Hochschule für Angewandte Wissenschaften Hamburg
*Hamburg University of Applied Sciences*

# Bachelorarbeit

Stefan Borries

## Beschleunigung der Werteübertragung bei neuvernetzten Geometrien von Finiten Elementen

*Fakultät Technik und Informatik*
*Department Informatik*

*Faculty of Engineering and Computer Science*
*Department of Computer Science*

Stefan Borries

# Beschleunigung der Werteübertragung bei neuvernetzten Geometrien von Finiten Elementen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Stephan Pareigis
Zweitgutachter : Prof. Dr. -Ing. Franz Korf

Abgegeben am 30. Mai 2008

**Stefan Borries**

## Thema der Bachelorarbeit

Beschleunigung der Werteübertragung bei neuvernetzten Geometrien von Finiten Elementen

## Stichworte

Parallelisierung, Binärer Suchbaum, Bounding Volumes, OpenMP, Nearest Neighbour, PCM

## Kurzzusammenfassung

Ein vorhandener Algorithmus wurde parallelisiert. Neue Algorithmen wurden implementiert, um eine Beschleunigung bei der Werteübertragung bei neuvernetzten Geometrien von Finiten Elementen zu erreichen. Verschiedene Finite-Element-Modelle wurden getestet und das Laufzeitverhalten verglichen.

**Stefan Borries**

## Title of the paper

Acceleration of the remapping of remeshed geometries of Finite Elements

## Keywords

Parallelisation, Binary Search Tree, Bounding Volumes, OpenMP, Nearest Neighbour, PCM

## Abstract

A given algorithm was parallelised. New Algorithms were implemented to achieve an acceleration of the remapping of remeshed geometries of Finite Elements. Different Finite-Element-Models were tested and the runtimebehaviour compared.

# Danksagung

Alle Menschen, die mich während meines Studium und bei der Durchführung dieser Arbeit unterstützt haben, gebührt mein Dank. Alle Namen hier aufzuführen würde den Rahmen sprengen.

Besonders Peter Kraft und Dirk Bächle von der Firma *Femutec GmbH* sollen hier erwähnt werden. Sie haben mich tatkräftig beim Erstellen meiner Abschlussarbeit unterstützt.

# Inhaltsverzeichnis

# Tabellenverzeichnis

# Abbildungsverzeichnis

# 1 Einleitung

## 1.1 Beschreibung des Problems

In Umformsimulationen nach der Finite Elemente Methode(*FEM*) werden meist diskrete Raumgitter (Netze) für die Beschreibung der Geometrien verwendet. Wird während der Berechnung ein Bauteil stark deformiert, so ergeben sich für die einzelnen Elemente zum Teil sehr spitze Winkel. Diese wirken sich nachteilig auf die numerische Stabilität des Gleichungssystems aus. Zur Vermeidung dieses Problems werden die Geometrien während der Umformsimulation erneut vernetzt. Es wird ein neues Gittermodell erzeugt, das die Form des Körpers weiterhin gut beschreibt, dabei aber weniger Verzerrungen in den einzelnen Elementen aufweist.
In diesem Schritt werden die Ergebniswerte wie *Temperatur*, *Spannungen* usw. vom alten Netz auf das neue übertragen. Die Werte sind grundsätzlich an Knoten oder Elemente gebunden. Diese haben durch die Neuvernetzung eine andere räumliche Lage. Zudem können Elemente neu hinzugekommen oder weggefallen sein. Altes und neues Netz sind also nur bedingt kongruent. Das Übertragen der Werte (auch *Remapping* oder *Rezoning* genannt), und die damit meist verbundene Interpolation bzw. Mittelwertbildung für die einzelnen Elemente, führt zu einem Informationsverlust, der möglichst gering gehalten werden sollte. Das bei der *Femutec GmbH* eingesetzte Programm *simufact.forming* zur Simulation von Umformprozessen erstellt hierzu ein Zwischennetz. Dieses Netz zerlegt den einschließenden Raum des Objektes in möglichst viele kleine Hexaeder. Die Ergebnisse werden zuerst von der alten Geometrie auf das Würfelgitter übertragen und von dort aus auf die neue diskrete Beschreibung des Körpers. Dieser speicher- und rechenintensive Schritt kann für ein einzelnes Bauteil komplexer Größe (ca. 400.000 Elemente) bis zu einer halben Stunde dauern und führt zu einer merklichen Verzögerung des Programms für den Benutzer.

## 1.2 Ziel der Arbeit

Im idealen Fall läuft das *Remapping* so schnell ab, dass der Benutzer es nicht bemerkt. Da dies Aufgrund der Komplexität der Berechnungen jedoch nicht wahrscheinlich ist, soll die

Verzögerung minimiert werden.
Im ersten Schritt der Arbeit steht die Parallelisierung des derzeit vorhandenen sequentiellen Algorithmus mit Threads (Shared-Memory-Architektur) im Mittelpunkt. Hiermit soll die Berechnung auf - heutzutage üblichen - Multi-Core-Computern beschleunigt werden. Hierzu ist zunächst zu prüfen inwieweit der gegebene Fortran-Quellcode dafür geeignet ist. Nach der erfolgreichen Aufteilung in mehrere Threads beliebiger Anzahl sollte in entsprechenden Laufzeituntersuchungen eine passende Partitionierung gefunden werden. Hier ist das Teilziel, in einer Art Vorstufen-Modul anhand der Geometriegröße (Anzahl der Elemente) zu entscheiden, ob überhaupt eine Aufteilung vorgenommen wird und gegebenenfalls eine günstige Partitionierung zu wählen.
In einem zweiten Schritt sollen weitere Beschleunigungstechniken untersucht, implementiert und bewertet werden. In entsprechenden Tests sind dann die Unterschiede zwischen den beiden Verfahren hinsichtlich der Geschwindigkeit und Genauigkeit herauszuarbeiten und darzustellen.

## 1.3 Einführung in die Theorie der Finite Elemente Methode

Die Finite Element Methode (*FEM*) wurde erstmals 1956 von Turner, Clough, Martin und Topp als solches behandelt. Die theoretische Vorarbeit wurde bereits am Anfang des 19.Jahrhunderts von Walter Ritz (1878-1909) erarbeitet und ist als das Ritz'sche Verfahren oder auch als Ritz'sches Variationsprinzip bekannt [Ste92].
Bei der *FEM* handelt es sich um ein Verfahren mit dem Verformungen in der Technischen Mechanik berechnet werden können. Im Allgemeinen wird die *FEM* dann eingesetzt, wenn der Bau von Prototypen teuer, zeitaufwendig oder unmöglich wäre und eine Simulation einen effizienteren Einblick in mögliche Verbesserungen geben könnte.
Beispiele hierfür sind:

- Crashtests an PKWs
- Pressung von Schrauben (Ermittlung der Möglichkeit den Verschnitt zu minimieren)
- Berechnung der Beanspruchung eines Staudamms
- Verschleissberechnung von Bauteilen

Solche geometrischen Modelle können nicht mit einer mathematischen Formel beschrieben werden, da das Problem so, wie es sich im Gesamten darstellt, nicht zu formulieren ist. Aus diesem Grund wird das Objektmodell in viele kleine Teile zerlegt, die Finiten Elemente. Das durch sie dargestellte Teilproblem ist mathematisch berechenbar.

Die daraus resultierenden Teilergebnisse werden zu einem Gesamtergebnis zusammengefasst, welches eine Annäherung an ein reales Ergebnis darstellt. Es handelt sich somit bei der FEM um ein Approximationsverfahren. Die Ergebnisse stellen eine Näherung dar und sollten stets kritisch betrachtet werden.

Bei der *FEM* wird ein realer Körper durch verschiedene geometrische Körper repräsentiert, wobei jeder dieser geometrischen Körper ein Finites Element symbolisiert. Die Erstellung der Modelle der realen Körper geschieht in CAD-Programmen. Die Daten werden an das Finite Element Programm exportiert. Später werden die Ergebnisse grafisch dargestellt. Die Aufteilung des Modells in die Finiten Elemente wird dabei entweder vom CAD- oder vom eigentlichen *FEM*-Programm durchgeführt. Da ein steigendes Interesse an der *FEM* deutlich wird, integrieren immer mehr CAD-Anbieter die Aufteilung der Geometrien in Finite Elemente in ihre Software oder implementieren eigene Finite Element-Solver (FE-Solver).

Bei den Finiten Elementen handelt es sich um Körper mit den Dimensionen 1 bis 3. Mit höherer Dimension steigt die Genauigkeit, aber auch die Zeit bis zur Berechnung des Ergebnisses. In Abb.1.1 werden exemplarisch drei Elemente dargestellt. Kreise verdeutlichen die Eckpunkte in der Abbildung. Geschnittene Kreise, wie beim Plattenelement stellen zusätzliche Knoten des Körpers dar. Mit ihnen wird eine Erhöhung der Genauigkeit erzielt. Heutzutage werden bevorzugt Tetraederelemente eingesetzt, um reale Körper abzubilden.

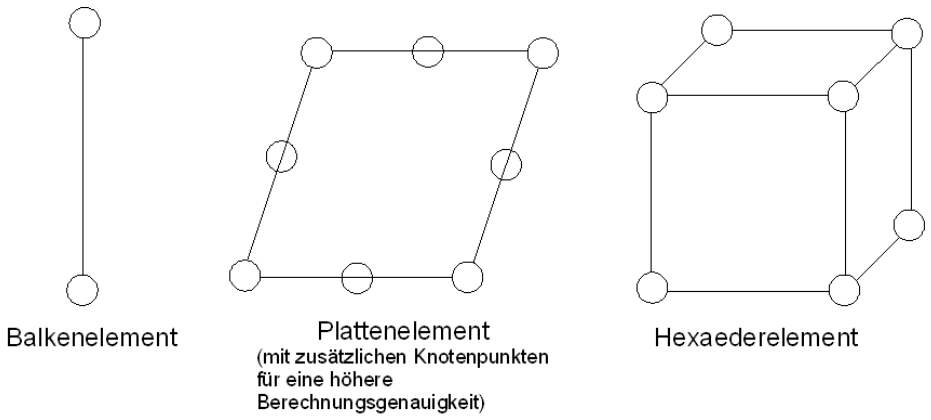



Abbildung 1.1: Beispiele für Finite Elemente der Dimensionen 1 bis 3

Sie weisen eine genügend hohe Genauigkeit auf und führen im Vergleich mit Hexaederelementen schneller zum gewünschten Ziel.
Um eine höhere Genauigkeit erreichen zu können, gibt es folgende Ansätze.

- Die Vernetzung eines abgebildeten Körpers kann den Anforderungen angepasst werden. Orte höheren Interesses werden feiner vernetzt als Orte niedrigeren Interesses (siehe Abb.1.4).

- Es werden zusätzliche Zwischenpunkte in die Elemente eingebracht, um eine höhere Ordnung und damit eine höhere Genauigkeit zu erreichen (siehe Abb.1.1 *Plattenelement*).
- Es wird eine Harmonisierung der Elementformen durchgeführt, d. h. Elemente sollten ihrem Ideal entsprechen und nicht verzerrt sein (siehe Abb.1.2).

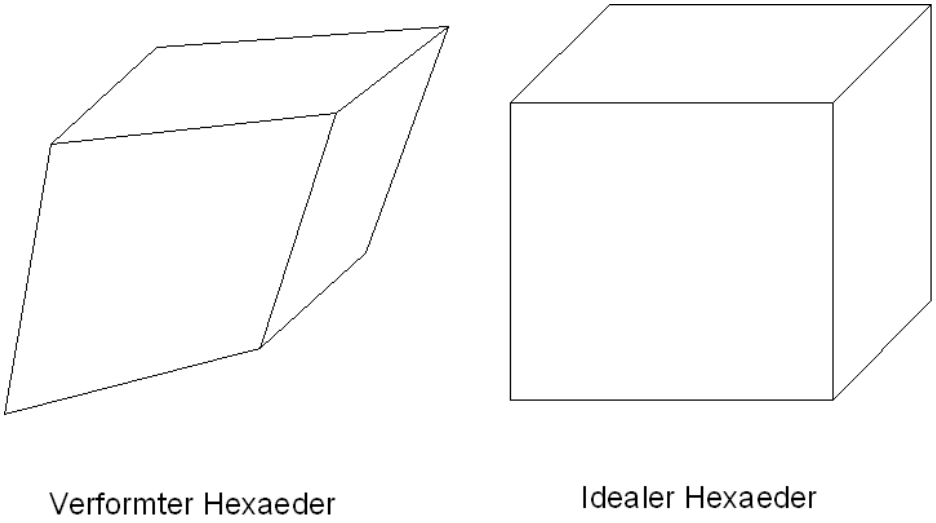



Abbildung 1.2: Beispiele für verzerrten und idealen Hexaeder

Finite Elemente bestehen aus Knoten, welche den Eckpunkten des geometrischen Körpers entsprechen. So werden z.B. für ein Hexaederelement die Koordinaten seiner 8 Knoten gespeichert. In einem Feld wird festgelegt, welche Knoten zu einem Element gehören (siehe Tab.1.1). In einem zweiten Feld werden die Koordinaten der Knoten gespeichert (siehe

| Tetraeder | Pentaeder | Hexaeder |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 4 | 8 |
| 3 | 3 | 3 |
| 4 | 2 | 5 |
| | 5 | 4 |
| | | 6 |
| | | 7 |
| | | 2 |

Tabelle 1.1: 3D-Elemente mit Knotenzugehörigkeit

| Knoten | x | y | z |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 2 | 2.4 | -2 | 0.5 |
| 3 | -2.3 | -3 | -2.5 |
| 4 | 7.4 | 4 | -3.3 |
| 5 | 0.5 | 0 | 0 |
| 6 | 1.1 | 0.6 | 9.6 |
| 7 | 2.07 | 3.7 | 1.2 |
| ... | ... | ... | ... |

Tabelle 1.2: Koordinaten der Knoten

Tab.1.2). So erhält man zwei Felder, welche die Geometrie des Netzes abbilden. Das erste

Feld beinhaltet alle Knoten (Eckpunkte) der Elemente. Im zweiten Feld sind die Kanten der Elemente gespeichert. In Abb.1.3 sind die Eckpunkte aufsteigend von 0 bis 7 nummeriert. In Klammern stehen die Knoten, welche über die Beziehung mit dem Feld aus Tab.1.1 in Verbindung gesetzt werden. Die Knoten haben festgelegte Verbindungen (siehe Abb.1.3). Knoten 1 wird dem Feld 0 zugeordnet. Die Anordnung in diesem Beispiel entspricht nicht

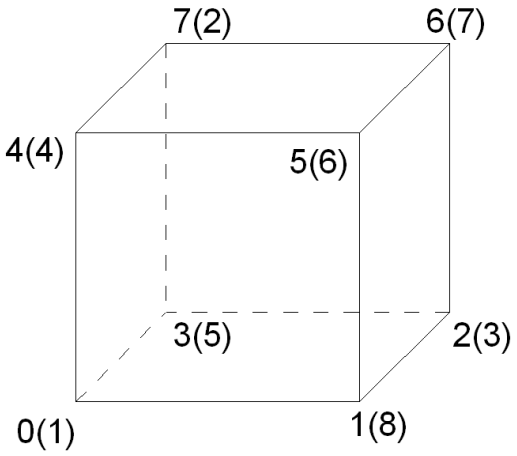



Abbildung 1.3: Zuordnung zwischen Eckpunkten und Knotennummer eines Elements

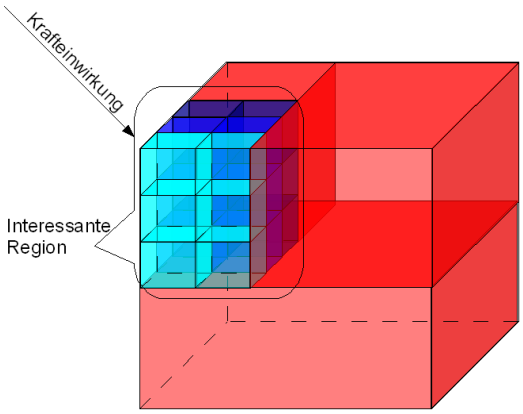



Abbildung 1.4: Beispiel eines verfeinerten Modells

den Koordinaten aus der Tab.1.2. Es soll lediglich das Prinzip verdeutlicht werden.

Möchte man die Genauigkeit der Ergebnisse erhöhen, kann man weitere Knotenpunkte einsetzen, welche sich dann in der Mitte der Kanten zwischen den Knoten befinden (siehe Abb.1.1). Dementsprechend müssen die Felder dann erweitert werden.
Die eigentlichen interessanten Ergebniswerte befinden sich im Integrationspunkt. Dieser Punkt kann, muss sich aber nicht im Mittelpunkt des geometrischen Körpers befinden. In der Praxis werden die Ergebniswerte häufig auf den Knotenpunkten angenommen, da man so eine Interpolation der Werte auf den Integrationspunkt vermeiden kann.

Wird eine Kraft auf das geometrische Netz ausgeübt, kommt es zu Verformungen. Ab einem bestimmten Grad an Verformung gelten die mathematischen Formeln für die Finiten Elemente nicht mehr. Daher müssen die geometrischen Formen der Finiten Elemente harmonisiert werden [Frö05]. Dies ist auch unter den Namen *Remeshing* bekannt.
Wenn das alte Netz so verschoben ist, dass die mathematischen Formeln nicht mehr gelten, muss ein neues Netz erstellt werden.

Dieses neue Netz ist mit dem alten Netz nicht Deckungsgleich. Es stellt sich daher die Frage, welche Ergebniswerte des alten Finiten Elementes einem neuen Finiten Element zugewiesen werden? Oder bekommt das neue Finite Element einen berechneten Wert aus

Ergebnisswerten von verschiedenen Finiten Elementen des alten Netzes? Diese Übertragung von Werten wird *Remapping* genannt. *Remapping* wird ebenfalls verwendet, wenn im neuen Netz mehr oder weniger Elemente als im Alten sind. Dies kann der Fall sein, wenn eine Verfeinerung vorgenommen wurde. Eine solche Verfeinerung wird vorgenommen, wenn man an einem Teil des Modells eine höhere Genauigkeit der Ergebnisse erhalten möchte. Dementsprechend können Teile der Geometrie gröber neu vernetzt werden, wenn sie von geringerem Interesse sind. Ist eine Geometrie grob vernetzt lässt sie sich schneller berechnen. Es besteht also ein Zusammenhang zwischen dem Aufbau des Netzes und der Berechnung der Ergebniswerte.
Ein Beispiel hierfür ist eine Eisenstange, die nur an einem Ende gebogen werde soll. An dem Ende, an dem die Verformung simuliert wird, ist das Modell der Eisenstange verfeinert und besteht aus hunderten von Finiten Elementen. An dem Ende der Stange welches nicht verformt wird und welches von niedrigem Interesse ist, besteht das Modell nur aus wenigen Elementen. Unter Umständen nur aus einem.
Da die Aufteilung des Modells in Finite Elemente stark davon abhängt, was man untersuchen möchte, kann auch das nicht zu verformende Ende mit vielen Finiten Elementen modelliert sein. Dies könnte der Fall sein, wenn man die inneren Spannungen des Materials ermitteln möchte, die sich durch die Verformung am anderen Ende ergeben.

## 1.4 Definitionen

Multi-Prozessor-System
: Ein System mit mehreren Prozessoren. Jeder Prozessor hat seinen eigenen Cache.

