Praktikum Multicore-Programmierung
Abschlussprojekt 1

Gruppe 3: Sarah Lutteropp und Johannes Sailer

6. Februar 2016

Zusammenfassung

Dies ist eine Ausarbeitung für das Abschlussprojekt des Praktikums Multicore-Programmierung im Wintersemester 2015/16. Ziel des Projektes war es, am Beispiel des Jacobi-Verfahrens und des Gauß-Seidel-Verfahrens parallele Lösungsmethoden partieller Differentialgleichungen zu implementieren. Wir haben sowohl das Jacobi-Verfahren als auch das Gauß-Seidel-Verfahren mittels OpenMP parallelisiert. Hierbei haben wir für das Gauß-Seidel-Verfahren sowohl einen Parallelisierungsansatz mittels Wavefront als auch einen Parallelisierungsansatz mittels Rot-Schwarz-Unterteilung gewählt. Unsere mit SIMD-Instruktionen beschleunigte Rot-Schwarz-Implementierung des Gauß-Seidel-Verfahrens zeigte die schnellste Laufzeit bei unseren Messungen, gefolgt von der nicht vektorisierten Rot-Schwarz-Implementierung.

1 Mathematischer Hintergrund

Anhand des Beispiels der Approximation von Stoffkonzentrationen innerhalb eines festgelegten zweidimensionalen durch ein Gitter angenäherten Gebietes ergibt sich mittels der auf dem Aufgabenblatt dargestellten Umformungen, Randbedingungen und Argumentationsschritte das lineare Gleichungssystem $Au = b$, das wir mittels Iterationsverfahren lösen sollen.

Hierbei ist $A = \begin{pmatrix} T & -I & & & \\ -I & T & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & T & -I \\ & & & -I & T \end{pmatrix} \in \mathbb{Z}^{n^2 \times n^2}, \quad T \in \mathbb{Z}^{n \times n}$

mit $T_{i,j} = \begin{cases} 4 & \text{falls } i = j \\ -1 & \text{falls } |i-j| = 1 \\ 0 & \text{sonst} \end{cases}, \quad u = \begin{pmatrix} u_{1,1} \\ \vdots \\ u_{n,n} \end{pmatrix} \in \mathbb{R}^{n^2}$ und $b = h^2 * \begin{pmatrix} f(x_1, y_1) \\ \vdots \\ f(x_n, y_n) \end{pmatrix} \in \mathbb{R}^{n^2}$.

Es ist $h \leq 1$, $\frac{1}{h} \in \mathbb{N}$, $n = \frac{1}{h} - 1$, $x_i = y_i = h * i$ für $i = 1, \ldots, n$ und $f : \mathbb{R} \to \mathbb{R}$ eine Funktion. Bei I handelt es sich um die $n \times n$-Einheitsmatrix.

Für $h = \frac{1}{3}$ ergibt sich beispielsweise das folgende Gleichungssystem:

$$\begin{pmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{pmatrix} * \begin{pmatrix} u_{1,1} \\ u_{1,2} \\ u_{2,1} \\ u_{2,2} \end{pmatrix} = \left(\frac{1}{3}\right)^2 * \begin{pmatrix} f(1/3, 1/3) \\ f(1/3, 2/3) \\ f(2/3, 1/3) \\ f(2/3, 2/3) \end{pmatrix}$$

Man könnte dieses lineare Gleichungssystem natürlich auch mit direkten Verfahren wie dem Gaußschen-Eliminationsverfahren lösen. Dieses ist jedoch nur sehr schlecht parallelisierbar. Außerdem ist das Gaußsche-Eliminationsverfahren sehr anfällig für numerische Störungen. Das ist

bei iterativen Verfahren normalerweise nicht der Fall. Hinzu kommt, dass bei großen linearen Gleichungssystemen mit sehr vielen Unbekannten das Gaußsche Eliminationsverfahren viel zu lange dauert.

Unsere Interpretation des Lösungsvektors u In unserer Bearbeitung der Aufgabenstellung haben wir die Lösungsvektor u als Lösungsmatrix U uminterpretiert. Hierbei haben wir ausgenutzt, dass in der Aufgabenstellung die Randbedingungen $u_{i,j} = 0$ für $i \in \{0, n+1\}$ oder $j \in \{0, n+1\}$ gelten. Die Lösungsmatrix U ergibt sich so beispielsweise für $h = \frac{1}{3}$ als:

$$U = \begin{array}{|c|c|c|c|} \hline u_{0,0} & u_{0,1} & u_{0,2} & u_{0,3} \\ \hline u_{1,0} & u_{1,1} & u_{1,2} & u_{1,3} \\ \hline u_{2,0} & u_{2,1} & u_{2,2} & u_{2,3} \\ \hline u_{3,0} & u_{3,1} & u_{3,2} & u_{3,3} \\ \hline \end{array}$$

Die Einträge von U, die per Randbedingung gleich 0 sind, sind hierbei ausgegraut. Die Matrix U hat die Größe $size \times size = (n+2) \times (n+2)$, was dasselbe ist wie $(\frac{1}{h}+1) \times (\frac{1}{h}+1)$. Das $+2$ kommt daher, dass wir die Einträge am Rand hinzunehmen.

1.1 Jacobi-Verfahren

Das Jacobi-Verfahren zerlegt die Matrix A in eine Diagonalmatrix D, eine strikte untere Dreiecksmatrix L und eine strikte obere Dreiecksmatrix R, sodass $A = D + L + R$ gilt. Hierbei enthält D alle Elemente von A auf der Diagonalen, L alle Elemente von A unterhalb der Diagonalen und R alle Elemente von A oberhalb der Diagonalen.

Für die Bestimmung der Iterationsvorschrift wird A in die Teile D und $L + R$ zerteilt.

Das Jacobi-Verfahren ist ein Gesamtschrittverfahren, da alle Einträge von u in einer Iteration aus der vorhergehenden Iteration berechnet werden.

1.1.1 Herleitung

Es gilt

$$\begin{aligned} & Au = b \\ \Leftrightarrow\ & (D + L + R)u = b \\ \Leftrightarrow\ & (L + R)u = b - Du \\ \Leftrightarrow\ & D^{-1}(L + R)u = D^{-1}b - u \\ \Leftrightarrow\ & u + D^{-1}(L + R)u = D^{-1}b \\ \Leftrightarrow\ & u = D^{-1}b - D^{-1}(L + R)u \\ \Leftrightarrow\ & u = -D^{-1}(L + R)u + D^{-1}b \end{aligned}$$

Dies führt zu folgendem iterativen Verfahren, wobei $u^{(k)}$ den Vektor u in Iteration k meint:

$$\begin{aligned} & u^{(k)} = -D^{-1}(L + R)u^{(k-1)} + D^{-1}b \\ \Leftrightarrow\ & u^{(k)} = D^{-1}\left(b - (L + R)u^{(k-1)}\right) \end{aligned}$$

Oder elementweise:

$$u_i^{(k)} = \frac{1}{a_{i,i}} * (b_i - \sum_{j \neq i} a_{ij} u_j^{(k-1)}) \quad \forall i = 1, \ldots, n^2$$

Der Startvektor $u^{(0)}$ kann hierbei beliebig gewählt werden und es gilt, dass D invertierbar sein muss. Dies ist bei unserer Aufgabenstellung der Fall, da $a_{i,i} = 4$ ist für alle $i = 1, \ldots, n^2$.

1.1.2 Abbruchkriterium

Wir haben unser Abbruchkriterium aus dem zweiten Foliensatz der Vorlesung "Heterogene Parallele Rechensysteme" übernommen. Hierbei wird der mittlere Abstand der Einträge der Lösungsmatrix $U^{(k)}$ der aktuellen Iteration zu der Lösungsmatrix $U^{(k-1)}$ aus der vorherigen Iteration betrachtet.

Wir brechen in Iteration k ab, falls

$$\frac{\sum_{i,j} |u_{i,j}^{(k)} - u_{i,j}^{(k-1)}|}{size * size} \leq \texttt{TOL}$$

gilt. Hierbei ist `TOL` ein kleiner Wert. In unserer Implementierung haben wir `TOL` = $0,000001$ verwendet.

Dieses Abbruchkriterium ist zielführend, da es sich mit wenig Aufwand parallel bestimmen lässt und die mittlere absolute Differenz der Matrixeinträge misst. Ist diese gering, ändern weitere Iterationsschritte das Ergebnis nicht mehr stark. Hierfür ist es wichtig, einen Zufallsvektor als Startvektor zu wählen und nicht z.B. mit dem Nullvektor zu starten, da dieser bei einem Ergebnis mit vielen Nulleinträgen zu einem zu frühen Abbruch führen würde.

Wir hatten auch überlegt, stattdessen die maximale absolute Differenz $\max_{i,j} |u_{i,j}^{(k)} - u_{i,j}^{(k-1)}|$ als Entscheidungsgrundlage zu nehmen. Dies haben wir aber verworfen, da uns die mittlere absolute Differenz bereits gut genug erschien. Außerdem ist es mit OpenMP für C nicht möglich, in der `reduce`-Klausel den Maximumsoperator zu verwenden (bei OpenMP für Fortran geht es). Denkbar wäre es auch gewesen, stattdessen die relative Abweichung zu betrachten.

