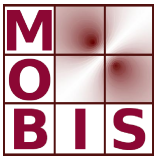
SpezialForschungsBereich F 32

Karl–Franzens Universität Graz
Technische Universität Graz
Medizinische Universität Graz



Parallele Matrixassemblierung auf Manycore-Prozessoren

M. Hrassnigg

SFB-Report No. 2015-021 July 2014

A–8010 GRAZ, HEINRICHSTRASSE 36, AUSTRIA

Supported by the
Austrian Science Fund (FWF)

SFB sponsors:

- **Austrian Science Fund** (FWF)
- **University of Graz**
- **Graz University of Technology**
- **Medical University of Graz**
- **Government of Styria**
- **City of Graz**

Parallele Matrixassemblierung auf Manycore-Prozessoren

Miriam Hrassnigg

23. Juli 2014

Inhaltsverzeichnis

1 Einleitung

Bevor wir uns mit der Umsetzung der Matrixassemblierung beschäftigen, wollen wir uns einen kurzen Überblick von der Methode der finiten Elemente (kurz FEM) verschaffen. Die FEM ist ein numerisches Verfahren zur Lösung von Randwertproblemen. Ein eindimensionales Beispiel dafür wäre folgendes Problem:

Klassische Formulierung

Gesucht ist $u \in C^2(a,b) \cap C^1(a,b] \cap C[a,b]$, *so dass*

$$\begin{aligned} -u''(x) + cu(x) &= f(x) && \forall x \in \Omega = (a,b), c \geq 0 \\ u(a) &= g_a && a = \Gamma_1 \\ -u'(b) &= \alpha_b(u(b) - u_b) \end{aligned}$$

gilt. Wobei c, u_b *und* α_b *vorgegebene Werte sind und* f *eine gegebene Funktion ist.*

Bei der FE-Methode geht man allerdings nicht von der klassischen Formulierung aus, sondern von der Variationsformulierung. Für unser Beispiel kann man sie wie folgt herleiten:

1. Man definiert zuerst einen Raum V_0 der Testfunktionen v durch:

$$V_0 = \{v \in H^1(\Omega) : v = 0 \text{ auf } \Gamma_1\}$$

Dabei ist $H^1(\Omega) = \{u \in L_2(a,b)^1 : \exists$ verallgemeinerte Ableitung ^{2} $u' \in L_2(a,b)\}$

Als Testfunktionen wählt man bewusst Funktionen, die auf dem Randstück mit Randbedingungen 1.Art gleich 0 sind.

2. Dann multipliziert man die Differentialgleichung mit einer Testfunktion v aus dem Raum V_0 und integriert das ganze über Ω. Für unser obiges Beispiel bedeutet das:

$$\int_a^b [-u''(x)v(x) + cu(x)v(x)]dx = \int_a^b f(x)v(x)dx$$

---

[1] $L_2(a,b) = \{u : \exists \int_a^b (u(x))^2 < \infty\}$

[2] verallgemeinerte Ableitung: Sind u und w integrierbare Funktionen und gilt:
$\int_a^b u(x)\phi'(x)dx = -\int_a^b w(x)\phi(x)dx$
für alle Funktionen $\phi \in C^1[a,b]$ mit $\phi(a) = \phi(b) = 0$, dann heißt w *verallgemeinerte Ableitung von* u *nach* x

3. Nun wendet man die Formel für die partielle Integration auf den Hauptteil (den Term mit der 2. Ableitung) an. Das liefert

$$-\int_a^b u''(x)v(x)dx = \int_a^b u'(x)v'(x)dx - u'(x)v(x)\Big|_a^b$$

4. Baut man jetzt noch die Randbedingungen ($-u'(b) = \alpha_b(u(b) - u_b)$ und $v(a) = 0$) ein, dann ergibt sich

$$-\int_a^b u''(x)v(x)dx = \int_a^b u'(x)v'(x)dx + \alpha_b u(b)v(b) - \alpha_b u_b v(b)$$

Damit können wir die ***Variationsformulierung*** des Beispiels angeben

Gesucht ist $u \in V_g = \{u \in H^1(a,b) : u(a) = g_a\}$, *so dass*

$$a(u,v) = \langle F, v\rangle \qquad \forall v \in V_0 = \{v \in H^1(a,b) : v(a) = 0\}$$

gilt, wobei

$$a(u,v) = \int_a^b [u'(x)v'(x) + cu(x)v(x)]dx + \alpha_b u(b)v(b)$$

$$\langle F, v\rangle = \int_a^b f(x)v(x)dx + \alpha_b u_b v(b)$$

An dieser Stelle wäre zu bemerken, dass es sich bei der Funktion $a(u,v)$ um eine symmetrische und positiv definite Bilinearform auf $V = H^1(\Omega)$ handelt und die Funktion $\langle F, v\rangle$ eine Linearform ist. Es gilt also:
$a(\nu_1 u_1 + \nu_2 u_2, v) = \nu_1 a(u_1, v) + \nu_2 a(u_2, v) \qquad \forall u_1, u_2 \in V, \quad \forall \nu_1, \nu_2 \in \mathbb{R}$
$a(u, \nu_1 v_1 + \nu_2 v_2) = \nu_1 a(u, v_1) + \nu_2 a(u, v_2) \qquad \forall u \in V, \quad \forall v_1, v_2 \in V, \quad \forall \nu_1, \nu_2 \in \mathbb{R}$
$a(v,v) > 0 \qquad \forall v \in V_0, v \neq 0$
und
$\langle F, \nu_1 v_1 + \nu_2 v_2\rangle = \nu_1\langle F, v_1\rangle + \nu_2\langle F, v_2\rangle \qquad \forall v_1, v_2 \in V, \forall \nu_1, \nu_2 \in \mathbb{R}$

1.1 Galerkin- und Ritz-Verfahren

Wir betrachten die Herleitung des Galerkin-Ritz-Verfahrens ausgehend von der Variationsformulierung. Die Idee ist, dass man die ∞-dimensionalen Mengen

$V_g = \{u \in H^1(\Omega) : u(x) = g_1(x) \text{ auf } \Gamma_1\}, V_0 = \{v \in H^1(\Omega) : v(x) = 0 \text{ auf } \Gamma_1\}$ und $V = H^1(\Omega)$ durch folgende endlichdimensionale Mengen ersetzt.

$$V_h = \{v_h : v_h(x) = \sum_{i \in \bar{w}_h} v_i p_i(x)\} = span\{p_i : i \in \bar{w}_h\} \subset V$$
$$V_{0h} = \{v_h : v_h(x) = \sum_{i \in w_h} v_i p_i(x)\} = span\{p_i : i \in w_h\}$$
$$V_{gh} = \{v_h : v_h(x) = \sum_{j \in w_h} v_j p_j(x) + \sum_{j \in \gamma_h} \tilde{u}_j p_j(x)\}$$

Die Funktionen p_i werden Ansatzfunktionen genannt, sie sind vorgegebene, linear unabhängige Funktionen. Die Koeffizienten v_i können frei gewählt werden und mit $\bar{w}_h, w_h$ und γ_h bezeichnen wir endliche Indexmengen. Für diese soll gelten: $\bar{w}_h = w_h \cup \gamma_h$. Außerdem wird gefordert, dass

$$\sum_{j \in \gamma_h} \tilde{u}_j p_j(x) \approx g_1(x) \qquad \forall x \in \Gamma_1$$

Gesucht wird nun also eine *Näherungslösung* der Variationsformulierung in der Form

$$u_h(x) = \sum_{j \in w_h} u_j p_j(x) + \sum_{j \in \gamma_h} \tilde{u}_j p_j(x) \tag{1}$$

Für die gilt:

$$a(u_h, v_h) = \langle F, v_h \rangle \qquad \forall v_h \in V_{0h}$$

Um die Näherungslösung zu bestimmen muss man die Koeffizienten $u_j,\ j \in w_h$ berechnen. Setzt man einen Lösungsansatz u_h wie in (1) in $a(u_h, v_h) = \langle F, v_h \rangle$ ein und wählt eine der Ansatzfunktionen $p_i,\ i \in w_h$ als v_h (d.h. $v_h = p_i$), dann erhält man:

$$\sum_{j \in w_h} u_j a(p_j, p_i) = \langle F, p_i \rangle - \sum_{j \in \gamma_h} \tilde{u}_j a(p_j, p_i)$$

Daraus ergibt sich das ***Galerkin-Gleichungssystem***

Gesucht ist $\underline{u}_h = [u_j]_{j \in w_h} \in \mathbb{R}^{N_h}$, *so dass*

$$K_h \underline{u}_h = \underline{f}_h$$

gilt, mit der sogenannten Steifigkeitsmatrix

$$K_h = [a(p_j, p_i)]_{i,j \in w_h}$$

und dem Lastvektor

$$\underline{f}_h = [\langle F, p_i \rangle - \sum_{j \in \gamma_h} \tilde{u}_j a(p_j, p_i)]_{i \in w_h}$$

Bemerkung 1. *Das Ritz-Verfahren geht von einem Minimierungsproblem aus. Gesucht wird das Minimum der Funktion* $J(w) = \frac{1}{2}a(w,w) - \langle F, w\rangle$. *Über die notwendige Bedingung 1.Ordnung für Minimalstellen, kommt man auf das Ritzsche-Gleichungssystem. Wenn die Funktion* $a(.,.)$ *bilinear, positiv und symmetrisch ist, dann ist das Ritz-System gleich dem Galerkin-System. Daher kommt auch der Name Ritz-Galerkin-System.*

1.2 Gebietsdiskretisierung und Beschreibung einer Vernetzung

Das Ziel ist es nun also das Ritz-Galerkin-Gleichungssystem zu lösen. Natürlich möchte man sich die Arbeit so einfach wie möglich machen. Die Frage ist also wie soll man die Ansatzfunktionen wählen, um eine möglichst leicht zu handhabende Matrix K_h zu bekommen. Eine Idee dafür ist Ansatzfunktionen mit lokalem Träger zu verwenden.
Also unterteilt man das Gebiet Ω auf dem man das Randwertproblem untersucht in "kleine", sich nicht überlappende Teilgebiete $T^{(r)}$. Diese Teilgebiete nennt man finite Elemente. Man wählt dafür z.B. Intervalle in eindimensionalen Gebieten, Dreiecke oder Vierecke in 2-dimensionalen und Tetraeder oder Hexaeder etc. in 3-dimensionalen Gebieten.

Für die Zerlegung fordert man folgende Eigenschaften:

1. $$\bar{\Omega} = \bigcup_{r=1}^{R_h} \bar{T}^{(r)} \quad \text{bzw.} \quad \bar{\Omega}_h = \bigcup_{r=1}^{R_h} \bar{T}^{(r)} \;\to\; \bar{\Omega} \quad \text{für } h \to 0$$

   Was bedeutet, dass die Vereinigung der $T^{(r)}$ das Gebiet Ω entweder überdeckt, oder zumindest für immer kleiner werdene Elemente approximiert.

2. Für alle $r, r' = 1, 2, ...R_h$ mit $r \neq r'$ ist

   $$\bar{T}^{(r)} \cap \bar{T}^{(r')} = \begin{cases} \emptyset \text{ oder} \\ \text{ein gemeinsamer Eckknoten oder} \\ \text{eine gemeinsame Kante oder} \\ \text{eine gemeinsame Fläche (im 3-dimensionalen)} \end{cases}$$

   Eine solche Zerlegung des Gebiets Ω nennt man Vernetzung. Die Eckpunkte der finiten Elemente werden als Knoten bezeichnet.

Für das Randwertproblem vom Anfang könnte man das Gebiet $[a, b]$ etwa in N gleichgroße Intervalle unterteilen.

Wie beschreibt man nun eine solche Vernetzung? Zuerst werden jeweils die finiten Elemente und die Knoten gesondert durchnummeriert. Danach weist man den Knoten der einzelnen Elemente noch eine lokale Knotennummer zu.

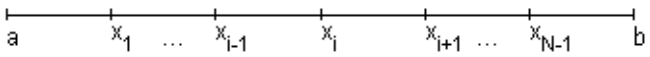


Abbildung 1: Unterteilung des Gebiets und Nummerierung der Knoten

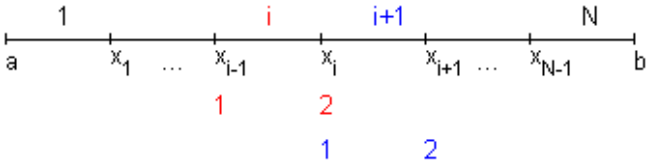


Abbildung 2: Nummerierung der Elemente und der lokalen Knoten

D.h. jeder Knoten hat eine globale Knotennummer und eine lokale Knotennummer. Die Verknüpfung von lokaler zu globaler Knotennummer wird in einer Elementzusammenhangstabelle angegeben. In dieser Tabelle stehen in der i-ten Zeile jene globalen Knotennummern, die den lokalen Knotennummern des i-ten Elements entsprechen. (Siehe Abbildung 2)
Würde man die Elemente (Intervalle) in Abbildung(1) einfach von links durchnummerieren. Bekäme man folgende Elementzusammenhangstabelle:

Elementnummer	lokale KnotenN. 1	lokale KnotenN. 2
0	0	1
1	1	2
⋮	⋮	⋮
i	i	i+1
⋮	⋮	⋮
N-1	N-1	N

Als Ansatzfunktionen werden wir grundsätzlich stetige, stückweise polynomiale Funktionen verwenden, die nur über ganz wenigen dieser Teilgebiete von Null verschieden sind. Wir sprechen hier also von Funktionen mit lokalem Träger.
In unserem Beispiel könnte man für $i \in \{1, ..., N-1\}$ etwa die Funktionen

$$p_i(x) = \begin{cases} \dfrac{x - x_{i-1}}{N} & \text{falls } x \in [x_{i-1}, x_i] \\ \dfrac{x_{i+1} - x}{N} & \text{falls } x \in (x_i, x_{i+1}] \\ 0 & \text{sonst} \end{cases}$$

und für $i = 0$ bzw. $i = N$

$$p_0(x) = \begin{cases} \dfrac{x-a}{N} & \text{falls } x \in [a, x_1] \\ 0 & \text{sonst} \end{cases}$$

$$p_N(x) = \begin{cases} \dfrac{b-x}{N} & \text{falls } x \in [x_{N-1}, b] \\ 0 & \text{sonst} \end{cases}$$

verwenden.

