Praktikum Multicore-Programmierung
Abschlussprojekt 1

Gruppe 3: Sarah Lutteropp und Johannes Sailer

5. 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`

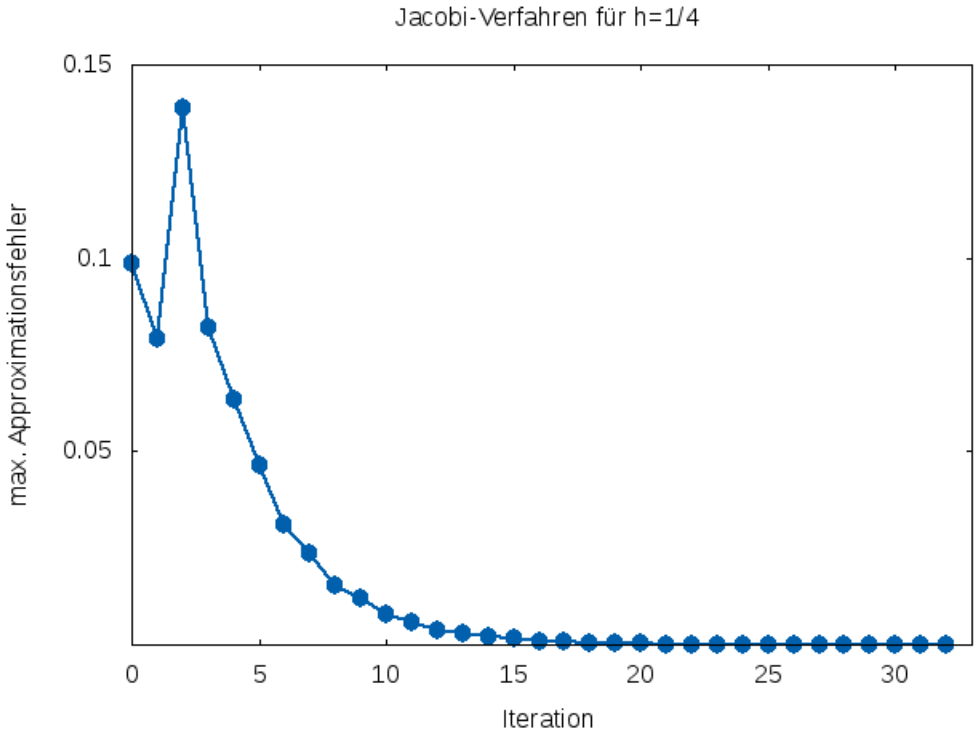
2.2 Approximationsfehler

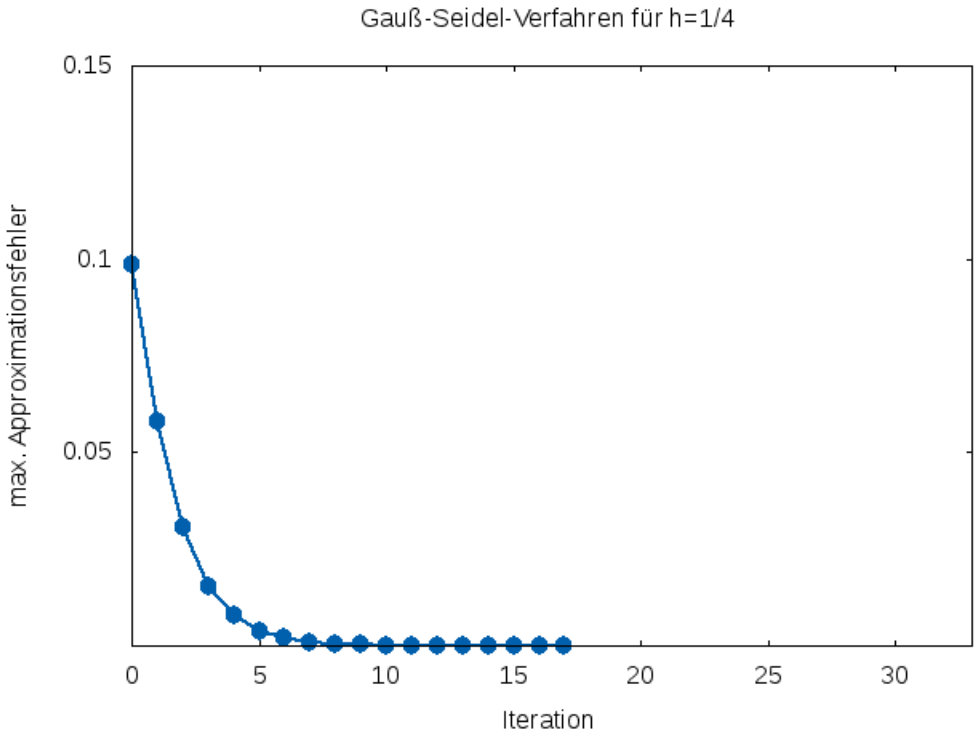
Wir haben den maximalen absoluten Approximationsfehler in der euklidischen Norm in jedem Iterationsschritt für das Jacobi- und das Gauß-Seidel-Verfahren gemessen. Sei $U^{(*)}$ die analytische Lösung für U. Der maximale absolute Approximationsfehler in euklidischer Norm in Iteration k ist definiert als

