Technische Universität Chemnitz

Sonderforschungsbereich 393

Numerische Simulation auf massiv parallelen Rechnern

Marko Meyer

Der objektorientierte hierarchische Netzgenerator
`Netgen69-C++`

Preprint SFB393/98-09

Preprint-Reihe des Chemnitzer SFB 393

SFB393/98-09 **Juni 1998**

Inhaltsverzeichnis

Adresse des Autors:

Marko Meyer
TU Chemnitz-Zwickau
Fakultät für Informatik
D-09107 Chemnitz

marme@informatik.tu-chemnitz.de

1 Einführung

Im Rahmen der Arbeit in der damaligen DFG-Forschungsgruppe "Scientific Parallel Computing" wurde ein hierarchischer paralleler Netzgenerator für das Finite-Elemente-Programmpaket SPC-PM CFD unter dem Namen NETGEN69 entwickelt [3].

Als Programmiersprache wurde seinerzeit – wie auch in den FEM-Programmen selbst – FORTRAN benutzt. Im Rahmen des Teilprojektes B2 im Sonderforschungsbereich 393 bestand nunmehr die Aufgabe, den Netzgenerator in ein objektorientiertes Layout zu fassen und in `C++` zu implementieren.

Die Beschreibung von Ein- und Ausgabedaten kann in [3] nachgelesen werden. Die Form der Eingabedaten hat sich aus Kompatibilitätsgründen nicht geändert und wird auch in Zukunft so beibehalten werden. Auch das der Assemblierung und FEM-Rechnung zugewandte Interface wurde vorerst nicht geändert. Ein Wrapper, der für die Generierung der erwarteten Ausgabedaten aus den netzgeneratoreigenen Datenbeständen sorgt, ist derzeit in Planung. Diese Lösung ist freilich nur vorübergehender Natur; sie ermöglicht es uns, den Netzgenerator innerhalb der FEM-Bibliotheken zu testen.

In Abschnitt 2 wird die Klassenstruktur des objektorientierten Netzgenerators dargestellt. Hier wird gleichzeitig auf die einzelnen Entwicklungsentscheidungen eingegangen.

In Abschnitt 3 findet sich eine Ablaufbeschreibung der Arbeit von `Netgen69-C++`. Da an den eigentlichen Grobnetzverfeinerungsalgorithmen keine Veränderungen vorgenommen wurden, stimmt der prinzipielle Ablauf mit dem von NETGEN69 überein.

In Abschnitt 4 werden einige Vergleichsmessungen zwischen der FORTRAN- und der `C++` - Version des Netzgenerators präsentiert.

Abschnitt 5 geht auf zukünftige Entwicklungen ein.

2 Klassenstruktur von `Netgen69-C++`

Zunächst wird die Entwicklung der Klassenstruktur dargestellt, wie sie sich aus der Analyse der existierenden FEM – Bibliothek (im Einzelnen aus dem Netzgenerator) ergibt. Dann folgen einige Kommentare zu bestimmten Techniken der objektorientierten Programmierung mit `C++`, die benutzt wurden.
Jeder der grundlegenden Klassen widmet sich ein einzelner Abschnitt, der eine ausführliche Beschreibung des öffentlichen Nutzerinterfaces und eine etwas sparsamer gehaltene Darstellung des internen Ablaufs enthält.

2.1 Klassenstruktur zum Aufbau von FEM-Netzen

Ausgehend von der Struktur des ursprünglichen Netzgenerators NETGEN69 wurde eine Klassenhierarchie entwickelt, die das FEM-Problem beschreibt.

Im betrachteten 2D-Fall besteht das **FEM–Netz** aus **Flächen (Faces)**, die ihrerseits aus **Kanten (Edges)** bestehen, die von je zwei **Knoten (Nodes)** begrenzt sind und in der Regel noch einen Mittelknoten enthalten. Jedes dieser Elemente wird als eigenständige Klasse implementiert. Auf diese Weise steht an der Oberfläche des Netzgenerators nur eine Liste mit Flächen zur Verfügung, die prinzipiell durch ihre Mitglieder das gesamte Netz beschreibt[1]. Die genannten Klassen sollen hier zukünftig zusammenfassend als **FEM-Klassen**, und die daraus entstehenden Instanzen als **FEM-Objekte** bezeichnet werden.

Es sollen sowohl Dreiecks- als auch Vierecksflächen betrachtet werden. Hierfür wurden die Klassen `TFace` (Triangular Face) und `RFace` (Rectangular Face) definiert, die von einer gemeinsamen abstrakten Basisklasse `Face` abgeleitet sind. Diese Spezialisierung macht sich nicht nur deshalb erforderlich, weil die verschiedenen Flächenformen unterschiedliche Anzahlen von Kanten-Objekten enthalten müssen; dies könnte auch über ein entsprechend dynamisch gehaltenes Array erreicht werden. Vielmehr kommt es darauf an, daß es auch unterschiedliche Methoden gibt. Nur bei Vierecksflächen tritt zum Beispiel der Bedarf nach einem Mittelpunktsknoten und einer Methode, diesen aus den Kanteninformationen zu generieren, auf. Weiterhin unterscheiden sich die Methoden zum Erzeugen der Teilflächen bei der Netzverfeinerung recht deutlich.

Außer bei den Flächen, gibt es auch bei den Kanten und Knoten verschiedene Typen. So unterscheidet die bisherige Implementierung zum Beispiel bei den Knoten zwischen Rand-, Crosspoint- und inneren Knoten, bei den Kanten immerhin zwischen Rand-, Koppel- und inneren Kanten.
Da Rand- und andere Kanten sich dadurch unterscheiden, ob Randbedingungen vorliegen oder nicht, wurde hier eine weitere Vererbungsstufe eingefügt. Außer der allgemeinen Kantenklasse `Edge` gibt es auch die Randkantenklasse `Boundary`. Letztere beinhaltet dann die entsprechenden Randbedingungswerte.
Die Unterschiede zwischen den verschiedenen Knotentypen rechtfertigen noch keine Hierarchiebildung, sodaß hier darauf verzichtet wurde. Die unterschiedlichen Typen lassen sich problemlos durch ein entsprechendes Flag-Feld darstellen, wie das auch bislang schon geschieht. Insbesondere fiel bei dieser Entscheidung ins Gewicht, daß sich die verschiedenen Knotentypen im Sinne der Verarbeitung im Netzgenerator (also während der Netzverfeinerung) nicht unterschiedlich verhalten.

[1] Momentan wird neben der Flächenliste auch eine globale Kanten- und Knotenliste verwaltet. Dies dient zur Zeit zur bequemen und schnellen Übertragung der Daten in die alten Strukturen, die in den nachfolgenden Arbeitsschritten der FEM-Rechnung benötigt werden. Das Eintragen neu entstandener Objekte in diese globalen Listen verursacht natürlich einen Overhead, der bei konsequenter Fortführung der objektorientierten Implementierung vermeidbar ist.

Inwieweit hier zukünftig neue Unterschiede zwischen den verschiedenen Knotentypen entstehen, wenn die objektorientierte Implementierung über den Zuständigkeitsbereich des Netzgenerators hinaus ausgedehnt wird, kann momentan noch nicht abgesehen werden. Es könnte sich jedoch erforderlich machen, auch hier – ähnlich wie bei den Flächen und Kanten – eine Klassenhierarchie zu bilden.

2.2 Die Klassen von `Netgen69-C++`

2.2.1 Die Klasse `Node`

Um einen Knoten im FEM-Netz darzustellen, wurde die Klasse `Node` entwickelt. Prinzipiell wurde dabei die Struktur des Knoten-Arrays im NETGEN69 beibehalten. Die Klasse ist in `include/node.h` definiert und wird im Quelltextfile `src/node.cc` implementiert.

Im `protected` - Bereich der Klasse befinden sich die folgenden Datenmember:

- `int Number;` – die Knoten-Nummer.
  Diese Angabe wird nur aus historischen Gründen gespeichert. Da die Hierarchieinformation im `Netgen69-C++` nicht mehr über eine spezielle Numerierung dargestellt wird, ist die Vergabe einer (eindeutigen) Knotennummer nicht mehr erforderlich.

- `double X, Y, Z;` – die Position des Knotens.
  Hierbei wurde bereits an eine Erweiterung für den 3D-Netzgenerator gedacht, und eine dritte Koordinate z eingeführt. Sie wird zur Zeit allerdings noch nicht benutzt.

- `int Kind;` – der Knotenanzeiger für die Strömungssimulation.

- `int Level;` – das Verfeinerungslevel, in dem der Knoten erzeugt wurde.

- `PF_t Processor_Flag;` – ein Flag, das angibt, auf welchen Prozessoren dieser Knoten vorhanden ist. Damit können zum späteren Zeitpunkt Rückschlüsse auf die erforderlichen Kommunikationen zum Austausch der Ketten gewonnen werden. Der Datentyp `PF_t` ist in `node.h` als `unsigned long long` definiert. Die Überleitung in ein `bitset<>` ist geplant.

