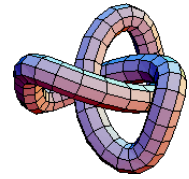




TECHNISCHE UNIVERSITÄT CHEMNITZ
FAKULTÄT FÜR MATHEMATIK



Optimierung von parallel-adaptiven FE-Rechnungen auf Master-Slave-Basis

An der Fakultät für Mathematik der Technischen Universität Chemnitz eingereichte

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Wirtschaftsmathematiker

(Dipl.-Math. oec.)

Vorgelegt von

Jens Rückert

geboren am 06. Februar 1981 in Erlabrunn

Betreuer: Prof. Dr. rer. nat. habil. Arnd Meyer

Chemnitz, den 19. August 2007

Inhaltsverzeichnis

Einleitung	2
1 Grundlagen der parallelen FEM	4
1.1 Grundlagen des parallelen Rechnens	4
1.2 Funktionenräume	6
1.3 Die Finite Elemente Methode	6
1.4 Domain Decomposition - Grundlegende Gebietszerlegung	9
2 Farming-Konzept	15
2.1 Realisierung	16
2.2 Datenstrukturen	17
2.3 Parallelisierung mittels Farming - Konzept	18
2.4 Kritik und Ansatzpunkte für die Optimierung	19
2.4.1 Kritik	19
2.4.2 Optimierungsmöglichkeiten im Schritt (2)	21
2.4.3 Optimierungsmöglichkeiten im Schritt (3)	21
3 Optimierung des Farming-Konzeptes	25
3.1 Verwendung von MPI-Kommunikationsroutinen	29
3.2 Einführung einer neuen Kommunikationsstruktur	30
3.3 Indizierte Addition des Lösungsvektors	33
3.4 Hypercube als neue Kommunikationsstruktur	39
4 Betrachtungen zur optimalen Anzahl von Prozessoren	44
4.1 Numerischer Aufwand für das Farming-Konzept	44
4.2 Optimale Wahl der Prozessoranzahl	46
5 Zusammenfassung	51
Verzeichnisse	53
Abbildungsverzeichnis	53
Tabellenverzeichnis	54
Thesen	56

Einleitung

Mit der rasanten Entwicklung der Computerindustrie in den letzten 20 Jahren hat die numerische Simulation bei der Behandlung ingenieurwissenschaftlicher Probleme sehr stark an Bedeutung gewonnen. Mit der Verbesserung der Hardware können komplexere Sachverhalte simuliert und wirklichkeitsnah dargestellt werden. Neben der Komplexität der Probleme spielt dabei auch die Laufzeit einer Simulation eine große Rolle. In der Industrie und Forschung ist man bestrebt eine möglichst kurze Berechnungszeit zu erreichen, ohne dabei Einbußen bei der Genauigkeit hinnehmen zu müssen.

Viele dieser ingenieurwissenschaftlichen Fragestellungen lassen sich auf Randwertprobleme bei partiellen Differentialgleichungen zurückführen, wie beispielsweise die Simulation der Wirkung von verschiedenen Kräften auf Bauteile, die eine Verformung hervorrufen oder die Wärmeausbreitung in einem Werkstück bei der thermischen Bearbeitung. Gesucht wird dabei immer eine Funktion $\varphi(x)$, die auf einem vorgegebenen Berechnungsgebiet $\Omega \subset \mathbb{R}^d$ eine partielle Differentialgleichung löst und gleichzeitig weitere Bedingungen auf dem Rand $\Gamma := \partial\Omega$ von Ω erfüllt.

Zur Lösung dieser Randwertprobleme existieren verschiedene Klassen von mathematischen Methoden. Eine dieser Klassen benutzt zur Lösung des Problems die Diskretisierung des gesamten Berechnungsgebietes Ω , zu der auch die Finite Elemente Methode (FEM) zu zählen ist.

An der TU Chemnitz, Fakultät für Mathematik, wurde ein Programm zur Simulation von Verformungen infolge von Kräfteinwirkungen erstellt, welches als mathematische Grundlage die FEM benutzt. Dieses Programm ist für die Nutzung auf einem Parallelrechner konzipiert, wie zum Beispiel dem *CHIC* an der TU Chemnitz.

Ziel dieser Diplomarbeit ist eine Laufzeitverbesserung des vorhandenen Programmes, ohne dabei einen Verlust bei der Genauigkeit der Ergebnisse zu erleiden. Dazu gliedert sie sich folgendermaßen. Das erste Kapitel beschäftigt sich mit den Grundlagen der parallelen FEM, wobei verschiedene Begriffe für die Nutzung paralleler Systeme eingeführt werden. Zunächst wird näher auf die Theorie der Funktionenräume eingegangen, da diese für die FEM benötigt werden. Anschließend wird die klassische Parallelisierungsidee, die *Domain Decomposition*, vorgestellt. Bei der *Domain Decomposition* handelt es sich um eine Methode, bei der das Berechnungsgebiet in kleinere disjunkte Teilgebiete untergliedert wird. Die Anzahl dieser Teilgebiete entspricht dabei der Anzahl der verwendeten Prozessoren. Jeder Prozessor berechnet dann die Lösung der partiellen Differentialgleichung auf dem ihm zugeordneten Teilgebiet.

Im Kapitel zwei folgt die Darstellung eines völlig neuen Ansatzes zur Parallelisierung der FEM, dem *Farming Konzept*. Im Unterschied zur *Domain Decomposition* werden keine disjunkten Teilgebiete gebildet, auf denen die Prozessoren die Lösung der Differentialgleichung berechnen,

sondern es werden die finiten Elemente gleichmäßig auf die Prozessoren verteilt und die Lösung der Differentialgleichung wird elementweise berechnet. Im weiteren Verlauf dieser Arbeit wird auf die Realisierung dieser Methode eingegangen und die Vor- sowie Nachteile werden herausgearbeitet.

Mit der Optimierung des *Farming Konzepts* beschäftigt sich ausführlich das nächste Kapitel, wobei sowohl die programmtechnische Umsetzung als auch die numerischen Ergebnisse vorgestellt werden. Wie bereits erwähnt, spielen bei Computersimulationen die Laufzeiten eines Programms eine große Rolle. Deshalb werden in diesem Kapitel die Laufzeiten der Optimierungen miteinander verglichen, um Beschleunigungen oder auch eine Verlangsamung gegenüber der vorherigen Implementierung oder auch der Ausgangsimplementierung darzustellen.

Der letzte Teil dieser Arbeit beschäftigt sich mit dem numerischen Aufwand des PCG-Verfahrens, das zur Lösung der partiellen Differentialgleichung benutzt wird. Dabei stellt sich zudem die Frage nach der optimalen Prozessoranzahl. Deshalb wurde untersucht, ob sich die Laufzeiten des Programmes bei einer wachsenden Prozessoranzahl verbessern.

Kapitel 1

Grundlagen der parallelen FEM

Ausgangspunkt dieses Kapitels bilden die Grundlagen des parallelen Rechnens. Dem Leser sollen dabei zuerst die in dieser Diplomarbeit betrachteten Größen näher gebracht werden. Im weiteren Verlauf wird die Finite Elemente Methode, ein numerisches Verfahren zur Lösung partieller Differentialgleichungen, erklärt. Schließlich wird die *Domain Decomposition*, das klassische Verfahren zur Parallelisierung der Finiten Elemente Methode (FEM), beschrieben.

1.1 Grundlagen des parallelen Rechnens

Gegeben sei ein Algorithmus A . Nach dessen Parallelisierung lassen sich dann der Speedup und die parallele Effizienz ermitteln. Damit können die Beschleunigung und die Effizienz der Parallelisierung des Algorithmus A auf p Prozessoren gegenüber dem Algorithmus auf einem Prozessor beschrieben werden.

Definition 1.1. (*Speedup*) Es sei mit $T(p)$ die Zeit bezeichnet, die zur Berechnung des Algorithmus A durch p Prozessoren benötigt wird und $T(1)$ die benötigte Zeit zur Berechnung des Algorithmus A durch einen Prozessor. Dann ergibt sich der Speedup $S(p)$ für den Algorithmus A aus

$$S(p) = \frac{T(1)}{T(p)}.$$

Definition 1.2. (*Parallele Effizienz*) Die Effizienz eines Algorithmus A lässt sich berechnen durch

$$Eff(p) = \frac{1}{p} S(p).$$

Folgerung 1.1. Bezeichne $T(1)$ die Zeit, die der schnellste Algorithmus benötigt, um ein Problem auf einem Prozessor zu lösen und $T(p)$ die Zeit, die für den Algorithmus A auf p Prozessoren zur Lösung desselben Problems gebraucht wird. Dann gilt

- $S(p) = \frac{T(1)}{T(p)} \leq p$ und
- $Eff(p) = \frac{1}{p} S(p) \leq 1.$

Satz 1.1 (Amdahlsches Gesetz). *Wir betrachten einen Algorithmus A und $T(p)$ die Zeit, die zur Berechnung des Algorithmus A mit p Prozessoren benötigt wird. Bezeichne weiterhin $sT(1)$ den sequentiellen Anteil des Algorithmus A , das heißt den nicht parallelisierbaren Anteil, und dementsprechend $(1 - s)T(1)$ den parallelisierbaren Anteil, dann gilt für den Speedup*

$$S(p) = \frac{1}{s + \frac{1-s}{p}}.$$

Der Speedup hängt bei steigender Prozessoranzahl also immer mehr vom sequentiellen Anteil ab.

Beweis. Es sei mit N die Anzahl der Operationen bezeichnet, die im Algorithmus A zu lösen sind und τ sei die Zeit, die für eine Operation gebraucht wird. Zudem sei der Anteil $(1 - s)$ mit dem Parallelisierungsgrad p parallelisierbar. Die Zeit, die benötigt wird, um den Algorithmus A auf einem Parallelrechner mit p Prozessoren zu lösen, setzt sich aus dem nicht parallelisierbaren Anteil mit der Laufzeit $sN\tau$ und der Zeit des parallelisierbaren Anteils zusammen. Es gilt

$$T(p) = sN\tau + (1 - s)\frac{N\tau}{p}.$$

Folglich erhält man für den Speedup

$$\begin{aligned} S(p) &= \frac{T(1)}{T(p)} \\ &= \frac{N\tau}{sN\tau + (1 - s)\frac{N\tau}{p}} \\ &= \frac{1}{s + \frac{1-s}{p}}. \end{aligned}$$

□

Folgerung 1.2. *Unabhängig von der Anzahl der Prozessoren p wird der Speedup durch den sequentiellen Anteil begrenzt, denn es gilt*

$$S(p) \leq \frac{1}{s}.$$

Bemerkung 1.1. *Bei der Parallelisierung von Algorithmen tritt immer ein gewisser Zeitverlust V auf. Der Speedup ist also nicht nur abhängig vom nicht parallelisierbaren Anteil des Algorithmus, sondern auch von diesem Zeitverlust infolge von Kommunikation zwischen den einzelnen Prozessoren. Dieser Zeitverlust ist in der Folgerung (1.2) noch gar nicht beachtet, so dass der Speedup noch weiter sinkt. Es gilt dann*

$$S(p) < \frac{1}{s}.$$

Definition 1.3 (Hypercube). *Als Hypercube wird eine Rechnerarchitektur mit 2^n Prozessoren bezeichnet. Dabei ist n die Dimension des Hypercubes. Ein Hypercube lässt sich darstellen als eine Menge von gerichteten Graphen $H_n = (V_n, E_n)$ mit der Vertexmenge V_n , die Menge aller Prozessoren mit $|V_n| = 2^n = p$, sowie der Menge der Kanten E_n , die jeweils genau zwei benachbarte Prozessoren miteinander verbinden. Diese Prozessoren unterscheiden sich dann immer in genau einem Bit.*

1.2 Funktionenräume

Die Diskretisierung bei der Finiten Elemente Methode erfolgt auf den sogenannten Sobolevräumen. Die in dieser Arbeit verwendeten Sobolevräume bauen auf dem Funktionenraum L_2 auf.

Definition 1.4. Sei $\Omega \subset \mathbb{R}^d$ offen. Dann wird mit

$$L^2(\Omega) := \left\{ f : \Omega \rightarrow \mathbb{R} \mid \int |f|^2 dx < \infty \right\}$$

der Raum aller quadratisch integrierbaren Funktionen über Ω definiert.

Satz 1.2. Versieht man den Raum $L^2(\Omega)$ mit dem Skalarprodukt

$$(u, v)_{L^2(\Omega)} := \int_{\Omega} u(x) v(x) dx$$

und der Norm

$$\|u\|_{L^2(\Omega)} := (u, u)_{L^2(\Omega)}^{\frac{1}{2}},$$

so wird der $L^2(\Omega)$ zu einem Hilbertraum.

Definition 1.5 (Sobolevräume). Sei $\Omega \subset \mathbb{R}^d$ beschränkt und $l \geq 0$. Dann wird der Sobolevraum zur Ordnung l definiert durch

$$H^l(\Omega) := \{u \in L^2(\Omega) : \partial^\alpha u \in L^2(\Omega), \forall \alpha \leq l\},$$

wobei $\partial^\alpha u$ die schwache partielle Ableitung von u zum Multiindex α darstellt.

Satz 1.3. Versehen mit dem Skalarprodukt

$$(u, v)_{H^l} := \sum_{|\alpha| \leq l} \int_{\Omega} (\partial^\alpha u(x), \partial^\alpha v(x)) dx$$

und der folgenden Norm

$$\|u\|_{H^l}^2 := (u, u)_{H^l}$$

wird H^l zu einem Hilbertraum.

1.3 Die Finite Elemente Methode

Bei der Finiten Elemente Methode, kurz FEM, handelt es sich um ein Verfahren zur näherungsweisen Lösung Partieller Differentialgleichungen mittels Diskretisierung eines Gebietes Ω .

Das Verfahren wird dem Leser anhand eines Beispielles näher gebracht. Für weitergehende Erklärungen sei auf das Buch [3, D. Braess] verwiesen.

Die Ausführungen zur Diskretisierung wurden aus dem Buch [2, Hoffmann, Meyer] übernommen.

Beispiel 1.1. (Laplace Gleichung)

$$\begin{aligned} -\Delta u &= f & |_{\Omega} & \Omega \subset \mathbb{R}^2 \\ \text{Randbedingungen : } u &= g & |_{\Gamma_D} & \text{Dirichlet-Rand} \\ \frac{\partial u}{\partial n} &= 0 & |_{\Gamma_N} & \text{Neumann-Rand} \end{aligned}$$

Dabei soll $\Gamma_D \cup \Gamma_N = \partial\Omega$ gelten.

Die partielle Differentialgleichung wird zur weiteren Berechnung in die Bilinearform überführt, d.h. wir suchen ein

$$\begin{aligned} u &\in H^1(\Omega) & \text{mit } u &= g \text{ } |_{\Gamma_D} \text{ und} \\ a(u, v) &= \langle f, v \rangle & \text{für alle } v &\in H_0^1(\Omega) := \{v \in H^1(\Omega) : v = 0 \text{ } |_{\Gamma_D}\}. \end{aligned}$$

Für das Beispiel gilt dann

$$\begin{aligned} a(u, v) &= \int_{\Omega} \nabla u \cdot \nabla v \, d\Omega \\ \langle f, v \rangle &= \int_{\Omega} f \cdot v \, d\Omega. \end{aligned}$$

Um eine Diskretisierung zu erreichen, beschränkt man sich auf einen Teilraum $\mathbb{V} \subset H^1(\Omega)$. Die Bedingung $u = g$ auf dem Dirichlet-Rand Γ_D sei in \mathbb{V} erfüllbar und im Weiteren für die Näherungen immer gefordert. Man erhält also folgende Formulierung. Gesucht wird ein

$$\begin{aligned} u &\in \mathbb{V} \\ \text{mit } a(u, v) &= \langle f, v \rangle \text{ für alle } v \in \mathbb{V}_0 = \mathbb{V} \cap H_0^1(\Omega). \end{aligned}$$

Nun existiert eine Basis Φ in \mathbb{V} , bestehend aus stetigen, stückweise linearen Funktionen φ_i deren Träger um einen Knoten einer Dreiecksvernetzung des Gebietes Ω so existiert, dass gilt

$$\varphi_i(x_j) = \delta_{i,j}.$$

Wir erhalten also die folgenden Zusammenhänge

$$\begin{aligned} \Phi &:= [\varphi_1, \varphi_2, \dots, \varphi_N], \\ \mathbb{V} &:= \text{span}(\Phi). \end{aligned}$$

Daraus folgt

$$u = \sum_{i=1}^N u_i \varphi_i = \Phi \underline{u} \quad \text{mit } \underline{u} = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_N \end{pmatrix}.$$

Für die Bilinearform gilt nun

$$a(\Phi \underline{u}, \varphi_i) = \langle f, \varphi_i \rangle \iff \sum_{j=1}^N u_j a(\varphi_j, \varphi_i) = \langle f, \varphi_i \rangle \text{ für alle } i : x_i \notin \Gamma_D \text{ und} \\ u_j = g(x_j) \text{ für alle } j : x_j \in \Gamma_D.$$

Mit den Bezeichnungen

$$b_i := \langle f, \varphi_i \rangle \quad \forall i \\ K := [k_{i,j}]_{i,j=1}^N \\ k_{i,j} := a(\varphi_j, \varphi_i)$$

erhalten wir also das lineare Gleichungssystem

$$\mathbf{K} \underline{u} = \underline{b}. \quad (1.1)$$

Dieses Gleichungssystem muss aufgrund der Größe der Systemmatrix K über Iterationsverfahren, z.B. dem PCG, gelöst werden.

Das Kapitel endet mit der Darstellung des Algorithmus für die PCG-Methode zur Lösung des Linearen Gleichungssystems 1.1.

Algorithmus 1.1. (PCGM)

Start :

$$\begin{array}{lll} \underline{u} & & \text{Startvektor} \\ \underline{r} & = & K \underline{u} - \underline{b} \quad \text{Residuum} \\ \underline{w} & = & C^{-1} \underline{r} \quad \text{Vorkonditioniertes Residuum} \\ \underline{q} & = & \underline{w} \quad \text{Suchrichtung} \\ \gamma & = & \underline{w}^T \underline{r} \end{array}$$

Iteration :

$$\begin{array}{ll} \underline{v} & := K \underline{q} \\ \delta & := \underline{v}^T \underline{q} \\ \alpha & := -\gamma / \delta \\ \underline{u} & := \underline{u} + \alpha \underline{q} \\ \underline{r} & := \underline{r} + \alpha \underline{v} \\ \underline{w} & := C^{-1} \underline{r} \\ \hat{\gamma} & := \underline{w}^T \underline{r} \\ \beta & := \hat{\gamma} / \gamma \\ \underline{q} & := \underline{w} + \beta \underline{q} \\ \gamma & := \hat{\gamma} \end{array}$$

1.4 Domain Decomposition - Grundlegende Gebietszerlegung

In diesem Abschnitt werden die Zerlegung des vorgegebenen Gebietes Ω und die klassische Parallelisierungsidee der FE-Rechnung, die Domain Decomposition (DD), näher betrachtet.