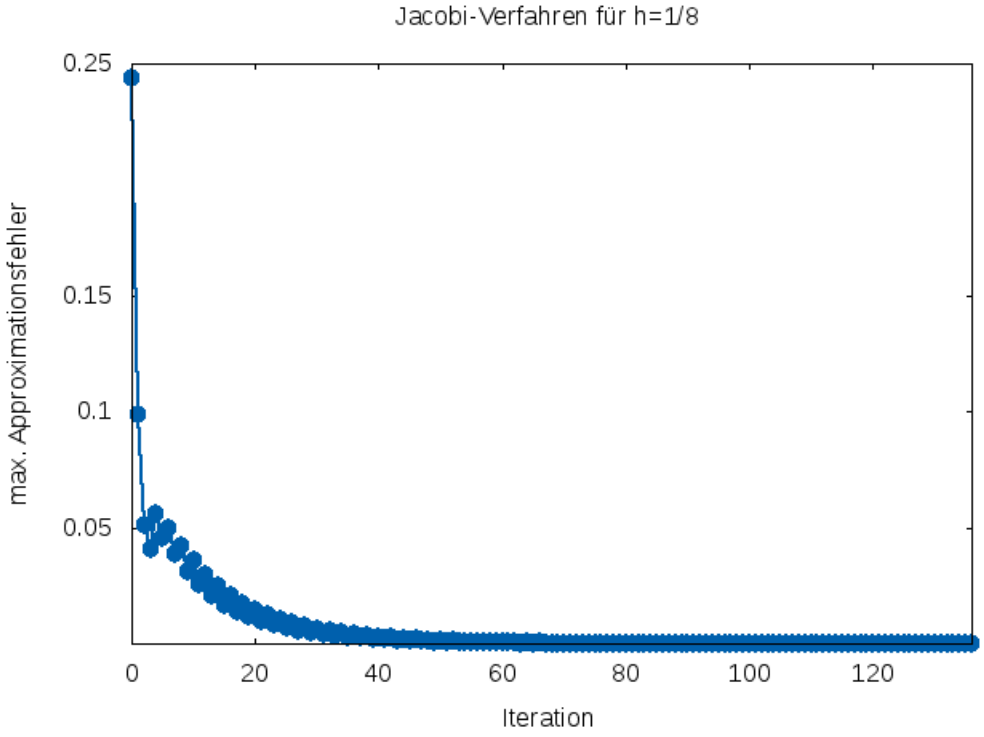
$$\max_{(i,j)\in\{1,\dots,n\}^2} \sqrt{\left(U_{i,j}^{(k)} - U_{i,j}^{(*)}\right)^2}$$

