, da ein asymptotisches Verhalten hier nur ungenau beobachtet werden kann. Die Anzahl der sinnvoll benutzbaren Prozessoren (insbesondere aufgrund Speicherbedarf, Zeitverbrauch und wie wir später sehen auch wegen der Kommunikation zwischen den Prozessoren) ist im Vergleich zu asymptotischen Aussagen für einen Gesamtverbrauch sehr gering, jedoch können die einzelnen Schritte mit Hilfe der bisherigen Analyse sehr gut abgeschätzt werden. Für die globale Komplexität fassen wir die

einzelnen Operationen zusammen. Da Schritt $s = 0$ eine Sonderrolle einnimmt, betrachten wir zunächst die Komplexität für Schritt $s > 0$. In jedem dritten Schritt werden die inneren Elemente und die Interface-Elemente vervierfacht, das heißt, es gilt [4]

$$z_s = (\sqrt[3]{4})^{s-1} z_1 = (\sqrt[3]{4})^{s-1} \left(\frac{N}{P}\right)^{2/3} \tag{5.13}$$

$$i_s = (\sqrt[3]{4})^{s-1} i_1 = 10 \cdot (\sqrt[3]{4})^{s-1} \left(\frac{N}{P}\right)^{2/3} \tag{5.14}$$

Insgesamt erhalten wir also mit (5.5), (5.6), (5.8), (5.9), (5.10), (5.11), (5.12) eine Komplexität für Schritt s von

$$\begin{aligned}
&O\left(\frac{z_s^3}{P_s^2} + \frac{z_s^3}{P_s^2} + \frac{z_s^3}{P_s} + \frac{i_s z_s^2}{P_s} + \frac{i_s z_s^2}{P_s^2} + \frac{i_s z_s^2}{P_s} + \frac{i_s^2 z_s}{P_s}\right) \\
&= O\left(\frac{4^{s-1}}{P_s^2}(1+1+10)\cdot\left(\frac{N}{P}\right)^2 + \frac{4^{s-1}}{P_s}(1+10+10+100)\cdot\left(\frac{N}{P}\right)^2\right) \\
&\overset{P_s=2^s}{=} O\left(\left(\frac{N}{P}\right)^2\right) + O\left(\frac{2^{2s-2}}{2^s}\left(\frac{N}{P}\right)^2\right) = O\left(\left(\frac{N}{P}\right)^2\right) + O\left(2^{s-2}\left(\frac{N}{P}\right)^2\right).
\end{aligned}$$

Insgesamt werden $S = \log_2(P)$ Schritte durchgeführt und wegen

$$\sum_{s=1}^{S} 2^{s-2} = \frac{1}{4}\sum_{s=1}^{S} 2^s = \frac{1}{4}(2^{S+1} - 1) = O(P)$$

erhalten wir mit (5.4) (Schritt $s = 0$) die asymptotische Gesamtkomplexität für den Rechenaufwand von

$$C_{\text{ber}} = O\left(\left(\frac{N}{P}\right)^{7/3}\right) + O\left(\left(\frac{N}{P}\right)^2\right) + O\left(\left(\frac{N}{P}\right)^2 \log_2(P)\right) + O\left(\frac{N^2}{P}\right). \tag{5.15}$$

5.8. Analyse des Kommunikationsaufwands

In der parallelen LR-Zerlegung werden nicht nur Berechnungen angestellt, sondern insbesondere zwischen Prozessoren kommuniziert. Eine Kommunikation ist gewöhnlich langsamer als ein Rechenschritt; vor allem wenn viele Prozessoren daran beteiligt sind, wirkt sich mehr und mehr die zusätzliche Kommunikation auf die Zerlegungszeit aus. Um eine optimale Zerlegungszeit zu erreichen, muss eine gute Balance zwischen reinen Rechnungen und Kommunikation gefunden werden. Daher betrachten wir in diesem Kapitel die verwendeten Kommunikationsschritte in Algorithmus 3.30.

In unserem Fall existieren zwei Arten einer Kommunikation: Die Eins-zu-Eins-Kommunikation, wobei ein Prozessor genau einem anderem eine Nachricht übermittelt und das Broadcasting, wobei ein Prozessor seine Informationen an alle anderen Prozessoren in der jeweiligen Prozessormenge sendet. Hierbei ist letztendlich die Anzahl der Aufrufe und die Menge der zu kommunizierenden Daten entscheidend. Dafür definieren wir folgende Bezeichnungen für Schritt s:

[4] Hier wird nur die höchste Ordnung angegeben, da ein asymptotisches Verhalten untersucht werden soll

Bezeichnung 5.1 (Variablennamen für Kommunikationsaufwand)**.**

$N_{\text{one},s}$	Anzahl der Eins-zu-Eins-Kommunikationsaufrufe
$D_{\text{one},s}$	Gesamtzahl der Daten für eine Eins-zu-Eins-Kommunikation
$N_{\text{bc},s}$	Anzahl der Broadcasts
$D_{\text{bc},s}$	Gesamtzahl der Daten für die Broadcasts

In Zeile 12 wird in Schritt s die gesamte Matrix $Z^{(s),t}$ der Größe $z_s \times z_s$ über insgesamt P_s Broadcasts verschickt, sowie in der nächsten Zeile 13 die gesamte Matrix $L_{\text{loc}}^{(s),t}$ der Größe $z_s \times i_s$. Ein weiterer Broadcast ist notwendig für jeden Pivotvektor, der sich aus der Lapack-Routine zur Berechnung der einzelnen LR-Zerlegungen ergibt. Somit gilt

$$\begin{aligned} D_{\text{bc},s} &= z_s^2 + z_s \cdot i_s \\ N_{\text{bc},s} &= 3P_s\,. \end{aligned}$$

Wir nehmen an, dass eine Broadcast-Operation in Abhängigkeit der beteiligten Prozessoren insgesamt $O(\log_2(P_s)) = O(s)$ Zeitschritte benötigt. Mit (5.13) und (5.14) ergibt sich für den Kommunikationsaufwand eine Abschätzung von

$$\begin{aligned} C_{\text{bc}}^s &= O\left(s \cdot D_{\text{bc},s}\right) = O\left(s \cdot (z_s^2 + z_s i_s)\right) \\ &= O\left(s \cdot \left[\left(\left(\sqrt[3]{4}\right)^{s-1}\left(\frac{N}{P}\right)^{2/3}\right)^2 + 10 \cdot \left(\left(\sqrt[3]{4}\right)^{s-1}\left(\frac{N}{P}\right)^{2/3}\right)\right]\right) \\ &= O\left(s\left(\sqrt[3]{4}\right)^{2s-2}\left(\frac{N}{P}\right)^{4/3}\right) = O\left(s\left(\sqrt[3]{4}\right)^{2s}\left(\frac{N}{P}\right)^{4/3}\right)\,. \end{aligned}$$

Wir setzen $K = 2^{4/3}$, somit summiert sich der Gesamtaufwand auf

$$\begin{aligned} C_{\text{bc}} &= \sum_{s=0}^{S} C_{\text{bc}}^s = O\left(\sum_{s=0}^{S} s \cdot K^s \left(\frac{N}{P}\right)^{4/3}\right) \\ &= O\left(\sum_{s=0}^{S} s \cdot K^s \left(\frac{N}{P}\right)^{4/3}\right) \overset{(*)}{=} O\left(S \cdot K^S \left(\frac{N}{P}\right)^{4/3}\right)\,, \end{aligned}$$

wobei die Gleichung mit $(*)$ mit $K > 1$ aus

$$K + 2K^2 + 3K^3 + \cdots + SK^S = O(SK^S)$$

folgt. Mit $P = 2^S$ folgt weiterhin

$$C_{\text{bc}} = O\left(S \cdot \left(\frac{K}{2^{4/3}}\right)^S N^{4/3}\right) = O\left(\log_2 P \cdot N^{4/3}\right)\,. \tag{5.16}$$

Ein Vergleich zwischen Kommunikations- (C_{bc}) und Berechnungsaufwand (C_{ber} aus (5.15)) ergibt, dass die Zeit für den Broadcast die Berechnungszeit im Verlauf von mehr verwendeten Prozessoren übersteigt.

Die Eins-zu-Eins-Kommunikationen sind in Algorithmus 3.30 nicht direkt ersichtlich. Diese treten in Zeile 5 bis 7 auf. Hierbei werden die zwei Schur-Komplemente aus Schritt $s-1$ zusammengefasst und die Matrix $A_{\text{loc}}^{(s),t}$ erstellt. Ein zu versendendes Schur-Komplement wird beschrieben durch die Matrix $S_{\text{loc}}^{(s-1),t}$ der Größe i_{s-1}. Die Gesamtzahl an Daten, die kommuniziert werden muss, beträgt somit

$$N_{\text{one},s} = 2 \cdot i_{s-1}^2\,.$$

Da jeweils die gesamte Spalte von $S_{\text{loc}}^{(s-1),t}$ an einen neuen Prozessor versendet wird, können wir die Anzahl der Kommunikationsaufrufe für jeden Prozessor mit

$$D_{\text{one},s} = O(1)$$

abschätzen aus folgendem Grund: Wir können annehmen, dass die Spalten in $S_{\text{loc}}^{(s-1),t}$ so sortiert sind, dass sie auch aufsteigend in $A_{\text{loc}}^{(s),t}$ zu finden sind, das heißt, für nebeneinander stehende Spalten, die zu einem neuen Prozessor gehören, wird nur eine Kommunikation benötigt. Des Weiteren erhöht sich die Anzahl der Prozessoren nur um den Faktor 2, wobei die Matrix $A_{\text{loc}}^{(s),t}$ mindestens die Größe von $S_{\text{loc}}^{(s-1),t}$ besitzt. Damit erhält jeder Prozessor in Schritt s mindestens die Hälfte der Anzahl der Spalten aus dem vorigen Schritt, die jeweils zusammenhängend sind. Diese sind auf maximal zwei Prozessoren im vorigen Schritt verteilt.

5.9. Vergleich zwischen Theorie und Praxis

Um eine Aussage zwischen Theorie und Praxis zu erhalten, müssen die zugehörigen Probleme groß genug sein, so dass eine gewisse Fluktuation in den Berechnungen (zum Beispiel durch günstige Speicherbelegung) und in der Kommunikation ausgeglichen werden kann. Dies ist für viele Prozessoren allerdings nicht mehr gegeben, da dort die Matrixgrößen recht klein werden können.

Für die theoretische Berechnung der Rechenzeit in den einzelnen Schritten sollte daher angenommen werden, dass die Berechnung zum Beispiel einer LR-Zerlegung bis zu einer Mindestgröße N_{LR} konstant ist und erst dann in $O(N^3)$ wächst. Dies ist auch abhängig von der jeweiligen Rechnerplattform, durch verschiedene Testläufe kann jedoch eine solche Grenze bestätigt werden. Für die Kommunikation gilt Ähnliches - hier kommt es zusätzlich darauf an, wie die einzelnen Prozessoren miteinander verbunden sind. Um eine gewisse Aussage treffen zu können, müssen wir die eingesetzten Routinen (in unserem Fall `SuperLU`, `LAPACK`) unterschiedlich bewerten.

Für die Untersuchungen verwenden wir für die Zerlegung in Schritt $s = 0$ die Routine `SuperLU` [DEG+99] als Sparse-Löser. Als Alternative wurde als zweiter Sparse-Löser `MUMPS` [ADLK01] implementiert. Dieser Löser hatte zwar allgemein bessere zeitliche Ergebnisse sowohl in der Zerlegung als auch zum Lösen erzielt, die Anbindung von `SuperLU` war auf dem Parallelcluster `HERMIT` in Stuttgart jedoch deutlich einfacher zu realisieren. Sofern die Matrixgrößen klein genug sind, das heißt, wenn das Gesamtproblem auf genügend Prozessoren verteilt sind, fällt der Unterschied insgesamt nicht mehr ins Gewicht. Andere Sparse-Löser, wie `PaStiX` [PPJ02] oder `ParDiSo` [SWH07] sind für diesen Schritt selbstverständlich auch geeignet. Die Berechnung des Schur-Komplements $\hat{S}_{\text{loc}}^{(0),p} = S_{\text{loc}}^{(0),p} - L_{\text{loc}}^{(0),p}(Z^{(0),p})^{-1}R_{\text{loc}}^{(0),p}$, also insbesondere die Anwendung von $L_{\text{loc}}^{(0),p}$ auf die geänderte Matrix $R_{\text{loc}}^{(0),p}$ wurde direkt und ohne andere Hilfsmittel programmiert, da die Matrix $L_{\text{loc}}^{(0),p}$ als Sparse-Matrix gegeben ist.

Für alle Berechnungen in den folgenden Schritten $s > 0$ werden `LAPACK` - Routinen (vergleiche auch Anhang A) benutzt, insbesondere zur LR-Zerlegung der Diagonalmatrizen (`dgetrf`), die Anwendung dieser auf die rechten Seiten $R_{\text{loc}}^{(s),t}$ (`dgetrs`), sowie die Berechnung des Schur-Komplements über eine Matrix-Matrix-Multiplikation (`dgemm`).

Mit Hilfe der obigen Komplexitätsangaben können wir die Berechnungs- und Kommunikationszeit in jedem Schritt abschätzen. Für die Berechnungen reicht es dabei aus anzugeben, wie lange in etwa eine Operation pro Berechnungsschritt

benötigt. Aufgrund der guten Ausnutzung von Speicherzugriffen wird jedoch angenommen, dass diese nur für die Operationen in `LAPACK` gelten. Für die Berechnungen in Schritt $s = 0$ hat sich über Testkonfigurationen ein zusätzlicher Faktor von fast 8 herausgestellt. Dies liegt allerdings vor allem daran, dass `LAPACK` optimierte BLAS2 bzw. BLAS3-Routinen benutzt, während `SuperLU` eigene Routinen beinhaltet. Außerdem müssen zusätzlich zu den eigentlichen Berechnungen auch Vorkonfigurationen von `SuperLU` bestimmt werden, damit die Dünnbesetztheit der Matrix möglichst gut ausgenutzt und möglichst wenig zusätzlicher Speicherplatz benötigt wird. Da die Gesamtmatrix in dem untersuchten Fall sogar diagonaldominant ist, benötigen die Routinen in `LAPACK` keine Permutation, so dass dieser Faktor durchaus nachvollziehbar ist.

Für die Kommunikationsuntersuchungen wurde eine Testkonfiguration erstellt, wobei bestimmt wurde, wie viele Einheiten pro Sekunde übertragen werden können, sowie um zu bestimmen, wie groß die durchschnittliche Ansetzzeit ist. Weiterhin muss zwischen einem Broadcast und einer Eins-zu-Eins-Kommunikation unterschieden werden. Die Eins-zu-Eins-Kommunikation findet in unserem Fall in der Belegung der Matrix $A_{\text{loc}}^{(s),t}$ aus den vorigen Schur-Komplementen statt. Dort werden viele Vorgänge wiederum parallel ausgeführt (indem zum Beispiel Prozessor 1 und 2 miteinander kommunizieren und im gleichen Zeitraum auch Prozessor 3 und 4). Dabei können umso mehr Vorgänge parallelisiert durchgeführt werden, je mehr Prozessoren zur Verfügung stehen. Für die tatsächliche Kommunikationsdauer führen wir noch einen Faktor F_K ein, da die angegebene Bandbreite nur eine optimale obere Grenze darstellt.

Damit definieren wir folgende Konstanten für den Parallelcluster `HERMIT`, dabei wurde der Faktor F_0 und F_{K} aus den entsprechenden Testkonfigurationen gewonnen, während alle anderen Konstanten aus der Hardware-Konfiguration von `HERMIT` direkt stammen (vergleiche auch Anhang C).

Name	Wert	Beschreibung
T_{flop}	$4.3E-10$	Zeit pro flop, entspricht 2.3 GHz.
T_{send}	$3.2E-09$	Sendezeit für 1 double (8 Byte), entspricht 20GBit/s
T_{A}	$4.2E-07$	Ansetzzeit für eine Kommunikation
F_0	8	Faktor für Berechnungen in Schritt $s = 0$
F_{K}	1.5	Faktor für tatsächliche Kommunikationsdauer

TABELLE 4. Daten für den Parallelcluster `HERMIT`

Mit diesen Daten betrachten wir das Problem mit einer Verfeinerung auf Level $l = 7$ (insgesamt 2 246 689 Unbekannte) auf $P = 512$ Prozessoren (Tabelle 8) und auf $P = 2048$ Prozessoren (Tabelle 9). Die Zeitmessung im Programm ist nicht optimal, somit stellt die gemessene Zeit in den einzelnen Schritten nur eine ungefähre Zeit in den Berechnungen und in der Kommunikation dar. Es werden dabei folgende Operationen betrachtet:

Nr.	Name	zugehörige Abschätzung	Referenz
(1)	LR	C^s_{LR}	(5.1), (5.5)
(2)	Solve in Z	C^s_{Solve}	(5.6)
(3)	MM in Z	C^s_{MM}	(5.8)
(4)	Solve in R	$C^s_{\text{Solve:}R}$	(5.2), (5.10)
(5)	MM in L	$C^s_{\text{MM:}L}$	(5.9)
(6)	MM in S	$C^s_{\text{MM:}S}$	(5.3), (5.12)
(7)	MM in R	$C^s_{\text{MM:}R}$	(5.11)

TABELLE 5. Referenzen für die Operationen der parallelen Block-LR-Zerlegung

Die Berechnungsvorschriften mit zugehörigen Vorkonstanten in Schritt $s = 0$ lauten:

$$\begin{aligned} \text{aus (5.1):} \quad & (1) = (2/3)^2 \cdot z_0^{7/3} \\ \text{aus (5.2):} \quad & (4) = 2 \cdot z_0^{5/3} i_0 \\ \text{aus (5.3):} \quad & (6) = z_0^{4/3} i_0 \,, \end{aligned}$$

und für die anderen Schritte $s > 0$ mit $P_s = 2^s$

$$\begin{aligned} \text{aus (5.5):} \quad & (1) = 2/3 \cdot z_s^3 / P_s^2 \\ \text{aus (5.6):} \quad & (2) = 2 \cdot z_s^3 / P_s^2 \\ \text{aus (5.8):} \quad & (3) = z_s^3 / P_s \\ \text{aus (5.10):} \quad & (4) = 2 \cdot z_s^2 i_s / P_s^2 \\ \text{aus (5.9):} \quad & (5) = z_s^2 i_s / P_s \\ \text{aus (5.12):} \quad & (6) = z_s i_s^2 / P_s \\ \text{aus (5.11):} \quad & (7) = z_s^2 i_s / P_s \,, \end{aligned}$$

wobei für die LR-Zerlegung der Faktor 2/3 als Standardzerlegung genommen wird, sowie für die Lösungsroutine der Faktor 2. Die Matrix-Matrix-Multiplikation ist als Hauptaufwand zu sehen (siehe Tabellen 8,9). Dort werden optimierte BLAS3-Routinen verwendet. Die Summe der Operationen $(1), \ldots, (7)$ in einem Schritt s wird mit N^s_{flops} bezeichnet, also

$$\begin{aligned} N^0_{\text{flops}} &= 2/3 z_0^{7/3} + 2 z_0^{5/3} i_0 + z_0^{4/3} i_0 \\ N^s_{\text{flops}} &= 8/3 \frac{z_s^3}{P_s^2} + \frac{z_s^3}{P_s} + 2\frac{z_s^2 i_s}{P_s^2} + 2\frac{z_s^2 i_s}{P_s} + \frac{z_x i_s^2}{P_s} \qquad (s > 0) \,. \end{aligned}$$

Die Zeitabschätzung für die Berechnungen wird dann über den Ausdruck

$$T_{\text{ber},s} = N_{\text{flops}} \cdot T_{\text{flop}} \tag{5.17}$$

angegeben, wobei für $T_{\text{ber},0}$ zusätzlich der Faktor F_0 hinzukommt. Die gemessene Zeit wird in den Tabellen mit $T^o_{\text{ber},s}$ bezeichnet.