Das Verfahren und dessen Voraussetzungen werden ausführlich in dem Buch [2, Hoffmann, Meyer] behandelt.

Aus der Definition des Gleichungssystems (1.1) ergeben sich zwei Typen von Vektoren in der DD - Idee.

Typ I : Vektoren, die „Knotenwerte“ enthalten, z.B. \underline{u}

Typ II : Vektoren und Matrizen, die Funktionalwerte enthalten, z.B. \underline{b} , \underline{r} oder K

Aus der Tatsache, dass diese Funktionalwerte (Typ II) alle als Integral über Ω dargestellt werden können, lassen sich diese Vektoren und Matrizen auf verschiedene Prozessoren verteilt als partielle Summen errechnen.

Zur näheren Betrachtung müssen noch einige Bedingungen an unser Gebiet Ω gestellt werden.

- Wir definieren Teilgebiete Ω_s von Ω durch

$$\bar{\Omega} = \bigcup_{s=1}^p \bar{\Omega}_s$$

$$\Omega_{s_1} \cap \Omega_{s_2} = \emptyset \quad \forall s_1 \neq s_2,$$

das heißt unser Gebiet Ω wird in disjunkte Teilgebiete unterteilt.

- $\Gamma_c = \bigcup_s \partial\Omega_s$ wird Koppelrand genannt.
- Es wird eine konforme Vernetzung gefordert, das heißt es gibt keine „hanging nodes“ in jedem Teilgebiet und über den Koppelrand Γ_c hinweg.

Für unser Beispiel (1.1) der Laplace-Gleichung ergibt sich dann

$$\begin{aligned} a(u, v) &= \int_{\Omega} \nabla u \cdot \nabla v \, d\Omega \\ &= \sum_s \int_{\Omega_s} \nabla u \cdot \nabla v \, d\Omega \\ &= \sum_s a_s(u, v). \end{aligned}$$

Nun wird noch eine besondere Nummerierung der Knoten eingeführt.

$$\left(\underbrace{x_1, \dots, x_{N_c}}_{\in \Gamma_c} \mid \underbrace{x_{N_c+1}, \dots, x_{N_c+N_{I,1}}}_{\in \text{int}(\Omega_1)} \mid \dots \mid \underbrace{\dots, x_N}_{\in \text{int}(\Omega_p)} \right)$$

Zuerst werden die Knoten auf dem Koppelrand Γ_c gezählt, dann die inneren Knoten auf Ω_1 bis Ω_p .

Entsprechend zerfällt dann \mathbb{V} in folgende Teilräume

$$\begin{aligned}\mathbb{V} &= \text{span}(\Phi) \\ \mathbb{V}_{I,1} &= \text{span}(\varphi_{N_c+1}, \dots, \varphi_{N_c+N_{I,1}}) \\ \mathbb{V}_{I,2} &= \text{span}(\varphi_{N_c+N_{I,1}+1}, \dots, \varphi_{N_c+N_{I,1}+N_{I,2}}), \dots \text{ usw.} \\ \mathbb{V}_c &= \text{span}(\varphi_1, \dots, \varphi_{N_c}).\end{aligned}$$

Dabei gilt

$$\begin{aligned}N_c &\hat{=} \text{Anzahl Knoten auf dem Koppelrand } \Gamma_c \\ N_{I,s} &\hat{=} \text{Anzahl Knoten im Inneren von } \Omega_s.\end{aligned}$$

Jetzt lassen sich die Werte eines lokalen Vektors

$$\underline{\mathbf{b}}_s = (b_i)_{x_i \in \bar{\Omega}_s} \in \mathbb{R}^{N_s}$$

auf dem Prozessor P_s unabhängig von den anderen Prozessoren errechnen, weil dieser sämtliche Informationen über $\bar{\Omega}_s$ enthält.

Die rechte Seite des linearen Gleichungssystems (1.1) erhält man dann durch

$$\underline{\mathbf{b}} = \sum_{s=1}^p H_s^T \underline{\mathbf{b}}_s, \quad (1.2)$$

mit einer speziellen $(N_s \times N)$ - Booleschen Matrix H_s . Falls der i -te Knoten im globalen Netz dem j -ten Knoten von $\bar{\Omega}_s$ entspricht, so ist $(H_s)_{ji} = 1$, sonst Null. Wie in der Formel (1.2) lassen sich alle Funktionalwerte über globale Summierung berechnen. So auch die Matrix

$$K = \sum_{s=1}^p H_s^T K_s H_s, \quad (1.3)$$

wobei K_s , die lokale Steifigkeitsmatrix, die zu Ω_s gehört, auf dem Prozessor P_s unabhängig von den anderen Prozessoren errechnet wird.

Die Vektoren vom Typ I lassen sich nicht durch eine Summationsformel wie (1.2) berechnen. Hier gilt

$$\underline{\mathbf{u}}_s = H_s \underline{\mathbf{u}}. \quad (1.4)$$

Deshalb muss der Prozessor P_s den Abschnitt von $\underline{\mathbf{u}}$ speichern, der zu den Knoten von $\bar{\Omega}_s$ gehört. Wie man in dem Algorithmus 1.1 sehen kann, werden die Vektoren eines Typs ausschließlich durch Vektoren des gleichen Typs verändert. Dies macht bei der Parallelisierung eine Kommunikation der Prozessoren untereinander für diese Berechnungen überflüssig.

Folgerung 1.3. *Wie in Formel (1.2) lassen sich alle Daten des Typs II berechnen:*

$$\begin{aligned}
 \underline{v} &= K \underline{q} \\
 &= \sum H_s^T K_s (H_s \underline{q}) \\
 &= \sum H_s^T (K_s \underline{q}_s) \\
 &= \sum H_s^T \underline{v}_s
 \end{aligned} \tag{1.5}$$

mit $\underline{v}_s = K_s \underline{q}_s$. Analog wird das lokale Residuum \underline{r}_s berechnet.

$$\begin{aligned}
 \underline{r} &= K \underline{u} - \underline{b} \\
 &= \sum H_s^T (K_s \underline{u}_s - \underline{b}_s) \\
 &= \sum H_s^T \underline{r}_s
 \end{aligned}$$

mit $\underline{r}_s = K_s \underline{u}_s - \underline{b}_s$.

Zusätzlich gilt

$$\begin{aligned}
 \underline{w}^T \underline{r} &= \underline{w}^T \left(\sum H_s^T \underline{r}_s \right) \\
 &= \sum (H_s \underline{w})^T \underline{r}_s \\
 &= \sum_{s=1}^p (\underline{w}_s^T \underline{r}_s)
 \end{aligned}$$

sowie

$$\underline{q}^T K \underline{q} = \underline{q}^T \left(\sum H_s^T (K_s \underline{q}_s) \right) = \sum \underline{q}_s^T K_s \underline{q}_s,$$

wodurch man δ berechnen kann.

Jetzt lässt sich der Algorithmus für die parallele PCG-Methode aufschreiben.

Algorithmus 1.2. (parallel PCGM)Start :

$$\begin{aligned}
\underline{u}_s &= H_s \underline{u} && \text{lokaler Startvektor entsprechend Formel (1.4)} \\
\underline{r}_s &= K_s \underline{u}_s - \underline{b}_s && \text{lokales Residuum} \\
\underline{r} &= \sum \underline{r}_s && \text{globales Residuum} \\
\underline{w}_s &:= H_s \underline{w} && \text{lokaler Teil des vorkonditionierten Residuums (aus } \underline{w} := C^{-1} \underline{r} \text{)} \\
\underline{q}_s &= \underline{w}_s && \text{Suchrichtung} \\
\gamma_s &:= \underline{w}_s^T \underline{r}_s \\
\gamma &= \sum \gamma_s
\end{aligned}$$

Iteration :

$$\begin{aligned}
1. \quad \underline{v}_s &:= K_s \underline{q}_s \\
2. \quad \delta_s &:= \underline{v}_s^T \underline{q}_s \\
3. \quad \delta &:= \sum \delta_s \\
4. \quad \alpha &:= -\gamma / \delta \\
5. \quad \underline{u}_s &:= \underline{u}_s + \alpha \underline{q}_s \\
6. \quad \underline{r}_s &:= \underline{r}_s + \alpha \underline{v}_s \\
7. \quad \underline{w}_s & \text{ aus } C \underline{w} = \underline{r} \text{ berechnen} \\
8. \quad \hat{\gamma}_s &:= \underline{w}_s^T \underline{r}_s \\
9. \quad \hat{\gamma} &:= \sum \hat{\gamma}_s \\
10. \quad \beta &:= \hat{\gamma} / \gamma \\
11. \quad \underline{q}_s &:= \underline{w}_s + \beta \underline{q}_s \\
12. \quad \gamma &:= \hat{\gamma}
\end{aligned}$$

Bemerkung 1.2. Eine Verbindung zwischen den Teilgebieten Ω_s wird nur im 7. Schritt notwendig, alle anderen Schritte sind lokale Berechnungen oder die Summe eines Wertes über alle Prozessoren. Wird Schritt sieben besonders betrachtet, zum Beispiel bei $C = I$, so gilt

$$\underline{w}_s = H_s \underline{w} = H_s \underline{r} = H_s \left(\sum_{j=1}^p H_j^T \underline{r}_j \right) = \underline{r}_s + \sum_{j \neq s} H_s H_j^T \underline{r}_j. \quad (1.6)$$

Hierbei definiert $H_s H_j^T \underline{r}_j$ den Teil der Daten von Prozessor P_j , der zu Prozessor P_s gesendet werden muss, um \underline{w}_s zu erzeugen.

Bemerkung 1.3. Ein guter Vorkonditionierer erfüllt folgende Eigenschaften :

- (A) Die arithmetischen Operationen für Schritt 7 sollten billig sein (proportional zur Anzahl der Unbekannten).
- (B) Die Konditionszahl $\kappa(C^{-1}K)$ ist klein, unabhängig von dem Diskretisierungsparameter h in der Netzverfeinerung oder nur langsam wachsend für $h \rightarrow 0$, wie etwa $\mathcal{O}(|\ln h|)$.
- (C) Die Kommunikation zwischen den Prozessoren zur Berechnung des 7. Schrittes sollte so gering wie möglich sein (am besten genau ein Datenaustausch pro Iterationsschritt).

Bemerkung 1.4. Bei der Einheitsmatrix I als Vorkonditionierer sind die Bedingungen (A) und (C) voll erfüllt, jedoch (B) nicht. Für die Konditionszahl gilt

$$\kappa(I^{-1}K) = \mathcal{O}(h^{-2}).$$

Die Anzahl der Iterationen würde mit h^{-1} sehr schnell anwachsen.

Bemerkung 1.5. Betrachten wir den Algorithmus 1.2. Für diesen Algorithmus erhalten wir eine parallele Effizienz derart

$$Eff(p) \sim 1 - \mathcal{O}(h).$$

Im folgenden Beweis werden Konstanten immer mit c bezeichnet. Es ist zu beachten, dass sich diese Konstanten von Formel zu Formel oder auch innerhalb einer Formel unterscheiden können.

Beweis. Betrachten wir also ein Problem mit N Unbekannten. Es seien die Teilgebiete Ω_s so gewählt, dass in jedem Teilgebiet eine ähnliche Anzahl von Verfeinerungen zu berechnen ist. Dann gilt für den Algorithmus 1.2

$$\begin{aligned} T(1) &= cN & \text{und} \\ T(p) &= \frac{cN}{p} + V, \end{aligned}$$

mit einem gewissen Zeitverlust V .

Desweiteren gilt für unseren Schrittweitenparameter h in der Diskretisierung $N \sim h^{-d}$.

Der Zeitverlust V bei der Berechnung des Algorithmus mit p Prozessoren ist insbesondere durch die Kommunikation der Prozessoren untereinander zu erklären. Diese Kommunikation ist notwendig für die Knoten auf dem Koppelfrand (im $2D$) bzw. auf der Koppelfläche (im $3D$). Sei also N_{Koppel} die Anzahl der Knoten auf der Koppelfläche, dann gilt für den Verlust V :

$$\begin{aligned} V &= cN_{\text{Koppel}} & \text{mit} \\ N_{\text{Koppel}} &\sim h^{-(d-1)}. \end{aligned}$$

Dann ist also

$$\begin{aligned} T(1) &= ch^{-d}, \\ T(p) &= \frac{ch^{-d}}{p} + ch^{-(d-1)}. \end{aligned}$$

Setzt man dies nun in die Formel für den Speedup ein, erhält man:

$$\begin{aligned} S(p) &= \frac{ch^{-d}}{\frac{ch^{-d}}{p} + ch^{-(d-1)}} \\ &= \frac{1}{\frac{1}{p} + ch} \\ &= \frac{p}{1 + ch}. \end{aligned}$$

Unsere Aussage erhalten wir jetzt durch Einsetzen der Ergebnisse für den Speedup in die Formel für die Effizienz.

$$\begin{aligned}
 \text{Eff}(p) &= \frac{1}{p} S(p) \\
 &= \frac{1}{p} \frac{p}{1+ch} \\
 &= \frac{1}{1+ch} \sim 1 - ch \\
 \implies \text{Eff}(p) &= 1 - \mathcal{O}(h).
 \end{aligned}$$

□

Bemerkung 1.6. *Beim Algorithmus 1.2 spielt das Amdahlsche Gesetz (Satz 1.1) keine Rolle, da hier der sequentielle Anteil s von N (bzw. von h) abhängt und gegen Null geht für $h \rightarrow 0$.*

Bemerkung 1.7 (parallele Realisierung im 3D).

- *Bei der parallelen Realisierung der Domain Decomposition im 3D wird die Formel (1.6) zur Berechnung des lokalen vorkonditionierten Residuums \underline{w}_s sehr kompliziert. Vorteilhaft wirkt sich jedoch die besondere Knotennummerierung aus, da hierdurch eine Balance im Rechenaufwand der einzelnen Prozessoren erzeugt wird.*
- *Bei der Adaptiven FEM erhalten wir eine Disbalance im Rechenaufwand der einzelnen Prozessoren, da in den einzelnen Teilgebieten Ω_s unterschiedlich gerechnet wird. Es müssen also zusätzliche Balancierungs Ideen eingefügt werden. Desweiteren erweist sich die Implementierung der Formel (1.6) als noch schwieriger gegenüber der DD mit gleichmäßiger Vernetzung.*
- *Einen möglichen Ausweg bietet das Farming-Konzept.*

Kapitel 2

Farming-Konzept

Beim Farming-Konzept handelt es sich um eine neue Idee zur Parallelisierung der Finiten-Elemente-Rechnung unter Verwendung eines Algorithmus für die adaptive FEM, der dem Programmablaufplan aus Abbildung 2.1 entspricht.

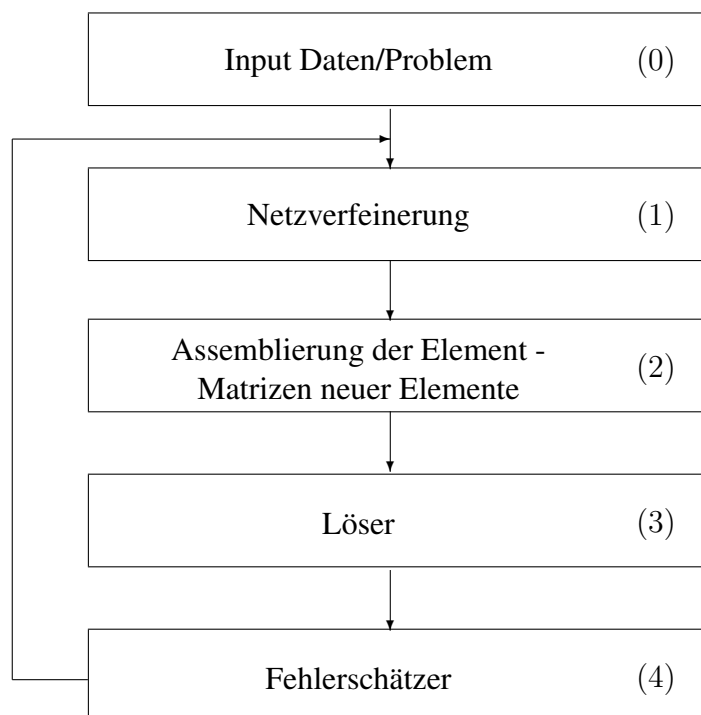


Abbildung 2.1: Programmablaufplan für die adaptive FEM

Im ersten Abschnitt dieses Kapitels wird näher auf die einzelnen Schritte des Programmablaufplanes in der Abbildung 2.1 eingegangen. Danach folgt eine Erklärung der einzelnen Datenstrukturen, die zur Durchführung der FEM mit diesem Algorithmus gebraucht werden, um im darauf folgenden Abschnitt auf dessen Parallelisierung eingehen zu können. Im vierten Abschnitt

werden die Vor- und Nachteile des Algorithmus dargestellt, um darauf aufbauend verschiedene Möglichkeiten zur Verbesserung der Laufzeit auszuarbeiten.

2.1 Realisierung

Um den Algorithmus aus Abbildung 2.1 zu verstehen, ist es notwendig, im folgenden Abschnitt näher auf die einzelnen Schritte einzugehen. Dabei wird kurz der Ablauf in jedem einzelnen Schritt erläutert.

Schritt (0) - Input Daten/Problem

In diesem Schritt erfolgt die Eingabe alle notwendigen Daten, die das zu lösende Problem betreffen, wie zum Beispiel

- das Gebiet Ω ,
- Randbedingungen an das Gebiet Ω ,
- Bestimmung der finiten Elemente (Dreiecke, Vierecke, Tetraeder) oder
- a priori Daten über die Lösung.

Schritt (1) - Netzverfeinerung

Hier wird die Teilung der finiten Elemente im aktuellen Netz realisiert, die aufgrund ihrer Eigenschaften und/oder der Eingangsinformationen markiert wurden. Dementsprechend müssen die jeweiligen Datenstrukturen NODES, EDGES, FACES und SOLIDS angepasst werden.

Schritt (2) - Assemblierung der Elementmatrizen

Jetzt wird die Steifigkeitsmatrix sowie die rechte Seite des linearen Gleichungssystems (1.1) zur Approximation des Problems im aktuellen Netz berechnet. Da die Elementmatrizen und die zugehörige rechte Seite aller bestehenden Elemente in der Datenstruktur Solids gespeichert sind, muss die Berechnung nur für die neuen Elemente ausgeführt werden.