1.2 Gauß-Seidel-Verfahren

Wie im Jacobi-Verfahren wird auch im Gauß-Seidel-Verfahren die Matrix A in eine Diagonalmatrix D, eine strikte untere Dreiecksmatrix L und eine strikte obere Dreiecksmatrix R zerlegt.

Bei dem Gauß-Seidel-Verfahren handelt es sich um ein Einzelschrittverfahren, da in jeder Iteration die bereits berechneten Einträge von u direkt weiterverwendet werden.

1.2.1 Herleitung

Es gilt

$$\begin{aligned} & Au = b \\ \Leftrightarrow\ & (D + L + R)u = b \\ \Leftrightarrow\ & Du = b - (L + R)u \\ \Leftrightarrow\ & Du = b - Lu - Ru \\ \Leftrightarrow\ & u = D^{-1}(b - Lu - Ru) \end{aligned}$$

Dies führt zu folgendem iterativen Verfahren, wobei $u^{(k)}$ den Vektor u in Iteration k meint:

$$u^{(k)} = D^{-1}\left(b - Lu^{(k)} - Ru^{(k-1)}\right)$$

Oder elementweise:

$$u_i^{(k)} = \frac{1}{a_{i,i}} * \left(b_i - \sum_{j=1}^{i-1} a_{i,j} u_j^{(k)} - \sum_{j=i+1}^{n^2} a_{i,j} * u_i^{(k-1)}\right) \quad \forall i = 1, \ldots, n^2$$

Der Startvektor $u^{(0)}$ kann hierbei beliebig gewählt werden und es gilt, dass D invertierbar sein muss. Dies bei unserer Aufgabenstellung der Fall, da $a_{i,i} = 4$ ist für $i = 1, \ldots, n^2$.

1.2.2 Abbruchkriterium

Für das Gauß-Seidel-Verfahren haben wir dasselbe Abbruchkriterium wie in Abschnitt 1.1.2 verwendet. Dies erachten wir für sinnvoll, da es sich beim Gauß-Seidel-Verfahren bloß um eine Verbesserung des Jacobi-Verfahrens handelt und nicht um ein vollkommen anderes Konzept. Auch hier ist es bei unserem Abbruchkriterium wieder wichtig, mit einem Zufallsvektor für u zu starten.

1.3 Vergleich der Konvergenz und Stabilität beider Verfahren

Beide Verfahren sind uneingeschränkt stabil.

Da unsere Matrix A strikt diagonaldominant ist, konvergieren sowohl das Jacobi- als auch das Gauß-Seidel-Verfahren in unserer Anwendung. Hierbei konvergiert das Gauß-Seidel-Verfahren deutlich schneller als das Jacobi-Verfahren, da das Jacobi-Verfahren im Gegensatz zum Gauß-Seidel-Verfahren mit veralteten Näherungen weiterrechnet.

Die Konvergenzgeschwindigkeit unserer Implementierung schwankt je nach Startvektor. Daher haben wir über zehn Aufrufe gemittelt, wobei in jedem Aufruf beide Verfahren mit demselben Startvektor ausgeführt wurden.

	$h = 1/4$	$h = 1/32$	$h = 1/64$	$h = 1/128$
Jacobi	33	1477	7195	26832
Gauß-Seidel	16	713	2249	6860

Abbildung 1: Über zehn Aufrufe gemittelte Anzahl Iterationen in beiden Verfahren für verschiedene Verfeinerungen h, bis unser Abbruchkriterium `true` wurde.

2 Sequentielle Implementierung

Die Matrixeinträge werden in dem auf dem Aufgabenblatt zur Verfügung gestellten Pseudocode spaltenweise durchlaufen. Daher haben wir in unserer Implementierung die Matrizen spaltenweise indiziert, um eine möglichst gute Cache-Lokalität zu erzielen. Dies bedeutet, dass wir statt U eigentlich U^T speichern.

Anstatt des Parameters h übergeben wir einen Parameter $size$, der $\left(\frac{1}{h} + 1\right)$ entspricht. Dies hat den Vorteil, dass der Nutzer den Grad der Verfeinerung exakt ohne viele Nachkommastellen angeben kann und U eine $size \times size$-Matrix ist. Um von $size$ auf h zurückzurechnen, gilt:

$$size = \left(\frac{1}{h} + 1\right)$$

$$\Leftrightarrow \frac{1}{h} = size - 1$$

$$\Leftrightarrow h = \frac{1}{size - 1}$$

Da sich die Einträge des Vektors b innerhalb der Iterationen nicht ändern, haben wir b vorberechnet.

Den Startvektor haben wir mit zufälligen Gleitkommazahlen gefüllt.

2.1 Laufzeiten bei verschiedenen Verfeinerungen

Wir haben die Laufzeiten unser sequentiellen Implementierungen für $h = \frac{1}{2^l}$ mit $l \in \{6, 7, 8, 9, 10\}$ gemessen, da für kleinere Werte von l die Messungen zu stark variierten. Wir haben die Messungen auf `i82sn02.itec.kit.edu` (Abb. 2) durchgeführt.

	$h = 1/64$	$h = 1/128$	$h = 1/256$	$h = 1/512$	$h = 1/1024$
Jacobi	0.113 s	1.5 s	20.7 s	345 s	4500 s
Gauß-Seidel	0.113 s	1.4 s	14.8 s	115 s	545 s

Abbildung 2: Laufzeiten der sequentiellen Implementierungen des Jacobi- und Gauß-Seidel-Verfahrens für verschiedene Verfeinerungen h auf `i82sn02.itec.kit.edu`