Für die Kommunikation in Schritt s erhalten wir folgende Berechnung für den Broadcast auf P_s Prozessoren:

$$T_{\text{bc},s} = F_K \cdot (3P_s \cdot T_{\text{A}} + (z_s^2 + z_s \cdot i_s) \cdot \log(P_s) \cdot T_{\text{send}}) \,, \tag{5.18}$$

und mit $T^o_{\text{bc,s}}$ wird die gemessene Zeit in den Tabellen angegeben.

Die Eins-zu-Eins-Kommunikationen sind nur sehr ungenau messbar, da die Datenmenge sehr oft nur relativ klein ist. Vor allem können dort innerhalb einer Prozessormenge sehr viele Kommunikationen gleichzeitig geschehen, wobei andere Kommunikationen eventuell erst später ausgeführt werden können, zum Beispiel, wenn ein Prozessor sehr lange warten muss, bis er seine ersten Daten senden beziehungsweise empfangen kann. Beim Broadcast hingegen ist es sehr klar geregelt, wann welcher Prozessor wie viele Daten senden muss. Insgesamt stellt sich auch heraus, dass die Broadcast-Kommunikationen im Verlauf des Algorithmus deutlich mehr Aufwand benötigen als die Eins-zu-Eins-Kommunikationen. Diese überwiegen allerdings am Anfang der Zerlegung. In den Tabellen sollen daher für die Eins-zu-Eins-Kommunikation nur die Ergebnisse aus den Testläufen wiedergegeben werden ohne diese explizit abzuschätzen.

In Kapitel 7.1 wird eine allgemeine Berechnungsvorhersage für das gesamte Problem gegeben. Da dort die Eins-zu-Eins-Kommunikationen in der Gesamtzeit nicht vernachlässigt werden können, wird die Berechnungsdauer davon abgeschätzt durch

$$T_{\text{one},s} = F_{\text{one}} N_{\text{one},s} T_{\text{send}} \,, \tag{5.19}$$

mit $F_{\text{one}} = 5$ als Strafterm für die Eins-zu-Eins-Kommunikation.

In den Tabellen 6 bis 9 werden gleichzeitig die Daten der Vorhersage mit der tatsächlichen Berechnungszeit angegeben. In Tabelle 8 und 9 sind diese zusätzlich aufgeteilt in jeden einzelnen Schritt, wobei die Vorhersage nach (5.17) berechnet wird und die zugehörigen flops einzeln aufgelistet sind nach Tabelle 5.

Insgesamt ergibt sich eine recht gute Vorhersage der Berechnungen und der Broadcast-Zeiten auf dem Parallelcluster `HERMIT`. Es ist weiterhin auch deutlich zu sehen, dass sowohl die Zeiten der Rechenoperationen als auch die Kommunikationsdauer auf 2048 Prozessoren deutlich niedriger sind als auf 512 Prozessoren.

5.9.1. Bemerkungen zum Praxisvergleich auf 512 Prozessoren.

Ein Vergleich der berechneten Zeit $T_{\text{ber},s}$ mit der gemessenen Zeit $T^o_{\text{ber},s}$ in Tabelle 8 ergibt eine insgesamt sehr gute Vorhersage der Gesamtberechnungen. Auch die vorhergesagten Broadcast-Zeiten entsprechen in einer sehr guten Toleranz den gemessenen Daten, insbesondere in den höheren Schritten. Allerdings muss hier beachtet werden, dass die Teilzeiten nicht genau gemessen werden konnten, so beträgt die Differenz der einfach zu messenden Gesamtzeit zu den gemessenen Einzelzeiten auf 512 Prozessoren $9.71s$ (Tabelle 6). Somit ist ein exakter Vergleich natürlich nicht machbar. Insgesamt wird jedoch deutlich, an welchen Stellen die meiste Arbeit geleistet werden muss. Diese findet sich in den Berechnungen vor allem in der Matrix-Matrix-Multiplikation zur Erstellung des Schur-Komplements ((6) MM in S). In den höheren Schritten muss weiterhin deutlich mehr kommuniziert werden, dies liegt insbesondere daran, dass die gesamte Matrix $Z^{(s),t}$ mit Größe $z_s \times z_s$ auf alle Prozessoren in der jeweiligen Prozessormenge geschickt werden muss.

5.9.2. Bemerkungen zum Praxisvergleich auf 2048 Prozessoren.

Auch auf 2048 Prozessoren stimmen die berechneten Zeiten sehr gut mit den gemessenen Zeiten für die Berechnungen überein (Tabelle 9). Hierbei muss erwähnt werden, dass in den letzten drei Schritten die Anzahl der Spalten je Prozessor nur etwa 8 beträgt. Lediglich in den höheren Schritten wird für die Berechnungen etwas mehr Zeit benötigt als angegeben wird. Dies liegt jedoch daran, dass die Einzelmatrizen auf den Prozessoren zu klein für eine optimale Berechnung von `LAPACK` sind. Auf jedem Prozessor sind jeweils nur etwa 8 Spalten der Matrix Z, so dass eine LR-Zerlegung, sowie die Anwendung mit dieser nicht effizient genug ist. Wie am Anfang des Kapitels erwähnt, muss für diesen Fall die theoretische Gesamtmatrixgröße von Z auf $P_s \cdot N_{\text{LR}}$ erhöht werden. Mit $N_{\text{LR}} = 10$ erhalten wir in Schritt $s = 11$ als Gesamtmatrixgröße $z_s = 20480$ und damit eine berechnete Gesamtzeit von $1.81s$, was insgesamt ein besseres Resultat liefert. Die theoretische Vorhersage der Broadcast-Kommunikation stimmt weitgehend mit den gemessenen Zeiten überein. Insgesamt wird hier auch deutlich, dass die Einzelzeitmessungen in der Summe wohl besser waren im Vergleich zu 512 Prozessoren. Hier beträgt die Differenz zur gemessenen Gesamtzeit nur $1.82s$ (Tabelle 7), obwohl insgesamt deutlich mehr Messungen durchzuführen waren als auf 512 Prozessoren.

5.9.3. Gesamtbemerkungen.

Insgesamt stimmen die Abschätzungen des Berechnungs- und Kommunikationsaufwands im verwendeten Parallelisierungsmodell in beiden Untersuchungen sehr gut mit den praktischen Resultaten überein. Sobald die Matrixgrößen auf den einzelnen Prozessoren zu klein werden, wird ein zusätzlicher Faktor in der Berechnung benötigt. Allgemein ist die Berechnung mit Hilfe von `LAPACK` insbesondere in den ersten Schritten sehr gut. Für die Broadcast-Kommunikation konnte eine gute Abschätzung angegeben werden, während für die Eins-zu-Eins-Kommunikationen aufgrund der Komplexität der einzelnen Routinen keine allgemein gültige Aussage getroffen werden konnte. Der Hauptaufwand liegt vor allem in der Matrix-Matrix-Multiplikation zur Berechnung des Schur-Komplements, dort wird meist mehr als die Hälfte der Gesamtzeit verbraucht.

Gesamtgröße des berechneten Problems:	2 146 489
berechnete Gesamtzeit der Rechenoperationen (Summe $T_{\mathrm{ber},s}$): **(Vorhersage)**	51.57 sec.
gemessene Gesamtzeit der Rechenoperationen T_R^o:	41.47 sec.
gemessene Gesamtzeit aller Kommunikationsschritte T_K^o:	46.74 sec.
gemessene Gesamtzeit T_G^o:	78.50 sec.
Differenz $(T_G^o - T_R^o - T_K^o)$	9.71 sec.

TABELLE 6. Gesamtbetrachtung des Problems auf 512 Prozessoren; Zeiten in Sekunden

Gesamtgröße des berechneten Problems:	2 146 489
berechnete Gesamtzeit der Rechenoperationen (Summe $T_{\mathrm{ber},s}$): **(Vorhersage)**	12.47 sec.
gemessene Gesamtzeit der Rechenoperationen T_R^o:	15.46 sec.
gemessene Gesamtzeit aller Kommunikationsschritte T_K^o:	34.61 sec.
gemessene Gesamtzeit T_G^o:	51.89 sec.
Differenz $(T_G^o - T_R^o - T_K^o)$	1.82 sec.

TABELLE 7. Gesamtbetrachtung des Problems auf 2048 Prozessoren; Zeiten in Sekunden

Schritt s	0	1	2	3	4	5	6	7	8	9
z_s	4096	256	512	1024	1024	2048	4096	4096	8256	16641
i_s	1538	2562	4098	6146	9281	12449	12481	16641	16641	0
LR (1)	179m	2.8m	5.6m	11.2m	2.8m	5.6m	11.2m	2.8m	5.7m	11.7m
Solve in Z (2)		8.4m	16.7m	33.6m	8.4m	16.8m	33.6m	8.4m	17.2m	35.2m
MM in Z (3)		8.3m	33.6m	134m	67.1m	268m	1074m	537m	2198m	9000m
Solve in R (4)	3225m	84m	134m	201m	76m	102m	102m	34.1m	34.6m	
MM in L (5)		84m	269m	806m	608m	1632m	3272m	2181m	4431m	
MM in S (6)	101m	840m	2150m	4835m	5513m	9919m	9970m	8862m	8931m	
MM in R (7)		84m	269m	806m	608m	1632m	3272m	2181m	4431m	
Summe N_{flops}	3505m	1112m	2877	6826m	6884m	13575m	17734m	13806m	20048m	9047m
ber. Zeit $T_{\text{ber},s}$ **(Vorhersage)**	11.9 sec.	0.48 sec.	1.24 sec.	2.94 sec.	2.96 sec.	5.84 sec.	7.63 sec.	5.94 sec.	8.62 sec.	3.89 sec.
gem. Zeit $T^o_{\text{ber},s}$	12.4 sec.	0.38 sec.	1.02 sec.	2.40 sec.	2.14 sec.	4.16 sec.	4.97 sec.	4.98 sec.	6.98 sec.	2.14 sec.
ber. Zeit $T_{\text{bc},s}$ **(Vorhersage)**		0.25 sec.	0.10 sec.	0.59 sec.	1.03 sec.	1.90 sec.	2.95 sec.	4.37 sec.	8.29 sec.	11.96 sec.
gem. Zeit T^o_{bc}		0.01 sec.	0.19 sec.	0.80 sec.	0.37 sec.	1.16 sec.	2.79 sec.	3.34 sec.	8.45 sec.	11.23 sec.
gem. Zeit T^o_{one}	0.1 sec.	0.36 sec.	0.77 sec.	1.53 sec.	2.60 sec.	1.92 sec.	2.46 sec.	5.37 sec.	2.30 sec.	

TABELLE 8. Anzahl der Rechenoperationen in Millionen (m; (1) – (7), sowie Summe N_{flops})), sowie berechnete und gemessene Zeit der Berechnung und Kommunikation auf $P = 512$ Prozessoren in Sekunden (s)

Schritt s	0	1	2	3	4	5	6	7	8	9	10	11
z_s	1024	128	256	256	512	1024	1024	2048	4096	4096	8256	16641
i_s	642	1026	1538	2562	4098	6146	9281	12449	12481	16641	16641	0
LR (1)	7.1m	0.4m	0.7m	0.2m	0.4m	0.7m	0.2m	0.4m	0.7m	0.2m	0.4m	0.7m
Solve in Z (2)		1.1m	2.1m	0.5m	1.1m	2.1m	0.5m	1.1m	2.1m	0.5m	1.1m	2.2m
MM in Z (3)		1.1m	4.2m	2.1m	8.4m	33.6m	16.8m	67.1m	268m	134m	550m	2250m
Solve in R (4)	133m	8.4m	12.6m	5.3m	8.4m	12.6m	4.8m	6.4m	6.4m	2.1m	2.2m	
MM in L (5)		8.4m	25.2m	21.0m	67.1m	201m	152m	408m	818m	545m	1108m	
MM in S (6)	6.6m	67.3m	151m	210m	537m	1209m	1378m	2480m	2492m	2215m	2233m	
MM in R (7)		8.4m	25.2m	21.0m	67.1m	201m	152m	408m	818m	545m	1108m	
Summe N_{flops}	147m	95m	221m	260m	690m	1660m	1704m	3370m	4406m	3443m	5001m	2253m
ber. Zeit $T_{\text{ber},s}$ **(Vorhersage)**	0.50 sec.	0.04 sec.	0.10 sec.	0.11 sec.	0.30 sec.	0.71 sec.	0.73 sec.	2.41 sec.	2.96 sec.	1.48 sec.	2.15 sec.	0.97 sec.
gem. Zeit $T^o_{\text{ber},s}$	0.42 sec.	0.02 sec.	0.06 sec.	0.09 sec.	0.27 sec.	0.63 sec.	0.79 sec.	1.65 sec.	2.43 sec.	2.56 sec.	4.59 sec.	1.95 sec.
ber. Zeit $T_{\text{bc},s}$ **(Vorhersage)**		0.04 sec.	0.02 sec.	0.09 sec.	0.21 sec.	0.47 sec.	0.85 sec.	2.80 sec.	3.59 sec.	4.85 sec.	9.87 sec.	14.6 sec.
gem. Zeit T^o_{bc}		0.00 sec.	0.01 sec.	0.03 sec.	0.07 sec.	0.19 sec.	0.34 sec.	0.92 sec.	2.50 sec.	3.23 sec.	9.15 sec.	10.77 sec.
gem. Zeit T^o_{one}	0.01 sec.	0.03 sec.	0.07 sec.	0.19 sec.	0.52 sec.	0.93 sec.	1.58 sec.	1.75 sec.	1.52 sec.	2.38 sec.	0.11 sec.	

TABELLE 9. Anzahl der Rechenoperationen in Millionen (m; (1) – (7), sowie Summe N_{flops}), sowie berechnete und gemessene Zeit der Berechnung und Kommunikation auf $P = 2048$ Prozessoren in Sekunden (s)

Teil 3

Anwendung

6 Modellprobleme

Um die Effizienz der parallelen Block-LR-Zerlegung zu validieren, werden mehrere Modellprobleme aufgestellt, die sich in der Komplexität - sowohl in der Problemstellung selbst als auch in der Geometrie - deutlich unterscheiden. Dabei ist vor allem die Aufteilung der Geometrie Ω auf die Prozessoren ausschlaggebend für die Effizienz, insbesondere sollten die Größe der Interfaces in den einzelnen Schritten möglichst klein sein. Wir betrachten sowohl zweidimensionale als auch dreidimensionale Probleme.

6.1. Poisson-Gleichung

Zur Einführung und für die Analyse (vergleiche Kapitel 5) betrachten wir das Poisson-Problem[1] im dreidimensionalen Einheitswürfel $\Omega = (0,1)^3$ mit homogenen Dirichlet-Randbedingungen.

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega = (0,1)^3, \\ u &= 0 \quad \text{auf } \partial\Omega\,. \end{aligned}$$

Δ bezeichnet hierbei den Laplace-Operator im dreidimensionalen Raum

$$\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}\,.$$

Die Poisson-Gleichung als Prototyp einer elliptischen Differentialgleichung findet in vielen Bereichen der Physik Anwendung, insbesondere zur Bestimmung von Potentialen[2].

Für die Finiten Elemente verwenden wir den Hilbertraum $V = H_0^1(\Omega)$ und die Bilinearform

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x\,.$$

Als Diskretisierung des Gebietes verwenden wir eine regelmäßige Zerlegung in Hexaedern. Auf diesen betrachten wir trilineare Ansatzfunktionen [**Bra03**, Kap. II.§5], das heißt, die Knotenpunkte sind an den Ecken der Hexaeder zu finden. Die Steifigkeitsmatrix ist symmetrisch positiv definit.

[1] Für die Zerlegung der entstehenden Matrix ist die rechte Seite nicht von Bedeutung
[2] Die Laplace-Gleichung $-\Delta u = 0$ wird auch Potentialgleichung genannt

6.2. Stokes-Gleichung (2D)

Im Bereich der Strömungsmechanik betrachten wir ein Stokes-Problem an einem zweidimensionalen Beispiel aus [**ESW05**, Beispiel 4.1.2]. Die Stokes-Gleichung ist gegeben durch

$$\begin{aligned} -\Delta u + \nabla p &= 0 \\ \operatorname{div} u &= 0\,, \end{aligned} \tag{6.1}$$

wobei $u = (u_x, u_y)\colon \Omega \to \mathbb{R}^2$ die Geschwindigkeit des Fluids und $p\colon \Omega \to \mathbb{R}$ den Druck beschreibt. In diesem Fall ist die Geschwindigkeit des Fluids so gering, dass Konvektionseffekte vernachlässigt werden können. Zusätzlich sind Dirichlet-Randbedingungen

$$u = 0 \quad \text{auf } \partial\Omega$$

gegeben. Hier verwenden wir die Hilberträume $X = H_0^1(\Omega)^2$ und $M = L_0^2(\Omega)$, sowie die Bilinearformen $a\colon X \times X \to \mathbb{R}$ und $b\colon X \times M \to \mathbb{R}$ mit

$$\begin{aligned} a(u,v) &= \int_\Omega \nabla u : \nabla v \, \mathrm{d}x \\ b(v,q) &= -\int_\Omega q \nabla \cdot v \, \mathrm{d}x\,. \end{aligned}$$

Hierbei beschreibt $\nabla u : \nabla v$ das komponentenweise definierte Skalarprodukt, hier im Zweidimensionalen also $\nabla u_x \cdot \nabla v_x + \nabla u_y \cdot \nabla v_y$. Mit dem Funktional $l\colon X \to \mathbb{R}$

$$l(v) = \int_{\Gamma_N} s \cdot v \, \mathrm{d}o$$

erhalten wir die schwache Formulierung

$$\begin{aligned} &\text{Suche } (u,p) \in X \times M, \text{ so dass} \\ &\qquad a(u,v) + b(v,p) = l(v) \qquad \forall v \in X \\ &\qquad\qquad b(u,q) = 0 \qquad\quad \forall q \in M\,. \end{aligned}$$

Eine Diskretisierung erfolgt nun über die beiden Räume $X_h \subset X$ und $M_h \subset M$. Da diese Räume unabhängig voneinander sind, wird diese Approximationsmethode auch als gemischte Finite Elemente Methode bezeichnet. Als Diskretisierung verwenden wir inf-sup-stabile Taylor-Hood-Serendipity $Q2/Q1$-Elemente[3] [**BF91**]. Insgesamt erhalten wir somit ein lineares Gleichungssystem

$$\begin{pmatrix} A & B^T \\ B & 0 \end{pmatrix} \begin{pmatrix} u \\ p \end{pmatrix} = \begin{pmatrix} f \\ g \end{pmatrix}\,.$$

Die Matrix $\begin{pmatrix} A & B^T \\ B & 0 \end{pmatrix}$ ist symmetrisch, aber indefinit.

[3] quadratisches 8-Knoten-Element/bi-linear

6.3. Elastizität

In der Elastizitätstheorie wird die Deformation von Körpern unter der Einwirkung von Kräften betrachtet. Dabei werden Verzerrungen und Spannungen, welche durch Deformationen erzeugt werden, untersucht [Bra03]. Es wird davon ausgegangen, dass für den untersuchten Körper eine spannungsfreie Referenzkonfiguration Ω bekannt ist. Der Zustand zum Zeitpunkt t wird durch eine Abbildung

$$\chi\colon [0,T]\times\Omega\to\mathbb{R}^3$$

beschrieben. Dabei beschreibt $\chi(t,x)$ den Ort des Punktes zum Zeitpunkt t, wobei $x\in\Omega$ den Ort des Punktes im Referenzzustand beschreibt. Mit

$$\chi=\mathrm{id}+\boldsymbol{u}$$

wird die Verschiebung $\boldsymbol{u}\colon [0,T]\times\Omega\to\mathbb{R}^3$ definiert. Die Verschiebungen werden als klein angenommen, so dass höhere Terme in $\boldsymbol{u}$ vernachlässigt werden.