Neu gegenüber der Knoteninformation im NETGEN69 sind die folgenden Datenmember: `Z`, `Level` und `Processor_Flag`.

Im `public` - Bereich der Klasse befinden sich die folgenden Methoden:

Konstruktion, Initialisierung , Destruktion

- `Node();`
  Der Default-Konstruktor. Er initialisiert alle Member mit -1; abgesehen von `Processor_Flag`, das mit 0 initialisiert wird.

- `Node(int Num, double PX, double PY, double PZ, int K, int L, PF_t PF);`

  Alle Datenmember werden mit den entsprechenden Übergabewerten initialisiert.

- `Node(const Node &N);`
  Der Copy-Konstruktor.

- `Node &operator = (const Node &N);`
  Der Copy-Zuweisungsoperator. Alle Datenmember von N werden kopiert, eine Referenz auf `this` wird zurückgegeben.

- `virtual int Init(Tokenizer &T, int L);`
  Diese Initialisierungsfunktion nimmt eine Referenz auf einen `Tokenizer` als Übergabe, neben der Information über das Einordnungslevel des Knotens. Sie ist für die Anfangsinitialisierung eines Knotens aus der Netzwerkbeschreibungsdatei gedacht. (s. 2.3.3 und 3.1)

- `virtual ~Node();`
  Der Destruktor.

Zugriff auf Datenmember

- `virtual int operator () () const;`
  Gibt die Knotennummer zurück.

- `PF_t get_pflag() const;`
  Liefert den Inhalt des Datenmembers `Processor_Flag` zurück.

- `void pflag(PF_t p);`
  Erlaubt das Setzen des `Processor_Flag`s von außen. Das Ergebnis ist eine ODER-Verknüpfung des aktuellen `Processor_Flag`s mit dem Übergabeparameter `p`.

- `void all_pflag(PF_t p);`
  Hier wird der Übergabeparameter direkt in den Datenmember `Processor_Flag` übertragen. Die dort gespeicherte Information geht damit verloren.

- `double x() const;`
  Gibt den Wert des Datenmembers `X` zurück, also die x-Koordinate des Knotens.
- `double y() const;`
  Gibt den Wert des Datenmembers `Y` zurück, also die y-Koordinate des Knotens.
- `double z() const;`
  Gibt den Wert des Datenmembers `Z` zurück, also die z-Koordinate des Knotens.
- `double kind() const;`
  Gibt den Wert des Datenmembers `Kind` zurück, also den Strömungsanzeiger des Knotens.
- `double level() const;`
  Gibt den Wert des Datenmembers `Level` zurück, also das Entstehungslevel des Knotens.

Verarbeitung

- `Node *New_Node(const Proxy<Node> &Other_Node, double Grad) const;`
  Konstruiert einen neuen Knoten als Geradenmittelpunkt entsprechend der `NeuNode` – Subroutine aus NETGEN69.
- ```
  Node *New_Node(const Proxy<Node> &Other_Node, double MiddleX,
                 double MiddleY, double Radius) const;
  ```
  Konstruiert einen neuen Knoten als Punkt auf einem Kreisbogen gemäß der `LotKreis` – Subroutine aus NETGEN69.
- ```
  Node *New_Node(const Proxy<Node> &ON1, const Proxy<Node> &ON2,
                 double fak1, double fak2, double fak3) const;
  ```
  Konstruiert einen neuen Knoten als Punkt auf einer Parabel nach der `Parabel` – Subroutine aus NETGEN69.

Visualisierung/Debugging

- `virtual ostream &Show(ostream &os) const;`
  Gibt auf dem `ostream os` den aktuellen Zustand des Objektes aus, also alle Datenmember.
- `virtual ostream &Show_One_Line(ostream &os) const;`
  Wie `Show`, nur wird die auszugebende Information hier auf eine Zeile verdichtet.

2.2.2 Die Klasse `Edge` und Erben

Die Klassen `Edge` und `Boundary` verwalten die Kanten des FEM-Netzes. Sie sind in `include/edge.h` deklariert und in src/edge.cc implementiert. Die Klasse `Edge` ist von `RefT` *public* abgeleitet und kann damit im Sinne des Referenzcountings eingesetzt werden.

Die Klasse `Edge` enthält im *protected* - Bereich folgende Datenmember:

- `int Number;` – Die Nummer der Kante. Wie bei den Knoten, wird diese Information momentan nur noch beim Einlesen aus der Netzbeschreibungsdatei eingetragen.
- `Proxy<Node> Start;` – Eine Referenz auf den Startknoten.
- `Proxy<Node> End;` – Eine Referenz auf den Endknoten.
- `Proxy<Node> Middle;` – Eine Referenz auf den Mittelknoten.
- `int Shape;` – Die Kantenform. Hierzu wurden die folgenden Konstanten definiert:
  - `const int E_SHAPE_LINE = 0;` – Lineare Kante.
  - `const int E_SHAPE_CIRCLE = 1;` – Kreisförmige Kante.
  - `const int E_SHAPE_PARABOLA = 2;` – Parabolische Kante.
- `int Type;` – Kantenart. Folgende Konstanten sind dafür möglich:
  - `const int E_ORD = 0;` – Innere Kante.
  - `const int E_BOUND = 1;` – Randkante.
  - `const int E_BOUND_D = 3;` – Randkante Dirichlet.
  - `const int E_BOUND_N = 5;` – Randkante Neumann.
  - `const int E_COUPL = 8;` – Koppelkante.
- `Proxy<Edge> *L_Child;` – Beim Splitten der Kante wird hier der linke Tochterast untergebracht.
- `Proxy<Edge> *R_Child;` – Beim Splitten der Kante wird hier der rechte Tochterast untergebracht.
- `double Grad;` – Gradient für Neuknoten-Berechnung.

In der Klasse `Boundary`, die *public* von `Edge` erbt, sind außerdem noch folgende Datenmember im *protected* - Bereich enthalten:

- `BC_t BC_Start;` – Randbedingungen für den Startknoten.

- `BC_t BC_Middle;` – Randbedingungen für den Mittelknoten.
- `BC_t BC_End;` – Randbedingungen für den Endknoten.
- `int BC_Code;` – Randbedingungscode.

Der Datentyp `BC_t` ist ein `vector<double>`.

Neben den Datenmembern gibt es in den beiden Klassen jeweils noch Memberfunktionen in den *protected* - Bereichen. Diese sind:

- `inline const Proxy<Edge> *find_adjacent_edge(const Proxy<Edge> &E1, int &dir) const;`
  Diese Methode gibt bei bereits gesplitteten Kanten die Hälfte der aktuellen Kante zurück, die direkt an die gegebene Kante *E1* angrenzt. Der Parameter *dir* dient zur Rückgabe der Suchrichtung. Ist *dir* gleich 1, so wurde in `End` ein gemeinsamer Knoten gefunden, ansonsten ist der gemeinsame Knoten `Start`.

- `void Dist_BC(const EdgeProxy &EP, int);` (nur in `Boundary`)
  die zur Verteilung der Randbedingungen beim Splitting der Kante aufgerufen wird. Die Verteilung geschieht hierbei analog zu bisherigen Algorithmen in NETGEN69.

Folgende Memberfunktionen stehen im *public* - Bereich von `Edge` bzw. `Boundary` zur Verfügung[2]:

Konstruktion, Initialisierung , Destruktion

- `Edge();`
  `Boundary();` Der Default-Konstruktor. Initialisiert die Kante als `E_ORD`, die Knoten werden jeweils default-konstruiert.

- ```
  Edge(int Num, const NodeProxy &SN, const NodeProxy &EN,
       const NodeProxy &MN, int Sh, int T, double G = 0);
  Boundary(int Num, const NodeProxy &SN, const NodeProxy &EN,
       const NodeProxy &MN, int Sh, int T);
  ```
  Alle Member werden entsprechend der Übergabewerte initialisiert. `Grad` wird mittels der in `include/meshalgo.h` definierten Funktion `gradfak`, die algorithmisch der entsprechenden Funktion des NETGEN69 entspricht, aus den Übergabewerten `SN`, `EN` und `MN` berechnet.

