Praktikum Parallele Numerik

Fabian Miltenberger, Sébastien Thill, Thore Mehr

Betreuer: Markus Hoffmann, Thomas Becker

Zusammenfassung Im Rahmen dieses Praktikums haben wir viel gelernt.

Projekt 1

In diesem Projekt lag der Schwerpunkt auf dem Kennenlernen der Bibliothek *OpenMP* sowie deren Handhabung. Weiter ging es um den *IntelThreadChecker*, ein Programm zum Analysieren von Programmcode auf potentielle Fehler in der Parallelisierung. Zu guter letzt haben wir uns mit der FEM-Methode beschäftigt, dabei im Speziellen mit dem Gauß-Seidel-Verfahren zum Lösen linearer Gleichungen wie sie bei der Differenzenmethode vorkommen. Zu guter letzt betrachteten wir einige andere Verfahren zum Lösen solcher Probleme und haben das CG-Verfahren implementiert.

Aufgabe 1.1

```
Hello World, this is Thread0
Hello World, this is Thread5
Hello World, this is Thread4
Hello World, this is Thread7
Hello World, this is Thread1
Hello World, this is Thread3
Hello World, this is Thread6
Hello World, this is Thread2
```

Listing 1.1. Beispielhafte Ausgabe des Programms bei Ausführung mit 8 Fäden.

Wie in der Ausgabe 1.1 zu sehen, folgt die Reihenfolge der ausgeführten Fäden keinem bestimmten Muster. Auch die Reihenfolge zwischen verschiedenen Ausführungen ist in der Regel verschieden.

Aufgabe 1.2

Aufgabe 1.3

1.3.a) Im Folgenden einige bekannte Größen der Parallelisierung, wie sie in der Vorlesung Rechnerstrukturen gelehrt wurden.

Vorab sei $T(n)$ die Ausführungszeit auf n Prozessoren, $P(n)$ die Anzahl der auszuführenden Einheitsoperationen und $I(n)$ der Parallelindex.

Der Speedup/die Beschleunigung $S(n)$:

$$S(n) = \frac{T(1)}{T(n)} \tag{1}$$

Die Effizienz $E(n)$:

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{n \cdot T(n)} \tag{2}$$

Der Mehraufwand $R(n)$:

$$R(n) = \frac{P(n)}{P(1)} \tag{3}$$

Die Auslastung $U(n)$:

$$U(n) = \frac{I(n)}{n} = R(n) \cdot E(n) = \frac{P(n)}{n \cdot T(n)} \tag{4}$$

1.3.b)

Race Condition
Wettlaufsituationen Dabei hängt Ergebnis von konkreter Ausführungsreihenfolge ab (daher Wettlauf) Entsteht, wenn verschiedene Fäden auf gleiche Variable zugreifen, und mindestens ein Faden deren Wert manipuliert Korrektheit der Ergebnisse hängt von Ausführungsreihenfolge ab

Dead lock
Zyklus im Allokationsgraphen

1.3.c) Die Vor- und Nachteile verschiedener paralleler Architekturen bezüglich zweier Aspekte sind in Tabelle 1 aufgetragen.

Architektur	Anwenderfreundlichkeit	Energieeffizienz
GPU	Gut	Mittel
CPU	Gut	Gering
FPGA	Gering	Sehr gut
MIC	Gut	Gut

Tabelle 1. Verschiedene Beschleuniger im Vergleich

Aufgabe 1.4

1.4.a) Zu aller erst nehmen wir eine Differenzierung der in der Aufgabenstellung genannten Variablen vor. Unser n bezeichne dasjenige, welches in der Beschreibung des Gauß-Seidel-Verfahrens auftritt. Das in dem gegebenen Problem genannte n benennen wir zu d um. Die restlichen Variablen behalten ihren Namen bei. Sei ein l für das Verfahren vorgegeben, dann ergeben sich einige andere Größen wie folgt:

$$\begin{aligned} d &= 2^l - 1 \\ h &= \frac{1}{2^l} \\ n &= d^2 \end{aligned} \tag{5}$$