Mit dem Deformationsgradienten $\boldsymbol{F}=D\chi$ ist χ eine Deformation, wenn

$$J=\det(\boldsymbol{F})>0$$

gilt, das heißt insbesondere, dass Teilmengen mit positiven Volumen wieder auf Teilmengen mit positiven Volumen abgebildet werden.

Neben der lokalen Deformation wirken weiterhin Volumenkräfte $\boldsymbol{f}\colon\Omega\to\mathbb{R}^3$ und Flächenkräfte $\boldsymbol{g}\colon\Gamma^N\to\mathbb{R}^3$. Dabei ist $\Gamma^N\subset\partial\Omega$ der Teil des Randes von Ω, auf dem diese Neumann-Randbedingung definiert ist. Dirichlet-Randbedingungen werden auf $\Gamma^D\subset\partial\Omega$ definiert.

Hier betrachten wir ein quasi-inkompressibles Neo-Hooke-Material, so dass für die Formänderungsenergie W gilt

$$W(\boldsymbol{F})=\frac{\mu}{2}\boldsymbol{F}\cdot\boldsymbol{F}+\Sigma(J)$$

mit

$$\Sigma(J)=\frac{\lambda}{4}J^2-(\frac{\lambda}{2}+\mu)\log J-\frac{\lambda}{4}-\frac{3\mu}{2}\,,$$

wobei λ und μ die Lamé-Konstanten beschreiben. Für hyperelastische Materialien gilt für den Piola-Kirchhoffschen Spannungstensor

$$\boldsymbol{T}(\boldsymbol{F})=\frac{\partial W}{\partial\boldsymbol{F}}=\mu\boldsymbol{F}+J\Sigma'(J)\boldsymbol{F}^{-T}=\mu\boldsymbol{F}+\frac{\lambda}{2}J^2\boldsymbol{F}^{-T}-\left(\frac{\lambda}{2}+\mu\right)\boldsymbol{F}^{-T}.$$

Mit dem Satz von Cauchy [Bra03, Kapitel IV, Satz 1.3] ist der Gleichgewichtszustand eines elastischen Körpers durch die Gleichungen

$$-\operatorname{div}\boldsymbol{T}(t,\boldsymbol{x})=\boldsymbol{f}(\boldsymbol{x})\qquad\boldsymbol{x}\in\bar{\Omega},\quad t\in[0,T]$$

gegeben, mit den Randbedingungen

$$\begin{aligned}\boldsymbol{T}(t,\boldsymbol{x})\cdot\boldsymbol{n}&=\boldsymbol{g}(x)&&x\in\Gamma^N\\ \boldsymbol{u}(t,\boldsymbol{x})&=0&&x\in\Gamma^D\,.\end{aligned}$$

In einem Finite Elemente Raum $V_h\subset\{v\in W^{1,\infty}(\Omega,\mathbb{R}^3)\colon v(x)=0\ \forall x\in\Gamma^D\}$ betrachten wir die folgende Variationsformulierung: Finde $\chi_h\in V_h$, so dass

$$\int_\Omega\boldsymbol{T}(D\chi_h)\colon D\boldsymbol{v}_h\,\mathrm{d}\boldsymbol{x}-\int_\Omega\boldsymbol{f}\cdot\boldsymbol{v}_h\,\mathrm{d}\boldsymbol{x}-\int_{\Gamma^N}\boldsymbol{g}\cdot\boldsymbol{v}_h\,\mathrm{d}\boldsymbol{a}=0$$

für alle $\boldsymbol{v}_h\in V_h$ erfüllt ist.

Die Lösung dieses Problems wird für festes t_n über ein Newton-Verfahren bestimmt[4]

S0 **Wähle** $\chi^0 \in V_h$**,** $\varepsilon > 0$**, setze** $k := 0$**. Berechne**
- $\boldsymbol{f} = \boldsymbol{f}(t_n)$
- $\boldsymbol{g} = \boldsymbol{g}(t_n)$

S1 **Berechne:**
- $\boldsymbol{F}^k = D\chi^k$
- $\boldsymbol{T}^k = D_{\boldsymbol{F}} W(\boldsymbol{F}^k)$
- $r^k(\boldsymbol{v}) = \int_\Omega \boldsymbol{T}^k \colon D\boldsymbol{v} \, \mathrm{d}\boldsymbol{x} - \int_\Omega \boldsymbol{f} \cdot \boldsymbol{v} - \int_{\Gamma^N} \boldsymbol{g} \cdot \boldsymbol{v} \, \mathrm{d}\boldsymbol{a}$

Wenn $\|r^k\| < \varepsilon$ **: STOP**

S2 **Berechne:**
- $\mathbb{C}^k = D^2_{\boldsymbol{F}} W(\boldsymbol{F}^k)$
- $\boldsymbol{w}$ **mit** $a(\boldsymbol{w}, \boldsymbol{v}) = -r^k(\boldsymbol{v}) \qquad \forall \boldsymbol{v} \in V_h$ **mit**

$$a(\boldsymbol{w}, \boldsymbol{v}) = \int D\boldsymbol{w} \colon \mathbb{C}^k \colon D\boldsymbol{v} \, \mathrm{d}\boldsymbol{x}$$

Wähle Schrittweite $s_k \in (0, 1]$ **und setze** $\chi^{k+1} := \chi^k + s_k \boldsymbol{w}$**.**
Setze $k := k + 1$ **und gehe zu S1.**

Bei der Diskretisierung erhalten wir in S2 zur Berechnung von $\boldsymbol{w}$ in jedem Schritt eine Matrix, die mit Hilfe des parallelen Block-LR-Verfahrens zerlegt wird. Die Matrix ist symmetrisch und positiv definit.

Das Elastizität-Problem wird auf eine Herz-Geometrie angewandt, die in [Nie12] beschrieben wird.

6.4. Maxwell'sches Eigenwertproblem und elektromagnetische Wellengleichung

6.4.1. Maxwell'sche Gleichungen.

Die Ausbreitung elektromagnetischer Wellen werden über ein System der Maxwell'schen Gleichungen beschrieben.
Gesucht sind Vektorfelder $\boldsymbol{D}, \boldsymbol{E}$ und $\boldsymbol{H}, \boldsymbol{B} \colon [0, T] \times \Omega \to \mathbb{R}^3$ so dass die Gleichungen

$$\text{(MW 1)} \qquad \nabla \cdot \boldsymbol{D} = \rho$$

$$\text{(MW 2)} \qquad \nabla \cdot \boldsymbol{B} = 0$$

$$\text{(MW 3)} \qquad \nabla \times \boldsymbol{E} = -\frac{\partial \boldsymbol{B}}{\partial t}$$

$$\text{(MW 4)} \qquad \nabla \times \boldsymbol{H} = \boldsymbol{J} + \frac{\partial \boldsymbol{D}}{\partial t} \,,$$

für gegebene $\boldsymbol{J} \colon \Omega \to \mathbb{R}^D$, $\rho \colon \Omega \to \mathbb{R}$ erfüllt sind. Dabei beschreibt $\boldsymbol{E}$ die elektrische Feldstärke, $\boldsymbol{D}$ die elektrische Flussdichte, $\boldsymbol{B}$ die magnetische Flussdichte (Induktion), $\boldsymbol{H}$ die magnetische Feldstärke, $\boldsymbol{J}$ die Stromdichte und ρ die Ladungsdichte.

Wir betrachten lineare Materialien mit

$$\boldsymbol{D}(t, \boldsymbol{x}) = \varepsilon \boldsymbol{E}(t, \boldsymbol{x}) \tag{6.2}$$

$$\boldsymbol{H}(t, \boldsymbol{x}) = \frac{1}{\mu} \boldsymbol{B}(t, \boldsymbol{x}) \,, \tag{6.3}$$

mit Permittivität $\varepsilon = \varepsilon_0 \varepsilon_r$ (dielektrische Leitfähigkeit) und Permeabilität $\mu = \mu_0 \mu_r$ (magnetische Leitfähigkeit). Hierbei beschreibt $\mathrm{c} = \sqrt{\frac{1}{\mu_0 \varepsilon_0}}$ die Lichtgeschwindigkeit. Ohne äußere Einwirkungen ist die Stromdichte $\boldsymbol{J}$ und die Ladungsdichte ρ

[4] Der Diskretisierungsindex h wird hier zur Übersichtlichkeit weggelassen

gleich null. Aus (MW 1) bis (MW 4) und (6.2), (6.3) folgt somit direkt

$$\text{(MW 1')} \qquad \nabla \cdot \varepsilon \boldsymbol{E} = 0$$

$$\text{(MW 2')} \qquad \nabla \cdot \mu \boldsymbol{H} = 0$$

$$\text{(MW 3')} \qquad \nabla \times \boldsymbol{E} = -\mu \frac{\partial \boldsymbol{H}}{\partial t}$$

$$\text{(MW 4')} \qquad \nabla \times \boldsymbol{H} = \varepsilon \frac{\partial \boldsymbol{E}}{\partial t}\,,$$

6.4.2. Herleitung des Eigenwertproblems und der Wellengleichung. Wir betrachten die dritte Maxwell'sche Gleichung (MW 3')

$$\frac{1}{\mu} \nabla \times \boldsymbol{E} = -\frac{\partial \boldsymbol{H}}{\partial t}$$

und wenden darauf den Rotationsoperator $\nabla\times$ an:

$$\nabla \times (\frac{1}{\mu} \nabla \times \boldsymbol{E}) = -\frac{\partial}{\partial t} (\nabla \times \boldsymbol{H})\,.$$

Mit (MW 4') und $\boldsymbol{J} = 0$ folgt

$$\nabla \times (\frac{1}{\mu} \nabla \times \boldsymbol{E}) = -\varepsilon \frac{\partial^2 \boldsymbol{E}}{\partial t^2}\,. \tag{6.4}$$

Somit erhalten wir eine partielle Differentialgleichung zweiter Ordnung für $\boldsymbol{E}$. Mit Anwendung des Rotationsoperators $\nabla\times$ auf (MW 4') und Einsetzen von (MW 3') erhalten wir ebenso eine partielle Differentialgleichung zweiter Ordnung für $\boldsymbol{H}$:

$$\nabla \times (\frac{1}{\varepsilon} \nabla \times \boldsymbol{H}) = -\mu \frac{\partial^2 \boldsymbol{H}}{\partial t^2}\,. \tag{6.5}$$

Somit genügt es, eines der Felder $\boldsymbol{E}$ oder $\boldsymbol{H}$ zu betrachten.

6.4.2.1. *Das Eigenwertproblem.*

Mit dem Ansatz zeitharmonischer Lösungen der Art $\boldsymbol{u}(x,t) = \exp(\mathrm{i}\omega t)\boldsymbol{u}(x)$ und $\mu_r \equiv 1$ folgt aus (6.5)

$$\nabla \times (\varepsilon_r^{-1} \nabla \times \boldsymbol{u}) - (\omega/\mathrm{c})^2 \boldsymbol{u} = 0\,. \tag{6.6}$$

Dies ist das Maxwell'sche Eigenwertproblem, das heißt, gesucht sind hier die Eigenfrequenzen $\omega \neq 0$.

6.4.2.2. *Die Wellengleichung.*

Allgemein gilt

$$\nabla \times (\nabla \times \boldsymbol{A}) = \nabla(\nabla \cdot \boldsymbol{A}) - \Delta \boldsymbol{A}$$

und mit (MW 1') folgt

$$\nabla \times (\nabla \times \boldsymbol{E}) = \nabla(\nabla \cdot \boldsymbol{E}) - \Delta \boldsymbol{E} = -\Delta \boldsymbol{E}\,. \tag{6.7}$$

Aus (6.4) und (6.7) ergibt sich somit die Wellengleichung

$$\Delta \boldsymbol{E} = \mu\varepsilon \frac{\partial^2 \boldsymbol{E}}{\partial t^2}\,. \tag{6.8}$$

6.5. Das diskrete Maxwell'sche Eigenwertproblem

Wir folgen [Mon03] und betrachten ein beschränktes polygonales Gebiet $\Omega \in \mathbb{R}^3$, aufgeteilt in Tetraeder und Nédélec-Elemente niedrigster Ordnung[5]

$$V_h \subset H_0(\mathrm{curl}, \Omega) = \{\boldsymbol{u} \in L_2(\Omega, \mathbb{R}^3) \colon \nabla \times \boldsymbol{u} \in L_2(\Omega, \mathbb{R}^3), \boldsymbol{u} \times \boldsymbol{n} = 0 \text{ auf } \partial\Omega\}\,.$$

Weiterhin seien $Q_h \subset H_0^1(\Omega)$ lineare Lagrange-Elemente. Damit lässt sich das diskrete Eigenwertproblem in schwacher Form schreiben als:

Finde Eigenfunktionen $\boldsymbol{u}_h \in V_h \setminus \{0\}$ und Eigenwerte $\lambda_h \in \mathbb{R}$, so dass

$$\begin{aligned} (\varepsilon_r^{-1} \nabla \times \boldsymbol{u}_h, \nabla \times \boldsymbol{v}_h)_0 &= \lambda_h (\boldsymbol{u}_h, \boldsymbol{v}_h)_0, & \boldsymbol{v}_h &\in V_h \\ (\boldsymbol{u}_h, \nabla q_h)_0 &= 0 & q_h &\in Q_h\,[6]\,. \end{aligned}$$

Die curl-bilinear-Form ist symmetrisch und elliptisch in

$$\boldsymbol{X}_h = \{\boldsymbol{v}_h \in V_h \colon \operatorname{div} \boldsymbol{v}_h = 0\}\,,$$

somit sind alle Eigenwerte positiv. Mit den diskreten Operatoren A_h, M_h, B_h und C_h, wobei

$$\begin{aligned} \langle A_h \boldsymbol{u}_h, \boldsymbol{v}_h \rangle &= (\epsilon^{-1} \mathrm{curl}\ \boldsymbol{u}_h, \mathrm{curl}\ \boldsymbol{v}_h)_0\,, & \langle M_h \boldsymbol{u}_h, \boldsymbol{v}_h \rangle &= (\boldsymbol{u}_h, \boldsymbol{v}_h)_0\,, \\ \langle B_h \boldsymbol{u}_h, q_h \rangle &= (\boldsymbol{u}_h, \nabla q_h)_0\,, & \langle C_h p_h, q_h \rangle &= (\nabla p_h, \nabla q_h)_0 \end{aligned}$$

mit $\boldsymbol{u}_h, \boldsymbol{v}_h \in V_h$ und $p_h, q_h \in Q_h$, lässt sich das Eigenwertproblem schreiben als

$$\begin{aligned} A_h \boldsymbol{u}_h &= \lambda_h M_h \boldsymbol{u}_h \\ B_h \boldsymbol{u}_h &= 0\,. \end{aligned}$$

Zur Lösung dieses Eigenwertproblems verwenden wir eine modifizierte LOBPCG[7]-Methode, welche von Knyazev [Kny01] beschrieben wurde. Diese Methode beschreibt eine Dreiterm-Rekursion mit einer vorkonditionierten projizierten Inversen Methode, welche in [Zag06] vorgestellt wird. In diesem Algorithmus wird eine Zerlegung von C_h benötigt (sogenanntes Laplace-Problem in der Projektion für den Eigenwertlöser), sowie eine Zerlegung der regularisierten Matrix $A_h^\delta = A_h + \delta M_h$ (sogenanntes Laplace-Problem). Beide Zerlegungen werden mit Hilfe der parallelen Block-LR-Zerlegung berechnet, wobei für größere Probleme ein Mehrgitter-Vorkonditionierer vorgeschaltet und diese Zerlegung auf dem gröbsten Level vollzogen wird. Die zugehörigen Matrizen sind symmetrisch.

6.6. Elektromagnetische Wellengleichung mit einem Ansatz der Discontinuous Galerkin Methode

Für ein aufwändigeres Problem betrachten wir die elektromagnetische Wellengleichung (6.8) mit einem Discontinuous-Galerkin-Ansatz. Es soll hier hauptsächlich die Problemstellung und die Lösungsidee dargestellt werden, daher wird auf eine genaue Herleitung und die Theorie dahinter verzichtet. Für weitere Informationen siehe [HW07, LeV02, LeV92, DPE11].

Der Unterschied zu den anderen Problemen liegt hier insbesondere in der Definition der Knotenpunkte. Diese sind im Zellmittelpunkt gegeben, wobei dieser theoretisch nur auf einem Prozessor vorhanden ist. Ein Eintrag in der Steifigkeitsmatrix ergibt sich jedoch durch zwei benachbarte Zellen. Sofern diese auf verschiedenen Prozessoren zu finden sind, werden beide Zellen als Interface-Zellen beschrieben. Dadurch ergibt sich letztendlich ein insgesamt größeres Interface im Gegensatz zu

[5] die Freiheitsgrade beschreiben das Integral der Tangentialkomponente entlang der Kanten des Tetraeders

[6] $(\cdot,\cdot)_0$ bezeichnet das Skalarprodukt in $L_2(\Omega)^3$

[7] Locally Optimal Preconditioned Conjugate Gradient

den bisherigen Problemstellungen. Dort befindet sich das Interface bei linearem Ansatz jeweils auf dem Rand der Zellen.

6.6.1. Problemstellung.
Wir betrachten nun wieder die Maxwell'schen Gleichungen als System von linearen partiellen Differentialgleichungen erster Ordnung im Intervall $[0,T]$.

$$\begin{aligned}\varepsilon_0 \frac{\partial \boldsymbol{E}}{\partial_t} - \nabla \times \boldsymbol{H} &= 0\\ \mu_0 \frac{\partial \boldsymbol{H}}{\partial_t} + \nabla \times \boldsymbol{E} &= 0\\ \nabla \cdot (\mu_0 \boldsymbol{H}) &= 0\\ \nabla \cdot (\varepsilon_0 \boldsymbol{E}) &= 0\,.\end{aligned}$$

Mit

$$\begin{aligned}M(\boldsymbol{H},\boldsymbol{E}) &= (\mu_0 \boldsymbol{H}, \varepsilon \boldsymbol{E}),\\ A(\boldsymbol{H},\boldsymbol{E}) &= (\operatorname{curl} \boldsymbol{E}, -\operatorname{curl} \boldsymbol{H}),\end{aligned}$$

dem Definitionsbereich

$$D(A) = \{(\boldsymbol{H},\boldsymbol{E}) \in H(\operatorname{curl},\Omega) \times H_0(\operatorname{curl},\Omega)\colon \operatorname{div}(\varepsilon_0 \boldsymbol{E}) = 0, \quad \operatorname{div}(\mu_0 \boldsymbol{H}) = 0\}$$

für A und dem Hilbertraum $V = L_2(\Omega)^3 \times L_2(\Omega)^3$ mit $D(A) \subset V$ und dem Skalarprodukt $(v,w)_V = (Mv,w)_0$ lässt sich das Problem auch schreiben als

$$\begin{aligned}&\text{Finde } \boldsymbol{u} \in D(A), \text{ so dass}\\ &M\partial_t \boldsymbol{u} + A\boldsymbol{u} = 0 \qquad \text{in } [0,T]\end{aligned} \tag{6.9}$$

Hier kann A aufgrund der Energieerhaltung definiert werden über einen linearen Fluss $\boldsymbol{F}$, so dass

$$A\boldsymbol{u} = \operatorname{div} \boldsymbol{F}(\boldsymbol{u}) = \sum_{d=1}^{3} B_d \partial_d \boldsymbol{u}\,,$$

mit symmetrischen Matrizen $B_d \in \mathbb{R}^{6\times 6}$ gilt.[8]

6.6.2. Discontinuous Galerkin-Methode für die Wellengleichung.
Sei $\Omega = \bigcup_{c\in\mathcal{C}} \Omega_c$ eine Diskretisierung des Gebietes Ω in Tetraeder mit $h = \max \operatorname{diam}(\Omega_c)$. Die zugehörigen Seitenflächen einer Zelle c wird mit $\mathcal{F}_c$ bezeichnet und zu einer Seitenfläche f einer Zelle c sei die Nachbarzelle mit c_f bezeichnet. Die äußere Einheitsnormale von f wird mit $\boldsymbol{n}_{c,f}$ bezeichnet und wir setzen $[\boldsymbol{v}]_{c,f} = \boldsymbol{v}_{c_f} - \boldsymbol{v}_c$ als Sprung.