Multi-Core-System
: Ein Prozessor mit mehreren Prozessorkernen. Die Kerne teilen sich einen gemeinsamen Cache (L2).

Thread
: Ein Thread ist ein Programm, welches von einem übergeordneten Programm (Prozess) aufgerufen wird. Der Thread kann auf den dem Prozess zugesicherten Speicher zugreifen. Er läuft in dem Kontainer des Prozesses ab. Ein Prozess kann mehrere Threads erschaffen, die unterschiedliche Aufgaben erfüllen können. Kommunikation kann auf unterschiedliche Art und Weise zwischen den Threads stattfinden. Die Threads können über den gemeinsamen Speicher des Prozesses miteinander in Verbindung treten. Eine weitere Möglichkeit besteht darin, Nachrichten untereinander auszutauschen.
  Beim Arbeiten auf dem gemeinsamen Speicher kann es geschehen, dass es zu Inkonsistenzen in den Daten kommt. Wenn zwei Threads ein Datum schreiben möchten, ist

nicht bestimmbar, welches Datum am Ende geschrieben wurde.
Hier müssen Zugriffskontrollen durchgeführt werden.

Unabhängigkeit von Threads
: Zwei Threads sind unabhängig voneinander, wenn die Ausführung ihrer Arbeiten den jeweils anderen nicht beeinflusst. Entweder arbeiten sie auf unterschiedlichen Daten oder greifen nur lesend auf gemeinsame Daten zu.

Wirkliche parallele Abläufe
: Zwei Abläufe laufen wirklich parallel, wenn es zwei ausführende Einheiten gibt und jeder Thread von einer Einheit bearbeitet wird. Diese Einheiten sind entweder Prozessoren oder Prozessorkerne.
Zwei Threads können auch von nur einer Einheit bearbeitet werden, allerdings liegt dann keine wirkliche Parallelität vor, da immer nur ein Thread zur Zeit ausgeführt wird, während der andere auf seinen Einsatz wartet. Da Zeit vergeht, wenn von einem Thread zum nächsten gewechselt wird (laden des Speicherbereiches des neuen Threads, suspendieren des alten Threads), kommt es zu einem schlechteren Laufzeitverhalten.
Eine Aufgabe auf mehrere Threads zu verteilen bringt daher nur dann Vorteile in der Laufzeit, wenn die Threads auch wirklich parallel ablaufen können.

Synchronisationstechniken
: Unter Synchronisation versteht man, das Abstimmen von Vorgängen, Daten usw.
Im Zusammenhang mit Threads gibt es zwei Synchronisationen auf die man achten muss.
Threads können sich untereinander synchronisieren. Thread B wartet die Freigabe der Variablen f oder führt andere Berechnungen aus (siehe Abb.1.5).
Arbeiten Threads auf den gleichen Daten, aber auf unterschiedlichen Prozessoren, kann es zu Problemen mit der Synchronizität der Daten kommen. Jeder Prozessor hat seinen eigenen Cache. Thread A läuft auf Prozessor 1, Thread B auf Prozessor 2. Die Variable f liegt jeweils im Cache der Prozessoren. Wird der Cache nicht synchronisiert, kann es sein, dass A den gewünschten Wert von f an den Server schickt. Wurde hingegen bereits eine Synchronisation durchgeführt, schickt A den Wert, den B geschrieben hat an den Server.
Dies kann von dem Programm nicht beinflusst werden. Die Architektur hat dafür zu sorgen, dass die Daten konsistent vorliegen.

Inkonsistente Daten
: Sind Threads nicht unabhängig voneinander, kann es passieren, dass sie zur gleichen Zeit auf denselben Daten arbeiten (siehe Abb.1.6).
In der Abbildung wird gezeigt, dass Thread A eine Variable liest, um sie an einen Server zu schicken. Nun braucht aber der Aufbau der Verbindung zum Server etwas

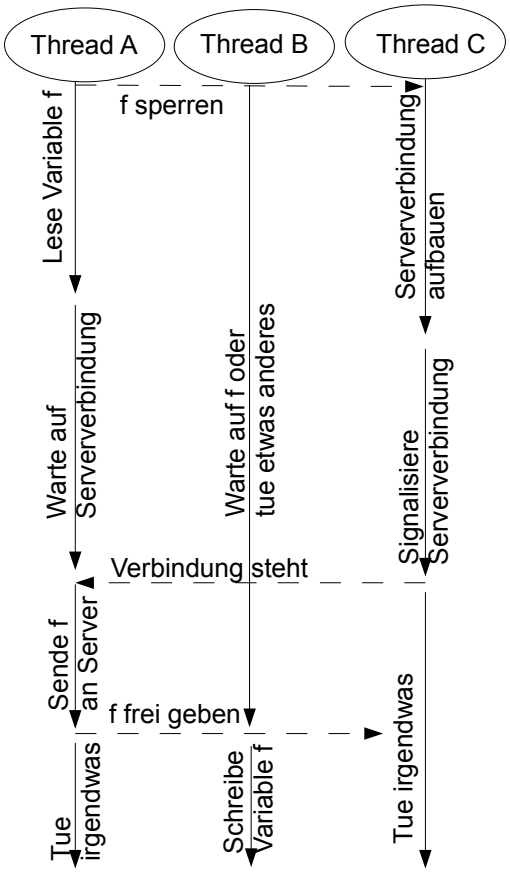



Abbildung 1.5: Synchronisation von Daten

länger. In der Zwischenzeit schreibt Thread B seine Daten in die Variable f. Wenn Thread A Glück hat, sind die korrekten Daten noch im Cache gespeichert. Es gibt aber auch den Fall, dass f bereits synchronisiert wurde. Dann stehen in f die Daten, die B hineingeschrieben hat. Thread A würde nicht die korrekten Daten an den Server schicken.

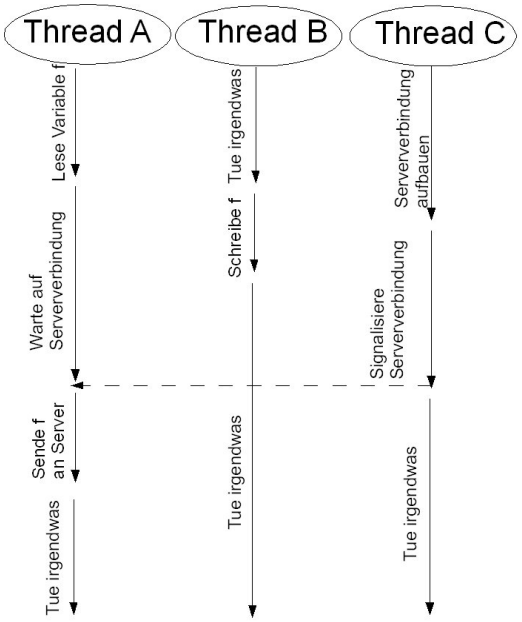



Abbildung 1.6: Inkonsistenz von Daten

Bounding Volumes

Ein Bounding Volume ist ein geometrischer Körper, der einen anderen geometrischen Körper vollständig einschließt.

Es gibt zwei Arten von Bounding Volumes. Zum einen die Bounding Box und die Bounding Sphere.

Sie unterscheiden sich dahingehend, wie sie berechnet werden (Kartesisches Koordinatensystem, Polarkoordinaten).

Die Berechnung der Bounding Volumes im Kartesischen Koordinatensystem ist schneller als die mittels Polarkoordinaten, daher werden hier Bounding Boxes verwendet.

Würden Polarkoordinaten verwendet werden, müssten die Koordinaten der Punkte in *simufact.forming* erst umgerechnet werden.

Es gibt verschiedene Arten von Bounding Boxen.

- Axis Aligned Bounding Boxes (AABB)
  AABB sind parallel zu den Achsen und können durch zwei Punkte beschrieben werden. Diese zwei Punkte sind der minimale und der maximale Punkt.

Sie setzen sich aus den minimalen x,y,z-Koordinaten und den maximalen x,y,z-Koordinaten zusammen.

- Discrete Oriented Polytope (DOP)
  DOP sind nicht unbedingt parallel zu den Achsen. Sie stellen einen Polyeder dar, der den Körper komplett einhüllt. DOP repräsentieren einen Körper genauer als AABB, sind aber in der Berechnung aufwendiger.

Abb.1.7 und Abb.1.8 zeigen zwei Beispiele von Bounding Boxes.

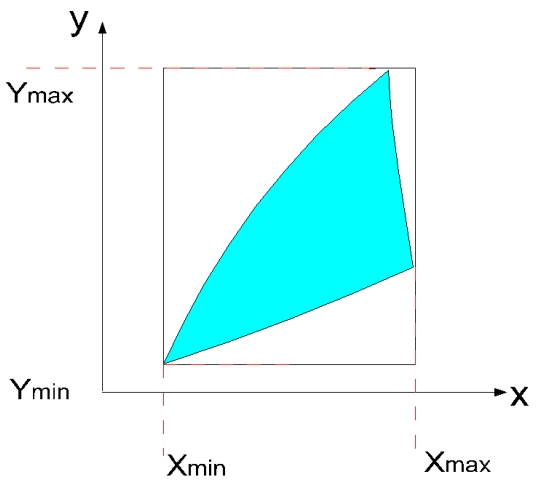



Abbildung 1.7: Bounding Box um ein 2D-Dreieck

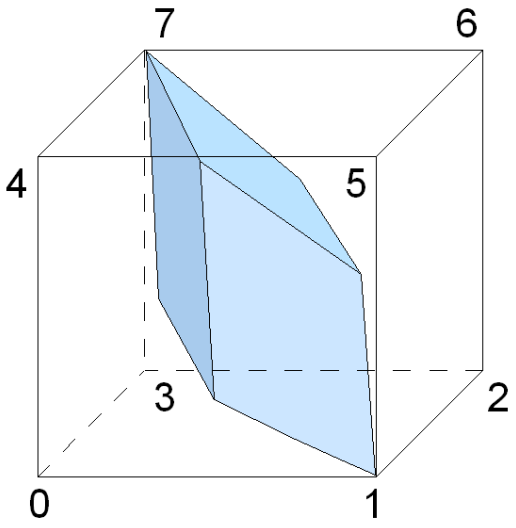



Abbildung 1.8: Bounding Box um einen verformten 3D-Würfel

Binärer Suchbaum

Ein Binärer Suchbaum (Binary Search Tree = *BST*) ist eine Struktur, die eine Datenmenge in Bereiche aufteilt. Es wird hierbei nach einem vergleichendem Prinzip vorgegangen. Es gibt einen Startpunkt (Wurzel) von dem aus verzweigt wird. Ist ein Datenelement größer als die Wurzel (*root*) wird es auf die rechte Seite geschoben, ist es kleiner auf die linke Seite. Es wird zu einem Blatt. Es werden weitere Datenelemente eingefügt. Spätestens nach dem zweiten Element kommt es dazu, dass ein Blatt an ein Blatt angehängt werden muss. In diesem Fall wird das bereits vorhandene Blatt zu einem Knoten. Knoten werden wie die Wurzel (*root*) behandelt. Daraus folgt, dass ein Knoten die Wurzel eines Teilbaums ist.
Der Vergleich zum Einfügen von neuen Datenelementen wird rekursiv durchgeführt. So entsteht eine Struktur die Graphisch als Baum dargestellt werden kann (wobei die Wurzel meist als oberstes Element dargestellt wird).
Ein Binärer Suchbaum wird unbalanciert genannt, wenn eine Seite mehr Elemente

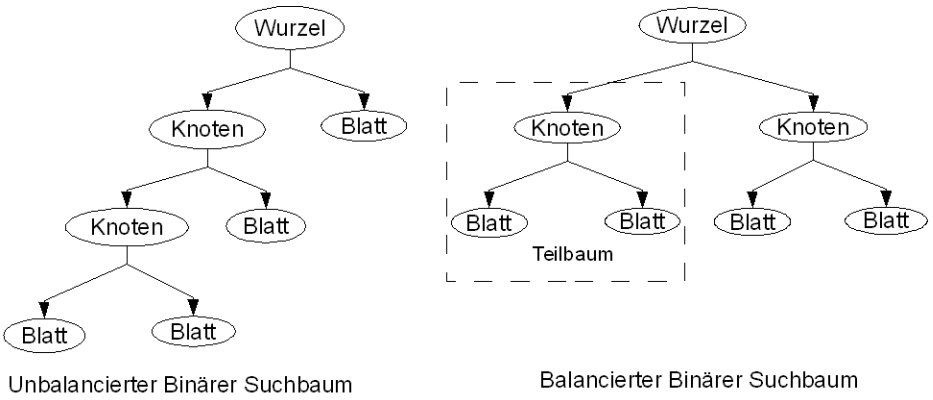



Abbildung 1.9: Balancierter und Unbalancierter Binärer Suchbaum

enthält, als die andere (siehe Abb.1.9). Ein solcher Baum sollte ausbalanciert werden, damit die Suche schneller vonstatten geht.
Sucht man ein Element, braucht man keine Vergleiche mit allen Elementen der Datenmenge durchführen und findet so schneller das gesuchte Element.

Ergebniswerte
: Die Ergebniswerte stellen die interessanten Daten einer Umformung dar. Sie beinhalten z. B. Angaben zur Spannung, zu verschiedenen Drücken, der Fließrichtung und Fließgeschwindigkeit. Während der Modellierung wird festgelegt, welche Ergebnisse berechnet werden sollen.

Altes/Neues Netz
: Das originale Modell, bevor es neu vernetzt wurde, wird hier altes Netz genannt. Daraus folgt die Benennung des neu vernetzten Modells als neues Netz.

Alte/Neue Elemente
: Elemente aus dem alten Netz, werden alte Elemente bezeichnet. Dementsprechend sind Elemente aus dem neuen Netz neue Elemente.

Alte/Neue Knoten
: Bei den Knoten gilt das gleiche Schema, wie bei den Elementen (alte Knoten aus nicht neu vernetztem Modell).

# 2 Analyse

## 2.1 Analyse des vorhandenen Systems

Der vorhandene Algorithmus wurde in *Fortran 77* geschrieben. Verschiedene Methoden werden sequentiell aufgerufen. Die Hauptmethode ist Teil eines komplexen Programmes, welches in *C/C++* geschrieben ist. Der Algorithmus bedient sich, da es in *Fortran 77* keine dynamischen Arrays gibt, C-Schnittstellen, die entsprechende Arrays global verwalten.
Es gibt zwei Netze (bzw. Modelle) die der Funktion übergeben werden. Zu diesen Netzen gehören jeweils zwei Felder zur Darstellung der Geometrie. Die Knotenkoordinaten und die Knotenverbindungen der Finiten Elemente sind jeweils in einem eigenen Feld gespeichert. Zu dem Netz, welches das verformte Modell beschreibt, gehört noch ein drittes Feld. In diesem Feld sind die Ergebniswerte gespeichert. Das zweite, neu vernetzte Modell hat ebenfalls ein Ergebnisfeld. Dieses Feld ist anfangs allerdings nicht mit Werten gefüllt. Die Füllung des Feldes erfolgt durch *remapping*.
Die Fortran-Methode erhält als Parameter einen Pointer auf das erste Element und die Größe des Feldes. Das Feld hat die Dimensionen [X][3], wobei die zweite Dimension die x,y,z-Koordinaten des Knotenpunktes beschreiben und X für den Knoten steht (in *Fortran 77* sind die Dimensionen vertauscht [3][X]).
Die Knotenzugehörigkeit zu einem Element wird in einem Feld mit der Größe [X][8] gespeichert. Um die maximale Anzahl von Knoten eines Hexaederelementes darstellen zu können, beträgt die Größe des Feldes 8. Hierbei kennzeichnet X das Element. Kleinere Elementtypen (z.B. Tetraederelement) werden ebenfalls als Hexaeder gespeichert. In diesem Fall haben einige der Einträge den gleichen Wert (siehe Tab.2.1). Diese Degenerierung der Elemente und die damit einhergehende Speicherverschwendung werden in Kauf genommen, da die Algorithmen mit Hexaederelementen rechnen. Dies ist ein Markenzeichen der Firma *Femutec GmbH*, hiermit setzt man sich von der Konkurrenz ab, die mit ungenaueren Tetraederelementen rechnet.
Ein kleines Beispiel zur Verdeutlichung des Zugriffs auf die Felder der Elemente:
Zugriff auf die x-Koordinate des dritten Knotens des siebten Elementes.
Elemente: connectivity[6][2] liefert den dritten Knoten des siebten Elementes (0 basiert).
x-Koordinate: coordinate[connectivity[6][2][0]] liefert die x-Koordinate des Elementes (0 basiert).

| Hexaeder | Tetraeder | Degenerierter Hexaeder |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 2 | 2 |
| 3 | 3 | 3 |
| 4 | 4 | 3 |
| 5 | | 4 |
| 6 | | 4 |
| 7 | | 4 |
| 8 | | 4 |

Tabelle 2.1: Degenerierte Hexaeder

x,y,z sind hintereinander im Feld abgelegt, entsprechen also 0,1,2 beim Zugriff.
Da Fortran 1 basierte Felder hat [Int03], muss auf den Zugriff besonderes Augenmerk gelegt werden. Ebenfalls ist zu beachten, dass die Felder unterschiedlich angelegt werden. connectivity[6][2] in *C/C++* entspricht connectivity[3][7] in *Fortran*.

## 2.2 Möglichkeiten der Beschleunigung

Um eine Beschleunigung des Algorithmus zu erreichen gibt es verschiedene Ansätze. Man kann den bestehenden Algorithmus parallelisieren, ihm optimieren oder ihn durch einen besseren ersetzen. Eine Parallelisierung macht auf jeden Fall Sinn, wenn man bedenkt, dass immer mehr Computer mit Multi-Core Prozessoren ausgestattet sind.
Eine Optimierung des Algorithmus sollte untersucht werden und unter Umständen in Betracht gezogen werden.
Schlussendlich muss der verbesserte Algorithmus sich gegen andere Algorithmen behaupten, sowohl was Geschwindigkeit, als auch Genauigkeit angeht.
Hierbei ist zu beachten, dass der schnellste Algorithmus nur dann gut ist, wenn er auch die korrekten Ergebnisse liefert.

## 2.3 Beschreibung der Testumgebung

GRUND FÜR TESTUMGEBUNG
Es stellt sich die Frage, warum man eine Testumgebung erstellt. Wenn die Anpassungen nur im originalen Quellcode durchgeführt worden wären, wäre es später schwer geworden, viele automatisierte Testdurchläufe durchzuführen. Es hätte jedes mal ein komplettes FE-Projekt in *simufact.forming* geladen und die Einstellungen für die Parallelisierung angepasst werden

müssen. Dieser Vorgang ist schwer zu automatisieren.
Die Entwicklung und das Testen von weiteren Algorithmen ist ein zusätzlicher Grund für die Erstellung einer Testumgebung. Mit einer Testumgebung kann über die Parameterliste schnell das Verhalten des Programms beeinflusst werden. Die Testumgebung wurde als Konsolenanwendung implementiert. Mittels verschiedener Batch-Dateien können schnell verschiedene Testreihen erstellt und durchgeführt werden.
Der wichtigste Grund für eine Testumgebung ist jedoch die Ausgabe von Auswertungsdaten.

#### COMPILER UND ENTWICKLUNGSUMGEBUNG

Die Testumgebung wurde mit Microsoft Visual Studio 2005 als C++ Projekt aufgebaut. Das Fortran Projekt, welches den Algorithmus enthält wurde eingebunden. Des weiteren mussten andere Projekte des Programms eingebunden werden, damit alle Abhängigkeiten erfüllt werden konnten. Zum Kompilieren des Fortran Quellcodes wurde der Intel Fortran Compiler 9.1 eingesetzt.