[2]Methoden, die nicht anders gekennzeichnet sind, stehen in beiden Klassen zur Verfügung.
  ```

- `Boundary(int Num, NodeProxy &SN, NodeProxy &EN, NodeProxy &MN,`
  `         int Sh, int T, const BC_t &RBS, const BC_t &RBM,`
  `         const BC_t &RBE, int RBC);`
  Mit diesem Konstruktor können zusätzlich auch die Randbedingungen übergeben werden.

- `Edge(const Edge &E);`
  `Boundary(const Boundary &B);`
  Der Copy-Konstruktor. Auch hier wird `Grad` wie o.g. aus den entsprechenden Knoten der übergebenen Kante berechnet.

- `Edge &operator = (const Edge &E);`
  `Boundary &operator = (const Boundary &B);`
  Der Copy-Zuweisungsoperator. Seine Wirkungsweise entspricht dem Copy-Konstruktor.

- `int Init(MeshFile &MF, Tokenizer &T, vector<NodeProxy> &N, int Level);`
  Diese Methode initialisiert die Kante aus dem `MeshFile MF` (s. 2.3.3). In `N` wird der aktuelle "globale" Knotenvektor übergeben. Damit wird sichergestellt, daß keine Knoten doppelt existieren und neu initialisierte Knoten sofort in den Knotenvektor eingetragen werden.

- `int Boundary::Init(MeshFile &MF, Tokenizer &T, vector<NodeProxy> &N,`
  `         vector< vector<char> > &BC, int Level);`
  Diese Methode wird in der abgeleiteten Klasse `Boundary` implementiert. Sie entspricht der obigen `Init`–Methode und belegt zusätzlich noch die Randbedingungsvektoren vor.

Zugriff auf Datenmember

- `virtual int operator() () const;`
  Gibt die Kantennummer zurück.

- `double grad() const;`
  `void grad(double g);`
  Liefert den aktuellen Wert des Gradientenfaktors `Grad` zurück, bzw. setzt diesen auf g.

- `int shape() const;`
  `void shape(int s);`
  Gibt die Kantenform, wie gespeichert im Datenmember `Shape`, zurück, oder erlaubt diesen Datenmember auf den Wert von s zu setzen.

- `int type() const;`
  `void type(int t);`
  Gibt den Typ der Kante zurück bzw. setzt den Typ auf t.

- `bool split() const;`
  Liefert dann `true`, wenn die Kante geteilt ist; also Tochteräste besitzt.

- `const Proxy<Edge> *l_child() const;`
  `void l_child(const Proxy<Edge> &L);`
  Gibt die "linke" Tochterkante zurück, bzw. setzt diese gleich der übergebenen Kante L.

- `const Proxy<Edge> *r_child() const;`
  `void r_child(const Proxy<Edge> &R);`
  Gibt die "rechte" Tochterkante zurück, bzw. setzt diese gleich der übergebenen Kante R.

- `const Proxy<Node> &start() const;`
  `const Proxy<Node> &middle() const;`
  `const Proxy<Node> &end() const;`
  Liefern die entsprechenden Knotenobjekte zurück.

- `virtual void bc_start(const BC_t &);`
  `virtual void bc_middle(const BC_t &);`
  `virtual void bc_end(const BC_t &);`
  `virtual void bc_code(int);`
  Hiermit können die aus dem Methodennamen hervorgehenden Parameter gesetzt werden. Obwohl die Methoden aus Gründen der Polymorphie auch in `Edge` definiert sind[3], bringen nur die Aufrufe für `Boundary` sinnvolle Resultate.

- `virtual const BC_t &get_bc_start() const;`
  `virtual const BC_t &get_bc_middle() const;`
  `virtual const BC_t &get_bc_end() const;`
  `virtual int get_bc_code() const;`
  Liefert die entsprechenden Parameter zurück. Nur beim Aufruf für ein `Boundary` – Objekt werden hier sinnvolle Ergebnisse erzeugt.

Verarbeitung

- `void set_node_proc(long long p);`
  Diese Methode addiert die in p angegebene Prozessor-Maske zu den Prozessormas-

[3] Allerdings als "nop".

ken der kantenzugehörigen Knoten. Damit wird die Kennzeichnung der Prozessorzugehörigkeit der Knoten ermöglicht, wie in 3.1 dargestellt.

- `virtual Edge *Create_Edge(int Num, const Proxy<Node> &SN,`
  `const Proxy<Node> &EN, const Proxy<Node> &MN,`
  `int Sh, int T) const;`
  Erzeugt eine neue Kante, die die Knoten *SN*, *MN* und *EN* umfaßt, vom Typ *T* ist und die Kantenform *Sh* hat. Wenn die Methode für eine `Boundary` – Kante aufgerufen wird, so wird damit ein neues Objekt der `Boundary` – Klasse erzeugt; ansonsten eine neue `Edge`.

- `virtual void New_Edge(const Proxy<Edge> &OE, double g,`
  `vector<Proxy<Node> > &NL, Proxy<Edge> &NE1,`
  `Proxy<Edge> &NE2, Proxy<Edge> &NE3) const;`
  Dies ist eine generalisierte Methode, die eine neue Kante als Verbindung zwischen Mittelknoten der aufrufenden Kante und Mittelknoten von *OE* erzeugt. Dabei entsteht automatisch eine Dreiecksfläche aus der neuen Kante und den beiden anliegenden Tochterkanten. Die entsprechenden drei Kanten werden zurückgeliefert. (Diese Methode wird "automatisch" nur für Dreiecksflächen aufgerufen. Für Vierecksflächen ist eine Methode geplant, die anstelle des Parameters *OE* den Mittelknoten des alten Vierecks als Argument erhält.)

- `virtual void Split(vector<Proxy<Node> > &NodeL);`
  Die Methode zur Kantenteilung. Der Split-Algorithmus wird ausführlich in 3.2.1 beschrieben. Die entstehenden neuen Mittelknoten werden automatisch in *NodeL* eingetragen.

Visualisierung/Debugging

- `virtual void Show(ostream &os) const;`
  Gibt komplette Information über das vorliegende `Edge`– (bzw. `Boundary`–) Objekt auf *os* aus. Die Methode ruft lediglich `Show_Generic` und `Show_Children` auf. Ist das Objekt ein `Boundary`, dann wird zusätzlich noch `Show_BC` aufgerufen.

- `virtual void Show_Line(ostream &os) const;`
  Gibt die wichtigsten Informationen[4] über das `Edge`– (bzw. `Boundary`–) Objekt auf *os* in einer Zeile aus.

- `virtual void Show_Generic(ostream &os) const;`
  Nach einem Aufruf von `Show_Line` wird hier die Form und der Typ der Kante auf *os* ausgegeben.

[4]Zur Zeit sind dies: die Nummer der Kante und die Nummern der zugehörigen Knoten; in künftigen Versionen werden stattdessen wahrscheinlich eher die Knotenkoordinaten ausgegeben werden.

- `virtual void Show_Children(ostream &os) const;`
  Ruft die Methode `Show` für die Datenmember `L_Child` und `R_Child` auf, sofern diese nicht-Null sind, und damit Tochterkanten existieren. Damit wird sozusagen ein rekursiver Aufruf von `Show` – Methoden erzeugt.

- `virtual void Show_BC(ostream &os) const;`
  Diese Methode ist nur für `Boundary` – Objekte implementiert und gibt die Randbedingungen auf *os* aus.

Neben diesen Methoden existiert noch der Funktor `Show_Edge`. Dieser ist in `include/edge.h` implementiert. Er kann im Zusammenhang mit dem globalen Kantenvektor eingesetzt werden, um für jede der dort gespeicherten Kanten die Methode `Show` aufzurufen. Ein entsprechender Aufruf könnte zum Beispiel so aussehen:

```
vector<Edge> VE;

// ... Kanten erzeugen und in VE speichern ...

// Alle Kantendaten auf cout ausgeben:
for_each(VE.begin(),VE.end(),bind2nd(Show_Edge(),&cout));
```

Zur Benutzung von Funktoren und STL-Vektoren siehe insbesondere [6] und [5].

2.2.3 Die Klasse `Face` und Erben

Die Klasse `Face` ist eine abstrakte Basisklasse. Sie wird bislang durch zwei Klassen spezialisiert: `TFace` (für Dreicksflächen) und `RFace` (für Vierecksflächen). Die Faceklassen sind in `include/face.h` deklariert und werden in `src/face.cc` implementiert.

Folgende Datenmember gibt es im *protected* – Bereich von `Face`:

- `int Number;` – Die Flächennummer. Wie auch bei `Edge` und `Node` wird diese Information nur noch vorübergehend mitgeführt.

- `int MatID;` – Materialbereichsnummer der Fläche.

Im *protected* – Bereich von `TFace` gibt es außerdem folgenden Datenmember:

- `Proxy<Edge> EVec[3];` – Die zugehörigen Kanten.

Im *protected* – Bereich von `RFace` gibt es hingegen folgende Datenmember:

- `Proxy<Edge> EVec[4];` – Die zugehörigen Kanten.
- `Proxy<Node> CentreNode;` – Der Mittelknoten der Vierecksfläche.

In `RFace` gibt es außerdem folgende Methode im *protected* – Bereich:

- `Node *Calc_Center(const EdgeProxy &EE1, const EdgeProxy &EE2,`
  `const EdgeProxy &EE3, const EdgeProxy &EE4) const;`
  Berechnung des Mittelknotens nach dem isoparametrischen Konzept (übernommen aus NETGEN69).

Folgende Memberfunktionen stehen im *public* – Bereich der Faceklassen zur Verfügung:

Konstruktion, Initialisierung , Destruktion

- `Face(int vsize = 0);`
  `TFace();`
  `RFace();`
  Der Default-Konstruktor. Im Falle der Klasse `Face` wird mit *vsize* die Größe des Parameters `EVec` bestimmt; also die Anzahl der zugehörigen Kanten.

- `Face(int Num, int MID, const vector<Proxy<Edge> > &EV);`
  `TFace(int Num, int MID, const Proxy<Edge> &E1, const Proxy<Edge> &E2,`
  `const Proxy<Edge> &E3);`
  `RFace(int Num, int MID, const Proxy<Edge> &E1, const Proxy<Edge> &E2,`
  `const Proxy<Edge> &E3, const Proxy<Edge> &E4);`
  Die entsprechenden Datenmember werden mit den übergebenen Parametern vorbelegt.

- `Face(const Face &F);`
  `TFace(const TFace &T);`
  `RFace(const RFace &R);`
  Der Copy-Konstruktor.

- `Face & operator = (const Face &F);`
  `TFace & operator = (const TFace &F);`
  `RFace & operator = (const RFace &F);`
  Der Copy-Zuweisungsoperator.