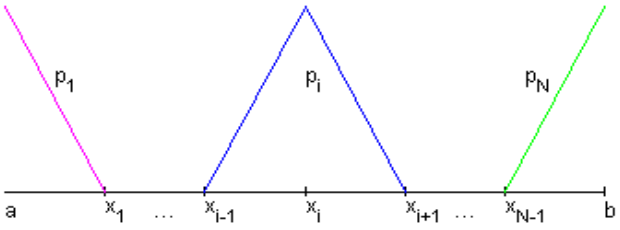


Abbildung 3: Ansatzfunktionen

Was sind jetzt aber die Vorteile von derart gewählten Ansatzfunktionen?

1. Die Steifigkeitsmatrix ist schwach besetzt.

$$\text{Falls } int(supp(p_i)) \cup int(supp(p_j)) = \emptyset \;\Rightarrow\; K_{ij} = a(p_j, p_i) = 0$$

2. Elementweise Berechnung der Steifigkeitsmatrix ist möglich. Die Eigenschaften

$$\bar{\Omega} = \bar{\Omega}_h = \bigcup_r \bar{T}^{(r)} \text{ und } T^{(r)} \cap T^{(r')} = \emptyset, \qquad \text{für } r \neq r'$$

ermöglichen es uns ein Integral über Ω wie folgt aufzuteilen:

$$\int_\Omega (...)dx = \sum_r \int_{T^{(r)}} (...)dx$$

Bemerkung 2. *Verwendet man komplexere finite Elemente, als nur Intervalle, dann definiert man zuerst eine Ansatzfunktion über einem Referenzelement. Um für ein beliebiges finites Element die passende Ansatzfunktion zu erhalten, führt man eine Koordinatentransformation auf das Referenzelement durch.*

1.3 Aufbau des Gleichungssystems

Im folgenden Kapitel beschäftigen wir uns mit dem Aufbau des FE-Gleichungssystems (d.h. des Ritz-Galerkin-Systems). Zur Erinnerung noch einmal die Aufgabenstellung:

Gesucht ist $\underline{u}_h \in \mathbb{R}^{N_h}$, so dass $K_h \underline{u}_h = \underline{f}_h$,
wobei $K_h = [a(p_j, p_i)]_{i,j \in w_h}$ und $\underline{f}_h = [\langle F, p_i \rangle - \sum_{j \in \gamma_h} \tilde{u}_j a(p_j, p_i)]_{i \in w_h}$.

In unserem Beispiel ergibt sich direkt aus der Variationsformulierung:

$$a(u,v) = \int_a^b [u'(x)v'(x) + cu(x)v(x)]dx + \alpha_b u(b)v(b)$$

$$\langle F, v \rangle = \int_a^b f(x)v(x)dx + \alpha_b u_b v(b)$$

Bevor man das vollständige, komplexe Problem behandelt, erzeugt man im ersten Schritt einmal die Steifigkeitsmatrix ohne die Randbedingungen zu berücksichtigen. Diese werden erst im Nachhinein eingebaut.

D.h. $a(u,v)$ wird in der Berechnung erstmal durch

$$\bar{a}(u,v) = \int_a^b [u'(x)v'(x) + cu(x)v(x)]dx$$

ersetzt und statt $\langle F, v \rangle$ verwendet man

$$\langle \bar{F}, v \rangle = \int_a^b f(x)v(x)dx.$$

Um das Gleichungssystem aufzubauen werden zuerst für jedes Element sogenannte Elementsteifigkeitsmatrizen und Elementlastvektoren erzeugt und dann in einem weiteren Schritt zu einer großen Steifigkeitsmatrix bzw. Lastvektor "assembliert". Dass diese elementweise Berechnung überhaupt möglich ist, folgt aus der Wahl der finiten Elemente.

1.3.1 Aufbau der Elementsteifigkeitsmatrizen und der Elementlastvektoren

Die Berechnungen in diesem Kapitel stützen sich im wesentlichen auf folgende Eigenschaften der Vernetzung:

$$\bar{\Omega} = \bigcup_{r=1}^{R_h} \bar{T}^{(r)}, \; T^{(r)} \cap T^{(r')} = \emptyset \text{ für } r, r' = 1, 2, \ldots, R_h, \quad r \neq r'$$

Da die Berechnungen der Elementsteifigkeitsmatrizen sehr problemabhängig sind, werden wir dieses Kapitel stellvertretend für unser Beispiel durchrechnen.
In unserem Beispiel ist

$$\Omega = [a,b] \text{ und } (x_{i-1}, x_i) \cap (x_{j-1}, x_j) = \emptyset \text{ falls } i \neq j.$$

Für beliebige Funktionen $u_h, v_h \in V_h = \{v_h : v_h(x) = \sum_{k=1}^{\bar{N}_h} v_k p_k(x)\}$ gilt

$$\bar{a}(u_h, v_h) = \bar{a}(\sum_{j \in \bar{w}_h} u_j p_j(x), \sum_{k \in \bar{w}_h} v_k p_k(x))$$

Für $\bar{a}(u,v)$ aus dem Beispiel ergibt sich:

$$\begin{aligned}
\bar{a}(u_h, v_h) &= \int_a^b \Big(\sum_{j=1}^N u_j p_j(x)\Big)' \Big(\sum_{k=1}^N v_k p_k(x)\Big)' + c\Big(\sum_{j=1}^N u_j p_j(x)\Big)\Big(\sum_{k=1}^N v_k p_k(x)\Big) dx \\
&= \sum_{i=1}^N \int_{x_{i-1}}^{x_i} \Big(\sum_{j=1}^N u_j p_j(x)\Big)' \Big(\sum_{k=1}^N v_k p_k(x)\Big)' + c\Big(\sum_{j=1}^N u_j p_j(x)\Big)\Big(\sum_{k=1}^N v_k p_k(x)\Big) dx
\end{aligned}$$

Nun wurden aber die p_j so gewählt, dass $p_j(x)$ nur dann $\neq 0$ ist, falls $x \in (x_{j-1}, x_{j+1})$. Das bedeutet auf dem Intervall (x_{i-1}, x_i) nehmen nur die Funktionen p_{i-1} und p_i Werte $\neq 0$ an. Also verkürzt sich der Ausdruck zu:

$$\begin{aligned}
&= \sum_{i=1}^N \int_{x_{i-1}}^{x_i} \big[u_{i-1} p'_{i-1}(x) + u_i p'_i(x)\big]\big[v_{i-1} p'_{i-1}(x) + v_i p'_i(x)\big] \\
&\qquad + c\big[u_{i-1} p_{i-1}(x) + u_i p_i(x)\big]\big[v_{i-1} p_{i-1}(x) + v_i p_i(x)\big] dx \\
&= \sum_{i=1}^N \begin{pmatrix} v_{i-1} & v_i \end{pmatrix} K^{(i)} \begin{pmatrix} u_{i-1} \\ u_i \end{pmatrix}
\end{aligned}$$

Die Matrizen $K^{(i)} = \begin{pmatrix} K_{11}^{(i)} & K_{12}^{(i)} \\ K_{21}^{(i)} & K_{22}^{(i)} \end{pmatrix}$ sind die sogenannten *Elementsteifigkeitsmatrizen.*
Die Einträge für $K^{(i)}$ lassen sich aus der oberen Gleichung ablesen. So ergibt sich etwa für $K_{11}^{(i)}$:

$$K_{11}^{(i)} = \int_{x_{i-1}}^{x_i} \big[p'_{i-1}(x) p'_{i-1}(x) + c p_{i-1}(x) p_{i-1}(x)\big] dx$$

Auf die gleiche Art und Weise geht man auch bei der rechten Seite des Problems vor.

$$\begin{aligned}\langle \bar{F}, v_h \rangle &= \sum_{i=1}^{N} \int_{x_{i-1}}^{x_i} f(x) v_h(x) dx \\ &= \sum_{i=1}^{N} \int_{x_{i-1}}^{x_i} f(x) \big[v_{i-1} p_{i-1}(x) + v_i p_i(x) \big] dx \\ &= \sum_{i=1}^{N} (v_{i-1} \quad v_i) \underline{f}^{(i)}\end{aligned}$$

Dabei ist $\underline{f}^{(i)} = \begin{pmatrix} \int_{x_{i-1}}^{x_i} f(x) p_{i-1}(x) dx \\ \int_{x_{i-1}}^{x_i} f(x) p_i(x) dx \end{pmatrix}$ der sogenannte Elementlastvektor.

1.3.2 Assemblierung der Steifigkeitsmatrix

Nun müssen die Elementsteifigkeitsmatrizen und die Elementlastvektoren noch zur Steifigkeitsmatrix $\underline{\bar{K}}_h$ und zum Lastvektor $\underline{\bar{f}}_h$ zusammengebaut werden. Diesen Vorgang nennt man *Assemblierung*. Dazu wird für jedes Element $T^{(i)}$ der Zusammenhang zwischen lokaler Knotennummerierung und globaler Knotennummerierung benötigt. Wie bereits zuvor beschrieben, ist diese Information in der Elementzusammenhangstabelle gespeichert.
Wir können also für jedes Element sogenannte Elementzusammenhangsmatrizen $C^{(r)}$ erzeugen mit

$$C^{(r)}_{\alpha,j} = \begin{cases} 1 & \text{falls der Knoten mit der globalen Knotennummer } j \text{ die lokale Knotennummer } \alpha \text{ hat.} \\ 0 & \text{sonst.} \end{cases}$$

Für unser Beispiel hat $C^{(i)}$ die Form:

$$C^{(i)} = \underbrace{\begin{pmatrix} 0 & \dots & 1 & 0 & 0 & \dots & 0 \\ 0 & \dots & 0 & 1 & 0 & \dots & 0 \end{pmatrix}}_{C^{(i)}_{1,i-1}=1 \text{ und } C^{(i)}_{2,i}=1}$$

Es gilt also für die Vektoren $\underline{\bar{u}}_h = (u_0, u_1, \dots, u_n)^T$ und $\underline{\bar{v}}_h = (v_1, v_2, \dots, v_n)^T$

$$\begin{pmatrix} u_{i-1} \\ u_i \end{pmatrix} = C^{(i)} \underline{\bar{u}}_h \text{ und } (v_{i-1} \quad v_i) = \big(C^{(i)} \underline{\bar{v}}_h \big)^T$$

Daraus ergibt sich folgende Beziehung:

$$\bar{a}(u_h, v_h) = \sum_{i=1}^{N} \begin{pmatrix} v_{i-1} & v_i \end{pmatrix} K^{(i)} = \sum_{i=1}^{N} \underline{\bar{v}}_h^T (C^{(i)})^T K^{(i)} C^{(i)} \underline{\bar{u}}_h$$
$$= \underline{\bar{v}}_h^T \Big(\underbrace{\sum_{i=1}^{N} (C^{(i)})^T K^{(i)} C^{(i)}}_{:=\bar{K}_h} \Big) \underline{\bar{u}}_h = \underline{\bar{v}}_h^T \bar{K}_h \underline{\bar{u}}_h$$

Auf dieselbe Weise wird nun natürlich auch mit dem Lastvektor verfahren.

$$\langle \bar{F}, v_h(x) \rangle = \sum_{i=1}^{N} \begin{pmatrix} v_{i-1} & v_i \end{pmatrix} \underline{f}^{(i)} = \sum_{i=1}^{N} \underline{\bar{v}}_h^T (C^{(i)})^T \underline{f}^{(i)} = \underline{\bar{v}}_h^T \underbrace{\sum_{i=1}^{N} (C^{(i)})^T \underline{f}^{(i)}}_{:=\underline{\bar{f}}_h}$$
$$= \underline{\bar{v}}_h^T \underline{\bar{f}}_h$$

Für die Assemblierung der Steifigkeitsmatrix $\bar{K}_h = \sum_{i=1}^{N} (C^{(i)})^T K^{(i)} C^{(i)}$ und des Lastvektors $\underline{\bar{f}}_h = \sum_{i=1}^{N} (C^{(i)})^T \underline{f}^{(i)}$ werden nun aber nicht diese Matrixmultiplikationen berechnet und dann zusammenaddiert, sondern es wird folgender Algorithmus ausgeführt.

Zuerst setzt man $\bar{K}_h = [K_{k,p}]_{k,p=1}^{\bar{N}_h}$ und $\underline{\bar{f}}_h = [f_k]_{k=1}^{\bar{N}_h}$ gleich 0. Dann werden für jedes Element die Elementsteifigkeitsmatrizen $K^{(i)}$ und die Elementlastvektoren $\underline{f}^{(i)}$ eingebaut.