Schritt (3) - Löser

Hier wird nun das lineare Gleichungssystem (1.1) selbst gelöst. Sind dem aktuellen Netz gerade N_n Knoten zugehörig, so hat dieses lineare Gleichungssystem die Dimension $N = N_n \cdot NDOF$. Mit $NDOF$ wird dabei die Anzahl der Freiheitsgrade für jeden Knoten bezeichnet. Im Beispiel (1.1) wäre $NDOF = 1$. Für Elastizitätsprobleme gilt etwa

$NDOF = d$, wobei d der Dimension des Berechnungsgebietes Ω entspricht. Werden gekoppelte Feldberechnungen durchgeführt, ist $NDOF$ sogar noch größer.

Schritt (4) - Fehlerschätzer

Der Fehlerschätzer ist der wohl wichtigste Schritt für die Kontrolle der adaptiven Verfeinerung des Netzes. Es wird für jede einzelne Seitenfläche der Fehler berechnet und hiermit jedes Element markiert, dessen Fehler einen festgelegten Prozentsatz des maximalen Fehlers überschreitet. Ohne diese Markierung wäre völlig unklar, welche finiten Elemente verfeinert werden sollten.

2.2 Datenstrukturen

Zur Lösung der adaptiven FEM mithilfe des Algorithmus aus der Abbildung 2.1 werden einzelne Datenstrukturen benötigt. In diesem Abschnitt werden diese Datenstrukturen benannt und die Größe des bereitgestellten Speicherplatzes sowie dessen Verwendung analysiert.

1. Nodes

Die relevanten Informationen zu den Knoten werden in einem Feld X gespeichert.

- (a) Das Feld hat die Größe $X(3 + NDOF + 1, *)$.
- (b) Jeder Knoten wird durch seine drei Koordinaten dargestellt.
- (c) Es folgt die Lösung u_i in diesem Knoten im aktuellen Netz.
- (d) Für jeden Knoten wird also folgende Struktur in X abgespeichert:

$$|X_i|Y_i|Z_i|u_1(\vec{x})| \dots |u_{NDOF}(\vec{x})|.$$

2. Edges

- (a) Die Informationen zu den Kanten sind in einem Feld $EDGE(6, *)$ gespeichert
- (b) Jede Kante wird eindeutig durch ihre Eckknoten und ihren Mittelknoten beschrieben
- (c) Jede Kante wird in der folgenden Struktur abgespeichert :

$$|Mittelknoten|Anfangsknoten|Endknoten|Sohn1|Sohn2|Polynomgrad|$$

- (d) Die Werte von $Sohn1$ und $Sohn2$ bestimmen sich je nachdem, ob die Kante geteilt ist oder nicht :
- $Sohn1 > 0$ Die Kante ist geteilt, die Nummer der zwei Söhne werden in $Sohn1$ und $Sohn2$ abgespeichert.
 - $Sohn1 \leq 0$ Die Kante ist nicht geteilt. In diesem Fall enthält $Sohn2$ geometrische Informationen zur Kante.
 - $Sohn1 < 0$ Die Kante enthält einen „hanging node“.

3. Faces

- (a) Die Informationen zu den Seitenflächen der Elemente sind in einem Feld $FACE(12, *)$ gespeichert.
- (b) | Edge1 | ... | Edge4 | Sohn1 | ... | Sohn4 | Typ | Markierung |
 | Mittelknoten | Polynomgrad |
- (c) Der Wert Typ enthält vier Informationen :
1. Byte : Pointer zur Geometrie
 2. Byte : Pointer zu den Neumann-Daten
 3. Byte : Pointer zu den Dirichlet-Daten
 4. Byte : Markierung für Randflächen
- (d) *Markierung* wird für den Fehlerschätzer gebraucht:
- 0 : Face ist nicht markiert
 - 1 : Face ist markiert

4. Volumes

Die Informationen für die Volumes werden im Feld $SOLID(MSOL, *)$ gespeichert.

- (a) $MSOL = MELFAC + MELNODE + MELMAT = 9 + 27 + MELMAT$.
 Jedes $SOLID$ ist also eine Struktur von $MSOL$ Informationen.
- (b) Jedes Volume wird eindeutig durch seine sechs Seitenflächen beschrieben.
- (c) (a) | Face1 | ... | Face6 | Material | Mittelknoten | Polynomgrad |
 (b) | Knoten1 | ... | Knoten27 |
 (c) | Steifigkeitsmatrix | rechte Seite |
- (d) Der Wert *Material* ist ein Zeiger auf das Feld $MATERIAL$.

2.3 Parallelisierung mittels Farming - Konzept

Bei der Parallelisierung von Algorithmen gibt es verschiedene Ansatzpunkte zur Organisation der einzelnen Prozessoren. In dieser Diplomarbeit wird eine Unterteilung der Prozessoren in

einen Master-Prozessor und sonstige Slave-Prozessoren verwendet. Der Master-Prozessor durchläuft den Algorithmus, wie er in der Abbildung 2.1 dargestellt ist. Die Slave-Prozessoren übernehmen einige wichtige Berechnungen. Die lokalen Ergebnisse werden dann wieder an den Master-Prozessor kommuniziert, damit dieser weiter im Algorithmus voranschreiten kann.

Beim Farming-Konzept wird, entgegen der klassischen Parallelisierungsidee, der Domain Decomposition, die im Abschnitt 1.4 beschrieben wurde, nicht der Algorithmus für die PCG-Methode parallelisiert wie im Algorithmus 1.2, sondern wir berechnen auf jedem Slave-Prozessor nur die Matrix-Vektor-Multiplikation, Formel (1.5), im Algorithmus 1.1 für einen Teil der Finiten Elemente.

Die eigentliche Parallelisierung wird also bei der Berechnung der Element-Steifigkeits-Matrizen und der dazugehörigen rechten Seiten des linearen Gleichungssystems (1.1) realisiert.

Für den allgemeinen adaptiven Algorithmus aus der Abbildung 2.1 bedeutet dies :

1. Die Schritte (0), (1) und (4) werden auf dem Master-Prozessor durchgeführt.
2. Die Schritte (2) und (3) werden über den Master-Prozessor organisiert. Die zeitaufwendigen Berechnungen finden aber auf den Slave-Prozessoren statt.
3. Im Schritt (3) wird also der Algorithmus 1.1 auf dem Master-Prozessor ausgeführt, aber die Matrix-Vektor-Multiplikation (1.5) wird auf den Slave-Prozessoren organisiert.

Es wird also die Berechnung des linearen Gleichungssystems (1.1) auf den Slaves durchgeführt. Dazu wird für jedes finite Element im aktuellen Netz die Element-Steifigkeits-Matrix im Feld $SOLID(MSOL, *)$ gespeichert. Diese Matrix wird für jedes Element auf einem der Slave-Prozessoren berechnet, damit die Matrix-Vektor-Multiplikation (1.5) durchgeführt werden kann. Auf jedem Slave-Prozessor wird also

$$\underline{v}_i = K_i q$$

berechnet. Wir erhalten $(p-1)$ Vektoren \underline{v}_i der Dimension N , die aufsummiert unseren gesuchten Vektor \underline{v} ergeben (siehe Gleichung (1.5)). Die Vektoren \underline{v}_i enthalten dabei gerade die Einträge aus \underline{v} , die zu den auf dem Prozessor P_i berechneten Elementen gehören.

2.4 Kritik und Ansatzpunkte für die Optimierung

Für eine bessere Ausarbeitung der einzelnen Möglichkeiten zur Beschleunigung der Laufzeit werden zuvor die Vor- und Nachteile des Farming-Konzeptes gegenüber der Domain Decomposition aus dem Abschnitt 1.4 hervorgehoben.

2.4.1 Kritik

Ein großer Vorteil des Farming-Konzeptes gegenüber der Domain Decomposition besteht darin, dass man sich nicht um die Balancierung des Aufwandes kümmern muss. Aufgrund der

gleichmäßigen Verteilung der Elemente auf die jeweiligen Prozessoren, ist der Aufwand pro Prozessor annähernd gleich groß.

Nachteilig wirkt sich das *Amdahlsche Gesetz*, siehe Satz 1.1, aus, wie im folgenden Beispiel zu sehen ist. Dabei wächst der Speedup bei der Parallelisierung nur geringfügig gegenüber der Steigerung der Anzahl der Prozessoren.

Beispiel 2.1. Wir betrachten den Algorithmus 1.1. Es sei mit N die Dimension des Gleichungssystems bezeichnet. Dann setzt sich der arithmetische Aufwand im Algorithmus folgendermaßen zusammen.

$$T(1) = 2N + 3N + 3N + mN,$$

also die Summe aus dem Aufwand für die Skalarprodukte ($2N$), den Vektorsummen ($3N$) und dem Vorkonditionierer ($3N$) sowie dem Aufwand für die Matrixmultiplikation (mN).

Damit erhalten wir für p Prozessoren die Zeit

$$T(p) = 8N + \frac{mN}{p} + cN$$

als Summe aus dem Aufwand für den sequentiellen Anteil, wie bei einem Prozessor, dem Aufwand für die Matrixmultiplikation (parallel) und dem Kommunikationsaufwand.

Mit der Folgerung 1.2 ergibt sich der Speedup als

$$S(p) \leq \frac{1}{s}.$$

Vernachlässigt man also den Aufwand für die Kommunikation erhalten wir einen Speedup von

$$S(p) \leq 1 + \frac{m}{8}.$$

Die Variable m hängt dabei von den gewählten finiten Elementen ab. Bei finiten Elementen für Elastizitätsprobleme mit drei Freiheitsgraden pro Knoten bedeutet dies:

- a) Besteht das Netz aus 8-Knoten-Hexaeder (also nur Eckknoten), so hat jeder Knoten 27 Verbindungen zu Nachbarknoten und m ergibt sich aus $m = 27 \cdot 3 = 81$. Für den Speedup erhalten wir dann $S(p) \leq 1 + \frac{81}{8} \approx 11$.
- b) Wählen wir 20-Knoten-Hexaeder, müssen zusätzlich die Kantenmittelknoten einbezogen werden. Die Eckknoten haben 85 Verbindungen zu Nachbarknoten, die Kantenmittelknoten etwa nur 51. Da die Eckknoten und die Kantenmittelknoten in einem gleichmäßig verfeinertem Netz etwa im Verhältnis 1 : 3 stehen, hat hier jeder Knoten durchschnittlich 59 Verbindungen zu seinen Nachbarknoten. Dadurch erhalten wir $m = 59 \cdot 3 = 177$ und der Speedup erhöht sich auf $S(p) \leq 1 + \frac{177}{8} \approx 23$.

Dabei ist zu beachten, dass die Kommunikationszeit den Speedup erheblich verringert. Akzeptabel ist ein Speedup von mindestens 10.

2.4.2 Optimierungsmöglichkeiten im Schritt (2)

Die Daten der Elemente aus dem Netz der vorangegangenen Verfeinerungen sind in den $(p - 1)$ Slave-Prozessoren gespeichert, insbesondere die Element-Steifigkeits-Matrizen und die dazugehörigen rechten Seiten. Es müssen also nur die Daten neuer Elemente an die Slave-Prozessoren verschickt werden. Diese erhebliche Reduzierung des Kommunikationsaufwandes wird zu einer Zeitersparnis führen.

2.4.3 Optimierungsmöglichkeiten im Schritt (3)

a) Optimierung bei der Matrix-Vektor-Multiplikation

- Unter der Voraussetzung einer besonderen Kommunikationsstruktur kann der Master-Prozessor schon Aufgaben erledigen, während die Slave-Prozessoren den Lösungsvektor \underline{v} kommunizieren. Dazu ist eine Kommunikationsstruktur notwendig, die dem Master erlaubt Daten an alle Slaves zu verschicken, jedoch nur von genau einem Slave Daten zu erhalten.

Betrachten wir also im Algorithmus 1.1, der auf dem Master-Prozessor berechnet wird, die einzelnen Gleichungen in jedem Iterationsschritt. Der Lösungsvektor \underline{v} wird erst zur Berechnung des Residuums \underline{r} benötigt. Es wäre also sinnvoll, dass jeder Slave ein δ_s errechnet über

$$\delta_s = (Kq_s, q_s)$$

und diese dann aufsummiert zu $\delta := \sum_{s=1}^p \delta_s$ an den Master geschickt werden. Die einzelnen δ_s sind Skalare, deren Kommunikation und Aufsummierung keinen großen Aufwand darstellen.

Jetzt kann der Master das α aus der dritten Gleichung sowie das u aus der vierten Gleichung berechnen, während die Slaves ihre Teilvektoren \underline{v}_s zusammenfügen zum Vektor \underline{v} und an den Master kommunizieren.

Die Berechnung des Skalarproduktes $\delta = (Kq, q)$ sowie die Berechnung des Vektors $\underline{u} = \underline{u} + \alpha \underline{q}$ erfolgen in der Zeit, in der die Slaves den Vektor \underline{v} kommunizieren. Es wird also die Zeit genutzt, in der der Master infolge der besonderen Kommunikationsstruktur von Aufgaben freigestellt ist. Der Aufwand wird daher für die Skalarprodukte von $2N$ auf N sowie für die Vektorsummen von $3N$ auf $2N$ reduziert. Für die Betrachtungen im obigen Beispiel 2.1 bedeutet dies eine Erhöhung des Speedup auf

$$S(p) \leq \frac{4}{3} + \frac{m}{6}.$$

- Auf jedem der $(p - 1)$ Slave-Prozessoren befindet sich nach der Matrix-Vektor-Multiplikation ein Vektor \underline{v}_i der Dimension N . Da nur ein Teil der Elemente aus dem aktuellen Netz und damit verbunden nur deren Knoten zur Berechnung von \underline{v}_i auf dem i -ten Slave-Prozessor eingehen, enthält

\underline{v}_i Null-Einträge.

Ideal wäre eine Verteilung der Elemente des aktuellen Netzes auf die Prozessoren so, dass die Knoten der Elemente stets nur auf einem Prozessor gespeichert sind. Dadurch würde die Anzahl der Null-Elemente in jedem \underline{v}_i maximiert werden. Außerdem wäre eine Addition überflüssig. Die Nicht-Null-Elemente müssten nur an dieselbe Stelle in den Vektor \underline{v} auf dem Master kopiert werden, da eine Überschneidung ausgeschlossen wäre.

Dieses Ideal ist nicht zu erreichen, da bei jeder Verteilung der Elemente auf die $(p - 1)$ Slave-Prozessoren immer auch Knoten von Elementen mit gemeinsamen Seitenflächen, Kanten oder Knoten auf verschiedenen Prozessoren gespeichert werden. Wenn die Teilung zwischen zwei benachbarten Elementen verläuft, müssen die gemeinsamen Knoten jeweils auf zwei Prozessoren gespeichert werden. Dies induziert eine Aufsummierung der Vektoren \underline{v}_i zum Vektor \underline{v} .

Ziel ist es also, die Anzahl der Null-Elemente in jedem Vektor \underline{v}_i zu maximieren, das heißt, eine geeignete Verteilung der Elemente auf den $(p - 1)$ Slave-Prozessoren zu finden, bei der die Anzahl der Knoten, die auf mehreren Prozessoren gespeichert sind, minimal ist.

b) Optimierung bei der Bildung des Vektors \underline{v} für den Algorithmus 1.1

- Kennt man nun den Index der Nicht-Null-Einträge der \underline{v}_i im Vektor \underline{v} , so kann die Zeit für die Kommunikation verringert werden, indem man nur die Nicht-Null-Elemente mit dem zugehörigen Index in \underline{v} kommuniziert.
- Es gibt verschiedene Möglichkeiten, wie man die Kommunikation der Vektoren \underline{v}_i und deren Summation zum Vektor \underline{v} gestaltet.

(i) Die Kommunikation der Prozessoren ist in einer Sterntopologie darstellbar. Alle Vektoren \underline{v}_i werden zum Master-Prozessor geschickt und dort aufsummiert ohne die Null-Elemente zu berücksichtigen.

Jeder Slave-Prozessor muss N Einträge an den Master-Prozessor kommunizieren. Insgesamt werden also $(p - 1)$ Kommunikationen mit gleichem Zeitaufwand notwendig. Im Master-Prozessor wird dann jeweils noch einmal eine gewisse Zeitdauer zur Summation des gesendeten Vektors \underline{v}_i mit dem gespeicherten Vektor benötigt.

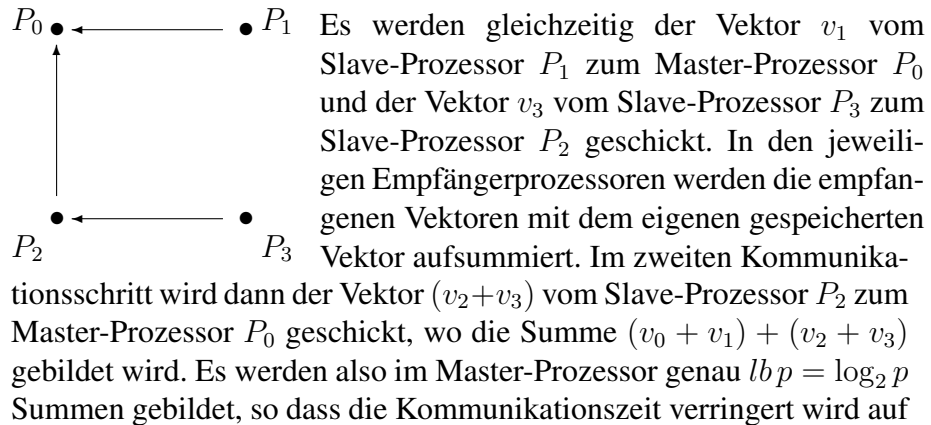
Die Sendezeit eines Vektors \underline{v}_i verhält sich proportional zu seiner Dimension, d.h zum Senden des Vektors \underline{v}_i wird ein Zeitaufwand von cN notwendig. Also erhalten wir einen Gesamtaufwand für die Kommunikation von

$$T_{Com} = (p - 1)cN.$$

(ii) Bei der Kommunikation und der Summation der Vektoren \underline{v}_i im Hypercube schickt der $i - te$ Prozessor seinen Vektor \underline{v}_i zu seinem unmittelbaren Nachbarprozessor, wo die Vektoren aufsummiert werden. Dieser Vorgang wiederholt sich, bis die Daten sämtlicher Slave-Prozessoren

beim Master-Prozessor angekommen sind.