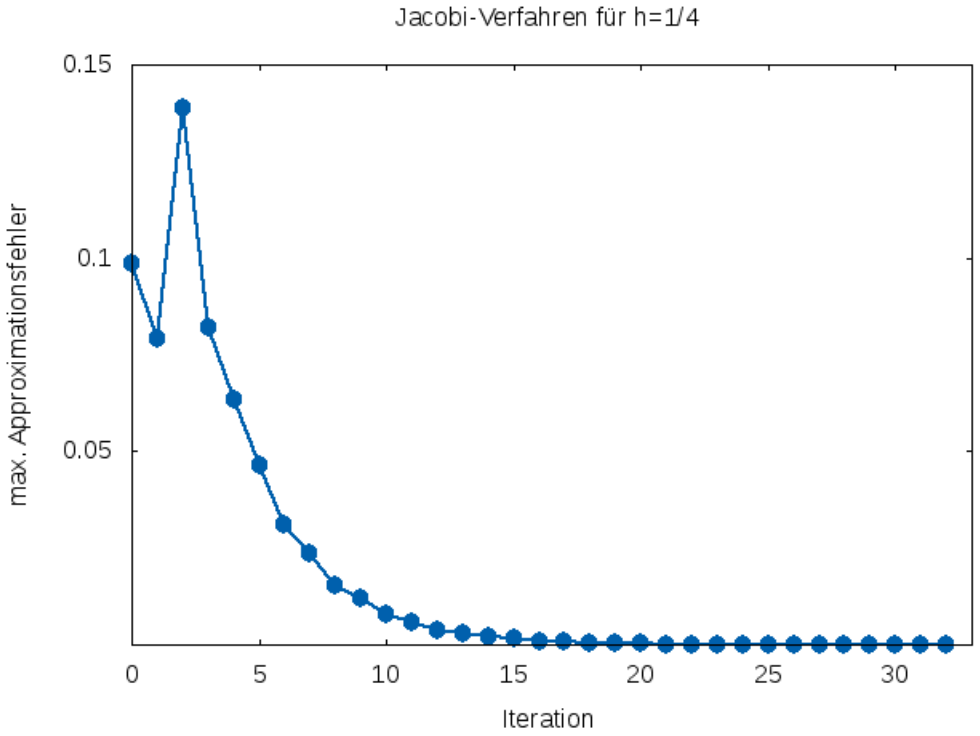
2.2 Approximationsfehler

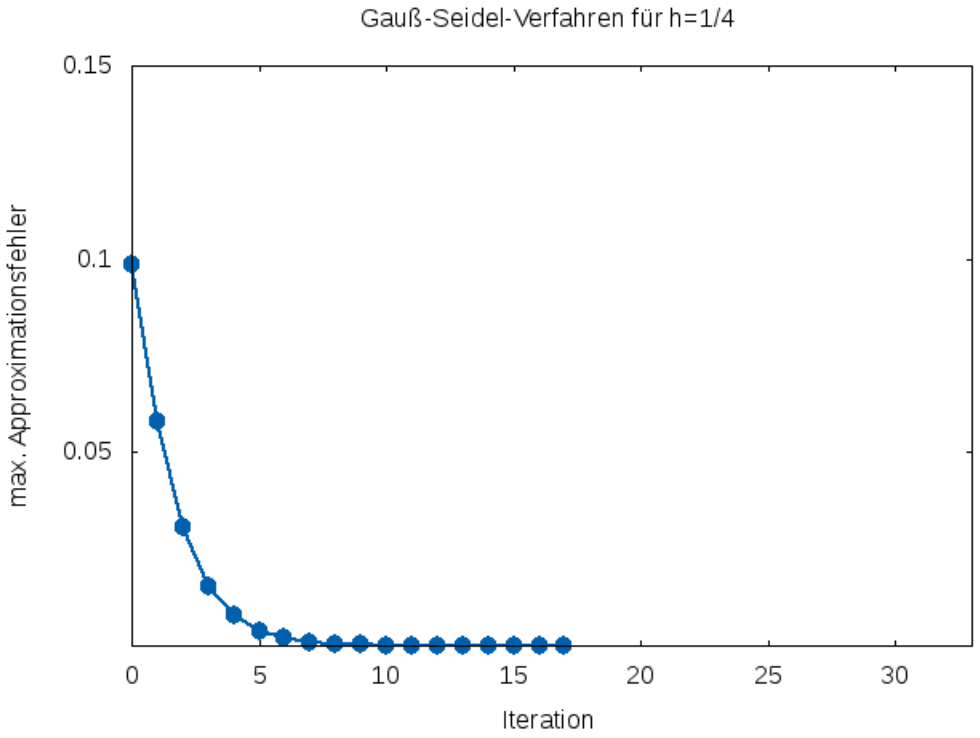
Wir haben den maximalen absoluten Approximationsfehler in der euklidischen Norm in jedem Iterationsschritt für das Jacobi- und das Gauß-Seidel-Verfahren gemessen. Sei $U^{(*)}$ die analytische Lösung für U und $U^{(k)}$ die Lösungsmatrix im k-ten Iterationsschritt. Der maximale absolute Approximationsfehler in euklidischer Norm in Iteration k ist definiert als

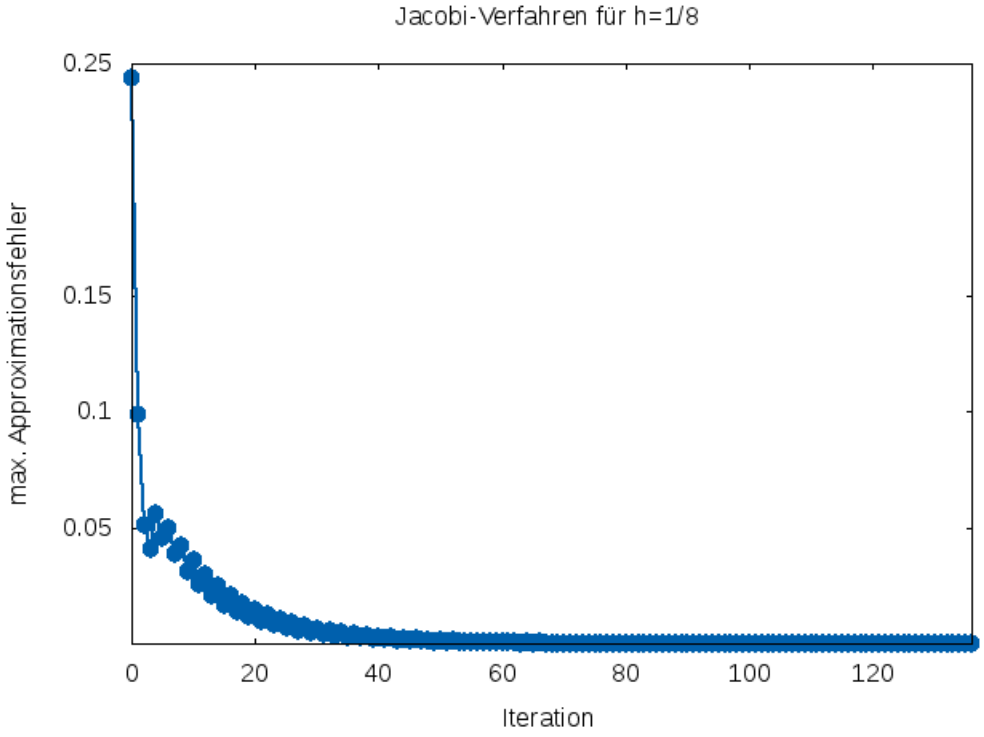
$$\max_{(i,j)\in\{1,\dots,n\}^2} \sqrt{\left(U_{i,j}^{(k)} - U_{i,j}^{(*)}\right)^2}$$

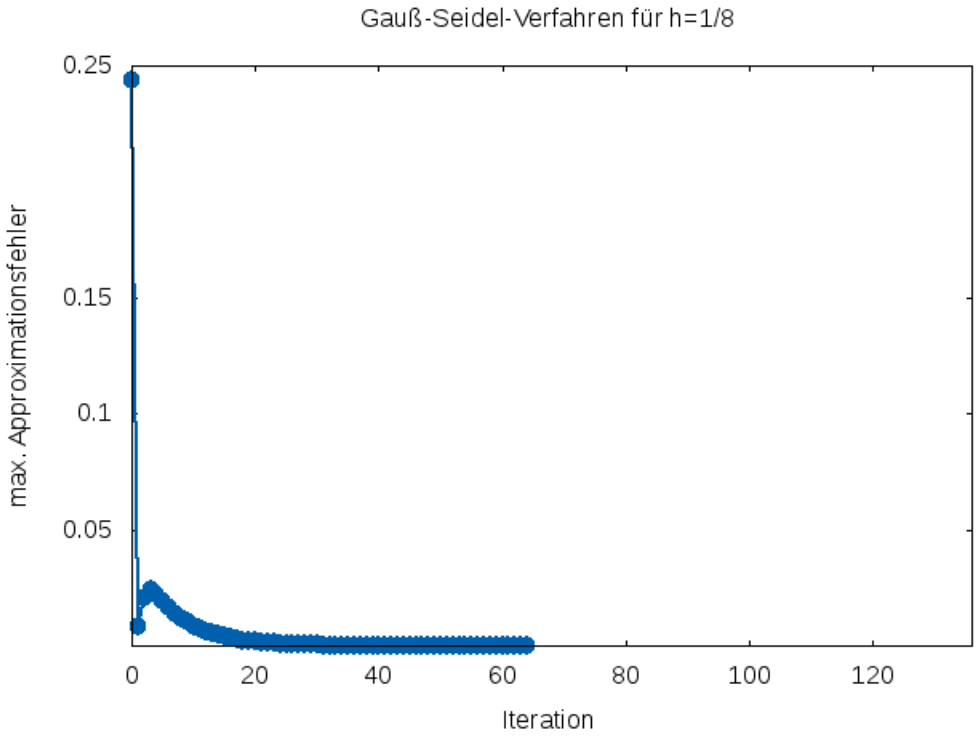
Sei $x_i = i * h$ und $y_j = j * h$ für $i, j = 0, \dots, n+1$. Wir haben die analytische Lösung mittels der Funktion $u(x_i, y_j) = \begin{cases} 16x_i(1-x_i)y_i(1-y_i) & \text{, falls } 0 < x_i < 1 \text{ und } 0 < y_j < 1 \\ 0 & \text{, sonst} \end{cases}$ berechnet.

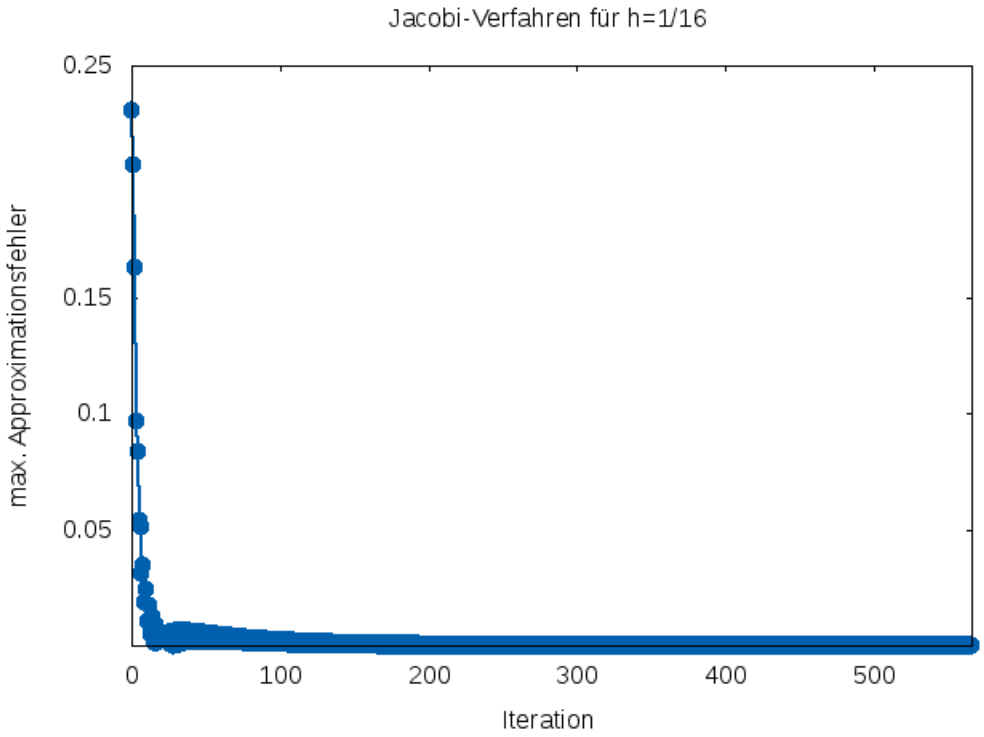
Wir haben die Messungen nur für die Verfeinerungen $h \in \left\{\frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}\right\}$ durchgeführt, da für kleinere h die Grafiken zu groß werden und man nichts mehr erkennt. Die Grafiken haben wir mit `gnuplot` erstellt. Beim Vergleich der Grafiken fällt auf, dass der maximale Approximationsfehler beim Gauß-Seidel-Verfahren schneller sinkt als beim Jacobi-Verfahren. Außerdem kann es in beiden Verfahren zu einer geringfügigen Verschlechterung des maximalen Approximationsfehlers in den ersten Iterationen kommen.