#### OPENMP

Damit der Quellcode mit *OpenMP* parallelisiert werden kann, muss dem Compiler über einen Parameter mitgeteilt werden, dass *OpenMP* verwendet wird.
Die OpenMP Direktiven wurden direkt in den Quellcode des Algorithmus eingefügt. Da es möglich sein soll, das Multi-Core Verhalten des Programms aus der Benutzeroberfläche des Hauptprogramms *simufact.forming* heraus zu steuern, wurde die Parameterliste von *sfremap_interface* erweitert.
Das Hauptprogramm ist eine MFC-Applikation, während die Funktion *sfremap_interface* in Fortran geschrieben wurde und als statische Bibliothek eingebunden wird.
Die Erweiterung der Parameterliste besteht aus einem Integerfeld der Größe drei. Damit können die nötigen Umgebungsvariablen für OpenMP gesetzt werden. Es kann die *Anzahl der Threads*, die *Dynamische Erzeugung* und die *Verschachtelte Erzeugung* von Threads eingestellt werden. Die dynamische und die verschachtelte Erstellung werden durch boolsche Werte repräsentiert.
Diese Einstellungen können zwar auch über die Umgebungsvariablen des Betriebssystems gesetzt werden. Hierauf wurde aber bewusst verzichtet. Innerhalb der Fortranmethode kann nicht sichergestellt werden, ob die Umgebungsvariable so eingestellt ist, wie sie benötigt wird. Daher wird das gewünschte Verhalten in der Fortranmethode festgelegt. Die Einstellungen innerhalb des Quellcodes überschreiben die Vorgaben der Umgebungsvariablen, die im Betriebssystem gesetzt wurden.

#### TESTDATEN

Um die nötigen Daten für die Testumgebung zu erhalten, mussten diese aus *simufact.forming* exportiert werden. Hierfür gibt es eine Methode, welche die Modelle in eine Datei speichert. Vor dem Aufruf von *sfremap_interface* wurde daher die Methode zum Exportieren eines Modells aufgerufen. Einmal für die Daten des alten und ein weiteres Mal für die des neuen

Netzes. In der Testumgebung werden diese Dateien eingelesen und die Daten für den Aufruf von *sfremap_interface* aufgearbeitet. Die Modelldaten werden von der importierenden Funktion als Objekt geliefert. Dieses Objekt kann nicht direkt an *sfremap_interface* übergeben werden. Daher wurden die benötigten Variablen aus dem Objekt extrahiert.
Es stellt sich das Problem, dass die Modelle unter Umständen nicht im gleichen Koordinatensystem vorliegen. Die Funktion zum Exportieren speichert nur die Geometrischen Daten und die Ergebniswerte, nicht aber die Transformationsmatrizen. Um die Daten in der Testumgebung im selben Koordinatensystem zu haben, wurde vor dem Exportieren der Netze die Transformationsfunktion ausgeführt. Dadurch konnten später die Transformationsmatrizen so initialisiert werden, dass sie keine Transformation durchführten. In der Testumgebung musste daher keine Berechnung der Beziehungen zwischen den verschiedenen Koordinatensystemen durchgeführt werden.
Ein kleiner Nachteil des Exportierens besteht darin, dass die Daten einem *downcast* unterzogen werden. Die Koordinaten und die Ergebniswerte sind in *simufact.forming* als *double* gespeichert, werden jedoch als *float* exportiert.
Diese Ungenauigkeit wurde bewusst in Kauf genommen. Am Ende des *remapping* wird ein Vergleich der Ergebniswerte der originalen und der parallelisierten Berechnung durchgeführt. Wobei die originalen Ergebniswerte dahingehend verändert wurden, dass sie ebenfalls mit *downcast* Werten berechnet werden.
Somit können Ergebniswerte des originalen Algorithmus und des parallelisierten Algorithmus exakt verglichen werden.

#### Ermittlung der Beschleunigung

Da die Geschwindigkeit der Algorithmen von Interesse ist, wurde vor dem Aufruf des Algorithmus mittels *clock()* (Funktion der stdlib) ein *clock* Wert eingelesen. Nach Beendigung des Algorithmus wurde dann mit einem weiteren Aufruf von *clock()* die Differenz zwischen den beiden Werten gebildet. Diese Differenz gilt hier als Indikator für die Zeit, welche der Algorithmus für seine Arbeit benötigt.
Es wurde versucht mittels eines Profilers genauere Daten über die Beschleunigung des Algorithmus zu gewinnen. Die getesteten Profiler konnten sich allerdings nicht in die parallelisierten Fortranmethoden einklinken. Daher wurde sich darauf beschränkt, die Verbesserungen mittels Logausgabe zu ermitteln und zu vergleichen.

#### Log Datei

Um die ermittelten Daten auswerten zu können, wurde eine Log Datei geschrieben. In dieser Datei werden verschiedene Angaben gespeichert.

- Namen der Dateien der eingelesenen Netze (alt und neu)
- Name der erstellten Ergebnisdatei
- Anzahl der Elemente des alten und neuen Netzes

- Wahl des Algorithmus
- Bezeichnung der Ergebnisvariablen
- Anzahl der bearbeitenden Threads
- Angabe, ob dynamische Erstellung verwendet wurde
- Angabe, ob verschachtelte Erstellung verwendet wurde
- Zeit, die der Algorithmus zur Bearbeitung braucht
- Anzahl der Unterschiede zwischen Ergebniswerten (original vs. parallelisiert)

Die Angabe über die dynamische und die verschachtelte Erstellung von Threads ist nur für die Parallelisierung mittels *OpenMP* interessant.
Die Anzahl der Unterschiede zwischen dem originalen und dem parallelisierten Algorithmus muss bei der Parallelisierung mit *OpenMP* Null sein. Beim entwickelten Algorithmus wird dieser Wert von Null abweichen. Der Grad dieser Abweichung kann ein Hinweis auf Genauigkeit des Ergebnisses des entwickelten Algorithmus sein.

PARAMETERLISTE
Über die Parameterliste der Testumgebung kann das Verhalten des Programms beeinflusst werden. In Klammern ist die Syntax des Aufrufs kurz angegeben.

- Namen der Dateien des alten und des neuen Netzes (-[ro|rn] *Filename*)
- Name der originalen Ergebnisdatei für den Vergleich (-[org] *Filename*)
- Anzahl der Threads (-[ot] *Number of Threads*)
- Angabe, ob dynamische Erstellung verwendet wird (-[od] *Dynamic Creation of Threads*)
- Angabe, ob verschachtelte Erstellung verwendet wird (-[on] *Nested Creation of Threads*)
- Angabe über die aktuelle Nummer des Durchlaufs (-[run] *Number of Run*)
- Wahl des Algorithmus (-[cc|cf] *Selection of Algorithm*)
- Toleranz beim Vergleich der Ergebniswerte (-[to] *Tolerance*)

Der Parameter für die dynamische Erstellung der Threads ist in erster Linie für die Parallelisierung mit *OpenMP* relevant. Für die Testreihen mit den entwickelten Algorithmus wurde dieser Parameter für andere Angaben missbraucht.
Die Nummer des Durchlaufes wurde eingeführt, um eine beliebige Anzahl an Durchläufen generieren zu können. Jeder dieser Durchläufe erzeugt eine eigene Log Datei. Mit Hilfe dieser Log Dateien kann ein Mittelwert über die Zeiten der Durchläufe erzeugt werden.

Um den Grad der Toleranz bei der Abweichung zwischen den Ergebniswerten des originalen und des entwickelten Algorithmus einstellen zu können, wurde ein weiterer Parameter hinzugefügt.

# 3 Remapping von Finiten Elementen

## 3.1 Möglichkeiten der Parallelisierung

### 3.1.1 Parallelisierung mit Threadbibliotheken

Die einfachste Möglichkeit, Arbeit auf mehrere Threads zu verteilen besteht darin, diese Threads zu erzeugen und ihnen ihre Aufgaben zuzuweisen. Hierbei gibt es verschiedene Ansätze, die sich nur gering unterscheiden. Allen Threads ist gemein, dass sie gestartet, gestoppt, synchronisiert und priorisiert werden können.
Im folgenden werden exemplarisch zwei Threadbibliotheken vorgestellt. Beide kommen für die Verwendung zur Parallelisierung in Frage.
*Pthreads* sind Teil des *POSIX* Standard [IEE96]. Sie erhalten per Parameterliste einen Funktionspointer und führen diese Funktion dann aus. In dieser Funktion können sie natürlich noch andere Funktionen aufrufen.
Die zweite Threadbibliothek wird von der Firma *Trolltech* zur Verfügung gestellt [Tro05]. Die *QThreads* des Qt-Frameworks sind abstrakte Klassen. Von *QThread* abgeleitete Klassen müssen die Funktion *run* implementieren. In dieser Funktion werden dann die Aufgaben des Threads festgelegt.
Im Unterschied zu *QThreads* sind *Pthreads* nicht in Klassen gekapselt. Sie haben den Vorteil, dass sie universelle Konstrukte sind, die lediglich einen Funktionspointer benötigen, um ihre Arbeit erledigen zu können. *QThreads* hingegen sind an eine Klasse gebunden und bieten daher die Vorteile der Objektorientierung, wie z. B. Kapselung von Daten, Vererbung. Sie haben jedoch auch die Nachteile, wie z. B. Speicherverschwendung, Zeitverlust beim Erstellen großer Objekte.
Eine Aufteilung der Arbeit kann auf folgende Art und Weise geschehen:

- Jeder Thread bekommt eine eigene Aufgabe
- Eine Aufgabe wird auf verschiedene Threads verteilt

Ein Beispiel für eigenständige Aufgaben von Threads ist ein Thread, der periodisch kontrolliert, ob eine Ressource noch vorhanden ist. Dies könnte eine Internetverbindung, genügend Festplattenplatz oder das Vorhandensein einer bestimmten Datei sein.
Als Beispiel für die Aufteilung einer Arbeit auf verschiedene Threads wird das Kodieren einer

Datei herangezogen. Jeder Thread bearbeitet einen Teilbereich. Sobald alle geendet haben, ist die Aufgabe erledigt. Danach müssen unter Umständen die Ergebnisse der Berechnungen zusammengeführt werden.
Ein Nachteil der Threadbibliotheken besteht darin, dass der Programmierer sich selber um die Synchronisation der Threads kümmern muss. Übersieht er etwas, können die Ergebnisse fehlerhaft sein. Im schlimmsten Fall werden diese Fehler nicht bemerkt.

### 3.1.2 OpenMP

*OpenMP* (Open Multi Processing) ist ein offener Standard zur Parallelisierung von sequentiellem Programmcode der Sprachen Fortran und C/C++, ohne den Quellcode größeren Veränderungen zu unterziehen [Boa07]. 1997 wurde OpenMP als industrieller Standard von einer Mehrheit aus Industrie und Entwicklern akzeptiert. Version 2.5 wurde bis Oktober 2007 als finale Version ausgegeben. Am 29 Oktober 2007 ist aber ein Entwurf für Version 3.0 veröffentlicht worden, der öffentlich kommentiert werden kann. Das Architecture Review Board (ARB) zeigt sich für die Weiterentwicklung und Betreuung des Standards verantwortlich.

FUNKTIONSWEISE
Die Verteilung der Arbeit funktioniert nach dem Fork/Join-Prinzip. Es gibt einen *MASTER*-Thread, der eine Anzahl von Child-Threads erschafft, wenn er einen zu parallelisierenden Abschnitt des Programms erreicht. Die Arbeit innerhalb des Abschnittes (*REGION* genannt) wird unter den erstellten Threads und dem *MASTER*-Thread aufgeteilt. Am Ende der parallelen Region gibt es eine implizite Barriere (*BARRIER*) und die Threads warten mit ihrer Ausführung, bis alle ihre Arbeit erledigt haben. Diese implizite Barriere kann jedoch außer Kraft gesetzt werden. Dies sollte nur dann (mit Vorsicht) getan werden, wenn die folgenden Anweisungen unabhängig von der parallelisierten Region sind.
Beispiel für die Erstellung von parallelen Regionen:
*PARALLEL*
*...*
*DO*
*arbeite auf dem Feld x*
*END DO*
*baue eine Verbindung zu einem Server auf*
*...*
*END PARALLEL*

Der Serveraufbau in diesem Beispiel ist vollständig unabhängig von der Arbeit auf dem Feld x. Daher kann die implizite Barriere am Ende der parallelisierten Region (*END DO*) außer Kraft gesetzt werden. Der erste Thread, der das Ende der Region passiert, führt die nächsten Anweisungen aus. Hierbei ist jedoch darauf zu achten, dass auch jeder weitere

Thread, der mit seiner Arbeit in der parallelen Region fertig ist, die Anweisungen danach ausführen würde. Mit geeigneten Mitteln kann dem entgegen gewirkt werden (siehe *SECTION*).
Es ist implementationsabhängig, ob die Child-Threads am Ende des parallelen Abschnittes vernichtet oder nur suspendiert werden. Um die Effizienz zu erhöhen, werden die Threads in den meisten Implementationen suspendiert.
Die Parallelisierung findet über *Direktiven* statt. Diese teilen dem Precompiler mit, welches Programmsegment auf welche Weise parallelisiert werden soll. Weitere *Direktiven* fungieren als Parameterangaben zur Bestimmung der Arbeitsaufteilung durch die einzelnen Threads (genauere Angaben hierzu [Boa07]).
Hauptsächlich werden Schleifendurchläufe auf verschiedene Threads aufgeteilt. Eine Arbeitsverteilung von sequentiellem, voneinander unabhängigem Programmcode ist aber auch möglich (siehe *SECTION*).
Die Priorisierung der einzelnen Threads ist nicht möglich. Darin liegt ein Nachteil von OpenMP. Alle Child-Threads laufen mit derselben Priorität.

UMGEBUNGSVARIABLEN

Unter einer Umgebungsvariable versteht man im Allgemeinen eine globale Variable, die für die gesamte Umgebung gültig ist. Es ist mit *OpenMP* möglich solche Umgebunsgvariablen zu setzen. Dies hat den Vorteil, dass ein einzelnes Programm sich nicht darum kümmern muss, diese Variablen für sich zu setzen. Es spart Funktionsaufrufe und ist damit schneller in der Ausführung. Andererseits stellt es einen Nachteil dar. Soll ein anderes paralleles Verhalten verwendet werden, ist es individuell einzustellen. Damit hat man wieder die Funktionsaufrufe, die vermieden werden sollten.
Über die Umgebungsvariablen lassen sich verschiedene Verhalten festlegen:

- Anzahl der maximalen Threads
  Es kann eingestellt werden, wie viele Threads maximal erschafft werden sollen. Man kann nicht die exakte Anzahl Threads angeben. Sie hängt vom Laufzeitsystem ab. Ist es stark ausgelastet, werden weniger Threads erstellt.

- Dynamische Erstellung von Threads
  Hier kann ein boolscher Wert angegeben werden. Wenn die dynamische Erstellung von Threads aktiviert ist, kann das Laufzeitsystem die Erschaffung von neuen Threads erlauben. Es kann aber maximal die in *Anzahl der Threads* festgelegte Threadanzahl erzeugt werden. Im ungünstigsten Fall (*Anzahl der maximalen Threads* = 1) wird kein Thread erzeugt und nur der *MASTER*-Thread erledigt die Arbeit.

- Verschachtelte Erstellung von Threads
  Die Umgebungsvariable wird durch einen boolschen Wert repräsentiert. Ist die verschachtelte Erstellung von Threads ausgeschaltet, können in Funktionen, die aus parallelen Regionen heraus aufgerufen werden, keine weiteren Threads erzeugt werden.

Ist sie allerdings aktiv, wird der erste Thread, der in eine Funktion eintritt und dort auf eine neue parallele Region trifft zu einem *MASTER*-Thread und erzeugt Child-Threads. Allerdings auch hier nur bis zu der, über die Umgebungsvariable angegebene maximale Anzahl.

VERWENDETE DIREKTIVEN
Jede Region, die parallelisiert werden soll wird von der *Direktive PARALLEL* eingeschlossen. Beispiel für die Verwendung der Direktive *PARALLEL*:
*PARALLEL*
*...*
*END PARALLEL*

Innerhalb einer Region gibt es verschiedene Ansätze diese Region zu parallelisieren. Hierbei muss auch auf die Sichtbarkeit von Variablen geachtet werden, da es sonst zu Inkonsistenzen kommen kann.

- Parallelisierung von Regionen
  - DO
    Parallelisierung von Schleifendurchläufen werden mit den Direktiven *DO ... END DO* durchgeführt. Es sollte die äußerste Schleife parallelisiert werden. Der Ablauf der inneren Schleifen bleibt sequentiell. Die Iterationsvariable ist implizit *PRIVATE*, es sei denn, es handelt sich um eine eingeschlossene Schleife. In diesem Fall muss die Iterationsvariable explizit so deklariert werden, wie man es benötigt (in den meisten Fällen *PRIVATE*).
  - WORKSHARE
    *WORKSHARE* wurde eingeführt, um die Bearbeitung von Feldern parallel ausführen zu können. Es arbeitet mit dem Konstrukt *FOR...EACH*, welches in *Fortan 77* noch nicht bekannt ist.
  - SECTION
    Die Aufteilung einer parallelen Region in verschiedene Sektionen kann mit dieser Direktive bewerkstelligt werden. Diese Sektionen müssen unabhängig voneinander sein. Ein Beispiel ist das Lesen eines Feldes, während eine Statusmeldung über den aktuellen Vorgang (lesen des Feldes) ausgegeben wird.
- Sichtbarkeit von Variablen
  - PRIVATE
    Die Variablen sind für jeden Thread privat. Kein anderer Thread kann diese Variablen lesen oder beschreiben. Daraus folgt, dass für jede *PRIVATE* Variable eigener Speicher reserviert werden muss.
    Ist eine Variable außerhalb der parallelen Region deklariert worden, muss sie

explizit *PRIVATE* deklariert werden. In diesem Fall hat sie einen undefinierten Wert, wenn der Thread in die parallele Region eintritt.
Innerhalb von parallelen Regionen erzeugte Variablen sind standardmäßig *PRIVATE*.

- FIRSTPRIVATE
  Möchte man das Problem umgehen, dass außerhalb der parallelen Region deklarierte Variablen einen undefinierten Wert in der parallelen Region haben, deklariert man die Variable als *FIRSTPRIVATE*. Sie ist dann für jeden Thread privat, hat aber den Wert der Variablen vor der parallelen Region als initialen Wert.
- SHARED
  Variablen dieses Typs sind für alle Threads verfügbar. Ist eine Variable vor der parallelen Region deklariert worden ist sie standardmäßig *SHARED*.

SENTINELS

*OpenMP* hat in Fortran nur einen Kommentarcharakter. Wenn die Option zum parallelisier-

Beispiel für den Gebrauch von Sentinels:

| | |
|---|---|
| a=10 | *Kein Sentinel* |
| | *Normaler Quellcode* |
| !$ OMP_SET_DYNAMIC(TRUE) | *Sentinel '!$'verwendet* |
| !$ if(OMP_GET_DYNAMIC.eq.1) | *Sentinel bei Fallunterscheidungen* |

ten Kompilieren mittels *OpenMP* nicht aktiviert ist, werden die eingefügten Zeilen wie Kommentare behandelt. Damit der Compiler weiß, welche Zeilen Kommentar sind und welche auszuwertenden Quellcode enthalten, gibt es *Sentinels*. Sie werden der Zeile vorangestellt. Wie man in dem Beispiel sehen kann, ist es durchaus möglich mit *OpenMP* und Veränderungen am Quellcode weitere Verfeinerungen zu gestalten. Wie diese Feinheiten genau aussehen, hängt von dem gewünschten Effekt ab.
Mittels *SENTINELS* kann eine Fallunterscheidung getroffen werden und abhängig von der Entscheidung verschiedene parallele Verhalten erzeugt werden.