- ```
  int Init(MeshFile &MF, Tokenizer &T, vector<Proxy<Edge> > &E,
        vector<Proxy<Node> > &N, int Level);
  ```
  Initialisiert das Flächenobjekt aus dem Netzdatenfile. Rekursiv werden hier die Initialisierungsmethoden der zugehörigen Kanten und der ihnen angehörenden Knoten gestartet.

- ```
  virtual ~Face();
  virtual ~TFace();
  vortual ~RFace();
  ```
  Der Destruktor.

- ```
  const Face *New_Face(const Proxy<Edge> &E1, const Proxy<Edge> &E2,
                       const Proxy<Edge> &E3); const
  ```
  (nur bei `TFace`)[5]
  Erzeugt ein neues Flächenobjekt aus den übergebenen Kanten.

Zugriff auf Datenmember

- `int operator () () const;`
  Gibt die Flächennummer zurück.

- `const Proxy<Edge> & operator [] (int elem);` Gibt eine const-Referenz auf die enthaltene Kante *elem* zurück. Ist *elem* größer als die Anzahl verfügbarer Kanten, so wird eine Referenz auf ein leeres Proxy-Objekt zurückgeliefert.

Verarbeitung

- `void set_node_proc(long long p);`
  Ruft für alle enthaltenen Kanten die entsprechende Methode `set_node_proc` auf. Damit wird die Prozessorzugehörigkeit jedes Knotens während der Initialisierung eingetragen.

- `virtual void Refine(vector<Proxy<Node> > &NL, Proxy<Face> *NFA);`
  Der Netzverfeinerungsprozeß, der hiermit gestartet wird, wird in 3.2 erläutert.

[5] Die entsprechende Methode ist auch für `RFace` geplant.
  ```

Visualisierung/Debugging

- `virtual ostream &Show(ostream &os) const;`
  Hiermit werden die Datenmember der Fläche auf dem Stream *os* angezeigt. Außerdem werden auch durch rekursiven Aufruf der entsprechenden `Show` – Methoden von Kanten und Knoten deren interne Parameter angezeigt.

2.2.4 Die Klasse `Mesh` und Erben

Die Klasse `Mesh` ist eine abstrakte Basisklasse zur Darstellung eines FEM-Netzes und der in ihm benötigten Funktionalität. Hiervon werden zwei Klassen abgeleitet: `CoarseMesh` und `DistMesh`. Die Klasse `CoarseMesh` war ursprünglich zur Darstellung des Grobnetzes gedacht. Der Verfeinerungsprozeß sollte dann implizit durch Übergabe des entsprechenden `CoarseMesh` – Objektes beim Anlegen eines `DistMesh` – Objektes angestoßen werden. Diese Art der Verarbeitung wurde jedoch mittlerweile verworfen. Dennoch besteht die `Mesh` – Vererbungshierarchie weiterhin, da noch weitere Untersuchungen an dieser Stelle – zum Beispiel auch im Hinblick auf die Verarbeitung des FEM-Netzes im Anschluß an die Netzgenerierung – erforderlich sind.

Im Rahmen dieses Dokuments wird jedoch nur die Klasse `DistMesh` beschrieben.

Die Klassen `Mesh`, `CoarseMesh` und `DistMesh` sind in `include/mesh.h` deklariert und teilweise implementiert; die Klasse `DistMesh` ist in `src/mesh.cc` implementiert.

Die Klasse `DistMesh` erbt von `Mesh` folgende Datenmember im *protected* – Bereich:

- `int my_procid;` – Der Name des eigenen Prozessors.
- `int my_nprocs;` – Die Anzahl der Prozessoren.
- `Tokenizer T;` – Eine Tokenizer – Instanz für das Parsen der Grobnetzbeschreibungsdatei.
- `int simtype;` – Art der Simulation (entspricht dem früheren Parameter `firesim`). Gültige Werte:
  - `const int SIM_SY = 0;`
  - `const int SIM_FIRE = 1;`
- `int facetype;` – Flächentyp. Gültige Werte:
  - `const int FACE_TRIANGLE = 0;` – Dreiecksflächen.

- `const int FACE_RECTANGLE = 1;` – Vierecksflächen.

- `int version;` – Version mit oder ohne Materialbereiche.
- `int dg_freedom;` – Freiheitsgrade der Randbedingungen.
- `int n_e_dir;` – Anzahl von Dirichlet-Randbedingungen.
- `int n_e_neum;` – Anzahl von Neumann-Randbedingungen.
- `vector<Proxy<Node> > Nodes;` – "globaler" Knotenvektor.
- `vector<Proxy<Edge> > Edges;` – "globaler" Kantenvektor.
- `vector<Proxy<Face> > Faces;` – "globaler" Flächenvektor.
- `vector<Proxy<MatArea> > MatAreas;` – "globaler" Vektor von Materialbereichen.

Außerdem enthält `DistMesh` eigene Datenmember im *protected* – Bereich:

- `vector< char > file;` – Der Inhalt der Grobnetzdatei.
- `int own_share;` – Eigenanteil an der Gesamtfläche des FEM-Gebietes.
- `int own_start;` – Start des Eigenanteils. (Eine Flächennummer.)

Im *protected* – Bereich von `DistMesh` befinden sich außerdem einige Memberfunktionen:

- ```
  void calc_own_share(int num_faces, int proc_num, int &start,
                      int &share);
  ```
  Die Funktion zur Berechnung des Eigenanteils am Gesamt-FEM-Gebiet.
- `void Read_Own_Vectors(MeshFile &MF, int facetype);`
  Eine Funktion zum Einlesen der Daten aus der Grobnetzdatei.

Folgende Memberfunktionen werden von `DistMesh` im *public* – Bereich zur freien Benutzung angeboten:

Konstruktion, Initialisierung , Destruktion

- `DistMesh(int nprocs, int me);`
  Dieser Konstruktor erwartet als Übergabeparameter die Anzahl der insgesamt für die FEM-Rechnung zur Verfügung stehenden Prozessoren in *nprocs* und den Namen (die Nummer) des eigenen Prozessors in *me*.
  ```

- `DistMesh(const DistMesh &D);`
  Der Copy-Konstruktor.

- `virtual ~DistMesh();`
  Der Destruktor.

Zugriff auf Datenmember

Zur Zeit noch nicht implementiert.

Verarbeitung

- `int Read(const vector<char> &f, int facetype);`
  Die Methode zum Einlesen der Netzdaten aus einem gegebenen Vektor *f*. Dieser muß zuvor vom Nutzer mit den Daten der Grobnetzdatei gefüllt werden. Das Argument *facetype* spezifiziert den Typ des anzulegenden Netzes. Da zur Zeit die Vierecksflächenklasse `RFace` noch nicht vollständig implementiert ist, wird vom Parameter `FACE_RECTANGLE` und den entsprechenden Grobnetzdateien abgeraten.

- `int Refine(int levels);`
  Startet den Netzverfeinerungsprozeß. Das Argument *levels* gibt dabei das angestrebte Verfeinerungslevel an.

- `void Create_Proc_Flag(class MeshFile &MF, int facetype);` Bestimmt für jeden Knoten die Prozessorzugehörigkeit und setzt die entsprechenden Flags.

Visualisierung/Debugging

- `int Show(ostream &os = cerr);`
  Startet die Ausgabe der Netzparameter auf dem Ausgabe-Stream *os*. Per default ist dies die Standard-Fehlerausgabe `cerr`. Hierin wird für alle Flächen die `Show` – Methode aufgerufen, die dann ihrerseits rekursiv für die Ausgabe der Kanten- und Knotendaten sorgt.

2.3 Objektorientierte Konstrukte zur Unterstützung der Verarbeitung

Hier soll auf Techniken und Klassen eingegangen werden, die die eigentliche FEM-Rechnung nur unterstützend begleiten bzw. Grundlage ihrer Implementierung sind. Diese Konstrukte sind nicht nebensächlich, auch wenn sie während des größten Teils der Verarbeitung nicht benötigt werden oder durch dem Benutzer dieses Netzgenerators näher liegende Klassen verdeckt werden. Dazu zählen in der vorliegenden Implementierung:

- `Proxy` und `RefT` – Proxyklassen und Referenzzähler.
- `MeshFile` – Ein objektorientiertes Abbild der Grobnetzbeschreibungsdatei.
- `Tokenizer` – Ein einfacher Parser zum Einlesen der Grobnetzbeschreibungsdatei.

2.3.1 `Proxy` und `RefT` – Proxyklassen und Referenzzähler

Wie in 2.1 erwähnt, werden die Node-, Edge- und Face-Objekte in "globalen" Listen gespeichert, um eine schnelle Übertragung der Informationen in die für die weitere Verarbeitung vorerst erforderlichen einfachen Datenstrukturen zu ermöglichen. Gleichzeitig enthält jedes Edge-Objekt Verweise auf die ihm zugehörigen Node-Objekte.

Durch diese Form der Speicherung entsteht ein Synchronisierungsproblem. Um sicherzustellen, daß jeweils das **selbe** Objekt im Face bzw. Edge und auch in den "globalen" Listen referenziert wird, kann man anstelle der Objektkopie einen Pointer auf das Objekt speichern. Jedoch ergibt sich dann ein Problem, wenn Objekte gelöscht werden, die ihrerseits Objekte referenzieren. Insbesondere ist dies der Fall, wenn ein Objekt von mehreren Objekten referenziert wird, wie dies zum Beispiel bei den Kanten angrenzender Flächen auftritt. (siehe Abbildung 1)

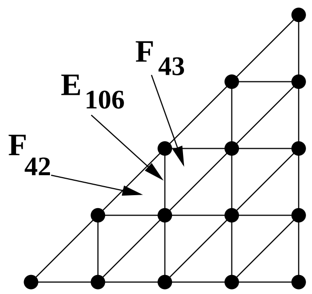

Abbildung 1: Zwei Flächen referenzieren eine gemeinsame Kante.

Das Löschen einer Fläche würde dann zum Verlust der Kante führen. Eine kurzsichtige Abhilfe wäre hier die Regelung, nur der globalen Liste die Befugnis zum Löschen der von ihr referenzierten Objekte zu geben. Für zukünftige Entwicklungen, die ohne die zusätzliche Speicherung der Objekte in derartigen Listen auskommen sollen, wäre dies jedoch ein Entwicklungshindernis. Abbildung 2 zeigt einen exemplarischen "Footprint" einer Kante in `Netgen69-C++`. Man erkennt deutlich, daß dasselbe Objekte von mehreren (im vorliegenden Beispiel 4) anderen Objekten referenziert wird.

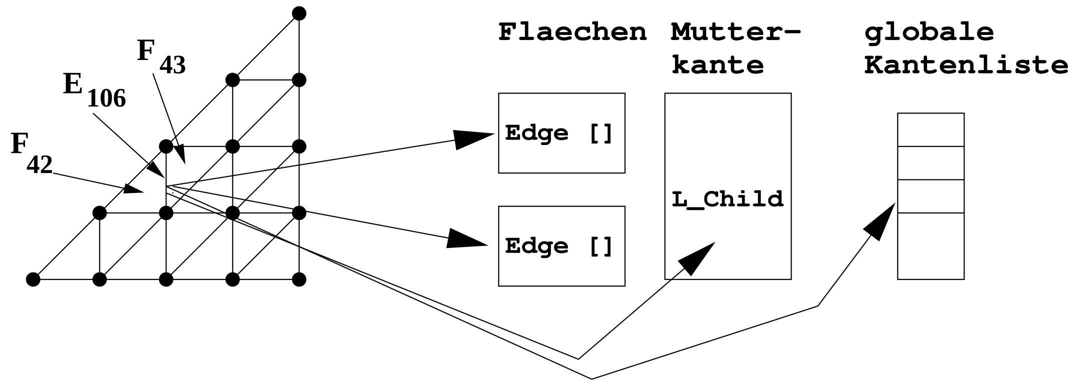

Abbildung 2: Footprint eines Kantenobjekts im `Netgen69-C++`

Abhilfe erreicht man durch Einführung einer Proxy-Klasse gekoppelt mit einem Referenzzähler. Das Entwurfsmuster "Proxy" wird in [1] ausführlich erläutert, daher soll hier nur kurz erklärt werden, wie das Verfahren im konkreten Fall implementiert ist.

Wir arbeiten hier mit einer abstrakten Klasse `RefT`, die nur einen Datenmember, den Referenzzähler `RefCount`, enthält. Diese Klasse dient als Basisklasse für alle FEM - Klassen, dadurch ist sichergestellt, daß der Referenzcounter für jedes FEM - Objekt verwendet werden kann.
Das *public* Interface von `RefT` hat den folgenden Aufbau:

```
class RefT
{
public:

    unsigned int get_ref() const;
    void add_ref();
    void sub_ref();
};
```

Die Konstruktoren, der Zuweisungsoperator und der Destruktor sind alle im *protected* - Bereich der Klasse implementiert. Daher können keine Instanzen von `RefT` angelegt werden, nur Vererbung ist möglich. (Da die Klasse außer dem Referenzzähler keine sonstigen wertvollen Informationen beinhaltet, ist das Instanziieren auch nicht erforderlich.)

Die Methode `get_ref` erlaubt die Abfrage des aktuellen Referenzzählerstandes, die Methode `add_ref` das Erhöhen und die Methode `sub_ref` das Verringern des Referenzzählers.

Es muß vom Nutzer abgesichert werden, daß der Referenzzähler verändert wird, wenn eine Referenz hinzugefügt oder gelöscht werden soll. Um diesen Vorgang zu automatisieren, wurde die Klasse `Proxy` entwickelt, die den Zugriff auf die FEM-Objekte verwalten soll. `Proxy` ist dabei eine Template – Klasse, die mit dem Typen des zu verwaltenden Objekts parametrisiert wird. Dieses Objekt wird intern als Pointer gespeichert. Implizit wird angenommen, daß das zu speichernde Objekt ein Erbe der Klasse `RefT` ist. Zur Compilezeit können allerdings entsprechende Fehlbenutzungen erkannt werden.

Das öffentliche Nutzerinterface von `Proxy` präsentiert sich wie folgt:

```
template <class T>
class Proxy
{
public:
    Proxy(T *t = 0);
    Proxy(const Proxy &P);

    Proxy &operator = (const Proxy &rhs);
    Proxy &operator = (T *t);

    T * operator -> () const;
    T & operator * () const;

    bool operator == (const Proxy<T> &P) const;
    bool operator == (const T &TT) const;

    operator T * () const;

    ~Proxy();
};
```

Intern speichert die `Proxy` – Klasse einen Pointer auf ein Objekt des Typs `T`. Dieser Typ soll hier als der **Inhalts-Typ** benannt werden; das Objekt, das der Proxy verwaltet, heißt dann **Inhalts-Objekt**.

Die Konstruktoren erlauben die Übergabe eines Inhalts-Objekts vom gespeicherten Typ, ebenso wie die Kopie eines existierenden Proxy - Objektes. Hierbei ist das Verhalten unterschiedlich. Wird ein Inhalts-Objekt übergeben, so verändert sich dessen Referenzzähler nicht, da davon ausgegangen werden muß, daß dieses Objekt bisher nur einmal referenziert wird. Ist dagegen das Objekt bereits über einen Proxy verwaltet, so wird man diesen eher als Argument für die Konstruktion eines neuen Proxies verwenden. Bei der Kopie eines anderen Proxy-Objektes wird der Referenzzähler des gespeicherten Objektes verringert (bzw. dieses implizit gelöscht, sofern der Referenzzähler auf 0 abgesunken ist), und der Referenzzähler des Inhalts-Objekts des übergebenen Proxies erhöht. Schließlich wird der Pointer kopiert.

Ähnliche Verhältnisse liegen bei der Verwendung des Zuweisungsoperators vor. Soll ein Inhalts-Objekt zugewiesen werden, so wird dessen Referenzzähler nicht verändert. Bei der Zuweisung eines anderen Proxies wird hier ebenso wie bei der Copy-Konstruktion vorgegangen.

Der `operator ->` wird benutzt, um auf Methoden und Daten des Inhalts-Objekts zuzugreifen. Ein kleines Beispiel soll dies verdeutlichen:

```
#include "refcount.h"
class A : public RefCount
{
        int a1;
public:
        A() : RefCount(),a1(0) {}
        void set_a1(int a) { a1 = a; }
        int get_a1() { return a1;}
};

void f(){
        Proxy<A> My_PA;