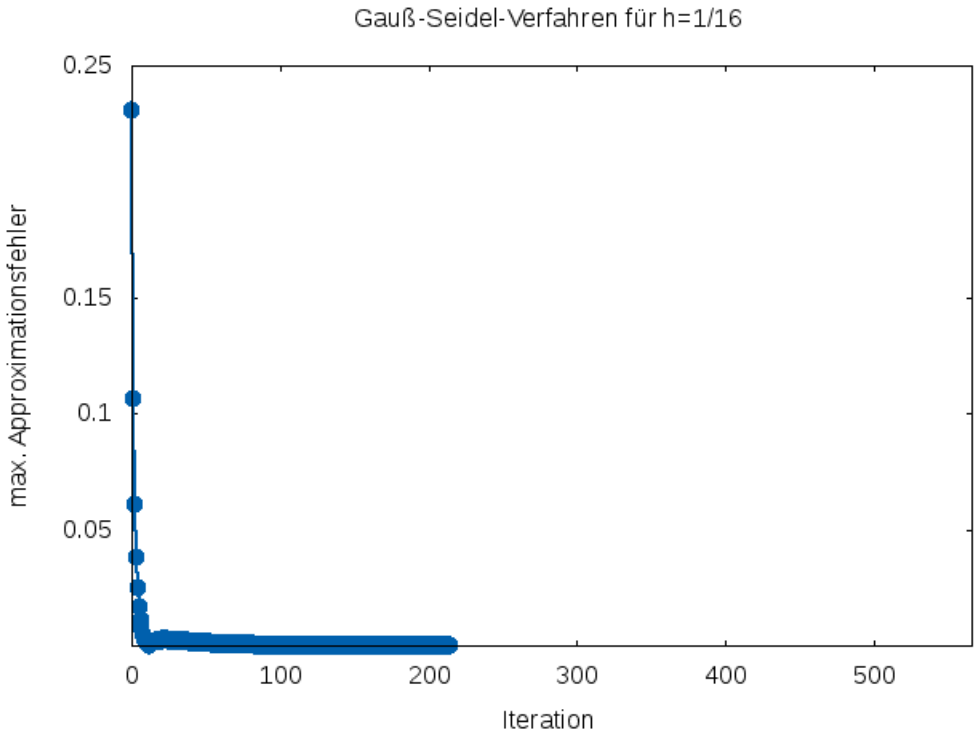


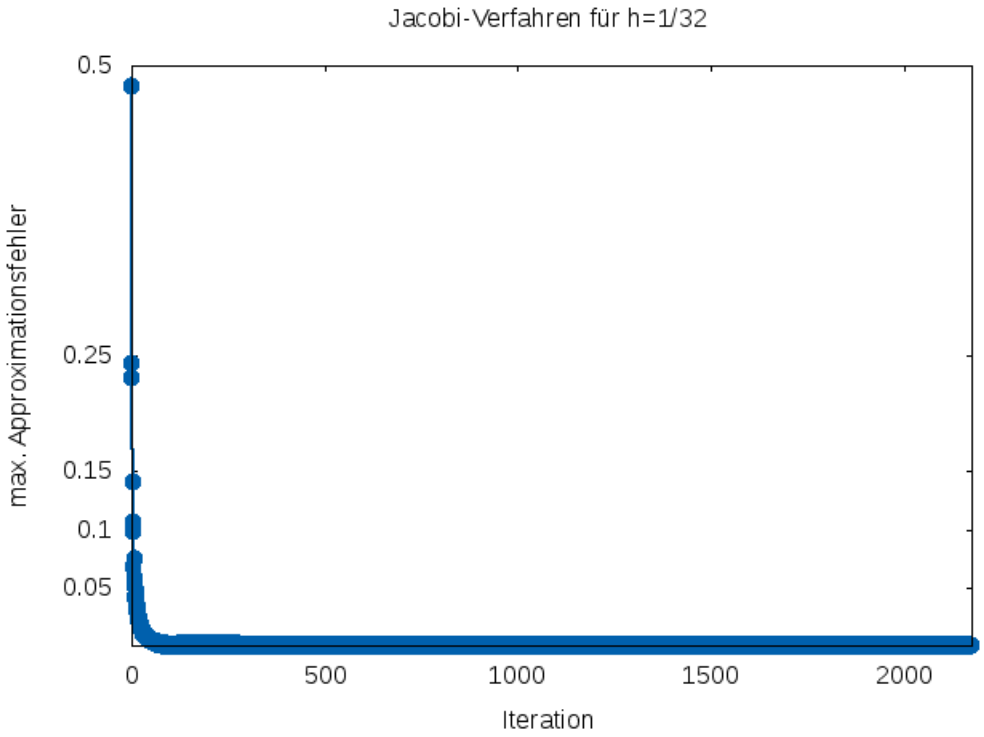

Gauß-Seidel-Verfahren für h=1/4
max. Approximationsfehler
0.15
0.1
0.05
0
5
10
15
20
25
30
Iteration

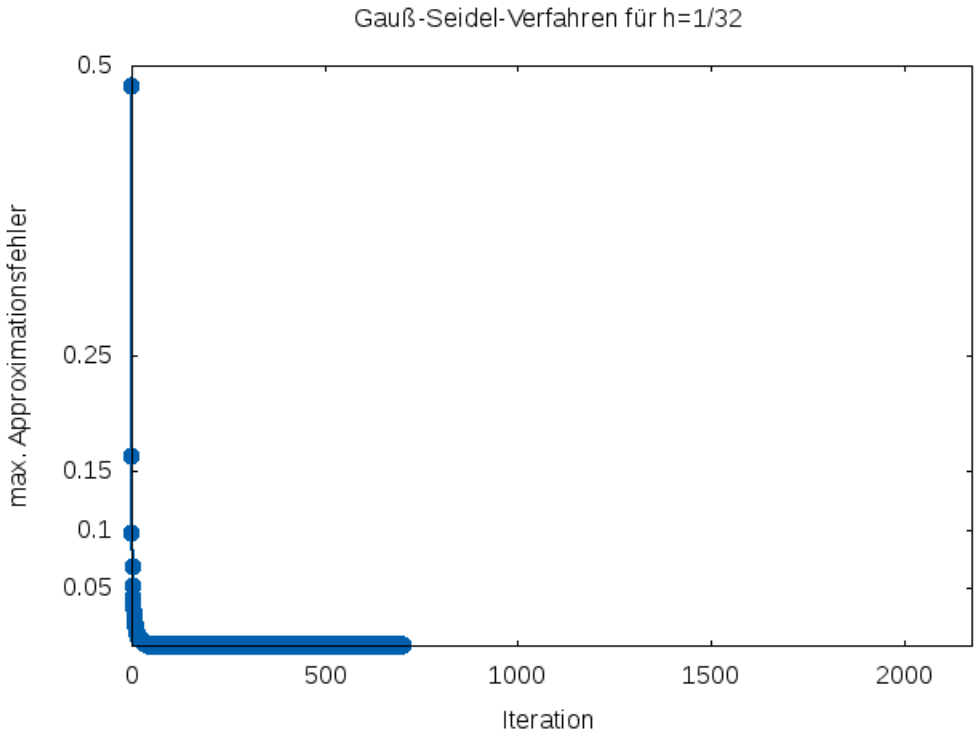

Jacobi-Verfahren für h=1/8
max. Approximationsfehler
0.25
0.15
0.1
0.05
0
20
40
60
80
100
120
Iteration


Gauß-Seidel-Verfahren für h=1/8
max. Approximationsfehler
0.25
0.15
0.1
0.05
0
20
40
60
80
100
120
Iteration


Jacobi-Verfahren für h=1/16
max. Approximationsfehler
0.25
0.15
0.1
0.05
0
100
200
300
400
500
Iteration


Gauß-Seidel-Verfahren für h=1/16
max. Approximationsfehler
0.25
0.15
0.1
0.05
0
100
200
300
400
500
Iteration


Jacobi-Verfahren für h=1/32
max. Approximationsfehler
0.5
0.25
0.15
0.1
0.05
0
500
1000
1500
2000
Iteration



3 Parallelisierung des Jacobi-Verfahrens

Da innerhalb der Iterationsschritte des Jacobi-Verfahrens keinerlei Datenabhängigkeiten bei der Berechnung der Matrixeinträge bestehen, haben wir mittels OpenMP (`#pragma omp parallel for`) die äußere Schleife parallelisiert.

