Praktikum Multicore-Programmierung
Abschlussprojekt 1

Gruppe 3: Sarah Lutteropp und Johannes Sailer

2. 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.

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

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

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 $TODO \times TODO$-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.

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 $(\frac{1}{h}+1) \times (\frac{1}{h}+1)$.

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.

TODO: Siehe z.B. hier: `http://www.mathematik.uni-muenchen.de/~lerdos/SS08/Num/9.pdf`

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:

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

Oder elementweise:

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

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

1.1.2 Abbruchkriterium

Unser Abbruchkriterium nimmt von allen Matrix Einträgen die Differenz zum neuen Punkt und summiert die Beträge davon auf. Dies schien uns recht Performant zu sein und da es auch in der Vorlesung Heterogene Parallele Rechensysteme Verwendung findet sollte es nicht so schlecht sein. Es werden auch keine Sprünge oder ähnliches verwendet, von demher ist der Overhead konstant. Zur Auswahl standen außerdem das selbe wie oben in der euklidischen Norm zu berechnen, jedoch erschien uns oberes schneller. Desweiteren dachten wir an den Vergleich der Veränderung der Hauptdiagonalelemente, was in einem zusätzlichen if geendet hätte. Dann gab es noch eine Version mit Eigenwerten, welche wir nicht haben, was sich dadurch erledigt hatte.

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.

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

1.2.1 Herleitung

Blabla

1.2.2 Abbruchkriterium

Für das Gauß-Seidel-Verfahren haben wir dasselbe Abbruchkriterium wie in Abschnitt 1.1.2 verwendet. (TODO: Begründung)

1.3 Vergleich der Konvergenz und Stabilität beider Verfahren

Beide Verfahren sind uneingeschränkt stabil.

	h=4	h=32	H=128
Gaus-Seidel	13	1067	13425
Jacobi	24	2280	29230

Abbildung 1: Beide Verfahren für verschiedene Verfeinerungen, bis das 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 $\frac{1}{h} + 1$ entspricht. Dies hat den Vorteil, dass der Nutzer den Grad der Verfeinerung exakt ohne Gleitkommaungenauigkeiten angeben kann und U eine $size \times size$-Matrix ist.

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

2.1 Laufzeiten bei verschiedenen Verfeinerungen

Blabla

2.2 Approximationsfehler

Wir haben den Approximationsfehler nur für h=4 und h=32 dargestellt, weil Open Office nicht genug Punkte verarbeiten kann.