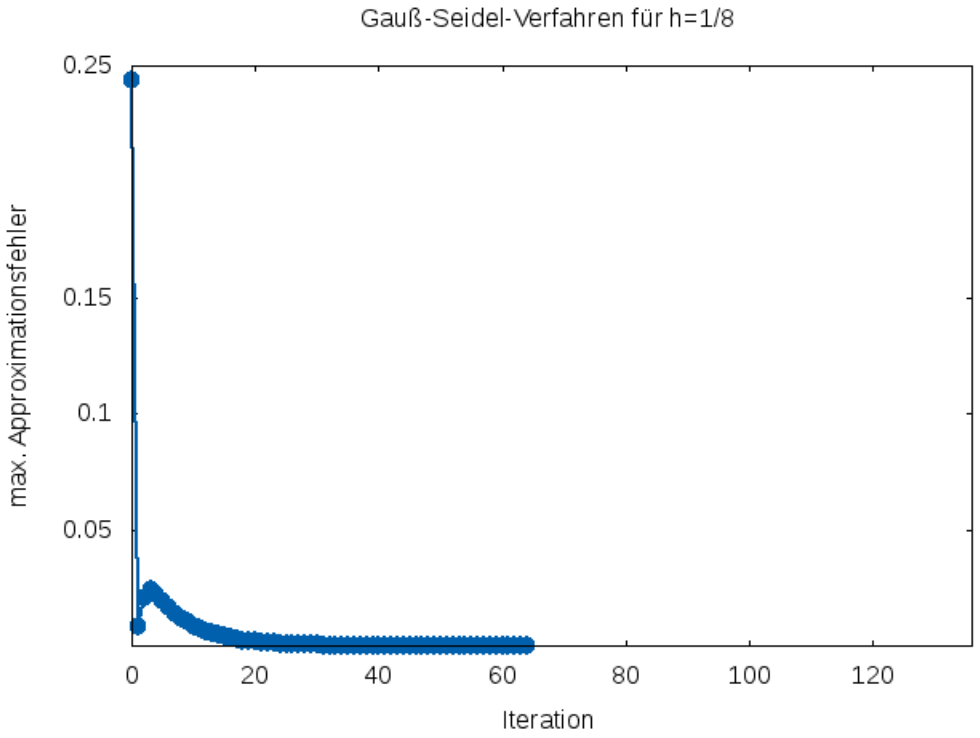
Sei $x_i = i * h$ und $y_j = j * h$ für $i, j = 0, \dots, n+1$. Wir haben die analytische Lösung mittels der Funktion $u(x_i, y_j) = \begin{cases} 16x_i(1-x_i)y_i(1-y_i) & \text{, falls } 0 < x_i < 1 \text{ und } 0 < y_j < 1 \\ 0 & \text{, sonst} \end{cases}$ berechnet.

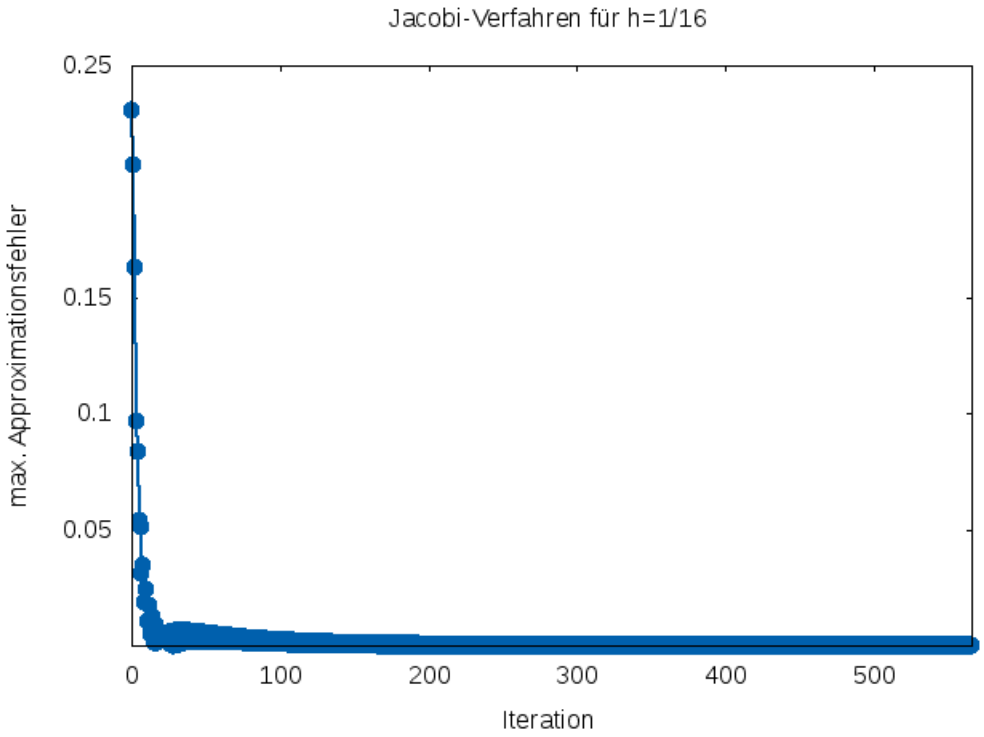
Wir haben die Messungen nur für die Verfeinerungen $h \in \left\{\frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}\right\}$ durchgeführt, da für kleinere h die Grafiken zu groß werden und man nichts mehr erkennt. Die Grafiken haben wir mit `gnuplot` erstellt. Beim Vergleich der Grafiken fällt auf, 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.