Zudem haben wir eine weitere Version des Jacobi-Verfahrens implementiert, in der wir zusätzlich zur Parallelisierung der `for`-Schleife auch den Code innerhalb der `for`-Schleife mit SSE vektorisiert haben.

Unsere Speedup-Messungen haben wir auf `i82sn02.itec.kit.edu` durchgeführt. Wir haben den Speedup jeweils relativ zu der sequentiellen Version bestimmt.

	1 Thread		4 Threads		8 Threads	
	Laufzeit	Speedup	Laufzeit	Speedup	Laufzeit	Speedup
Sequentiell	0.108 s	-	0.097 s	-	0.093 s	-
OpenMP	0.217 s	0.498	0.073 s	1.329	0.104 s	0.894
OpenMP + SSE	0.117 s	0.923	0.048 s	2.02	0.052 s	1.789

Tabelle 1: Speedup-Messungen für verschiedene Parallelisierungen des Jacobi-Verfahrens für $h = \frac{1}{64}$.

	1 Thread		4 Threads		8 Threads	
	Laufzeit	Speedup	Laufzeit	Speedup	Laufzeit	Speedup
Sequentiell	1.493 s	-	1.562 s	-	1.51 s	-
OpenMP	2.98 s	0.501	0.871 s	1.793	0.854 s	1.768
OpenMP + SSE	1.565 s	0.954	0.49 s	3.188	0.373 s	4.048

Tabelle 2: Speedup-Messungen für verschiedene Parallelisierungen des Jacobi-Verfahrens für $h = \frac{1}{128}$.

	1 Thread		4 Threads		8 Threads	
	Laufzeit	Speedup	Laufzeit	Speedup	Laufzeit	Speedup
Sequentiell	25.765 s	-	24.8 s	-	26.605 s	-
OpenMP	43.427 s	0.593	10.714 s	2.315	6.867 s	3.874
OpenMP + SSE	22.424 s	1.149	5.655 s	4.385	3.593 s	7.405

Tabelle 3: Speedup-Messungen für verschiedene Parallelisierungen des Jacobi-Verfahrens für $h = \frac{1}{256}$.

4 Parallelisierung des Gauß-Seidel-Verfahrens

Innerhalb der Iterationsschritte des Gauß-Seidel-Verfahrens bestehen Datenabhängigkeiten, da die Berechnung der Matrixeinträge in einer Iteration von den vorher berechneten Einträgen der selben Iteration abhängt. Das Gauß-Seidel-Verfahren ist daher inhärent sequentiell. Daher ist eine Parallelisierung des Gauß-Seidel-Verfahrens mit mehr Aufwand verbunden als beim "embarassingly parallel" Jacobi-Verfahren.

Wir haben sowohl den naiven, falschen Parallelisierungsansatz als auch zwei verschiedene funktionierende Parallelisierungsansätze implementiert.

4.1 Naiver Parallelisierungsansatz

Der naive Parallelisierungsansatz ist, wie in Abschnitt 3 die äußerste Schleife mittels `#pragma omp parallel for` zu parallelisieren. Da eine parallele Ausführung der Schleifeniterationen mittels OpenMP nicht die Iterationsreihenfolge garantiert, kann es hierbei zu Wettlaufsituationen ("Race Conditions") kommen.

Der im Praktikum vorgestellte Intel Thread Checker erkennt die auftretende Wettlaufsituation nicht.

4.2 Erweiterter Parallelisierungsansatz: Rot-Schwarz

Bei der Rot-Schwarz-Parallelisierung des Gauß-Seidel-Verfahrens wird ausgenutzt, dass jeder Matrixeintrag nur von seinem linken, rechten, oberen und unteren direkten Nachbarn abhängt. Wir färben also die Matrixeinträge in rote und schwarze Felder ein, wobei die Datenabhängigkeiten nur zwischen Einträgen unterschiedlicher Farbe bestehen. Hierbei ergibt sich ein Schachbrettmuster, wie in Abbildung 3 gezeigt.

Da die Matrixeinträge gleicher Farbe nicht voneinander abhängen, können wir diese parallel berechnen. Dies führt zu dem Ansatz, zuerst alle roten Einträge und danach alle schwarzen Einträge jeweils parallel zu berechnen.

Hierbei ist zu beachten, dass die einzelnen Iterationen des sequentiellen Gauß-Seidel-Verfahrens und dessen Rot-Schwarz-Parallelisierung nicht genau dasselbe Ergebnis liefern. Dies liegt darin begründet, dass durch die Rot-Schwarz-Aufteilung eine Umordnung der Matrixeinträge geschieht.

Um die Indexberechnung zu vereinfachen und die Cachelokalität zu verbessern, haben wir die roten und die schwarzen Einträge jeweils in eigenen, neuen Matrizen gespeichert. Dadurch liegen

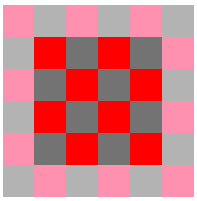
$U =$ 

Abbildung 3: Aufteilung der Lösungsmatrix U in rote und schwarze Zellen, sodass zwischen Zellen gleicher Farbe keine Datenabhängigkeiten bestehen. Die Randzellen, deren Einträge per Annahme immer den Wert 0 enthalten, sind in blasseren Farben markiert.

Einträge gleicher Farbe zusammenhängend im Speicher. Dies hat es uns auch ermöglicht, unsere Implementierung mittels SSE-Vektorinstruktionen zu beschleunigen. Die Vektorisierung mittels SIMD-Instruktionen haben wir explizit implementiert. Zwar ist mit OpenMP 4.0 eine Vektorisierung von Code auch mittels `#pragma omp simd` möglich, aber unser händisch vektorisierter Code war schneller als der von OpenMP generierte.

Bei den Indexberechnungen haben wir zwischen $size \times size$-Matrizen gerader und ungerader $size$ unterschieden.

Berechnung der Nachbarindizes für gerade Werte von $size$

Für rote $U_{i,j}$ gilt: Der zugehörige Index in der Matrix für die roten Einträge ist $idx = \lfloor \frac{i+j*size}{2} \rfloor$. Der linke Nachbar ist an Stelle $idx - \frac{size}{2}$ in der Schwarz-Matrix. Der obere Nachbar ist an Stelle $idx - (1 - (j \mod 2))$ in der Schwarz-Matrix. Der rechte Nachbar ist an Stelle $idx + \frac{size}{2}$ in der Schwarz-Matrix. Der untere Nachbar ist an Stelle $idx + (j \mod 2)$ in der Schwarz-Matrix.

Für schwarze $U_{i,j}$ gilt: Der zugehörige Index in der Matrix für die schwarzen Einträge ist $idx = \lfloor \frac{i+j*size}{2} \rfloor$. Der linke Nachbar ist an Stelle $idx - \frac{size}{2}$ in der Rot-Matrix. Der obere Nachbar ist an Stelle $idx - (j \mod 2)$ in der Rot-Matrix. Der rechte Nachbar ist an Stelle $idx + \frac{size}{2}$ in der Rot-Matrix. Der untere Nachbar ist an Stelle $idx + (1 - (j \mod 2))$ in der Rot-Matrix.

Abbildung 4 zeigt die Berechnung der Nachbarindizes am Beispiel $size = 6$.

$U =$

0	3	6	9	12	15
0	3	6	9	12	15
1	4	7	10	13	16
1	4	7	10	13	16
2	5	8	11	14	17
2	5	8	11	14	17

$\leftarrow: idx - \frac{size}{2}$ $\rightarrow: idx + \frac{size}{2}$ $\uparrow: idx - (1 - (j \mod 2))$ $\downarrow: idx + (j \mod 2)$

$\leftarrow: idx - \frac{size}{2}$ $\rightarrow: idx + \frac{size}{2}$ $\uparrow: idx - (j \mod 2)$ $\downarrow: idx + (1 - (j \mod 2))$

Abbildung 4: Berechnung der Nachbarindizes für gerade Werte von $size$ am Beispiel $size = 6$

Berechnung der Nachbarindizes für ungerade Werte von $size$

Für rote $U_{i,j}$ gilt: Der zugehörige Index in der Matrix für die roten Einträge ist $idx = \lfloor \frac{i+j*size}{2} \rfloor$. Der linke Nachbar ist an Stelle $idx - \lceil \frac{size}{2} \rceil$ in der Schwarz-Matrix. Der obere Nachbar ist an Stelle $idx - 1$ in der Schwarz-Matrix. Der rechte Nachbar ist an Stelle $idx + \lfloor \frac{size}{2} \rfloor$ in der Schwarz-Matrix. Der untere Nachbar ist an Stelle idx in der Schwarz-Matrix.

Für schwarze $U_{i,j}$ gilt: Der zugehörige Index in der Matrix für die schwarzen Einträge ist $idx = \lfloor \frac{i+j*size}{2} \rfloor$. Der linke Nachbar ist an Stelle $idx - \lfloor \frac{size}{2} \rfloor$ in der Rot-Matrix. Der obere

Nachbar ist an Stelle idx in der Rot-Matrix. Der rechte Nachbar ist an Stelle $idx + \lceil \frac{size}{2} \rceil$ in der Rot-Matrix. Der untere Nachbar ist an Stelle $idx + 1$ in der Rot-Matrix.