3 Parallelisierung

Blabla

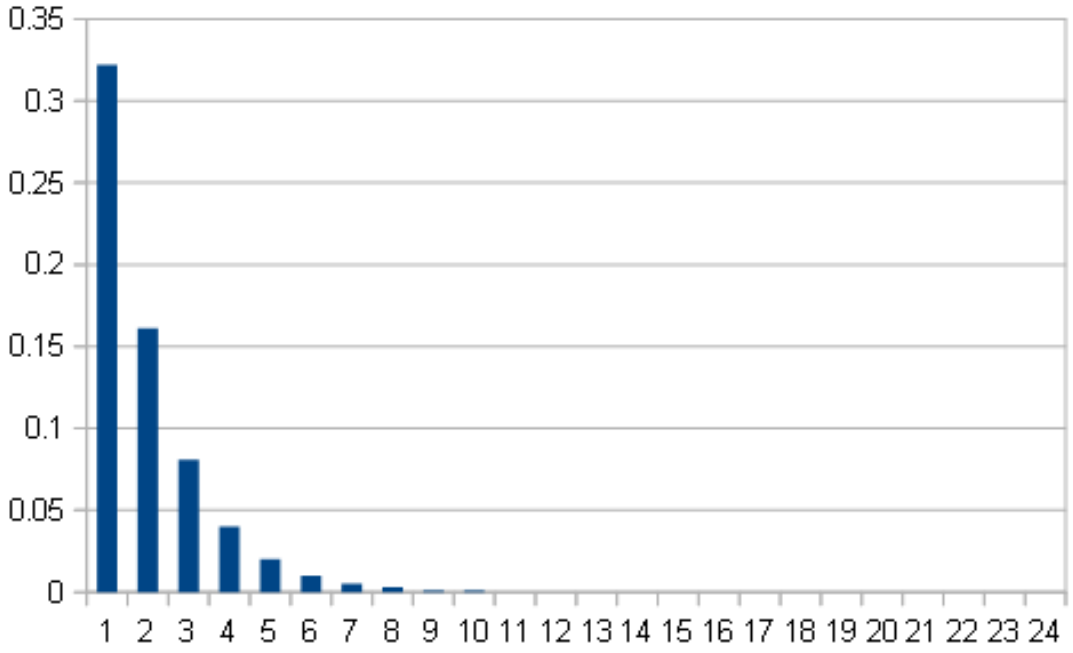

Abbildung 2: Jacobi Approximationsfehler für h=4

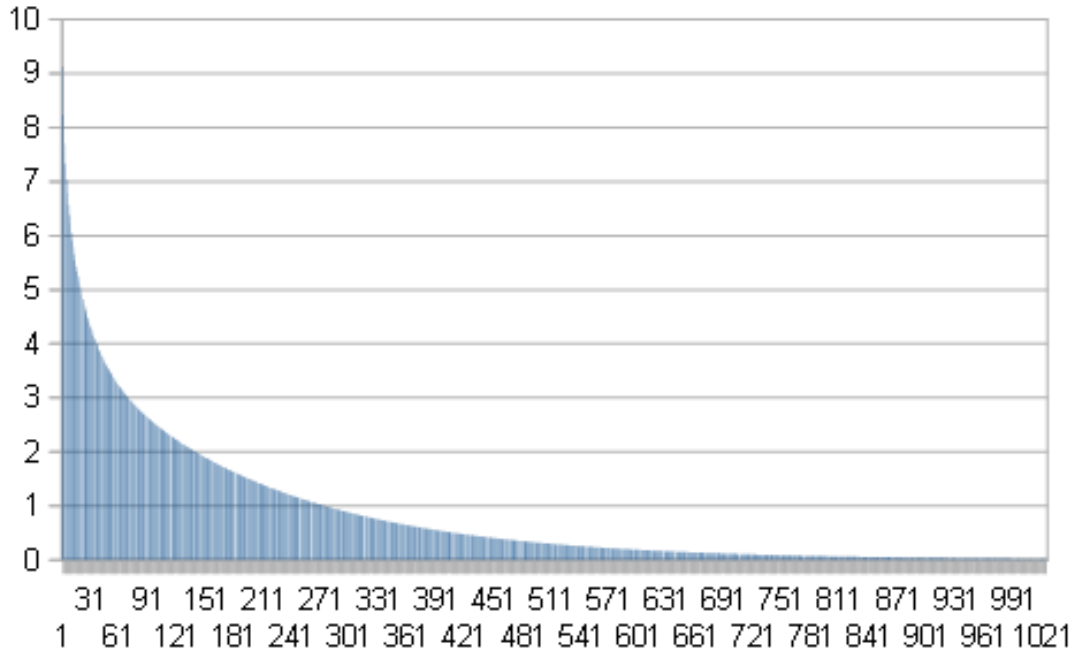

Abbildung 3: Jacobi Approximationsfehler für h=32

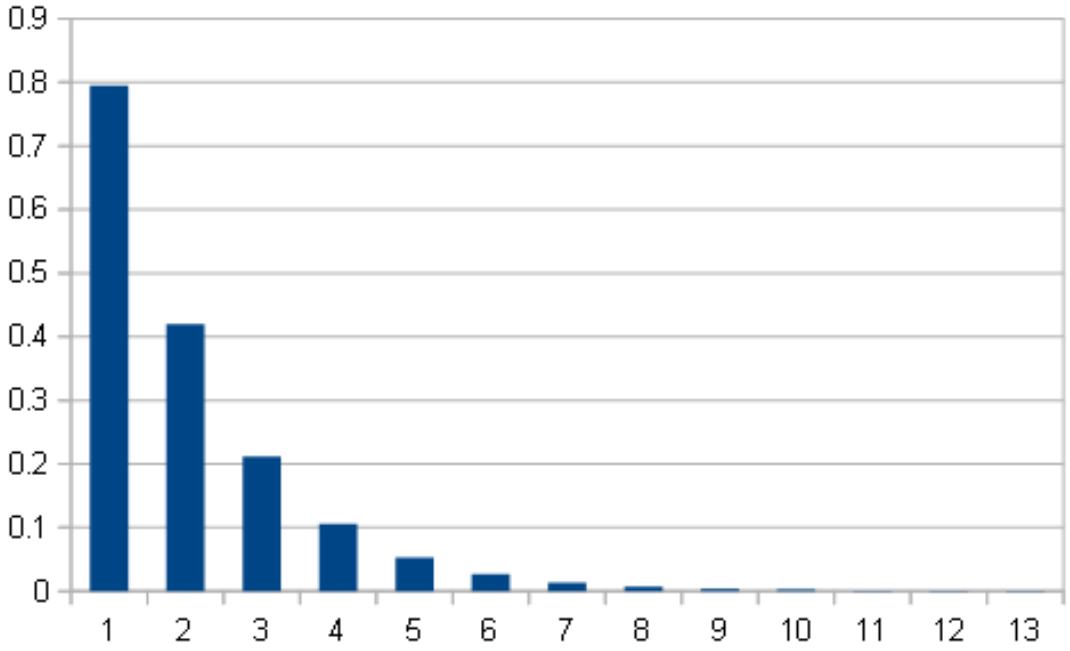

Abbildung 4: Gaus-Seidel Approximationsfehler für h=4

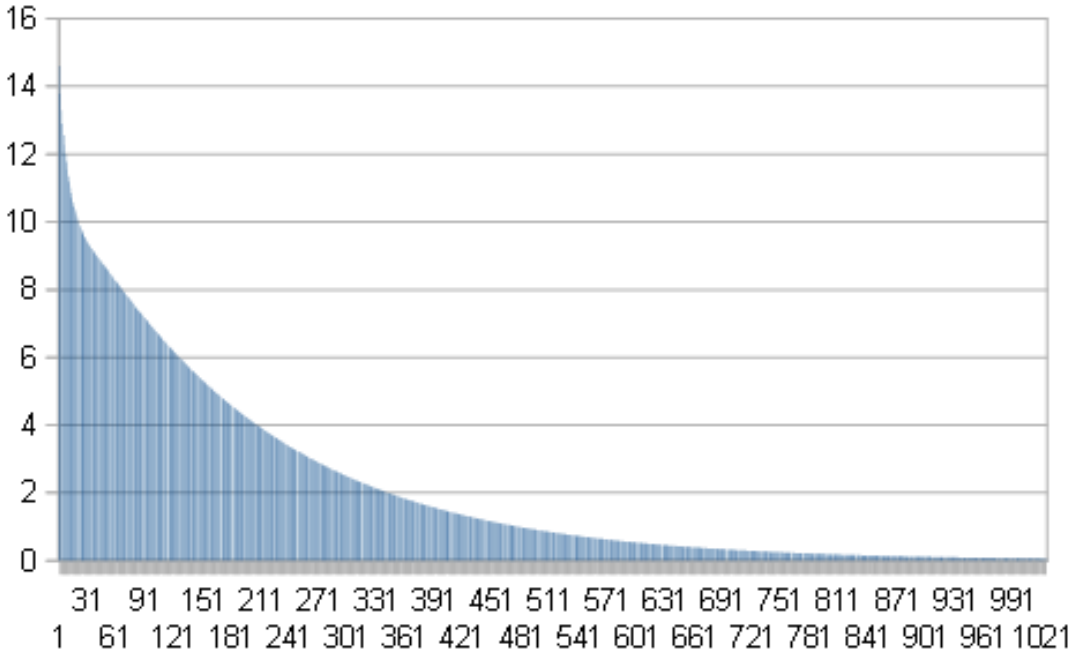

Abbildung 5: Gaus-SeidelApproximationsfehler für h=32

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: TODO

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.

Im Fall des Gauß-Seidel-Verfahrens tritt dies in folgendem Beispiel auf (TODO: Beispiel):

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

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 6 gezeigt.

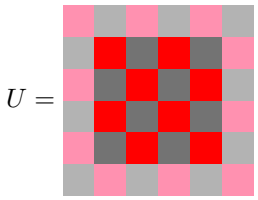

Abbildung 6: Aufteilung der Lösungsmatrix U in rote und schwarze Zellen, sodass zwischen Zellen gleicher Farbe keine Datenabhängigkeiten bestehen. Die Randzellen, deren Einträge per Annahme immer den Wert 0 enthalten, sind in blasseren Farben markiert.

