

Projekt 1 – Jacobi- und Gauß-Seidel-Verfahren

Sarah Lutteropp und Johannes Sailer

Lehrstuhl für Rechnerarchitektur und Parallelverarbeitung



Aufgabenstellung

Mathematischer Hintergrund

Parallelisierung

Experimentelle Auswertung

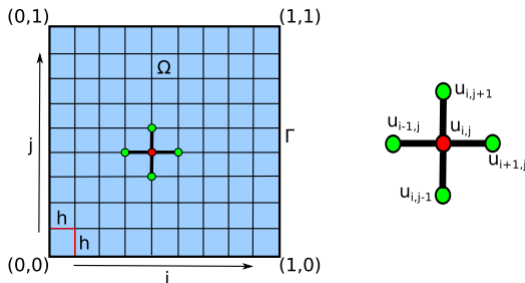
Fazit

Approximation von Stoffkonzentrationen

$$-\Delta u(x, y) = f(x, y) \quad \forall (x, y) \in (0, 1)^2 \quad (1)$$

$$u(x, y) = 0 \quad \forall (x, y) \in [0, 1]^2 \setminus (0, 1)^2 \quad (2)$$

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$



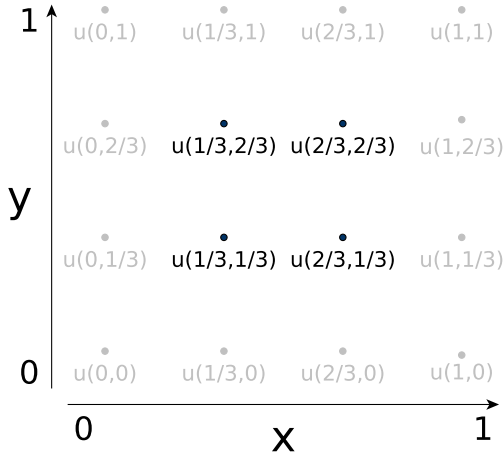
Approximation mit Methode der Finiten Differenzen

$$u_{i,j} = \frac{1}{4} * \left(u_{i,j-1} + u_{i-1,j} + u_{i,j+1} + u_{i+1,j} + h^2 f(x_i, y_j) \right)$$

$$-u_{i,j-1} - u_{i-1,j} + 4u_{i,j} - u_{i,j+1} - u_{i+1,j} = h^2 f(x_i, y_j)$$

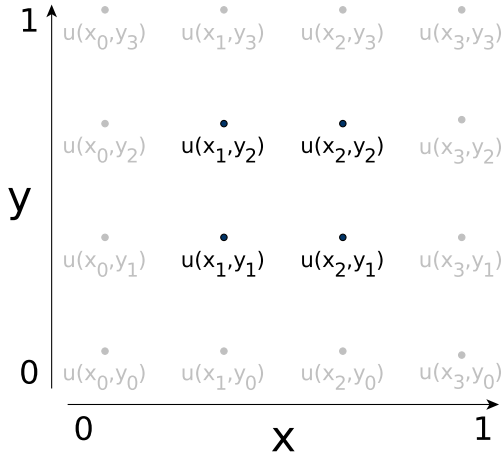
Konkretes Beispiel

Für $h = \frac{1}{3}$



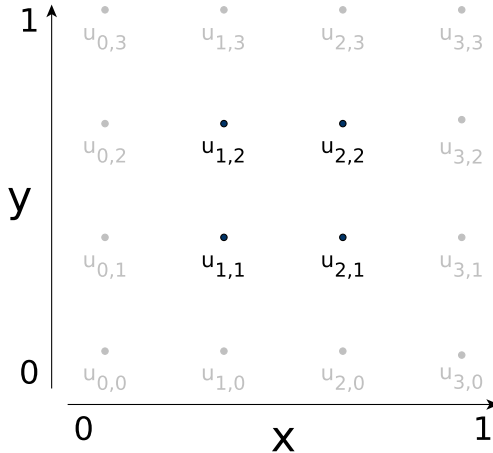
Konkretes Beispiel

Für $h = \frac{1}{3}$

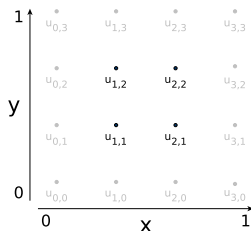


Konkretes Beispiel

Für $h = \frac{1}{3}$



Für $h = \frac{1}{3}$



$$-u_{i,j-1} - u_{i-1,j} + 4u_{i,j} - u_{i,j+1} - u_{i+1,j} = h^2 f(x_i, y_j)$$

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

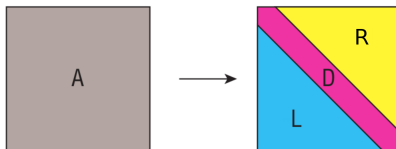
\Rightarrow Löse $Au = b$

Warum nicht einfach gaußen?