Abbildung 5 zeigt die Berechnung der Nachbarindizes am Beispiel $size = 7$.

$$U = \begin{array}{|c|c|c|c|c|c|c|} \hline 0 & 3 & 7 & 10 & 14 & 17 & 21 \\ \hline 0 & 4 & 7 & 11 & 14 & 18 & 21 \\ \hline 1 & 4 & 8 & 11 & 15 & 18 & 22 \\ \hline 1 & 5 & 8 & 12 & 15 & 19 & 22 \\ \hline 2 & 5 & 9 & 12 & 16 & 19 & 23 \\ \hline 2 & 6 & 9 & 13 & 16 & 20 & 23 \\ \hline 3 & 6 & 10 & 13 & 17 & 20 & 24 \\ \hline \end{array}$$

$\leftarrow: idx - \lceil \frac{size}{2} \rceil \quad \rightarrow: idx + \lfloor \frac{size}{2} \rfloor \quad \uparrow: idx - 1 \quad \downarrow: idx$

$\leftarrow: idx - \lfloor \frac{size}{2} \rfloor \quad \rightarrow: idx + \lceil \frac{size}{2} \rceil \quad \uparrow: idx \quad \downarrow: idx + 1$

Abbildung 5: Berechnung der Nachbarindizes für ungerade Werte von $size$ am Beispiel $size = 7$

Messung von Speedup und Effizienz unter verschiedenen Verfeinerungen

Unsere Speedup-Messungen haben wir auf `i82sn02.itec.kit.edu` durchgeführt. Wir haben den Speedup jeweils relativ zu der sequentiellen Gauß-Seidel Version (ohne unsere Rot-Schwarz-Optimierungen) bestimmt. Unsere Messergebnisse zeigen, dass bereits unsere Cache-Optimierung und die damit verbundene weniger rechenintensive Bestimmung der Nachbarindizes einen deutlichen Geschwindigkeitsvorteil bringt. Dies erklärt die hohen Speedup-Werte.

	1 Thread			4 Threads			8 Threads		
	Laufzeit	Speedup	Effizienz	Laufzeit	Speedup	Effizienz	Laufzeit	Speedup	Effizienz
Sequentiell	0,114 s	-	-	0,113 s	-	-	0,114 s	-	-
Rot-Schwarz OpenMP	0,062 s	1,839	1,839	0,025 s	4,52	1,13	0,052 s	2,192	0,274
Rot-Schwarz OpenMP + SSE	0,035 s	3,257	3,257	0,02 s	5,65	1,413	0,024 s	4,75	0,594

Tabelle 4: Speedup- und Effizienz-Messungen für verschiedene Parallelisierungen des Gauß-Seidel-Verfahrens mit der Rot-Schwarz-Methode für $h = \frac{1}{64}$.

	1 Thread			4 Threads			8 Threads		
	Laufzeit	Speedup	Effizienz	Laufzeit	Speedup	Effizienz	Laufzeit	Speedup	Effizienz
Sequentiell	1,427 s	-	-	1,388 s	-	-	1,423 s	-	-
Rot-Schwarz OpenMP	0,706 s	2,021	2,021	0,212 s	6,547	1,637	0,214 s	6,65	0,831
Rot-Schwarz OpenMP + SSE	0,423 s	3,374	3,374	0,143 s	9,706	2,427	0,142 s	10,021	1,253

Tabelle 5: Speedup- und Effizienz-Messungen für verschiedene Parallelisierungen des Gauß-Seidel-Verfahrens mit der Rot-Schwarz-Methode für $h = \frac{1}{128}$.

	1 Thread			4 Threads			8 Threads		
	Laufzeit	Speedup	Effizienz	Laufzeit	Speedup	Effizienz	Laufzeit	Speedup	Effizienz
Sequentiell	15,179 s	-	-	14,963 s	-	-	15,036 s	-	-
Rot-Schwarz OpenMP	7,371 s	2,059	2,059	1,973 s	7,584	1,896	1,118 s	13,449	1,681
Rot-Schwarz OpenMP + SSE	4,303 s	3,528	3,528	1,196 s	12,511	3,128	0,782 s	19,228	2,404

Tabelle 6: Speedup- und Effizienz-Messungen für verschiedene Parallelisierungen des Gauß-Seidel-Verfahrens mit der Rot-Schwarz-Methode für $h = \frac{1}{256}$.

Bewertung der Skalierbarkeit

TODO: Bildchen

4.3 Erweiterter Parallelisierungsansatz: Wavefront

Beim Wavefront Algorithmus wird anstatt über die Spalten oder Zeilen zu iterieren über die Antidiagonalen iteriert. Das hat den Vorteil, dass die Daten nicht voneinander abhängig sind und das selbe herauskommt, wie beim eigentlichen Gauß-Seidel Verfahren in serieller Form.

Vermutlicher Grund wieso unsere Wavefront-Implementierung langsamer ist als sequentielles Gauß-Seidel: Die Indexberechnung ist sehr aufwändig. Wir berechnen die Indizes in jeder Iteration neu, anstatt sie einmalig vorzuberechnen und in einer "Indextabelle" zu speichern. Außerdem richtet sich die Parallelität nach der Anzahl der Antidiagonalelemente. D.h. es lohnt sich erst bei großen Antidiagonalen.

Bei der Wavefront Cache Lösung wurden die Matrix zuerst in ein anderes Array kopiert, in dem die Antidiagonalen in den Zeilen stehen. Dadurch haben wir uns erhofft, dass es zu einer höheren Cache Hitrate kommt.
In beiden Wavefront Lösungen hätte man sicher noch einiges optimieren können(z.B. CUDA für große Antidiagonalen), jedoch erschien uns Rot-Schwarz so viel schneller, dass es sich nicht lohnen würde.

Berechnung der Indizes Für die Implementierung wurden drei Schleifen verwendet. Die äußere Schleife ist zwischen den Schritten des Gauß-Seidel Verfahrens, die mittlere geht über die Antidiagonalen und die innere über die Elemente der Antidiagonalen.
In der mittleren Schleife werden die Variablen für die aktuelle Anzahl an Elementen gesteuert sowie eine Variable die angibt in welchem Durchlauf man nach der Antidiagonalen ist(border). Diese dient dazu anzugeben, wie viele Elemente man am Rand weglassen kann, weil sie nicht mehr in der Matrix sind.
Bei der Wavefront Cache Lösung muss man bei jeder Indexberechnung die Eingangspermutation, welche die Antidiagonalen den Zeilen des Arrays zuweist umkehren um die Indizes zu berechnen. Hierbei verwenden wir die Hilfsvariablen `hack` und `hack2`. Diese werden benutzt um die Elemente um das zu berechnende Element zu adressieren. Die Variable `hack` wird 1 Durchgang vor der Antidiagonalen zu 1, weil sich ab hier die Abhängigkeit der Elemente um 1 verschiebt. Das selbe gilt für `hack2` ab der mittleren Antidiagonalen.
Da der Ansatz nicht so viel schneller wurde, dass man Rot-Schwarz hätte schlagen können, wurde der Algorithmus nicht platzeffizient implementiert und die Matrixgröße ist größer als bei den anderen Algorithmen. Es wäre jedoch auch möglich gewesen dieselbe Matrixgröße zu verwenden.

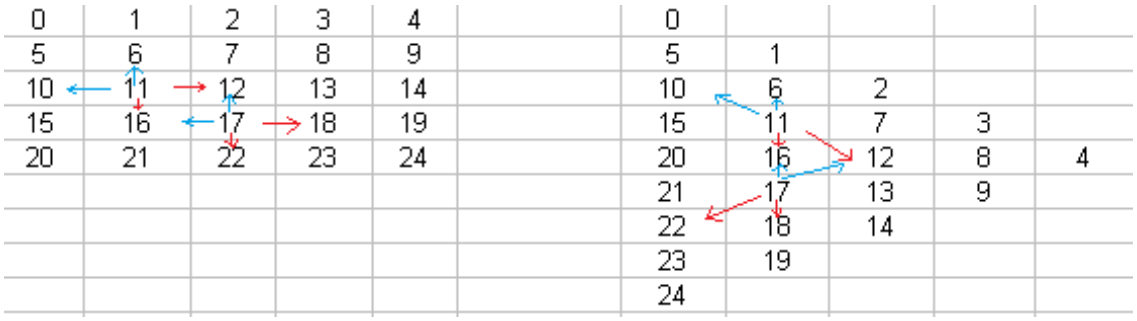


Abbildung 6: Veranschaulichung der Permutation, welche verwendet wird sowie der Verschiebung der Indizes anhand eines Beispiels. Die Variable `hack` ist dabei blau und `hack2` rot.

5 Methodenwahl

Es empfiehlt sich das Gauß-Seidel-Verfahren zu nutzen, weil es im allgemeinen schneller konvergiert als das Jacobi-Verfahren. Bis zu einer Größe der Lösungsmatrix U von $size \times size$ mit $size = 31$ (was $h = \frac{1}{30}$ entspricht) lohnt sich das Parallelisieren nicht. Bei größeren Matrizen (also $h < \frac{1}{30}$) ist die Rot-Schwarz SSE Implementierung des Gauß-Seidel Verfahrens am schnellsten.