Algorithm 1 Assemblierungsalgorithmus

```
Setze K̄_h = 0 und f̲̄_h = 0
for i = 0 bis N do
  Berechne K^(i) und f̲^(i)
  for α = 1 bis N̂ do
    Bestimme globale Knotennummer k von lokalem Knoten α in Element i
    Setze f_k := f_k + f_α^(i)
    for β = 1 bis N̂ do
      Bestimme globale Knotennummer p von lokalem Knoten β in Element i
      Setze K_{k,p} := K_{k,p} + K_{α,β}^(i)
    end for
  end for
end for
```

In unserem Beispiel haben die Matrizen $(C^{(i)})^T K^i C^{(i)}$ genau 4 Einträge ungleich 0. Das sind die Einträge $K_{11}^{(i)}, K_{12}^{(i)}, K_{21}^{(i)}$ und $K_{22}^{(i)}$ der Elementsteifigkeitsmatrix $K^{(i)}$. Aus der

Elementzusammenhangstabelle erfahren wir nun an welcher Stelle in der Steifigkeitsmatrix die Einträge aus der Elementsteifigkeitsmatrix stehen.

Im Element 1 hat der lokale Knoten Nummer 1 die globale Nummer 0 und der lokale Knoten Nummer 2 die globale Nummer 1. Es ist also

$$f_{neu} = \begin{pmatrix} f_1^{(1)} \\ f_2^{(1)} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad \text{und} \quad K_{neu} = \begin{pmatrix} K_{11}^{(1)} & K_{12}^{(1)} & 0 & \dots & 0 \\ K_{21}^{(1)} & K_{22}^{(1)} & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Führt man den gleichen Prozess für das 2. Element durch erhält man

$$f_{neu} = \begin{pmatrix} f_1^{(1)} \\ f_2^{(1)} + f_1^{(2)} \\ f_2^{(2)} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad \text{und} \quad K_{neu} = \begin{pmatrix} K_{11}^{(1)} & K_{12}^{(1)} & 0 & \dots & 0 \\ K_{21}^{(1)} & K_{22}^{(1)} + K_{11}^{(2)} & K_{12}^{(2)} & \dots & 0 \\ 0 & K_{21}^{(2)} & K_{22}^{(2)} & 0 \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Wird der Algorithmus bis zu Ende geführt, erhält man schließlich eine Matrix und einen Vektor mit folgender Struktur:

$$\underline{\bar{f}}_h = \begin{pmatrix} f_1^{(1)} \\ f_2^{(1)} + f_1^{(2)} \\ f_2^{(2)} \\ 0 \\ \vdots \\ 0 \\ f_2^{(n-1)} + f_1^{(n)} \\ f_2^{(n)} \end{pmatrix} \quad \text{und} \quad \bar{K}_h = \begin{pmatrix} K_{11}^{(1)} & K_{12}^{(1)} & 0 & 0 & \dots & 0 \\ K_{21}^{(1)} & K_{22}^{(1)} + K_{11}^{(2)} & K_{12}^{(2)} & 0 & \dots & 0 \\ 0 & K_{21}^{(2)} & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & K_{12}^{(n-1)} & 0 \\ 0 & \dots & 0 & K_{21}^{(n-1)} & K_{22}^{(n-1)} + K_{11}^{(n)} & K_{12}^{(n)} \\ 0 & \dots & 0 & 0 & K_{21}^{(n)} & K_{22}^{(n)} \end{pmatrix}$$

Auf diese Art und Weise assembliert man die Steifigkeitsmatrix für dieses Beispiel. Den Einbau der Randbedingungen werden wir nicht mehr durchrechnen. Das kann im Buch [10] nachgelesen werden. Ab jetzt beschäftigen wir uns damit, wie die Berechnung der Elementsteifigkeitsmatrizen und die Matrixassemblierung am geschicktesten zu parallelisieren ist.

2 Parallelisierung und data races

Wir kennen nun also einen Algorithmus zur Erzeugung der Steifigkeitsmatrix. Dabei dürfen wir aber nicht vergessen, dass die Finite Elemente Methode ein Werkzeug zu Lösung von sehr komplexen Problemen ist. Es ist also nicht unüblich, dass man dieses Gebiet auf dem man eine Lösung sucht, durch Millionen von Elementen approximieren muss. Das bedeutet aber, dass der Computer die Assemblierung der Elementsteifigkeitsmatrizen ebenfalls Millionen mal durchführen muss. Und das kostet Zeit. Es stellt sich also die Frage, wie man den Prozess der Assemblierung beschleunigen kann, ohne Berechnungen verändern zu müssen.

Betrachtet man die Formel für die Elementsteifigkeitsmatrizen und den Assemblierungsalgorithmus noch einmal, dann fällt auf, dass alle Berechnungen, die für ein einzelnes Element durchgeführt werden, keinerlei Informationen von anderen Elementen benötigen. Man könnte die Elementsteifigkeitsmatrizen für verschiedene Elemente also auch gleichzeitig berechnen.
In der Praxis setzt man diese Information so um, dass man dieses eine Codestück, in dem für ein Element die Elementsteifigkeitsmatrix berechnet und in die Steifigkeitsmatrix geschrieben wird, parallelisiert. Man führt diese Berechnungen also für unterschiedliche Elemente parallel auf mehreren Prozessorkernen durch.
Betrachten wir das ein wenig genauer. Welche Informationen und welche Resourcen sind für alle beteiligten Prozessorkerne zugänglich, welche nicht und welche sollten es nicht sein?

2.1 Parallelität auf Prozessorebene

Seit einigen Jahren versucht man die Leistung der Prozessorchips zu steigern, in dem man mehrere Prozessorkerne auf einem einzelnen Prozessorchip integriert. Dabei wird jeder dieser Prozessorkerne vom Betriebssystem als separater logischer Prozessor mit eigenen Ausführungsresourcen wahrgenommen. Solche Prozessoren mit mehreren Prozessorkernen werden Multicore-Prozessoren genannt. Über diese Prozessor-Architektur erhält man die Möglichkeit verschiedene, voneinander unabhängige Berechnungen eines Programms parallel auszuführen.

Betrachten wir stellvertretend für die verschiedenen Prozessor-Architekturmodelle, das sogenannte Hierarchische Design. Bei dieser Prozessorarchitektur hat jeder der Prozessorkerne einen eigenen Level1-Cache. Ein weiterer Level2-Cache wird von mehreren Prozessorkernen genutzt und auf einen externen Hauptspeicher können alle Prozessorkerne zugreifen. Der Grund für diese hierarchische Verteilung von Speicher ist, dass das Laden bzw. Verschieben von Daten viel Zeit braucht und je weiter die Daten vom Prozessor

entfert sind, desto teurer wird es.

Diese Architektur wurde unter anderem für die Intel Core 2 Duo Prozessoren verwendet, die auch in Laptops eingesetzt werden.

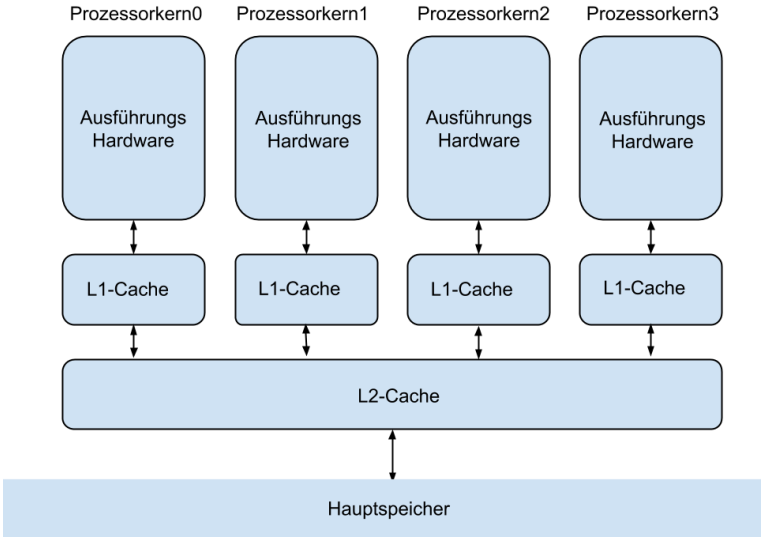


Abbildung 4: Mögliche Architektur eines Quad-Core-Prozessors [11]

An dieser Stelle werden wir nun ein paar Begriffe einführen. Ein Programm das gerade ausgeführt wird, nennt man einen Prozess. Jene Teile des Prozesses, die unabhängig voneinander sind und damit parallel ausgeführt werden können, werden als Threads bezeichnet. Threads sind also Folgen von Anweisungen oder auch voneinander unabhängige Berechnungsströme, die parallel zueinander abgearbeitet werden können.
Eine besondere Eigenschaft von Threads innerhalb eines Prozesses ist, dass diese sich einen Adressspeicher teilen. Wird ein Wert auf den gemeinsamen Adressspeicher geschrieben, dann können alle Threads des Prozesses diesen unmittelbar danach auslesen. Um es noch einmal hervorzuheben, die Threads eines Prozesses können parallel auf verschiedenen Prozessorkernen ausgeführt werden.

Wollen wir also unsere Matrixassemblierung parallel ausführen lassen, dann können alle Threads auf alle Anfangsdaten zugreifen. Lassen wir die Prozessoren nun also mehrere Elementsteifigkeitsmatrizen parallel berechnen. Wenn die Berechnungen abgeschlossen wurden, muss nur noch die große Steifigkeitsmatrix befüllt werden. Doch Achtung, hier ergibt sich ein Problem.

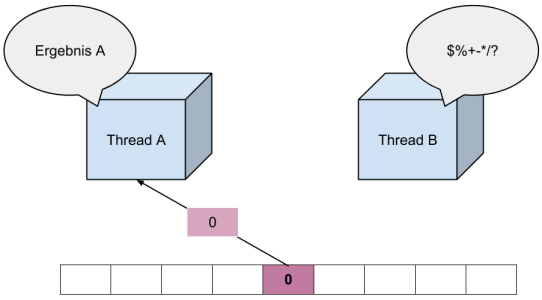
Erinnern wir uns an das das Beispiel im letzten Kapitel. Dessen Steifigkeitsmatrix hatte folgende Struktur:

$$\begin{pmatrix} * & * & 0 & 0 & 0 & \dots & 0 \\ * & * & * & 0 & 0 & \dots & . \\ 0 & * & * & * & 0 & \dots & . \\ 0 & 0 & * & * & * & 0 & . \\ . & & \ddots & \ddots & \ddots & * & 0 \\ . & & & 0 & * & * & * \\ 0 & \dots & \dots & 0 & 0 & * & * \end{pmatrix}$$

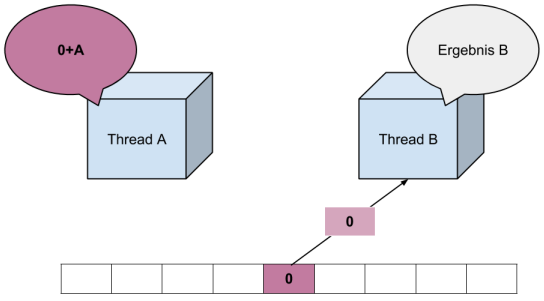
Wobei jene Positionen, die mit einer Farbe umrandet wurden, von dem selben Element abhängen. Wie man aber sehen kann, überschneiden sich diese. Das bedeutet also, dass wenn die Prozessoren ihre neu berechneten Elementsteifigkeitsmatrizen in die große Steifigkeitsmatrix einfügen möchten, dann müssen unterschiedliche Prozessoren auf denselben Speicherplatz zugreifen. Dabei kann folgendes passieren:

2.2 Data races

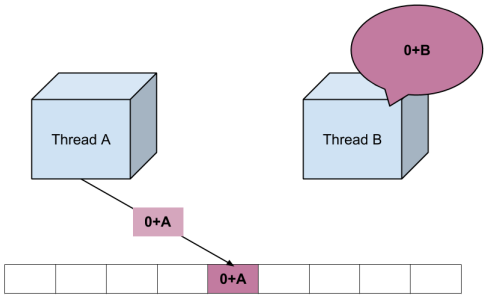
Angenommen 2 Threads, nennen wir sie Thread A und Thread B, berechnen die Elementsteifigkeitsmatrizen von 2 nebeneinander liegenden Elementen. Das bedeutet die beiden Threads werden irgendwann auf denselben Speicherplatz schreiben müssen. Tritt der Fall ein, dass einer der Threads, sagen wir Thread A, mit seinen Berechnungen einen Hauch schneller ist als Thread B, dann kann folgendes Scenario eintreten.



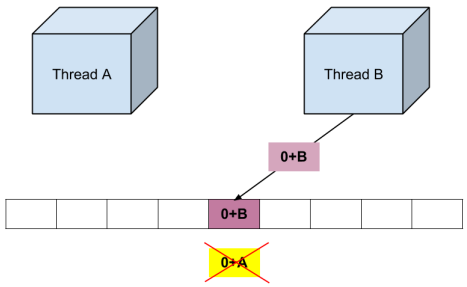
(a) Während Thread B noch rechnet, liest also Thread A bereits die Daten aus.



(b) Während Thread A sein Ergebnis zu den ursprünglichen Daten addiert, schließt Thread B seine Berechnungen ab und liest nun ebenfalls die Daten aus.



(c) Während Thread A die neuen Daten auf die ursprüngliche Speicherstelle schreibt, addiert Thread B die ursprünglichen Daten zu seinem Ergebnis..



(d) Nun schreibt Thread B seine Daten in die Speicherstelle. Er erkennt aber nicht, dass dort nicht mehr die Werte stehen, die er ausgelesen hat und überschreibt die neuen Werte einfach.