Die diskretisierte Gleichung von (6.9) ist gegeben als

$$M\partial_t \boldsymbol{u}_h(t) + A_h \boldsymbol{u}_h(t) = 0 \quad t \in [0,T] \tag{6.10}$$

in $V_h \subset V$ mit

$$V_h = \{\boldsymbol{v}_h \in L_2(\Omega,\mathbb{R}^6)\colon \boldsymbol{v}_h|_c \in V_{c,h}\}\,.$$

Hierbei bezeichnet $V_{c,h} = \mathbb{P}_p(c)^6$ den zugehörigen Polynomraum und für $\boldsymbol{v}_h \in V_h$ wird mit $\boldsymbol{v}_{c,h} = \boldsymbol{v}_h|_c \in V_{c,h}$ die Restriktion von $\boldsymbol{v}_h$ auf die Zelle c bezeichnet.

[8] es existieren jeweils 3 Freiheitsgrade in jede Richtung für $\boldsymbol{E}$ und $\boldsymbol{H}$

Der diskretisierte Operator $A_h \in \mathcal{L}(V_h, V_h)$ ist gegeben durch eine Discontinuous Galerkin-Diskretisierung mit Upwind-Fluss [**HW07**, **DPE11**]. Für diesen ergibt sich mit $(\boldsymbol{H}_h, \boldsymbol{E}_h) \in V_h$ und $(\boldsymbol{\varphi}_{c,h}, \boldsymbol{\psi}_{c,h}) \in V_{c,h}$

$$\begin{aligned}\big(A_h(\boldsymbol{H}_h, \boldsymbol{E}_h), (\boldsymbol{\varphi}_{c,h}, \boldsymbol{\psi}_{c,h})\big)_{0,c} &= (\operatorname{curl} \boldsymbol{E}_{c,h}, \boldsymbol{\varphi}_{c,h})_{0,c} - \big(\operatorname{curl} \boldsymbol{H}_{c,h}, \boldsymbol{\psi}_{c,h}\big)_{0,c} \\ &+ \sum_{f \in \mathcal{F}_c} \Big[\frac{\mathrm{c}_{c_f} \varepsilon_{c_f}}{\mathrm{c}_c \varepsilon_c + \mathrm{c}_{c_f} \varepsilon_{c_f}} \big(\boldsymbol{n}_{c,f} \times [\boldsymbol{E}_h]_{c,f}, \boldsymbol{\varphi}_{c,h}\big)_{0,f} \\ &\quad - \frac{\mathrm{c}_{c_f} \mu_{c_f}}{\mathrm{c}_c \mu_c + \mathrm{c}_{c_f} \mu_{c_f}} \big(\boldsymbol{n}_{c,f} \times [\boldsymbol{H}_h]_{c,f}, \boldsymbol{\psi}_{c,h}\big)_{0,f} \\ &\quad + \frac{1}{\mathrm{c}_c \mu_c + \mathrm{c}_{c_f} \mu_{c_f}} \big(\boldsymbol{n}_{c,f} \times (\boldsymbol{n}_{c,f} \times [\boldsymbol{E}_h]_{c,f}), \boldsymbol{\psi}_{c,h}\big)_{0,f} \\ &\quad + \frac{1}{\mathrm{c}_c \varepsilon_c + \mathrm{c}_{c_f} \varepsilon_{c_f}} \big(\boldsymbol{n}_{c,f} \times (\boldsymbol{n}_{c,f} \times [\boldsymbol{H}_h]_{c,f}), \boldsymbol{\varphi}_{c,h}\big)_{0,f} \Big] .\end{aligned}$$

Für die Randbedingungen an den Seitenflächen am Rand der Geometrie $f = \partial\Omega_c \cap \partial\Omega$ wird

$$\begin{aligned}\boldsymbol{n}_{c,f} \times \boldsymbol{E}_{c_f} &= -\boldsymbol{n}_{c,f} \times \boldsymbol{E}_c \\ \boldsymbol{n}_{c,f} \times \boldsymbol{H}_{c_f} &= \boldsymbol{n}_{c,f} \times \boldsymbol{H}_c\end{aligned}$$

gesetzt.

6.6.3. Zeitintegration über eine implizite Runge-Kutta-Methode. Insgesamt ergibt sich ein lineares Problem

$$\mathbb{M}\partial_t \boldsymbol{u} + \mathbb{A}\boldsymbol{u} = 0 \qquad u(0) = u_0$$

mit Matrizen $\mathbb{M}, \mathbb{A} \in \mathbb{R}^{N \times N}$. Die Massenmatrix $\mathbb{M} = ((M\boldsymbol{\phi}_m, \boldsymbol{\phi}_n)_{0,\Omega})$ ist symmetrisch positiv definit und insbesondere eine Block-Diagonal-Matrix. Die Steifigkeitsmatrix $\mathbb{A} = ((\boldsymbol{\phi}_m, A_h\boldsymbol{\phi}_n)_{0,\Omega})$ ist nahezu schiefsymmetrisch.

Die Lösung dieses Systems ist gegeben durch

$$u(t) = \exp(-t\mathbb{M}^{-1}\mathbb{A})u_0, \quad t \geq 0 .$$

Für einen festen Zeitschritt $\tau = T/K$ setzen wir $t_n = n\tau$ und eine Approximation für den nächsten Zeitschritt ist durch

$$u(t_{n+1}) = \exp(-\tau\mathbb{M}^{-1}\mathbb{A})u(t_n)$$

gegeben. Mit Hilfe der impliziten Mittelpunktsregel als Runge-Kutta-Methode setzen wir

$$u^{n+1} = u^n - \tau(\mathbb{M} + \frac{\tau}{2}\mathbb{A})^{-1}\mathbb{A}u^n .$$

Um $u(T)$ zu berechnen muss also K mal die Inverse von $\mathbb{M} + \frac{\tau}{2}\mathbb{A}$ angewandt werden. Dazu genügt jedoch eine einmalige Zerlegung, so dass es sich für viele Zeitschritte lohnt, diese Zerlegung mit Hilfe der parallelen Block-LR-Zerlegung zu berechnen.

7 Anwendung des parallelen direkten Lösers

Wir betrachten nun die Anwendung des parallelen direkten Lösers (Algorithmus 3.30) auf die Modellprobleme aus Kapitel 6. Sofern nicht anders angegeben, wurden alle Rechnungen auf dem Parallelcluster `HERMIT` in Stuttgart (vergleiche Anhang C) durchgeführt.

7.1. Poisson-Gleichung

Wir betrachten das Poisson-Problem aus Kapitel 6.1, angewandt auf einem Würfel $\Omega = (0,1)^3$. Aufgrund der einfachen Geometrie ergibt sich eine sehr gleichmäßige Aufteilung auf die einzelnen Prozessoren (vergleiche auch Kapitel 5 und Abbildung 1).

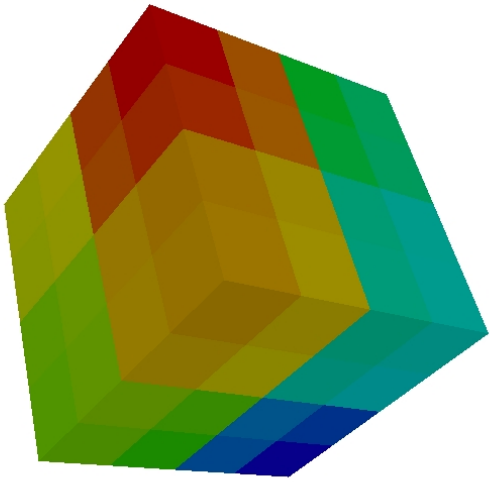


Abbildung 1. Aufteilung des Würfels $\Omega = (0,1)^3$ auf 64 Prozessoren

Um eine gewisse Mindestgröße für dieses Problem zu erhalten, betrachten wir eine fünffache Verfeinerung des Einheitswürfels. Damit erhalten wir eine Gesamtgröße von $N = 35\,937$ Unbekannten (vergleiche Tabelle 1 aus Kapitel 5). Die Zerlegung mit Hilfe des Sparse-Lösers `SuperLU` auf einem Prozessor benötigt dabei etwas mehr als eine Minute. Auf 32 Prozessoren wird ein Optimum erreicht, wobei

die Zerlegung dort 0.55 Sekunden benötigt. Von da an pendelt sich die Zerlegung bei etwa einer Sekunde ein. Die Mindestmatrixgröße auf den einzelnen Prozessoren wurde für alle Zerlegungen auf $N_{\mathrm{LR}} = 17$ gesetzt, um die Kommunikationsdauer insbesondere bei kleinen Problemgrößen zu senken. Damit wurden für eine große Prozessorzahl in den letzten Schritten nur relativ wenige Prozessoren zum Lösen benötigt.

Für eine weitere Verfeinerung ergeben sich 274 625 Unbekannte. Der Sparse-Löser `SuperLU` benötigt auf einem Prozessor dabei etwas mehr als zwei Stunden. Um eine rechte Seite zu lösen, benötigt `SuperLU` mit der Zerlegung dieser Sparse-Matrix etwa 5 Sekunden. Hier wird auch die Abhängigkeit des Sparse-Lösers für eine Verteilung auf wenige Prozessoren deutlich: In Schritt $s = 0$ ist die Sparse-Matrix relativ groß, da nur wenige Prozessoren beteiligt sind. Zusätzlich muss die Zerlegung der Sparse-Matrix auf viele rechte Seiten angewandt werden (nämlich auf alle Elemente, die auf dem Rand der Prozessormenge liegen). Insgesamt wird bei einer kleinen Prozessorzahl die meiste Zeit im ersten Schritt benötigt. Um die Berechnungszeit zu optimieren ist hier also ein "guter" Sparse-Löser notwendig. Erst wenn genug Prozessoren zur Verfügung stehen, fällt der Sparse-Löser nicht mehr ins Gewicht, jedoch erhöht sich die Kommunikationsdauer im Verlauf des Algorithmus (vergleiche auch Tabelle 8 und 9 auf Seite 90 bzw. 91).

In Kapitel 5 wurde für dieses Poisson-Problem eine Komplexitätsanalyse durchgeführt. Dort wurde jeder einzelne Schritt genau betrachtet, in Tabelle 1 wird für jede Zerlegung die Gesamtzeit der Vorhersage angegeben. Die Berechnungen erfolgen dabei durch (5.17), (5.18) und (5.19) aus Kapitel 5.9:

$$T_{\mathrm{Vorhersage}} = \sum_{s=0}^{S} T_{\mathrm{ber},s} + T_{\mathrm{bc},s} + T_{\mathrm{one},s}\,. \tag{7.1}$$

Es ist ersichtlich, dass die Abschätzung ein guter Anhaltspunkt zur tatsächlichen Berechnungsdauer ist.

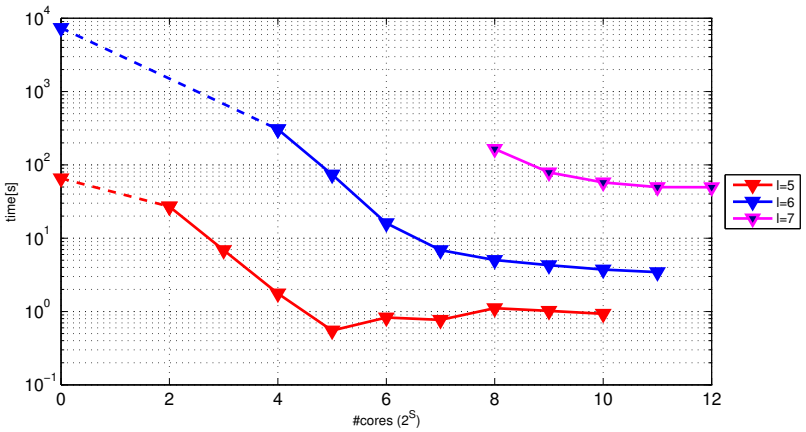


ABBILDUNG 2. Zerlegungszeiten des Poisson-Problems

Insgesamt wird ein "Optimum" der Zerlegungszeit für eine bestimmte Prozessorzahl in Abhängigkeit der Problemgröße erwartet. Diese optimale Prozessorzahl ist umso höher, je größer das zu lösende Problem ist. Für Level 5 mit 35 937 Unbekannten ist das Optimum bei $P_{\mathrm{best}} = 32$ Prozessoren erreicht, während für Level 6 mit 274 625 Unbekannten ein Optimum erst bei etwa 128 Prozessoren erreicht ist. Für mehr Prozessoren wird zwar noch eine kleine Verbesserung erzielt, diese ist

jedoch nicht mehr allzu ausschlaggebend. Für das größte betrachtete Problem mit über 2 Millionen Unbekannten sind schon 1024 Prozessoren sinnvoll. Hierbei stand für weniger als 256 Prozessoren nicht genug Speicher zur Verfügung, so dass diese Berechnungen nicht durchführbar waren.

Level	Unbekannte			
5	35 937			
# Proc	Zerlegungszeit	Vorhersage	Lösungszeit	Speedup[1]
1	1:05.58 min.	1:05.19 min.	0.20 sec.	
4	27.05 sec.	29.86 sec.	0.07 sec.	2.4
8	6.89 sec.	7.00 sec.	0.03 sec.	3.9
16	1.76 sec.	2.40 sec.	0.01 sec.	3.9
32	0.55 sec.	0.78 sec.	0.01 sec.	3.2
64	0.83 sec.	0.36 sec.	0.01 sec.	
128	0.77 sec.	0.25 sec.	0.01 sec.	
256	1.11 sec.	0.22 sec.	0.01 sec.	
512	1.02 sec.	0.21 sec.	0.01 sec.	
1024	0.93 sec.	0.21 sec.	0.01 sec.	

Level	Unbekannte			
6	274 625			
# Proc	Zerlegungszeit	Vorhersage	Lösungszeit	Speedup
1	2:03:25 std.	2:04:54 std.	5.12 sec.	
16	5:06.06 min.	4:21.65 min.	0.42 sec.	24.1
32	1:13.94 min.	1:08.05 min.	0.23 sec.	4.1
64	16.06 sec.	18.89 sec.	0.21 sec.	4.6
128	6.87 sec.	7.45 sec.	0.17 sec.	2.3
256	5.05 sec.	4.23 sec.	0.17 sec.	1.4
512	4.29 sec.	3.33 sec.	0.18 sec.	1.2
1024	3.73 sec.	3.01 sec.	0.18 sec.	1.2
2048	3.44 sec.	2.90 sec.	0.17 sec.	

Level	Unbekannte			
7	2 146 689			
# Proc	Zerlegungszeit	Vorhersage	Lösungszeit	Speedup
256	2:45.30 min.	3:00.23 min.	2.09 sec.	
512	1:19.11 min.	1:31.59 min.	2.04 sec.	2.0
1024	57.73 sec.	1:03.61 min.	2.01 sec.	1.4
2048	49.75 sec.	52.31 sec.	2.20 sec.	1.1
4096	49.55 sec.	48.31 sec.	2.20 sec.	

TABELLE 1. Berechnungszeiten der parallelen Block-LR-Zerlegung auf dem Parallelcluster `HERMIT` in Stuttgart für das Poisson-Problem; Vorhersage nach (7.1) mit Zeitabschätzungen aus Kapitel 5.9

[1]Der Speedup bezieht sich auf die Zerlegung im Vergleich zur jeweils vorangegangenen Prozessorzahl

7.2. Stokes-Gleichung (2D)

Wir betrachten das Stokes-Problem (6.1) auf das Beispiel 5.1.2 aus [ESW05]. Gegeben ist das Gebiet Ω als liegendes L (vergleiche Abbildung 3). Am linken Rand ($x = -1, 0 \leq y \leq 1$) wird eine Einflussbedingung und am rechten Rand ($x = 4, -1 < y < 1$) eine Ausflussbedingung als Neumann-Randbedingung gestellt, wobei der Druck p hier zu Null gesetzt wird. An den Wänden (oberer und unterer Rand der Geometrie) wird eine Dirichlet-Nullrandbedingung gesetzt.

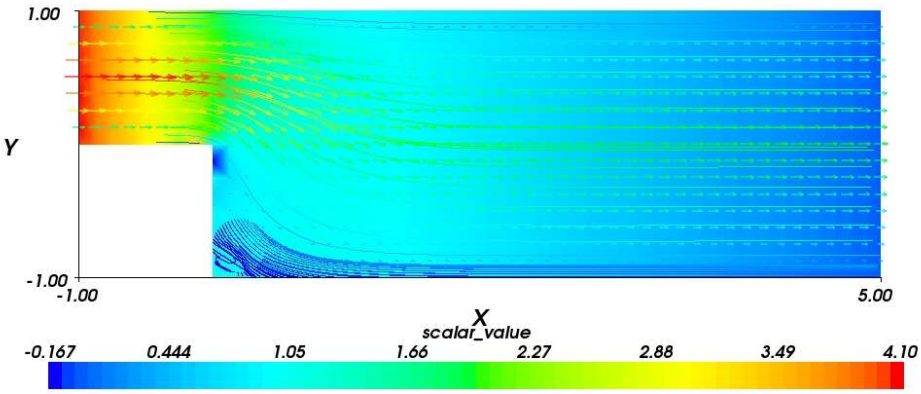


ABBILDUNG 3. Strömungslinien und Druck (farblicher Hintergrund) des Stokes-Problems

Da hier ein zweidimensionales Modell betrachtet wird, ergeben sich nur eindimensionale Interfaces. Diese sind somit sehr klein im Vergleich zu dreidimensionalen Problemen, wo sich zweidimensionale Interfaces ergeben. Zusätzlich ist die Matrix noch dünner besetzt, so dass in Schritt $s = 0$ der Sparse-Löser eine weitaus bessere Leistung liefern kann. Insgesamt erwarten wir eine sehr gute Performance in den Zerlegungszeiten.

Für das Stokes-Problem ergeben sich auch ausgezeichnete Zeiten für die Zerlegung, insbesondere wird eine Verbesserung der Zerlegungszeit auch für größere Prozessorzahlen selbst bei relativ kleinen Problemen erreicht. Für das größte berechnete Problem mit über 20 Millionen Unbekannten benötigt der parallele direkte Löser auf 4096 Prozessoren nur 24 Sekunden. Hierbei muss zusätzlich beachtet werden, dass die Verteilung der Zellen auf die Prozessoren nicht immer optimal ist[2]. Nichtsdestotrotz lohnen sich bei diesem Beispiel eine große Anzahl an verwendeten Prozessoren. Wie im Beispiel des Poisson-Problems steigt die optimale Prozessorzahl an, wenn das Problem größer wird. Für 317 955 Unbekannte sind zwischen 512 und 1024 Prozessoren sinnvoll, für das größte Problem mit 20 Millionen Unbekannten sollten $P_{\text{best}} = 4096$ Prozessoren genutzt werden. Weiterhin waren für größere Probleme auch eine Mindestgröße an Prozessoren notwendig, da ansonsten nicht genug Speicher zur Verfügung stand.

[2]Dies liegt insbesondere an der ausgeschnittenen Ecke des Rechtecks

level	5	6	7	8	9
Unbekannte	80 131	317 955	1 266 961	5 056 515	20 205 571
# Proc					
8	13.65 sec.				
16	4.25 sec.	53.86 sec.			
32	1.31 sec.	17.50 sec.			
64	0.66 sec.	5.88 sec.	1:06.44 min.		
128	0.43 sec.	2.06 sec.	20.43 sec.		
256	0.64 sec.	1.39 sec.	7.30 sec.	1:18.91 min.	
512	0.73 sec.	1.06 sec.	3.04 sec.	24.96 sec.	
1024	0.61 sec.	0.89 sec.	2.23 sec.	9.76 sec.	1:38.87 min.
2048		0.78 sec.	1.99 sec.	6.63 sec.	40.10 sec.
4096					23.99 sec.