Da die Matrixeinträge gleicher Farbe nicht voneinander abhängen, können wir diese parallel berechnen. Dies führt zu folgendem Ansatz: (TODO: Pseudocode)

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 7 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 7: 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 8 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 8: Berechnung der Nachbarindizes für ungerade Werte von $size$ am Beispiel $size = 7$

		size= 3							
		Thread Zahl=1		Thread Zahl=4		Thread Zahl=8		Thread Zahl=16	
		Zeit in ms	Speedup	Zeit in ms	Speedup	Zeit in ms	Speedup	Zeit in ms	Speedup
GCC	Jacobi Sequential	0,000		0,000		0,000		0,000	
	Jacobi	0,002		0,011		2,063		0,303	
	Jacobi SSE	0,002		0,009		0,015		0,303	
	Gaus-Seidel	0,000		0,000		0,000		0,000	
	Gaus-Seidel Naiv	0,002		0,011		0,017		0,310	
	Gaus-Seidel RS	0,002		0,013		0,022		0,475	
	Gaus-Seidel RS SSE	0,002		0,014		0,022		0,462	
	Wavefront	0,002		0,009		0,015		0,299	
	Wavefront Cache	0,002		0,007		0,011		0,223	
ICC	Jacobi Sequential	0,000		0,000		0,000		0,000	
	Jacobi	0,001		0,013		0,021		0,115	
	Jacobi SSE	0,001		0,013		0,021		0,067	
	Gaus-Seidel	0,000		0,000		0,000		0,000	
	Gaus-Seidel Naiv	0,001		0,013		0,021		0,081	
	Gaus-Seidel RS	0,002		0,019		0,034		0,167	
	Gaus-Seidel RS SSE	0,001		0,018		0,034		0,114	
	Wavefront	0,001		0,012		0,021		0,067	
	Wavefront Cache	0,001		0,009		0,014		0,053	

Abbildung 9: Performance für size=3 auf sn02(8 Kerne).

Messung von Speedup und Effizienz unter verschiedenen Verfeinerungen, Skalierbarkeit

3.2.3 Erweiterter Parallelisierungsansatz: Wavefront

Beim Wavefront Algorithmus wird anstatt über die Spalten oder Zeilen zu iterieren über die Diagonalen 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 Diagonalelemente. D.h. es lohnt sich erst bei großen Diagonalen.

Bei der Wavefront Cache Lösung wurden die Matrix zuerst in ein anderes Array kopiert, in dem die Diagonalen 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 Diagonalen), 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 Diagonalen und die innere über die Elemente der Diagonalen.

		size = 5							
		Thread Zahl=1		Thread Zahl=4		Thread Zahl=8		Thread Zahl=16	
		Zeit in ms	Speedup	Zeit in ms	Speedup	Zeit in ms	Speedup	Zeit in ms	Speedup
GCC	Jacobi Sequential	0,003	1	0,003	1,000	0,003	1,000	0,003	1
	Jacobi	0,024	0,13	0,158	0,019	0,945	0,003	4,128	0
	Jacobi SSE	0,020	0,15	0,121	0,025	0,186	0,016	4,045	0
	Gaus-Seidel	0,002	1,000	0,002	1,000	0,002	1,000	0,002	1
	Gaus-Seidel Naiv	0,013	0,154	0,130	0,015	0,215	0,009	3,812	0
	Gaus-Seidel RS	0,020	0,100	0,114	0,018	0,194	0,010	3,465	0
	Gaus-Seidel RS SSE	0,020	0,100	0,115	0,017	0,200	0,010	3,483	0
	Wavefront	0,450	0,004	0,279	0,007	0,497	0,004	7,732	0
	Wavefront Cache	0,480	0,004	-		-		0,479	0
ICC	Jacobi Sequential	0,002	1,000	0,002	1,000	0,002	1,000	0,002	1
	Jacobi	0,029	0,069	0,149	0,013	0,334	0,006	1,443	0
	Jacobi SSE	0,019	0,105	0,142	0,014	0,341	0,006	1,408	0
	Gaus-Seidel	0,002	1,000	0,002	1,000	0,002	1,000	0,002	1
	Gaus-Seidel Naiv	0,015	0,133	0,092	0,022	0,241	0,008	1,129	0
	Gaus-Seidel RS	0,018	0,111	0,149	0,013	0,346	0,006	1,423	0
	Gaus-Seidel RS SSE	0,018	0,111	0,149	0,013	0,342	0,006	1,454	0
	Wavefront	0,042	0,048	0,375	0,005	0,823	0,002	2,982	0
	Wavefront Cache	0,045	0,044	0,363	0,006	0,687	0,003	1,136	0