Anschaulich sollen die Werte von $u_{x,y}$ auf einem $(d+2)\cdot(d+2)$ großen Gitter berechnet werden. Die Abstände zwischen den Gitterpunkten sei dabei Gitterkonstante h. Durch die Vorgabe $u_{x,y} = 0$, für x, y auf dem Rand, vereinfacht sich die Problemstellung auf ein Gitter der Größe $d \cdot d$, da der Rand implizit als 0 angenommen werden kann. Zum Lösen des Problems geben wir den Gitterpunkten $u_{x,y}$ eine Ordnung, sodass u sich als einfacher Spaltenvektor mit $n = d \cdot d$ Elementen auffassen lässt. Anschaulich sieht diese Ordnung wie folgt aus:

$$u = (u_{1,1} \dots u_{d,1} u_{1,2} \dots u_{d,d})^T \tag{6}$$

Der Definitionsbereich von u (als Funktion aufgefasst) sei $\mathbb{R}^2$ (ergibt sich aus den gegebenen Randbedingungen). Damit befindet sich ein Gitterpunkt $u_{x,y}$ an der Position $(x \cdot h, y \cdot h)$. Damit berechnen wir den Wert von f für alle unsere Gitterpunkte, also $f_{x,y} = f(x \cdot h, y \cdot h)$. Für diese berechneten Werte von f nehmen wir die gleiche Ordnung vor wie für u und können somit f ebenfalls als einen Spaltenvektor der Größe n ansehen.

Die Matrix $A \in \mathbb{R}^{n \times n}$ sei definiert wie vorgegeben und ihre Elemente seien $(a_{i,j})$. Wir werden sehen, dass wir zur Lösung A nicht explizit berechnen und speichern müssen. Nun sind alle Zutaten zur Berechnung von u in $Au = h^2 f$ mittels Gauß-Seidel-Verfahren gegeben.

Die Implementierung geht nun Schritt für Schritt vor wie im Algorithmus vorgegeben. u^k sei u in der k-ten Iterierten von u. Wir beginnen mit $u^0 = 0$, wählen also 0 als Startvektor. Dies geschieht der Einfachheit wegen, es hat sich herausgestellt, dass es hierdurch bei keiner der gestellten Aufgaben zu Problemen kommt.

In jeder Iterierten k soll nun u^{k+1} berechnet werden. dies geschieht nach der gegebenen Berechnungsvorschrift

$$u_j^{k+1} := \frac{1}{a_{j,j}}(h^2 f_j - \sum_{i=1}^{j-1} a_{j,1} u_i^{k+1} - \sum_{i=j+1}^{n} a_{j,i} u_i^k) \tag{7}$$

für $j = 1, \dots, n$. Es fällt auf, dass für jedes u_j nur solche u_i betrachtet werden, die bereits in der gleichen oder vorherigen Iterierten berechnet, und noch nicht

überschrieben wurden. Somit müssen die einzelnen Iterierten u^k nicht explizit gespeichert werden, tatsächlich reicht hierfür ein Vektor u aus. Die Berechnungsvorschrift vereinfacht sich zu

$$u_j = \frac{1}{a_{j,j}}(h^2 f_j - \sum_{i=1, i\neq j}^{n} a_{j,1} u_i) \tag{8}$$

Nach der Definition von A ist klar, dass $a_{j,j} = 4$ gilt. Für $i \neq j$ gilt $a_{i,j} \in \{0, -1\}$. Um die dünne Struktur von A weiter auszunutzen, betrachten wir anschaulich, von welchen Gitterpunkten die Berechnung für einen Gitterpunkt (x, y) konkret abhängt (genau das wird von A beschrieben). Dargestellt wird dies in Abbildung1, wobei A genau für die vier Nachbarknoten den Wert -1 annimmt, sonst 0 (außer für den Knoten selbst). Durch diese Betrachtung zerfällt die Summe der Zuweisung 8 in vier bedingte Additionen, wie in Code 1.2 illustriert.

```
double sum = h * h * f[j];
if (j % d > 0) sum += u[j - 1];       // Linker Nachbarknoten
if (j % d < d - 1) sum += u[j + 1]; // Rechter Nachbarknoten
if (j / d > 0) sum += u[j - d];       // Oberer Nachbarknoten
if (j / d < d - 1) sum += u[j + d]; // Unterer Nachbarknoten
u[j] = sum / 4;
```

Listing 1.2. Ausnutzung der dünnen Struktur von A zur Berechnung von u_j. Die Zeilen in u sind genau d Elemente lang, daher entspricht bspw. $j - d$ dem Zugriff auf den oberen Nachbar von j aus gesehen. Die Bedingungen Prüfen genau darauf, ob es sich bei einem Nachbarknoten um einen Randpunkt handelt. Falls ja, so ist keine weitere Betrachtung nötig, da dessen Wert 0 ist.