Das Gaußsche Eliminationsverfahren ist ...

- anfällig für Rechenfehler
- schlecht parallelisierbar
- langsam für viele Unbekannte
- schlechte Wahl bei dünn besetzter Matrix

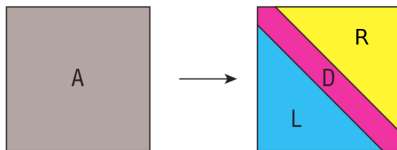
Im Folgenden: **Iterative Verfahren!**



$$Au = b$$

$$\Leftrightarrow (D + L + R)u = b \Leftrightarrow \dots$$

- Jacobi-Verfahren: $u^{(k)} = D^{-1} \left(b - (L + R)u^{(k-1)} \right)$
- Gauß-Seidel-Verfahren: $u^{(k)} = D^{-1} \left(b - Lu^{(k)} - Ru^{(k-1)} \right)$



■ Jacobi-Verfahren:

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

■ Gauß-Seidel-Verfahren:

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

Unsere Lösungsmatrix U

Für $h = \frac{1}{3}$

Betrachte statt

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

(Die Randeinträge sind hierbei 0)

Vorteil

- Jeder Eintrag in U entspricht einem Punkt im Gitter
- Parallelisierung intuitiver

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

Vorteile

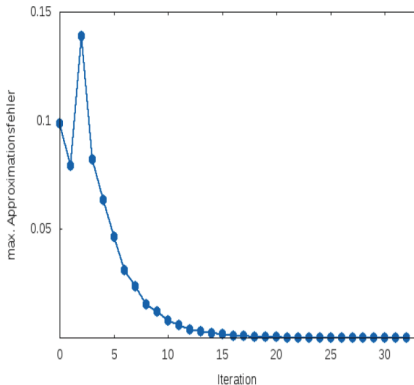
- Sprunglos
- Implementierung mit
`#pragma omp reduce`

Nachteile

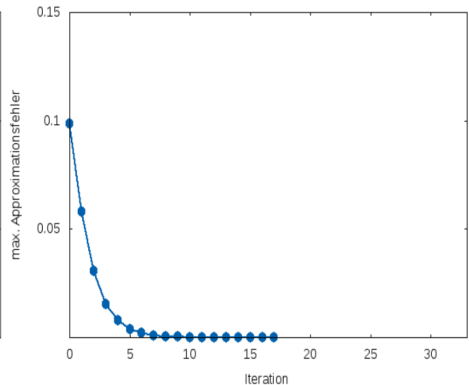
- Maximum der Differenzen
wäre exakter

Beide Verfahren konvergieren.

Jacobi-Verfahren für $h=1/4$



Gauß-Seidel-Verfahren für $h=1/4$



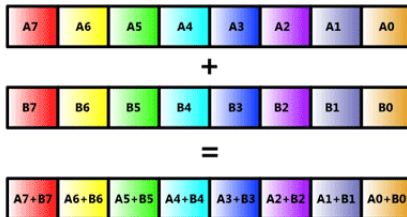
Keine Abhängigkeiten innerhalb einer Iteration

```
#pragma omp parallel for private(j, i) reduction(+:diff) collapse(2)
for (j = 1; j < size - 1; j++)
{
    for (i = 1; i < size - 1; i++)
    {
        a1[CO(i,j)] = a0[CO(i, j - 1)]
                    + a0[CO(i - 1, j)]
                    + a0[CO(i, j + 1)]
                    + a0[CO(i + 1, j)]
                    + functionTable[CO(i, j)];
        a1[CO(i,j)] *= 0.25;

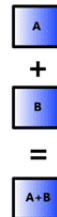
        diff += fabsf(a1[CO(i,j)] - a0[CO(i,j)]);
    }
}
```