Abbildung 10: Performance für size=5 auf sn02(8 Kerne).

		size=32							
		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	0,008	1,000	0,008	1,000	0,008	1,000	0,008	1,000
	Jacobi	0,016	0,500	0,011	0,727	0,021	0,381	0,197	0,041
	Jacobi SSE	0,009	0,889	0,007	1,143	0,014	0,571	0,193	0,041
	Gaus-Seidel	0,012	1,000	0,012	1,000	0,012	1,000	0,012	1,000
	Gaus-Seidel Naiv	0,013	0,923	0,007	1,714	0,009	1,333	0,100	0,120
	Gaus-Seidel RS	0,007	1,714	0,006	2,000	0,012	1,000	0,164	0,073
	Gaus-Seidel RS SSE	0,006	2,000	0,006	2,000	0,011	1,091	0,163	0,074
	Wavefront	0,025	0,480	0,161	0,075	0,280	0,043	4,300	0,003
	Wavefront Cache	0,026	0,462	0,173	0,069	0,313	0,038	4,500	0,003
ICC	Jacobi Sequential	0,008	1,000	0,008	1,000	0,008	1,000	0,008	1,000
	Jacobi	0,075	0,107	0,026	0,308	0,023	0,348	0,090	0,089
	Jacobi SSE	0,008	1,000	0,009	0,889	0,018	0,444	0,054	0,148
	Gaus-Seidel	0,012	1,000	0,012	1,000	0,012	1,000	0,021	1,000
	Gaus-Seidel Naiv	0,037	0,324	0,014	0,857	0,015	0,800	0,050	0,420
	Gaus-Seidel RS	0,006	2,000	0,008	1,500	0,015	0,800	0,055	0,382
	Gaus-Seidel RS SSE	0,006	2,000	0,009	1,333	0,015	0,800	0,056	0,375
	Wavefront	0,026	0,462	0,220	0,055	0,441	0,027	1,600	0,013
	Wavefront Cache	0,026	0,462	0,230	0,052	0,512	0,023	2,000	0,011

Abbildung 11: Performance für size=32 auf sn02(8 Kerne).

		size = 65							
		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	0,142	1,000	0,142	1,000	0,142	1,000	0,142	1,000
	Jacobi	0,288	0,493	0,105	1,352	1,000	0,142	0,830	0,171
	Jacobi SSE	0,155	0,916	0,066	2,152	0,081	1,753	0,799	0,178
	Gaus-Seidel	0,202	1,000	0,202	1,000	0,202	1,000	0,202	1,000
	Gaus-Seidel Naiv	0,211	0,957	0,073	2,767	0,055	3,673	0,409	0,494
	Gaus-Seidel RS	0,111	1,820	0,050	4,040	0,057	3,544	0,668	0,302
	Gaus-Seidel RS SSE	0,062	3,258	0,040	5,050	0,056	3,607	0,637	0,317
	Wavefront	0,241	0,838	1,310	0,154	2,300	0,088	34,827	0,006
	Wavefront Cache	0,257	0,786	1,500	0,135	2,700	0,075	39,000	0,005
ICC	Jacobi Sequential	0,151	1,000	0,151	1,000	0,151	1,000	0,151	1,000
	Jacobi	1,511	0,100	0,426	0,354	0,249	0,606	0,420	0,360
	Jacobi SSE	0,147	1,027	0,073	2,068	0,091	1,659	0,242	0,624
	Gaus-Seidel	0,207	1,000	0,207	1,000	0,207	1,000	0,257	1,000
	Gaus-Seidel Naiv	0,662	0,313	0,194	1,067	0,119	1,739	0,240	1,071
	Gaus-Seidel RS	0,094	2,202	0,049	4,224	0,066	3,136	0,208	1,236
	Gaus-Seidel RS SSE	0,061	3,393	0,044	4,705	0,067	3,090	0,201	1,279
	Wavefront	0,245	0,845	1,856	0,112	3,700	0,056	14,000	0,018
	Wavefront Cache	0,249	0,831	2,100	0,099	4,200	0,049	17,000	0,015

Abbildung 12: Performance für size=65 auf sn02(8 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 13: Performance für size=128 auf sn02(8 Kerne).