### 3.1.3 Weitere Möglichkeiten

MESSAGE PASSING INTERFACE

Das Message Passing Interface (*MPI*) wird von dem *Message Passing Interface Forum* verwaltet und liegt momentan in der Version 2.0 vor [For07]. Es verteilt Aufgaben auf verschiedene Rechner in einem Netzwerk. Die Daten für diese Aufteilung werden ebenfalls verteilt.
Der Quellcode muss mit MPI-Anweisungen angepasst werden. Eine explizite Verteilung der

Aufgaben ist wie bei den Threadbibliotheken nötig.
MPI-2.0 ist sowohl für *C/C++*, wie auch *Fortran 90* verfügbar.

HIGH PERFORMANCE FORTRAN
High Performance Fortran (*HPF*) ist eine Erweiterung der Sprache *Fortran 90*, welche die parallele Bearbeitung von Feldern ermöglicht. Sie wurde vom High Performance Fortran Forum *HPFF* 1993 veröffentlicht [For06].

## 3.2 Bestehender Algorithmus

### 3.2.1 Wahl der Parallelisierung

Der bestehende Algorithmus wurde in der Programmiersprache *Fortran 77* programmiert und eine Verwendung von Spezifikationen späterer *Fortran*-Versionen war nicht gestattet. Daher wurde *HPF* nicht verwendet, denn es setzt auf *Fortran 90* auf. Obwohl *Fortran 90* abwärts kompatibel zu *Fortran 77* ist, kann *HPF* nicht verwendet werden, da es auf dem Konstrukt *FOR...EACH* aufbaut. Mit diesem Konstrukt kann jedes Element eines Feldes parallel bearbeitet werden. Da es *FOR...EACH* allerdings in *Fortran 77* noch nicht gab, wird *HPF* hier nicht verwendet.

Nach Vorgabe der Aufgabenstellung soll die Parallelisierung auf Shared-Memory-Architektur stattfinden. *MPI* verfolgt hingegen den Ansatz auf einer Distributed-Memory-Architektur (verteilter Speicher Architektur) zu funktionieren. Es bleibt von der Firma *Femutec GmbH* zu entscheiden, ob in Zukunft das *Remapping* auf einer Distributed-Memory-Architektur lauffähig sein soll.
Die Aufteilung der Arbeit auf verschiedene Threads muss bei *MPI* im Programm vorgenommen werden. Hierzu muss das Programm durch Erweiterungen verändert werden.

Ein Einführen von Threads im Quellcode (z.B. *Pthreads*) würde eine größere Umstrukturierung des Quellcodes mit sich führen. Da *Fortran 77* keine Dynamische Speicherverwaltung kennt, ist das Programm auf die Interaktion mit den in C/C++ geschriebenen Funktionen angewiesen. *Fortran 77* kennt keine Threadmechanismen, daher müsste auch eine Veränderung in dem umschließenden Quellcode vorgenommen werden.

Um die Veränderungen an dem Algorithmus so gering wie nur möglich zu halten und um schnell zu einem Ergebnis zu gelangen, wurde *OpenMP* gewählt. Mit *OpenMP* wird an dem Quellcode nichts verändert und es ist leicht zu implementieren.
Es werden lediglich Kommentare hinzugefügt, die vom Precompiler ausgewertet werden und den generierten Maschinencode parallelisieren. Über die Aufteilung der Arbeit muss also keine Überlegung angestellt werden.

Da im Quellcode sehr häufig Schleifendurchläufe auftreten und diese viele Iterationen beinhalten, bietet sich *OpenMP* an. Es wurde für die Parallelisierung von Schleifendurchläufen entwickelt. Die Parallelisierung von voneinander unabhängigen Arbeitsbereichen lässt sich mit *OpenMP* ebenfalls durchführen. So kann ein Thread dafür abgestellt werden eine Ausgabe zu tätigen, während ein anderer mit den Berechnungen fortfährt.

### 3.2.2 Analyse der Funktionsweise des Algorithmus

In Abb.3.1 ist der Ablauf der Funktion des vorhandenen Algorithmus in einem Flussdiagramm aufgeführt.

DEGENERATION
Der Algorithmus sorgt als erstes dafür, dass die Elemente des alten und des neuen Netzes als degenerierte Hexaederelemente gespeichert werden, da der Algorithmus nur mit Hexaederelementen rechnet. Ein degeneriertes Hexaederelement hat Knoten, welche dieselben Koordinaten teilen (siehe Tab.2.1 Seite 21). So hat ein Tetraederelement als degeneriertes Hexaederelement fünf Knoten, mit denselben Koordinaten (ein Pentaederelement hat demnach vier gleiche Knoten). Hier wird bewusst Speicher verschwendet, um eine einheitliche Methodik verwenden zu können.

TRANSFORMATION
Das alte Netz wird transformiert. Somit befindet es sich für die späteren Berechnungen im gleichen Koordinatensystem, wie das neue Netz. Hierbei ist darauf zu achten, dass auch die Ergebniswerte angepasst werden müssen. Vektorielle Ergebnisgrößen haben eine Richtung und müssen daher ebenfalls transformiert werden.

BOUNDING BOX
Für jedes alte Element wird eine Bounding Box erstellt.

$$\min_{i=1}^{n}(x_i), \text{ mit n = Anzahl der Knoten} \tag{3.1}$$

$$\max_{i=1}^{n}(x_i), \text{ mit n = Anzahl der Knoten} \tag{3.2}$$

Gleiches gilt für die y- und z-Koordinate. Hiermit erhält man also sechs Werte (zwei Koordinatenpunkte) mit denen sich ein Hexaeder aufspannen lässt. Allgemein kann gesagt werden, dass aus den Koordinaten beliebig gestalteter Polyeder, mit dieser Methode eine Box erstellt werden kann, die den Polyeder vollständig enthält.

ZWISCHENNETZ ERSTELLEN
Die Bounding Box um das neue Netz wird in 500.000 kleine, gleichgroße Würfel zerteilt. Es wird für jeden dieser Würfel gespeichert, mit welcher Bounding Box des alten Netzes er eine

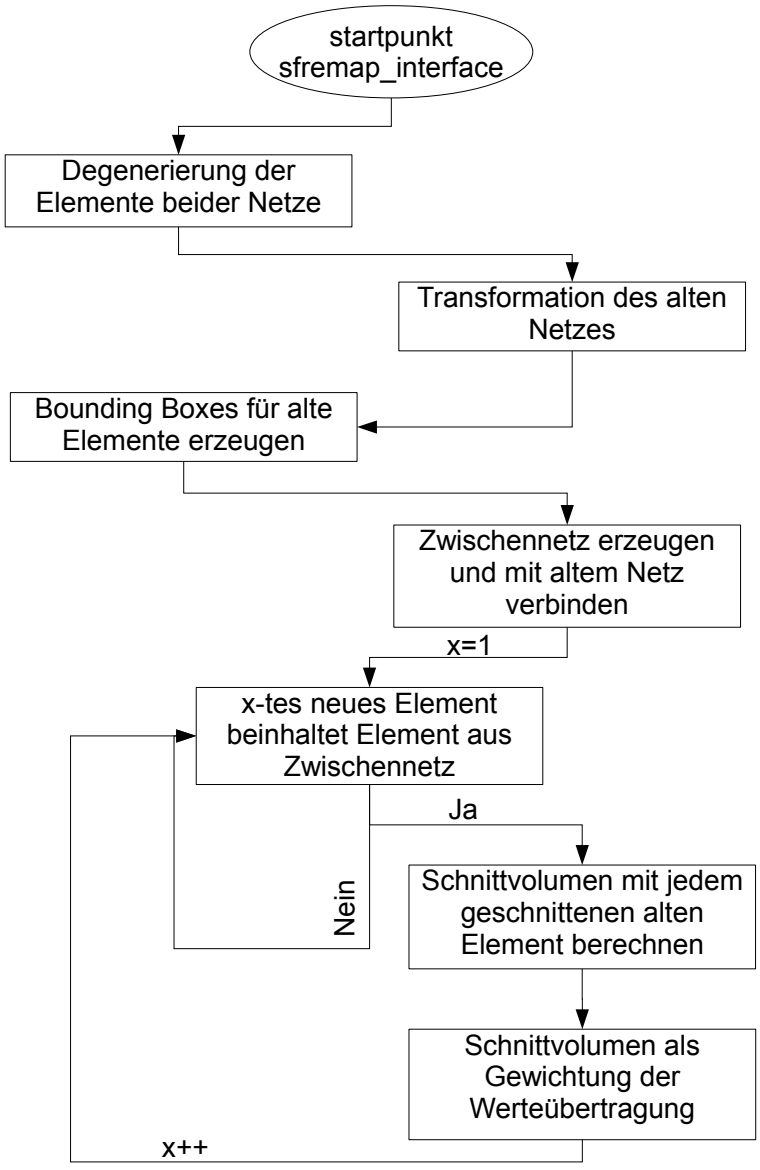



Abbildung 3.1: Ablaufdiagramm des bestehenden Algorithmus

Überschneidung hat. Die Aufteilung in 500.000 kleine Würfel ist fest einprogrammiert und stellt eine *Magic Number* dar. Die Zahl begründet sich auf die Annahme, dass die Anzahl der neuen Elemente im Netz kleiner ist, als die *Magic Number* und damit alle kleinen Würfel den neuen Elementen zugeordnet werden können. Mit steigender Anzahl an neuen Elementen ist dies aber nicht unbedingt gegeben. Die *Magic Number*, müsste vergrößert werden, wobei eine dynamische Anpassung zur Laufzeit, anhand der Anzahl der neuen Elemente die effizienteste Lösung darstellt, da so eine ständige Anpassung der *Magic Number* vermieden werden würde.

HAUPTSCHLEIFE

Sequentiell werden die neuen Elemente bearbeitet.
Zuerst wird in einem Vorabverfahren kontrolliert, welche der erzeugten 500.000 Würfel das neue Element vollständig beinhaltet. Verweise auf diese alten Elemente werden gespeichert. Die Oberfläche jedes neuen Elementes wird in zwölf Dreiecke geteilt (siehe Abb.3.2). Diese

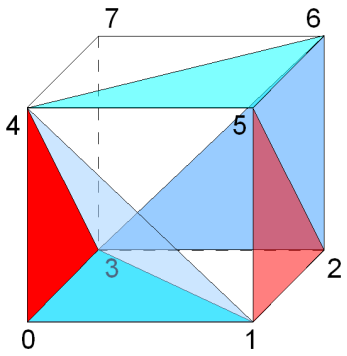



Abbildung 3.2: Aufteilung einer Hexaederoberfläche in Dreiecke

Aufteilung ist für alle Elemente gleich. Die Elemente des alten Netzes, welche das neue Element schneiden, werden ebenfalls in Dreiecke aufgeteilt. Anhand dieser Dreiecke wird überprüft, ob sich die Elemente überschneiden. Mit Hilfe dieser Überschneidungen werden die Schnittvolumina errechnet. Die Übertragung der Werte vom alten auf das neue Netz erfolgt nach einem Volumenprinzip. Je größer das Volumen ist, welches durch die Überschneidung von altem und neuen Element entsteht, desto größer ist der Einfluss des alten Elementes auf den Ergebniswert des neuen Elementes. Daraus ergibt sich, dass alte Elemente, die ein größere Schnittvolumen mit dem neuen Element haben, als andere alte Elemente, einen entsprechend größeren Einfluss auf den Ergebniswert haben. Für vektorielle Größen, wie z.B. Spannungen wird der größte für das alte Element vorhandene vektorielle Wert für das neue Element übernommen.
Die genaue Berechnungen der Schnittflächen und Volumen wird hier nicht behandelt. Sie unterliegen dem Firmengeheimnis.

### 3.2.3 Veränderungen am originalen Quellcode

Damit der Algorithmus mit *OpenMP* parallelisiert werden kann, müssen einige Veränderungen an ihm vorgenommen werden. Diese Anpassungen sollen so gering wie möglich ausfallen und den eigentlichen Algorithmus nicht beeinflussen.

ERMITTELN DER SYSTEMDATEN
Wie viele Threads sollen erschaffen werden? Die Beantwortung hängt von dem System ab, auf dem gearbeitet wird. Über die Oberfläche von *simufact.forming* ist die Anzahl der Threads einstellbar, mit welcher der Solver rechnen soll. Entweder wird dieser Wert übernommen oder dynamisch angepasst. Im ersten Fall wird die Parameterliste erweitert. Bei der zweiten Möglichkeit wird die dynamische Einstellung für die Threadanzahl im Fortran-Quellcode eingebaut. Um flexibel zu bleiben, wurden beide Wege implementiert.

PARAMETERLISTE
Die Parameterliste der Funktion *sfremap_interface* wurde dahingehend angepasst, dass Parameter für die Einstellung der Umgebungsvariablen für *OpenMP* übergeben werden können.
Es wird ein Pointer auf ein Integerfeld der Größe drei übergeben.
Das Feld besteht aus:

- maximale Anzahl der Threads
- dynamische Erstellung von Threads
- verschachtelte Erstellung von Threads

DYNAMISCHE ARRAYS
Da es bei der Parallelisierung zu Konflikten kam, mussten zwei dynamische Arrays durch statische Felder ersetzt werden.
Es wird daher Speicher für zwei Felder reserviert. Als Zwischenspeicher enthalten sie die Schnittmenge zwischen Elementen des alten und des Zwischennetzes. Die Größe der Felder ist durch die Anzahl der alten Elemente gegeben, da nicht vorhergesagt werden kann, wie viele Elemente sich schneiden. Daraus ergibt sich leider auch eine Speicherverschwendung. Je weniger Elemente sich schneiden, desto mehr Speicher wird verschwendet, da er nicht verwendet wird.

*OpenMP*
Verschiedene Stellen des Quellcodes wurden in Direktiven eingefasst. Diese Direktiven haben Kommentarcharakter und werden nur ausgewertet, wenn die *OpenMP*-Unterstützung für den Compiler aktiviert wurde. Ansonsten werden die Direktiven wie Kommentare behandelt und dementsprechend vom Precompiler ignoriert.
Damit eine Parallelisierung stattfinden kann, müssen für *OpenMP* Umgebungsvariablen gesetzt werden.

- !$ call OMP_SET_NUM_THREADS(nOpenMP(1))
- !$ call OMP_SET_DYNAMIC(nOpenMP(2).gt.0)
- !$ call OMP_SET_NESTED(nOpenMP(3).gt.0)

*nOpenMP* ist das Integerfeld, welches die Daten für das Setzen der Umgebungsvariablen enthält.
*OMP_SET_DYNAMIC* und *OMP_SET_NESTED* erwarten Boolsche Werte. Prinzipiell ist ein Vergleich überflüssig, da alle Werte größer eins als *true* angenommen werden.
Die meisten Veränderungen wurden an Schleifen durchgeführt. Da es relativ wenige Programmabschnitte gibt, die keine Schleifen und unabhängig und damit parallelisierbar sind, gibt es wenige *WORKSHARE*- oder *SECTION*-Regionen.
*SECTION* wird meist dort eingesetzt, wo Ausgaben in Log Dateien oder auf eine Ausgabekonsole erfolgen. Da diese Schreibzugriffe unabhängig vom rechnenden Algorithmus sind, können sie in einen Thread ausgelagert werden, während der *MASTER* mit der Arbeit fortfährt.
Es war lediglich darauf zu achten, dass sich die schreibenden Threads nicht gegenseitig beeinflussen.
Die Liste mit den *FIRSTPRIVATE*-, *PRIVATE*- und *SHARED*-Variablen ist sehr lang und umfasst jede Variable, die in der Hauptschleife der Funktion verwendet wird.

## 3.3 Nearest Neighbour

### 3.3.1 Wahl der Parallelisierung

Da jetzt von Grund auf ein neuer Algorithmus implementiert werden soll, muss kaum Rücksicht auf bestehende Strukturen genommen werden. Der Algorithmus wird vollständig in *C/C++* implementiert, daher kann die dynamische Speicherreservierung mit eingebaut werden. Auf *OpenMP* braucht nicht mehr zurück gegriffen werden. Die Parallelisierung des Algorithmus sollte mit einer Threadbibliothek durchgeführt werden, die sowohl unter Windows, als auch unter Linux zur Verfügung steht.
Zur Auswahl standen Pthreads und QThreads. Beide sind unter verschiedenen Betriebssystemen verfügbar und bieten die erforderlichen Funktionalitäten.

PTHREADS
*Pthread*s ist ein POSIX (Portable Operating System Interface) Standard der von der IEEE (Institute of Electrical and Electronics Engineers) zertifiziert wird.

Sie bieten jeden gewünschten Komfort einer Threadbibliothek. Es werden verschiedene Synchronisationsmechanismen angeboten und auch eine Priorisierung der Threads ist möglich. *Pthread*s sind frei verfügbar und es müssen keine Lizenzen entrichtet werden.

QTHREADS
*QThread*s sind Threads, die von dem Qt-Framework der Firma Trolltech bereit gestellt werden.
Es werden nicht so viele Synchronisationsmechanismen wie bei Pthreads geboten, so fehlt die Möglichkeit eine *Barrier* zu erstellen. Die Funktion kann nachgebaut werden, erfordert aber einen Programmieraufwand. QThreads lassen sich priorisieren.
Es gibt das Framework sowohl als Open Source, wie auch als lizensierte Version.

ENTSCHEIDUNG
Trotz der Vorteile der *Pthreads* (siehe Tab.3.1), wurde sich für die *QThreads* entschieden.
Funktionspointer werden in der Firma *Femutec GmbH* nicht verwendet und es besteht bereits eine Abhängigkeit zum Qt-Framework.
Der eingeschränkte Funktionsumfang der QThreads spielt in der Entscheidungsfindung keine Rolle, da alle benötigten Funktionen vorhanden sind.

| | Vorteile | Nachteile |
|---|---|---|
| Pthread | Standard<br>Alle erforderlichen Funktionen | Funktionspointer |
| QThread | Klassenstruktur | Abhängigkeit vom Framework<br>Eingeschränkter Funktionsumfang |

Tabelle 3.1: Vor-/Nachteile der Threadbibliotheken Pthread/QThread

### 3.3.2 Funktionsweise des Algorithmus Nearest Neighbour

Es war ein neuer Algorithmus zu entwickeln, der schneller als der alte Algorithmus ist. Dabei sollte er aber auch nicht allzu sehr an Genauigkeit verlieren. Der einfachste Weg ein Programm schneller ablaufen zu lassen, ist es, die Anzahl der Berechnungen zu reduzieren. Daher wurde auf die aufwendige Berechnung der Schnittvolumina verzichtet und nach einem anderen Ansatz gesucht.

Die Grundlegende Idee des neuen Algorithmus besteht darin, die Suche nach Elementen für das *Remapping* zu beschleunigen.
Es soll eine Verringerung der Vergleiche zwischen den Elementen erreicht werden.
In Abb.3.3 wird der Ablauf des implementierten Algorithmus in einem Flussdiagramm dargestellt.