Zum besseren Verständnis wird hier ein kurzes Beispiel gegeben. Wir betrachten einen Hypercube mit der Dimension 2, also ein System mit $p = 4$ Prozessoren P_i , $i = 0, \dots, 3$. Auf jedem Prozessor P_i ist ein Vektor \underline{v}_i gespeichert.



$$T_{Com} = \log_2 p \cdot cN.$$

Bemerkung 2.1. Sind die Indizes der Nicht-Null-Elemente des Vektors \underline{v}_i in \underline{v} bekannt für alle Prozessoren P_i und gelingt es, wie in Abschnitt 2.4.3 a) beschrieben, einen Block bestehend aus allen Nicht-Null-Elementen in den Vektoren \underline{v}_i zu erzeugen, verringert sich die Kommunikation weiter. Statt des ganzen Vektors \underline{v}_i muss lediglich der Block der Nicht-Null-Elemente kommuniziert werden. Bei der Verwendung der Sterntopologie werden dann etwa $\frac{N}{p-1}$ Einträge sowie deren Indizes in \underline{v} als einfach genaue Werte gesendet. Der Aufwand dafür beträgt dann lediglich

$$\begin{aligned} T_{Com} &= (p-1)c \left(\frac{N}{p-1} + \frac{N}{2(p-1)} \right) \\ &= \frac{3}{2}cN. \end{aligned}$$

Im Hypercube bedeutet dies, dass im ersten Kommunikationsschritt gerade $\frac{N}{p}$ Elemente aus den Vektoren \underline{v}_i sowie noch einmal $\frac{N}{2p}$ Indizes als einfach genaue Werte übertragen werden. Im zweiten Schritt sind es dann $\frac{2N}{p}$ Nicht-Null-Elemente und noch einmal $\frac{N}{p}$ Indizes usw. Wir erhalten also für die Kommunikation der Nicht-Null-Elemente:

$$\begin{aligned}
T_{El} &= \sum_{i=0}^{n-1} \frac{2^i cN}{p} \\
&= cN \left(\sum_{i=1}^n \left(\frac{1}{2}\right)^i \right) \\
&< cN \left(\sum_{i=1}^{\infty} \left(\frac{1}{2}\right)^i \right) \\
&= cN.
\end{aligned}$$

Für die Indizes ist noch einmal folgender Kommunikationsaufwand nötig mit

$$\begin{aligned}
T_{Ind} &= \sum_{i=1}^{n-1} \frac{2^i cN}{2p} \\
&= cN \left(\sum_{i=2}^{n+1} \left(\frac{1}{2}\right)^i \right) \\
&< cN \left(\sum_{i=2}^{\infty} \left(\frac{1}{2}\right)^i \right) \\
&= \frac{1}{2}cN.
\end{aligned}$$

Der gesamte Kommunikationsaufwand für die Nicht-Null-Elemente und deren Indizes in \underline{v} beträgt dann

$$T_{Com} = T_{El} + T_{Ind} = \frac{3}{2}cN.$$

Bemerkung 2.2. Nach Bemerkung 2.1 ist die Kommunikationszeit für beide Varianten gleich. Auffällig ist aber, dass bei der Beachtung der Nullen in den Vektoren \underline{v}_i die Kommunikationszeit unabhängig von der Prozessoranzahl p bzw. der Dimension des Hypercubes n ist.

Kapitel 3

Optimierung des Farming-Konzeptes und dessen Parallelisierung

In diesem Kapitel wird die Umsetzung der genannten Optimierungsmöglichkeiten aus dem Abschnitt 2.4 betrachtet. Dem Leser werden die erzielten Ergebnisse anhand eines ausgewählten Beispiels näher gebracht. Dabei werden neben der graphischen Darstellung des Zeitverlaufs auch einzelne Zeiten für bestimmte Anzahlen von Elementen in der Verfeinerung in einer Tabelle ausgeben.

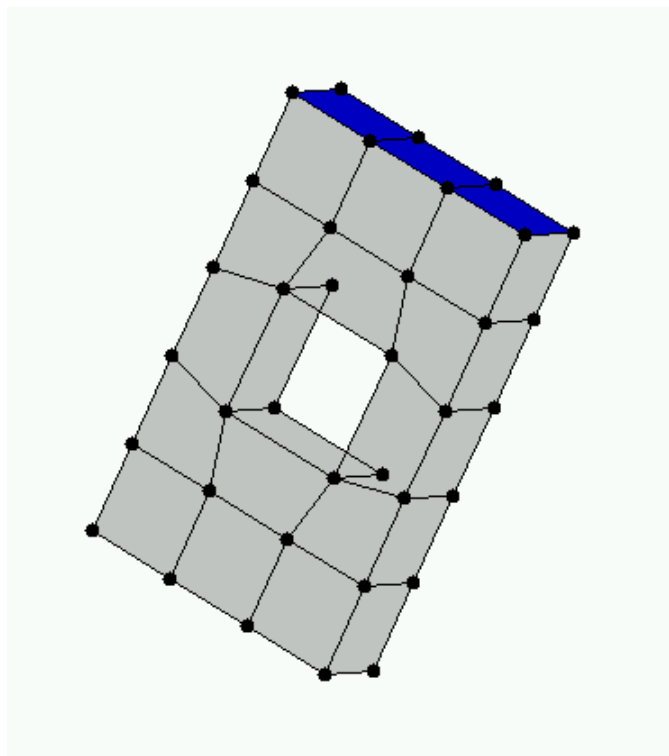
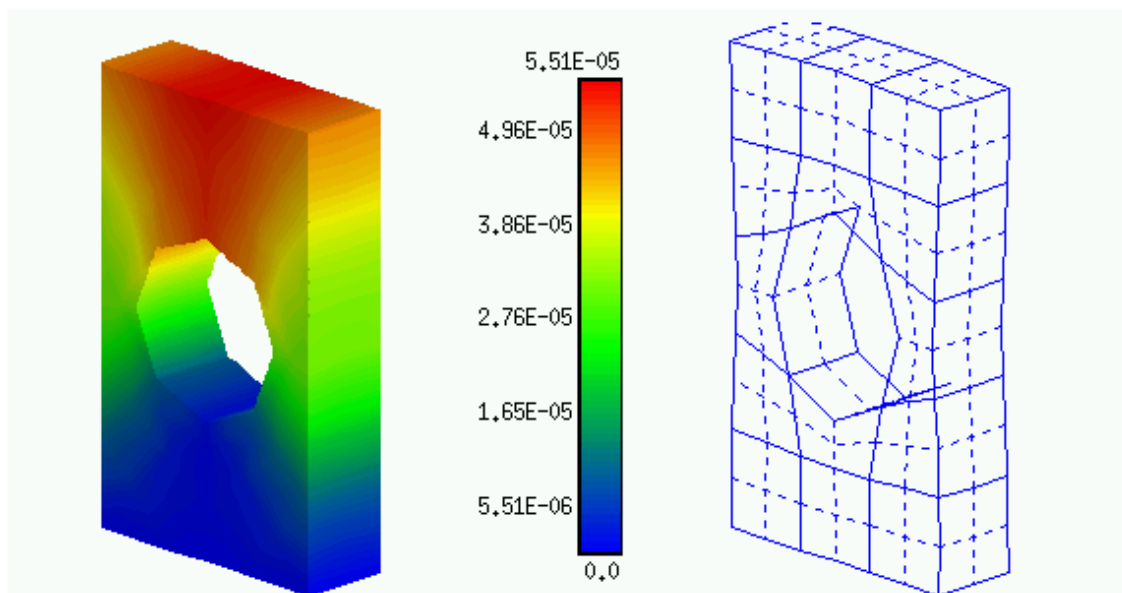
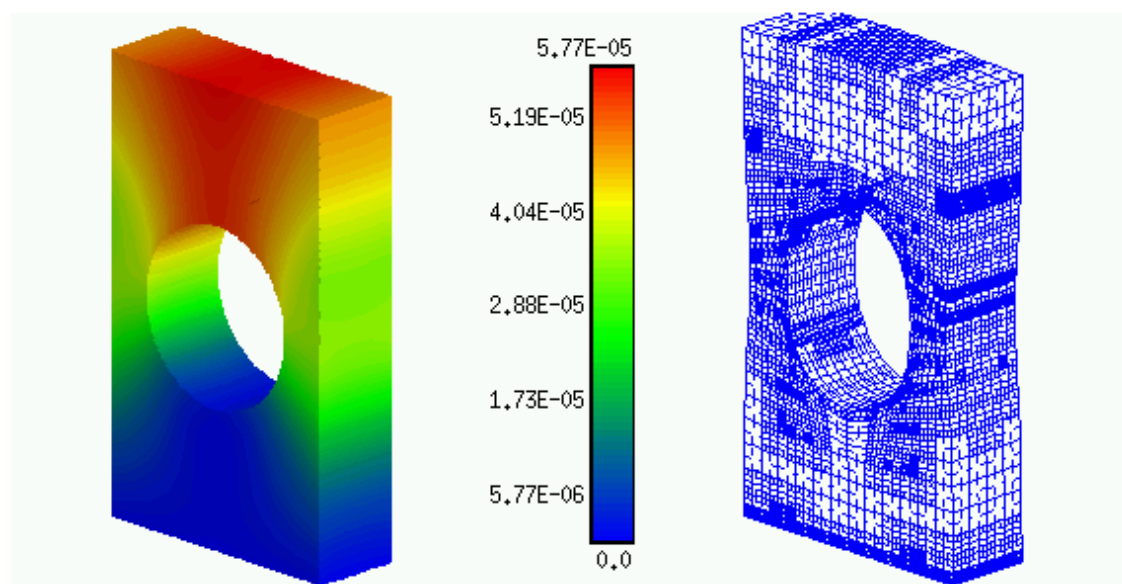


Abbildung 3.1: Beispiel: Grobnetz loch3D

In der Abbildung 3.1 sieht man das Grobnetz für das Beispiel loch3D, mit dem sämtliche Rechnungen durchgeführt wurden und welche nun im Folgenden aufgeführt und verglichen werden. Die untere verdeckte Fläche ist dabei mit Dirichlet-Randbedingungen versehen und fixiert. Die gegenüberliegende blaue Fläche besitzt Neumann-Randbedingungen. In der Simulation wirken Kräfte auf diese Fläche, so dass sich das Bauteil verformt.



(a) Verfeinerung 14 Elemente



(b) Verfeinerung 9520 Elemente

Abbildung 3.2: Netzverfeinerung und Fehler

Die Abbildung 3.2 zeigt die graphische Darstellung des FEM-Netzes für das Beispiel loch3D, einmal für 14 Elemente und einmal für 9520 Elemente. Zudem wird die euklidische Norm der Verschiebung, also der Betrag der Verschiebung, farbig codiert. Sehr schön erkennbar ist die starke Verfeinerung um das Loch, wo sich die Verformung am stärksten auswirkt.

Um die Rechnungen vergleichbar zu machen, wurden diese immer auf dem Parallelrechner *CHIC* der TU-Chemnitz durchgeführt. Es wurden dabei in jeder Rechnung jeweils 17 Prozessoren benutzt.

Ausgangspunkt dieser Arbeit war eine Implementierung der FEM deren Kommunikation über einen Hypercube realisiert wurde. Der Master-Prozessor war dabei in diese Topologie integriert. Damit verbunden nahm der Master-Prozessor neben der Durchführung des PCG-Algorithmus 1.1 auch an der Summation des Lösungsvektors teil. Zur Durchführung des Programmes wurden also nur 16 Prozessoren benutzt.

Die Kommunikation innerhalb des Hypercubes basierte auf einer Punkt-zu-Punkt-Kommunikation, die über MPI-Funktionen realisiert wurde. Sämtliche in den Abschnitten 2.4.2 und 2.4.3 betrachteten Optimierungsmöglichkeiten waren nicht in dieses Programm integriert. Im Folgenden wird dieses Programm immer mit Ausgangsimplementierung bezeichnet.

Aus den Betrachtungen zum Speedup in Abschnitt 2.4.1 geht hervor, dass die Kommunikation zwischen den Prozessoren einen erheblichen Anteil an der Laufzeit des Programmes hat. Diese Kommunikation findet, wie in Abschnitt 2.3 bereits erwähnt, nur im Schritt (2) des Algorithmus 2.1, der Assemblierung, sowie im Schritt (3), dem Löser, statt.

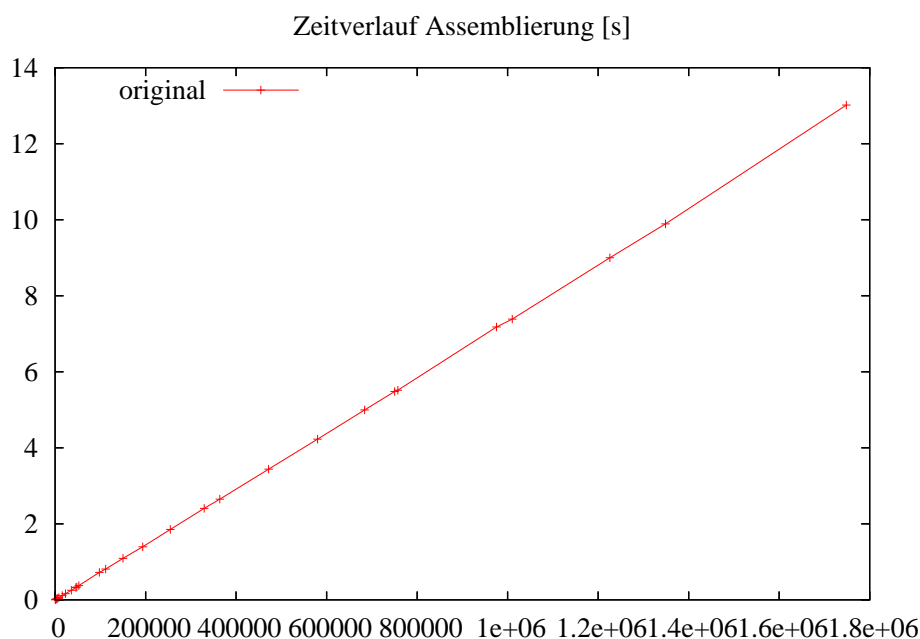


Abbildung 3.3: Zeitverlauf Assemblierung Ausgangsimplementierung

In der Abbildung 3.3 sieht man den Zeitverlauf für die Assemblierung der Elementmatrizen (Schritt (2)) in der Ausgangsimplementierung über die Anzahl der Freiheitsgrade in der aktuellen

Vernetzung. Dabei erkennt man leicht ein annähernd lineares Wachstum der Zeit bei steigender Anzahl der Elemente im aktuellen Netz.

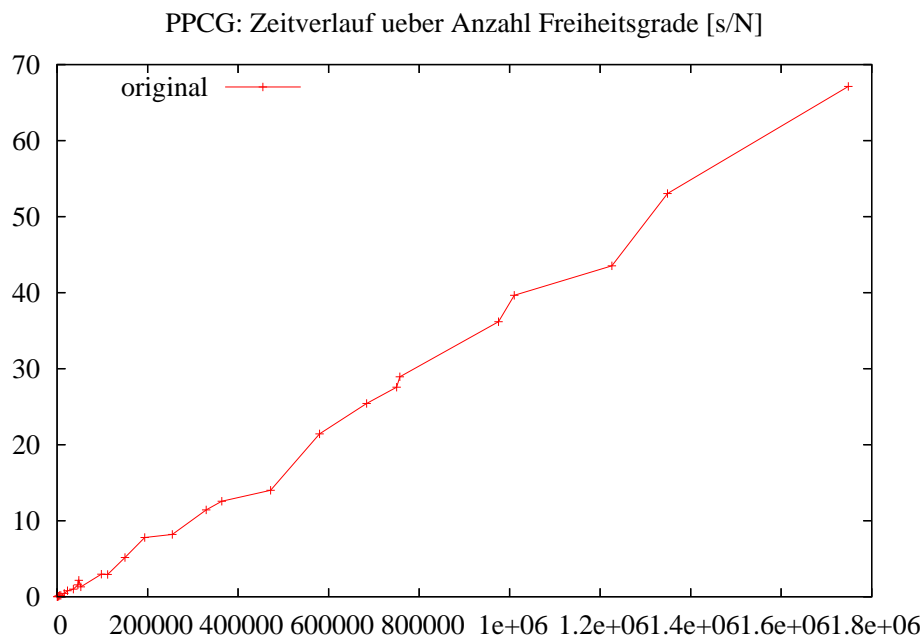


Abbildung 3.4: Zeitverlauf PPCGM Ausgangsimplementierung

Anzahl		PPCGM		geschätzter Fehler
Elemente	Knoten	Iterationen	Zeit in [s]	
9520	97930	24	2.973	6.5e-05
32536	329468	28	11.422	4.8e-06
97468	975348	30	36.168	1.2e-06
176477	1748304	28	67.129	2.0e-07

Tabelle 3.1: Zeitverlauf PPCGM Ausgangsimplementierung

Die Abbildung 3.4 zeigt den Zeitverlauf der Ausgangsimplementierung für die parallele Berechnung der PCG-Methode wieder über die Anzahl der Freiheitsgrade der aktuellen Vernetzung. Auch hier erhält man für den Graphen ein annähernd lineares Wachstum der Laufzeit bei steigender Elementanzahl im aktuellen Netz. Im Unterschied zur Abbildung 3.3 ist der Anstieg des Graphen jedoch viel steiler. Der steilere Anstieg und damit verbunden der größere Zeitaufwand für den Schritt (3) im Algorithmus 2.1 begründet sich in der Tatsache, dass im Schritt der Assemblierung ein erheblich geringerer Kommunikationsaufwand erforderlich ist als im Lösungsschritt. Aufgrund dessen werden wir uns im Folgenden auch nur auf die Optimierungsmöglichkeiten für den Lösungsschritt konzentrieren, welche im Abschnitt 2.4.3 beschrieben wurden.

3.1 Verwendung von MPI-Kommunikationsroutinen

Wie schon in den vorangegangenen Kapiteln erwähnt, hat der Kommunikationsaufwand zwischen den Prozessoren bei einem Parallelrechner einen erheblichen Einfluss auf die Laufzeit des Programmes. Je mehr Daten zwischen den einzelnen Prozessoren oder auch zwischen allen Prozessoren kommuniziert werden müssen, desto größer wird die Laufzeit.

In der Ausgangsimplementierung wurde die Kommunikation über Kommunikationsroutinen realisiert, die an der TU Chemnitz, Fakultät für Mathematik erstellt wurden. Diese verwendeten die sogenannte Punkt-zu-Punkt-Kommunikation durch MPI-Routinen. Dies ist eine Kommunikationsart, die zwischen genau zwei Prozessoren, einem Sender und einem Empfänger, stattfindet. Daneben existiert auch eine globale Kommunikation, bei der ein Prozessor mit allen anderen Prozessoren gleichzeitig kommuniziert. Bei dem **Message Passing Interface**, kurz MPI, handelt es sich um eine Bibliothek von Kommunikationsroutinen, welche seit etwa 1993 erstellt und weiterentwickelt werden. Ziel war und ist es, eine Bibliothek zu erhalten, welche systemunabhängig die Kommunikation zwischen zwei oder mehr Prozessoren regelt. Dabei gibt es heute verschiedene MPI-Standards. Nähere Informationen zum MPI, dessen Standards und Verwendung findet man in dem Kapitel von N. K. Jimack und N. Touheed aus dem Buch [1, Topping, B. H. V.].

In dieser Diplomarbeit wurden die einzelnen Ergebnisse mit OpenMPI und MVAPICH2 getestet. Die Ergebnisse der Ausgangsimplementierung wurden im OpenMPI-Standard gemessen.