Abbildung 5: Entstehung eines Fehlers

Im letzten Schritt wurde also der gerade eben neu berechnete Eintrag überschrieben. Um solche Situationen gar nicht erst aufkommen zu lassen, bietet z.B. die Programmierumgebung OpenMP eine spezielle Funktionen an, die mit der Direktiven *#pragma omp atomic* aufgerufen wird. Diese Funktion bewirkt, dass die Aktualisierung von einer bestimmten Variable, bzw. in unserem Fall die Aktualisierung der Daten in der Steifigkeitsmatrix, zu einer atomaren Operation wird. Das bedeutet während der Aktualisierung kann kein anderer Thread die Variable lesen oder manipulieren. [12]

Unglücklicherweise ist die Verwendung von *#pragma omp atomic* dafür bekannt eventuell sehr langsam zu sein. Deshalb werden bei vielen Problemen Möglichkeiten gesucht, wie man *#pragma omp atomic* vermeiden kann. In den nächsten Kapiteln wird für die Assemblierung der Steifigkeitsmatrix eine Strategie betrachtet, die ein gegenseitiges Blockieren der Threads unnötig macht.

3 Colorierung

Nachdem wir ja versuchen den Assemblierungsalgorithmus zu beschleunigen, freut es uns natürlich nicht, wenn sich die parallel arbeitenden Threads gegenseitig blockieren. Wie wir aber im letzten Kapitel gesehen haben, ist das notwendig, wenn zwei Threads schreibend auf denselben Speicherplatz zugreifen.
Wir hätten also gerne einen Algorithmus bei dem schon von vornherein ausgeschlossen ist, dass sich zwei Threads gegenseitig ihre Ergebnisse überschreiben.

Auf der Suche nach so einem Algorithmus betrachten wir wieder einmal die Struktur der Steifigkeitsmatrix. Dabei fällt folgendes auf: Es müssen nur dann die Einträge von zwei Elementsteifigkeitsmatrizen an denselben Platz in der Steifigkeitsmatrix geschrieben

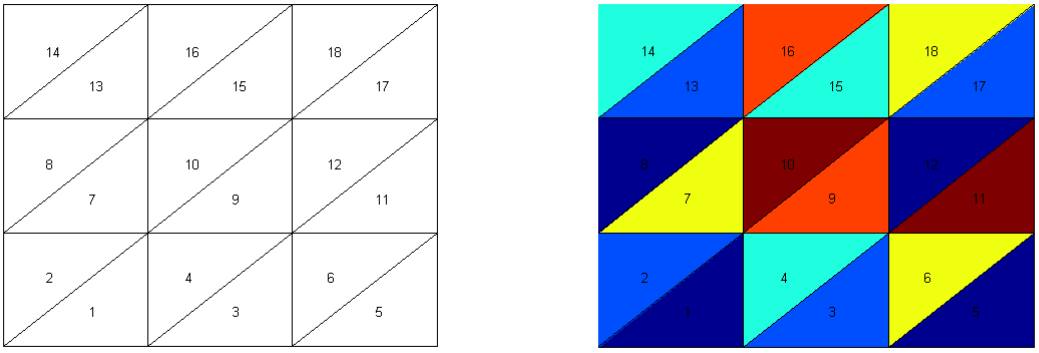
werden, wenn die zu Grunde liegenden Finiten-Elemente einen gemeinsamen Knoten besitzen. Was dann der Fall ist, wenn die Finiten-Elemente nebeneinander liegen.

3.1 Greedy Algorithmus

Wir suchen also nach einer Möglichkeit die Finiten-Elemente so in Gruppen einzuteilen, dass keines der Elemente mit einem weiteren Element dieser Gruppe benachbart ist. Damit wir uns das graphisch besser vorstellen können, formulieren wir das Problem ein wenig um. Wie kann man ein Netz aus finiten Elementen so einfärben, dass keine zwei nebeneinander liegenden Elemente dieselbe Farbe haben? Damit haben wir ein graphentheoretisches Problem zu lösen.
Die vermutlich einfachste Lösung dieses Problems ist ein Greedy-Algorithmus.

Algorithm 2 Greedy-Algorithmus

Nummeriere alle Elemente von 1 bis N durch
Weise dem Element 1 die Farbe 0 zu.
for $i = 2$ bis N **do**
  Bestimme alle Nachbarn des Element i mit Elementnummer $< i$
  Bestimme alle Farben dieser Nachbarn
  $k = 0$
  while k Farbe eines Nachbarn **do**
    $k++$
  end while
  Weise Element i die Farbe k zu
end for



(a) Netz mit nummerierten Elementen.

(b) Mithilfe des Greedy-Algorithmus eingefärbtes Netz.

Abbildung 6: Beispiel für die Verwendung des Greedy-Algorithmus zur Einfärbung eines Netzes.

3.2 Die chromatische Zahl

Wir haben nun einen Weg gefunden, wie man ein beliebiges Netz so einfärben kann, dass keine zwei benachbarten Elemente dieselbe Farbe haben. Es liegt nun also nahe, die Assemblierung der Elementsteifigkeitsmatrizen zur globalen Steifigkeitsmatrix nur für Elemente mit derselben Farbe gleichzeitig durchzuführen.

Dabei wäre es doch praktisch so wenig Farben wie möglich zu verwenden. Denn je weniger Farben verwendet werden, desto mehr Elemente haben die übrigen Farben unter sich aufzuteilen. Doch wie viele Farben braucht man denn mindestens um ein Netz aus finiten Elementen einzufärben? Die Antwort darauf liefert die chromatische Zahl.

Ab jetzt betrachten wir unser Netz als Graph. Wir ändern also die Bezeichnungen ein wenig. Die Elemente des Netzes werden wir die Knoten des Graphs nennen. Die Knoten des Netzes, die ja die Verbindungsstellen der Elemente waren, nennen wir ab nun Kanten des Graphs. Wir untersuchen also einen Graph G mit einer Knotenmenge V und einer Kantenmenge E.

In der Graphentheorie nennt man die minimale Anzahl an Farben, die man benötigt um eine zulässige Färbung eines Graphs G zu erhalten *chromatische Zahl*. Mit zulässiger Färbung ist natürlich jene Färbung gemeint, bei der alle Knoten des Graphs, die über eine Kante miteinander verbunden sind, unterschiedliche Farben haben. Die chromatische Zahl bezeichnen wir mit $\chi(G)$.

Eine Eigenschaft der chromatischen Zahl sieht man sofort. Für einen Graphen G mit n Knoten gilt:

$$1 \leq \chi(G) \leq n$$

Für das Beispiel in Abbildung 6 gelten also die Abschätzungen:

$$1 \leq \chi(G) \leq 18$$

Diese Abschätzungen sind natürlich so grob, dass man davon rein gar nichts ablesen kann. Es gibt allerdings auch sehr aussagekräftige Abschätzungen. Eine nützliche Abschätzung nach oben liefert der Maximalgrad des Graphen. Wenn der $deg(v)$ die Anzahl aller Kanten ist, die vom Knoten $v \in V$ ausgehen, dann ist der Maximalgrad von G:

$$\Delta G = \max_{v \in V} \{deg(v)\}$$

Die chromatische Zahl lässt sich durch den Maximalgrad nach oben beschränken:

$$\chi(G) \leq \Delta(G) + 1$$

Den Beweis zu dieser Aussage findet man in [13]. In unserem Beispiel wäre $\Delta(G) = 12$. Denn die Elemente mit den Nummern 9 und 10, welche in der Mitte des Netzes liegen, haben jeweils 12 Nachbarn. Alle anderen Elemente haben weniger.

Eine Schranke nach unten findet man über die vollständigen Untergraphen $\bar{G} = (\bar{V}, \bar{E})$ mit $\bar{V} \subset V$ und $\bar{E} \subset E$ von G. Ein vollständiger Graph ist ein Graph mit n Knoten, in dem jeder Knoten mit jedem anderen Knoten über eine Kante verbunden ist. Um einen vollständigen Graphen mit n Knoten zulässig einzufärben, benötigt man genau n Farben.
Wir bezeichnen mit $\omega(G)$ die maximale Anzahl an Knoten, die ein vollständiger Untergraph von G haben kann. Dann gilt für die chromatische Zahl

$$\omega(G) \leq \chi(G)$$

In unserem Beispiel ist $\omega(G)$ die maximale Anzahl an Elementen, die sich einen Knoten teilen können. Ein Blick auf Abbildung 6 liefert also $\omega(G) = 6$. Die chromatische Zahl des Beispiels muss also

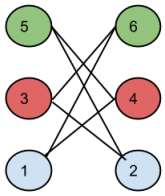
$$6 \leq \chi(G) \leq 13$$

erfüllen. Nachdem wir aber bereits eine zulässige Färbung des Netzes mit 6 Farben gefunden haben, wissen wir damit, dass die Einfärbung durch den Greedy Algorithmus für dieses Beispiel optimal war.

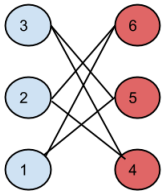
3.3 Optimale Färbung durch Greedy-Algorithmus?

Ein Greedy-Algorithmus ist dadurch gekennzeichnet, dass er in jedem Schritt die Entscheidung trifft, die im Moment am Besten erscheint. Er trifft also lokal eine optimale Entscheidung und hofft darauf, dass diese zu einer global optimalen Lösung führt. [2] In unserem Colorierungs-Algorithmus wählt er für jedes neue Element die Farbe aus, welche die kleinste Farbnummer hat, die noch nicht einem Nachbarelement zugeteilte wurde.

In unserem Beispiel hat der Greedy-Algorithmus eine optimale Lösung gefunden, das ist aber in keinster Weise selbstverständlich. Betrachten wir ein Gegenbeispiel:



(a) Colorierung bei ungünstiger Elementnummerierung



(b) Colorierung bei günstiger Elementnummerierung

Das heißt also, es hängt von der Nummerierung der Knoten ab, ob der Greedy Algorithmus eine optimale Lösung findet. Dazu formulieren wir folgenden Satz:

Satz 1. *Für jeden Graphen $G = (V, E)$ gibt es eine Knotennummerierung, so dass der Greedy-Algorithmus eine optimale Färbung mit genau $\chi(G)$ Farben findet. [14] Seite 143*

Beweis. Angenommen es ist eine optimale Färbung bekannt und es seien $c_1, \ldots, c_N$ die Farben dieser Färbung. Es werden alle Knoten mit Farbe c_1 durchnummeriert, danach die Knoten mit Farbe c_2 usw. bis zu den Knoten mit Farbe c_N. Dann liefert der Greedy-Algorithmus eine Colorierung mit genau $\chi(G)$ Farben. Denn jeder Knoten v, kann nur eine Farbnummer zugewiesen bekommen, die kleiner ist als jene die er in der optimalen Färbung hätte. □

Ein weiterer Satz liefert eine Abschätzung dafür, wie nahe das Ergebnis des Greedy-Algorithmus bei der optimalen Färbung liegt.

Satz 2. *Sei M gleich der Anzahl der Farben die der Greedy-Algorithmus für eine zulässige Färbung des Graphen G benötigt[5]:*

$$M \leq \Delta(G) + 1$$

Beweis. Für jeden neuen Knoten, den der Greedy Algorithmus einfärben möchte, können höchstens so viele Farben verboten sein, wie der jeweilige Knoten Kanten (deg(v)) hat. Deshalb wird der Greedy-Algorithmus maximal $\Delta(G) + 1$ benötigen um den Graphen einzufärben. □

Da bei den Netzen, die wir untersuchen der Maximalgrad relativ klein ist, ist der Greedy-Algorithmus für unsere Zwecke durchaus geeignet.

3.4 Tatsächliche Implementierung der Colorierung

Wenn wir die Colorierung eines vorhandenen Netzes vornehmen möchten, dann brauchen wir erstmal genügend Informationen über dieses Netz. Die Daten, die uns bei der Assemblierung der Steifigkeitsmatrix zur Verfügung stehen, sind die Einträge der Elementzusammenhangstabelle. Zur Erinnerung, das ist eine Tabelle, in der in der i-ten Zeile die globalen Knotennummern der Knoten des i-ten Elements, geordnet nach den lokalen Knotennummern, stehen.(Tabelle 1.2)

Wir kennen also die Anzahl der Elemente und die lokalen und globalen Knotennummern von jedem einzelnen Element.

- Im ersten Schritt müssen wir wissen, welche Elemente benachbart sind. Da wir Suchprozesse vermeiden möchten, konstruieren wir zuerst einen Vektor, der an der Stelle i all jene Elemente speichert, die den Knoten i berühren. Dieses Konstrukt realisieren wir als Vektor aus dynamischen Vektoren.

Algorithm 3 Elemente der Knoten (eon)

```
Lade die Elementzusammenhangstabelle
Erzeuge vector < vector < int >> eon
for i = 0 bis Anzahl der Elemente do
  for j = 0 bis Anzahl der Knoten pro Element do
    Speicher das Element i in eon an der Stelle der globalen Knotennummer des
    Knoten j
  end for
end for
```

- Dann erzeugen wir einen Vektor, der an der Stelle i die Nachbarn des i-ten Elements gespeichert hat.

Algorithm 4 Nachbarn der Elemente (noe)

```
Lade Vektor eon "Elemente der Knoten"
Erzeuge vector < vector < int >> noe
for i = 0 bis Anzahl der Elemente do
  for j = 0 bis Anzahl der Knoten pro Element do
    Speichere in noe an der Stelle i alle Elemente, die in eon an der Stelle der globalen
    Knotennummer des lokalen Knoten j von Elements i stehen.
  end for
end for
```

Zum Schluss ordnen wir die Elemente in den inneren Vektoren aufsteigend und entfernen alle Elemente die mehr als ein Mal vorkommen.

- Wir haben nun also einen Vektor zur Verfügung in dem für jedes Element alle seine Nachbarelemente gespeichert sind. Mithilfe dieses Vektors können wir nun auch jedem Element eine geeignete Farbe zuweisen.
  Um das durchzuführen erzeugen wir für jedes Element einen kleinen Vektor, der alle Farben der bereits eingefärbten Nachbarelemente speichert. Die kleinste Farbnummer, die in diesem Vektor nicht vorkommt wird dem aktuellen Element zugewiesen.