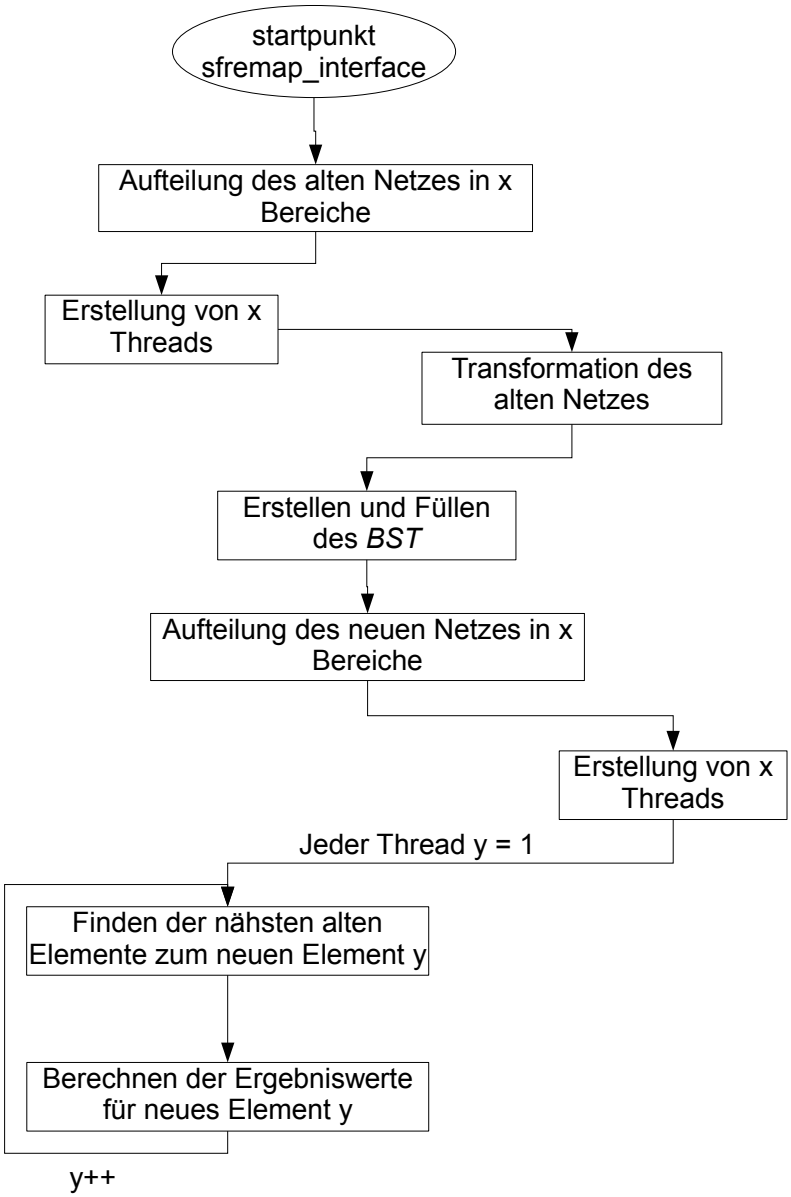



Abbildung 3.3: Ablaufdiagramm implementierter Algorithmus

ERMITTELN DER SYSTEMDATEN

Wieviele Threads sollen erschaffen werden? Die Beantwortung hängt von dem System ab, auf dem gearbeitet wird. Über die Oberfläche von *simufact.forming* ist die Anzahl der Threads einstellbar, mit welcher der Solver rechnen soll. Dieser Wert kann übernommen werden. Der

neue Algorithmus wurde so entwickelt, dass er kompatibel zum alten ist. Daher gibt es eine Funktion mit dem Namen *sfremap_interface*. Mit dieser Methode wird das *Remapping* gestartet. Die Einstellung für die Erstellung der Threads können also per Parameter mit dem eingeführten *OpenMP*-Feld übergeben werden. Wird kein Wert übergeben, wird überprüft, wie viele Kerne zur Verfügung stehen und die Anzahl der Threads auf die ermittelte Kernanzahl gesetzt.

AUFTEILUNG DES WORKLOADS

Die Suche nach alten Elementen im *BST* (Binary Search Tree) und die anschließende Werteübertragung besteht aus immer der gleichen Anzahl an Schritten. Daher ist es leicht die Arbeit auf eine variable Zahl von Threads aufzuteilen.
Jedes neue Element muss bearbeitet werden. Die neuen Elemente sind in einem Feld gespeichert, welches in x Teile zerlegt wird. Hierbei ist x die Anzahl der Threads für die Berechnung der Werteübertragung. Jeder Thread bekommt die nahezu gleiche Anzahl an neuen Elementen, die er bearbeiten soll.
Die Zuordnung der neuen Elemente zu den Threads erfolgt über die Division der Anzahl der neuen Elemente durch die Threadanzahl.

$$NumLoad = \frac{NumElements}{NumThreads} \tag{3.3}$$

$$Rest = NumElements\%NumThreads \tag{3.4}$$

Bei der Division entsteht höchstwahrscheinlich ein Rest. Dieser wird mit der *modulo*-Berechnung ermittelt.
In einem Feld werden die Bereiche der neuen Elemente gespeichert, die den Threads zugeordnet werden sollen.
Beispiel:

- Anzahl der Threads = 4 (NumThreads)
- Anzahl der neuen Elemente = 6014 (NumElem)
- NumLoad = Zuordnungsbereich
- Rest = Rest der Division
- BereichsFeld[4]

Der Sourcecode sieht dann wie folgt aus.

```
...
int NumLoad = NumElem / NumThreads; //NumLoad bekommt den Wert 1503
int Rest = NumElem % NumThreads; //Rest bekommt den Wert 2
BereichsFeld[0] = NumLoad;
```

*for( int i=1; i<BereichsFeld.size(); i++){*
*BereichsFeld[i] = NumLoad + BereichsFeld[i-1];*
*if(Rest){*
*BereichsFeld[i]++;*
*Rest–;}}*
*...*
In Tab.3.2 sind die Start- und Ende-Grenzen, in denen die Threads arbeiten dargestellt. Die

| BereichsFeld | Wert | Anzahl der Elemente |
|---|---|---|
| 1 | 1504 | 1504 |
| 2 | 3008 | 1504 |
| 3 | 4511 | 1503 |
| 4 | 6014 | 1503 |

Tabelle 3.2: Aufteilung des Workloads

| Thread | Start | Ende |
|---|---|---|
| 1 | 0 | 1503 |
| 2 | 1504 | 3007 |
| 3 | 3008 | 4510 |
| 4 | 4511 | 6013 |

Tabelle 3.3: Grenzen der Workloads der Threads

Anzahl der Elemente, die sie bearbeiten wird ebenfalls aufgezeigt.
Tab.3.3 stellt die Grenzen dar. Die Zuordnung der Grenzen erfolgt mittels des Feldes *BereichsFeld*.
Beispiel:
*...*
*for (int i=1; i < BereichsFeld.size(); i++){*
*Thread[i].Start = BereichsFeld[i-1];*
*Thread[i].Ende = BereichsFeld[i]-1;}*
*Thread[0].Start = 0;*
*Thread[0].Ende = BereichsFeld[0]-1;*
*...*

So wurde jedem Thread ein Bereich zugeordnet, den er bearbeiten muss. Hierbei werden keine Arbeiten doppelt ausgeführt.
Jeder Thread erhält den gleichen *BST*, in dem er nach Elementen sucht.
Hier musste eine Entscheidung getroffen werden, wie jeder einzelne Thread auf den *BST* zugreifen darf. Es besteht die Möglichkeit jedem Thread den *BST* als Referenz oder als Kopie zu übergeben.
Zuerst wurde der *BST* als Referenz übergeben. Dies führte jedoch zu Inkonsistenzen im Zugriff auf den Baum. Die Klasse des *BST* wurde daraufhin modifiziert, um eine Zugriffsicherheit gewährleisten zu können.
Da verschiedene Threads mit dem Baum arbeiten, ist es wahrscheinlich, dass sie auf unterschiedliche Bereiche zugreifen. Diese Zugriffe auf verschiedene Speicherbereiche kosten Zeit, wenn sich die gewünschten Daten nicht im Cache befinden. Eine mögliche Look-Ahead

Strategie des Systems bezüglich des Cacheinhalts greift dadurch nicht. Es wurde daher darüber nachgedacht, jedem Thread eine Kopie des Baumes zu spendieren. Ein Vorteil, wenn die Prozessoren eigene Cache Speicher besitzen. Ein erster Test zeigte aber gleich den Nachteil dieser Strategie. Das Kopieren des Baumes für einen Thread dauerte deutlich länger als die eigentliche Ausführung des Algorithmus. Daher wurde dieser Ansatz wieder verworfen.
Jeder Thread arbeitet damit auf demselben *BST*.
Es gibt verschiedene Ansätze, wie sich der Haupt-Thread verhält, der die anderen aufruft. Er kann einen Teil der Arbeit übernehmen, die anderen Threads koordinieren oder einfach warten, bis die Arbeit erledigt ist.
Zwischen den Threads wird keine Koordination benötigt und das Warten auf die Erledigung der Arbeit besteht nicht nur aus Warten. Periodisch überprüft der Haupt-Thread, ob die Threads, die er erschaffen hat noch existieren. Dieses periodische Aufwecken und wieder Schlafen legen, beansprucht die Prozessorkerne. Mindestens einer der arbeitenden Threads muss suspendiert werden. Aus diesen Gründen führt der Haupt-Thread seinerseits einen Teil der Arbeit aus. Dies hat auch den Vorteil, dass weniger Speicher für die Threads reserviert werden muss. So kann man mit drei vom Haupt-Thread erschaffenen Threads die Arbeit auf vier Threads aufteilen.
Um dies realisieren zu können wurde der Thread-Klasse eine Methode hinzugefügt. *StartNoThread()* führt die Funktion aus, ohne als Thread zu starten.
*run()* wurde dementsprechend so implementiert, dass es *StartNoThread()* aufruft. Hiermit ist sichergestellt, dass die gleiche Arbeit erledigt wird.

### 3.3.3 Binärer Suchbaum

Es wird ein Binärer Suchbaum (*BST*) erstellt, der alle alten Elemente enthält. Die Elemente in dem *BST* sind nach ihrer Lage im Raum sortiert. So kann man schnell anhand der Entfernung zwischen einem Element aus dem *BST* und einem neuen Element darauf schließen, wie weit die Elemente voneinander entfernt sind und ob sie sich unter Umständen schneiden.

### 3.3.4 Methoden zum Füllen des *BST*

Der *BST* wird mit Elementen gefüllt. Diese repräsentieren die des alten Netzes. Die Elementerstellung der *BST*-Elemente wurde mit drei unterschiedlichen Ansätzen durchgeführt. Jedes dieser erstellten Elemente erhält einen Verweis auf das Element des alten Netzes, aus

dem es gebildet wurde. Ermittelt man ein Element aus dem *BST* welches für das *Remapping* verwendet werden soll, hat man schnell das dazugehörige Element des alten Netzes.

#### Altes Element mit Bounding Box

Es wird eine Bounding Box um das alte Element herum aufgespannt (Verfahren siehe Seite 17). Mit diesen Daten wird der *BST* gefüllt. Die Bounding Box ist im dreidimensionalen Fall ein Hexaeder und stellt unter Umständen das Element unzureichend dar (siehe Abb.1.8 Seite 18).
Der Vorteil dieses Ansatzes liegt in der Tatsache, dass die Bounding Box für das alte Element schnell ermittelt werden kann.

#### Altes Element als Integration Point

Das alte Element wird auf seinen Integration Point reduziert. Der Integration Point ist in diesem Fall der Punkt, welcher in der geometrischen Mitte der Eckpunkte liegt.

$$\frac{\sum_{i=1}^{n} x_i}{n}, \text{ mit n = Anzahl der Knoten} \tag{3.5}$$

Hier ist die Formel für die x-Koordinate angegeben. Gleiches muss für y und z durchgeführt werden.
Unter der Vorraussetzung einer geraden Kantenführung, liegt der gesuchte Punkt mit hoher Wahrscheinlichkeit in dem alten Element.
Ist das alte Element sehr groß, kann es zu einer Schnittmenge zwischen alten und neuem Element kommen. Ein Nachteil ist die Tatsache, dass der Integration Point des alten Elementes weit von dem neuen Element entfernt liegt, während die Oberfläche des alten Elementes unter Umständen das neue Element schneidet. Dies ist der Fall, wenn das alte Element relativ groß ist, im Vergleich zu anderen alten Elementen deren Integration Point näher am neuen Element liegt. Somit kann es sein, dass ein altes Element für die Berechnung des Ergebniswertes eines neuen Elementes vernachlässigt wird, da der Integration Point zu weit entfernt liegt.

#### Oberfläche des alten Elementes in Dreiecke zerlegt

Die Oberfläche des alten Elementes wird in Dreiecke zerlegt.
Um jedes Dreieck wird eine Bounding Box erstellt. Das Zentrum der Bounding Box wird in der Regel nicht auf der Fläche des Dreiecks liegen (ausgenommen das Dreieck ist parallel zu einer Achse). Diese Bounding Boxes werden nun in den *BST* eingefügt und sortiert.
Weiterhin wird für jede Bounding Box gespeichert, zu welchem alten Element sie gehört.
Mit höherer Wahrscheinlichkeit werden nun große alte Elemente, mit weit entfernten Integration Points in die Betrachtung mit einbezogen.
Ein Nachteil besteht in der erhöhten Anzahl an Elementen, die in den *BST* eingefügt werden müssen.

Die drei eben beschriebenen Ansätze zur Füllung des *BST* wurden durchgeführt. Es lag

die Vermutung nahe, dass die dritte Methode (*Zerlegung der Oberfläche in Dreiecke*) die Erfolgversprechendsten Ergebnisse liefert. Gefolgt von der Methode der *Bounding Boxes* und an letzter Position der *Integration Point*-Ansatz.

NEAREST NEIGHBOUR

Es wurde zuerst der einfachst mögliche Algorithmus implementiert, um ein Gefühl für Geschwindigkeit und Genauigkeit zu erhalten. Dieser Algorithmus sucht sich das am nähsten liegende alte Element zu einem neuen Element und übernimmt die Ergebniswerte. Dieses dichteste alte Element ist der *Nearest Neighbour* (*NN*).
Als Erweiterung werden die x am nähsten liegenden alte Elemente genommen. Wobei x eine beliebige Anzahl ist, die in der Testumgebung über einen Parameter angegeben werden kann.
Diese x *NN* werden dann dazu herangezogen, den Ergebniswert für das neue Element zu ermitteln. Es findet keine Überprüfung einer Schnittmenge statt.
Die Ergebniswerte der alten Elemente werden gewichtet. Die Gewichtung resultiert aus der Entfernung der alten Elemente zu dem neuen Element. Hierbei werden die Elemente, die näher liegen mit einem höheren Gewicht versehen, als die weiter entfernt liegenden. Die Gewichtung wurde mit folgender Formel durchgeführt.

$$f(x) = \frac{\sum_{i=1}^{n} x_i}{x}, \text{ mit n = Anzahl der NN} \tag{3.6}$$

$$g(x) = \frac{f(x)}{\sum_{j=1}^{n} f(x_j)}, \text{ mit n = Anzahl der NN} \tag{3.7}$$

g(x) ist die Gewichtung mit der die Ergebniswerte des alten Elementes in die Ergebniswerte des neuen Elementes einfließen.
Exemplarisch soll hier ein Beispiel aufgeführt werden. Die Gewichtungen von vier Punkten, die Entfernungen zu einem Punkt *X* und der Ergebniswert für den Punkt werden in Tab.3.4 dargestellt. Man erkennt, dass Elemente, die weiter entfernt liegen (*D*) schwächer einbezo-

| Elemente | Entf.zu X | Ergebniswert Elem. | Gewichtung | Gew.Ergebnis |
|---|---|---|---|---|
| A | 3 | 36 | 0.2912 | 10.4854 |
| B | 5 | 42 | 0.1748 | 7.3398 |
| C | 2 | 14 | 0.4369 | 6.1165 |
| D | 9 | 75 | 0.0971 | 7.2816 |
| Summe | | | | 31.2233 |

Tabelle 3.4: Werteübertragung mittels gewichteter Entfernung

gen werden, als Elemente die näher an Punkt *X* liegen (*C*). Element *D* hat einen höheren Ergebniswert als Element *B*, dennoch sind ihre Anteile am Ergebniswert für *X* beinahe identisch.

Diese *NN*-Methoden wurden mit den Ansätzen kombiniert. Es wurde hier bewusst auf ein Volumenprinzip verzichtet, da ein solches Verfahren viele Rechenschritte umfasst. Hierdurch würde die gesamte Rechenzeit des Algorithmus negativ beeinflusst werden.

Es ist darauf zu achten, dass vektorielle Größen nicht so behandelt werden, wie andere Ergebniswerte. Zu den vektoriellen Größen gehört die Spannung, welche innerhalb eines Körpers wirkt. Sie setzt sich zusammen aus den Haupt-($\sigma_{xx}$, $\sigma_y$, $\sigma_{zz}$) und den Schubspannungen ($\tau_{xy}$, $\tau_{xz}$, $\tau_{yz}$). Die Hauptspannungen wirken senkrecht, während die Schubspannungen tangential zur Fläche wirken.
Hier gab es folgende zwei Ansätze, wie mit den vektoriellen Größen umzugehen ist:

- Absoluter Maximalwert
- Maximalwert

| Absoluter Maximalwert | Maximalwert | |
|---|---|---|
| -7<br>-18<br>5<br>4 | -7<br>-18<br>5<br>4 | |
| -18 | 5 | Ermittelter Wert |

Tabelle 3.5: Vergleich zwischen Absoluten Maximalwert und Maximalwert

ABSOLUTER MAXIMALWERT
Es wird der Ergebniswert des alten Elementes übernommen, der den höchsten absoluten Maximalwert hat. Hiermit hat man den maximalen Wert, unabhängig davon, ob der Ergebniswert negativ oder positiv ist (siehe Tab.3.5)
MAXIMALWERT
Für das neue Element wird der Maximalwert der vektoriellen Ergebniswerte der geometrisch dichtesten alten Elemente genommen. Damit ist sichergestellt, dass immer der größtmögliche Spannungswert für das neue Element übernommen wird. (siehe Tab.3.5)

## 3.4 Polygonzug nach Hippmann

Das *Polygonal Contact Model* (*PCM*) von Dr. Gerhard Hippmann [Hip04] bietet eine Möglichkeit Schnittvolumina zu berechnen. Es sollte eigentlich vermieden werden, dass die Werteübertragung über Schnittvolumina geschieht. Es handelt sich hierbei jedoch um einen anderen Ansatz. Daher soll untersucht werden, ob diese Methode schneller rechnet und genauere Ergebnisse liefert.

### 3.4.1 PCM

Das *PCM* liefert einen Polygonzug, der die Umrandung der Fläche zweier sich schneidender Körper beschreibt. Es zerlegt die Oberfläche der beiden Modelle in Dreiecke und überprüft dann die Überschneidungen dieser Dreiecke. Die gefundenen Schnittpunkte werden dann zu einem Polygonzug zusammengefügt.
Die Schnittpunkte zwischen den Dreiecken werden nach einer Methode von Tomas Möller berechnet [Möl97].
Die Einzelheiten der Schnittpunktbestimmung nach Möller und der *PCM* werden hier nicht genauer erläutert.