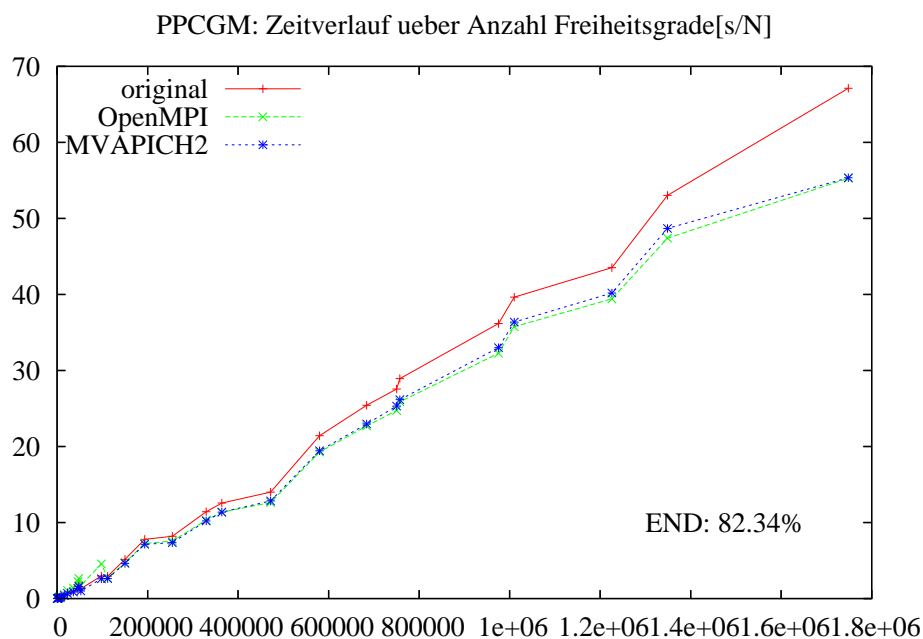


Abbildung 3.5: Zeitverlauf PPCGM Ausgangsimplementierung, MPI-Standards

In der obigen Abbildung 3.5 ist der Zeitaufwand für den Lösungsschritt in der Ausgangsimplementierung sowie in der verbesserten Implementierung, unter Nutzung der global operierenden Kommunikationsroutine *MPI_REDUCE*, für die beiden MPI-Standards OpenMPI und MVA-

Anzahl		PPCGM				geschätzter Fehler
Elemente	Knoten	Iterationen	Zeit in [s]			
			original	OpenMPI	MVAPICH2	
9520	97930	24	2.973	4.562	2.594	6.5e-05
32536	329468	28	11.422	10.312	10.219	4.8e-06
97468	975348	30	36.168	32.266	32.992	1.2e-06
176477	1748304	28	67.129	55.273	55.352	2.0e-07

Tabelle 3.2: Vergleich Zeitverlauf Ausgangsimplementierung mit MPI-Standards

PICH2 über die Anzahl der Freiheitsgrade in der aktuellen Vernetzung zu sehen.

Es ist leicht erkennbar, dass das Ersetzen der hauseigenen Kommunikationsroutinen durch global operierende MPI-Routinen einen erheblichen Einfluss auf die Laufzeit des Programmes hat. Der Graph der Zeitmessung der Ausgangsimplementierung hebt sich dabei deutlich von den Graphen der MPI-Standards ab. Zwischen den zwei MPI-Standards hingegen gibt es aber kaum Unterschiede. Die Prozentangabe im unteren rechten Teil der Graphik bezieht sich dabei auf die Dauer der Zeitmessung der getesteten Implementierung im OpenMPI-Standard gegenüber der Ausgangsimplementierung.

3.2 Einführung einer neuen Kommunikationsstruktur

Wie schon in Abschnitt 2.4.3 a) beschrieben, soll eine neue Kommunikationsstruktur eingeführt werden, die dem Master-Prozessor ermöglicht, im Algorithmus 1.1 voranzuschreiten, während die Slave-Prozessoren den Lösungsvektor \underline{y} berechnen. Dabei ist zu beachten, dass sich die Kommunikation nur zwischen den Prozessoren ausführen lässt, die in ein und demselben Kommunikator organisiert sind. Ein Kommunikator stellt eine Vereinigung von einer Anzahl von Prozessoren dar, denen innerhalb dieses Kommunikators ein eindeutiger Rang zugeordnet ist. Ein Prozessor kann dabei in mehreren Kommunikatoren organisiert sein und natürlich in jedem Kommunikator auch einen anderen Rang zugewiesen bekommen. Standardmäßig wird von MPI ein globaler Kommunikator bereitgestellt, der sämtliche Prozessoren enthält und nicht gebildet werden muss. Mehr zu diesem Thema kann man im Buch [1, B.H.V. Topping] nachlesen.

Zusätzlich zu diesem globalen Kommunikator wurde von mir ein weiterer Kommunikator erstellt, der nur die $(p - 1)$ Slave-Prozessoren beinhaltet. Die Kommunikationsstruktur lässt sich dann wie in Abbildung 3.6 darstellen.

Darin sieht man die Anordnung der einzelnen Prozessoren nach Einführung der neuen Kommunikationsstruktur. Der Unterschied zu der bisher verwendeten Sterntopologie besteht darin, dass zwischen den Slave-Prozessoren und dem Master-Prozessor ein Master-Slave-Prozessor geschaltet wird. Dieser stellt eine Kommunikationsbrücke zwischen den Slave-Prozessoren und dem Master-Prozessor dar. Die Kommunikationstopologie, also die Art der Organisation der Kommunikation zwischen dem Masterslave-Prozessor und den Slave-Prozessoren, entspricht dabei einer der in Abschnitt 2.4.3 b) diskutierten Prozessoranordnungen.

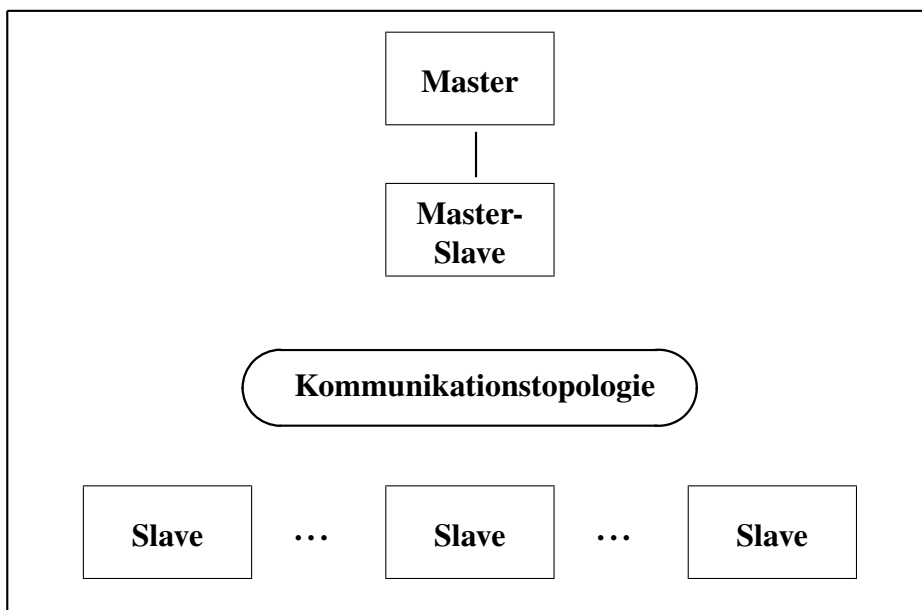


Abbildung 3.6: Kommunikationsstruktur mit Slave-Kommunikator

Wird zur Kommunikation und Summation der lokalen Lösungsvektoren die MPI-Funktion *MPI_REDUCE* verwendet, so kann nicht genau bestimmt werden, welche Topologie verwendet wird.

Anzahl		PPCGM				geschätzter Fehler
Elemente	Knoten	Iterationen	Zeit in [s]			
			original	OpenMPI	MVAPICH2	
9520	97930	24	2.973	4.554	2.555	6.5e-05
32536	329468	28	11.422	10.091	10.094	4.8e-06
97468	975348	30	36.168	31.667	32.555	1.2e-06
176477	1748304	28	67.129	53.383	55.039	2.0e-07

Tabelle 3.3: Vergleich Zeitverlauf Ausgangsimplementierung und MPI-Standards mit neuer Kommunikationsstruktur

Die Abbildung 3.7 zeigt den Vergleich des Zeitaufwandes der Ausgangsimplementierung mit den MPI-Standards OpenMPI und MVAPICH2 über die Anzahl der Freiheitsgrade der gerade betrachteten Vernetzung. Aufgrund der Einführung der neuen Kommunikationsstruktur mit einem Masterslave und den verbleibenden Slaves erhalten wir einen geringen Zeitgewinn, der darauf zurückzuführen ist, dass der Master im PCG-Algorithmus 1.1 voranschreiten kann, während die Slaves die Lösung berechnen. Bei der Kommunikation und Summation der lokalen Lösungsvektoren wurde die globale Kommunikationsfunktion *MPI_REDUCE* verwendet.

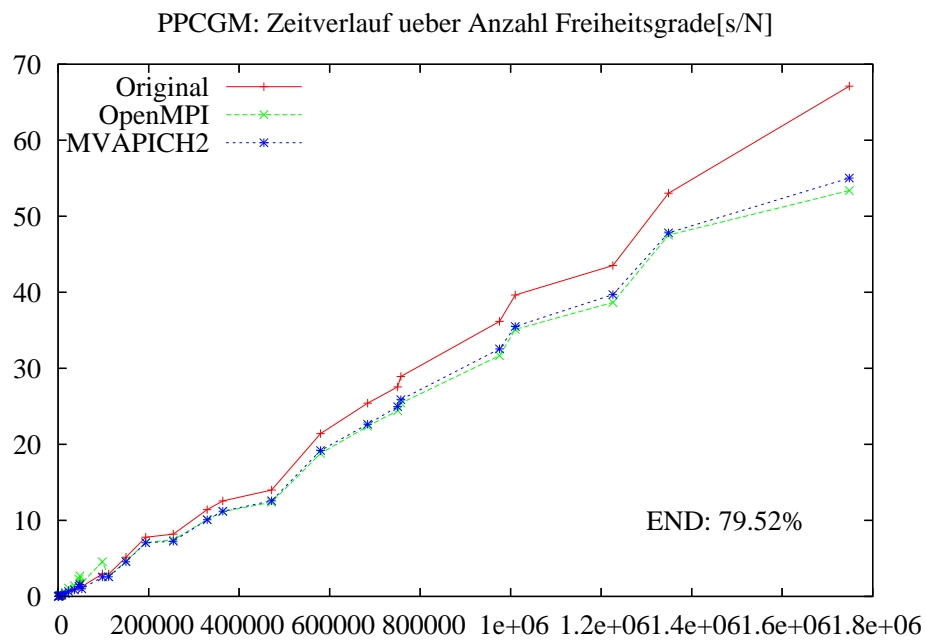


Abbildung 3.7: Zeitverlauf PPCGM Ausgangsimpementierung, MPI-Standards, neue Kommunikationsstruktur

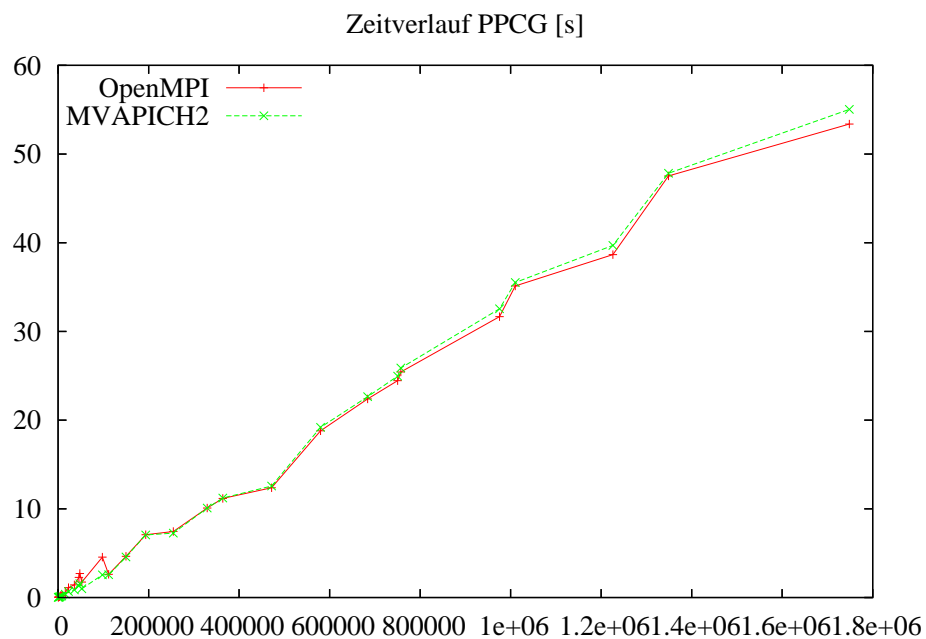


Abbildung 3.8: Zeitverlauf PPCGM MPI-Standards, neue Kommunikationsstruktur

Anzahl		PPCGM			geschätzter Fehler
Elemente	Knoten	Iterationen	Zeit in [s]		
			OpenMPI	MVAPICH2	
9520	97930	24	4.554	2.555	6.5e-05
32536	329468	28	10.091	10.094	4.8e-06
97468	975348	30	31.667	32.555	1.2e-06
176477	1748304	28	53.383	55.039	2.0e-07

Tabelle 3.4: Vergleich Zeitverlauf MPI-Standards mit neuer Kommunikationsstruktur

Betrachtet man die Abbildung 3.8, so sieht man erneut den Zeitaufwand für den Lösungsschritt im FEM-Programm unter Nutzung der neuen Kommunikationsstruktur über die Anzahl der Freiheitsgrade im jeweiligen Netz. Dargestellt sind dabei nur die Graphen der beiden MPI-Standards. Im Bereich bis etwa 800.000 Freiheitsgraden sind beide Standards gleichwertig, danach zeigt OpenMPI eine geringfügig schnellere Performance als MVAPICH2. Die Unterschiede sind dabei aber so gering, dass man keinesfalls davon ausgehen kann, einen der beiden Standards zu favorisieren.

3.3 Indizierte Addition des Lösungsvektors

Zur Berechnung der Lösung im Algorithmus 1.1 wurde in den bisherigen Implementierungen die Funktion *MPI-REDUCE* verwendet, welche den globalen Kommunikationsfunktionen zuzuordnen ist. Außerdem wurde diese Funktion aufgrund der langen Entwicklung des MPI sehr gut optimiert.

Bei der Implementierung der nun folgenden Optimierungsmöglichkeiten aus dem Abschnitt 2.4.3 muss diese Funktion ersetzt werden, da die weiteren Optimierungen eine Punkt-zu-Punkt-Kommunikation erforderlich machen. Dies bringt erst einmal einen deutlichen Zeitverlust mit sich, wie man in Abbildung 3.9 gut sehen kann. Aufgrund dessen werden die Optimierungsmöglichkeiten zwar eine zeitliche Verbesserung gegenüber der Ausgangsimplementierung bewirken, ob deren Zeiten dann aber unter den Zeiten liegen, die man bei Benutzung der *MPI-REDUCE* Funktion erhält, werden die weiteren Tests zeigen.

In der Abbildung 3.9 sind der Graph für die Ausgangsimplementierung sowie der Graph für die Implementierung mit MPI-Funktionen unter Nutzung der Punkt-zu-Punkt-Kommunikation zwischen den Prozessoren zu sehen. Die Kommunikation ist in einer Sterntopologie realisiert, wie es auch in den zukünftigen Implementierungen sein wird, die keinen Hypercube verwenden. Es ist eine deutliche Verschlechterung der Zeiten gegenüber der Ausgangsimplementierung zu sehen. Grund für diese Verschlechterung ist die Organisation der Kommunikation zwischen den Prozessoren, die in der Ausgangsimplementierung als Hypercube angeordnet sind.

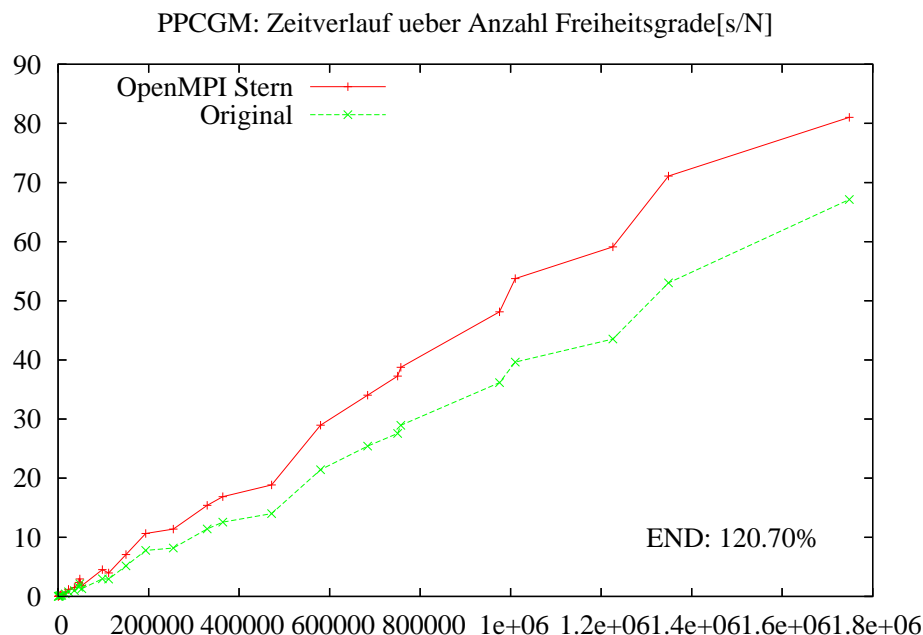


Abbildung 3.9: Zeitverlauf PPCGM Ausgangsimplementierung, OpenMPI mit Sterntopologie

Anzahl		PPCGM			geschätzter Fehler
Elemente	Knoten	Iterationen	Zeit in [s]		
			Original	OpenMPI Stern	
9520	97930	24	2.973	4.539	6.5e-05
32536	329468	28	11.422	15.398	4.8e-06
97468	975348	30	36.168	48.141	1.2e-06
176477	1748304	28	67.129	81.027	2.0e-07

Tabelle 3.5: Vergleich Zeitverlauf Ausgangsimplementierung mit OpenMPI, Sterntopologie

Im Abschnitt 3.3 wird sich auf die Darstellung des Zeitverlaufs unter Verwendung einer Anordnung der Slave-Prozessoren in einer Sterntopologie beschränkt, in der jeder Slave seinen lokalen Lösungsvektor \underline{v}_i an den Master-Slave sendet, der diese dann aufsummiert, bevor der gesamte Lösungsvektor des Gleichungssystems (1.1) vom Master-Slave an den Master kommuniziert wird.