TODO: Laufzeitmessungen

6 Anhang: Noch mehr Laufzeitmessungen

In den folgenden Messungen wurde der Speedup immer relativ zu unseren sequentiellen Implementierungen gemessen. Die Messwerte können leicht von den bisher angegebenen abweichen, da sie teils mit anderen Werten für `TOL` ausgeführt wurden und das Abbruchkriterium dadurch erst später erreicht wurde.
Die folgenden Messungen sind vielleicht nicht ganz fair, weil die parallelen Versionen mit SSE beschleunigt wurden und die sequentiellen nicht. Die Werte für $h > \frac{1}{128}$ sind sehr fehleranfällig, weil sie so klein sind, und fast unbrauchbar.

	size = 65								
	Laufzeit in Sekunden								
Sequentiel Jacobi	0,113								
Sequential Gaus-Seidel	0,113								
Thread Zahl	1			4			8		
	Laufzeit in Sekunden	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz
Jacobi SSE	0,121	0,934	0,934	0,050	2,260	0,57	0,071	1,592	0,2
Gaus-Seidel RS SSE	0,034	3,324	3,324	0,020	5,650	1,41	0,032	3,531	0,44

Abbildung 7: Ergebnisse der Messung auf sn02 (8 Kerne)

	Size = 129								
	Laufzeit in Sekunden								
Sequentiel Jacobi	1,500								
Sequential Gaus-Seidel	1,400								
Thread Zahl	1			4			8		
	Laufzeit in Sekunden	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz
Jacobi SSE	1,600	0,938	0,938	0,450	3,333	0,83	0,300	5,000	0,63
Gaus-Seidel RS SSE	0,400	3,500	3,500	0,145	9,655	2,41	0,135	10,370	1,3

Abbildung 8: Ergebnisse der Messung auf sn02 (8 Kerne)

	Size = 513								
	Laufzeit in Sekunden								
Sequentiel Jacobi	345,000								
Sequential Gaus-Seidel	115,000								
Thread Zahl	1			4			8		
	Laufzeit in Sekunden	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz
Jacobi SSE	329,000	1,049	1,049	76,000	4,539	1,13	43,000	8,023	1
Gaus-Seidel RS SSE	34,000	3,382	3,382	8,600	13,372	3,34	5,500	20,909	2,61

Abbildung 9: Ergebnisse der Messung auf sn02 (8 Kerne)

	Size = 1025								
	Laufzeit in Sekunden								
Sequentiel Jacobi	4500,000								
Sequential Gaus-Seidel	545,000								
Thread Zahl	1			4			8		
	Laufzeit in Sekunden	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz
Jacobi SSE	4430,000	1,016	1,016	1051,000	4,282	1,07	546,000	8,242	1,03
Gaus-Seidel RS SSE	201,000	2,711	2,711	40,000	13,625	3,41	20,900	26,077	3,26

Abbildung 10: Ergebnisse der Messung auf sn02 (8 Kerne)

	size = 65								
	Laufzeit in Sekunden								
Sequentiel Jacobi	0,090								
Sequential Gaus-Seidel	0,083								
Thread Zahl	1			2			4		
	Laufzeit in Sekunden	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz
Jacobi SSE	0,189	0,476	0,476	0,101	0,891	0,45	0,060	1,500	0,38
Gaus-Seidel RS SSE	0,056	1,482	1,482	0,035	2,371	1,19	0,027	3,074	0,77

Abbildung 11: Ergebnisse der Messung auf sn03 (4 Kerne)

	Size = 129								
	Laufzeit in Sekunden								
Sequentiel Jacobi	1,150								
Sequential Gaus-Seidel	1,000								
Thread Zahl	1			2			4		
	Laufzeit in Sekunden	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz
Jacobi SSE	2,500	0,460	0,460	1,350	0,852	0,43	0,747	1,539	0,38
Gaus-Seidel RS SSE	0,700	1,429	1,429	0,375	2,667	1,33	0,227	4,405	1,1

Abbildung 12: Ergebnisse der Messung auf sn03 (4 Kerne)

	Size = 513								
	Laufzeit in Sekunden								
Sequentiel Jacobi	974,000								
Sequential Gaus-Seidel	88,000								
Thread Zahl	1			2			4		
	Laufzeit in Sekunden	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz
Jacobi SSE	607,000	1,605	1,605	291,000	3,347	1,67	150,000	6,493	1,62
Gaus-Seidel RS SSE	54,000	1,630	1,630	29,000	3,034	1,52	14,000	6,286	1,57

Abbildung 13: Ergebnisse der Messung auf sn03 (4 Kerne)

	Size = 1025								
	Laufzeit in Sekunden								
Sequentiel Jacobi	13174,000								
Sequential Gaus-Seidel	521,000								
Thread Zahl	1			2			4		
	Laufzeit in Sekunden	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz
Jacobi SSE	7668,000	1,718	1,718	4000,000	3,294	1,65	2023,000	6,512	1,63
Gaus-Seidel RS SSE	270,000	1,930	1,930	141,000	3,695	1,85	67,000	7,776	1,94

Abbildung 14: Ergebnisse der Messung auf sn03 (4 Kerne)

	Size = 129														
	Laufzeit in Sekunden														
Sequentiel Jacobi	1,800														
Sequential Gaus-Seidel	1,700														
Thread Zahl	1			4			8			12			32		
	Laufzeit in Sekunden	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz
Jacobi SSE	1,900	0,947	0,947	0,600	3,000	0,75	0,500	3,600	0,45	0,452	3,982	0,33	2,300	0,783	0,02
Gaus-Seidel RS SSE	0,500	3,400	3,400	0,190	8,947	2,24	0,170	10,000	1,25	0,170	10,000	0,83	0,400	4,250	0,13

Abbildung 15: Ergebnisse der Messung auf sn07 (32 Kerne)

	Size = 513														
	Laufzeit in Sekunden														
Sequentiel Jacobi	428,000														
Sequential Gaus-Seidel	142,000														
Thread Zahl	1			4			8			12			32		
	Laufzeit in Sekunden	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz
Jacobi SSE	364,000	1,176	1,176	102,000	4,196	1,05	50,600	8,458	1,06	36,000	11,889	0,99	43,000	9,953	0,31
Gaus-Seidel RS SSE	41,000	3,463	3,463	10,400	13,654	3,41	5,500	25,818	3,23	4,000	35,500	2,96	3,700	38,378	1,2

Abbildung 16: Ergebnisse der Messung auf sn07 (32 Kerne)

	Size = 1025														
	Laufzeit in Sekunden														
Sequentiel Jacobi	5650,000														
Sequential Gaus-Seidel	652,000														
Thread Zahl	1			4			8			12			32		
	Laufzeit in Sekunden	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz	Laufzeit in s	Speedup	Effizienz
Jacobi SSE	5300,000	1,066	1,066	1200,000	4,708	1,18	657,000	8,600	1,07	420,000	13,452	1,12	180,000	31,389	0,98
Gaus-Seidel RS SSE	260,000	2,508	2,508	53,000	12,302	3,08	25,000	26,080	3,26	17,000	38,353	3,2	9,870	66,059	2,06

Abbildung 17: Ergebnisse der Messung auf sn07 (32 Kerne)

		size=128							
		Thread Zahl=1		Thread Zahl=4		Thread Zahl=8		Thread Zahl=16	
		Zeit in Sekunden	Speedup	Zeit in Sekunden	Speedup	Zeit in Sekunden	Speedup	Zeit in Sekunden	Speedup
GCC	Jacobi Sequential	1,8	1	1,8	1	1,8	1	1,8	1
	Jacobi	3,4	0,53	1	1,8	1	1,8	3,7	0,49
	Jacobi SSE	1,8	1	0,5	3,6	0,4	4,5	2,7	0,67
	Gaus-Seidel	2,7	1	2,7	1	2,7	1	2,7	1
	Gaus-Seidel Naiv	2,7	1	0,7	3,86	0,44	6,14	2	1,35
	Gaus-Seidel RS	1,3	2,08	0,4	6,75	0,31	8,68	2,1	1,29
	Gaus-Seidel RS SSE	1,1	2,45	0,3	9	0,29	9,25	2,1	1,29
	Wavefront	2,1	1,29	9,2	0,29	16,4	0,16	224	0,01
	Wavefront Cache	2	1,35	10,6	0,25	18,4	0,15	261	0,01
ICC	Jacobi Sequential	1,8	1	1,8	1	1,8	1	1,8	1
	Jacobi	19,7	0,09	5	0,36	2,6	0,69	4,3	0,42
	Jacobi SSE	1,7	1,06	0,52	3,46	0,41	4,42	1,1	1,64
	Gaus-Seidel	2,7	1	2,7	1	2,7	1	2,7	1
	Gaus-Seidel Naiv	8,9	0,3	2,4	1,13	1,29	2,09	2,2	1,23
	Gaus-Seidel RS	1,1	2,45	0,37	7,3	0,32	8,54	0,9	3
	Gaus-Seidel RS SSE	1,08	2,5	0,37	7,28	0,32	8,52	0,9	3
	Wavefront	3	0,9	12	0,23	24	0,11	94	0,03
	Wavefront Cache	2	1,35	13	0,21	28	0,1	108	0,03