Zusätzliche Optimierung: SSE-Vektorinstruktionen

SIMD Mode



Scalar Mode



- Abhängigkeiten innerhalb einer Iteration:

$$u_{i,j}^{(k+1)} = \frac{1}{4} u_{i,j-1}^{(k+1)} + u_{i-1,j}^{(k+1)} + u_{i,j+1}^{(k)} + u_{i+1,j}^{(k)} + h^2 f(x_i, y_j)$$

- 1. Möglichkeit: Wavefront

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 2 | 3 | 4 | 5 |
| 3 | 4 | 5 | 6 |
| 4 | 5 | 6 | 7 |

Nachteile Wavefront

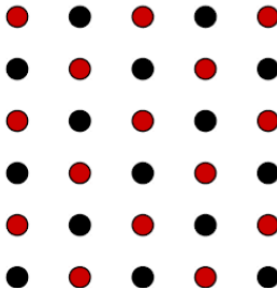
- Schlecht für Cache

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

- Aufwändige Berechnung der Indizes
- Geringe Parallelität bei kleinen Diagonalen
- Allgemein großer Overhead

Zweite Möglichkeit: Rot-Schwarz-Iteration

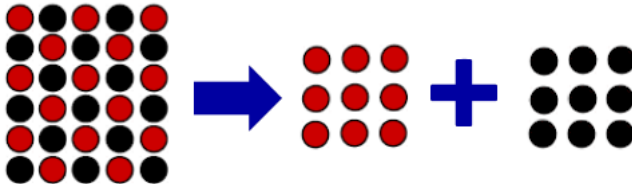
Färben der Matrixeinträge nach folgendem Schema:



1. Berechnen aller roten Einträge
2. Berechnen aller schwarzen Einträge

Optimierung: Rot-Matrix und Schwarz-Matrix separat

- Cache-Effizienz
- Sehr einfache Berechnung der Indizes
- Beschleunigung mit SSE-Vektorinstruktionen



Berechnung der Indizes, *size* ungerade für *size* = 7

$$U =$$

| | | | | | | |
|---|---|----|----|----|----|----|
| 0 | 3 | 7 | 10 | 14 | 17 | 21 |
| 0 | 4 | 7 | 11 | 14 | 18 | 21 |
| 1 | 4 | 8 | 11 | 15 | 18 | 22 |
| 1 | 5 | 8 | 12 | 15 | 19 | 22 |
| 2 | 5 | 9 | 12 | 16 | 19 | 23 |
| 2 | 6 | 9 | 13 | 16 | 20 | 23 |
| 3 | 6 | 10 | 13 | 17 | 20 | 24 |

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

Berechnung der Indizes, *size* gerade für *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 |

$$\begin{array}{llll} \leftarrow: idx - \frac{size}{2} & \rightarrow: idx + \frac{size}{2} & \uparrow: idx - (1 - (j \bmod 2)) & \downarrow: idx + (j \bmod 2) \\ \leftarrow: idx - \frac{size}{2} & \rightarrow: idx + \frac{size}{2} & \uparrow: idx - (j \bmod 2) & \downarrow: idx + (1 - (j \bmod 2)) \end{array}$$

Loop-Unrolling vermeidet Modulo-Operation!