Es wurde gezeigt, wie eine einzelne Iterierte k von u sich berechnen lässt. Würde man dies nun immer wiederholen, so würde sich die Lösung u einem Optimum annähern. Nun haben wir aber eine begrenzte Rechenzeit und müssen daher die Berechnung irgendwann abbrechen, sobald uns das Ergebnis *gut genug* ist. In der Regel kennen wir jedoch die analytische Lösung nicht und wissen daher nicht, wie gut unsere bisherige Lösung ist (sie kann ohnehin durch das Verfahren selbst stark davon abweichen, wie wir in Aufgabe Aufgabe 1.5 sehen werden). Um das Problem zu lösen, brechen wir ab, sobald der *Fortschritt* zwischen zwei Iterationen klein genug ist. Die Annahme hierbei ist, dass sich die Lösung auch durch weitere Iterationen nur noch wenig ändert. Konkret betrachten wir die maximale Veränderung eines Eintrags zwischen den Iterierten k und $k+1$ von u. Formal $maxError := \|u^{k+1} - u^k\|_{max}$. Unsere Lösung nehmen wir als gut genug an, sobald $maxError < \epsilon_{Error}$ für eine Schranke ϵ_{Error}. Für unsere Implementierung verwenden wir $\epsilon_{Error} = 1 \times 10^{-6}$, da es einerseits ausreichend klein ist, um in unseren Tests gute Ergebnisse zu liefern, andererseits groß genug, um in absehbarer Zeit berechnet werden zu können.

1.4.b) Eine naive Parallelisierung des in Aufgabe 1.4.a) implementierten Gauß-Seidel-Verfahrens ist nicht möglich, da innerhalb der Berechnung von u^{k+1}

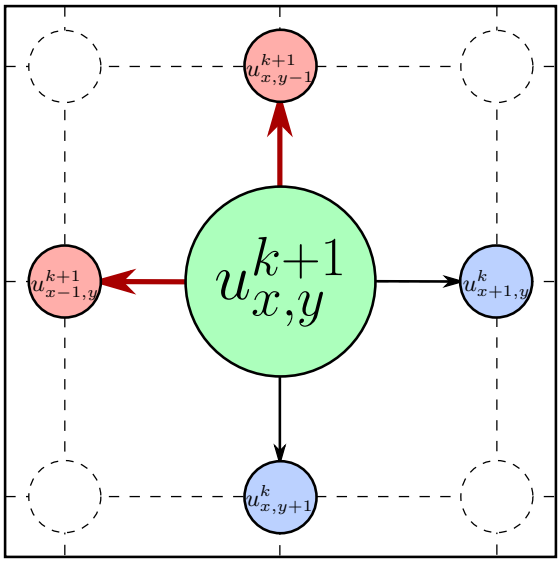

Abbildung 1. Abhängigkeiten zu Nachbarknoten, um Knoten $u^{k+1}_{x,y}$ zu berechnen. Es fällt auf, dass zwei Werte der gleichen Iterierten $k+1$ benötigt werden (die roten Knoten). Durch diese Abhängigkeit können nicht alle Einträge einer Iterierten gleichzeitig, das heißt parallel, berechnet werden.

– eine naive Parallelisierung würde versuchen genau diese Berechnung zu parallelisieren, also eine einzelne Iterierte – Abhängigkeiten zwischen den Einträgen bestehen. Für die Berechnung von $u^{i+k}_{x,y}$ werden diese Abhängigkeiten in Abbildung 1 dargestellt. Um etwa $u^{i+k}_{x,y}$ zu berechnen, werden zu erst die Werte von $u^{i+k}_{x-1,y}$ und $u^{i+k}_{x,y-1}$ aus der gleichen Iteration benötigt. Diese wiederum benötigen in gleicher Weise aktuelle Werte von Nachbarknoten.

Eine Mögliche, nicht mehr ganz so naive Lösung, bestünde darin, nun über die Diagonalen zu parallelisieren. Innerhalb der Diagonalen (in $(1,-1) - \textit{Richtung}$) bestehen keine Abhängigkeiten zwischen den Knoten. Dennoch hat sich auch diese Herangehensweise nicht als Vorteilhaft erwiesen. Für jede Iterierte müssten d Thread-Pools mit maximal $2d-1$ Fäden gestartet und synchronisiert werden. Der Synchronisierungsaufwand hierbei scheint um Größenordnungen größer, als die eigentliche Berechnung (in der Literatur ist dieses Verfahren auch unter dem Namen Wavefront zu finden).