Abbildung 18: Alte Performancemessung für *size*=128 auf sn02(8 Kerne) mit allen Implementierungen. Das Abbruchkriterium ist hierbei noch etwas feiner eingestellt.

		size=256					
		Thread Zahl=1		Thread Zahl=4		Thread Zahl=8	
		Zeit in Sekunden	Speedup	Zeit in Sekunden	Speedup	Zeit in Sekunden	Speedup
GCC	Jacobi Sequential	27	1	27	1	27	1
	Jacobi	46	0,59	12	2,25	6,3	4,29
	Jacobi SSE	25	1,08	6,4	4,22	3,7	7,3
	Gaus-Seidel	36	1	36,6	1	36,4	1
	Gaus-Seidel Naiv	35	1,03	9	4,07	4,8	7,58
	Gaus-Seidel RS	18	2	4,8	7,63	2,7	13,48
	Gaus-Seidel RS SSE	14	2,57	4	9,15	2,3	15,83
	Wavefront	33	1,09	64	0,57	107	0,34
	Wavefront Cache	21	1,71	78	0,47	128	0,28
ICC	Jacobi Sequential	27	1	27	1	27	1
	Jacobi	277	0,1	71	0,38	36	0,75
	Jacobi SSE	22,5	1,2	6,2	4,35	3,6	7,5
	Gaus-Seidel	36	1	37	1	37	1
	Gaus-Seidel Naiv	127	0,28	33	1,12	17	2,18
	Gaus-Seidel RS	14,4	2,5	4	9,25	2,4	15,42
	Gaus-Seidel RS SSE	13,9	2,59	4	9,25	2,4	15,42
	Wavefront	28	1,29	81	0,46	166	0,22
	Wavefront Cache	22,5	1,6	96	0,39	197	0,19

Abbildung 19: Alte Performancemessung für *size*=256 auf sn02(8 Kerne) mit allen Implementierungen. Das Abbruchkriterium ist hierbei noch etwas feiner eingestellt.

		size=32									
		Thread Zahl=1		Thread Zahl=4		Thread Zahl=8		Thread Zahl=12		Thread Zahl=16	
		Zeit in Sekunden	Speedup	Zeit in Sekunden	Speedup	Zeit in Sekunden	Speedup	Zeit in Sekunden	Speedup	Zeit in Sekunden	Speedup
GCC	Jacobi Sequential	0,014	1,000	0,014	1,000	0,014	1,000	0,014	1,000	0,014	1,000
	Jacobi	0,020	0,700	0,020	0,700	0,023	0,609	0,027	0,519	0,050	0,280
	Jacobi SSE	0,011	1,273	0,012	1,167	0,020	0,700	0,025	0,560	0,037	0,378
	Gaus-Seidel	0,015	1,000	0,015	1,000	0,015	1,000	0,015	1,000	0,015	1,000
	Gaus-Seidel Naiv	0,015	1,000	0,013	1,154	0,015	1,000	0,020	0,750	0,031	0,484
	Gaus-Seidel RS	0,009	1,667	0,010	1,500	0,016	0,938	0,022	0,682	0,032	0,469
	Gaus-Seidel RS SSE	0,007	2,143	0,010	1,500	0,015	1,000	0,021	0,714	0,031	0,484
	Wavefront	0,031	0,484	0,218	0,069	0,044	0,345	0,562	0,027	0,860	0,017
	Wavefront Cache	0,033	0,455	0,248	0,060	0,503	0,030	0,064	0,234	0,927	0,016
ICC	Jacobi Sequential	0,014	1,000	0,014	1,000	0,014	1,000	0,014	1,000	0,014	1,000
	Jacobi	0,096	0,146	0,032	0,438	0,035	0,400	0,043	0,326	0,064	0,219
	Jacobi SSE	0,010	1,400	0,011	1,273	0,025	0,560	0,044	0,318	0,070	0,200
	Gaus-Seidel	0,015	1,000	0,015	1,000	0,015	1,000	0,015	1,000	0,015	1,000
	Gaus-Seidel Naiv	0,044	0,341	0,018	0,833	0,020	0,750	0,029	0,517	0,050	0,300
	Gaus-Seidel RS	0,008	1,875	0,011	1,364	0,027	0,556	0,038	0,395	0,062	0,242
	Gaus-Seidel RS SSE	0,007	2,143	0,010	1,500	0,025	0,600	0,043	0,349	0,067	0,224
	Wavefront	0,030	0,500	0,264	0,057	0,501	0,030	1,200	0,013	1,800	0,008
	Wavefront Cache	0,031	0,484	0,301	0,050	0,647	0,023	1,400	0,011	2,100	0,007

Abbildung 20: Alte Performancemessung für $size$=32 auf sn07(32 Kerne) mit allen Implementierungen.Das Abbruchkriterium ist hierbei noch etwas feiner eingestellt.

		size=128									
		Thread Zahl=1		Thread Zahl=4		Thread Zahl=8		Thread Zahl=12		Thread Zahl=16	
		Zeit in Sekunden	Speedup	Zeit in Sekunden	Speedup	Zeit in Sekunden	Speedup	Zeit in Sekunden	Speedup	Zeit in Sekunden	Speedup
GCC	Jacobi Sequential	2,150	1,000	2,150	1,000	2,150	1,000	2,150	1,000	2,150	1,000
	Jacobi	4,000	0,538	1,100	1,955	0,730	2,945	0,650	3,308	0,900	2,389
	Jacobi SSE	2,150	1,000	0,670	3,209	0,550	3,909	0,530	4,057	0,700	3,071
	Gaus-Seidel	3,200	1,000	3,200	1,000	3,200	1,000	3,200	1,000	3,200	1,000
	Gaus-Seidel Naiv	3,200	1,000	0,900	3,556	0,550	5,818	0,470	6,809	0,480	6,667
	Gaus-Seidel RS	1,640	1,951	0,500	6,400	0,410	7,805	0,420	7,619	0,510	6,275
	Gaus-Seidel RS SSE	1,354	2,363	0,440	7,273	0,400	8,000	0,440	7,273	0,510	6,275
	Wavefront	2,600	1,231	11,500	0,278	22,000	0,145	31,000	0,103	42,000	0,076
	Wavefront Cache	2,470	1,296	12,500	0,256	27,000	0,119	37,000	0,086	51,000	0,063
ICC	Jacobi Sequential	2,200	0,977	2,200	0,977	2,200	0,977	2,200	0,977	2,200	0,977
	Jacobi	24,000	0,090	6,200	0,347	3,200	0,672	2,200	0,977	2,000	1,075
	Jacobi SSE	2,000	1,075	0,700	3,071	0,601	3,577	0,700	3,071	1,000	2,150
	Gaus-Seidel	3,200	1,000	3,200	1,000	3,200	1,000	3,200	1,000	3,200	1,000
	Gaus-Seidel Naiv	10,700	0,299	2,940	1,088	1,600	2,000	1,200	2,667	1,200	2,667
	Gaus-Seidel RS	1,300	2,462	0,470	6,809	0,480	6,667	0,500	6,400	0,900	3,556
	Gaus-Seidel RS SSE	1,290	2,481	0,480	6,667	0,480	6,667	0,588	5,442	0,910	3,516
	Wavefront	3,700	0,865	18,000	0,178	40,000	0,080	65,000	0,049	103,000	0,031
	Wavefront Cache	2,400	1,333	21,000	0,152	45,000	0,071	69,000	0,046	117,000	0,027

Abbildung 21: Alte Performancemessung für $size$=128 auf sn07(32 Kerne) mit allen Implementierungen.Das Abbruchkriterium ist hierbei noch etwas feiner eingestellt.

		Thread Zahl =4		Thread Zahl = 8	
		Zeit in Sekunden	Speedup	Zeit in Sekunden	Speedup
GCC	Jacobi Sequential	70,5	1	47,9	1
	Jacobi	19	3,71	9,5	5,04
	Jacobi SSE	10,6	6,65	5,4	8,87
	Gaus-Seidel	135	1	133	1
	Gaus-Seidel Naiv	32,1	4,21	16	8,31
	Gaus-Seidel RS	16,3	8,28	8,2	16,22
	Gaus-Seidel RS SSE	13,4	10,07	6,9	19,28
	Wavefront	98	1,38	113	1,18
	Wavefront Cache	77	1,75	113	1,18
ICC	Jacobi Sequential	72	1	49	1
	Jacobi	125	0,58	62	0,79
	Jacobi SSE	9,4	7,66	4,7	10,43
	Gaus-Seidel	135	1	136	1
	Gaus-Seidel Naiv	125	1,08	62	2,19
	Gaus-Seidel RS	13	10,38	6,7	20,3
	Gaus-Seidel RS SSE	12	11,25	6,4	21,25
	Wavefront	116	1,16	160	0,85
	Wavefront Cache	94	1,44	157	0,87

Abbildung 22: Alte Performancemessung für $size$=1024 auf sn02(8 Kerne) mit allen Implementierungen. Der Lauf ging nur bis 10000 Iterationen.