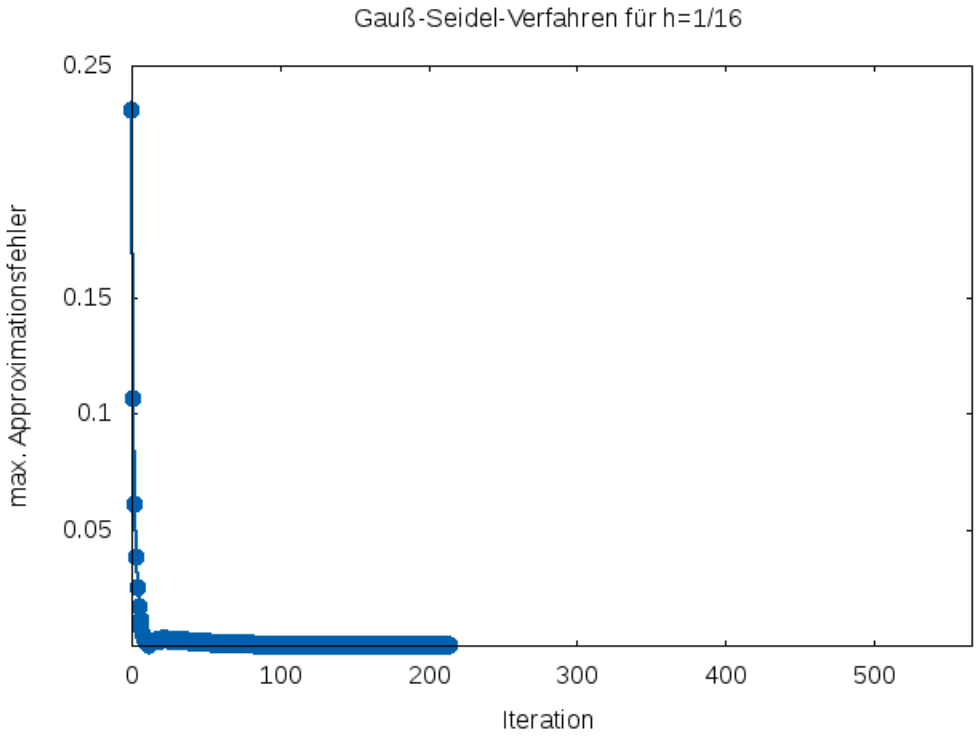


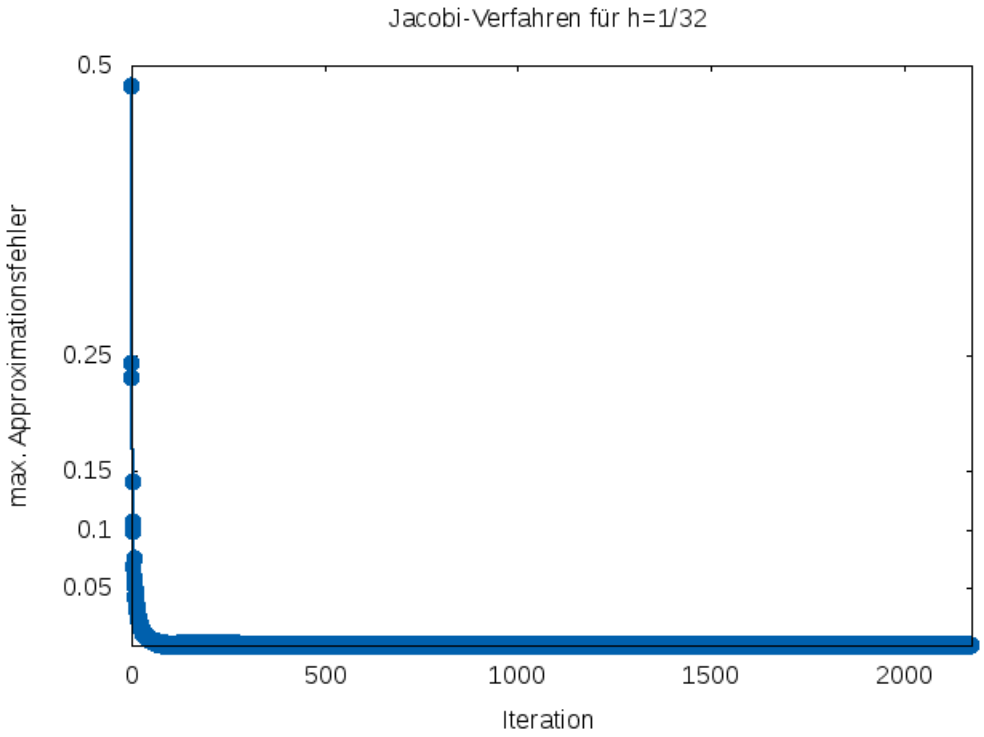

Gauß-Seidel-Verfahren für h=1/4
max. Approximationsfehler
0.15
0.1
0.05
0
5
10
15
20
25
30
Iteration

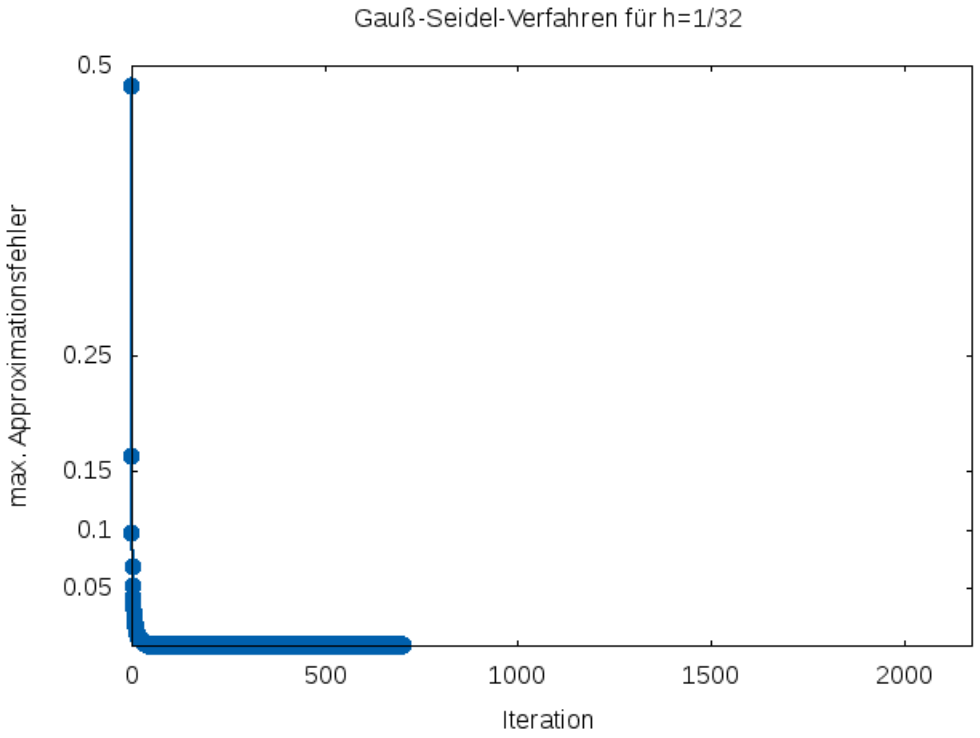

Jacobi-Verfahren für h=1/8
max. Approximationsfehler
0.25
0.15
0.1
0.05
0
20
40
60
80
100
120
Iteration


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

3.1 Jacobi-Verfahren

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.