1.4.c) In Aufgabe 1.4.b) haben wir erläutert, dass eine naive Herangehensweise für die Parallelisierung nicht zielführend ist. Insbesondere haben wir hierfür die zahlreichen Abhängigkeiten innerhalb der Berechnung einer Iterierten verantwortlich gemacht. Um also das Gauß-Seidel-Verfahren nun doch effizient zu parallelisieren, ist eine übergeordnete Betrachtung des Problems von Nöten.

Die Idee hinter unserer Lösung besteht darin, in einem parallelisierbaren Schritt Einträge von u für verschiedene Iterierte k der sequentiellen Variante zu berechnen. Wie in Abbildung 1 zu sehen, hängt die Berechnung eines Knotens von den *aktuellen* Werten des linken und des oberen Nachbarn ab. Diese müssen also schon vorher berechnet worden sein. Hieraus ergibt sich eine Reihenfolge, in der die Knoten berechnet werden müssen, nämlich von links oben nach rechts unten (in unserer Lösung zu 1.4.b) haben wir die dabei auftretenden, parallelisierbaren Diagonalen bereits erwähnt). Diese Reihenfolge können wir beibehalten, wenn wir für jede Diagonale eine andere Iterierte berechnen.

Angenommen wir berechnen links oben $u_{1,1}^{k+1}$, dann können wir parallel dazu auch $u_{3,1}^{k}$, $u_{2,2}^{k}$ und $u_{1,3}^{k}$ berechnen (sowie auch alle weiteren Diagonalen mit je 2 Abstand). Im nächsten Schritt können wir dann alle nun noch nicht abgedeckten Diagonalen gleichzeitig berechnen. In Abbildung 2 sind all diese Diagonalen noch einmal aufgetragen. Die Felder, über die gleichzeitig berechnet werden kann, bilden ein Schachbrettmuster.

Der Grund, weshalb immer eine Diagonale ausgelassen werden muss, liegt in den vorhandenen Datenabhängigkeiten (dieses Mal in die andere Richtung, also zu den Nachbarn rechts und unten). Angenommen, man möchte $u_{1,1}^{k+1}$ und $u_{2,1}^{k}$ parallel berechnen, dann würde man bereits für ersteren den Wert von letzterem benötigen (gemäß Abbildung 1). Lässt man jedoch eine Diagonale frei, beispielhaft zwischen $u_{1,1}^{k+1}$ und $u_{3,1}^{k}$, so konnte man $u_{2,1}^{k}$ bereits vorher berechnen. Im nächsten Schritt (das heißt nach Synchronisierung) kann aus den berechneten $u_{1,1}^{k+1}$ und $u_{3,1}^{k}$ das dazwischen liegende $u_{2,1}^{k+1}$ berechnet werden usw.

$u_{1,1}^{k+1}$	$u_{2,1}^{k+1}$	$u_{3,1}^{k}$	$u_{4,1}^{k}$	$u_{5,1}^{k-1}$	$u_{6,1}^{k-1}$	$u_{7,1}^{k-2}$
$u_{1,2}^{k+1}$	$u_{2,2}^{k}$	$u_{3,2}^{k}$	$u_{4,2}^{k-1}$	$u_{5,2}^{k-1}$	$u_{6,2}^{k-2}$	$u_{7,2}^{k-2}$
$u_{1,3}^{k}$	$u_{2,3}^{k}$	$u_{3,3}^{k-1}$	$u_{4,3}^{k-1}$	$u_{5,3}^{k-2}$	$u_{6,3}^{k-2}$	$u_{7,3}^{k-3}$
$u_{1,4}^{k}$	$u_{2,4}^{k-1}$	$u_{3,4}^{k-1}$	$u_{4,4}^{k-2}$	$u_{5,4}^{k-2}$	$u_{6,4}^{k-3}$	$u_{7,4}^{k-3}$
$u_{1,5}^{k-1}$	$u_{2,5}^{k-1}$	$u_{3,5}^{k-2}$	$u_{4,5}^{k-2}$	$u_{5,5}^{k-3}$	$u_{6,5}^{k-3}$	$u_{7,5}^{k-4}$
$u_{1,6}^{k-1}$	$u_{2,6}^{k-2}$	$u_{3,6}^{k-2}$	$u_{4,6}^{k-3}$	$u_{5,6}^{k-3}$	$u_{6,6}^{k-4}$	$u_{7,6}^{k-4}$
$u_{1,7}^{k-2}$	$u_{2,7}^{k-2}$	$u_{3,7}^{k-3}$	$u_{4,7}^{k-3}$	$u_{5,7}^{k-4}$	$u_{6,7}^{k-4}$	$u_{7,7}^{k-5}$