Für *size* = 1024 mit 4 Threads

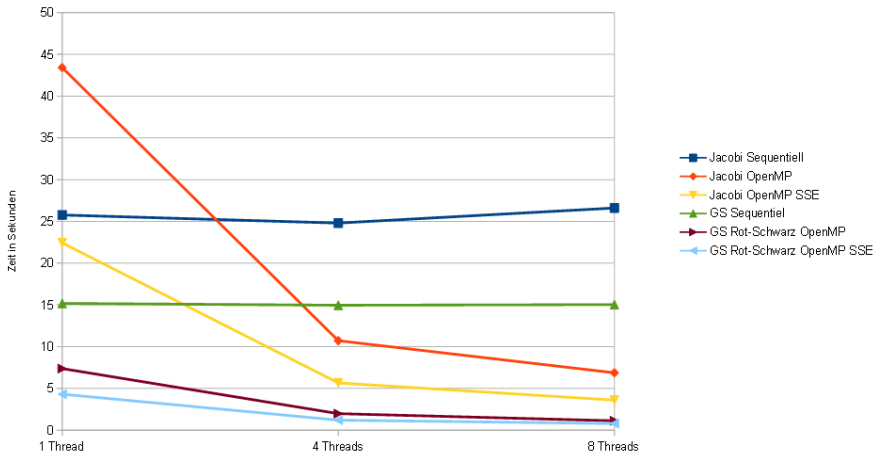
| | GCC | | ICC | |
|------------------------------|----------|---------|----------|---------|
| | Laufzeit | Speedup | Laufzeit | Speedup |
| Jacobi sequentiell | 70.5 s | - | 72 s | - |
| Jacobi | 19 s | 3.71 | 125 s | 0.58 |
| Jacobi SSE | 10.6 s | 6.65 | 9.4 s | 7.66 |
| Gauß-Seidel sequentiell | 135 s | - | 135 s | - |
| Gauß-Seidel naiv | 32.1 s | 4.21 | 125 s | 1.08 |
| Gauß-Seidel Rot-Schwarz | 16.3 s | 8.28 | 13 s | 10.38 |
| Gauß-Seidel Rot-Schwarz SSE | 13.4 s | 10.07 | 12 s | 11.25 |
| Gauß-Seidel Wave-front | 98 s | 1.38 | 116 s | 1.16 |
| Gauß-Seidel Wave-front Cache | 77 s | 1.75 | 94 s | 1.44 |

Für *size* = 128

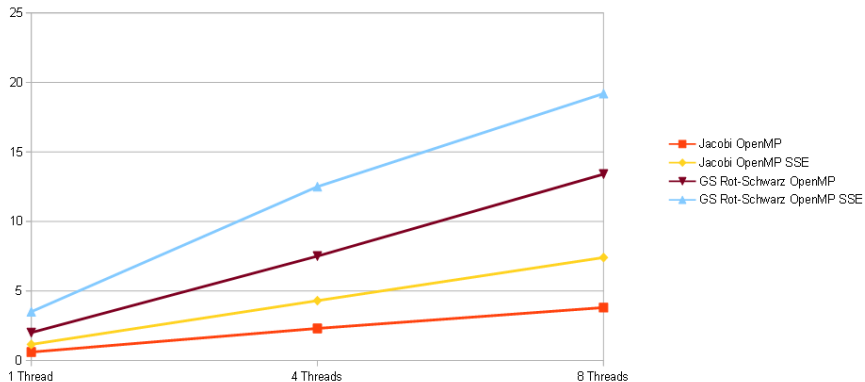
| | 1 Thread | | 4 Threads | | 8 Threads | |
|------------------------------|----------|---------|-----------|---------|-----------|---------|
| | Laufzeit | Speedup | Laufzeit | Speedup | Laufzeit | Speedup |
| Jacobi sequentiell | 2.15 s | - | 2.15 s | - | 2.15 s | - |
| Jacobi | 4 s | 0.538 | 1.1 s | 1.955 | 0.73 s | 2.945 |
| Jacobi SSE | 2.15 s | 1 | 0.67 s | 3.209 | 0.550 s | 3.909 |
| Gauß-Seidel sequentiell | 3.2 s | - | 3.2 s | - | 3.2 s | - |
| Gauß-Seidel naiv | 3.2 s | 1 | 0.9 s | 3.556 | 3.55 s | 5.818 |
| Gauß-Seidel Rot-Schwarz | 1.64 s | 1.951 | 0.5 s | 6.4 | 0.41 s | 7.805 |
| Gauß-Seidel Rot-Schwarz SSE | 1.354 s | 2.363 | 0.44 s | 7.273 | 0.4 s | 8 |
| Gauß-Seidel Wave-front | 2.6 s | 1.231 | 11.5 s | 0.278 | 22 s | 0.145 |
| Gauß-Seidel Wave-front Cache | 2.47 s | 1.296 | 12.5 s | 0.256 | 27 s | 0.119 |

Auswertung mit Abbruchkriterium

Laufzeiten für size = 257



Speedup für size = 257



Jacobi-Verfahren für $size = 257$

| | 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 |

Gauß-Seidel-Verfahren für $size = 257$

| | 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 |

- Jacobi leichter zu parallelisieren
- Gauß-Seidel konvergiert doppelt so schnell
- Rot-Schwarz-Iteration liefert den besten Speedup
- Vektorisierung lohnt sich