Algorithm 5 Colorierung

```
Lade Vektor noe "Nachbarn der Elemente"
Erzeuge vector < int > lokalen Farbvektor
Erzeuge vector < int > globalen Farbvektor
setze globalen Farbvektor[0]=0 (Element Nr. 1 hat Farbe 0)
for Element i = 2 bis Anzahl der Elemente do
  Speichere alle Farben der Nachbarn von Element i (mit Elementnummer < i)
  in den lokalen Farbvektor
  Setze Farbe f=0
  while (Farbe f in lokalem Farbvektor enthalten ist) do
    f++
  end while
  Setze globalen Farbvektor[i]=f (Element i wurde aktueller Wert von f zugewiesen)
end for
```

- Nachdem wir nun also jedem Element eine Farbe zugeordnet haben, hätten wir noch gerne einen Vektor, der uns die Elemente nach Farben geordnet speichert. Das brauchen wir damit wir ohne lange suchen zu müssen, sofort wissen, von welchen Elementen die Elementsteifigkeitsmatrizen gleichzeitig in die globale Steifigkeitsmatrix geschrieben werden dürfen.

Algorithm 6 Zuordnungsvektor

```
Der globale Farbvektor ist bekannt
Erzeuge vector < vector < int >> Zuordnungsvektor
for Element i=1 bis Anzahl der Elemente do
  Zuordnungsvektor[globaler Farbvektor[i]]=i
end for
```

Das Befüllen des Zuordnungsvektors mit der Farbe von Element i kann selbstverständlich sofort geschehen, wenn die Farbe des Elements bekannt ist. Es kann der Code von Algorithmus 6 natürlich ohne weiteres in Algorithmus 5 eingebaut werden.

Bemerkung 3. *Wenn man ein bestimmtes Netz einmal eingefärbt hat, dann ist diese Colorierung für immer gültig. Ganz gleich ob man nun eine gänzlich andere Partielle Differentialgleichung auf diesem Gebiet untersucht, oder einfach nur andere Ansatzfunktionen betrachtet. Man darf also annehmen, dass die Colorierung eines Netzes nur ein einziges Mal durchgeführt werden muss.*

4 Versionen der Assemblierung

Da wir nun mehrere Möglichkeiten kennen wie Data races zu verhindern sind, werden wir uns jetzt die Implementierung dieser Möglichkeiten ansehen und sie in Bezug auf Laufzeiten und Speicherplatz vergleichen.

4.0.1 Prozessorinformation

Die Messungen der Laufzeiten wurden auf einem Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz durchgeführt. Dieser Prozessor enthält 10 physische Prozessorkerne auf denen man jeweils bis zu 2 Threads ausführen lassen kann. Jeder Core besitzt einen L1-Cache mit (32 [i] + 32 [d]) kB und einen eigenen L2-Cache mit 256kB. Zusätzlich enthält der Prozessor einen L3-Cache mit 25MB, der von allen Prozessorkernen gemeinsam genutzt wird.
Der Intel(R) Xeon(R) CPU E5-2660 v2 hat eine maximalen Speicherbandbreite von 59,7 GB/s (4 Speicherkanäle, ECC). Im Arbeitsspeicher stehen uns insgesammt 65 GB zur Verfügung. [4][8]

4.0.2 Compiler Optionen

Damit die nachfolgenden Laufzeitmessungen auch nachvollziehbar sind, werden an dieser Stelle die verwendenten Compileroptionen beschrieben. Für das Übersetzten des Codes haben wir den GNU-Compiler GCC 4.8.2 verwendet.

OpenMP Für die Parallelisierung verwenden wir die standardisierte Programmierschnittstelle OpenMP. [7]

Optimierung Bei den ersten Zeitmessungen werden wir auf eine Optimierung mithilfe des Compilers gänzlich verzichten. Wir verwenden die Compileroption -O0.

Der Vollständigkeit halber, werden wir aber nach der Vorstellung der verschiedenen Implementierungen die Messungen ohne Optimierung, mit jenen mit Optimierung vergleichen.
Dazu verwenden wir die Compileroptionen -O2. Diese lässt den GCC-Compiler alle Möglichkeiten zur Laufzeitoptimierung ausschöpfen, die die Menge an benötigtem Speicherplatz nicht negativ beeinflusst. Diese Option erhöht zwar die Zeit der Kompilierung, beschleunigt dafür aber die Laufzeit des Programms. [6]

4.0.3 Testbeispiel

In den folgenden Kapiteln werden wir versuchen die Implementierung des Assemblierungsalgorithmus Schritt für Schritt zu verbessern. Dazu werden wir die verschiedenen Versionen in Bezug auf ihre Laufzeiten vergleichen.
Die Laufzeitmessungen des Codes werden anhand eines Testbeispiels vorgenommen. Wir verwenden ein Netz aus Tetraedern mit 15918120 Elementen und 3298715 Knoten und erzeugen die Steifigkeitsmatrix für die 3-dimensionale Wärmeleitungsgleichung.

4.1 Original Version

Beginnen wir mit der Beschreibung der ursprünglichen Implementierung des Assemblierungsalgorithmus. Bei dieser Version wird noch keine Parallelisierung durchgeführt. Sie interessiert uns deshalb vor allem als Vergleichsversion zu späteren Implementierungen, die eine Parallelisierung ausnützen.
Wir erhoffen uns Informationen darüber, wieweit wir diesen Code über eine Parallelisierung mit OpenMP verbessern können und welchen Effekt die innere Speicherstruktur haben kann.

4.1.1 Originale Version nicht parallelisiert

Den Assemblierungsalgorithmus aus dem Einführungskapitel können wir in 2 getrennt zu betrachtende Hauptteile aufspalten. Der eine wäre die Berechnung der Elementsteifigkeitsmatrizen und der andere die Assemblierung der Steifigkeitsmatrix.

Die Berechnung der Elementsteifigkeitsmatrizen wird in dieser Version in mehrere kleine Zwischenschritte aufgeteilt, wobei in jedem dieser Zwischenschritte die dazugehörigen Berechnungen immer gleich für alle Elemente des Netzes durchgeführt werden. Die dabei erhaltenen Ergebnisse werden in langen Vektoren gespeichert.

Bei der Assemblierung der Steifigkeitsmatrix werden die Einträge der zuvor berechneten Elementsteifigkeitsmatrizen nacheinander in die Steifigkeitsmatrix addiert. Wie schon vorhin bemerkt findet auch hier keine Parallelisierung statt, weshalb man sich um Data Races keine Sorgen machen muss.

Es wäre außerdem noch hervorzuheben, dass die Berechnungen und die Assemblierung in dieser Version zeitlich getrennt voneinander stattfinden.

Algorithm 7 ursprüngliche Assemblierung der Steifigkeitsmatrix (nicht parallelisiert)

for $i = 1$ bis Anzahl der Elemente **do**
  berechne für Element i Zwischenergebnis 1
end for
  ⋮
for $i = 1$ bis Anzahl der Elemente **do**
  berechne für Element i aus Zwischenergebnissen $1 \dots N$ die Elementsteifigkeitsmatrix
end for
for $i = 1$ bis Anzahl der Elemente **do**
  schreibe Elementsteifigkeitsmatrix in globale Steifigkeitsmatrix
end for

Die Laufzeitmessungen dieses Codes ergeben, dass für die Berechnungen durchschnittlich 6.4223 Sekunden und für die Assemblierung durchschnittlich 3.4744 Sekunden benötigt werden. In der vorhandenen Form kann der Assemblierungsalgorithmus ohne großen Aufwand parallelisiert werden.

4.1.2 Original Version parallelisiert

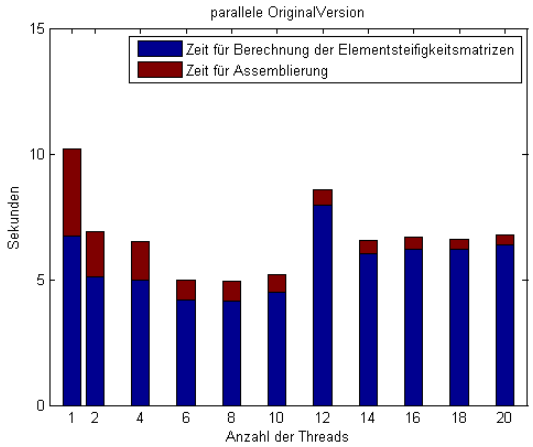
Den Algorithmus in der oben beschriebenen Form können wir parallelisieren, indem wir jeden Zwischenschritt für beliebige Elemente parallel berechnen lassen. Data Races während der Assemblierung verhindern wir mit der *# pragma atomic* Direktiven.

Algorithm 8 ursprüngliche Assemblierung der Steifigkeitsmatrix (parallelisiert)

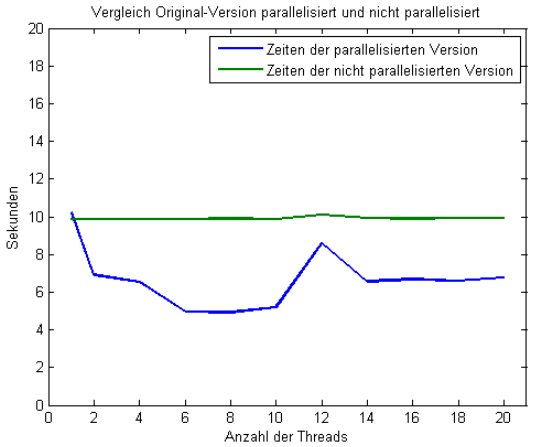
# pragma omp parallel for
for $i = 1$ bis Anzahl der Elemente **do**
  berechne für Element i Zwischenergebnis 1
end for
  ⋮
# pragma omp parallel for
for $i = 1$ bis Anzahl der Elemente **do**
  berechne für Element i aus Zwischenergebnissen $1 \dots N$ die Elementsteifigkeitsmatrix
end for

# pragma omp parallel for
for $i = 1$ bis Anzahl der Elemente **do**
  # pragma omp atomic
  schreibe Elementsteifigkeitsmatrix in globale Steifigkeitsmatrix
end for

Die Laufzeitmessungen dieser Implementierung ergeben folgendes Bild:



(c) Zeiten der parallelisierten Original-Version in Abhängigkeit von der Anzahl an Threads



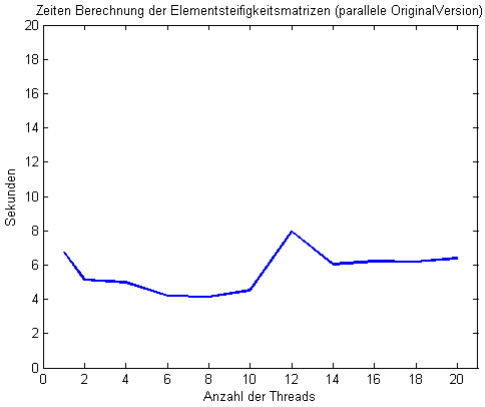
(d) Vergleich der Laufzeiten der parallelisierten und nicht parallelisierten Original-Version .

Abbildung 7: Graphik der Laufzeiten der parallelisierten ursprünglichen Version

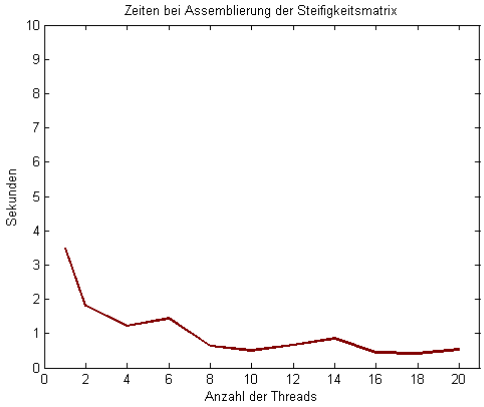
Wenn wir diese zwei Graphiken betrachten, dann fällt uns folgendes auf: Durch die Parallelisierung können wir zwar zu Beginn die Laufzeiten verbessern, aber bei etwa 8 Threads beginnen sich diese wieder etwas zu verschlechtern. Beim Übergang von 10 zu 12 Threads erfolgt sogar eine markante Verschlechterung. Woran könnte das liegen?

Bei der Berechnung der Elementsteifigkeitsmatrizen wurden zwar mehrere Teilberechnungen parallelisiert, diese sind aber sehr kurz. Die verschiedenen Threads müssen also sehr schnell hintereinander Daten laden und wieder in den Hauptspeicher speichern. Da alle Threads für das Laden und Speichern von Daten dieselbe Leitung verwenden, könnten sie sich dabei gegenseitig blockieren.
Außerdem haben wir bekanntermaßen nur 10 physische Cores zur Verfügung, wenn wir also mehr als 10 Threads verwenden möchten, müssen wir auf Multithreading zurückgreifen.

Betrachten wir an dieser Stelle die Laufzeiten der Berechnungen der Elementsteifigkeitsmatrizen und die Laufzeiten der Assemblierung voneinander getrennt.
Wir sehen anhand von Graphik 8, dass sich die Laufzeiten beider Teile des Algorithmus bei etwa 10 Threads zu verschlechtern beginnen. Während die Assemblierung aber wieder schneller wird, scheint sich die Berechnung der Elementsteifigkeitsmatrizen nicht mehr auf ein früheres Niveau verbessern zu können.



(a) Zeiten der Berechnung der Elementsteifigkeitsmatrizen in Abhängigkeit von der Anzahl an Threads



(b) Zeiten der Assemblierung

Abbildung 8: getrennte Betrachtung der Zeiten für die Berechnungen und die Assemblierung

Da wir ja vermuten, dass die vorhanden Speicherstruktur des Algorithmus für die Verwendung von mehreren Threads nicht ideal ist, implementieren wir den Assemblierungsalgorithmus ein weiteres Mal. Diesmal werden wir die Berechnungen aber etwas anders strukturieren.