		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 14: Performance für size=256 auf sn02(8 Kerne).

		size=3									
		Thread Zahl=1		Thread Zahl=4		Thread Zahl=8		Thread Zahl=12		Thread Zahl=16	
		Zeit in ms	Speedup	Zeit in ms	Speedup	Zeit in ms	Speedup	Zeit in ms	Speedup	Zeit in ms	Speedup
GCC	Jacobi Sequential	0,000		0,000		0,050		0,000		0,000	
	Jacobi	0,002		0,017		0,025		0,409		0,044	
	Jacobi SSE	0,002		0,013		0,003		0,035		0,035	
	Gaus-Seidel	0,000		0,000		0,000		0,000		0,000	
	Gaus-Seidel Naiv	0,002		0,015		0,003		0,042		0,044	
	Gaus-Seidel RS	0,003		0,019		0,037		0,052		0,055	
	Gaus-Seidel RS SSE	0,003		0,019		0,036		0,053		0,057	
	Wavefront	0,002		0,013		0,021		0,035		0,037	
	Wavefront Cache	0,002		0,009		0,016		0,026		0,027	
ICC	Jacobi Sequential	0,000		0,000		0,000		0,000		0,000	
	Jacobi	0,002		0,018		0,031		0,050		0,070	
	Jacobi SSE	0,002		0,017		0,028		0,049		0,076	
	Gaus-Seidel	0,000		0,000		0,000		0,000		0,000	
	Gaus-Seidel Naiv	0,002		0,017		0,028		0,049		0,074	
	Gaus-Seidel RS	0,003		0,026		0,048		0,087		0,013	
	Gaus-Seidel RS SSE	0,003		0,030		0,048		0,089		0,013	
	Wavefront	0,002		0,018		0,030		0,054		0,076	
	Wavefront Cache	0,002		0,014		0,020		0,035		0,047	

Abbildung 15: Performance für size=3 auf sn07(12 Kerne).

		size=5									
		Thread Zahl=1		Thread Zahl=4		Thread Zahl=8		Thread Zahl=12		Thread Zahl=16	
		Zeit in ms	Speedup	Zeit in ms	Speedup	Zeit in ms	Speedup	Zeit in ms	Speedup	Zeit in ms	Speedup
GCC	Jacobi Sequential	0,006		0,006		0,006		0,006		0,006	
	Jacobi	0,029		0,210		0,370		0,510		0,800	
	Jacobi SSE	0,024		0,170		0,280		0,390		0,618	
	Gaus-Seidel	0,002		0,002		0,002		0,002		0,002	
	Gaus-Seidel Naiv	0,016		0,124		0,331		0,440		0,700	
	Gaus-Seidel RS	0,025		0,156		0,270		0,400		0,606	
	Gaus-Seidel RS SSE	0,025		0,161		0,280		0,390		0,618	
	Wavefront	0,058		0,430		0,640		0,980		1,400	
	Wavefront Cache	0,060		-		0,043		0,056		0,083	
ICC	Jacobi Sequential	0,005		0,005		0,005		0,004		0,003	
	Jacobi	0,035		0,240		0,490		0,750		1,300	
	Jacobi SSE	0,022		0,230		0,510		0,800		1,300	
	Gaus-Seidel	0,002		0,002		0,002		0,002		0,002	
	Gaus-Seidel Naiv	0,018		0,145		0,410		0,780		1,000	
	Gaus-Seidel RS	0,022		0,235		0,510		0,750		1,350	
	Gaus-Seidel RS SSE	0,021		0,300		0,500		0,730		1,400	
	Wavefront	0,051		0,600		1,200		1,900		3,350	
	Wavefront Cache	0,053		0,500		1,000		0,108		0,166	

Abbildung 16: Performance für size=5 auf sn07(12 Kerne).