TABELLE 2. Berechnungszeiten der parallelen Block-LR-Zerlegung auf dem Parallelcluster `HERMIT` in Stuttgart für das Stokes-Problem

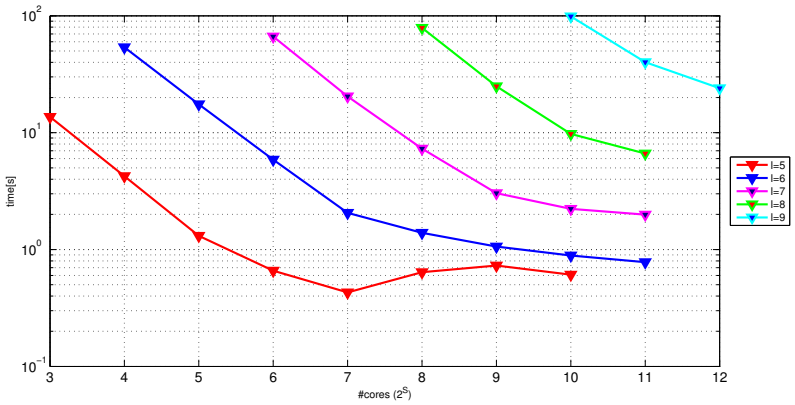


ABBILDUNG 4. Zerlegungszeiten für das Stokes-Problem

7.3. Elastizität

Wir betrachten nun das Modellproblem aus [Nie12] als Elastizitäts-Problem. Es wird ein menschliches Herz modelliert und ein Teil des Herzschlags simuliert. Dazu wird ein Newton-Verfahren verwendet, wobei in jedem Schritt eine neue parallele Block-LR-Zerlegung benötigt wird. Die in Tabelle 3 angegebenen Zerlegungszeiten beziehen sich dabei auf jeweils eine Zerlegung.
Da das Herzmodell eine relativ komplexe Geometrie beschreibt, ist eine optimale Verteilung auf die Prozessoren ausschlaggebend für die Zerlegungszeiten. Die "Herzwand" ist relativ dünn (siehe Abbildung 5), somit lassen sich mit einer guten Verteilung relativ kleine Interfaces erzeugen. Für solche Geometrien ist ein guter Graphpartitionierer erforderlich, daher erfolgt die Verteilung der Zellen auf die Prozessoren über `Kaffpa` [SS11].
Für ein Problem mit über 1.4 Millionen Unbekannten werden auf 512 Prozessoren nur 20 Sekunden benötigt. Es tritt hier sogar noch eine weitere Verbesserung auf, wenn noch mehr Prozessoren benutzt werden.

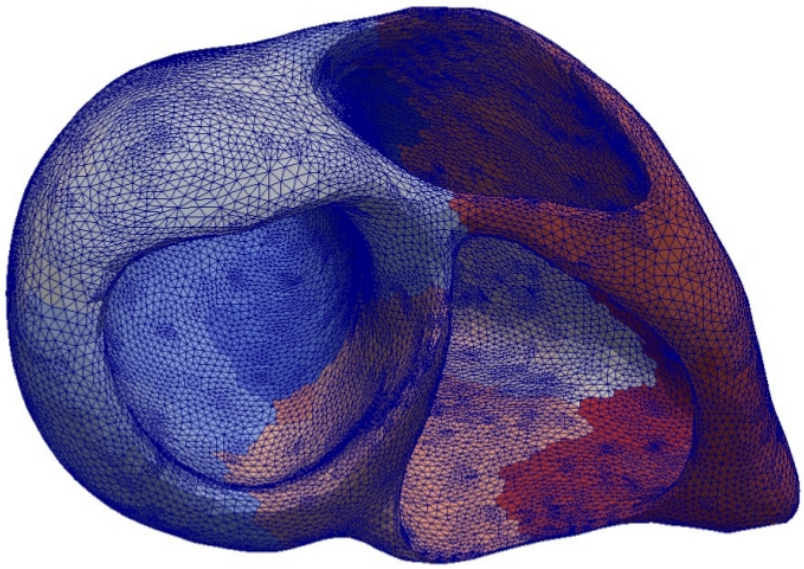


ABBILDUNG 5. Herzmodell - Gitter mit Gebietszerlegung

Level	Unbekannte	# Proc	Zerlegungszeit	Lösungszeit
1	205 884	16	1:06.62 min.	0.15 sec.
		32	18.97 sec.	0.10 sec.
		64	3.52 sec.	0.07 sec.
		128	1.82 sec.	0.07 sec.
		256	1.31 sec.	0.07 sec.
		512	1.05 sec.	0.10 sec.
		1024	1.15 sec.	0.11 sec.
		2048	1.74 sec.	0.18 sec.
2	1 424 628	128	1:51.53 min.	0.70 sec.
		256	40.00 sec.	0.58 sec.
		512	20.50 sec.	0.60 sec.
		1024	16.50 sec.	0.70 sec.
		2048	13.50 sec.	0.80 sec.

TABELLE 3. Berechnungszeiten der parallelen Block-LR-Zerlegung auf dem Parallelcluster `HERMIT` in Stuttgart für das Elastizitäts-Problem am Herz-Beispiel

7.4. Maxwell'sche Gleichungen

7.4.1. Eigenwertproblem.

Wir betrachten das Maxwell'sche Eigenwertproblem aus Kapitel 6.5 an einer Geometrie eines Containers mit Lüftungsschlitz (vergleiche Abbildung 6). Diese Geometrie wurde uns im Rahmen des ASIL-Projekts von `CST` zur Verfügung gestellt. Es handelt sich hierbei um ein Problem der elektromagnetischen Verträglichkeit, wobei durch den Lüftungsschlitz Felder in den Außenraum eindringen können, wo sie beispielsweise andere Geräte stören können. In der Regel werden diese Problemstellungen als angeregtes Problem simuliert, hierbei erfolgt die Anregung über eine Koaxialleitung. Für die Anordnung gilt $\varepsilon_r = 1$, $\mu_r = 1$ (Vakuum), im Koaxialkabel

weichen die Materialparameter ab, dort gilt $\varepsilon_r = 1.419$ und $\mu_r = 1$. Die Geometrie ist in unstrukturierte Tetraeder unterteilt. Die Koaxialleitung selbst ist deutlich feiner unterteilt als der Rest.

In diesem Fall wird der parallele direkte Löser als Grobgitterlöser für ein Mehrgitterverfahren eingesetzt. Außerdem betrachten wir hier sowohl lineare als auch quadratische Elemente. Da die Eigenwertiteration häufig durchgeführt werden muss um ein adäquates Ergebnis zu erhalten, wird das Mehrgitterverfahren und damit der parallele direkte Löser mehrfach angewendet. Jedoch muss insgesamt nur eine Zerlegung durchgeführt werden.

In [MW12] findet sich die Vorgehensweise zur Lösung des Maxwell'schen Eigenwertproblems, wobei das iterative LOBPCG-Verfahren verwendet wird. Dabei wird in jedem Schritt das Mehrgitterverfahren aufgerufen und es können mehrere Eigenwerte gleichzeitig berechnet werden. Das heißt für die parallele Block-LR-Zerlegung, dass diese auch auf mehrere rechte Seiten gleichzeitig angewendet werden kann. Für das Mehrgitterverfahren selbst verwenden wir eine "nested iteration". Hierbei wird das Problem zunächst auf dem gröbsten Gitter gelöst[3] und die Lösung als Initialwert für die nächstfeinere Zerlegung verwendet. Dies wird iterativ fortgeführt, bis die feinste Zerlegung erreicht ist[4].

Der Vorteil ist, dass die parallele Block-LR-Zerlegung als Grobgitterlöser in diesem Fall nur einmal berechnet werden muss, weiterhin benötigt das Verfahren auf dem feinsten Gitter durch den besseren Initialwert deutlich weniger Iterationen, was sich auch in der Gesamtzeit auswirkt.

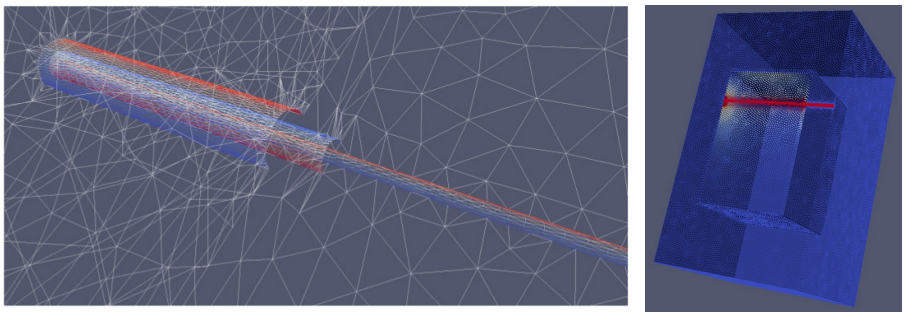


Abbildung 6. Geometrie des Containers

Für die reine Zerlegung des Maxwell- beziehungsweise Laplace-Problems ergeben sich im Fall von linearen Elementen die Daten aus Tabelle 4. Es muss allerdings bemerkt werden, dass die Erstellung der Datenstruktur (das heißt, insbesondere die Belegung des Speichers der einzelnen Matrizen und vor allem die Erstellung der Kommunikationsmodule) ab 2048 Prozessoren hier sehr viel Zeit in Anspruch nimmt (für 2048 Prozessoren etwa 1 Minute, für 4096 Prozessoren knapp 10 Minuten). Woran dies liegt konnte nicht abschließend geklärt werden, vor allem, da es bei anderen Problemklassen, wie zum Beispiel beim Poisson-Problem diese Diskrepanzen nicht gegeben hat[5]. Nichtsdestotrotz ergibt sich für die Zerlegung des Maxwell-Problems mit quadratischen Ansatz-Elementen auf 4096 Prozessoren eine bemerkenswerte Zerlegungszeit von 63 Sekunden für über 3.7 Millionen Unbekannte.

[3] Dabei findet noch kein wirkliches Mehrgitterverfahren statt

[4] Da das Grobgitter in diesem Beispiel schon relativ fein ist, kann hier nur maximal einmal verfeinert werden - für andere Testprobleme ergaben sich deutliche Geschwindigkeitsvorteile

[5] Insbesondere die Erstellung des ersten Schritts inklusive aller Kommunikationsmodule scheint problematisch zu sein

# Proc	Maxwell 698 231	Laplace 105 629
256	11.36 sec.	0.83 sec.
512	7.62 sec.	1.10 sec.
1024	7.71 sec.	1.68 sec.
2048	8.88 sec.	2.60 sec.

TABELLE 4. Zerlegungszeit mit linearen Elementen

# Proc	Maxwell 3 721 946	Laplace 803 860
512	2:31 min.	14.31 sec.
1024	1:27 min.	13.91 sec.
2048	1:23 min.	16.62 sec.
4096	1:03 min.	23.02 sec.

TABELLE 5. Lösung mit quadratischen Elementen

7.4.2. Wellengleichung.

Als abschließendes Problem betrachten wir einen impliziten Zeitschritt für die Wellengleichung, welche sich aus den Maxwell'schen Gleichungen ergibt (vergleiche Kapitel 6.6). Durch den Discontinuous-Galerkin-Ansatz befinden sich die Knotenpunkte in den Zellmittelpunkten, welche theoretisch nur auf einem Prozessor gegeben sind, für die Anwendung der parallelen Block-LR-Zerlegung müssen diese jedoch als echtes Interfaceelement betrachtet werden, sollte sich eine Nachbarzelle auf einem anderen Prozessor befinden.

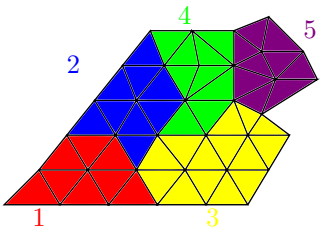


ABBILDUNG 7. Gebiet Ω verteilt auf 5 Prozessoren für die Discontinuous Galerkin-Methode

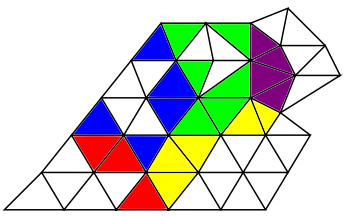


ABBILDUNG 8. Interface-Elemente

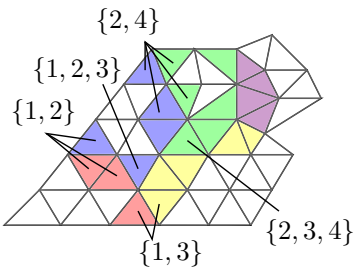


ABBILDUNG 9. Interface-Elemente mit Prozessormengen

Wir betrachten zur Vereinfachung ein Gebiet Ω im Zweidimensionalen, welches in Dreiecke aufgeteilt und auf 5 Prozessoren verteilt ist.

Die Interface-Elemente in Abbildung 8 sind farblich hervorgehoben, wobei sich die Zellmittelpunkte jeweils nur auf einem Prozessor befinden.

Für den Algorithmus müssen die Prozessormengen angepasst werden, zu sehen ist in Abbildung 9 eine Auswahl, wie sich die zugehörigen Prozessormengen einzelner Zellen ergeben. Zur Bestimmung der eigentlichen Prozessormenge des Index innerhalb der Zelle genügt es somit, die jeweiligen Prozessormengen der Kanten (beziehungsweise im vorliegenden Fall des Dreidimensionalen die Seitenflächen) zu vereinigen.

Das heißt für eine Zelle c und zugehörigen Seitenflächen $f \in \mathcal{F}_c$ wird die zugehörige Prozessormenge $\pi(c)$ zu

$$\pi(z_c) = \bigcup_{f \in \mathcal{F}_c} \pi(z_f)$$

gesetzt, wobei $\pi(f)$ die schon existierenden Prozessormengen der Seitenflächen bezeichnet.

Für das Zweidimensionale Beispiel ergeben sich die Berechnungszeiten aus Tabelle 6. Hier fällt auf, dass die Lösungszeit für größere Probleme recht ungünstig ist, vor allem, da der Löser in jedem Zeitschritt im Algorithmus genutzt wird.

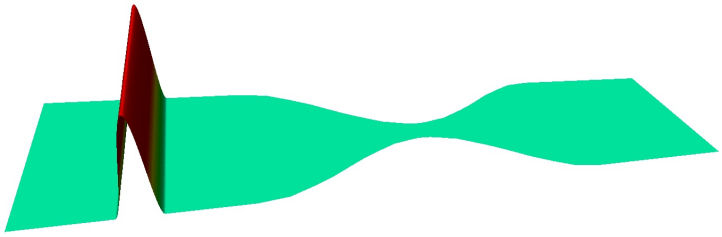


ABBILDUNG 10. Anfangskonfiguration für die Zweidimensionale Wellengleichung

Level	Unbekannte	# Proc	Zerlegungszeit	Lösungszeit
4	1 741 824	256	44.27 sec.	≈ 3 sec.
		512	27.18 sec.	≈ 2.2 sec.
		1024	18.79 sec.	≈ 2.5 sec.
		2048	17.34 sec.	≈ 2.4 sec.
5	6 967 296	1024	2:02.28 min.	≈ 11 sec.
		2048	1:25.61 min.	≈ 10 sec.
		4096	1:08.12 min.	≈ 9 sec.

TABELLE 6. Berechnungszeiten der parallelen Block-LR-Zerlegung auf dem Parallelcluster `HERMIT` in Stuttgart für die Wellengleichung im Zweidimensionalen

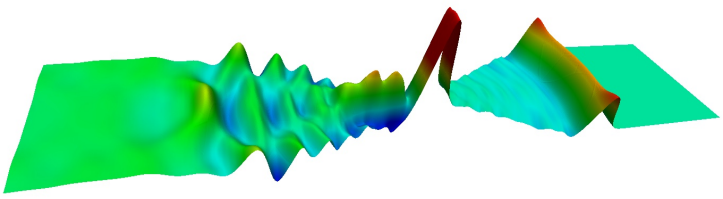


ABBILDUNG 11. Wellenberge nach 32 Zeitschritten

Für das Beispiel im Dreidimensionalen traten Speicherprobleme auf der `HERMIT` in Stuttgart auf, so dass diese Rechnungen auf dem Parallelcluster HC3 in Karlsruhe durchgeführt wurden. Dort konnten jedoch maximal 1024 Prozessoren benutzt werden (vergleiche Tabelle 7).

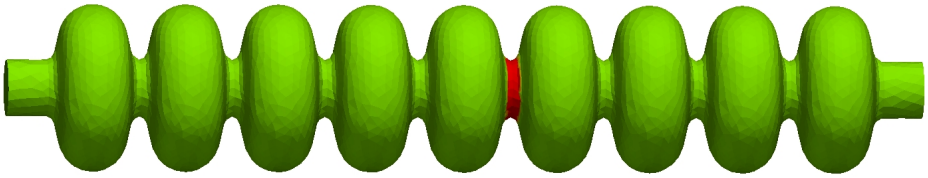


ABBILDUNG 12. Anfangskonfiguration für die Dreidimensionale Wellengleichung

Unbekannte	# Proc	Zerlegungszeit	Lösungszeit
783 804	512	3:24 min.	$\approx$ 1.5 sec.
	1024	2:58 min.	$\approx$ 2.5 sec.

TABELLE 7. Berechnungszeiten der parallelen Block-LR-Zerlegung auf dem Parallelcluster HC3 in Karlsruhe für die Wellengleichung im Dreidimensionalen

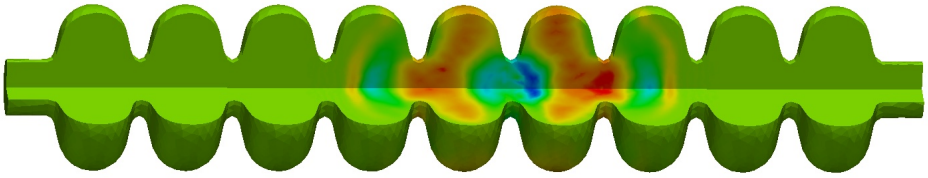


ABBILDUNG 13. Konfiguration nach 100 Zeitschritten (Schnitt)

7.5. Anwendung auf $P \neq 2^S$ Prozessoren

In diesem Kapitel soll die Anwendbarkeit von Algorithmus 3.30 mit $P \neq 2^S$ Prozessoren untersucht werden. Als Modellbeispiel verwenden wir hier das Poisson-Problem aus Kapitel 7.1 und betrachten die Effizienz zwischen 60 und 131 Prozessoren auf Level 6 (vergleiche Tabelle 1). Dieses Spanne ist aus dem Grund gewählt, da die Rechnungen eine annehmbare Zeit benötigen und der Geschwindigkeitsvorteil zwischen 64 und 128 Prozessoren groß genug ist. Es muss hier jedoch beachtet werden, dass die Gebietsverteilung für $P \notin \{64, 128\}$ nicht optimal ist, da der RCB-Algorithmus nur für eine Zweierpotenz Sinn macht. Zur Gebietszerlegung wurde der Graphpartitionierer `Kaffpa` [SS11] verwendet. Hierbei ergibt sich jedoch häufig eine Diskrepanz in der Verteilung der Zellen auf die Prozessoren.