4.2 Atomic Version

In der vorherigen Version haben wir bei der Berechnung der Elementsteifigkeitsmatrizen die diversen Zwischenberechnungen immer jeweils für alle Elemente ausgeführt, bevor die nächsten Zwischenberechnungen begonnen wurden. Nun möchten wir aber für jedes Element die gesammte Berechnung der Elementsteifigkeitsmatrix in einem Zug durchführen. Damit erhalten wir die Möglichkeit diese Berechnungen für verschiedene Elemente gleichzeitig durchzuführen, also zu parallelisieren.

Außerdem ist es nun nicht mehr nötig für jedes Zwischenergebnis einen riesigen Vektor zum Speichern der Ergebnisse aller Elemente zu erzeugen. Es genügt für jedes Element und jeden Zwischenschritt einen kleinen privaten Vektor zu erzeugen, der wenn die Berechnungen für dieses Element abgeschlossen sind nicht mehr benötigt wird und wieder freigegeben werden kann.

Algorithm 9 neu strukturierte Assemblierung der Steifigkeitsmatrix

```
# pragma omp parallel for
for i = 1 bis Anzahl der Elemente do
   berechne für Element i Elementsteifigkeitsmatrix
end for
# pragma omp parallel for
for i = 1 bis Anzahl der Elemente do
   # pragma omp atomic
   schreibe Elementsteifigkeitsmatrix in globale Steifigkeitsmatrix
end for
```

Betrachten wir die Graphik der Laufzeitmessungen dazu:

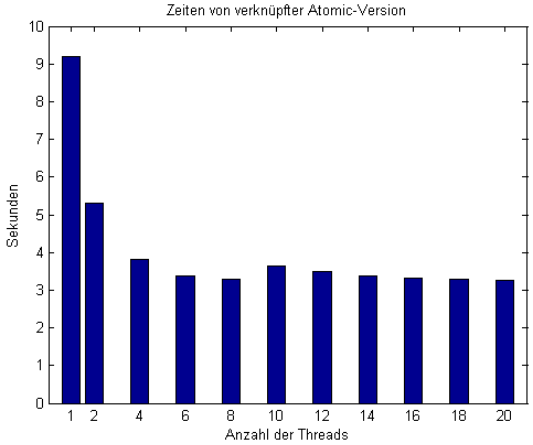
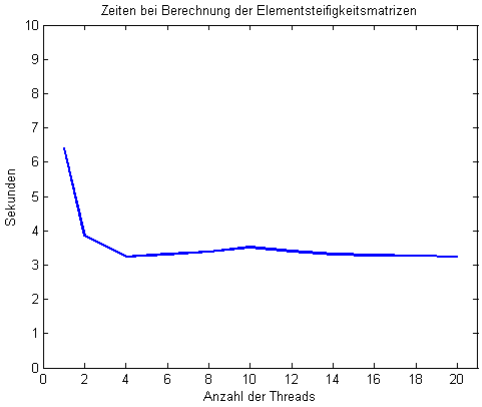


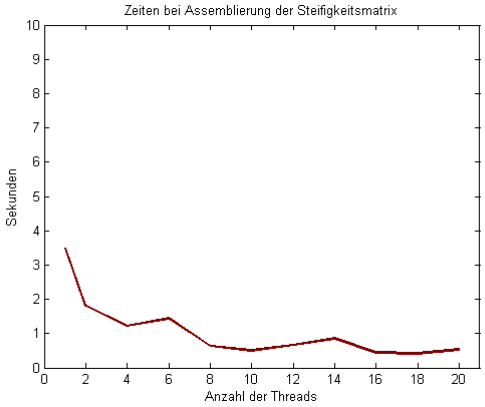
Abbildung 9: Zeiten der Atomic-Version in Abhängigkeit von der Anzahl an Threads

Wie man sieht verbessern sich die Laufzeiten zu Beginn recht gut. Ab der Verwendung von mehr als 8 Threads kann man allerdings keine wesentlichen Verbesserungen mehr erkennen. An dieser Stelle steigen die Zeiten kurzzeitig sogar ein wenig, beginnen dann aber wieder zu fallen.

Betrachten wir die Zeiten für die Berechnungen der Elementsteifigkeitsmatrizen wieder extrahiert:



(a) Zeiten für die Berechnung der Elementsteifigkeitsmatrizen in Abhängigkeit von der Anzahl an Threads



(b) Zeiten der Assemblierung

Abbildung 10: Zeiten separat betrachtet

Wie man an den Graphiken 13 erkennen kann, erzielt man bei der Berechnung der Elementsteifigkeitsmatrizen das beste Ergebnis, wenn etwa 4 Threads verwendet werden. Die Zeiten für die Assemblierung der Steifigkeitsmatrix entsprechen denen im vorigen Abschnitt.
Ein kurzer Vergleich der beiden Versionen lässt uns erkennen, dass wir durch die Neustrukturierung bei der Berechnung der Elementsteifigkeitsmatrizen durchaus Zeit eingespart haben.

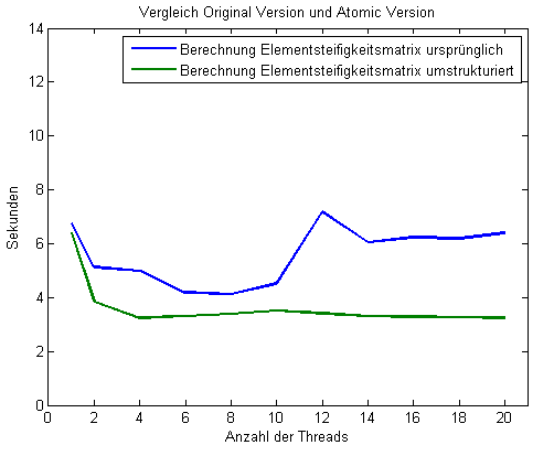


Abbildung 11: Vergleich Berechnungen der Elementsteifigkeitsmatrizen in Original Version und in neu strukturierter Version

Aus diesem Grund werden wir nun versuchen auch die Assemblierung der Steifigkeitsmatrix nicht mehr getrennt von der Berechnung der Elementsteifigkeitsmatrizen durchzuführen, sondern diese direkt nach ihrer Erzeugung in die Steifigkeitsmatrix zu schreiben.

Algorithm 10 Elementweise Berechnung und Befüllung der Steifigkeitsmatrix

```
# pragma omp parallel for
for i = 1 bis Anzahl der Elemente do
    berechne für Element i Elementsteifigkeitsmatrix
    # pragma omp atomic
    schreibe Elementsteifigkeitsmatrix in globale Steifigkeitsmatrix
end for
```

Der Vergleich der beiden Atomic-Versionen zeigt, dass sich auch hier ein kleiner Vorteil aus der neuerlichen Umstrukturierung ergibt.

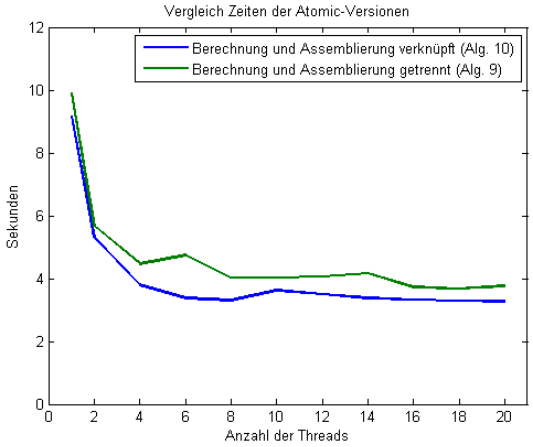


Abbildung 12: Vergleich der beiden Atomic-Versionen

4.3 Farben Version

Zu guter Letzt betrachten wir jene Implementierung, die wir schon so lange vorbereitet haben. Wir betrachten also eine Umsetzung des Assemblierungsalgorithmus, bei der die Colorierung des Netzes ausgenützt wird. Dabei können wir davon ausgehen, dass der "Zuordnungsvektor", also jener Vektor der zu jeder Farbe die dazugehörigen Elemente speichert, bekannt ist.

Wie bereits besprochen wurde, werden beim Assemblieren der Steifigkeitsmatrix die Einträge von zwei Elementsteifigkeitsmatrizen ganz sicher nicht auf den selben Speicherplatz

geschrieben, wenn die dazugehörigen Elemente keinen gemeinsamen Knoten haben.
Mithilfe der Colorierung des Netzes haben wir die Elemente so in Gruppen eingeteilt, dass sich keine zwei Elemente einer Gruppe einen Knoten teilen.
Werden die Elementsteifigkeitsmatrizen von Elementen aus derselben Gruppe gleichzeitig in die globale Steifigkeitsmatrix geschrieben, muss man sich also um Data Races keine Sorgen machen.

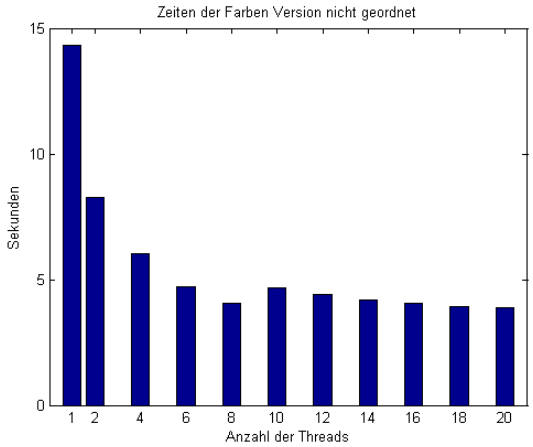
Wir nützen das bei der Implementierung des Assemblierungsalgorithmus aus, indem wir für Elemente mit der gleichen Farbe die Berechnung der Elementsteifigkeitsmatrizen und das Hineinschreiben in die globale Steifigkeitsmatrix parallel auf mehreren Threads durchführen lassen.
Die Elementgruppen die zu verschiedenen Farben gehören werden dabei hintereinander abgearbeitet.
Das bedeutet, die Berechnungen und die Assemblierung können für zwei Elemente mit derselben Farbe gleichzeitig durchgeführt werden, für Elemente mit unterschiedlichen Farben werden sie aber nacheinander ausgeführt.

Der Algorithmus sieht also wie folgt aus:

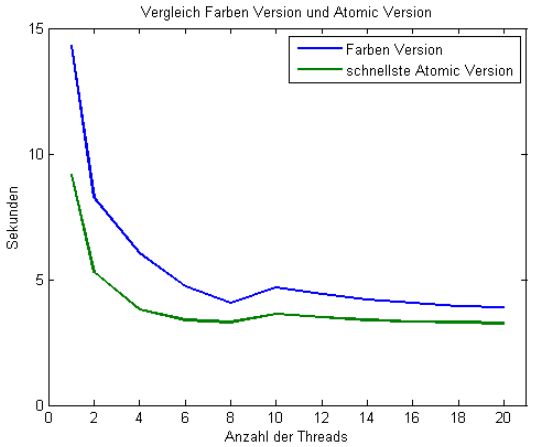
Algorithm 11 Assemblierung mit Colorierung

for $i = 1$ bis Anzahl der Farben **do**
  #pragma omp parallel for
  for $j =$ erstes Element der Farbe i bis letztes Element der Farbe i **do**
    berechne Elementsteifigkeitsmatrix $K^{(j)}$ für Element j
    schreibe Elementsteifigkeitsmatrix in globale Steifigkeitsmatrix
  end for
end for

Nun interessiert uns natürlich die Laufzeit dieses Codestücks. Betrachten wir dazu folgende Graphik:



(a) Zeiten der Colorierungsversion in Abhängigkeit von der Anzahl an Threads

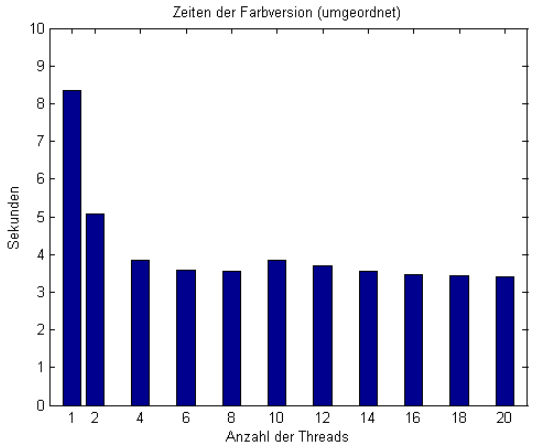


(b) Vergleich Colorierungsversion und Atomic-Version

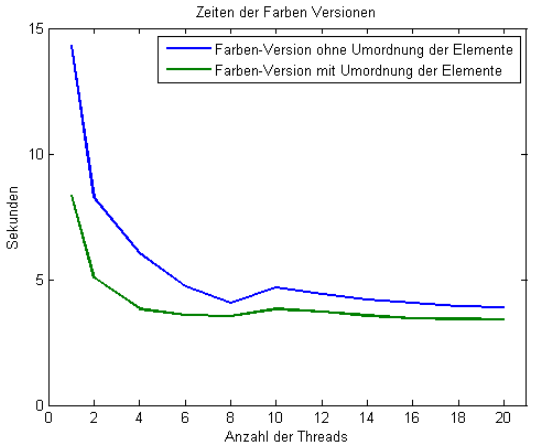
Abbildung 13: Laufzeitmessungen der Colorierungsversion und Vergleiche mit der Atomic Version

Die Laufzeiten, die wir mit dieser Implementierung des Assemblierungsalgorithmus erhalten haben, sind zwar gut, wir hätten uns aber im Vorfeld erwartet, dass diese besser sind.
Es stellt sich für uns nun die Frage, ob es einen negativen Effekt auf die Laufzeit dieses Codestücks hat, dass wir die Elementsteifigkeitsmatrizen nicht in derselben Reihenfolge berechnen, wie die Elemente intern gespeichert sind.