Im Abschnitt 2.4.3 b) wurde eine Verbesserung des Programmes diskutiert, die bei der Summierung der lokalen Lösungsvektoren \underline{v}_i auf den Slaves zum Lösungsvektor \underline{v} des Gleichungssystems (1.1) die Null-Einträge der lokalen Lösungsvektoren ausnutzt. Dabei werden von jedem Slave-Prozessor jeweils zwei Vektoren kommuniziert, die bei einer geeigneten Verteilung der Elementmatrizen sehr viel weniger Einträge haben als die lokalen Lösungsvektoren \underline{v}_i . Der erste Vektor enthält dann die Indizes der Nicht-Null-Elemente aus den \underline{v}_i und der zweite Vektor die

jeweiligen Nicht-Null-Elemente selbst.

Es folgen einige Laufzeitbetrachtungen für die parallele PCG-Methode unter Ausnutzung der Null-Einträge in den \underline{v}_i der Slave-Prozessoren bei deren Summation zum Lösungsvektor \underline{v} des linearen Gleichungssystems (1.1). Die Vektoren der Slave-Prozessoren werden dabei an den Masterslave kommuniziert, der die Nicht-Null-Elemente der lokalen Lösungsvektoren \underline{v}_i ihren Indizes entsprechend aufsummiert und den fertigen Lösungsvektor \underline{v} an den Master sendet.

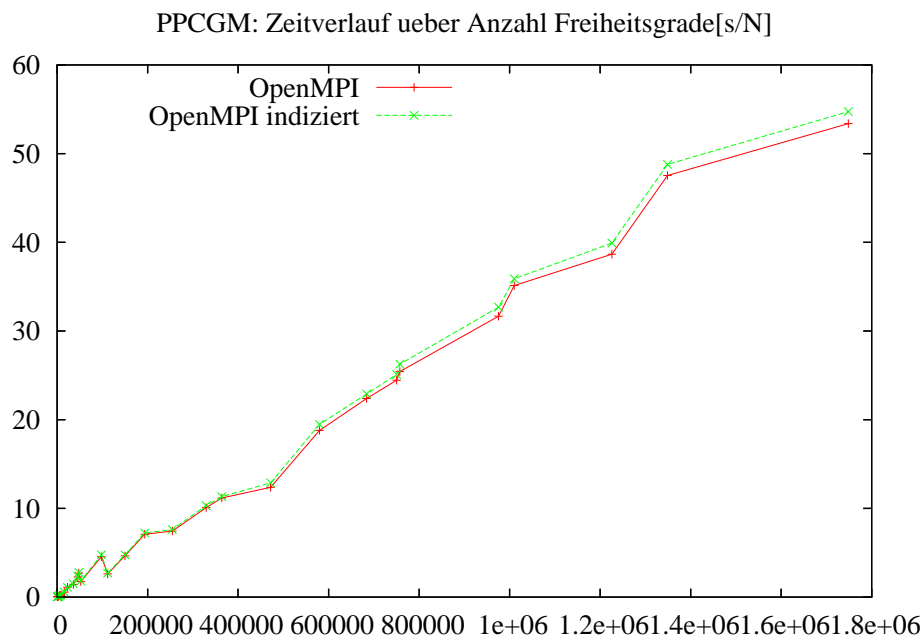


Abbildung 3.10: Zeitverlauf PPCGM OpenMPI, indizierter Addition

Anzahl		PPCGM OpenMPI			geschätzter Fehler
Elemente	Knoten	Iterationen	Zeit in [s]		
			indiziert	nicht indiziert	
9520	97930	24	4.770	4.554	6.5e-05
32536	329468	28	10.039	10.091	4.8e-06
97468	975348	30	31.824	31.667	1.2e-06
176477	1748304	28	53.816	53.383	2.0e-07

Tabelle 3.6: Vergleich Zeitverlauf PPCGM OpenMPI indiziert, nicht indiziert

In der Abbildung 3.10 wird der Zeitaufwand des Programmes im Lösungsschritt über die Anzahl der Freiheitsgrade im derzeitigen Netz für den MPI-Standard OpenMPI dargestellt. Dabei zeigt der grüne Graph den zeitlichen Aufwand für die parallele PCG-Methode mit indizierter Addition, das heißt unter Ausnutzung der Null-Elemente in den lokalen Lösungsvektoren \underline{v}_i , sowie der neuen Kommunikationsstruktur.

Der rote Graph zeigt zum Vergleich den Zeitverlauf für den Lösungsschritt ohne indizierte Addition, wie er schon in den jeweiligen Abbildungen 3.7 und 3.8 zu sehen war.

Die Abbildung 3.10 zeigt eine geringfügig längere Laufzeit für das indizierte Verfahren gegenüber dem mit der MPI-Funktion *MPI_REDUCE*. Das Ergebnis ist einerseits Folge einer guten Optimierung der Funktion *MPI_REDUCE* und andererseits einer nur suboptimalen Verteilung der Elementmatrizen auf den Slaves, wodurch weniger Null-Elemente in den lokalen Lösungsvektoren \underline{v}_i entstehen. Da sich diese Optimierung im Allgemeinen als etwas schwieriger erweist, wurde hier eine suboptimale Anweisung zur Verteilung der Elementmatrizen angewendet.

Dass eine gesonderte Verteilung der Elementmatrizen auf den Slaves dennoch sinnvoll erscheint, zeigt die folgende Abbildung 3.11.

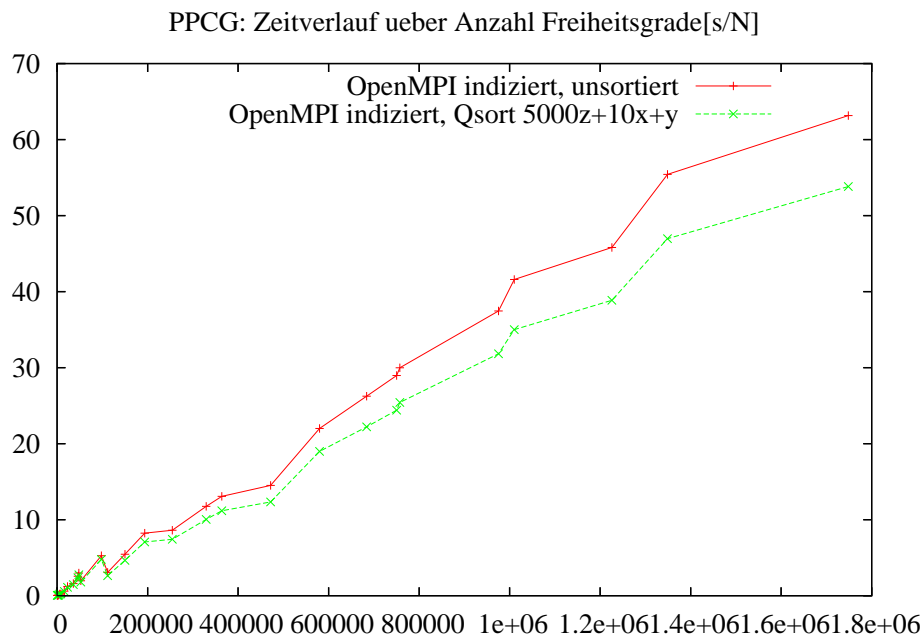


Abbildung 3.11: Zeitverlauf PPCGM OpenMPI indiziert, sortiert, unsortiert

In der Abbildung 3.11 sieht man den Zeitverlauf des Lösungsschrittes mit indizierter Addition zum einen mit sortierter und zum anderen mit chaotischer Speicherung der Elementmatrizen auf den Slaves. Diese Graphik macht die Notwendigkeit der Umsortierung der Elementmatrizen auf den Slaves besonders deutlich und dient nur zur Anschauung, da die vorhergehende Abbildung 3.10 keine Verbesserung des zeitlichen Aufwandes aufgrund der Einführung der Umverteilung der Elementmatrizen auf den Slaves aufzeigte.

Anzahl		PPCGM OpenMPI			geschätzter Fehler
Elemente	Knoten	Iterationen	Zeit in [s]		
			sortiert	nicht sortiert	
9520	97930	24	4.770	5.258	6.5e-05
32536	329468	28	10.039	11.746	4.8e-06
97468	975348	30	31.824	37.457	1.2e-06
176477	1748304	28	53.816	63.156	2.0e-07

Tabelle 3.7: Vergleich Zeitverlauf OpenMPI sortiert, unsortiert

Ein ganz ähnlicher Effekt zeigte sich auch bei der Verwendung des MPI-Standards MVAPICH2.

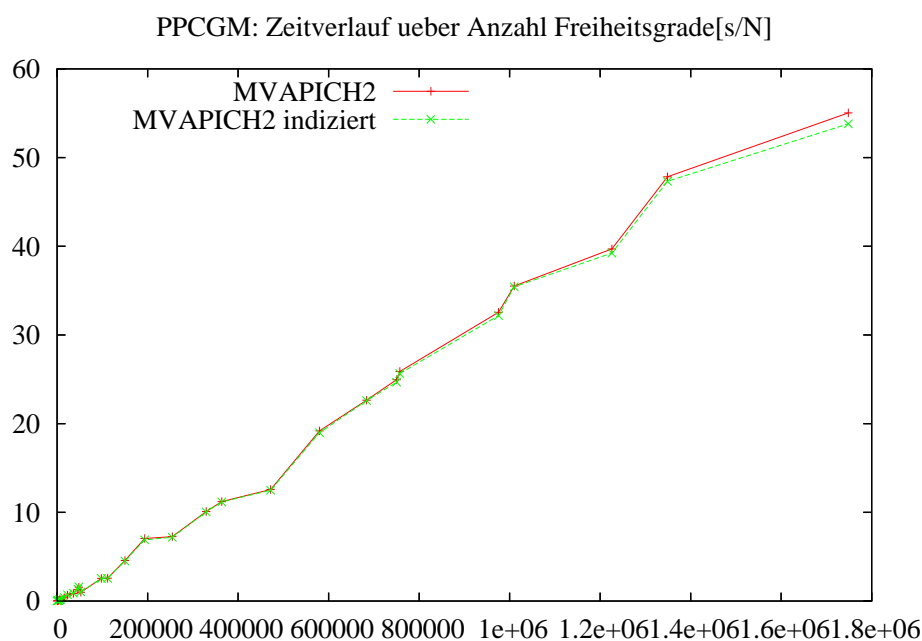


Abbildung 3.12: Zeitverlauf PPCGM MVAPICH2, indizierter Addition

Die Abbildung 3.12 zeigt die Graphen der Laufzeiten für MVAPICH2 einmal mit und einmal ohne indizierter Addition. In diesem Standard bringt die indizierte Addition noch einen Zeitgewinn. Jedoch ist dieser wiederum viel geringer als erwartet. Es zeigt sich derselbe Effekt wie im anderen MPI-Standard OpenMPI. Natürlich ist auch bei MVAPICH2 eine gesonderte Verteilung der Elementmatrizen auf die Slaves sinnvoll.

Anzahl		PPCGM MVAPICH2			geschätzter Fehler
Elemente	Knoten	Iterationen	Zeit in [s]		
			indiziert	nicht indiziert	
9520	97930	24	2.574	2.555	6.5e-05
32536	329468	28	10.176	10.094	4.8e-06
97468	975348	30	32.531	32.555	1.2e-06
176477	1748304	28	54.566	55.039	2.0e-07

Tabelle 3.8: Vergleich Zeitverlauf PPCGM MVAPICH2 indiziert, nicht indiziert

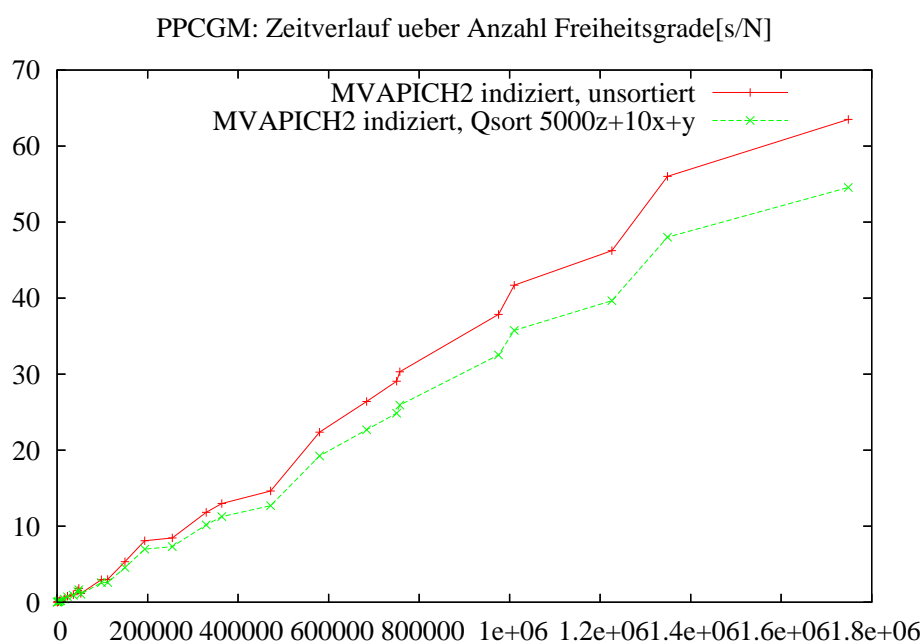


Abbildung 3.13: Zeitverlauf PPCGM MVAPICH2 indiziert, sortiert, unsortiert

Anzahl		PPCGM MVAPICH2			geschätzter Fehler
Elemente	Knoten	Iterationen	Zeit in [s]		
			sortiert	nicht sortiert	
9520	97930	24	2.574	2.965	6.5e-05
32536	329468	28	10.176	11.820	4.8e-06
97468	975348	30	32.531	37.855	1.2e-06
176477	1748304	28	54.566	63.508	2.0e-07

Tabelle 3.9: Vergleich Zeitverlauf MVAPICH2 sortiert, unsortiert

Wie schon in OpenMPI erhalten wir auch für den MPI-Standard MVAPICH2 für eine unvorteilhafte Verteilung der Elementmatrizen auf den Slaves eine sehr viel schlechtere Laufzeit für die parallele PCG-Methode als bei gut verteilten Elementmatrizen. Auch hier wurde die gleiche suboptimale Anweisung zur Verteilung der Elementmatrizen auf den Slaves angewendet wie bei OpenMPI. Die Abbildung 3.13 zeigt einen deutlich geringeren Zeitaufwand für die suboptimale Verteilung der Elementmatrizen.

Am Ende des Abschnittes werden noch einmal kurz die beiden MPI-Standards miteinander verglichen.

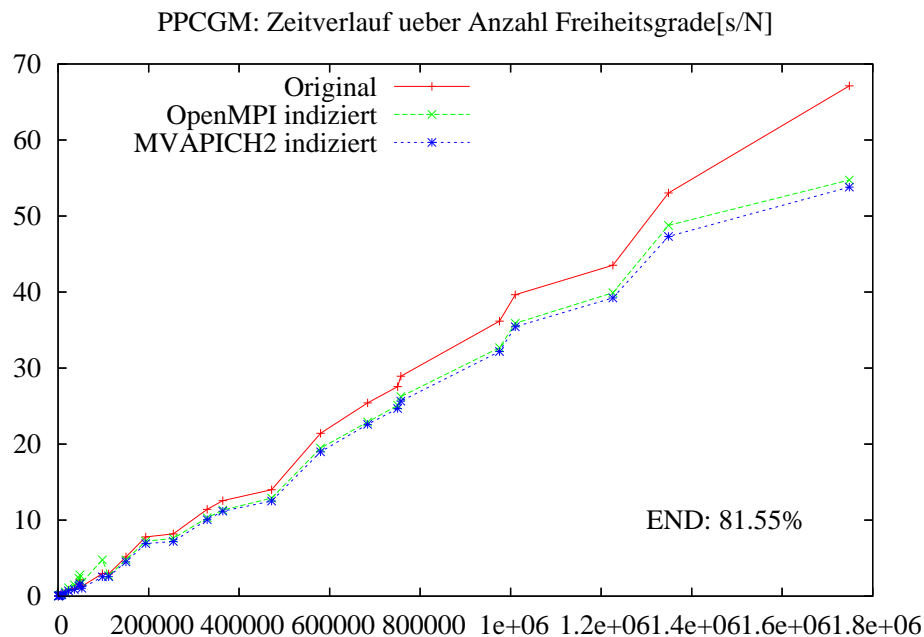


Abbildung 3.14: Zeitverlauf PPCGM MPI-Standards, indizierter Addition

Die Abbildung 3.14 zeigt den Vergleich des Zeitverlaufes des Programmes mit indizierter Addition für OpenMPI und MVAPICH2 sowie der Ausgangsimplementierung.

Beide MPI-Standards zeigen einen ähnlichen Zeitaufwand im Lösungsschritt des FEM-Programmes und unterscheiden sich nur geringfügig, insbesondere bei sehr stark verfeinerten Netzen.

Insgesamt hat die indizierte Addition eine viel geringere Zeitersparnis gebracht als erwartet. Verglichen mit der Ausgangsimplementierung wurde jedoch eine Verbesserung von 18,45% im letzten Schritt der Verfeinerung gegenüber der Ausgangsimplementierung erreicht.

3.4 Hypercube als neue Kommunikationsstruktur

Im Abschnitt 3.2 wurden die numerischen Ergebnisse nach der Einführung einer neuen Kommunikationsstruktur aufgezeigt. Im Zuge dessen wurde dort näher auf die Kommunikationstopologie für die Slaves eingegangen. Der Kommunikationsfluss zwischen den Slaves entspricht in

jenem und dem darauf folgenden Abschnitt 3.3 einer Sterntopologie.

Im Weiteren werden nun die numerischen Ergebnisse dargelegt, die bei der Wahl eines Hypercubes als Topologie gemessen wurden.

Ein Hypercube der Dimension d besteht aus 2^d Prozessoren. Da diese Topologie nur auf die Kommunikation innerhalb der Slaves angewendet wird, damit der Master-Prozessor im PCG-Algorithmus 1.1 voranschreiten kann, während die Slave-Prozessoren die Lösung \underline{v} berechnen, muss das Programm auf einer ungeraden Anzahl von Prozessoren gestartet werden. Aus Gründen der Vergleichbarkeit wurde deswegen das Programm von Anfang an auf 17 Prozessoren getestet. Dementsprechend benötigt man hier zur Darstellung des Kommunikationsflusses einen Hypercube der Dimension 4, bestehend aus 16 Prozessoren.

Nach der Implementierung des Hypercubes erhalten wir nun folgende Ergebnisse.