		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 17: Performance für size=32 auf sn07(12 Kerne).

		size=65									
		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	0,176	1,000	0,174	1,000	0,174	1,000	0,175	1,000	0,174	1,000
	Jacobi	0,349	0,504	0,139	1,252	0,125	1,392	0,160	1,094	0,160	1,088
	Jacobi SSE	0,186	0,946	0,085	2,047	0,100	1,740	0,140	1,250	0,170	1,024
	Gaus-Seidel	0,242	1,000	0,245	1,000	0,243	1,000	0,244	1,000	0,243	1,000
	Gaus-Seidel Naiv	0,252	0,960	0,087	2,816	0,079	3,076	0,080	3,050	0,085	2,859
	Gaus-Seidel RS	0,135	1,793	0,065	3,769	0,090	2,700	0,110	2,218	0,113	2,150
	Gaus-Seidel RS SSE	0,075	3,227	0,050	4,900	0,070	3,471	0,100	2,440	0,120	2,025
	Wavefront	0,282	0,858	1,750	0,140	3,400	0,071	4,800	0,051	6,500	0,037
	Wavefront Cache	0,300	0,807	1,890	0,130	4,000	0,061	5,500	0,044	7,500	0,032
ICC	Jacobi Sequential	0,185	1,000	0,185	1,000	0,185	1,000	0,185	1,000	0,185	1,000
	Jacobi	1,850	0,100	0,500	0,370	0,325	0,569	0,315	0,587	0,370	0,500
	Jacobi SSE	0,179	1,034	0,090	2,056	0,128	1,445	0,202	0,916	0,280	0,661
	Gaus-Seidel	0,249	1,000	0,249	1,000	0,248	1,004	0,248	1,004	0,248	1,004
	Gaus-Seidel Naiv	0,795	0,313	0,249	1,000	0,157	1,586	0,159	1,566	0,195	1,277
	Gaus-Seidel RS	0,113	2,204	0,073	3,411	0,104	2,394	0,170	1,465	0,250	0,996
	Gaus-Seidel RS SSE	0,074	3,365	0,070	3,557	0,104	2,394	0,195	1,277	0,255	0,976
	Wavefront	0,301	0,827	2,700	0,092	6,000	0,042	10,000	0,025	14,000	0,018
	Wavefront Cache	0,296	0,841	2,900	0,086	6,700	0,037	11,500	0,022	17,000	0,015

Abbildung 18: Performance für size=65 auf sn07(12 Kerne).

		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 19: Performance für size=128 auf sn07(12 Kerne).

		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 20: Performance für size=1024 auf sn02(8 Kerne). Ab hier macht Wavefront zumindest etwas sinvolles. Der lauf lief nur bis 10000 Iterationen und nicht bis zum Abbruchkriterium, weil sonst zu lange dauert(alles nicht nur wavefront).

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 Diagonalen 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 Diagonalen den Zeilen des Arrays zuweist umkehren um die Indexe 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 Diagonalen zu 1, weil sich ab hier die Abhängigkeit der Elemente um 1 verschiebt. Das selbe gilt für `hack2` ab der mittleren Diagonalen.
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.

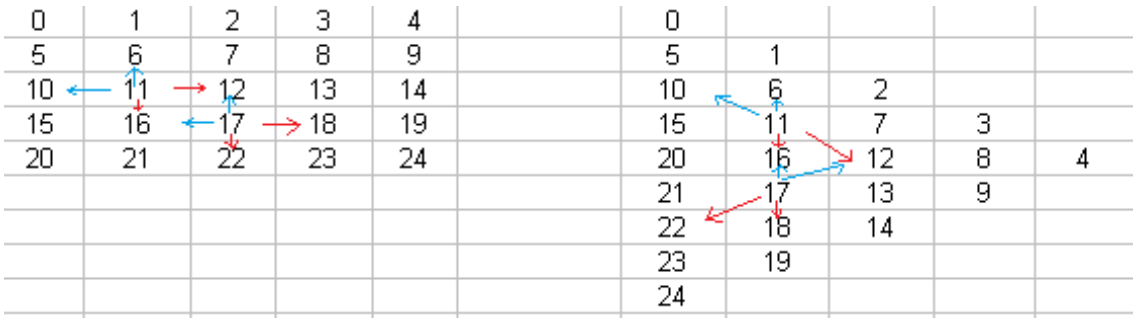

Abbildung 21: Veranschaulichung der Permutation, welche verwendet wird sowie der Verschiebung der Indizes anhand eines Beispiels. Die Variable `hack` ist dabei blau und `hack2` rot.

4 Methodenwahl

Es empfiehlt sich Gaus-Seidel, weil es im allgemeinen schneller konvergiert als Jacobi. Bis zu einer Problemgröße von 31 lohnt sich das parallelisieren nicht. Ab da macht die Rot-Schwarz SSE Implementierung am meisten Zeit gut.