Um das zu überprüfen ordnen wir, bevor wir den Assemblierungsalgorithmus anwenden, die Elemente physisch so um, dass Elemente mit der selben Farbe auch nebeneinander gespeichert sind. Vergleichen wir nun die Zeiten dieser beiden Varianten:



(a) Zeiten der Colorierungsversion in Abhängigkeit von der Anzahl an Threads



(b) Vergleich der Laufzeit des Assemblierungsalgorithmus bei Umordnung der Elemente.

Wie wir anhand der Graphik sehen können, ergibt sich für die beiden Colorierungs-Versionen, die sich nur in der Anordnung der Elemente im Speicher unterscheiden, tatsächlich ein Unterschied in den Laufzeiten. Wodurch wird dieser Effekt ausgelöst?

Moderne Prozessoren unterstützen sogenanntes "Data-prefetching". Dabei werden Daten und Instruktionen aus dem Hauptspeicher in die Caches geladen, bevor sie eigentlich gebraucht werden. Da die Prozessoren auf die Daten in den Caches viel schneller zugreifen können, als auf jene im Hauptspeicher, beschleunigt das in vielen Fällen das Programm.

Um auszuwählen welche Daten der Prozessor quasi "sicherheitshalber" schon einmal lädt, wird oftmals ein sehr simpler Algorithmus verwendet.
Dieser lässt den Prozessor nicht nur die aktuell benötigten Daten laden, sondern einfach auch ein paar jener Daten, die im Speicher gleich dahinter liegen.
Nachdem wir aber in der ersten Version parallel mit Elementen gearbeitet haben, die weder im Netz, noch im Speicher nebeneinander liegen, konnten wir in diesem Fall das "Data-prefetching" nicht ausnützen. Und das hat man in der Performance des Programms gemerkt. [1]

Zu guter Letzt möchten wir nun wissen, ob wir einen zeitlichen Vorteil erwarten können wenn wir statt der # pragma atomic Direktiven eine Colorierung verwenden um Data Races zu verhindern.

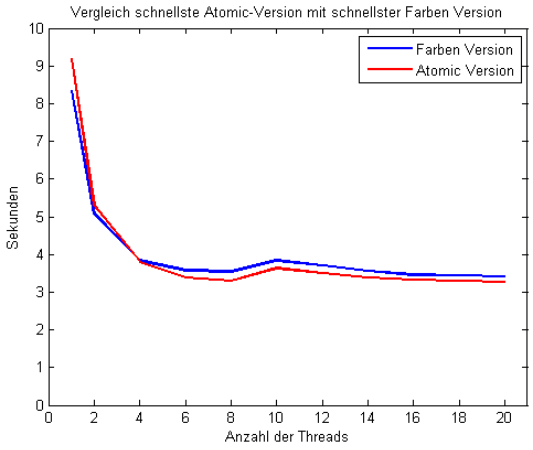


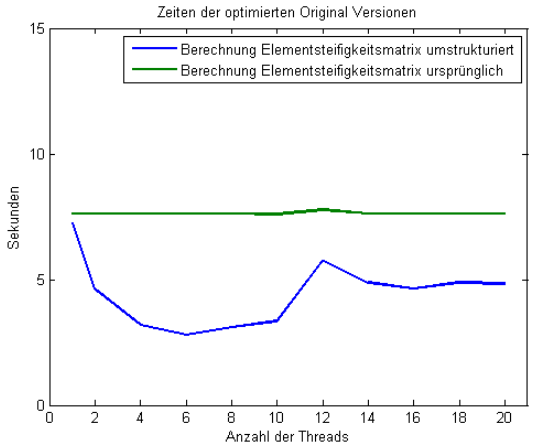
Abbildung 14: Vergleich der Farben Version und der Atomic-Version

Wie die Abbildung 14 uns zeigt, ist es in Hinblick auf die Laufzeiten für dieses Problem vollkommen egal, ob nun eine Colorierung oder die # pragma atomic Direktiven verwendet, solange die Speicherstruktur günstig für eine Parallelisierung ist.

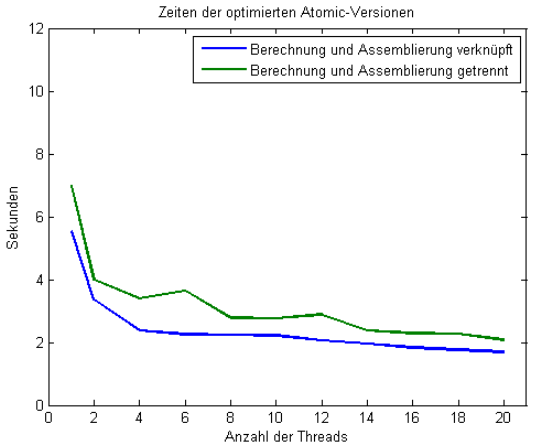
5 Laufzeitenvergleich der Versionen mit und ohne Optimierung

Die Laufzeitmessungen des vorigen Kapitels wurden ohne Optimierung von Seiten des Compilers durchgeführt. Da aber die Codeoptimierung des Compilers einen großen Effekt auf die Laufzeit eines Programms haben kann, vergleichen wir an dieser Stelle die gemessenen Werte der Codes ohne Optimierung mit jenen mit Optimierung. Wir verwenden also die Compileroption -O2.

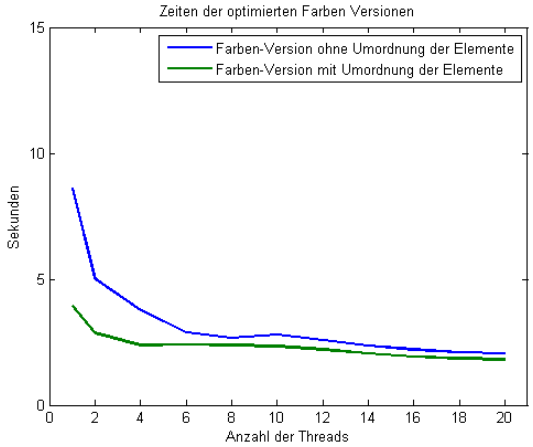
In den folgenden Graphiken werden wir die optimierten Varianten der drei vorgestellten Versionen vergleichen, um zu sehen ob die Verhältnisse trotz Optimierung gleich bleiben. Die jeweils schnellsten Varianten werden wir noch einmal in einer eigenen Graphik darstellen.



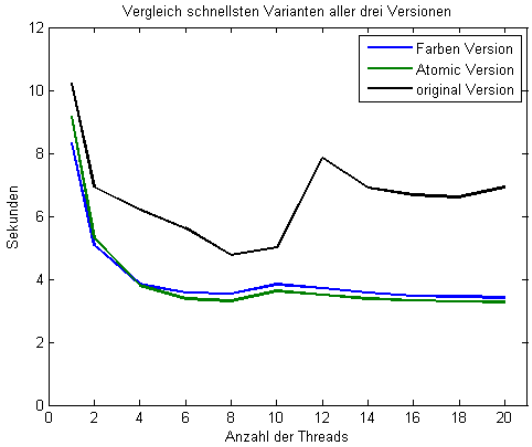
(a) Vergleich optimierte Original Versionen



(b) Vergleich der optimierten Atomic Versionen



(c) Vergleich der optimierten Farben Versionen



(d) Vergleich der schnellsten Varianten der optimierten Versionen

Bemerkung 4. *An dieser Stelle müssen wir leider einen Blick auf eine Unstimmigkeit bei den Laufzeiten werfen, die leicht übersehen werden könnte. Wird nur ein einziger Thread genutzt, dann ist die verknüpfte Atomic Version langsamer als die schnellste Farben Version. Das hätten wir uns nicht erwartet.*
Dieser negative Effekt wird durch die # pragma omp atomic Direktive ausgelöst. Es scheint, als könnte der von uns genutzte Compiler nicht erkennen, dass # pragma omp atomic bei der Verwendung von nur einem Thread nicht benötigt wird. Dadurch vergibt er sich vermutlich die Möglichkeit dieses Codestück vollständig zu optimieren.
Da wir jedoch ohnehin eher an den tatsächlich parallelisierten Versionen interessiert sind und Unterscheidungen innerhalb unserer Schleifen die Laufzeiten bei mehr als einem Thread verschlechtern würden, werden wir unsere Versionen nicht verändern. Man sollte

diesen Effekt der # pragma omp atomic Direktive allerdings nicht vergessen.

Zu guter Letzt vergleichen wir noch die schnellste nicht optimierte Version, also die verknüpfte Atomic Version mit der schnellsten optimierten Version, was ebenfalls die verknüpfte Atomic Version wäre.

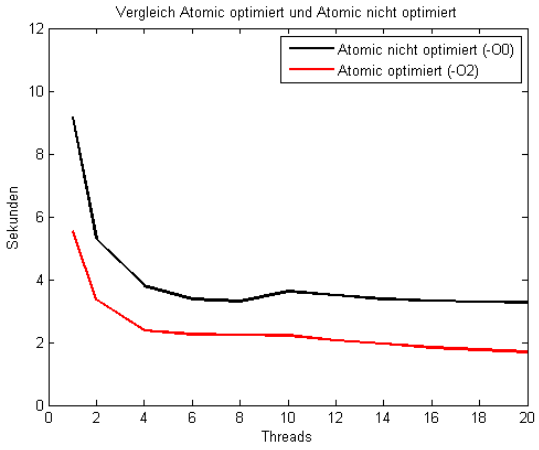


Abbildung 15: Vergleich der optimierten mit der nicht optimierten verknüpften Atomic Version

Diese Graphik zeigt uns, dass wir durch Optimierung noch einmal eine gute Verbesserung der Laufzeiten erreichen können.

Die verschiedenen Varianten, die wir implementiert haben, interessieren uns aber nicht nur in Bezug auf ihre Laufzeiten, sondern auch auf ihren Speicherplatzverbrauch. Diesbezügliche Messungen werden im nächsten Kapitel vorgestellt.

6 Vergleich des Speicherplatzbedarfs

In den vorigen Kapiteln haben wir immer wieder die Speicherstrukturen in unserem Code geändert um Verbesserungen der Laufzeit zu erreichen. Deshalb sollten wir uns nun auch mit der Menge an Speicherplatz beschäftigen, die die verschiedenen Versionen des Assemblierungsalgorithmus benötigen.

In der folgenden Graphik sehen wir den Speicherplatzverbrauch der implementierten Versionen in Abhängigkeit von der Anzahl der verwendeten Threads. Für die Messungen wurden die Einstellungen und das Testbeispiel des vorigen Kapitels übernommen.

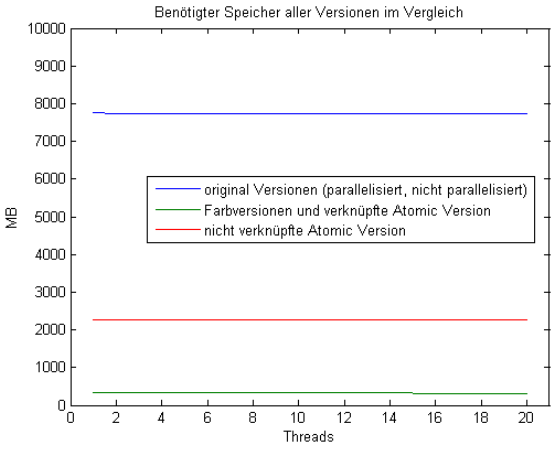


Abbildung 16: Speicherplatzverbrauch während dem Assemblierungsalgorithmus

Anhand von Graphik 16 können wir erkennen, dass die ursprünglichen Versionen bei der Berechnung der Elementsteifigkeitsmatrizen und der Assemblierung bei weitem am meisten Speicherplatz benötigen. Diese Versionen nutzen während der Assemblierung etwa 7750 MB Speicherplatz.
Etwas weniger Speicherplatz verbraucht die nicht verknüpfte Atomic Version mit 2270 MB. Die restlichen Versionen, also die verknüpfte Atomic Version und die Farbversionen, benötigen jeweils um die 320 MB.
Außerdem fällt auf, dass der benötigte Speicherplatz bei allen Versionen unabhängig von der Anzahl der verwendeten Threads ist.

Nun dürfen wir aber nicht vergessen, dass wir ja für die Farbversionen einige Berechnungen im Vorfeld durchführen mussten. Dabei werden temporär etwas mehr als 8900 MB benötigt, die dann aber bis auf 200 MB wieder freigegeben werden. Wenn wir das in unserer Graphik berücksichtigen, dann schneiden die Farbversionen um einiges schlechter ab.
Diese Zusatzberechnungen der Farbversionen müssen zwar nur einmal durchgeführt werden, aber da die Farbversionen uns keine Verbesserung der Laufzeit bringen, ist das ein Punkt, der gegen die Verwendung einer Colorierung spricht.

Zusätzlich müssen wir beachten, dass im Vorfeld einmalige Berechnungen durchgeführt wurden, um die Verknüpfung von Berechnung und Assemblierung der Elementsteifigkeitsmatrizen zu erleichtern. Den Speicherplatzverbrauch dieser Berechnungen müssen wir ebenfalls berücksichtigen.

Betrachten wir nun die Ergebnisse. Mit einem Speicherplatzverbrauch von 9490 MB benötigen die Farbversionen insgesamt sogar mehr Speicherplatz als die Original Versionen. Jene Atomic Version bei der Berechnung und Assemblierung der Steifigkeitsmatrizen verknüpft ist, benötigt mit 560 MB mit Abstand am wenigsten Speicher.