In Tabelle 8 wird zusätzlich zu den Zerlegungszeiten auch das Minimum und Maximum der auf die Prozessoren verteilten Zellen angegeben. Für 64 und 128 Prozessoren ist die Verteilung optimal, so dass diese Zeiten auch ein Optimum im jeweiligen Bereich der verwendeten Prozessoren sind. Wir betrachtet zunächst den Bereich um $P = 64$. Der größte Unterschied zwischen 64 und 65 Prozessoren liegt neben der schlechteren Gebietszerlegung darin, dass ein zusätzlicher Schritt (insgesamt $S = 7$) benötigt wird und jeweils eine Prozessormenge in Schritt $s = 1$ bis $s = 6$ während der aktuellen Zerlegung keine Arbeit verrichten muss. In einer ähnlichen Weise setzt sich das auch für die darauf folgende Prozessoranzahl fort, bis ein Maximum bei etwa $P = 75$ Prozessoren erreicht ist. Für eine größere Prozessorzahl werden die Zerlegungszeiten nun immer besser, wobei hier noch mal darauf

aufmerksam gemacht werden muss, dass wegen der ungleichmäßigen Verteilung die Zeiten etwas schwanken[6]. Insbesondere bei $P = 117$ Prozessoren ist der Unterschied zwischen minimaler und maximaler Anzahl an Zellen je Prozessor sehr deutlich, so dass hier auch eine recht lange Zeit benötigt wird. Erst kurz vor $P = 128$ Prozessoren ist eine Verbesserung in der Zerlegungszeit spürbar, bei $P = 128$ Prozessoren wird ein Minimum durch die optimale Verteilung sprunghaft erreicht. Ein solcher Sprung ist auch zwischen $P = 63$ und $P = 64$ deutlich zu sehen.

Insgesamt ergibt sich – auch wenn die schlechtere Gebietsverteilung vernachlässigt wird – eine optimale Zerlegungszeit für $P = 2^S$ Prozessoren, so dass – wenn möglich – diese Anzahl gewählt werden sollte.

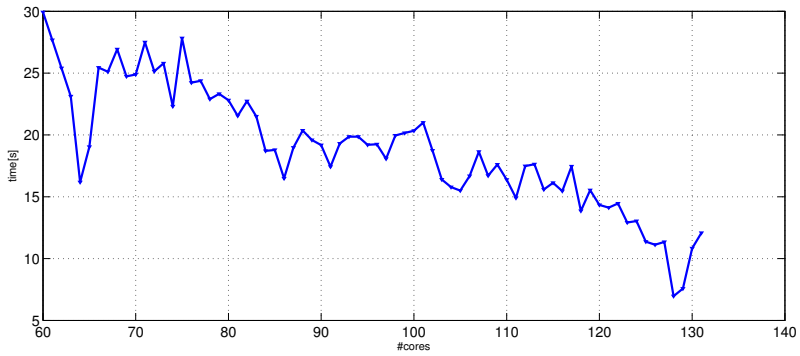


ABBILDUNG 14. Zerlegungszeiten für das Laplace-Problem auf 60-131 Prozessoren

[6]Es sollte daher sinnvollerweise die Durchschnittszeit über mehrere verwendete Prozessoren genommen werden

Procs	LR	Sol	min	max	Procs	LR	Sol	min	max
60	29.93 sec.	0.31 sec.	64	70	96	19.25 sec.	0.40 sec.	41	44
61	27.68 sec.	0.29 sec.	63	70	97	18.07 sec.	0.37 sec.	37	44
62	25.40 sec.	0.27 sec.	60	68	98	19.96 sec.	0.44 sec.	38	44
63	23.12 sec.	0.25 sec.	64	67	99	20.16 sec.	0.44 sec.	36	43
64	**16.18 sec.**	**0.16 sec.**	**64**	**64**	100	20.34 sec.	0.42 sec.	34	43
65	19.04 sec.	0.20 sec.	48	64	101	21.00 sec.	0.43 sec.	36	42
66	25.45 sec.	0.31 sec.	50	64	102	18.74 sec.	0.37 sec.	36	42
67	25.11 sec.	0.30 sec.	50	63	103	16.40 sec.	0.33 sec.	36	41
68	26.93 sec.	0.35 sec.	50	63	104	15.78 sec.	0.33 sec.	36	41
69	24.74 sec.	0.32 sec.	49	62	105	15.50 sec.	0.32 sec.	33	41
70	24.89 sec.	0.35 sec.	48	70	106	16.68 sec.	0.33 sec.	35	40
71	27.50 sec.	0.35 sec.	51	60	107	18.64 sec.	0.37 sec.	32	40
72	25.14 sec.	0.34 sec.	54	59	108	16.70 sec.	0.34 sec.	31	40
73	25.79 sec.	0.39 sec.	50	58	109	17.61 sec.	0.37 sec.	32	39
74	22.31 sec.	0.32 sec.	48	57	110	16.38 sec.	0.35 sec.	32	39
75	27.83 sec.	0.39 sec.	45	57	111	14.90 sec.	0.31 sec.	32	39
76	24.23 sec.	0.36 sec.	50	56	112	17.48 sec.	0.37 sec.	35	38
77	24.38 sec.	0.39 sec.	48	55	113	17.63 sec.	0.38 sec.	32	38
78	22.90 sec.	0.38 sec.	48	55	114	15.59 sec.	0.32 sec.	32	38
79	23.32 sec.	0.37 sec.	48	54	115	16.13 sec.	0.33 sec.	32	37
80	22.81 sec.	0.37 sec.	48	53	116	15.46 sec.	0.33 sec.	32	37
81	21.54 sec.	0.37 sec.	45	53	117	17.46 sec.	0.34 sec.	24	36
82	22.73 sec.	0.39 sec.	48	52	118	13.85 sec.	0.28 sec.	30	36
83	21.50 sec.	0.34 sec.	45	51	119	15.54 sec.	0.31 sec.	31	36
84	18.71 sec.	0.33 sec.	45	51	120	14.34 sec.	0.30 sec.	27	36
85	18.79 sec.	0.33 sec.	42	50	121	14.12 sec.	0.30 sec.	32	35
86	16.49 sec.	0.25 sec.	40	50	122	14.46 sec.	0.28 sec.	32	35
87	18.98 sec.	0.34 sec.	36	49	123	12.92 sec.	0.27 sec.	31	35
88	20.36 sec.	0.36 sec.	42	48	124	13.04 sec.	0.26 sec.	30	35
89	19.59 sec.	0.37 sec.	36	48	125	11.37 sec.	0.23 sec.	32	34
90	19.18 sec.	0.35 sec.	41	47	126	11.12 sec.	0.22 sec.	32	34
91	17.43 sec.	0.34 sec.	36	47	127	11.35 sec.	0.24 sec.	32	34
92	19.31 sec.	0.37 sec.	40	46	**128**	**6.96 sec.**	**0.14 sec.**	**32**	**32**
93	19.86 sec.	0.40 sec.	33	46	129	7.58 sec.	0.16 sec.	16	32
94	19.86 sec.	0.39 sec.	40	45	130	10.83 sec.	0.22 sec.	16	32
95	19.20 sec.	0.37 sec.	36	45	131	12.08 sec.	0.23 sec.	24	32

TABELLE 8. Berechnungszeiten des Laplace-Problems mit 274 625 Unbekannten auf 64-128 Prozessoren im Detail

7.6. Zusammenfassung und Ausblick

Die parallele Block-LR-Zerlegung für Finite Elemente Diskretisierungen hat sich als ein effizienter, hochskalierbarer Löser herausgestellt. Insbesondere ist er für die unterschiedlichsten Problemklassen – angefangen von der Poisson-Gleichung über Stokes, bis hin zu den Maxwell-Gleichungen – geeignet. Hervorzuheben ist die beachtliche Zerlegungszeit des zweidimensionalen Stokes-Problems mit über 20 Millionen Unbekannten, welche auf 4096 Prozessoren nur 24 Sekunden benötigt hat. Auch das dreidimensionale Maxwell'sche Eigenwertproblem mit quadratischen Elementen konnte in bemerkenswerten 63 Sekunden gelöst werden. Dieses Problem hatte immerhin eine Größe von 3.7 Millionen Unbekannten.

Wesentlich ist dabei eine gute und gleichmäßige Verteilung des Gebietes auf die Prozessoren, so dass sich insgesamt relativ kleine Interfacegrößen ergeben. In den meisten Testfällen wurde der Graphpartitionierer `Kaffpa` eingesetzt, wobei eine recht grobe Schranke für die Verteilung angegeben wurde, so dass diese nicht immer optimal war. Dies kann insbesondere in Tabelle 8 beobachtet werden. Dort wurde der Einheitswürfel auf die Prozessoren verteilt, wobei sich für eine Prozessorzahl ungleich einer Zweierpotenz teilweise recht große Diskrepanzen in der Verteilung ergaben. Nichtsdestotrotz war die Verteilung absolut ausreichend, wie sich in allen getesteten Modellproblemen an den jeweiligen Zerlegungszeiten zeigt (siehe Tabellen 1 bis 7). Dass der Algorithmus auch für beliebige Prozessorzahlen durchgeführt werden kann wird in Kapitel 7.5 gezeigt. Tabelle 8 zeigt weiterhin, dass möglichst eine Zweierpotenz an Prozessoren genutzt werden sollte. In diesem Fall sind in jedem Schritt alle verfügbaren Prozessoren an der Zerlegung beteiligt, während in den anderen Fällen manche Prozessoren gelegentlich nicht involviert sind. Aus dem Grund wurden die Rechnungen jeweils für $P = 2^S$ durchgeführt.

Als zweiter Punkt für die Effizienz der Zerlegung ist die Geschwindigkeit der Datenübertragung zu nennen. In der aktuellen Rechnerentwicklung wird sich die Geschwindigkeit der Prozessoren nicht mehr sehr stark steigern[7] – im Gegenteil: Diese stagniert schon seit einigen Jahren – so dass zusätzliche Rechenleistung nur noch durch Multicore-Prozessoren und Cluster-Zusammenschlüssen gewonnen werden kann. Für Probleme, die auf großen Rechenmaschinen mit mehreren Tausend Prozessoren gerechnet werden, ist die Übertragungsgeschwindigkeit ein aktueller Engpass. Somit ergibt sich für jedes Problem eine bestmögliche Prozessorzahl P_{best}, für die die Zerlegung schnellstmöglich durchgeführt werden kann. Für größere Probleme muss mehr Rechenleistung investiert werden, so dass P_{best} umso höher wird, je größer das zu betrachtende Problem ist (vergleiche dazu insbesondere Tabelle 1 und 2).

Weiterhin spielt auch der zur Verfügung stehende Speicher eine große Rolle. Da die Block-LR-Zerlegung mit vollbesetzten Matrizen arbeitet[8], steigt der Speicherbedarf bei größeren Problemen stark an. Beispielsweise wurden für das Stokes-Problem mit 5 Millionen Unbekannten mindestens 256 Prozessoren benötigt (vergleiche Tabelle 2), so dass für kleinere Prozessorzahlen keine Ergebnisse angegeben werden konnten. Hierbei war die optimale Prozessorzahl P_{best} allerdings auch bei etwa 2048 Prozessoren, wie sich an den Zerlegungszeiten verifizieren lässt. Somit ist es sinnvoll, auch diese Anzahl an Prozessoren zu verwenden.

[7] zumindest bis zur Entwicklung der Quantencomputer
[8] Die Ausnahme bildet hier Schritt $s = 0$

Sobald eine optimale Prozessorzahl P_{best} erreicht ist, steigt die Zerlegungszeit selbst für eine Vervielfachung der Prozessoren nicht mehr stark an (dies kann vor allem in Tabelle 1 beobachtet werden). Dies kann dadurch erreicht werden, dass eine Mindestgröße der Blockmatrizen auf den einzelnen Prozessoren angegeben wird. Somit sind in den letzten Schritten nicht mehr alle Prozessoren an der Zerlegung beteiligt, so dass der Kommunikationsaufwand – insbesondere in der Broadcast-Kommunikation – verhältnismäßig gering bleibt.

Damit ist die parallele Block-LR-Zerlegung insbesondere als Grobgitterlöser in einem Mehrgitterverfahren geeignet. Wenn das gesamte Problem auf dem feinsten Level größer wird, müssen häufig mehr Prozessoren benutzt werden, damit sich der Speicherbedarf pro Prozessor in Grenzen hält. Das Grobgitter hat jedoch noch die gleiche Größe, so dass auch bei der Verwendung von deutlich mehr Prozessoren die Zerlegungszeit durch die (zum Teil) "verhinderte" zusätzliche Kommunikation nicht (stark) ansteigt. Selbstverständlich ist die Zerlegung auch ohne vorgeschaltetes Mehrgitterverfahren ein exzellenter Löser, da die gesuchte Lösung nicht iterativ gefunden werden muss. Insbesondere für mehrere rechte Seiten muss die Zerlegung nur einmal durchgeführt werden. Sofern gleichzeitig mehrere rechte Seiten gelöst werden müssen, wird auch nur eine Durchführung der Lösungsroutine benötigt.

Mit der Weiterentwicklung der Rechner – insbesondere über die Geschwindigkeit der Datenübertragung und des größeren Speichers – werden auch weitaus größere Probleme auf noch mehr Prozessoren in einer akzeptablen Zeit berechenbar. Im Vergleich zu iterativen Lösern ist ein direkter Löser robust, somit ist der Einsatz dann sinnvoll, sofern sich die Zerlegungszeit im Rahmen hält. In der Welt der direkten parallelen direkten Löser basieren viele auf shared memory-Technologien. Die in dieser Arbeit entwickelte parallele Block-LR-Zerlegung ist hierbei komplett auf verteilten Speicher ausgelegt. Der Löser ist für mehrere tausend Prozessoren sinnvoll einzusetzen – vor einigen Jahren stand die Entwicklung hochskalierbarer paralleler Algorithmen für unterschiedliche Problemfälle bei wenigen hundert Prozessoren. Somit wird der unbedingte Einsatz von iterativen Lösern erst für noch größere Maschinen mit mehreren Zehntausend Prozessoren notwendig. Neue Rechnergenerationen bieten die Herausforderung, diese Grenze noch weiter nach hinten zu schieben, indem neuartige parallele Algorithmen entwickelt werden. Dabei wird insbesondere auch die Struktur der Rechner Einfluss geben. Beispielsweise könnten Teilprobleme auf Grafikprozessoren (GPU) ausgelagert werden, sofern die Programmierbarkeit dies zulässt.

Eine interessante Fragestellung betrifft die Approximation über die Idee der parallelen Block-LR-Zerlegung. Die ersten Schritte der Zerlegung sind mit einer großen Prozessorzahl sehr schnell erledigt[9]. Wenn das entstehende Schur-Komplement billig und schnell approximiert werden kann, kann dieser Löser als ein iterativer Löser eingesetzt werden, wobei die Konstruktion relativ schnell vonstatten geht[10].

[9]Die meiste Zeit wird – sofern die einzelnen Problemgrößen auf den Prozessoren klein sind – in den "hinteren Schritten" benötigt

[10]Ein erster Ansatz einer approximativen Block-LR-Zerlegung wurde im Verlauf der Entstehung dieser Arbeit gemacht, allerdings aufgrund schlechter Performancewerten wieder verworfen

Anhang

A LAPACK

Im Allgemeinen werden mit `A`,`B`,`C` Matrizen beschrieben. Über die Variable `transX` (mit `X`$\in \{$`A`, `B`, `C`$\}$) wird dabei die Art der Operation im Verfahren beschrieben, wobei

$$\begin{array}{ll} op(X) = X & \texttt{transX} = \text{'N'} \\ op(X) = X^T & \texttt{transX} = \text{'T'} \ . \end{array}$$

In der Variablen `info` werden dabei eventuelle Fehlermeldungen oder die Meldung einer erfolgreichen Bearbeitung der Funktion zurückgegeben.

Weiterhin wird beschrieben, an welcher Stelle in Algorithmus 3.30 die jeweilige Operation benutzt wird. Die symmetrischen Funktionen sind der Vollständigkeit halber aufgeführt. In [MW12] wurde eine symmetrische Variante der parallelen Block-LR-Zerlegung in einer ersten Version angegeben. Diese war jedoch (noch) nicht optimal, so dass wir uns hier auf die generelle Block-LR-Zerlegung konzentriert haben.

Die Lapack-Funktionen sind Hardware-optimiert und verwendet BLAS[1]-Unterroutinen. Insbesondere in der Matrix-Matrix-Multiplikation wird Level 3 BLAS eingesetzt. Die Hauptarbeit der Algorithmus 3.30 liegt in diesen Matrix-Matrix-Multiplikationen (siehe auch Tabelle 9). Somit ist ein systematischer Einsatz dieser Funktionen sinnvoll und aus Performancegründen erforderlich.

A.1. Allgemeine Funktionen

```
► dgemm(const char     transA,
        const char     transB,
        const int      M,
        const int      N,
        const int      K,
        const double   alpha,
        const double[] A,
        const int      ldA,
        const double[] B,
        const int      ldB,
        const double   beta,
              double[] C,
        const int      ldC);
```

[1] Basic Linear Algebra Subprogram

`dgemm` berechnet eine Matrix-Matrix-Operation

$$\texttt{C} := \texttt{alpha} \cdot op(\texttt{A}) \cdot op(\texttt{B}) + \texttt{beta} \cdot \texttt{C}\,.$$

Im Algorithmus wird diese Routine in den Zeilen 18, 19, 22 und 23 verwendet und ist als Hauptkomponente zu sehen. Ein Vergleich der Operationen in Tabelle 8 bzw. 9 in Kapitel 5 zeigt, dass die Hauptarbeit in der Matrix-Matrix-Multiplikation liegt.

```
▶ dgetrf(const int      M,
         const int      N,
               double[] A,
         const int      ldA,
         const int[]    ipiv,
               int      info);
```

`dgetrf` Funktion berechnet eine LR-Zerlegung der Form

$$\texttt{A} = \texttt{P} \cdot \texttt{L} \cdot \texttt{R}$$

wobei `P` eine Permutationsmatrix ist, `L` eine untere normierte Dreiecksmatrix und `R` eine obere Dreiecksmatrix ist. Die Permutation wird dabei in `ipiv` gespeichert und `A` mit den Matrizen `L` und `R` überschrieben.
Diese Routine berechnet die LR-Zerlegung in Zeile 11 und ist Voraussetzung für die Anwendung von `dgetrs`.

```
▶ dgetri(const int      N,
               double[] A,
         const int      ldA,
         const int[]    ipiv,
               double[] work,
         const int      lwork,
               int      info);}
```

`dgetri` berechnet die Inverse einer Matrix `A` mit Hilfe der LR-Zerlegung, die über `dgetrf` berechnet wurde.
In der Endversion der parallelen Block-LR-Zerlegung wurde keine Inversen-Berechnung durchgeführt. Da insbesondere am Anfang viele rechte Seiten zu lösen sind, könnte eine Anwendung hiervon Sinn machen, sofern die Matrix-Matrix-Multiplikation deutlich schneller durchgeführt wird als die Anwendung der LR-Zerlegung auf die rechten Seiten.

```
▶ dgetrs(const char     transA,
         const int      N,
         const int      nrhs,
         const double[] A,
         const int      ldA,
         const int      ipiv,
               double[] B,
         const int      ldB,
               int      info);
```

`dgetrs` löst ein Gleichungssystem der Art

$$op(\texttt{A})\texttt{X} = \texttt{B}\,,$$

wobei die Matrix `A` über `dgetrf` LR-zerlegt ist.
Diese Routine wird in den Zeilen 17 und 21 zur Berechnung von

$$(Z_q^{(s),t}[k:m])^{-1} Z_r^{(s),t}[k:m] \quad \text{bzw.} \quad (Z_q^{(s),t}[k:m])^{-1} R_r^{(s),t}[k:m]$$

benötigt.

```
► dgemv(const char     transA,
        const int      M,
        const int      N,
        const double   alpha,
        const double[] A,
        const int      ldA,
        const double[] X,
        const int      incX,
        const double   beta,
              double[] Y,
        const int      incY);
```

`dgemv` berechnet eine Matrix-Vektor-Operation

$$\texttt{Y} = \texttt{alpha} \cdot op(\texttt{A}) \cdot \texttt{X} + \texttt{beta} \cdot Y\,.$$

Diese Operation wird im eigentlichen Algorithmus nicht benötigt, kann allerdings in den Lösungsalgorithmen der Vorwärts- (Algorithmus 3.37) bzw. Rückwärts-Substitution (Algorithmus 3.38) eingesetzt werden, sofern nur eine rechte Seite gelöst werden soll. Da allgemein jedoch mehrere rechte Seiten zulässig sind, wird dort auch direkt mit `dgemm` gearbeitet.