        My_PA->set_a1(10);
        cout <<My_PA->get_a1() <<endl;
}
```

Der unäre `operator *` gibt eine Referenz auf das Inhalts-Objekt zurück. Die Funktion `f` in obigem Beispiel könnte unter Benutzung dieses Operators folgendermaßen umgeschrieben werden:

```
void f(){
        Proxy<A> My_PA;

        (*My_PA).set_a1(10);
        cout << (*My_PA).get_a1() << endl;
}
```

Die Abarbeitung wäre im vorliegenden Falle äquivalent.

Die Vergleichsoperatoren können sowohl mit einem anderen Proxy-Objekt, als auch mit einem anderen Objekt des Inhalts-Typs vergleichen.

Weiterhin erlaubt ein casting-Operator `operator T *` den impliziten Cast eines Proxy-Objekts in ein Objekt seines Inhalts-Typs. Auf diese Art und Weise kann das Proxy-Objekt auch an den Stellen verwendet werden, wo eigentlich ein Pointer auf den Inhalts-Typ als Argument erwartet wird.

Im Destruktor wird der Referenzzähler des Inhalts-Objekts dekrementiert. Ist er bereits 0, dann wird `delete` auf dem verwalteten Pointer aufgerufen.

Zur Effizienz des verwendeten Verfahrens ist zu sagen, daß die meisten Funktionen der `Proxy` - Klasse vom Compiler als Inlinefunktionen ausgeführt werden können. Damit entsteht kaum Funktionsaufrufoverhead. Das Speicherplatzverhalten dieser Lösung steht ebenfalls in keiner Weise der Speicherung eines Objekt-Pointers nach. Für jedes unterschiedliche referenzierte Objekt wird lediglich der Speicherplatz eines Pointers auf dieses Objekt belegt.

2.3.2 `Tokenizer` – Ein einfacher Parser zum Einlesen der Grobnetzbeschreibungsdatei

Der Parser `Tokenizer` wird zum Einlesen von Tokens aus Instanzen der STL-`vector`-Klasse benutzt. Hierbei können mehrere gleichrangige Trennzeichen angegeben werden.

Mit Hilfe des Tokenizers werden die durch Komma getrennten Datenfelder der Grobnetzbeschreibungsdatei separiert und einzeln zurückgegeben[6].

Der `Tokenizer` wurde unabhängig von `Netgen69-C++` entwickelt und ist daher nicht ausschließlich Bestandteil von `Netgen69-C++`. Dokumentation zum `Tokenizer` findet sich in [4].

2.3.3 `Meshfile` – ein objektorientiertes Abbild der Grobnetzbeschreibungsdatei

Die Klasse `MeshFile` ist in `include/meshfile.h` definiert und in `src/meshfile.cc` implementiert. Sie wird zum Speichern des Inhaltes der Grobnetzdatei während des Starts des Netzgenerators verwendet, um einen zielgerichteten und wahlfreien Zugriff auf die Daten zu gestatten. Auf diese Weise wird sichergestellt, daß jeder Prozessor am Startpunkt der

[6] Im Unterschied zu FORTRAN kann `C++` nicht von sich aus mit dem Komma als Feldtrennzeichen umgehen. Daher machte sich die Benutzung des `Tokenizers` erforderlich.

Verfeinerung nur diejenigen Daten hält, die zu dem Teilbereich des FEM-Gebietes gehören, den er bearbeitet.

Da die Daten vollständig im Speicher gehalten werden, ist ein wahlfreier Zugriff ohne Aufwand möglich, sodaß die Konstruktion der FEM-Elemente rekursiv erfolgen kann[7]. Dabei liest jede Fläche aus dem `MeshFile` – Objekt die für sie interessanten Kanten ein, und diese Kanten im nächsten Schritt die ihnen zugehörigen Knoten.

Die Klasse hält die Netzbeschreibungsdaten mit Hilfe von `vector`- und `map`-Instanzen der Standard Template Library. Sie liest mittels des Tokenizers (s. 2.3.2) direkt aus der Datei bzw. dem zwischengeschalteten `vector<char>`. Über Ausgabefunktionen greift die Klasse `DistMesh` auf die gespeicherten Informationen zu.

Auf die Implementierung soll hier nicht näher eingegangen werden, da an dieser Klasse noch Verbesserungen zur Effizienzsteigerung geplant sind.

3 Abarbeitung des Netzgenerators

3.1 Einlesen der Grobnetzdatei

Am Anfang der Verarbeitung durch den Netzgenerator steht das Einlesen der Grobnetzdatei. Das Format dieser Datei wird in [3] erläutert. Zum Zwecke der Verarbeitung muß der Nutzer im Hauptprogramm zunächst die gewünschte Datei finden und in einen `vector<char>` einlesen[8]. Hierbei ist zu beachten, daß für die spätere Verarbeitung dieses Vektors Tabulatoren, Leer- und Zeilenendezeichen maßgeblich sind. Sie müssen also mit in den Vektor eingelesen werden. In `C++` bietet sich dafür die Verwendung eines `istream_iterators` an, wie in folgendem Beispiel gezeigt:

```
ifstream infile;                // stream anlegen
vector<char> file;              // vector anlegen

infile.open("mesh3/qu128.net"); // Datei oeffnen
infile.unsetf(ios::skipws);     // Whitespace wird eingelesen
copy(istream_iterator<char>(infile),istream_iterator<char>(),
     back_inserter(file));      // Inhalt in vector kopieren
```

[7]Aufgrund der Reihenfolge dieser Daten in der Grobnetzbeschreibungsdatei kann eine solche Arbeitsweise nicht direkt auf dem Filestream erfolgen, da dann sehr hohe Kosten für das Vor- und Rückwärtsbewegen im Datenstrom nötig wäre.

[8]Hierbei könnte die Klasse `CoarseMesh` hilfreich sein, an einer entsprechenden Implementierung wird noch gearbeitet.

Der entstandene Vektor wird dann an die Methode `Read` der erzeugten `DistMesh` – Instanz weitergegeben:

```
DistMesh DM;
DM->Read(file,FACE_TRIANGLE);
```

In `DistMesh::Read` werden zunächst die Grobnetzdaten aus dem übergebenen `vector<char>` in eine Instanz der Klasse `MeshFile` (s. 2.3.3) eingelesen. Aus diesem Objekt werden dann zunächst die allgemeinen Informationen zur FEM-Rechnung (Daten mit oder ohne Materialbereiche, Anzahl der Freiheitsgrade der Randbedingungen, ...) gewonnen. Ein Aufruf der Methode `DistMesh::calc_own_share` ermittelt den Anteil des aktuellen Prozessors am Gesamt-FEM-Gebiet. Für dieses Gebiet werden nun die "globalen" Datenvektoren des `DistMesh` – Objekts gefüllt, indem gleichzeitig rekursiv Flächen-, Kanten- und Knoten-Objekte angelegt werden (`DistMesh::Read_Own_Vectors`). Jedes neu entstandene Objekt wird von seinem "Verwalter" in den entsprechenden "globalen" Vektor eingetragen.

Anschließend findet eine Prüfung der Prozessorzugehörigkeit für alle Knoten des auf dem entsprechenden Prozessor zu berechnenden FEM-Teilgebietes statt (`DistMesh::Create_Proc_Flag`). In jedem Knoten wird für jeden diesen Knoten besitzenden Prozessor ein Bit in einem Flagfeld gesetzt, sodaß später bei der FEM-Rechnung sehr schnell Informationen darüber gewonnen werden können, welche Knotenwerte zwischen verschiedenen Prozessoren ausgetauscht werden müssen. Dies erleichtert den Aufbau von MPI-Kommunikatoren erheblich [2].

In Abbildung 3 wird der Ablauf noch einmal zusammengefaßt.

3.2 Die Netzverfeinerung

Die Netzverfeinerung läuft prinzipiell in drei Schritten ab:

1. Kantensplitting
2. Erzeugen neuer Kanten
3. Erzeugen neuer Flächen

Die exemplarische Darstellung soll für den Fall von Dreiecksflächen gelten; die Unterschiede zum Verfeinern von Viereckflächen sind aber marginal.

In Abbildung 4 wird der Urzustand einer Fläche vor der Verfeinerung dargestellt.

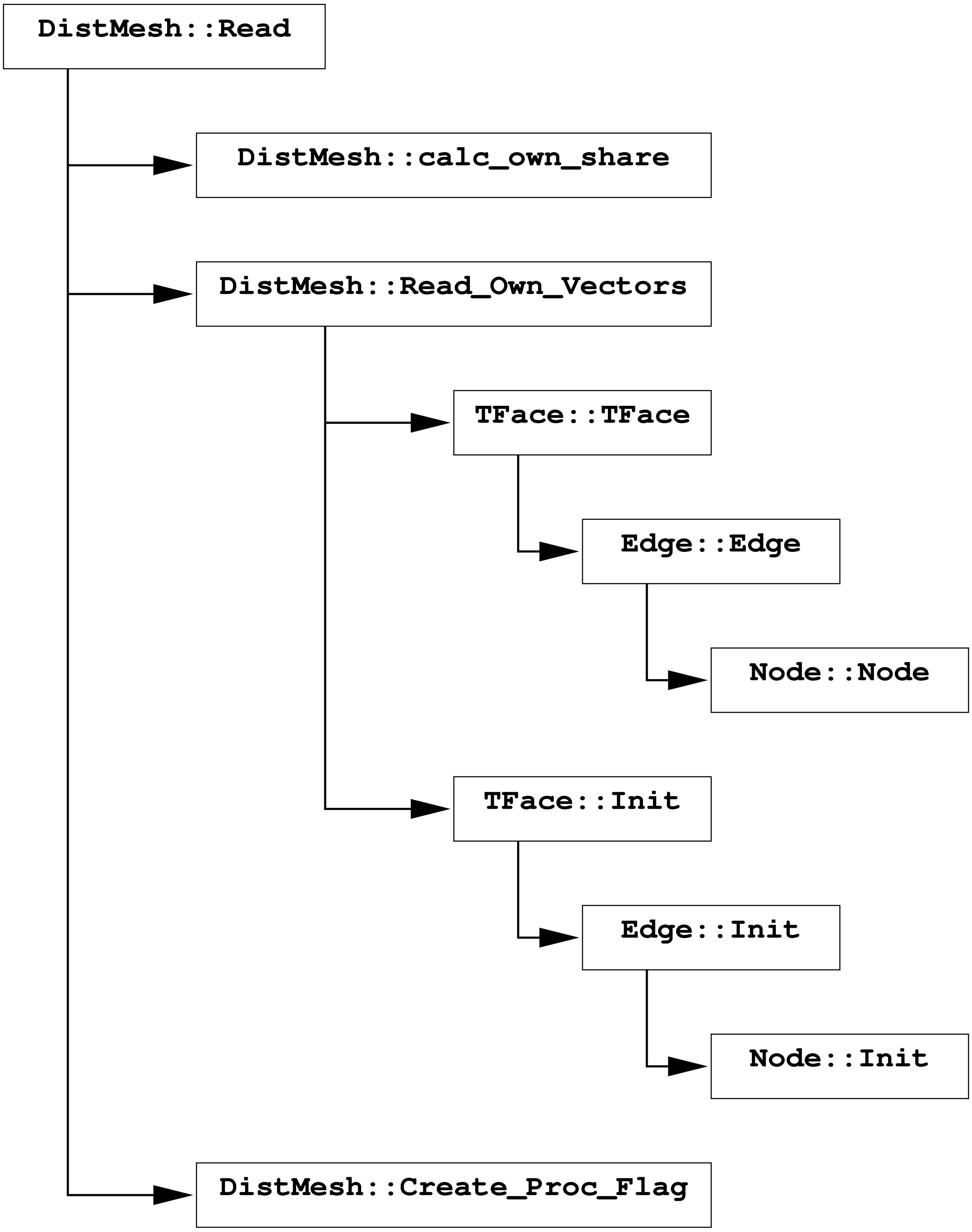

Abbildung 3: Das Einlesen der Grobnetzbeschreibungsdatei.

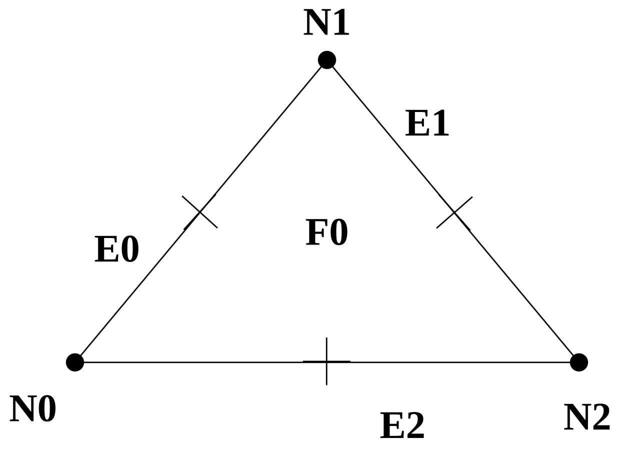

Abbildung 4: Der Ausgangszustand bei der Netzverfeinerung

Ausgelöst wird der Verfeinerungsprozeß durch den Aufruf der Methode `DistMesh::Refine` mit einem Zahlenwert, der die Anzahl der Verfeinerungsschritte ("Level") angibt. Diese Methode ruft ihrerseits in einer Schleife über diese Level-Anzahl die Methode `Face::Refine` für alle Flächenobjekte in der globalen Flächenliste auf. Diese starten dann einen rekursiven Prozeß, der unten erläutert wird. Sie geben je vier neue Flächenobjekte zurück, die in einem neuen globalen Flächenvektor gespeichert werden. Dieser wird im nächsten Level als Basis für die Verfeinerung genommen, und so fort.

3.2.1 Das Kantensplitting

Die Methode `Face::Refine`, die von `DistMesh::Refine` aufgerufen wird, ruft zunächst für alle enthaltenen Kanten die Methode `Edge::Split` auf. Diese Methode bildet aus den beiden Kantenhälften `Start` $\Longrightarrow$ `Middle` und `Middle` $\Longrightarrow$ `End` zwei neue Kanten. Dabei werden zwei neue Mittelknoten gebildet und in die globale Knotenliste (die als Referenz übergeben wird) eingetragen. Die Mutterkante speichert die beiden neuen Kanten in ihren Datenmembern `L_Child` und `R_Child`; eine Eintragung in die globale Kantenliste erfolgt nicht. Die Berechnung der neuen Mittelknoten erfolgt in Abhängigkeit der Kantengeometrie mittels Algorithmen aus dem ursprünglichen NETGEN69.

Wird die Methode `Edge::Split` für eine bereits geteilte Kante aufgerufen, so erkennt das Kantenobjekt dies, indem es bereits belegte Datenmember `L_Child` und `R_Child` feststellt. In diesem Falle kehrt die Methode unmittelbar zurück.

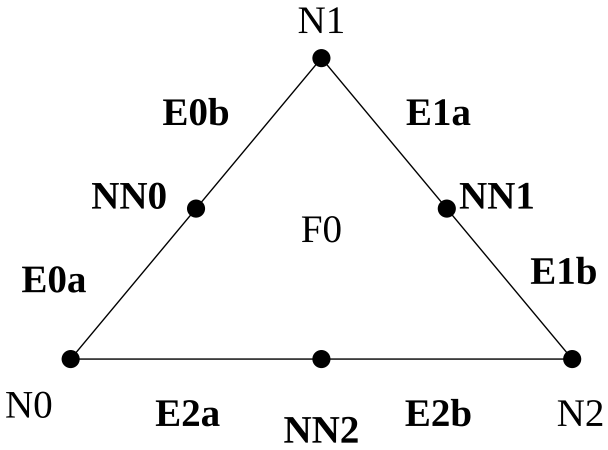

Abbildung 5: Nach Aufruf von `Edge::Split` für alle Kanten der Fläche

3.2.2 Das Erzeugen neuer Kanten

Nachdem die Kanten des Ursprungsflächenobjekts gesplittet sind, müssen die Mittelknoten der ursprünglichen Kanten verbunden werden; dabei entstehen drei "völlig neue" Kanten[9]. Die Konstruktion der Kanten wird wiederum innerhalb von `Face::Refine` durchgeführt. Hierbei kommt die Methode `Edge::New_Edge` zum Einsatz, welche jeweils von einem Kantenobjekt mit dem Mittelknoten der direkt benachbarten Kante und dem Anstieg der gegenüber dem gemeinsamen Knoten liegenden Kante als Parameter ausgeführt wird.