Abbildung 2. Verdeutlichung der Vorgehensweise der Parallelisierung. Zuerst werden diejenigen Einträge von u parallel berechnet, die sich in grau Markierten Feldern befinden. Anschließend parallel die Einträge in den weißen Feldern. Es ist zu beachten, dass die Einträge $u_{x,y}$ für unterschiedliche Iterierte k berechnet werden.

Durch diese Schachbrett-artige Parallelisierung hat ein Berechnungsschritt die Gestalt:

- Berechne Parallel über alle schwarzen Felder
- Berechne Parallel über alle weißen Felder

Es sei nicht zu verschweigen, dass hier im Gegensatz zur seriellen Variante sehr wohl mehrere u gleichzeitig zu speichern sind. Sobald in einem Berechnungsschritt das Feld ganz rechts unten, also $u_{d,d}^{k+2-d}$, berechnet wurde, müssen im Falle des Abbruchs (weil das Abbruchkriterium aus 1.4.a) erfüllt ist), alle $u_{x,y}^{k+2-d}$ noch immer bekannt sein, um u^{k+2-d} als Ergebnis ausgeben zu können. Ebenso muss das Abbruchkriterium stets auf den Werten der gleichen Iterierten arbeiten, um exakt das gleiche Resultat wie die serielle Variante zu erhalten.

In jedem Berechnungsschritt wird genaue eine Iterierte des Verfahrens berechnet. Allerdings gilt dies erst, sobald $d-1$ Schritte ausgeführt wurden, denn erst dann ist schließlich der Eintrag rechts unten $u_{d,d}^{k+2-d} = u_{d,d}^{1}$ berechnet (die *Historie* muss erst gefüllt werden). So kommt es, dass die parallele Version bei gleichen Eingaben exakt die gleichen Ergebnisse wie die serielle Version ausgibt, jedoch dabei genau $d-1$ Iterationen mehr benötigt. Wie der Tabelle 2 zu entnehmen, ist die parallele Version mit 32 Kernen bei großer Problemgröße ($l = 9$) mit einem *Speedup* von $7,57$ aber immerhin noch deutlich schneller. Die eher geringe Effizienz schreiben wir dem Synchronisationsaufwand sowie den eher Cache-unfreundlichen Speicherzugriffen zu.

Problemgröße			Seq.	2 Threads		4 Threads		...	32 Threads	
l	d	n	Laufzeit	Laufzeit	Speedup	Laufzeit	Speedup	...	Laufzeit	Speedup
4	15	225	0,005	0,007	0,357	0,008	0,625	...	1,361	0,0389
5	31	961	0,053	0,056	0,473	0,049	1,08	...	4,041	0,178
8	255	65.025	115	81	1,42	45,3	2,54	...	25,33	4,54
9	511	261.121	1340	943	1,42	504	2,67	...	177	7,57

Tabelle 2. Laufzeiten unserer Parallelisierung gegenüber der sequentiellen Version. Getestet wurde auf dem Rechner *i82sn07*. Um Ungenauigkeiten auszugleichen wurden die Zeiten jeweils über 100 (für große Eingaben über 10) Durchläufe gemittelt.

Aufgabe 1.5

1.5.a) Die Bedingungen lauten nach den Folien der FEM-Einführung:
Ω sei ein beschränktes Gebiet
Γ sei hinreichend glatt
$f : \Omega \to \mathbb{R}$ gegebene Funktion, wie es hier der Fall ist.