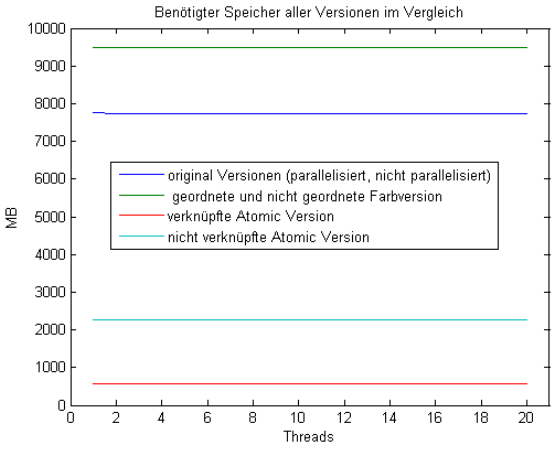


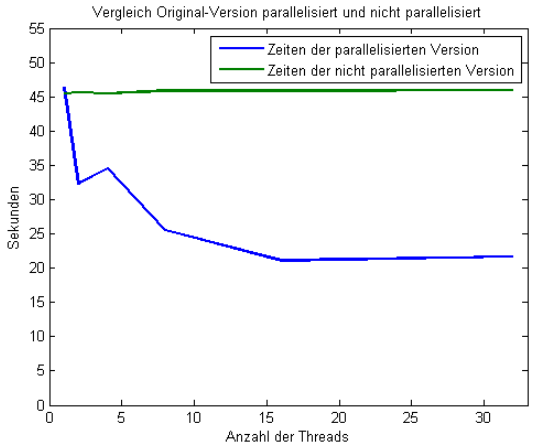
Abbildung 17: Speicherplatzverbrauch des Assemblierungsalgorithmus inklusive des Speicherplatzes der bei der Erzeugung des Zuordnungsvektors benötigt wird.

Die verknüpfte Atomic Version ist also nicht nur die schnellste unserer Versionen, sondern auch jene mit dem geringsten Speicherplatzbedarf.

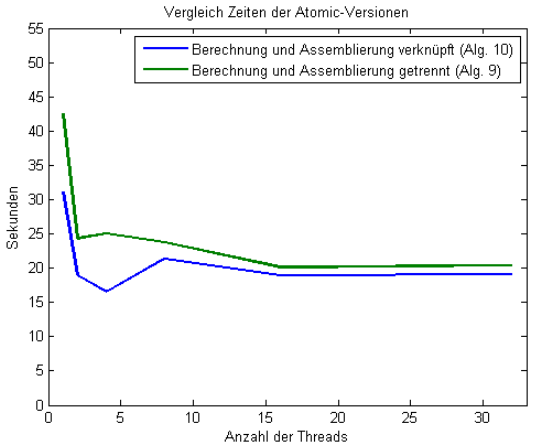
7 Laufzeitvergleich am Mephisto

Um unsere Laufzeitmessungen noch einmal zu verifizieren, möchten wir unseren Code zusätzlich noch auf anderen Prozessoren testen. Dazu verwenden wir zwei Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz Prozessoren mit jeweils 8 Prozessorkernen. Einer dieser Prozessorkerne kann wieder bis zu zwei Threads ausführen. Jeder der Prozessorkerne hat einen L1-Cache mit (32 [i] + 32 [d]) kB und eine L2-Cache mit 256 kB. Der von allen Prozessorkernen gemeinsam genützte L3-Cache bietet 20 MB Speicherplatz. Jeder der Prozessoren hat eine Speicherbandbreite von 51,2 GB/s. Außerdem stehen uns 260 GB im Arbeitsspeicher zur Verfügung. [3] [9]

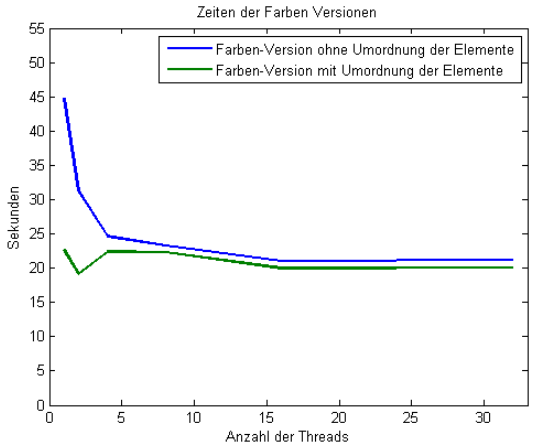
Da wir nun mehr Arbeitsspeicher nützen können als zuvor, dürfen wir es uns auch erlauben ein größeres Testbeispiel auszuprobieren. Wir verwenden deshalb ein Netz aus 53121770 Elementen mit 10976000 Knoten.
Wir verwenden außerdem die Compileroption -O2, die bewirkt, dass der Code hinsichtlich seiner Laufzeiten optimiert wird. Die Laufzeitmessungen ergeben nun folgendes Bild.



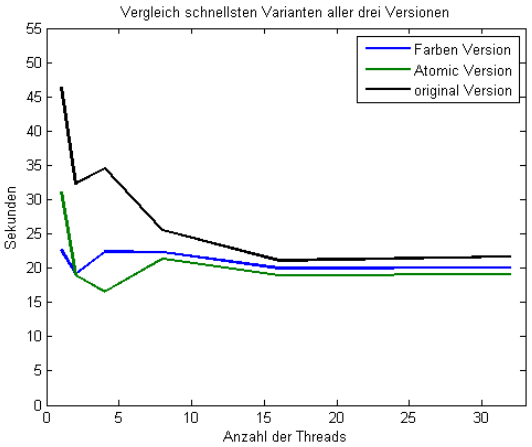
(a) Vergleich optimierte Original Versionen



(b) Vergleich der optimierten Atomic Versionen



(c) Vergleich der optimierten Farben Versionen



(d) Vergleich der schnellsten Varianten der optimierten Versionen

Die Graphiken zeigen uns, dass wir durch die Parallelisierung auch hier eine gute Verbesserung der Laufzeit erreichen können. Das beste Ergebnis wird in diesem Beispiel von der verknüpften Atomic Version unter der Verwendung von 4 Threads erreicht. Hier können wir die Laufzeit von ursprünglichen 45 Sekunden ohne Parallelisierung auf 16 Sekunden senken. Einfach blindlings so viele Threads wie möglich zu verwenden, macht allerdings nicht viel Sinn, da sich die Laufzeiten, wie wir hier gesehen haben, auch wieder verschlechtern können.

8 Erkenntnisse

Fassen wir zum Schluss noch einmal zusammen. Wir haben gesehen, dass man, wenn man die innere Struktur der aktuellen Prozessoren ausnützen möchte, seinen Code parallelisieren sollte. Leider ist das nicht immer möglich, aber die Berechnung und die Assemblierung von Elementsteifigkeitsmatrizen zu einer globalen Steifigkeitsmatrix in der Finiten Elemente Methode ist für eine Parallelisierung sehr gut geeignet.

Das einzige Problem das auftreten kann und über das man sich Gedanken machen sollte, sind Data Races. Zum Glück gibt es aber mehrere Möglichkeiten, wie man diese verhindern kann. Wir haben uns speziell für den Assemblierungsalgorithmus der FEM zwei dieser Möglichkeiten genauer angesehen.

Die verschiedenen Methoden haben wir ausgehend von einem existierenden, aber noch nicht parallelisierten Codestück implementiert und sie bezüglich ihrer Laufzeiten verglichen. Dabei ist uns aufgefallen, dass auch die Speicherstruktur einen wesentlichen Einfluss auf die Laufzeiten haben kann.

Ausgehend von unseren Laufzeitmessungen und den Speicherplatzmessungen kommen wir nun zu folgenden Erkenntnissen:

- Es ist geschickter weniger, dafür aufwendigere Rechnungen zu parallelisieren, als viele kurze und einfache Rechnungen.
- Zwischenergebnisse in kurzen, lokalen Vektoren zu speichern, statt in langen, globalen Vektoren spart nicht nur Speicherplatz, sondern auch Zeit. (Die Atomic Versionen sind schneller als die original Versionen)
- Wenn möglich Berechnungen, die Ergebnisse von einander benötigen, auch gleich hintereinander durchführen. (Die verknüpfte Atomic Version ist schneller als die nicht verknüpfte Atomic Version)
- Daten, die direkt nacheinander benötigt werden, sollten auch nebeneinander gespeichert sein. (Die umgeordnete Farben Version ist schneller als die nicht geordnete Farbenversion)
- Ein Netz zu colorieren, um Elemente zu finden die ohne Gefahr von Data Races gleichzeitig parallelisiert werden können, ist sehr zeitintensiv und benötigt viel Speicherplatz. Da die Laufzeiten der Farbenversionen nicht besser sind als jene der schnellsten Atomic Version ist das im konkreten Fall nicht zu empfehlen.
- Sind Data Races nur selten zu befürchten, dann ist # pragma omp atomic demnach bestens geeignet um Data Races auszuschließen.
- Die Möglichkeiten der Parallelisierung sind begrenzt, auch die Verwendung von beliebig vielen Threads wird die Laufzeit nur um ein bestimmtes Maß verbessern. (Das Laufzeitenminimum wird bei unseren Implementierungen meistens schon bei

der Verwendung zwischen 4 und 8 Threads erreicht. Die Ursache davon liegt in der beschränkten Speicherbandbreite.)

Alles in Allem können wir sagen, dass wir auf beiden Maschinen mit der "verknüpften Atomic Version" (Algorithmus 10 auf Seite 31) die besten Ergebnisse erzielt haben. Das Laufzeitenminimum wurde bereits bei einer Verwendung von zwischen 4 und 8 Threads erreicht. Je nach Größe des Beispiels, blieben die Zeiten bei mehr als 4 Threads mehr oder weniger konstant, oder haben sich sogar wieder verschlechtert.
Wir haben also sowohl in Bezug auf die Laufzeiten, als auch in Bezug auf den benötigten Speicher mit der "verknüpften Atomic Version", bei der Verwendung von 4 Threads, die besten Ergebnisse erziehlt.

Literatur

[1] Shannon Cepeda. *What you Need to Know about Prefetching.* URL: https://software.intel.com/en-us/blogs/2009/08/24/what-you-need-to-know-about-prefetching.

[2] Thomas H. Cormen u. a. *Algorithmen - Eine Einführung [Gebundene Ausgabe].* Oldenbourg Wissenschaftsverlag; Auflage: überarbeitete und aktualisierte Auflage, 2010. ISBN: 3486590022. URL: http://www.amazon.de/Algorithmen-Eine-Einf\%C3\%BChrung-Thomas-Cormen/dp/3486590022.

[3] Cpu.world. *Intel Xeon E5-2650 - CM8062100856218 / BX80621E52650.* URL: http://www.cpu-world.com/CPUs/Xeon/Intel-XeonE5-2650.html.

[4] Cpu.world. *Intel Xeon E5-2660 v2 - CM8063501452503 / BX80635E52660V2.* URL: http://www.cpu-world.com/CPUs/Xeon/Intel-XeonE5-2660v2.html.

[5] Th. Emden-Weinert u. a. *Einführung in Graphen und Algorithmen.* URL: http://www.or.uni-bonn.de/~hougardy/paper/ga.pdf.

[6] Gcc.gnu.org. *Optimize Options - Using the GNU Compiler Collection (GCC).* URL: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[7] Annika Hagemeier. *OpenMP Teil1: Einführung in OpenMP.* URL: http://www.fz-juelich.de/ias/jsc/EN/AboutUs/Organisation/ComputationalScience/Simlabs/slms/teaching/docs-parallel-programming/OpenMP-Slides.pdf?__blob=publicationFile.

[8] Intel. *ARK — Intel® Xeon® Processor E5-2660 v2 (25M Cache, 2.20 GHz).* URL: http://ark.intel.com/de/products/75272/Intel-Xeon-Processor-E5-2660-v2-25M-Cache-2_20-GHz.

[9] Intel. *Intel® Xeon® Processor E5-2650 (20M Cache, 2.00 GHz, 8.00 GT/s Intel® QPI).* URL: http://ark.intel.com/de/products/64590/Intel-Xeon-Processor-E5-2650-20M-Cache-2_00-GHz-8_00-GTs-Intel-QPI?q=E5-2650.

[10] Michael Jung und Ulrich Langer. *Methode der finiten Elemente für Ingenieure*. Wiesbaden: Vieweg+Teubner Verlag, 2001. ISBN: 978-3-519-02973-1. DOI: 10.1007/978-3-663-10785-9. URL: http://link.springer.com/10.1007/978-3-663-10785-9.

[11] Arndt Prof. Dr. Bode. *Multicore-Architekturen - GI - Gesellschaft für Informatik e.V.* URL: http://www.gi.de/service/informatiklexikon/detailansicht/article/multicore-architekturen.html.

[12] Thomas Rauber und Gudula Rünger. *Multicore:: Parallele Programmierung*. 2007. ISBN: 354073113X. URL: http://books.google.at/books/about/Multicore.html?id=XKTBQ3SAlE8C\&pgis=1.

[13] Peter Tittmann. *Graphentheorie: eine anwendungsorientierte Einführung ; mit zahlreichen Beispielen und 80 Aufgaben*. 2003. ISBN: 3446223436. URL: http://books.google.de/books/about/Graphentheorie.html?hl=de\&id=EV6qGBCuSe4C\&pgis=1.

[14] Volker Turau. *Algorithmische Graphentheorie*. Oldenbourg Verlag, 2009, S. 445. ISBN: 348659057X. URL: http://books.google.com/books?id=HB0kxef_s10C\&pgis=1.

[15] Tomás Zegard. *Greedy Graph Coloring and Parallel FEM Assembly*. 2010. URL: http://paulino.cee.illinois.edu/education_resources/GreedyColoring/GreedyColoring.pdf.