### 3.4.2 Schnittpolyeder erstellen

Nach der Bestimmung des Polygonzuges, sollte dieser zu einem Polyeder kompletiert werden. Mit Hilfe dieses Polyeders wird dann das Schnittvolumen ermittelt. Aus dem errechneten Schnittvolumen und dem Volumen des neuen Elementes wird dann eine Gewichtung für die Ergebniswerte bestimmt.

$$E_{neu} = \frac{E_{alt} * V_{schnitt}}{V_{neu}} \tag{3.8}$$

Es muss ermittelt werden, welche Eckpunkte der Elemente vollständig innerhalb des anderen Elementes liegen. Dies wird überprüft, indem von jedem Eckpunkt, der in Frage kommt, eine Gerade parallel zu einer Achse gezogen wird. Bei zwei Hexaedern müssen also sechzehn Punkte überprüft werden. Die Gerade wird parallel zu einer Achse gewählt, da sie dadurch schnell und einfach zu berechnen ist. Schneidet diese Gerade das andere Element in einer geraden Anzahl an Schnitten, liegt es außerhalb des Elementes. Schneidet es in einer ungeraden Anzahl, muss es nicht innerhalb des Elementes liegen, kann aber. Hat die Gerade einen Eckpunkt oder eine Kante getroffen, liegt keine Sicherheit vor. Dann wird eine neue Gerade von dem Eckpunkt aus erzeugt, dieses mal parallel zu einer anderen Achse. Trifft die Gerade wieder auf einen Eckpunkt oder eine Kante des anderen Elementes, wird eine dritte Gerade erzeugt und erneut die Schnittpunkte überprüft.
Abb.3.4 verdeutlicht das Problem im zweidimensionalen Fall. Hier darf die gezogene Gerade keinen Eckpunkt treffen und nicht auf einer der Kanten sein.
Die rote Gerade wurde als erstes gewählt. Sie schneidet das Dreieck einmalig. Dies würde bedeuten, dass der Eckpunkt in dem Dreieck liegt. Wie man sehen kann, tut er dies nicht. Daher muss eine weitere Gerade parallel zu einer anderen Achse erstellt werden. Die blaue Gerade schneidet das linke Dreieck des rechten Elementes einmalig, das rechte Dreieck des rechten Elementes zweimalig. Sie schneidet weder einen Eckpunkt, noch liegt sie auf einer Kante. Daher weiß man, dass der Eckpunt in dem linken (ein Schnitt), aber nicht in dem

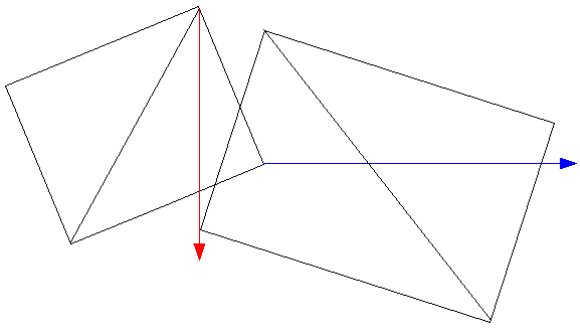


Abbildung 3.4: Ermittlung ob ein Punkt in einem Körper liegt

rechten (zwei Schnitte) Dreieck liegt.
Sind die Körper unregelmäßig geformt kann es auch zu mehr als zwei Schnitten kommen. Die Wahrscheinlichkeit ein unregelmäßiges Element zu haben, ist sehr hoch, da es einer Umformung unterzogen wurde. Ein weiteres Problem stellt die dritte Dimension dar.

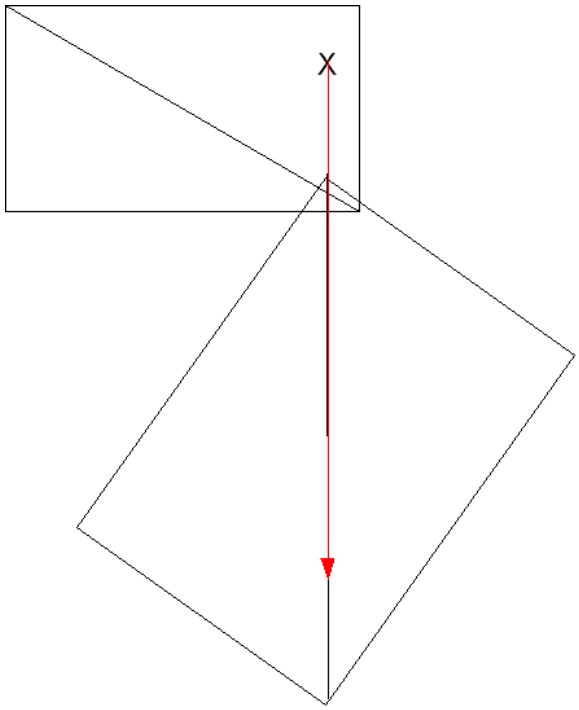



Abbildung 3.5: Probleme bei Gerade auf Kante

Beispiel:
Ein Eckpunkt des alten Elementes soll überprüft werden. Das neue Element wird in zwölf Dreiecke zerlegt. Von dem Eckpunkt (*E1*) wird eine Gerade gebildet. Die Schnittpunkte zwi-

schen *E1* und den zwölf Dreiecken werden ermittelt. Wie im zweidimensionalen Fall bedeutet eine ungerade Anzahl an Schnittpunkten, dass *E1* innerhalb des neuen Elementes liegt.
Schneidet die Gerade einen Eckpunkt, gibt es das bereits bekannte Problem. Es gibt allerdings einen Unterschied zwischen dem zwei- und dem dreidimensionalen Fall. In 2D darf die Gerade nicht auf einer der Kanten liegen (siehe Abb.3.5). In 3D ist dieses Kriterium verschärft. Hier darf die Gerade die Kante nicht schneiden.
Schneidet die Gerade eine Kante, gilt dies als Schnittpunkt mit zwei Dreiecken. Da zwei Dreiecke sich eine Kante teilen, bedeutet ein Schnittpunkt auf einer Kante, dass dieser Punkt doppelt gezählt wird. Einmal für jedes Dreieck. Hierdurch kommt es zu einer Verzerrung der Schnittpunkanzahl. Daher kann nicht mit Gewissheit ermittelt werden, ob *E1* innerhalb des neuen Elementes liegt.
Liegt also ein Schnittpunkt auf einer Ecke oder auf einer Kante, muss eine neue Gerade ermittelt werden und die Anzahl der Schnittpunkte neu ermittelt werden.
Die Ermittlung, ob ein Punkt innerhalb eines Polyeders liegt ist eine rechenintensive Aufgabe. Es kann nicht vorhergesagt werden, wie oft eine Gerade auf einen Eckpunkt oder eine Kante trifft. Eine Vorhersage über die Laufzeit dieses Algorithmus kann damit nicht getroffen werden.
Das *Remapping* eines Modells mit vielen Elementen kann schnell berechnet werden, wenn die Gerade nicht allzu häufig neu ermittelt werden müssen. Im Gegenzug kann ein Modell mit wenigen Elementen lange rechnen, würde die Gerade häufig neu ermittelt.

Bei der Implementierung des Algorithmus ist das Phänomen aufgetreten, dass Polygonzüge nicht geschlossen werden konnten. Wie in [Hip04] vorgeschlagen, wurde mittels einer Entfernung $d_\epsilon$ nach geeigneten Punktkandidaten in der Menge der Punkte des Polyeders gesucht, um den Polygonzug zu schließen. Auch diese Suche verbrauchte nicht zu kalkulierende Rechenzeit. Für einige Punkte musste eine Schranke $d_\epsilon$ gewählt werden, die zu hoch war, um akzeptabel zu sein. Zur Schließung des Polygons musste teilweise eine Schranke gewählt, die größer, als die Elementgröße war. Solche Polygonzüge konnten nicht verwendet werden. Dadurch wurden einige Schnitte nicht in die Ergebnisübertragung einbezogen. Die Ergebnisse einer solchen Berechnung sind Fragwürdig.

### 3.4.3 Schnittvolumen ermitteln

Hat man den Polyeder berechnet, kann man versuchen das Volumen des Polyeders zu berechnen. Dies ist nicht trivial.

Eine mögliche Methode:
Es wird ein Punkt ermittelt, der innerhalb des Polyeders liegt. Die Methode ist bereits aus dem vorigen Kapitel bekannt. Es könnten nun Pyramiden von den Flächen zu dem inneren Punkt erstellt werden. Von diesen Pyramiden kann man das Volumen einfach berechnen. Da

aber nicht alle Flächen des Polyeders eine Pyramide mit dem inneren Punkt bilden können, sollten Tetraeder verwendet werden.
Hier zeigt sich dann aber auch der Nachteil dieser Methode. Abb.3.6 zeigt im zweidimensionalen zwei rote Dreiecke. Sie wurden durch die Eckpunkte des Polygons und einen frei gewählten Punkt im Inneren gebildet. Die roten Dreiecke tragen jedoch mehr zum Volumen des Polyeders bei, als sie dürften.
Abb.3.7 zeigt eine mögliche Lösung. Um Konflikte zu vermeiden, muss kontrolliert werden, ob sich eine Gerade zwischen einem Eckpunkt und einem Inneren Punkt mit einer Kante schneidet. Ist dies der Fall, wird sie nicht gezogen. Kann der erste gewählte Innere Punkt (*P1*) mit einem Eckpunkt verbunden werden, wird er selber zu einem Eckpunkt in der weiteren Betrachtung. Es wird ein neuer Innerer Punkt *P2* gewählt und wieder versucht, alle Eckpunkte mit ihm zu verbinden, ohne eine Kante zu schneiden. Kommt es zu einem Schnitt, wird *P2* zu einem Eckpunkt und ein weiterer Punkt wird ermittelt.
Bei der Ermittlung neuer Punkte ist darauf zu achten, dass der neue Punkt nicht innerhalb eines Bereiches liegt, der von einem vorherigen Punkt aufgespannt wird.
Diese Berechnungen sind nicht trivial und nehmen einige Rechenzeit in Anspruch. Auch hier kann nicht vorhergesagt werden, wie lange eine solche Berechnung dauert. Die Laufzeit hängt von der Wahl des Inneren Punktes ab.

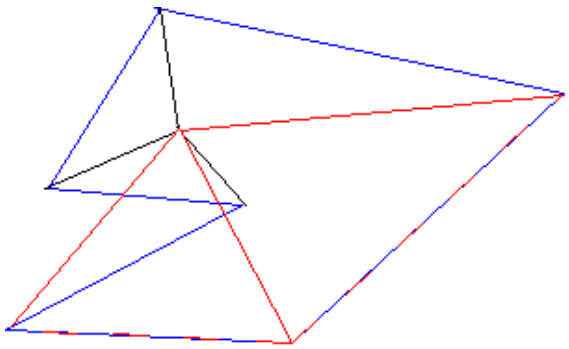


Abbildung 3.6: Probleme bei Volumenermittlung

### 3.4.4 Bewertung

Die Berechnung der Polygonzüge, die anschließende Berechnung der Polyeder und die daran anschließende Berechnung des Schnittvolumens sind Berechnungen, die äußerst Rechenintensiv sind. Ihre Laufzeiten aufgrund der Sonderfälle nicht vorhersagbar. Trotz parallelem Ansatz, zeigten erste Tests, dass bereits die Berechnung der Polygonzüge und der Polyeder mehr als doppelt so viel Zeit in Anspruch nimmt, wie das *Remapping* durch den nicht parallelisierten originalen Algorithmus.

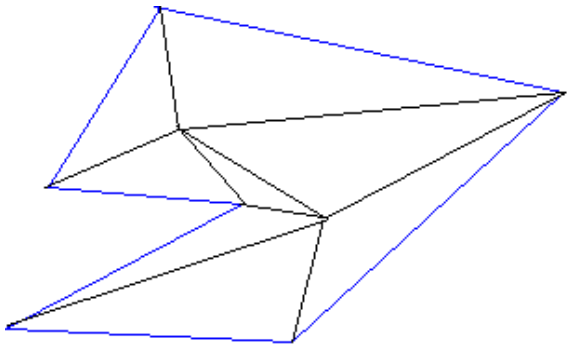


Abbildung 3.7: Mögliche Lösung bei Volumenermittlung

# 4 Auswertung der Testergebnisse

## 4.1 Testbedingungen

### 4.1.1 Getestete Modelle

Verschiedene Modelle mit einer unterschiedlichen Anzahl an Elementen und Knoten wurden für die Tests ausgewählt (siehe Tab.4.1).

Des weiteren wurde das Modell *Tiefziehhülse* gewählt, welches in dieser Arbeit optisch gezeigt werden darf. Aus rechtlichen Gründen dürfen einige der Modelle nicht veröffentlicht werden.

TIEFZIEHHÜLSE
Dieses Modell stellt ein gepresstes Verbindungsstück dar. Abb.4.1 zeigt den Aufbau und die Durchführung einer Simulation. In der Mitte liegt das Werkstück (grau dargestellt), welches verformt werden soll. Unten befindet sich die Auflagefläche (violett) und von oben senkt sich die Presse (grün) mit einem Druck von ca.3150 Tonnen. Dieses Modell wird so, wie es dargestellt ist, in der Wirtschaft eingesetzt. Das Werkstück besteht aus Stahl. Es wird mit dem Verfahren des Gesenkschmiedens verformt.
Ausschlaggebend für die Auswahl der Modelle war das Verhältnis zwischen den Knoten und den Elementen des alten und des neuen Netzes (siehe Tab.4.2 und Tab.4.3). Es wurden Modelle ausgesucht, welche die möglichen Vorkommen an Modellen exemplarisch abbilden können.
Eine Verfeinerung des Modells (alte Elemente < neue Elemente) wurde ebenso ausgewählt,

| Modell | Kn. alt. Netz | Kn. neu. Netz | Elm. alt. Netz | Elm. neu. Netz |
|---|---|---|---|---|
| 3DFEFE | 2067 | 1511 | 1564 | 6041 |
| Doppelpleuel2 | 8550 | 164652 | 5866 | 142098 |
| Torx3FEBox | 36769 | 6681 | 34575 | 5163 |
| Tripode4FE | 32666 | 6503 | 28280 | 27583 |
| Tiefziehhülse | 5814 | 8974 | 4754 | 7737 |

Tabelle 4.1: Knoten- und Elementanzahl der getesteten Modelle

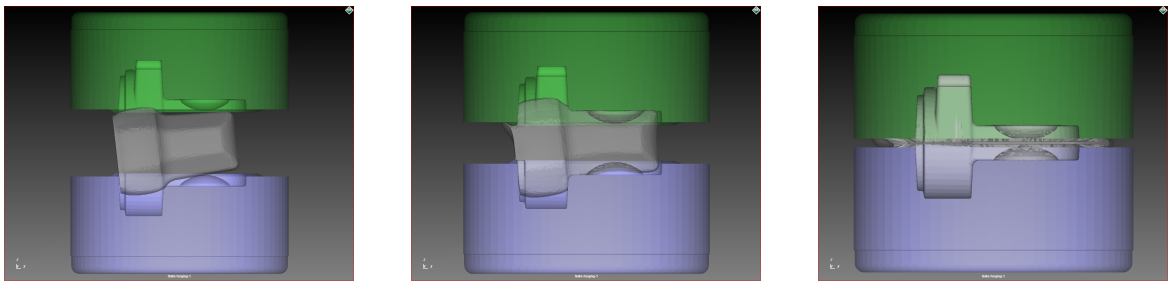


Abbildung 4.1: Simulation des Gesenkschmiedens eines Werkstückes

| Modell | neue Knoten/alte Knoten | neue Elemente/alte Elemente |
|---|---|---|
| 3DFEFE | 0,7310 | 3,8625 |
| Doppelpleuel2 | 19,2575 | 24,2240 |
| Torx3FEBox | 0,1817 | 0,1493 |
| Tripode4FE | 0,199 | 0,9753 |
| Tiefziehhülse | 1,6274 | 1,5435 |

Tabelle 4.2: Verhältnis der Knoten und Elemente zwischen neuem und altem Netz

wie eine Vergröberung des Netzes (alte Elemente > neue Elemente). Auch eine Wandlung von Elementkörpern wurde mit einbezogen (alte Knoten $\neq$ neue Knoten). Als drittes wurden noch Modelle gewählt, bei denen sowohl Element-, als auch Knotenanzahl ungefähr gleich geblieben ist (alte Elemente/Knoten $\simeq$ neue Elemente/Knoten).

### 4.1.2 Systeme

Um vergleichen zu können, wie sich die Laufzeit mit steigender Anzahl von Threads verändert, wurden die Tests auf verschiedenen Systemen durchgeführt (siehe Tab.4.4). Die genauen Spezifikationen der einzelnen Systeme spielen hier keine Rolle. Es wird kein

| Anzahl | Modellauswahl |
|---|---|
| alte Knoten = neue Knoten | 3DFEFE |
| alte Knoten > neue Knoten | Torx3FEBox |
| alte Knoten < neue Knoten | Doppelpleuel2 |
| alte Elemente = neue Elemente | Tripode4FE |
| alte Elemente > neue Elemente | Torx3FEBox |
| alte Elemente < neue Elemente | Doppelpleuel2 |

Tabelle 4.3: Auswahl der Modelle

| Name | Prozessoren | Kerne pro Prozessor | ges. Anzahl Kerne |
|---|---|---|---|
| Pluto | 1 | 2 | 2 |
| Ananke | 1 | 4 | 4 |
| Orion | 2 | 2 | 4 |

Tabelle 4.4: Systeme auf denen getestet wurde

Vergleich zwischen verschiedenen Systemen durchgeführt. Die Verbesserung des Laufzeitverhaltens auf einem System steht im Mittelpunkt.
Auf dem System Pluto ist Microsoft Windows XP 64-Bit installiert, ansonsten ist die 32-Bit-Version des Betriebssystems verwendet worden.

### 4.1.3 Ausreißer eliminieren

Jedes Modell wurde fünfzig mal auf jedem System berechnet. Damit konnte eine durchschnittliche Bearbeitungszeit errechnet werden.
Es kann geschehen, dass Ausreißer auftreten. Werden Programme während der Durchläufe ausgeführt, wirkt sich das negativ auf die gemessene Zeit aus.
Der Thread muss angehalten werden, wenn ein anderes Programm vom Scheduler aktiviert wird. Später wird der Thread dann wieder gestartet. Die Zeit, die verstreicht bis der Thread ruht, bzw. die er für die Reaktivierung benötigt, fließt mit in die gemessene Zeit ein.
Um diese Einflüsse so gering wie möglich zu halten, wurden die Testdurchläufe auf ruhenden Systemen durchgeführt. Ein ruhendes System ist hierbei ohne Einfluss eines Benutzers.
Trotzdem kann es zu Unterbrechungen des Durchlaufs kommen, wenn Systemdienste den Prozessor beanspruchen und damit Threads verdrängen (z. B. der Scheduler).
Diese Beeinflussungen der Zeit sollen heraus gefiltert werden. Dadurch ist es möglich eine repräsentative Aussage zu treffen. Mittels Medianbildung sollen die Ausreißer aus der Auswertung heraus gehalten werden.