1.5.b) Gesucht ist f mit

$$u(x,y) = \sin(2M\pi x)\sin(2N\pi y) \tag{9}$$

Dies kann durch einfache Anwendung des *Laplace-Operators* Δ berechnet werden:

$$\begin{aligned} f(x,y) &= -\Delta u(x,y) \\ &= -\frac{\partial u}{\partial x^2} - \frac{\partial u}{\partial y^2} \\ &= -\frac{\partial}{\partial x}(2M\pi\cos(2M\pi x)\sin(2N\pi y)) - \frac{\partial}{\partial y}(2N\pi\sin(2M\pi x)\cos(2N\pi y)) \\ &= 4M^2\pi^2\sin(2M\pi x)\sin(2N\pi y) + 4N^2\pi^2\sin(2M\pi x)\sin(2N\pi y) \\ &= (M^2+N^2)4\pi^2\sin(2M\pi x)\sin(2N\pi y) \end{aligned} \tag{10}$$

1.5.c) Da in Gleichung 10 $M, N \in \mathbb{N}$ beliebig, wählen wir der Einfachheit halber $M = N = 1$ zur Lösung dieser Teilaufgabe. Damit ergibt sich

$$f(x,y) = 8\pi^2\sin(2\pi x)\sin(2\pi y) \tag{11}$$

Die dazugehörige analytische Lösung ist dann nach Aufgabenstellung gegeben als

$$u(x,y) = \sin(2\pi x)\sin(2\pi y) \tag{12}$$

und wird verwendet um den maximalen Fehler der Näherung zu berechnen.

Das f auf Gleichung 11 kann einfach in den Code der vorangegangenen Aufgabe eingesetzt werden. Es ergeben sich die in Abbildung 3 gezeigten Näherungen. Mit zunehmendem l bzw. abnehmender Gitterkonstante h werden die Lösungen nicht nur feiner, sondern der maximale Fehler zur analytischen Lösung von u auch kleiner wird. Ab $l = 8$ wird der Fehler allerdings wieder größer, wie Tabelle 3 entnommen werden kann. An dieser Stelle scheint das für das Abbruchkriterium gewählte ϵ_{Error} aus Aufgabe 1.4.a) zu grob gewählt zu sein, denn eine weitere Verfeinerung dessen reduzierte den Fehler der Näherung (nur) für große l deutlich (was die Frage aufwirft, ob man ϵ_{Error} abhängig von l wählen sollte).

l	2	3	4	5	6	7	8	9
d	3	7	15	31	63	127	255	511
h	0,25	0,125	0,0625	0,0313	0,0156	0,00781	0,00391	0,00195
Maximaler Fehler	0,234	0,053	0,0130	0,00326	0,000866	0,000326	0,00174	0.0266
Iterationen	21	64	175	409	1148	3515	11043	98558

Tabelle 3. Maximaler Fehler und Anzahl der durchgeführten Operationen in Abhängigkeit von l bzw. Gitterkonstante h.

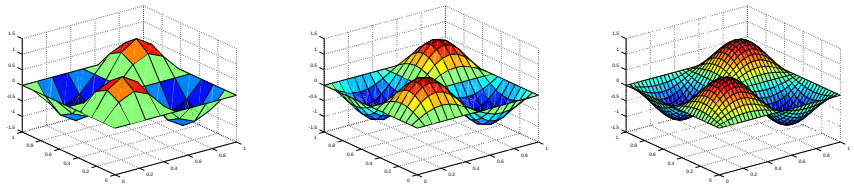

Abbildung 3. Näherungsweise Lösung für u, berechnet mittels Gauß-Seidel-Verfahren. Von links nach rechts: $l = 2, l = 3, l = 4$. Geplottet mittels GNU Octave.

Das verwendete Vorgehen wird als *h-FEM* bezeichnet, da wir lediglich die Gitterkonstante h anpassen. Eine *p-FEM*- oder gar *hp-FEM*-Methodik würde den Grad des zum Abtasten verwendeten Polynoms erhöhen.

Aufgabe 1.6

1.6.a) Implementierung soll mit Hilfe der *kennengelernten Werkzeuge* analysiert werden.

1.6.b) Zu bestimmen: *Speedup* und *Efficiency*

Wie auch unsere Implementierung des Gauß-Seidel-Verfahrens aus Aufgabe Aufgabe 1.4, nutzen wir hier für unsere Implementierung die dünne Struktur der Matrix A aus. Eine Vorkonditionierung würde diese günstige Struktur beeinträchtigen, wodurch ein sehr viel höherer Rechenaufwand zu erwarten wäre. Gesetzt den Fall unsere Matrix A wäre nicht dünn besetzt, so könnte eine Vorkonditionierung Sinn machen, um die Stabilität des Algorithmus ggf. zu erhöhen.

1.6.c)

Projekt 2