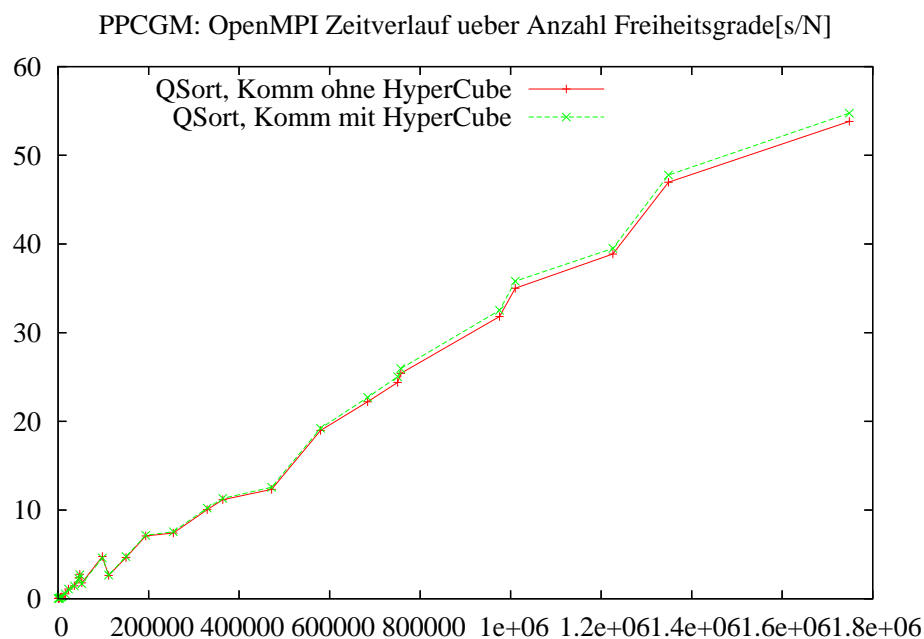


Abbildung 3.15: Zeitverlauf PPCGM OpenMPI, mit Hypercube

In der Abbildung 3.15 ist der Zeitverlauf des Lösungsschrittes im MPI-Standard OpenMPI bei suboptimaler Verteilung der Elementmatrizen und indizierter Summation der lokalen Lösungsvektoren einmal mit und einmal ohne den Hypercube als Kommunikationstopologie der Slaves dargestellt. Wie man darin gut erkennen kann, hat die Implementierung des Hypercubes keinen neuen Zeitgewinn gebracht.

In der Bemerkung 2.1 wurde der Kommunikationsaufwand für die Sterntopologie und den Hypercube betrachtet. Zusammen mit dem obigen Ergebnis und dieser Bemerkung lässt sich also folgern, dass die verwandte Verteilung der Elementmatrizen auf den Slave-Prozessoren nahezu optimal ist.

Für den zweiten getesteten MPI-Standard MVAPICH2 erhalten wir ein ähnliches Ergebnis.

Anzahl		PPCGM OpenMPI			geschätzter Fehler
Elemente	Knoten	Iterationen	Zeit in [s]		
			ohne HC	mit HC	
9520	97930	24	4.770	4.609	6.5e-05
32536	329468	28	10.039	10.230	4.8e-06
97468	975348	30	31.824	32.520	1.2e-06
176477	1748304	28	53.816	54.750	2.0e-07

Tabelle 3.10: Vergleich Zeitverlauf OpenMPI mit und ohne Hypercube

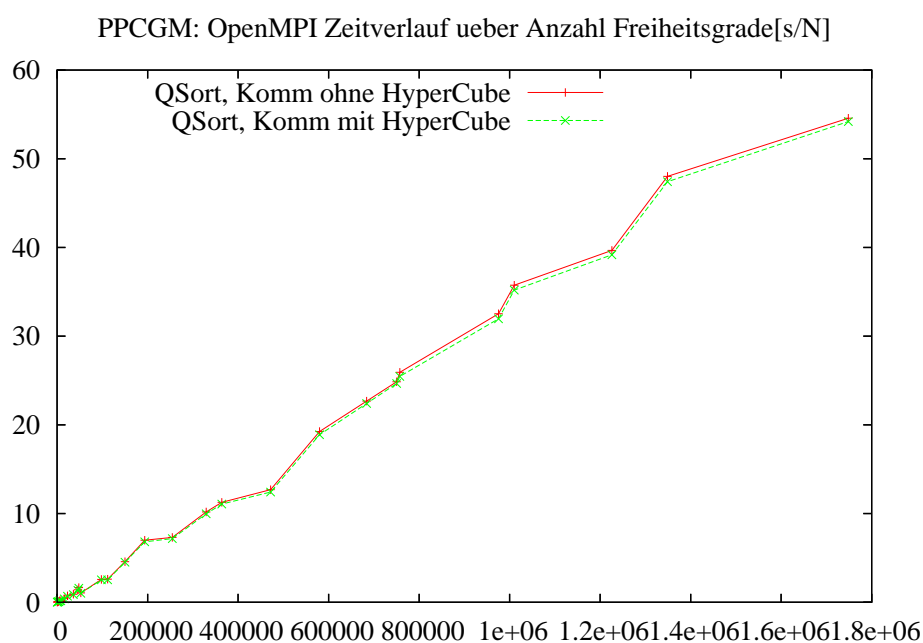


Abbildung 3.16: Zeitverlauf PPCGM MVAPICH2, mit Hypercube

Anzahl		PPCGM MVAPICH2			geschätzter Fehler
Elemente	Knoten	Iterationen	Zeit in [s]		
			ohne HC	mit HC	
9520	97930	24	2.574	2.521	6.5e-05
32536	329468	28	10.176	9.974	4.8e-06
97468	975348	30	32.531	31.955	1.2e-06
176477	1748304	28	54.566	54.197	2.0e-07

Tabelle 3.11: Vergleich Zeitverlauf MVAPICH2 mit und ohne Hypercube

Die Abbildung 3.16 zeigt den Graphen der Zeitmessung für den dritten Schritt im Farming-Konzept, das heißt dem Löser, im MPI-Standard MVAPICH2. Es wird der Graph bei induzierter Vektoraddition mit suboptimaler Verteilung angezeigt, einmal mit sternförmigen Kommunikationsfluss und einmal mit der Kommunikation über einen Hypercube.

Wie zuvor im Standard OpenMPI erhält man keine nennenswerte Reduzierung des Zeitaufwandes aufgrund der Bemerkung 2.1 und der Verteilung der Elementmatrizen.

Zum Abschluss dieses Kapitels werden noch einmal die Graphen der Zeitmessung im Schritt (3) des Farming-Konzepts für die beiden MPI-Standards unter Nutzung des Hypercubes zur Kommunikation und Summation des Lösungsvektors sowie dem Zeitverlauf im Lösungsschritt der Ausgangsimplementierung in einem Diagramm dargestellt.

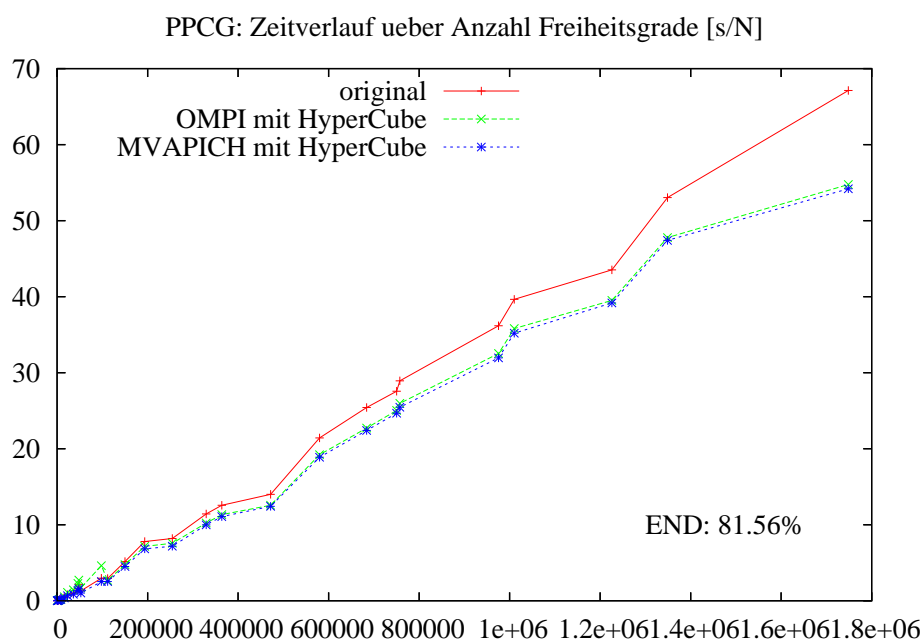


Abbildung 3.17: Zeitverlauf PPCGM MPI-Standards, mit Hypercube

Betrachtet man die blaue und die grüne Kurve in der Abbildung 3.17, so kann man eine recht deutliche Reduzierung der Laufzeit für das FEM-Programm unter Nutzung des Farming-Konzeptes erkennen.

Vergleicht man diese Ergebnisse mit denen aus der Abbildung 3.2, wo die Zeiten unter Verwendung der globalen Kommunikationsfunktion *MPI_REDUCE* dargestellt waren, so erzeugen sämtliche Optimierungsmöglichkeiten aus Abschnitt 2.4.3 kein verbessertes Zeitverhalten gegenüber der Implementierung mit der *MPI_REDUCE*-Funktion.

Dieses Ergebnis basiert auf der Tatsache, dass für die Implementierung der einzelnen Optimierungsmöglichkeiten aus Abschnitt 2.4.3 die sehr gut arbeitende, globale Kommunikationsfunktion *MPI_REDUCE* durch Punkt-zu-Punkt-Kommunikationsfunktionen ersetzt werden musste,

was erst einmal einen starken Zeitverlust verursachte, wie man in Abbildung 3.9 erkennen konnte.

Kapitel 4

Betrachtungen zur optimalen Anzahl von Prozessoren

In den beiden vorangegangenen Kapiteln wurde dargestellt, wie sich durch die Implementierung verschiedener Optimierungsmöglichkeiten die Laufzeit des parallelen FEM-Programmes reduzieren lässt. Die Art der Implementierung spielt natürlich eine sehr große Rolle bei der Laufzeitbetrachtung, ist aber nicht die einzige Möglichkeit zur Verringerung der Laufzeit.

So liegt es zum Beispiel recht nahe zu glauben, dass eine Erhöhung der Prozessoranzahl neben einer Erweiterung des Speichers auch eine Beschleunigung des parallelen Programmes mit sich bringt.

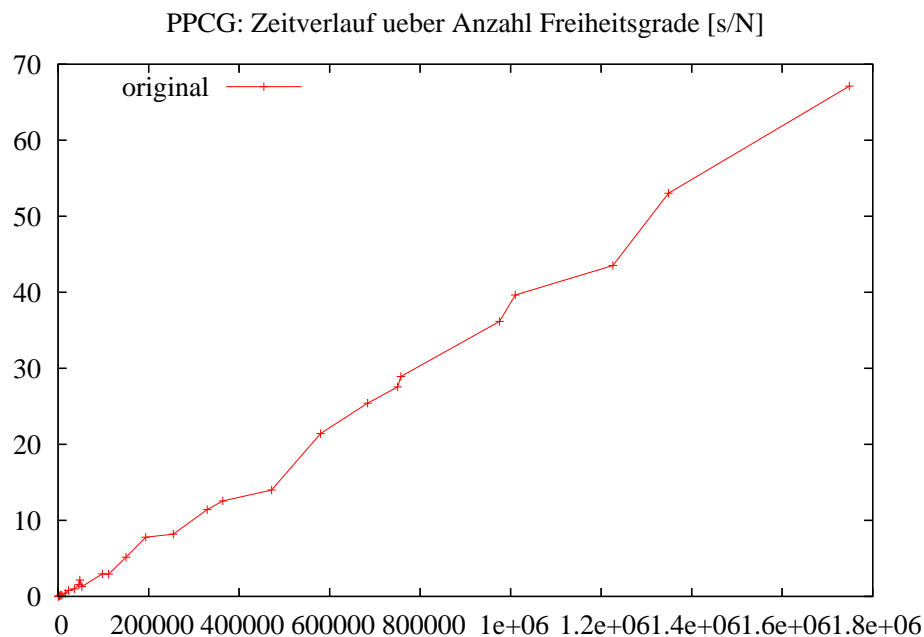
Dieses Kapitel beschäftigt sich also erneut mit der Aufwandbetrachtung für die parallele PCG-Methode aus dem Algorithmus 1.1 im Farming-Konzept, welches im Kapitel 2 vorgestellt wurde sowie mit der optimalen Prozessoranzahl für das FEM-Programm. Dazu werden die Laufzeiten für eine bestimmte, vorher festgelegte Anzahl von Elementen im FE-Netz betrachtet und für verschiedene Implementierungen verglichen.

4.1 Numerischer Aufwand für das Farming-Konzept

Betrachtet man die Zeitverläufe aus dem Kapitel 3, so sieht man deutlich einen annähernd linearen Verlauf des Graphen. Zur Betrachtung folgt hier noch einmal der Graph des Zeitverlaufs, wie er schon einmal in der Abbildung 3.4 zu sehen war.

Wir erhalten also eine annähernd lineare Abhängigkeit der Laufzeit von der Anzahl der Freiheitsgrade N . Welche Rolle spielt dabei aber die gewählte Anzahl der Prozessoren?

Dazu wird erneut der Aufwand im Algorithmus 1.1 für das Farming-Konzept betrachtet.



Der Aufwand im Farming-Konzept setzt sich zusammen aus

1. dem arithmetischen Aufwand $N a_1$ auf dem Master-Prozessor,
2. dem arithmetischen Aufwand $\frac{N}{p} a_2$ der Slave-Prozessoren,
3. dem Kommunikationsaufwand $N f(p) c$ zwischen den Prozessoren sowie
4. einem startup-Aufwand $p s$ für die Prozessoren.

Unter Vernachlässigung des Aufwandes für den Startup der Prozessoren erhalten wir also einen Aufwand

$$T = N a_1 + \frac{N}{p} a_2 + N f(p) c, \quad (4.1)$$

wobei $f(p)$ eine monoton wachsende Funktion, abhängig von der Anzahl der Prozessoren p ist, die durch die gewählte Kommunikationstopologie bestimmt wird.

Für die Sterntopologie gilt zum Beispiel

$$f(p) \sim \mathcal{O}(p)$$

und für den Hypercube

$$f(p) \sim \mathcal{O}(\log(p)),$$

wie schon in Abschnitt 2.4.3 b) angeführt wurde.

Aus der Gleichung (4.1) folgt

$$T = N \cdot \left(a_1 + \frac{a_2}{p} + f(p) c \right). \quad (4.2)$$

Dabei kann man die lineare Abhängigkeit des Zeitaufwandes T von der Anzahl der Freiheitsgrade N erkennen. Des Weiteren sieht man aber auch, dass die Laufzeit auch von der Anzahl der Prozessoren p abhängt. Um Aussagen über das Optimum der Prozessoranzahl machen zu können, muss also die Aufgabe

$$\left(a_1 + \frac{a_2}{p} + f(p) c \right) \longrightarrow \min \quad (4.3)$$

gelöst werden.

4.2 Optimale Wahl der Prozessoranzahl

Eine Möglichkeit, die optimale Prozessoranzahl zu bestimmen, ist die Lösung des Minimierungsproblem (4.3). Eine weitere Möglichkeit besteht darin, die Zeiten für eine bestimmte Elementanzahl im FE-Netz zu betrachten, die bei unterschiedlicher Wahl der Prozessoranzahl erreicht wurden.

Dazu erfolgten einige Messungen für drei verschiedene Implementierungsstufen, die in den nachfolgenden Abbildungen zu sehen sind und im Weiteren näher bestimmt werden. Der darin dargestellte Wert

Laufzeit / Freiheitsgrade / Iterationen im parallelen PCG

für jeweils eine bestimmte Anzahl von finiten Elementen im Verfeinerungsnetz bei 5, 9, 17, 33, und 65 Prozessoren entspricht gerade dem Wert, den man aus der Berechnungsvorschrift im Minimierungsproblem (4.3) für die jeweilige Anzahl p der Prozessoren erhält. Die Messpunkte sind durch Geraden miteinander verbunden.

In der letzten Implementierungsstufe wird ein Hypercube zur Organisation der Kommunikation zwischen den Slaves verwendet. Da in einem Hypercube immer eine Zweierpotenz als Anzahl der Prozessoren gewählt werden muss, erklären sich auch die fest gewählten Anzahlen für die Prozessoren p . Es gibt also darin einen Hypercube für die Slaves sowie den Master-Prozessor.

Den Messwerten in der Abbildung 4.1 liegt die Implementierungsstufe zugrunde, bei der die hauseigenen Kommunikationsroutinen durch die MPI-Funktionen ersetzt wurden. Dabei wird die neue Kommunikationstruktur, bei der die Slaves in einem Unterkommunikator operieren, genutzt und die Lösung des linearen Gleichungssystems (1.1) über die MPI-Funktion *MPI_REDUCE* berechnet. Dies entspricht also dem Stand der Implementierung wie in Abschnitt 3.2.

Die Graphen zeigen dabei ein konvexes Verhalten und man kann leicht sehen, dass die Wahl der

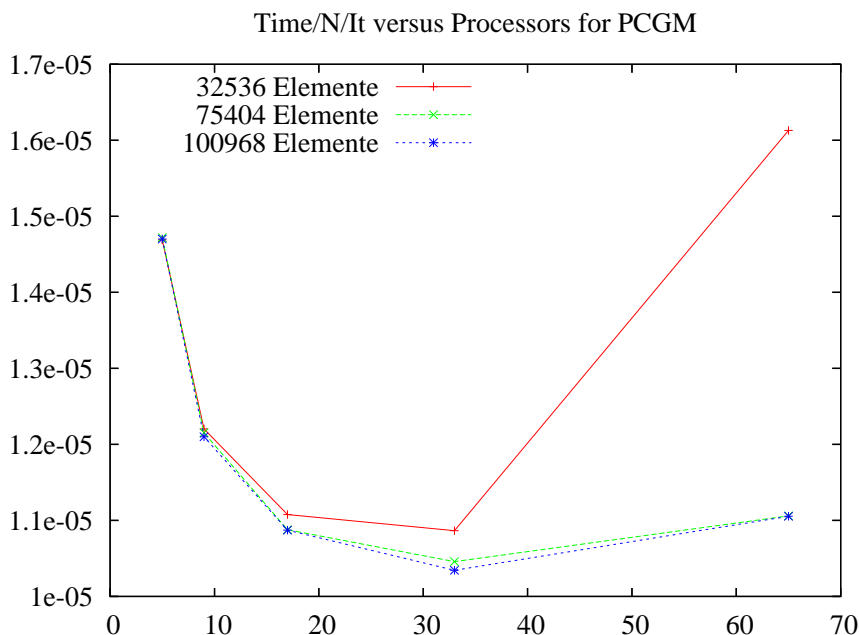


Abbildung 4.1: Laufzeitbetrachtung, unterschiedliche Anzahl Prozessoren, *MPI_REDUCE*

Anzahl Elemente	32536	75404	100968
Anzahl Prozessoren			
05	1.45e - 06	1.46e - 06	1.47e - 06
09	1.21e - 06	1.21e - 06	1.21e - 06
17	1.09e - 06	1.08e - 06	1.09e - 06
33	1.07e - 06	1.08e - 06	1.03e - 06
65	1.59e - 06	1.10e - 06	1.10e - 06

Tabelle 4.1: Messpunkte Zeitaufwand in der PPCG-Methode, *MPI-REDUCE*

Prozessoranzahl einen Einfluss auf die Laufzeit hat. Betrachtet man nun die erreichten Werte für 65 Prozessoren, so stellt man einen teilweise deutlich höheren Wert fest als bei 33 oder gar 17 Prozessoren. Desweiteren zeigt sich, dass 17 Prozessoren nicht die optimale Wahl für den Zeitaufwand sind. Mit 33 Prozessoren ist eine schnellere Berechnung möglich.