## 4.2 Auswertung der Ergebnisse des bestehenden Algorithmus

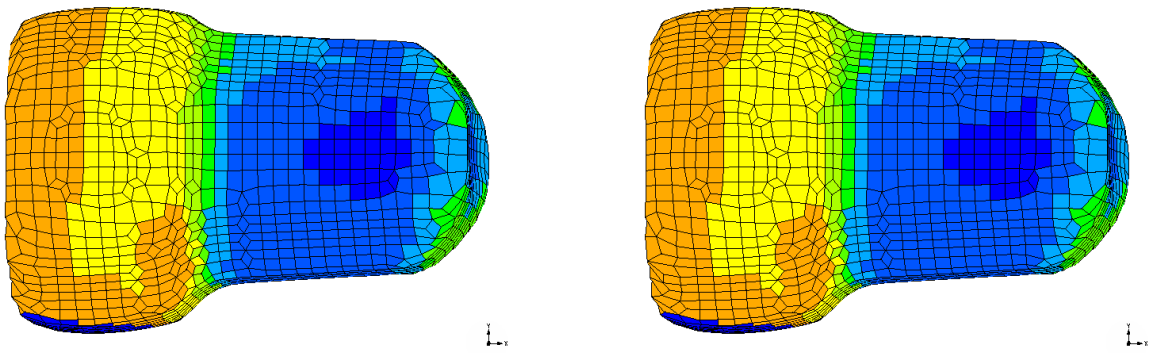
### 4.2.1 Vergleich der Ergebnisse

Nicht nur die Beschleunigung spielt eine Rolle, wenn versucht wird, ein Programmablauf zu verbessern. Es ist darauf zu achten, dass die Ergebnisse korrekt bleiben.

Beispielhaft werden im Folgenden zwei Ergebniswerte dargestellt. Exemplarisch für normale Ergebnisgrößen wurde EFFPLS gewählt. EFFPLS steht für *Effective Plastic Strain* ( = Umformgrad). Der Umformgrad beschreibt den Grad der Verformung. Er ist positiv bei einer Streckung bzw. negativ bei einer Stauchung des Werkstückes.
Als Beispiel für vektorielle Ergebnisdaten wurde $\tau_{xy}$ genommen.

$$S = \begin{bmatrix} \sigma_x & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \sigma_y & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \sigma_z \end{bmatrix} \tag{4.1}$$

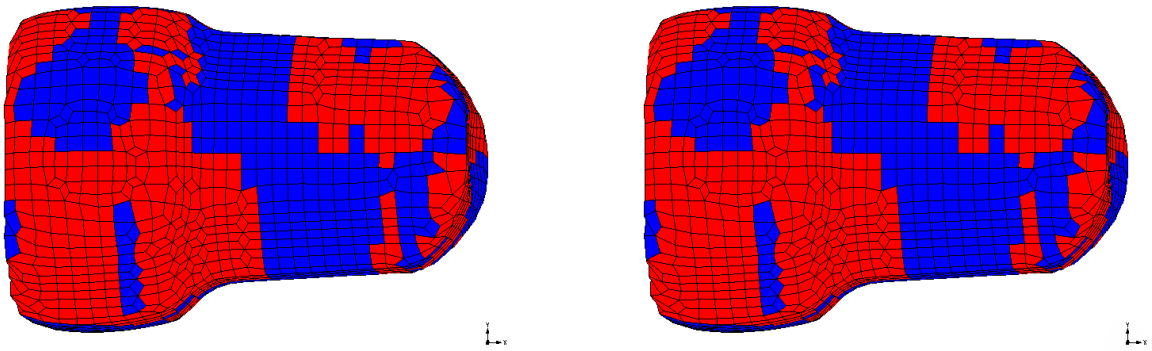
$\tau_{xy}$ ist eine Schubspannung und Teil des Spannungstensors (siehe Formel 4.1). Der Spannungstensor setzt sich aus den Haupt- ($\sigma_{xx}$, $\sigma_{yy}$, $\sigma_{zz}$)und Schubspannungen ($\tau_{xy}$, $\tau_{xz}$, $\tau_{yx}$, $\tau_{yz}$, $\tau_{zx}$, $\tau_{zy}$) zusammen. Er beschreibt die Kräfte, die auf einen Punkt im Körper wirken, durch den drei gedachte Schnittflächen gehen. Die Hauptspannungen wirken senkrecht zu den Flächen des Körpers. Die Schubspannungen wirken tangential zur Fläche. In Abb.4.2



(a) EFFPLS Ergebnis Original (b) EFFPLS Ergebnis OpenMP

Abbildung 4.2: Visuelle Darstellung des EFFPLS Ergebnisses

sieht man eine Gegenüberstellung zwischen den Ergebniswerten des unveränderten Algorithmus und des mittels OpenMP parallelisierten Algorithmus.
Optisch gibt es keine Unterschiede zwischen den Darstellungen und auch ein Vergleich der Werte ergab keine Diskrepanzen.
Wird eine Parallelisierung vorgenommen, muss sichergestellt sein, dass das Ergebnis stets dasselbe ist. Unabhängig davon, von wie vielen Threads die Arbeit ausgeführt wird.
Abb.4.3 zeigt graphisch die Ergebniswerte für $\tau_{xy}$ bei der Berechnung mit einem bzw. vier Threads an. Auch hier gibt es weder optisch, noch in den numerischen Werten Unterschiede. Die gewählte Methode ist ein gangbarer Weg bei der Beschleunigung des Algorithmus, da die Ergebnisse mit denen des originalen Algorithmus übereinstimmen.

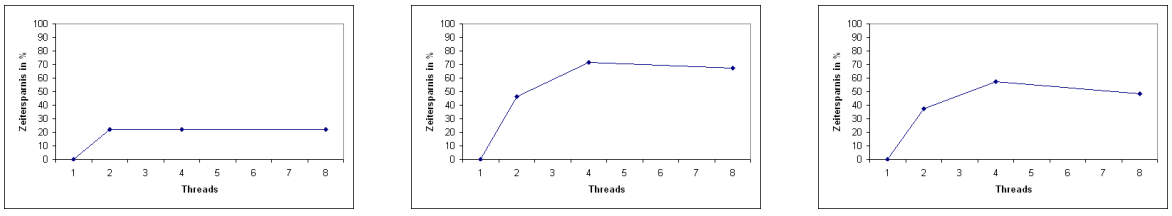


(a) $\tau_{xy}$ Ergebnis OpenMP mit 1 Thread (b) $\tau_{xy}$ Ergebnis OpenMP mit 4 Threads

Abbildung 4.3: Visuelle Darstellung des $\tau_{xy}$ Ergebnisses

### 4.2.2 Zeitersparnis durch Parallelisierung

Wie bereits dargelegt, werden die Ergebnisse durch die Parallelisierung nicht verändert. Es wird nun untersucht, wie sich das Laufzeitverhalten verändert.
Hierfür wurde die prozentuale Beschleunigung der Laufzeit genommen. In Graphen werden die Angaben für zwei, vier und acht Threads dargestellt.
Bei den Systemen *Ananke* und *Orion* ist ein Abfall der Beschleunigung bei der Berechnung mit mehr als vier Threads zu erkennen, wenn mit mehr als vier Threads gerechnet wird. Beide Systeme haben vier Kerne und können dementsprechend viele Threads parallel arbeiten lassen. Steigt nun die Anzahl der rechnenden Threads über die Anzahl der Kerne eines Systems, werden arbeitende Threads immer wieder von Scheduler suspendiert. Dieser Vorgang kostest Rechenzeit. Daher sollten maximal so viele Threads erschaffen werden, wie es Kerne im System gibt. Tab.4.5 gibt einen Überblick über die Zeitersparnis bei der Berechnung




(a) Pluto (b) Ananke (c) Orion

Abbildung 4.4: Laufzeitbeschleunigung der Umformung 3DFEFE auf verschiedenen Systemen

der verschiedenen Modelle auf den Systemen. Man erkennt, dass die Beschleunigung nicht

nur vom System abhängt. *Tripode4FE* weist auf allen Systemen eine geringere Verbesserung der Laufzeit auf, als die anderen Modelle.
Wie zu erkennen ist, wird auf System *Pluto* eine Verbesserung von ca. 20% erreicht. Dies ist ein unerwartet schlechtes Ergebnis. Das Programm wurde als 32-Bit-Version kompiliert und auf dem System mit dem 64-Bit-System ausgeführt. Hieraus resultiert unter Umständen die geringere Verbesserung des Laufzeitverhaltens, da das Programm emuliert werden muss.
Die Unterschiede in der Zeitersparnis zwischen *Ananke* und *Orion* resultieren aus dem Aufbau des Systems. Beide haben vier Kerne, da liegt die Vermutung nahe, dass die Beschleunigung gleich ist. *Orion* bringt aber nicht dieselben zeitlichen Vorteile wie *Ananke*. Grund hierfür ist, dass *Orion* mit zwei Prozessoren arbeitet, die beide ihren eigenen (L2-)Speicher haben. *Ananke* hingegen verfügt nur über einen (L2-)Speicher. Die Daten für die Berechnung müssen bei *Orion* über die Prozessorgrenzen hinweg zur Verfügung gestellt werden. Diese Synchronisation benötigt Zeit, die bei *Ananke* nicht anfällt, da es nur einen Prozessor hat.

| System | Modell | Zeitersp. 2 Thr. | Zeitersp. 4 Thr. | Zeitersp. 8 Thr. |
|---|---|---|---|---|
| Pluto | 3DFEFE | 22,19490441 | 22,20048862 | 22,17815178 |
| | Doppelpleuel2 | 21,50233505 | 21,50506281 | 21,4780204 |
| | Torx3FEBox | 22,08573016 | 21,98993554 | 22,09698801 |
| | Tripode4FE | 18,11663131 | 18,12476601 | 18,14117408 |
| Ananke | 3DFEFE | 46,14447826 | 71,47695936 | 67,41140274 |
| | Doppelpleuel2 | 46,69384256 | 67,49908086 | 59,07830311 |
| | Torx3FEBox | 49,67899192 | 70,93699669 | 60,42584938 |
| | Tripode4FE | 41,0161395 | 63,34092855 | 58,11569565 |
| Orion | 3DFEFE | 37,44309081 | 57,12020238 | 48,55742909 |
| | Doppelpleuel2 | 36,36481439 | 52,96901339 | 47,5045638 |
| | Torx3FEBox | 39,44585666 | 56,67548102 | 49,49908382 |
| | Tripode4FE | 34,84979284 | 50,49083073 | 46,22810443 |
| | Tiefziehhülse | | | |

Tabelle 4.5: Prozentuale Zeitersparnis bei der Berechnung verschiedener Modelle auf unterschiedlichen Systemen

## 4.3 Auswertung der Ergebnisse des implementierten Algorithmus

Im Gegensatz zur Auswertung der Parallelisierung des vorhandenen Algorithmus, musste bei implementierten Algorithmen anders vorgegangen werden.

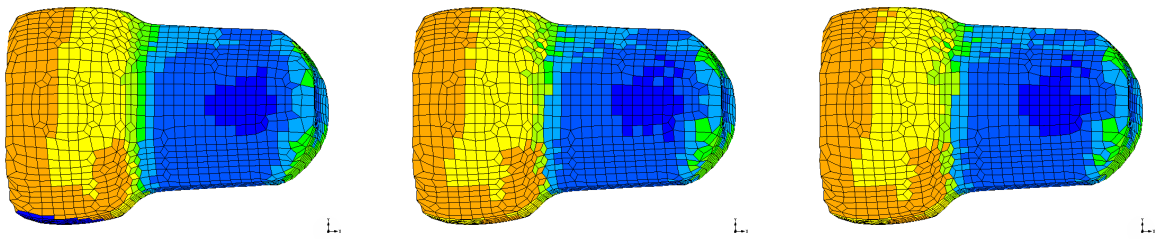
Da klar war, dass die Ergebnisse unterschiedlich sein würden, war es wichtig diese Unterschiede zu untersuchen.
Die primäre Gewichtung bei der Auswertung liegt nicht mehr allein auf dem Zeitgewinn. Die Ergebnisse müssen stimmig sein. Daher werden zuerst die Ergebnisse verglichen. Sollten diese nicht korrekt sein, werden keine Laufzeituntersuchungen durchgeführt. Es stellt sich die Frage, ab welchem Wert ein Ergebnis nicht mehr akzeptabel ist. Als grobe Richtlinie wurde die fünfte Nachkommastelle genommen. In der Testumgebung konnte über Parameter eingegeben werden, wie hoch die Toleranz beim Vergleich sein sollte. Wurden abweichende Werte außerhalb der Toleranz gefunden, wurden diese gezählt. Bei einem zu großen Verhältnis zwischen akzeptablen und nicht akzeptablen Werten kann die Methode zur Berechnung der Ergebniswerte nicht als Nutzbar eingestuft werden.
$C_{Toleranz}$ == Anzahl der Ergebniswerte innerhalb der Toleranz
$C_{!Toleranz}$ == Anzahl der Ergebniswerte außerhalb der Toleranz

$$\text{Akzeptabel} = \frac{C_{!Toleranz}}{C_{Toleranz}} \approx 0.0 \tag{4.2}$$

$$\text{Nicht Akzeptabel} = \frac{C_{!Toleranz}}{C_{Toleranz}} > 0.0 \tag{4.3}$$

### 4.3.1 Vergleich der Ergebnisse *NN*

OBERFLÄCHENZERLEGUNG IN DREIECKE

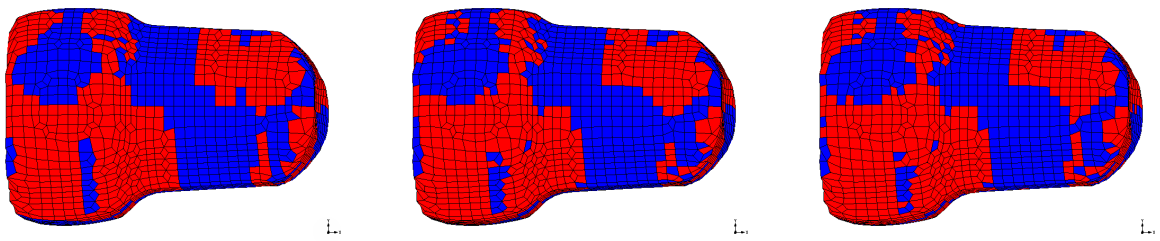


(a) Original (b) 1 NN Dreieck (c) 4 NN Dreieck

Abbildung 4.5: Visuelle Darstellung des EFFPLS Ergebnisses für Dreieckzerlegung

Bis auf kleine Unstimmigkeiten stimmen die EFFPLS Ergebnisse im Groben überein (siehe Abb.4.5). Auch das Verhältnis zwischen Werten innerhalb zu außerhalb der Toleranz bleibt genügend klein, so dass diese Methode Erfolgversprechend ist.

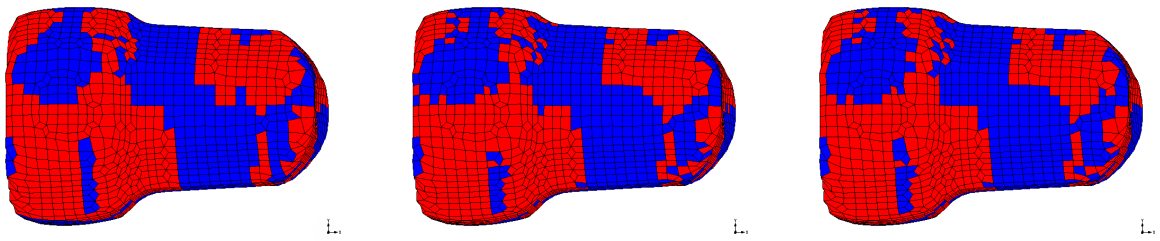
Für die Ergebniswerte für die Spannungen wurde die Unterscheidung in absoluten und maximalen Wert getroffen.



(a) Original (b) 1 NN Dreieck (c) 4 NN Dreieck

Abbildung 4.6: Visuelle Darstellung der maximalen Stress Ergebnisses für Dreieckzerlegung

In Abb.4.6 kann man erkennen, dass die Ergebniswerte im Groben, optisch übereinstimmen. Leider kann man nicht erkennen welche Werte die einzelnen Elemente haben. Die Skalierung bei den vektoriellen Werten ist nicht so fein, wie bei den nicht vektoriellen Ergebniswerten. Dies führt dazu, dass es so scheint, als wenn die Ergebnisse annehmbar sind. Vergleicht man jedoch die konkreten Werte, stellt man starke Diskrepanzen fest. Daraus folgt, dass diese Methode der Ergebniswertberechnung nicht akzeptable Ergebnisse liefert und für eine Laufzeituntersuchung nicht in Frage kommt.



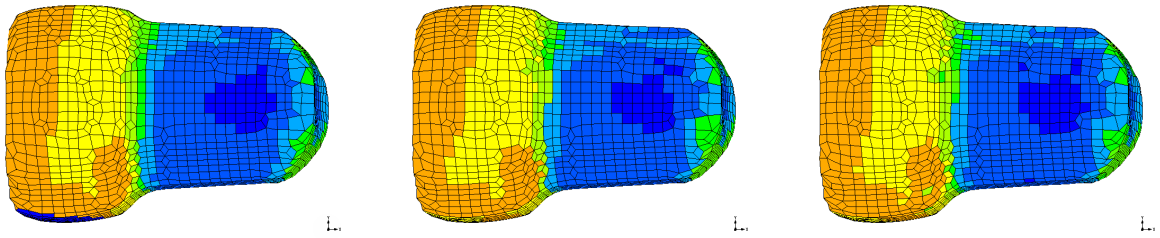
(a) Original (b) 1 NN Dreieck (c) 4 NN Dreieck

Abbildung 4.7: Visuelle Darstellung der absoluten Stress Ergebnisses für Dreieckzerlegung

INTEGRATION POINT

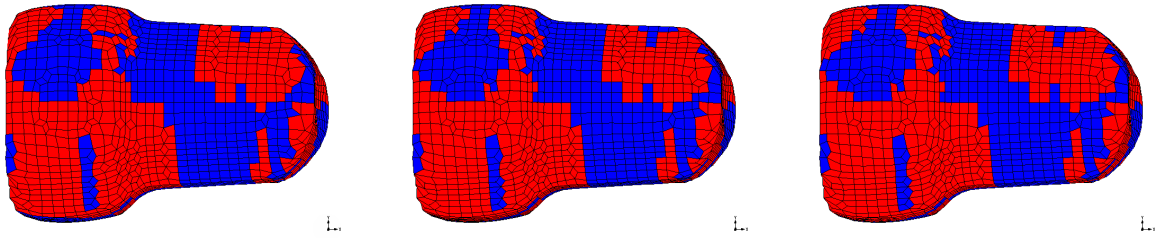
Wie man in den Abb.4.8(b) und Abb.4.8(c) erkennen kann, tritt eine Verschlechterung der EFFPLS Ergebniswerte auf, wenn die Anzahl der *NN* erhöht wird. Diese Verschlechterung liegt allerdings noch im Rahmen der Toleranzen. Somit ist auch diese Methode geeignet die Ergebniswerte zu übertragen. Bei einem Vergleich der $\tau_{xy}$-Ergebniswerte sieht das Ergebnis anders aus (siehe Abb.4.9). Wie bei der Methode mit der Oberflächenzerlegung in Dreiecke, sieht auch hier das Ergebnis mit den absoluten Werten relativ gut aus. Leider liegen auch hier zu viele Ergebniswerte außerhalb der Toleranz.