A.2. Funktionen für symmetrische Matrizen

In [MW12] wurde eine symmetrische Variante einer parallelen Block-LR-Zerlegung angegeben[2]. Dabei wurde die Symmetrie der Matrizen ausgenutzt, wobei sowohl eine Cholesky-Zerlegung für positiv definite Matrizen und eine LDL^T-Zerlegung möglich war. Insbesondere in den Block-Diagonal-Matrizen wurde nur ein Dreiecksanteil abgespeichert, so dass die Routinen darauf abgestimmt werden mussten. Der Vollständigkeit halber sollen hier die damals verwendeten – sowie einige weitere wichtige symmetrische – Lapack-Operationen vorgestellt werden.

```
► dsymm(const char     side,
        const char     uplo,
        const int      M,
        const int      N,
        const double   alpha,
        const double[] A,
        const int      ldA,
        const double[] B,
        const int      ldB,
        const double   beta,
              double[] C,
        const int      ldC);
```

`dsymm` berechnet eine der Matrix-Matrix-Operationen

$$\begin{aligned} \texttt{C} &= \texttt{alpha} \cdot \texttt{A} \cdot \texttt{B} + \texttt{beta} \cdot \texttt{C} && \text{für } \texttt{side} = \text{‘l‘} \\ \texttt{C} &= \texttt{alpha} \cdot \texttt{B} \cdot \texttt{A} + \texttt{beta} \cdot \texttt{C} && \text{für } \texttt{side} = \text{‘r‘}\,. \end{aligned}$$

Dabei ist `A` eine symmetrische Matrix, in der nur der obere (`uplo` = ’u’) bzw. untere (`uplo` = ’l’) Dreiecksanteil abgespeichert ist.
Im Algorithmus selbst wird diese Funktion nicht benötigt.

[2]Diese war allerdings weit entfernt von optimaler Komplexität im Vergleich zum aktuellen Algorithmus

► `dpptrf(const char uplo,`
`const int N,`
`double[] A,`
`int info);`

`dpptrf` berechnet die Cholesky-Faktorisierung einer symmetrisch positiv definiten Matrix A im komprimierten Format.
Diese Routine wird anstelle der LR-Zerlegung für symmetrisch positiv definite Matrizen in Zeile 11 verwendet.

► `dsptrf(const char uplo,`
`const int N,`
`double[] A,`
`int[] ipiv,`
`int info);`

`dsptrf` berechnet eine LDL^T-Faktorisierung einer symmetrischen Matrix A im komprimierten Format.
Diese Routine wird anstelle der LR-Zerlegung in Zeile 11 verwendet.

► `dpotrf(const char uplo,`
`const int N,`
`double[] A,`
`const int ldA,`
`int info);`

`dpotrf` berechnet die Cholesky-Faktorisierung einer symmetrisch positiv definiten Matrix A.
Sofern die Diagonal-Block-Matrix nicht im komprimierten Format gegeben ist, wird diese Routine in Zeile 11 verwendet.

► `dpptrs(const char uplo,`
`const int N,`
`const int nrhs,`
`const double[] A,`
`double[] B,`
`const int ldB,`
`int info);`

`dpptrs` löst ein Gleichungssystem der Art

$$op(\mathtt{A})\mathtt{X} = \mathtt{B}\,,$$

wobei eine Cholesky-Zerlegung der symmetrisch positiv definiten Matrix `A` im komprimierten Format über `dpptrf` gegeben ist.
Diese Routine ersetzt die `dgetrs`-Routine.

► `dsptrs(const char uplo,`
`const int N,`
`const int nrhs,`
`const double[] A,`
`const int[] ipiv,`
`double[] B,`
`const int ldB,`
`int info);`

`dsptrs` löst ein Gleichungssystem der Art

$$op(\mathtt{A})\mathtt{X} = \mathtt{B}\,,$$

wobei eine LDL^T-Zerlegung der Matrix `A` im komprimierten Format über `dpptrf` gegeben ist.
Diese Routine ersetzt die `dgetrs`-Routine.

▶
```
dpotrs(const char     uplo,
       const int      N,
       const int      nrhs,
       const double[] A,
       const int      ldA,
             double[] B,
       const int      ldB,
             int      info);
```
`dpotrs` löst ein Gleichungssystem der Art

$$op(\texttt{A})\texttt{X} = \texttt{B}\,,$$

wobei eine Cholesky-Zerlegung der Matrix `A` über `dpotrf` gegeben ist.
Sofern die Matrix im Standard-Format abgespeichert ist, ersetzt diese Routine die `dgetrs`-Routine.

▶
```
dtrsm(const char     side,
      const char     uplo,
      const char     transA,
      const char     diag,
      const int      M,
      const int      N,
      const double   alpha,
      const double[] A,
      const int      ldA,
            double[] B,
      const int      ldB);
```
`dtrsm` löst ein Gleichungssystem der Art

$$op(\texttt{A})\texttt{X} = \texttt{alpha} \cdot \texttt{B} \text{ bzw. } \texttt{X}op(\texttt{A}) = \texttt{alpha} \cdot \texttt{B}\,,$$

wobei `A` eine obere oder untere (`uplo`) (normierte (`diag` = 'u')) Dreiecksmatrix ist.
Im Algorithmus wird diese Funktion in den Zeilen 17 und 21 verwendet. Die eigentliche Lösungsroutinen `dports` ist hier nicht sinnvoll, da somit die Symmetrie verloren geht und die originalen rechten Seiten sonst zusätzlich abgespeichert werden müssen, um das Schur-Komplement zu erstellen.

▶
```
dsyrk(const char     uplo,
      const char     trans,
      const int      N,
      const int      K,
      const double   alpha,
      const double[] A,
      const int      ldA,
      const double   beta,
            double[] C,
      const int*     ldC);
```

`dsyrk` berechnet die Matrix-Matrix-Operation

$$\texttt{C} := \texttt{alpha} \cdot \texttt{A} \cdot \texttt{A}^T + \texttt{beta} \cdot \texttt{C} \qquad \text{bzw.}$$

$$\texttt{C} := \texttt{alpha} \cdot \texttt{A}^T \cdot \texttt{A} + \texttt{beta} \cdot \texttt{C},$$

wobei `C` eine symmetrische Matrix ist.
Im Algorithmus wird diese Funktion zur Aktualisierung der Diagonalmatrizen im aktuellen Schur-Komplement benötigt.

B MPI-Routinen

B.1. Ausführung von M++ mit MPI

Zur parallelen Durchführung von M++ wird die MPI-Bibliothek angebunden [GLS07, MPI]. Die wichtigsten Funktionen sollen hier vorgestellt werden. Zur Berechnung der Zerlegungen werden weiterhin SuperLU für die Sparse-Matrix im ersten Schritt und LAPACK für alle weiteren Schritte benötigt. In Kapitel A finden sich die genutzten Funktionen von LAPACK .

Das Programm M++ wird über den Befehl

```
mpirun -np <P> M++
```

gestartet, wobei <P> ein Platzhalter für die Anzahl der benötigten Prozesse ist. Wenn mehrere Rechner zur Verfügung stehen, die untereinander mit einem Switch verbunden sind, kann über den Befehl

```
mpirun -np <P> -machinefile <mfile> M++
```

und einer entsprechenden Datei <mfile> bestimmt werden, wie viele Prozesse auf den einzelnen Maschinen gestartet werden sollen.

Nachdem M++ mit dem obigen Befehl gestartet wurde, befinden sich insgesamt <P> Prozesse auf den Prozessoren[1], die im machinefile angegeben wurden. Eine Initialisierung der MPI-Umgebung findet zu Anfang des Programms über MPI_Init statt und ein Kommunikator MPI_COMM_WORLD wird erstellt. In diesem befinden sich alle Prozesse, so dass eine Kommunikation über diesen Kommunikator zwischen beliebigen Prozessen stattfinden kann. Dieser Kommunikator wird also insbesondere im letzten Schritt S benötigt. Mit Hilfe dieses Kommunikators werden rekursiv weitere Kommunikatoren mit Hilfe von MPI_Comm_split erstellt, bis in Schritt $s = 0$ nur noch jeweils ein Prozess in einem Kommunikator zu finden ist. Die Erstellung der Kommunikatoren ergibt sich also in der entgegengesetzten Reihenfolge zur Erstellung der Schritte des parallelen direkten Lösers. Eine Aktion über die kleineren Kommunikatoren, insbesondere eine Broadcast-Operation wirkt sich damit nur noch auf die beinhaltenden Prozesse aus.

Im Folgenden werden die wichtigsten im Programm M++ benutzten Funktionen und Befehle der MPI-Bibliothek aufgelistet und kurz beschrieben [GLS07, ALO02]. Es existieren noch eine Vielzahl weiterer Funktionen. Eine Online-Dokumentation findet sich auf [MPI].

[1] In diesem Kapitel muss zwischen Prozessen und Prozessoren unterschieden werden, da auch mehrere Prozesse auf einem Prozessor stattfinden können

B.2. Datentypen, Datenobjekte, Variablen und Operationen

Im Folgenden werden die im Programm benutzten elementaren Datentypen von MPI aufgelistet.

- `MPI_INT` für den Datentyp `signed int`.
- `MPI_DOUBLE` für den Datentyp `double`.
- `MPI_CHAR` für den Datentyp `char`.
- `MPI_BYTE` für einen Datentyp, der im Speicher genau ein Byte belegt.

Die meisten Kommunikationen geschehen auf `Byte`-Ebene, das heißt, die Objektgrößen werden auf diese Einheit zum Versenden zurückgerechnet.

Weitere MPI-spezifische Datenobjekte sind unter anderem

- `MPI_Datatype` beschreibt einen der oben genannten elementaren Datentypen.
- `MPI_Group` definiert eine Gruppe.
- `MPI_Comm` definiert einen Kommunikationsverbund (Kommunikator).
- `MPI_Request` wird insbesondere für die Sende-/Empfangsroutinen benötigt und kann zum Beispiel dafür verwendet werden um zu testen, ob das Versenden bereits beendet ist.
- `MPI_Status` wird für die Empfangsroutinen benötigt und zeigt den aktuellen Status an.
- `MPI_Op` beschreibt eine Operation.

Wichtige vordefinierte Konstanten und Variablen sind

- `MPI_SUCCESS` ist der Rückgabewert bei einer erfolgreichen Durchführung eines MPI-Kommandos.
- `MPI_Error` beschreibt als Integer-Variable einen Fehlercode.
- `MPI_COMM_WORLD` ist der Hauptkommunikator, in dem sich alle Prozesse befinden.

Der Kommunikator `MPI_COMM_WORLD` wird im letzten Schritt $s = S$ benötigt. Alle anderen Schritte davor sind in mehrere Cluster aufgeteilt, wobei jedem Cluster eine Teilmenge dieses Kommunikators als Prozessmenge zugewiesen ist. Weiterhin kann in unserem Fall eine Gruppe mit einem Kommunikator gleichgesetzt werden, da für jeden Cluster ein eigener Kommunikator erstellt wird und das Programm in jedem Schritt innerhalb des Kommunikators Daten versendet und empfängt.

Zusätzlich existieren vordefinierte Operationen, die für die Reduktionsoperationen `MPI_Reduce` bzw. `MPI_Allreduce` (siehe B.5) benötigt werden:

- `MPI_MAX` für die globale Maximumbildung.
- `MPI_MIN` für die globale Minimumbildung.
- `MPI_SUM` für die globale Summation.

B.3. Initialisierung und allgemeine Funktionen

Jede Funktion gibt standardmäßig einen Fehlercode als Integer zurück[2], mit dem der Erfolg einer Ausführung angegeben wird.

- ```
  MPI_Init(int*   argc,
           char** argv[ ]);
  ```
  Initialisierung der MPI-Umgebung. `argc` und `argv` bezeichnen die Parameteranzahl bzw. die Parameter selbst, mit denen das C++-Programm gestartet wurde.

[2]Ausnahme: Die Funktionen `MPI_WTime` und `MPI_Wtick` zur Zeitmessung geben ein `double` zurück
  ```

Diese Funktion wird ganz am Anfang im Programm aufgerufen, wobei insbesondere der globale Kommunikator `MPI_COMM_WORLD` erstellt wird, über den weitere Kommunikatoren erstellt werden.

► `MPI_Finalize(void);`
Beenden der MPI-Umgebung.
Die Beendigung wird am Ende des Programms durchgeführt.

►
```
MPI_Comm_size(MPI_Comm comm,     // input
              int*     size);    // output
```
In `size` wird die Anzahl der beteiligten Prozesse im Kommunikationsverbund `comm` zurückgegeben. Für den Kommunikator `MPI_COMM_WORLD` entspricht dies die Anzahl der Prozesse P und im Fall von $P = 2^S$ gilt für einen Kommunikator in Schritt s, dass `size`$= 2^s$.

►
```
MPI_Comm_rank(MPI_Comm comm,     // input
              int*     rank);    // output
```
In `rank` wird die eigene Prozessnummer im Kommunikationsverbund `comm` zurückgegeben.
Im globalen Kommunikator `MPI_COMM_WORLD` gibt diese Funktion die eigene Prozessnummer p aus, der in den Algorithmen verwendet wird.

►
```
MPI_Group_size(MPI_Group group,     // input
               int*      size);     // output
```
In `size` wird die Anzahl der beteiligten Prozesse der Gruppe `group` zurückgegeben.

►
```
MPI_Group_rank(MPI_Group group,     // input
               int*      rank);     // output
```
In `rank` wird die eigene Prozessnummer innerhalb der Gruppe `group` zurückgegeben.

►
```
MPI_Comm_split(MPI_Comm  comm,      // input
               int       color,     // input
               int       key        // input
               MPI_Comm* newcomm);  // output
```
Ausgehend vom Kommunikationsverband `comm` werden neue Kommunikationsverbände `newcomm` gebildet. Ein neuer Kommunikationsverband erhält diejenigen Prozesse mit dem gleichen Wert in `color`. Die Reihenfolge der Prozesse innerhalb der neuen Kommunikationsverbände wird über den Wert `key` gesteuert.
Im Programm selbst wird ausgehend von `MPI_COMM_WORLD` und angefangen von Schritt $s = S$ jeweils zwei Teilkommunikatoren für Schritt $s-1$ gebildet, indem jeweils die Hälfte der Prozesse einen gemeinsamen Wert in `color` zugewiesen bekommt.

► `MPI_Comm_free(MPI_Comm* comm);       // input`
Gibt den Kommunikationsverbund `comm` frei. Sobald alle Prozesse innerhalb des Kommunikationsverbund diesen freigegeben haben, wird dieser gelöscht.

►
```
MPI_Comm_group(MPI_Comm   comm,      // input
               MPI_Group* group);    // output
```
Gibt die zum Kommunikationsverbund `comm` zugehörige Gruppe `group` zurück.

► `double MPI_Wtime();`
Diese Funktion dient der Zeitmessung. Zurückgegeben wird ein Wert, der angibt, wie viel Zeit in Sekunden seit einem (willkürlich festgelegten) Zeitpunkt vergangen ist. Im Gegensatz zu den anderen Funktionen wird hier ein `double`-Wert zurückgegeben. Über