Hierbei haben wir folgenden Speedup bei verschiedenen Problemgrößen h und sowie verschiedenen Prozessorzahlen p gemessen: **??**

3.2 Gauß-Seidel-Verfahren

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.

3.2.1 Naiver Parallelisierungsansatz

Der naive Parallelisierungsansatz ist, wie in Abschnitt 3.1 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.

```
#pragma omp parallel for private(j, i) collapse(2) reduction(+:diff)
for (j = 1; j < size - 1; j++)
{
    for (i = 1; i < size - 1; i++)
    {
        float old = a1[CO(i,j)];
        a1[CO(i,j)] = a1[CO(i, j - 1)]
                    + a1[CO(i - 1, j)]
                    + a1[CO(i, j + 1)]
                    + a1[CO(i + 1, j)]
                    + functionTable[CO(i, j)];
        a1[CO(i,j)] *= 0.25;

        diff += fabsf(a1[CO(i,j)] - old);
    }
}
```

Abbildung 3: Naive Parallelisierung des Gauß-Seidel-Verfahrens. Hierbei kann es zu Wettlaufsituationen kommen, da nicht garantiert ist, in welcher Reihenfolge beispielsweise auf den Eintrag `a1[CO(i,j+1)]` zugegriffen wird.

3.2.2 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 4 gezeigt.

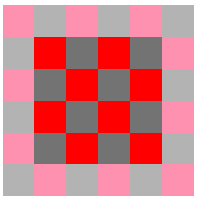
$U =$ 

Abbildung 4: 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.

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

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

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

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

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

Abbildung 5: 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 6 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 6: Berechnung der Nachbarindizes für ungerade Werte von $size$ am Beispiel $size = 7$

Messung von Speedup und Effizienz unter verschiedenen Verfeinerungen, Skalierbarkeit TODO

3.2.3 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 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.

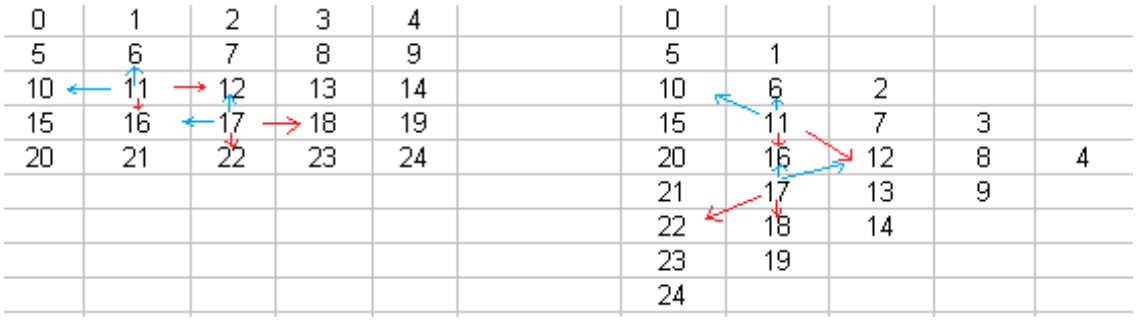


Abbildung 7: Veranschaulichung der Permutation, welche verwendet wird sowie der Verschiebung der Indizes anhand eines Beispiels. Die Variable `hack` ist dabei blau und `hack2` rot.

4 Methodenwahl

Es empfiehlt sich das Gauß-Seidel-Verfahren zu nutzen, weil es im allgemeinen schneller konvergiert als das Jacobi-Verfahren. Bis zu einer Matrixgröße 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