Das $\tau_{xy}$-Ergebnis verschlechtert sich deutlich, betrachtet man das Ergebnis mit den maximalen Ergebniswerten (siehe Abb.4.10). Dies ist auch nicht verwunderlich. Wenn der maximale



(a) Original (b) 1 NN Integration Point (c) 4 NN Integration Point

Abbildung 4.8: Visuelle Darstellung der absoluten EFFPLS Ergebnisses für Integration Point



(a) Original (b) 1 NN Dreieck (c) 4 NN Dreieck

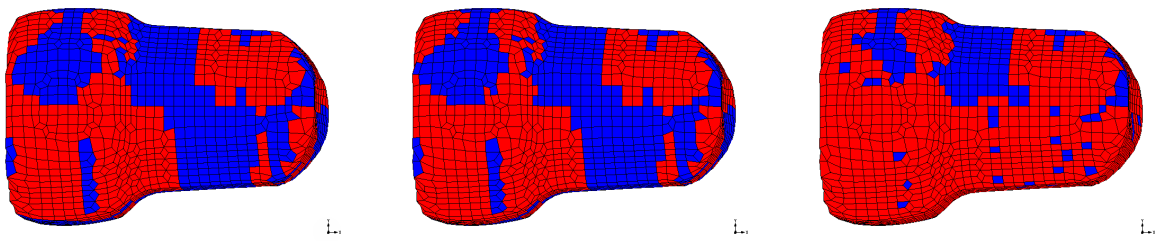
Abbildung 4.9: Visuelle Darstellung der absoluten Stress Ergebnisses für Integration Point

Wert verwendet wird, gibt es dementsprechend mehr Bereiche mit einer hohen Spannung. Dies entspricht jedoch weder den Erfahrungswerten, noch den Ergebnissen des originalen Algorithmus. Betrachtet man die Bilder 4.10(c) und 4.9(c) kann man vermuten, dass der Ansatz mit dem absoluten Ergebniswerten sinnvoller ist. Physikalisch gesehen macht es auf jeden Fall Sinn. Spannungen sind Kräfte, die in einer Fläche wirken. Es sollte also nicht der Maximalwert verwendet werden. Dieser repräsentiert die physikalischen Kräfte schlechter. Hat man eine sehr hohe Kraft in negative Richtung, darf sie nicht durch eine kleinere Kraft in positiver Richtung ersetzt werden.
Dieser Ansatz entspricht nicht dem Ergebnisbild des ursprünglichen Algorithmus. Die Anzahl der Werte außerhalb des Toleranzbereiches ist zu hoch, daher muss dieser Ansatz verworfen werden.

#### Altes Element mit Bounding Box

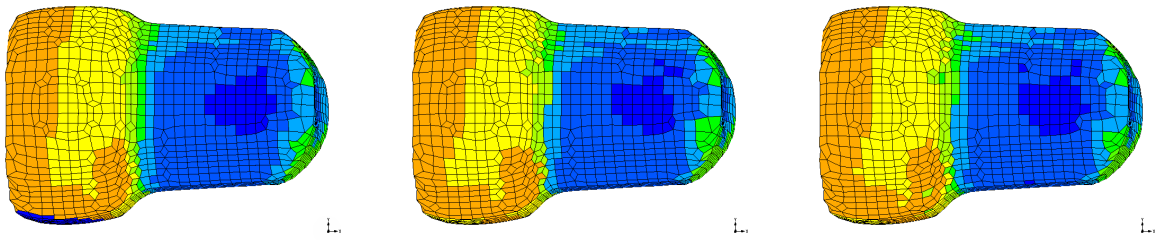
Dieser Ansatz scheint eine höhere Aussicht auf Erfolg zu haben, als der Algorithmus ohne *Bounding Box*. Hier werden auch Elemente betrachtet, die relativ groß sind. Mit dem Ansatz ohne *Bounding Box* würden diese Elemente unter Umständen unbeachtet bleiben.

Auch hier sind Verschlechterungen bei den Ergebnisbildern (siehe Abb.4.11(b) und



(a) Original (b) 1 NN Integration Point (c) 4 NN Integration Point

Abbildung 4.10: Visuelle Darstellung der maximalen Stress Ergebnisses für Integration Point



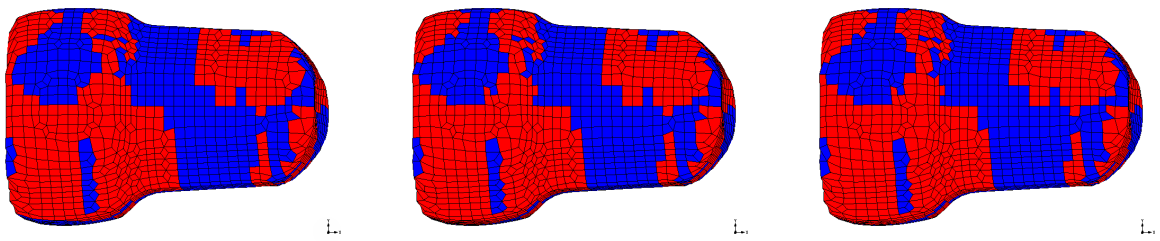
(a) Original (b) 1 NN Bounding Box (c) 4 NN Bounding Box

Abbildung 4.11: Visuelle Darstellung der EFFPLS Ergebnisses für Bounding Box

Abb.4.11(c)) zu erkennen. Diese Unterschiede liegen allerdings unterhalb der Toleranzgrenze.

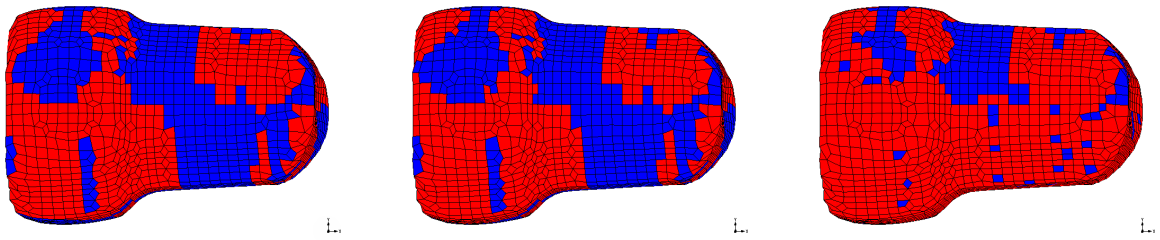
Beim Vergleich der Spannungsbilder der $\tau_{xy}$-Ergebnisse (siehe Abb.4.12(b) und Abb.4.12(c)) treten geringe Differenzen auf. Wieder kann eine Verschlechterung des Ergebnisses im Vergleich zum originalen Algorithmus anhand der Toleranzen ermittelt werden.

Es wurde bereits herausgearbeitet, dass der absolute maximale Ergebniswert sinnvoller ist, als der maximale Ergebniswert. Um dies aufzuzeigen wurden wieder die Ergebnisbilder verglichen. Es bestätigte sich die Vermutung. Das Ergebnis von vier *NN* sieht schlechter aus (siehe Abb.4.13(c)). Dies resultiert aus der Tatsache, dass auch große Elemente mit einbezogen werden. So wird ein Wert übernommen, der relativ weit entfernt liegt. Durch seine Ausdehnung wird er mit ausgewertet, da seine Entfernung zum neuen Element minimal ist. Bei dem Ansatz mit dem *Integration Point* würde dieses Element heraus fallen, da er zu weit entfernt liegt.



(a) Original (b) 1 NN Bounding Box (c) 4 NN Bounding Box

Abbildung 4.12: Visuelle Darstellung des absoluten Stress Ergebnisses für Bounding Box



(a) Original (b) 1 NN Bounding Box (c) 4 NN Bounding Box

Abbildung 4.13: Visuelle Darstellung des maximalen Stress Ergebnisses für Bounding Box

### 4.3.2 Polygonzug nach Hippmann

Da keine zufriedenstellende Version des Algorithmus gefunden werden konnte, wurde auf Laufzeittests und Ergebnisvergleiche verzichtet.

## 4.4 Bewertung und Vergleich der Ergebnisse

### 4.4.1 Geschwindigkeitsgewinn

Es konnte gezeigt werden, dass es eine Verringerung der Wartezeit für den Benutzer beim *Remapping* möglich ist. Die Parallelisierung des vorhandenen Algorithmus hat da bereits gute Vorarbeit geleistet.
Die Einführung des *BST* und die Reduzierung der Rechenschritte durch Verzicht auf Schnittvolumenberechnungen verkürzte die Wartezeit deutlich.

### 4.4.2 Ergebnisgenauigkeit

Die Durchführung der Parallelisierung des bestehenden Algorithmus führte nicht zu einer Veränderung der Ergebniswerte und entsprach damit den Vorgaben.
Leider brachten die implementierten Algorithmen nicht die erwünschten Ergebnisse in Anbetracht der Korrektheit der Ergebniswerte. Das *Nearest Neighbour*-Verfahren lieferte die nähesten Resultate bei den nicht vektoriellen Werten. Bei den vektoriellen Größen sah dies anders aus. Die numerischen Werte weichten stark von den erwarteten Werten ab.
Keine der Methoden konnte bei den vektoriellen Werten ein brauchbares Ergebnis erzielen. Das ist nicht verwunderlich, da es bei einer Werteübernahme, wie sie bei den vektoriellen Werten durchgeführt wurde, auf das Element ankommt, von dem man den Wert übernimmt. Da dieses Element auf einem andere Weg ermittelt wird, ist die Wahrscheinlichkeit, dass es sich um dasselbe Element handelt, äußerst gering.
Die Vermutung, mit mehr *NN* ein besseres Ergebnis zu erhalten, bestätigte sich nicht.
Man könnte die normalen Ergebnisse getrennt von den vektoriellen Größen dem *Remapping* unterziehen. Dies ist allerdings kein akzeptabler Weg. Die kombinierte Laufzeit, vom *Remapping* der normalen Ergebniswerte und dem der vektoriellen Werte, wäre zu lang.

# 5 Fazit und Ausblick

## 5.1 Fazit

Es ist nicht wünschenswert für vekorielle und nicht vektorielle Ergebnisgrößen unterschiedliche Methoden der Berechnung zu haben. In vielen Fällen werden beide Arten von Ergebniswerten gewünscht. Zwei Berechnungen durchzuführen würde die Wartezeit für den Benutzer wieder verlängern.
Momentan existiert keine zweite Methode, welche die Ergebniswerte für vektorielle Größen mit ähnlichen Resultaten berechnet, wie der bestehende Algorithmus. Würde man jetzt nur die vektoriellen Größen mit diesem Algorithmus berechnen und die anderen Ergebnisgrößen mit einem anderen Algorithmus, hätte man verschiedene Algorithmen zu pflegen. Dies entspricht weder den Vorgaben, noch den Vorstellungen der *Femutec GmbH*.
Die *PCM* stellte einen interessanten Ansatz dar, warf jedoch zu viele Sonderfälle auf, die in ihren Berechnungen nicht vorhersagbar waren. Für ein Kontaktmodell ist sie gut geeignet, für eine schnelle Werteübertragung dauert sie allerdings zu lange. Sollte ein Kontakt einmal übersehen werden ist dies nicht wünschenswert, kann aber kompensiert werden. Bei einer Werteübertragung sieht dies anders aus. Werden da Werte nicht übertragen oder vergessen, verfälscht sich das Ergebnis. Aufgrund der vielen Sonderfälle und der damit einhergehenden Schwierigkeit die Laufzeit zu bestimmen ist die *PCM* in diesem Fall nicht geeignet.

## 5.2 Ausblick

### 5.2.1 BST

Die Erstellung des *BST* könnte parallelisiert werden. Ob dies jedoch getan werden sollte, muss in Laufzeittests überprüft werden.
Der Overhead der Threaderstellung könnte größer sein, als der Geschwindigkeitsgewinn.

### 5.2.2 Remapping

Der nächste logische Schritt, wäre es, auf Grundlage des vorhandenen Algorithmus, einen Algorithmus zu entwickeln, den man in einen *BST* integrieren kann.
Eine weitere Möglichkeit besteht darin, einen Weg zu finden, wie man die nicht geschlossenen Polygonzüge mit angemessenen Aufwand schließen kann. Des weiteren gäbe es noch die Monte Carlo Methode. Sie wurde aus Zeitgründen nicht implementiert.

### 5.2.3 Monte Carlo

Das MonteCarlo-Verfahren stammt aus der Stochastik.
Bei diesem Ansatz wird auf dem Binären Suchbaum aufgebaut, der bereits besprochen wurde.
Es werden zufällig eine beliebige Anzahl an Punkten im Raum bestimmt. Hier wäre der Raum eine *Bounding Box* um das Schnittvolumen. Diese *Bounding Box* kann durch verschiedene Fallunterscheidungen ermittelt werden. Will man sich diese Fallunterscheidungen ersparen, nimmt man einfach die *Bounding Box*, welche beide Elemente umspannt.
Das Verhältnis zwischen Punkten, die in beiden Elementen liegen, zu den Punkten, die nicht in beiden Elementen liegen, gibt eine prozentuale Angabe des Schnittvolumens $V_s$.

$$f_x = \begin{cases} 0, \text{ wenn } x \in A \cap B \\ 1, \text{ wenn } x \notin A \cap B \end{cases} \tag{5.1}$$

$$g = \sum_{x=1}^{n} f_x, \text{ mit n = Anzahl der Punkte} \tag{5.2}$$

$$V_s = V_{BB} * \frac{g}{n}, \text{ mit } V_{BB} \text{ = Volumen der Bounding Box} \tag{5.3}$$

Diese Methode ist keine exakte Lösung. Auch nicht, wenn man jeden darstellbaren Punkt in der *Bounding Box* einbeziehen würde. Die Koordinaten sind als *double* Werte angegeben. Um jeden Punkt mit einzubeziehen, müsste man unendlich viele Schleifendurchläufe durchführen. Dies führt zu einem schlechteren Laufzeitverhalten. Mit einer steigenden Anzahl an zufälligen Punkten steigt die Genauigkeit der Abschätzung des Schnittvolumens. Hier müssten Tests zeigen, wie viele zufällige Punkte gewählt werden müssen. Die Anzahl wird abhängig von der Größe der *Bounding Box* sein, welche das Schnittvolumen einschließt (bzw. von der *Bounding Box*, welche das alte und das neue Element umfasst). Hierbei darf nicht vergessen werden, dass diese Schnittvolumenberechnung nur Sinn macht, wenn alle schneidenden alten Elemente mit einbezogen werden.
Es muss sichergestellt werden, dass die Punkte wirklich in dem Element liegen. Es genügt nicht, wenn man kontrolliert, ob der Punkt in den *Bounding Boxes* liegt. Die Methodik der

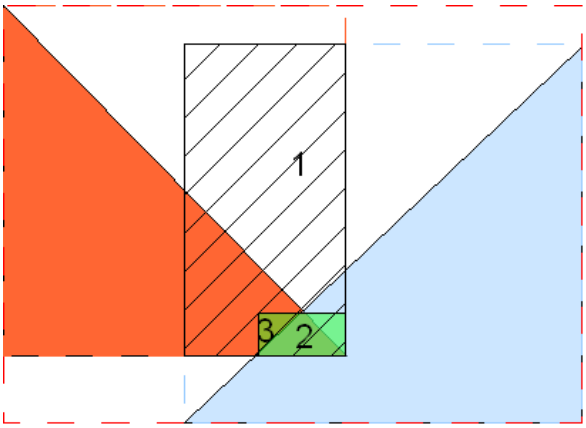



Abbildung 5.1: Problem bei Monte Carlo

Kontrolle, ob ein Punkt in einem Körper liegt, wurde bereits besprochen.
Die große, schraffierte Box in Abb.5.1 symbolisiert eine Bounding Box, die mit einer Fallunterscheidung erstellt wurde. Sie beinhaltet auf jeden Fall die Schnittmenge der beiden Dreiecke. Auch hier gilt, dass der dreidimensionale Fall nicht trivial ist und nicht so einfach ermittelt werden kann.
Punkt *1* verdeutlicht, warum es keinen Sinn macht bei der Monte-Carlo-Simulation mit dieser *Bounding Box* zu arbeiten. Der Punkt würde als in den Elementen liegend akzeptiert werden, obwohl er in keinem der beiden Elemente liegt.
Punkt *2* wäre ein Treffer, da er in beiden Elementen liegt. Der dritte Punkt hingegen liegt nur in einem Element, muss daher als außerhalb gezählt werden.
Die kleinere *Bounding Box*, die um die Schnittmenge erzeugt wurde, würde für den Fall besser passen. Sie zu ermitteln ist aber ungleich aufwendiger. Es müssen die Schnitte zwischen den Dreiecken ermittelt werden. Hierfür schneidet man jede Seite des einen Dreiecks mit jeder Seite des anderen Dreiecks und stellt dann sicher, dass ein möglicher Schnittpunkt auf den Geradenabschnitten der Dreiecke liegt.
Als Beispiel:
Die Hypotenuse des linken Dreiecks aus Abb.5.1 hat einen Schnittpunkt mit der Senkrechten Kathete des rechten Dreiecks. Dieser Schnittpunkt liegt jedoch nicht mehr im rechten Dreieck.
Dies funktioniert noch relativ einfach für Dreiecke im zweidimensionalen Raum. Befindet man sich allerdings im dreidimensionalen Raum, was bei der *FEM* häufig der Fall ist, muss man schon deutlich mehr Aufwand treiben. Ein Ansatz ist es, das Problem in ein zweidimensionales zu projiezieren. Diese Projiezierungen, Schnittmengenbildungen und wiederholten Kontrollen ob ein Punkt in einem oder beiden Körpern (altes Element, neues Element) liegt, ist sehr rechenintensiv.

Scheut man diesen Rechenaufwand nicht, erhält man eine *Bounding Box*, welche die Schnittfläche (in 2D) bzw. das Schnittvolumen (in 3D) besser repräsentiert.

Man könnte, hat man einen Punkt gefunden, der in beiden Körpern liegt, auf die Idee kommen nur noch in der näheren Nachbarschaft zu suchen. Dies verfälscht allerdings den stochastischen Ansatz der Monte Carlo Methode und würde wahrscheinlich zu falschen Ergebnissen führen.

Ob sich diese Berechnungen im Rahmen halten und eine Laufzeit für das *remapping* ergeben, die unter der des parallelisierten originalen Algorithmus liegt, müsste in Test ermittelt werden.
Die Monte Carlo Methode wird es schwer haben. Die wiederholte Ermittlung von Punkten und die Überprüfung, ob sie in einem Körper liegen, ist sehr aufwendig.

# Literaturverzeichnis


[Boa07] The OpenMP Architecture Review Board. Openmp application program interface, 2007.

[For06] High Performance Fortran Forum. High performance fortran, 2006.

[For07] Message Passing Interface Forum. Message passing interface, 2007.

[Frö05] Peter Fröhlich. *FEM - Anwendungspraxis Einstieg in die Finite Element Analyse Zweisprachige Ausgabe Deutsch/Englisch*. Friedr. Vieweg & Sohn Verlag, 2005.

[Hip04] Gerhard Hippmann. Modellierung von kontakten komplex geformter körper in der mehrkörperdynamik. Dissertation, Technischen Universität Wien, 2004.

[IEE96] IEEE. Iso/iec 9945-1:1996 [ansi/ieee std 1003.1, 1996 edition] information technology - portable operating system interface (posix®)-part 1: System application program interface (api) [c language], 1996.

[Int03] Intel. Intel® fortran programmer s reference, 2003.

[Möl97] Tomas Möller. A fast triangle-triangle intersection test. in journal of graphics tools, 1997, 1997.

[Ste92] Peter Steinke. *Finite-Element-Methode*. Cornelsen Girardet, 1992.

[Tro05] Trolltech. Qthread class reference, 2005.

# Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

| | |
|---|---|
| Hamburg, 30. Mai 2008 | |
| Ort, Datum | Unterschrift |