```
double starttime = MPI_Wtime();
  ... Programmteil, dessen Zeit gemessen werden soll
double used_time = WPI_Wtime()-starttime;
```

kann bestimmt werden, wie viel Zeit ein bestimmter Programmabschnitt benötigt.
Diese Funktion wurde insbesondere für Kapitel 5 benötigt, um die Zeitmessungen für die Kommunikation durchzuführen.

► `double MPI_Wtick();`
Gibt die Genauigkeit der Zeitmessung in Sekunden zurück. Wenn zum Beispiel die Systemuhr hardwaremäßig so implementiert ist, dass ein Zähler jede Millisekunde erhöht wird, gibt `MPI_Wtick` den Wert 0.001 zurück. Auch hier wird im Gegensatz zu den anderen Funktionen ein `double`-Wert zurückgegeben.

B.4. Nichtblockierende Kommunikation

Bei einer nichtblockierenden Kommunikation wird der Programmablauf weiter durchgeführt. Dabei wird für eine Sende- beziehungsweise Empfangsoperation ein `request`-Objekt erstellt, welches aktiv bleibt, bis die Empfangsoperation beendet ist. Über `MPI_Wait` kann der Programmablauf gestoppt werden und es wird auf das Beenden der Datenübertragung gewartet.

Dient

►
```
MPI_Isend(void*        buf,        // input
          int          count,      // input
          MPI_Datatype datatype,   // input
          int          dest,       // input
          int          tag,        // input
          MPI_Comm     comm,       // input
          MPI_Request* request);   // output
```

Startet eine nichtblockierende Sendeoperation. Es werden insgesamt `count` Datenmengen des Typs `datatype`, welche im Speicherplatz ab `buf` stehen an den Prozess `dest` innerhalb des Kommunikationsverbunds `comm` gesendet. Zur Unterscheidung verschiedener Nachrichten wird zusätzlich ein `tag` angegeben. Als Rückgabewert wird ein `request` erwartet, über den mit Hilfe der Funktion `MPI_Wait` festgestellt werden kann, ob das Versenden beendet ist.
Diese Funktion wird zum Zusammenfügen zweier Schur-Komplemente benötigt (Zeilen 5 bis 7 in Algorithmus 3.30). Jeder Prozess p muss dort jedes seiner Daten genau einem anderen Prozess q senden.

►
```
MPI_Irecv(void*        buf,        // output
          int          count,      // input
          MPI_Datatype datatype,   // input
          int          source,     // input
          int          tag,        // input
          MPI_Comm     comm,       // input
          MPI_Request* request);   // output
```

Startet eine nichtblockierende Empfangsoperation. Es werden insgesamt `count` Datenmengen des Typs `datatype` von Prozess `source` innerhalb des Kommunikationsverbunds `comm` empfangen und in `buf` geschrieben. Zur Unterscheidung verschiedener Nachrichten wird zusätzlich ein `tag` angegeben. Ein weiterer Rückgabewert ist das `request`, über den mit Hilfe der Funktion `MPI_Wait` festgestellt werden kann, ob das Versenden beendet ist.
Diese Funktion wird vom jeweils anderen Prozess q (siehe `MPI_Isend`) beim Zusammenfügen der Schur-Komplemente aufgerufen.

►
```
MPI_Wait(MPI_Request* request,     // input
         MPI_Status*  status);     // output
```
Blockiert den zugehörigen Prozess so lange, bis die Sende- bzw. Empfangsoperation, welche sich über den `request` definiert, beendet ist. In `status` wird der Status der Datenübertragung angegeben.

B.5. Kollektive Kommunikation

Eine kollektive Kommunikation zeichnet sich dadurch aus, dass gleichzeitig alle Prozesse innerhalb des Kommunikationsverbunds `comm` beteiligt sind. Das heißt insbesondere, dass alle Prozesse in `comm` die jeweilige Funktion aufrufen müssen.

►
```
MPI_Barrier(MPI_Comm comm);
```
Der Funktionsaufruf dient einer Synchronisation. Erst wenn alle Prozesse innerhalb des Kommunikationsverbunds `comm` die Funktion `MPI_Barrier` aufgerufen haben, geht die Programmausführung auf allen Prozessen weiter.
Diese Funktion kann insbesondere auch für eine genauere Zeiterfassung der einzelnen Schritte verwendet werden.

►
```
MPI_Bcast(void*        buf,        // input/output
          int          count,      // input
          MPI_Datatype datatype,   // input
          int          root,       // input
          MPI_Comm     comm);      // input
```
Die Broadcast-Operation versendet eine Nachricht der Länge `count` mit Datentyp `datatype` von `root` an alle anderen Prozesse innerhalb des Kommunikationsverbunds `comm`. Die zu sendenden Daten beziehungsweise die empfangenen Daten befinden sich im Speicher von `buf`.
`MPI_Bcast` ist die Hauptroutine für die Kommunikation in der parallelen Block-LR-Zerlegung. Diese wird in den Zeilen 12 bis 15 verwendet.

►
```
MPI_Reduce(void*        sendbuf,     // input
           void*        recvbuf,     // output
           int          count,       // input
           MPI_Datatype datatype,    // input
           MPI_Op       op,          // input
           int          root,        // input
           MPI_Comm     comm);       // input
```
Die Funktion führt eine globale Operation `op` auf den Datentyp `datatype` innerhalb des Kommunikationsverbunds `comm` aus. Dabei werden insgesamt `count` (gleiche) Operationen komponentenweise auf der Datenmenge `sendbuf` durchgeführt. Das Ergebnis erhält der Prozess `root` in der Datenmenge `recvbuf`.

Die folgenden Funktionen sind in der Endversion der parallelen Block-LR-Zerlegung nicht mehr enthalten, wurden allerdings für einige Vorversionen verwendet und sollen der Vollständigkeit halber noch dargestellt werden.

► `MPI_Allreduce(void* sendbuf, // input`
`void* recvbuf, // output`
`int count, // input`
`MPI_Datatype datatype, // input`
`MPI_Op op, // input`
`MPI_Comm comm); // input`

Die Funktion führt eine globale Operation wie in `MPI_Reduce` aus, wobei alle Prozesse das Ergebnis in der Datenmenge `recvbuf` erhalten.

► `MPI_Gather(void* sendbuf, // input`
`int sendcount, // input`
`MPI_Datatype sendtype, // input`
`void* recvbuf, // output`
`int recvcount, // input`
`MPI_Datatype recvtype, // input`
`int root, // input`
`MPI_Comm comm); // input`

Die Funktion sammelt nacheinander – in der Reihenfolge der Ränge der Prozesse – `sendcount` Daten von jedem Prozess innerhalb des Kommunikationsverbunds `comm` ein. Diese stehen jeweils im Speicher von `sendbuf` mit Datentyp `sendtype`. Der Prozess mit dem Rang `root` erhält alle Daten im Speicher von `recvbuf` vom Datentyp `recvtype`, jeweils mit der Länge von `recvcount`.

► `MPI_Allgather(void* sendbuf, // input`
`int sendcount, // input`
`MPI_Datatype sendtype, // input`
`void* recvbuf, // output`
`int recvcount, // input`
`MPI_Datatype recvtype, // input`
`MPI_Comm comm); // input`

Die Funktion sammelt Daten wie in `MPI_Gather` ein und verteilt das Resultat an alle Prozesse des Kommunikationsverbunds `comm`.

► `MPI_Gatherv(void* sendbuf, // input`
`int sendcount, // input`
`MPI_Datatype sendtype, // input`
`void* recvbuf, // output`
`int* recvcounts, // input`
`int* displs, // input`
`MPI_Datatype recvtype, // input`
`int root, // input`
`MPI_Comm comm); // input`

Ähnlich wie `MPI_Gather` sammelt die Funktion in der Reihenfolge der Ränge der Prozesse Daten von jedem Prozess innerhalb des Kommunikationsverbunds `comm` ein. Der Unterschied besteht dabei, dass jeder Prozess unterschiedlich viele Daten verschicken kann. Der Prozess `root` empfängt dabei vom Prozess `i` insgesamt `recvcounts[i]` Daten, welche in `recvbuf` mit Shift `displs[i]` geschrieben werden.

►
```
MPI_Allgatherv(void*        sendbuf,    // input
               int          sendcount,  // input
               MPI_Datatype sendtype,   // input
               void*        recvbuf,    // output
               int*         recvcounts, // input
               int*         displs,     // input
               MPI_Datatype recvtype,   // input
               MPI_Comm     comm);      // input
```

Wie in `MPI_Gatherv` werden Daten in der Reihenfolge der Ränge der Prozesse eingesammelt, wobei jeder Prozess eine unterschiedliche Anzahl von Daten verschicken kann. Das Resultat erhalten hier alle Prozesse des Kommunikationsverbunds `comm`.

►
```
MPI_Scatter(void*        sendbuf,       // input
            int          sendcount,     // input
            MPI_Datatype sendtype,      // input
            void*        recvbuf,       // output
            int          recvcount,     // input
            MPI_Datatype recvtype,      // input
            int          root,          // input
            MPI_Comm     comm);         // input
```

Die zu versendenden Daten befinden sich im Speicher von `sendbuf` auf dem Prozess `root`. Dieser Prozess sendet jedem Prozess im Kommunikationsverbund `comm` jeweils `sendcount` Daten vom Typ `sendtype`, wobei Prozess `0` die ersten `sendcount` Daten in `sendbuf` erhält, Prozess `1` die nächsten `sendcount` Daten und so weiter. Die jeweiligen Prozesse erhalten die Daten in `recvbuf`, wobei hier wiederum die Anzahl `recvcount` und der Typ `recvtype` angegeben werden muss.

►
```
MPI_Scatterv(void*        sendbuf,      // input
             int*         sendcnts,     // input
             int*         displs,       // input
             MPI_Datatype sendtype,     // input
             void*        recvbuf,      // output
             int          recvcnt,      // input
             MPI_Datatype recvtype,     // input
             int          root,         // input
             MPI_Comm     comm);        // input
```

Wie bei `MPI_Scatter` werden von einem Prozess `root` Daten zu allen anderen Prozessen versendet. Hierbei können die Anzahl der zu versendeten Daten unterschiedlich sein: Prozess `i` erhält vom Prozess `root` insgesamt `sendcnts[i]` Daten, die sich im Speicher `sendbuf` mit Shift `displs[i]` befinden.

C Hardware-Spezifikation

C.1. Parallelcluster Stuttgart (Hermit)

Zur Effizienzbestimmung der parallelen Block-LR-Zerlegung wurden die Rechnungen auf dem Parallelcluster `HERMIT` in Stuttgart durchgeführt[1].

Der große Vorteil dieses Clusters liegt insbesondere an der hohen Prozessorzahl, so dass auch Rechnungen auf bis zu 4096 Kernen durchgeführt werden konnten. Die Wartezeit war zudem sehr gering, so dass jede Rechnung innerhalb eines Tages durchgeführt werden konnte[2].
Auf

https://wickie.hlrs.de/platforms/index.php/...
CRAY_XE6_Hardware_and_Architecture

findet sich folgende Hardware-Spezifikation zu diesem Parallelcluster:

- 3552 compute nodes / 113.664 cores
  - dual socket G34
    - 2x AMD Opteron(tm) 6276 (Interlagos) processors with 16 Cores @ 2.3 GHz (with TurboCore up to 3.3 GHz)
    - 32MB L2+L3 Cache, 16MB L3 Cache
    - HyperTransport HT3, 6.4GT/s=102.4 GB/s
    - Peak performance: 2.3*4*16=147.2 GFLOP/s per socket, 294.4 GFLOP/s per node, about 1 PFLOP/s (1045708.8 GFLOP/s) for the whole system
  - standard of 32GB RAM; 480 nodes equipped with 64GB memory (total of 126TB)
- 96 service nodes (Network nodes, mom nodes, router nodes, DVS nodes, boot, database, syslog)
- High Speed Network CRAY Gemini
- users HOME filesystem:
  - 60TB (BLUEARC mercury 55)
- workspace filesystem:
  - Lustre parallel filesystem
  - capacity 2.7 PB realized with 16 DDN SFA10K controllers
  - IO bandwith 150GB/s

[1] Eine Ausnahme bildet die dreidimensionale Wellengleichung, welche aufgrund Speicherproblemen auf der `HERMIT` auf dem Parallelcluster HC3 in Karlsruhe durchgeführt wurde

[2] Auf der HC3 konnten maximal 1024 Kerne genutzt werden und die Wartezeit betrug teilweise bis zu einer Woche

- special user nodes:
  - external login servers
  - pre-post processing and visualization nodes 128GB memory one node comes with 1TB memory local disks Powerful Graphic Cards with GP-GPU post processing support direct access to parallel filesytem
- infrastructure servers

Literaturverzeichnis

[ADLK01] P.R. Amestoy, I.S. Duff, J.Y. L'Excellent, and J. Koster, *A fully asynchronous multifrontal solver using distributed dynamic scheduling*, SIAM Journal on Matrix Analysis and Applications **23** (2001), no. 1, 15–41.

[ALO02] G. Alefeld, I. Lenhardt, and H. Obermaier, *Parallele Numerische Verfahren*, Springer-Lehrbuch Masterclass, Springer Verlag, 2002.

[BBJ+97] P. Bastian, K. Birken, K. Johannsen, S. Lang, N. Neuß, H. Rentz-Reichert, and C. Wieners, *UG – A flexible software toolbox for solving partial differential equations*, Computing and Visualization in Science **1** (1997), 27–40, 10.1007/s007910050003.

[BBJ+98] P. Bastian, K. Birken, K. Johannsen, S. Lang, V. Reichenberger, H. Rentz-Reichert, C. Wieners, and G. Wittum, *A parallel software-platform for solving problems of partial differential equations using unstructured grids and adaptive multigrid methods*, High Performance Computing in Science and Engineering (1998), 326–339.

[BF91] Franco Brezzi and Michel Fortin, *Mixed and hybrid finite element methods*, Springer series in computational mathematics ; 15, Springer, New York, 1991.

[BGMS97] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith, *Efficient Management of Parallelism in Object Oriented Numerical Software Libraries*, Modern Software Tools in Scientific Computing (E. Arge, A. M. Bruaset, and H. P. Langtangen, eds.), Birkhäuser Press, 1997, pp. 163–202.

[BJL+99] P. Bastian, K. Johannsen, S. Lang, S. Nägele, V. Reichenberger, C. Wieners, G. Wittum, and C. Wrobel, *Advances in High-Performance Computing: Multigrid Methods for Partial Differential Equations and its Applications*, High Performance Computing in Science and Engineering (1999), 506–519.

[BJL+00] P. Bastian, K. Johannsen, S. Lang, V. Reichenberger, C. Wieners, G. Wittum, and C. Wrobel, *Parallel solutions of partial differential equations with adaptive multigrid methods on unstructured grids*, High Performance Computing in Science and Engineering (2000), 496–508.

[BJL+01] P. Bastian, K. Johannsen, S. Lang, V. Reichenberger, C. Wieners, and G. Wittum, *High-accuracy simulation of density driven flow in porous media*, High Performance Computing in Science and Engineering (2001), 500–511.

[Bra93] J.H. Bramble, *Multigrid Methods*, Pitman Research Notes in Mathematics Series, Taylor & Francis Group, 1993.

[Bra03] Dietrich Braess, *Finite Elemente*, 3rd ed., Springer, Berlin, 2003.

[Bö09] S. Börm, *Construction of Data-Sparse $\mathcal{H}^2$-Matrices by Hierarchical Compression*, SIAM Journal on Scientific Computing **31** (2009), no. 3, 1820–1839.

[DEG+99] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu, *A Supernodal Approach to Sparse Partial Pivoting*, SIAM J. Matrix Anal. Appl. **20** (1999), 720–755.

[DH02] Peter Deuflhard and Andreas Hohmann, *Numerische Mathematik I - Eine algorithmisch orientierte Einführung*, de Gruyter Lehrbuch, 2002.

[DPE11] Daniele Antonio Di Pietro and Alexandre Ern, *Mathematical aspects of discontinuous Galerkin methods*, vol. 69, Springer, 2011.

[DR08] W. Dahmen and A. Reusken, *Numerik für Ingenieure und Naturwissenschaftler*, Springer, 2008.

[ESW05] Howard Elman, David Silvester, and Andy Wathen, *Finite Elements and Fast Iterative Solvers with applications in incompressible fluid dynamics*, Oxford University Press, 2005.

[Fro90] A. Frommer, *Lösung Linearer Gleichungssysteme auf Parallelrechnern*, Vieweg, 1990.

[Geo73] Alan George, *Nested Dissection of a Regular Finite Element Mesh*, SIAM Journal on Numerical Analysis **10** (1973), 345–633.

[GL81] A. George and J. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice–Hall, Englewood Cliffs, N.J., 1981.

[GL96] G.H. Golub and C.F.V. Loan, *Matrix computations*, Johns Hopkins studies in the mathematical sciences, Johns Hopkins University Press, 1996.

[GLS07] William Gropp, Ewing Lusk, and Anthony Skjellum, *MPI – Eine Einführung*, 2007.

[Hac91] W. Hackbusch, *Iterative Lösung großer schwachbesetzter Gleichungssysteme*, Teubner Studienbücher Mathematik, 1991.

[Hac03] Wolfgang Hackbusch, *Multi-Grid Methods and Applications*, Springer Series in Computational Mathematics, Springer, 2003.

[Hac09] W. Hackbusch, *Hierarchische Matrizen - Algorithmen und Analysis*, Springer, 2009.

[Her] *https://wickie.hlrs.de/platforms/index.php/cray_xe6_hardware_and_architecture.*

[Hig96] Nicholas J. Higham, *Accuracy and stability of numerical algorithms*, Society for Industrial and Applied Mathematics, 1996.

[HW07] Jan S Hesthaven and Tim Warburton, *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*, vol. 54, Springer, 2007.

[IEE08] *IEEE standard for Floating-Point Arithmetic - IEEE Standard 754 - 2008 (Revision of IEEE Standard 754 - 1985)*, 2008.

[Kan04] C. Kanzow, *Numerik linearer Gleichungssysteme: Direkte und iterative Verfahren*, Springer-Lehrbuch, Springer, 2004.

[KK95] G. Karypis and V. Kumar, *Analysis of Multilevel Graph Partitioning*, Proceedings of the 1995 ACM/IEEE conference on Supercomputing (1995).

[KK98] G. Karypis and V. Kumar, *A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering*, Journal of Parallel and Distributed Computing **48** (1998), 71–95.

[Kny01] A. Knyazev, *Toward the Optimal Preconditioned Eigensolver: Locally Optimal Block Preconditioned Conjugate Gradient Method*, SIAM Journal on Scientific Computing **23** (2001), no. 2, 517–541.

[LeV92] Randall J LeVeque, *Numerical methods for conservation laws*, Birkhäuser, 1992.

[LeV02] Randall J LeVeque, *Finite volume methods for hyperbolic problems*, vol. 31, Cambridge university press, 2002.

[M+98] Gordon E Moore et al., *Cramming more components onto integrated circuits*, Proceedings of the IEEE **86** (1998), no. 1, 82–85.

[Mon03] P. Monk, *Finite element methods for Maxwell's equations*, Clarendon Press, Oxford, 2003.

[MPI] *http://www.mpi-forum.org/docs/docs.html.*

[MW11] Daniel Maurer and Christian Wieners, *A parallel block LU decomposition method for distributed finite element matrices*, Parallel Computing **37** (2011), no. 12, 742 – 758, 6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'10).

[MW12] Daniel Maurer and Christian Wieners, *Parallel Multigrid Methods and Coarse Grid LDL^T Solver for Maxwell's Eigenvalue Problem*, Competence in High Performance Computing 2010 (Christian Bischof, Heinz-Gerd Hegering, Wolfgang E. Nagel, and Gabriel Wittum, eds.), Springer Berlin Heidelberg, 2012, pp. 205–213 (English).

[Mü09] Wolfgang Müller, *Numerische Analyse und parallele Simulation von nichtlinearen Cosserat-Modellen*, Ph.D. thesis, Karlsruher Institut für Technologie (KIT), 2009.

[NCMW07] Patrizio Neff, Krzystof Chelminski, Wolfgang Müller, and Christian Wieners, *A numerical solution method for an infinitesimal elasto-plastic cosserat model*, Mathematical Models and Methods in Applied Sciences **17** (2007), no. 08, 1211–1239.

[Nie12] Simon Niedermaier, *Elektrokardiomechanische Modellierung und parallele Berechnung des Herzschlags*, Ph.D. thesis, Karlsruher Institut für Technologie (KIT), 2012.

[OG96] J.M. Ortega and G. Golub, *Scientific Computing: Eine Einführung in das Wissenschaftliche Rechnen und Parallele Numerik*, Vieweg+Teubner Verlag, 1996.

[OSS12] V. Osipov, P. Sanders, and C. Schulz, *Engineering Graph Partitioning Algorithms*, Proceedings of the 11th International Symposium on Experimental Algorithms (SEA' 12), vol. 7276, Springer, 2012, pp. 18–26.

[Pla04] R. Plato, *Numerische Mathematik kompakt: Grundlagenwissen für Studium und Praxis*, Vieweg Studium, Vieweg, 2004.

[PPJ02] Henon P., Ramet P., and Roman J., *Pastix: a high-performance parallel direct solver for sparse symmetric positive definite systems*, Parallel Computing **28** (2002), no. 2, 301–321.

[PS07] Eric Polizzi and Ahmed Sameh, *SPIKE: A parallel environment for solving banded linear systems.*, Comput. Fluids **36** (2007), no. 1, 113–120 (English).

[PSL90] A. Pothen, H.D. Simon, and K.-P. Liou, *Partitioning sparse matrices with eigenvectors of graphs*, SIAM Journal on Matrix Analysis and Applications **11** (1990), 430–452.

[Sau10] Martin Sauter, *Numerical Analysis of Algorithms for Infinitesimal Associated and Non-Associated Elasto-Plasticity*, Ph.D. thesis, Karlsruher Institut für Technologie (KIT), 2010.

[Sch13] Christian Schulz, *Kaffpa - ein graphpartitionierer, in preparation*, Ph.D. thesis, Karlsruher Institut für Technologie (KIT), 2013.

[SG02] O. Schenk and K. Gärtner, *Solving unsymmetric sparse systems of linear equations with PARDISO.*, Sloot, Peter M. A. (ed.) et al., Computational science - ICCS 2002. 2nd international conference, Amsterdam, the Netherlands, April. 21–24, 2002. Proceedings. Part 2. Berlin: Springer. Lect. Notes Comput. Sci. 2330, 355-363 (2002)., 2002.

[SGFS01] O. Schenk, K. Gärtner, W. Fichtner, and A. Stricker, *PARDISO: A high-performance serial and parallel sparse linear solver in semiconductor device simulation.*, FGCS. Future Generation Computer Systems **18** (2001), no. 1, 69–78 (English).

[SS11] P. Sanders and C. Schulz, *Engineering Multilevel Graph Partitioning Algorithms*, Proceedings of the 19th European Symposium on Algorithms, LNCS, vol. 6942, Springer, 2011, pp. 469–480.

[SWH07] Olaf Schenk, Andreas Wächter, and Michael Hagemann, *Matching-based preprocessing algorithms to the solution of saddle-point problems in large-scale nonconvex interior-point optimization*, Computational Optimization and Applications **36** (2007), 321–341, 10.1007/s10589-006-9003-y.

[Wie07] Christian Wieners, *Nonlinear solution methods for infinitesimal perfect plasticity*, Z. Angew. Math. Mech. (ZAMM) **87(8-9)** (2007), 643–660.

[Wie08] Christian Wieners, *SQP Methods for Incremental Plasticity with Kinematic Hardening*, IUTAM Symposium on Theoretical, Computational and Modelling Aspects of Inelastic Media (B.Daya Reddy, ed.), IUTAM BookSeries, vol. 11, Springer Netherlands, 2008, pp. 143–153.

[Wie10] Christian Wieners, *A geometric data structure for parallel finite elements and the application to multigrid methods with block smoothing*, Computing and Visualization on Science **13** (2010), 161–175.

[Wil91] Roy D. Williams, *Performance of dynamic load balancing algorithms for unstructured mesh calculations*, Concurrency: Practice and Experience **3** (1991), no. 5, 457–481.

[Zag06] S. Zaglmayr, *High order finite element methods for electromagnetic field computation*, Ph.D. thesis, Johannes Kepler Universität Linz, 2006.

[Zha05] Fuzhen [Hrsg.] Zhang (ed.), *The Schur Complement and Its Applications*, Numerical Methods and Algorithms ; 4, Springer US, Berlin, 2005, Erscheint: 13. April 2005.

[Zum03] G. Zumbusch, *Parallel Multilevel Methods: Adaptive Mesh Refinement and Loadbalancing*, Advances in Numerical Mathematics, B.G. Teubner, 2003.

Index