In der Abbildung 4.2 werden die Messwerte für die Implementierung der Sterntopologie innerhalb des Slave-Kommunikators mit indizierter Addition der vorsortierten lokalen Lösungsvektoren dargestellt. Auf Entwicklungsstufe dieser Implementierung wurde schon im Abschnitt 3.3 näher eingegangen.

Ähnlich wie in der Abbildung 4.1 verhalten sich alle Funktionen konvex. Es zeigt sich, dass für diese Implementierung die optimale Prozessoranzahl bei 17 Prozessoren liegt. Jedoch sind die

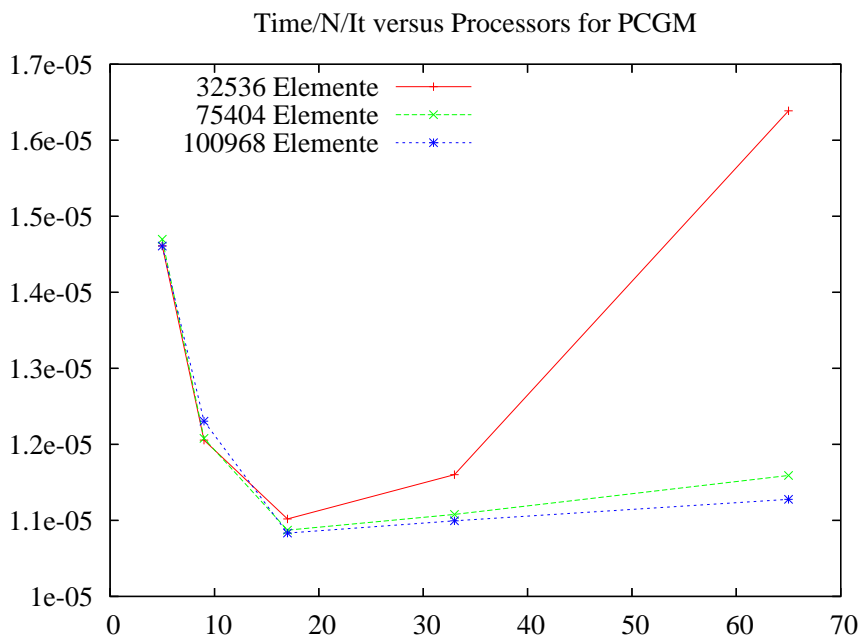


Abbildung 4.2: Laufzeitbetrachtung für unterschiedliche Anzahl Prozessoren, Addition indiziert

Anzahl Elemente	32536	75404	100968
Anzahl Prozessoren			
05	$1.44e - 06$	$1.46e - 06$	$1.46e - 06$
09	$1.19e - 06$	$1.20e - 06$	$1.23e - 06$
17	$1.09e - 06$	$1.08e - 06$	$1.08e - 06$
33	$1.15e - 06$	$1.10e - 06$	$1.10e - 06$
65	$1.62e - 06$	$1.15e - 06$	$1.13e - 06$

Tabelle 4.2: Messpunkte Zeitaufwand in der PPCG-Methode, Addition indiziert

Werte für 33 Prozessoren nur geringfügig schlechter.

Auch diese Abbildung zeigt deutlich, dass es ein Optimum für die Anzahl der Prozessoren p gibt. Dies beträgt hier 17 Prozessoren. Übersteigt die Anzahl der Prozessoren dieses Optimum, werden die Messwerte wieder schlechter.

Die Abbildung 4.3 stellt die Graphen für die Messwerte der letzten Implementierungsstufe dar, welche sämtliche Optimierungsmöglichkeiten aus Abschnitt 2.4.3 beinhaltet, also auch die Organisation der Kommunikation zwischen den Slaves als Hypercube. Die entsprechenden Laufzeiten für die parallele PCG-Methode für diese Implementierungsstufe sind im Abschnitt 3.4 dargestellt.

Die Graphen zeigen, wie auch in den Abbildungen zuvor, ein konvexes Verhalten, welches hier

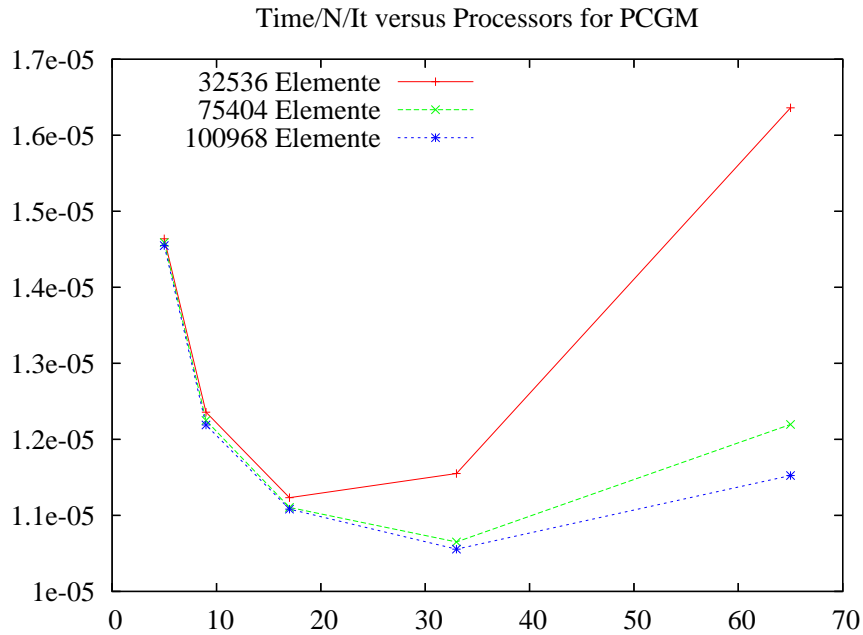


Abbildung 4.3: Laufzeitbetrachtung für unterschiedliche Anzahl Prozessoren, Hypercube

Anzahl Elemente	32536	75404	100968
Anzahl Prozessoren			
05	1.45e – 06	1.45e – 06	1.45e – 06
09	1.22e – 06	1.22e – 06	1.22e – 06
17	1.11e – 06	1.11e – 06	1.11e – 06
33	1.14e – 06	1.06e – 06	1.06e – 06
65	1.62e – 06	1.21e – 06	1.15e – 06

Tabelle 4.3: Messpunkte Zeitaufwand in der PPCG-Methode, Hypercube

auch wieder deutlich erkennbar ist. Die optimale Anzahl p an Prozessoren ist in dieser Abbildung aber nicht eindeutig. Für eine grobere Vernetzung mit weniger Elementen ist die optimale Prozessoranzahl $p = 17$, für feinere Vernetzungen ist sie $p = 33$.

Aufgrund des konvexen Verhaltens der Graphen ist auch hier wieder erkennbar, dass bei Über-, aber auch bei Unterschreitung der optimalen Prozessoranzahl die Messwerte teilweise sogar stark wachsen.

Zusammenfassend lässt sich sagen, dass eine optimale Prozessoranzahl p existiert, bei der die einzelnen Implementierungen am schnellsten berechnet werden. Für die hier dargestellten Implementierungen zeigt sich eine optimale Wahl für die Prozessoranzahl bei $p = 17$ Prozessoren für die ersten Verfeinerungsschritte und $p = 33$ Prozessoren, wenn die Vernetzung stärker verfeinert wird. Weicht man von der optimalen Prozessoranzahl ab, so muss man teilweise mit enormen

Laufzeiteinbußen rechnen.

Kapitel 5

Zusammenfassung

Ziel der vorliegenden Diplomarbeit war die Erarbeitung einer verbesserten Version für das an der TU Chemnitz bestehende Programm zur parallelen, numerischen Lösung von Elastizitätsproblemen. Dabei lag das Hauptaugenmerk auf der Verkürzung der Laufzeit. Dieses Ziel wurde durch die Kombination verschiedener Optimierungen erreicht.

Die verwendeten Optimierungen können in zwei Gruppen untergliedert werden. Zum einen sind mathematische Aufwandsbetrachtungen im Algorithmus 1.1 gemacht worden, die dann zu Änderungen im Ablauf des Algorithmus führten und damit einen Zeitgewinn erbrachten. Zum anderen konnte die Laufzeit durch die Nutzung der Optimierungsmöglichkeiten aus dem Bereich der Informatik weiter verbessert werden. Es zeigte sich, dass die Nutzung der globalen Kommunikationsroutine *MPI_Reduce* zusammen mit der Einführung der neuen Kommunikationsstruktur für die Slave-Prozessoren den höchsten Zeitgewinn erbrachten. Das Gros wurde dabei durch die MPI-Kommunikationsroutine erbracht, wohingegen nur ein kleiner Teil auf die Freisetzung des Master-Prozessors bei der Berechnung der Lösung des linearen Gleichungssystems (1.1) zurückzuführen ist.

Die weiteren ausgearbeiteten Optimierungsideen machten es notwendig, wieder auf die Punkt-zu-Punkt-Kommunikation zwischen den Prozessoren zurückzugreifen. Die Abbildung 3.9 zeigt deutlich den dadurch erlittenen Nachteil. Der darin dargestellte Zeitverlust gegenüber der Ausgangsimplementierung ist auf die Nutzung des darin verwendeten Hypercubes zurückzuführen, während erst einmal die Optimierungsmöglichkeiten unter Nutzung der Sterntopologie getestet wurden. Die indizierte Addition in Kombination mit der Optimierung der Elementeverteilung auf den einzelnen Slave-Prozessoren konnte einen Großteil des Zeitverlustes wieder aufholen, jedoch liegen die dadurch erreichte Laufzeit immer noch über der Laufzeit unter Nutzung der globalen MPI-Kommunikationsroutine.

Die Implementierung eines Hypercubes mit der Nutzung der indizierten Addition und der optimierten Verteilung der Elemente auf den Slave-Prozessoren brachte keinen weiteren Zeitgewinn. Nach der Bemerkung 2.1 war auch ein Zeitgewinn durch Nutzung des Hypercube bei einer optimalen Verteilung der Elemente zur Berechnung der Element-Steifigkeitsmatrizen auf den Slave-Prozessoren nicht zu erwarten. Es zeigt sich dadurch, dass die gewählte Verteilung nahe am Optimum ist.

Im letzten Kapitel wurde dann die Auswirkung der Prozessoranzahl auf die Laufzeit des Pro-

grammes in seinen unterschiedlichen Implementierungsstufen untersucht. Es zeigte sich, dass eine optimale Prozessoranzahl existiert und die Laufzeiten teilweise deutlich stiegen, wenn man diese unter- oder überschritt. Das Optimum liegt für eine grobe Vernetzung bei 17 Prozessoren. Für feinere Vernetzungen sind 33 Prozessoren zu nutzen.

Abbildungsverzeichnis

2.1	Programmablaufplan für die adaptive FEM	15
3.1	Beispiel: Grobnetz loch3D	25
3.2	Netzverfeinerung und Fehler	26
3.3	Zeitverlauf Assemblierung Ausgangsimplementierung	27
3.4	Zeitverlauf PPCGM Ausgangsimplementierung	28
3.5	Zeitverlauf PPCGM Ausgangsimplementierung, MPI-Standards	29
3.6	Kommunikationsstruktur mit Slave-Kommunikator	31
3.7	Zeitverlauf PPCGM Ausgangsimplementierung, MPI-Standards, neue Kommunikationsstruktur	32
3.8	Zeitverlauf PPCGM MPI-Standards, neue Kommunikationsstruktur	32
3.9	Zeitverlauf PPCGM Ausgangsimplementierung, OpenMPI mit Sterntopologie . .	34
3.10	Zeitverlauf PPCGM OpenMPI, indizierter Addition	35
3.11	Zeitverlauf PPCGM OpenMPI indiziert, sortiert, unsortiert	36
3.12	Zeitverlauf PPCGM MVAPICH2, indizierter Addition	37
3.13	Zeitverlauf PPCGM MVAPICH2 indiziert, sortiert, unsortiert	38
3.14	Zeitverlauf PPCGM MPI-Standards, indizierter Addition	39
3.15	Zeitverlauf PPCGM OpenMPI, mit Hypercube	40
3.16	Zeitverlauf PPCGM MVAPICH2, mit Hypercube	41
3.17	Zeitverlauf PPCGM MPI-Standards, mit Hypercube	42
4.1	Laufzeitbetrachtung, unterschiedliche Anzahl Prozessoren, <i>MPI_REDUCE</i>	47
4.2	Laufzeitbetrachtung für unterschiedliche Anzahl Prozessoren, Addition indiziert .	48
4.3	Laufzeitbetrachtung für unterschiedliche Anzahl Prozessoren, Hypercube	49

Tabellenverzeichnis

3.1	Zeitverlauf PPCGM Ausgangsimplementierung	28
3.2	Vergleich Zeitverlauf Ausgangsimplementierung mit MPI-Standards	30
3.3	Vergleich Zeitverlauf Ausgangsimplementierung und MPI-Standards mit neuer Kommunikationsstruktur	31
3.4	Vergleich Zeitverlauf MPI-Standards mit neuer Kommunikationsstruktur	33
3.5	Vergleich Zeitverlauf Ausgangsimplementierung mit OpenMPI, Sterntopologie	34
3.6	Vergleich Zeitverlauf PPCGM OpenMPI indiziert, nicht indiziert	35
3.7	Vergleich Zeitverlauf OpenMPI sortiert, unsortiert	37
3.8	Vergleich Zeitverlauf PPCGM MVAPICH2 indiziert, nicht indiziert	38
3.9	Vergleich Zeitverlauf MVAPICH2 sortiert, unsortiert	38
3.10	Vergleich Zeitverlauf OpenMPI mit und ohne Hypercube	41
3.11	Vergleich Zeitverlauf MVAPICH2 mit und ohne Hypercube	41
4.1	Messpunkte Zeitaufwand in der PPCG-Methode, MPI-REDUCE	47
4.2	Messpunkte Zeitaufwand in der PPCG-Methode, Addition indiziert	48
4.3	Messpunkte Zeitaufwand in der PPCG-Methode, Hypercube	49

Literaturverzeichnis

- [1] P. K. Jimack, N. Touheed, *An Introdoction to MPI for Computational Mechanics* in: *Parallel and Distributed Processing for Computational Mechanics: Systems and Tools*, Topping B. H. V. eds., Saxe-Coburg Publications, Edingburgh 1999, S. 24-45
- [2] Arnd Meyer, *Basic Approach to Parallel Finite Element Computations: The DD Data Splitting* in: *Parallel Algorithms and Cluster Computing*, Karl Heinz Hoffmann, Arnd Meyer eds., Springer-Verlag, Berlin 2006
- [3] D. Braess : *Finite Elemente-Theorie, schnelle Löser und Anwendungen in der Elastizitätstheorie*, Springer-Verlag, 1997, 2. Auflage

In der vorliegenden Diplomarbeit wurde ein Programm zur parallelen Berechnung von Finite-Elemente-Problemen betrachtet, das wesentlich eine Master-Slave-Topologie zur Parallelisierung der rechenaufwändigsten Teile nutzt. Es wurden Optimierungsmöglichkeiten zur Verringerung der Laufzeit erarbeitet und in das vorhandene Programm integriert. Zur Dokumentation der Verbesserungen wurden für die verschiedenen Entwicklungsstufen Zeitmessungen durchgeführt und grafisch dargestellt.

Thesen

Das Ersetzen der Kommunikationsroutinen der Fakultät für Mathematik an der TU Chemnitz durch die Funktionen der Programmbibliothek MPI zeigt einen deutlichen Zeitgewinn. Im letzten Verfeinerungsschritt werden nur noch 82,34% der Zeit zur Berechnung der parallelen PCG-Methode benötigt.

Die Einführung einer neuen Kommunikationsstruktur, die den Master aus der Berechnung der Matrixmultiplikation ausschließt, damit er im PCG-Algorithmus 1.1 voranschreiten kann, bringt eine weitere Verbesserung. Die Laufzeit für die parallele PCG-Methode in der letzten Verfeinerung sinkt auf 79,52% der Laufzeit in der Ausgangsimplementierung.

Eine Aussage zur genutzten Kommunikationstopologie kann nicht gemacht werden, da zur Kommunikation und Summation der Lösung die MPI-Routine *MPI_REDUCE* verwendet wurde, die zu den globalen Kommunikationsroutinen zu zählen ist.

Die Implementierung der indizierten Addition implizierte das Ersetzen der globalen Kommunikation zwischen den Slave-Prozessoren durch die sogenannte Punkt-zu-Punkt-Kommunikation. Dadurch verlängerte sich die Laufzeit für die parallele PCG-Methode gegenüber der vorangegangenen Implementierung wieder, blieb jedoch mit 81,55% der Laufzeit im letzten Verfeinerungsschritt deutlich unter der Laufzeit in der Ausgangsimplementierung. Neben der indizierten Addition ist bei diesen Prozentangaben auch schon eine Optimierung der Verteilung der Elementmatrizen auf den Slave-Prozessoren berücksichtigt. Zur Kommunikation zwischen den Slave-Prozessoren wurde eine Sterntopologie implementiert.

Die Änderung der Kommunikationstopologie zu einem Hypercube brachte kaum eine Veränderung der Laufzeiten gegenüber der letzten Implementierung mit Sterntopologie. Mit 81,56% der Laufzeit zur Berechnung der PCG-Methode im letzten Verfeinerungsschritt gegenüber der Ausgangsimplementierung sind die Laufzeiten nahezu gleich geblieben.

Die Anzahl der verwendeten Prozessoren spielen bei der Laufzeitbetrachtung auch eine große Rolle. Für die Implementierungen zeigte sich, dass in den ersten Verfeinerungsschritten 17 Prozessoren optimal, bei den weiteren Verfeinerungsschritten aber 33 Prozessoren besser sind. Wählt man eine geringere oder größere Prozessoranzahl, so verschlechtert sich die Laufzeit teilweise sogar enorm.

Ich erkläre an Eides Statt, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Chemnitz, den 19. August 2007

Jens Rückert