Die Methode `Edge::New_Edge` liefert drei Kanten zurück; die durch diese Methode erzeugte neue Kante, eine Hälfte der Kante, für die die Methode aufgerufen wurde und die Hälfte der anderen Kante, die mit den beiden vorgenannten ein Dreieck bildet. Damit wird die spätere Flächenbildung erleichtert. Die direkt aneinandergrenzenden Hälften zweier Kanten werden von einer der beiden Kanten selbst durch die Methode `Edge::find_adjacent_edge` gefunden. Die in diesem Schritt neu erzeugte Kante wird in die "globale" Kantenliste eingetragen. Damit stehen in dieser Kantenliste alle Kanten zur Verfügung, die nicht unmittelbar durch Splitting entstanden sind. Auf diese Weise ist ein hierarchischer Abstieg über wiederholte Dereferenzierung von `L_Child` und `R_Child` der jeweiligen Kante möglich.

3.2.3 Das Erzeugen neuer Flächen

Bereits nach jeder neu erzeugten Kante (vgl. 3.2.2) wird jeweils eine neue Fläche aus den von `Edge::New_Edge` zurückgelieferten Kanten gebildet. Das Erzeugen der Flächen geschieht durch Aufrauf der Methode `Face::New_Face` direkt aus der Methode

[9] Auch die im vorherigen Schritt entstandenen Kanten sind neu; sie entstanden jedoch durch die Teilung bereits vorhandener Kanten.

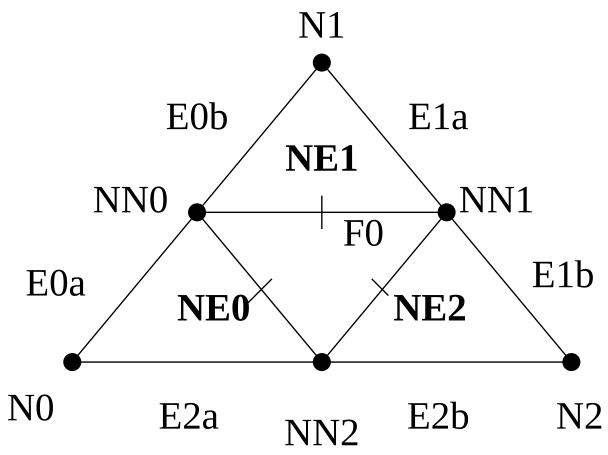

Abbildung 6: Nach Aufruf von `Edge::New_Edge` zur Erzeugung dreier neuer Kanten.

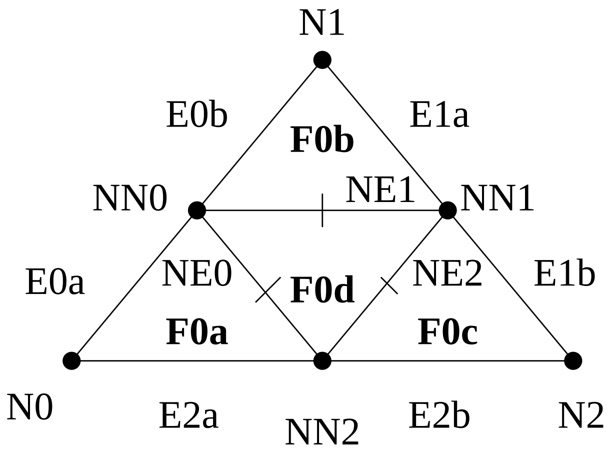

Abbildung 7: Nach der Abarbeitung von `Face::New_Face` für vier neue Flächen.

`Face::Refine`, in der wir uns noch immer befinden. Nachdem die drei neuen Kanten gebildet und daraus auch bereits die neuen "Rand"-Flächen erzeugt wurden, produziert ein weiterer Aufruf von `Face::New_Face` mit den drei Neukanten als Argumente die vierte (zentrale) Fläche.

Hiermit ist die Verfeinerung einer Fläche beendet; die vier neu erzeugten Flächen werden an `DistMesh::Refine` zurückgeliefert.

Die gesamte Netzverfeinerung wird nochmals in Abbildung 8 zusammengefaßt.

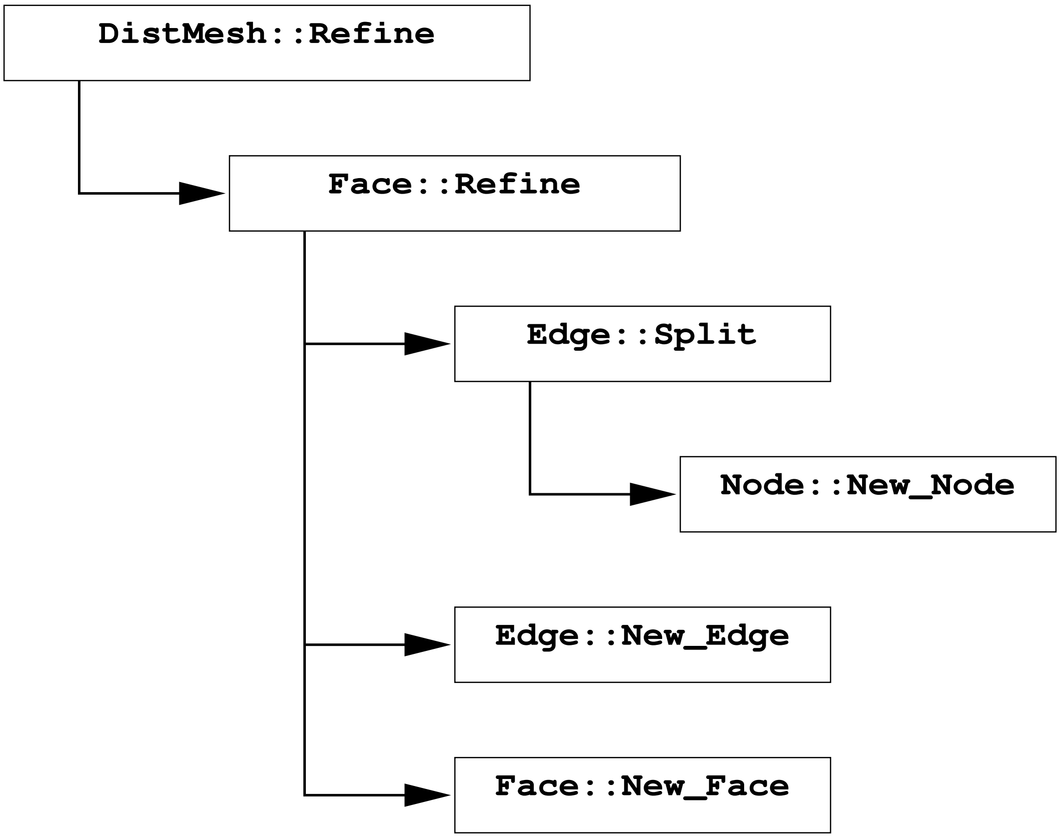

Abbildung 8: Die Netzverfeinerung

4 Vergleich zwischen NETGEN69 und `Netgen69-C++`

Ein Vergleich zwischen dem in FORTRAN geschriebenen NETGEN69 und dem `Netgen69-C++` führt einerseits unmittelbar auf einen Leistungsvergleich hinaus, jedoch können auch einige allgemeinere Aussagen getroffen werden.

Zunächst ist zu bemerken, daß ein Vergleich schon allein deshalb schwierig ist, weil neben einer neuen Programmiersprache auch ein neues Programmierparadigma verwendet wurde. Darüberhinaus sind einige Vergleichspunkte anfällig für subjektive Betrachtungen.

1. `Netgen69-C++` ist einfacher zu warten und zu erweitern als NETGEN69.

   Diese Aussage stützt sich auf die größere Übersichtlichkeit und Modularität von objektorientiertem Code. Schon anhand der Klassen- und Klassenmemberfunktionen läßt sich der Abarbeitungsablauf leichter verfolgen, als wenn dies nur über unterschiedliche Funktionsnamen geschieht. Es wird deutlicher, welche **Objekte** bearbeitet werden, und mit welchen **Methoden** dies geschieht.

2. `Netgen69-C++` ist sowohl in Binär- als auch in Quelltextform größer als NETGEN69.

   Der Größenunterschied beträgt im Binärformat ungefähr 10:1, während die statische Bibliothek `libnetgen69.a` aus FORTRAN-Quelltext ungefähr 50 kB groß ist, erreicht die `C++`–Version die zehnfache Größe[10].
   Ein Grund für die Größenunterschiede sowohl in Quelltext- als auch in Binärform liegt vor allem in dem natürlichen Overhead, den die Klassenstruktur erzeugt. Gerade dieser Overhead, zum Beispiel die Deklaration der Klassen und die Definition wiederkehrender Funktionen, wie zum Beispiel Konstruktoren, gewissen Operatoren, etc., schafft die Voraussetzungen für die oben genannte bessere Wartbarkeit.

3. Die Ergebnisse von NETGEN69 und `Netgen69-C++` sind äquivalent.

   Diese Tatsache wird durch Messungen unterstützt, die nach Teil-Fertigstellung des `Netgen69-C++` vorgenommen wurden. Die Ergebnisse (Position der Knoten im verfeinerten Netz, propagierte Randbedingungen) stimmen überein.

4. `Netgen69-C++` bietet größeren Komfort im Speichermanagement als NETGEN69.

   Da in FORTRAN keine dynamische Speicherverwaltung existiert, wurde die Speicherzuteilung im NETGEN69 über ein zum Start der Anwendung angelegtes großes Feld gehandhabt. Die Dimension dieses Feldes ließ sich in einer include-Datei voreinstellen, sodaß für größere Felddimensionen die Anwendung neu compiliert werden mußte.

[10] Man kann beim Compilieren der `C++`–Bibliothek entscheiden, ob man Debug-Informationen mit einbindet; in diesem Fall steigt die Größe nochmals auf das 3 ... 4fache, also verglichen zur FORTRAN–Version, die diesen "Luxus" nicht hat, auf das 30 ... 40fache.

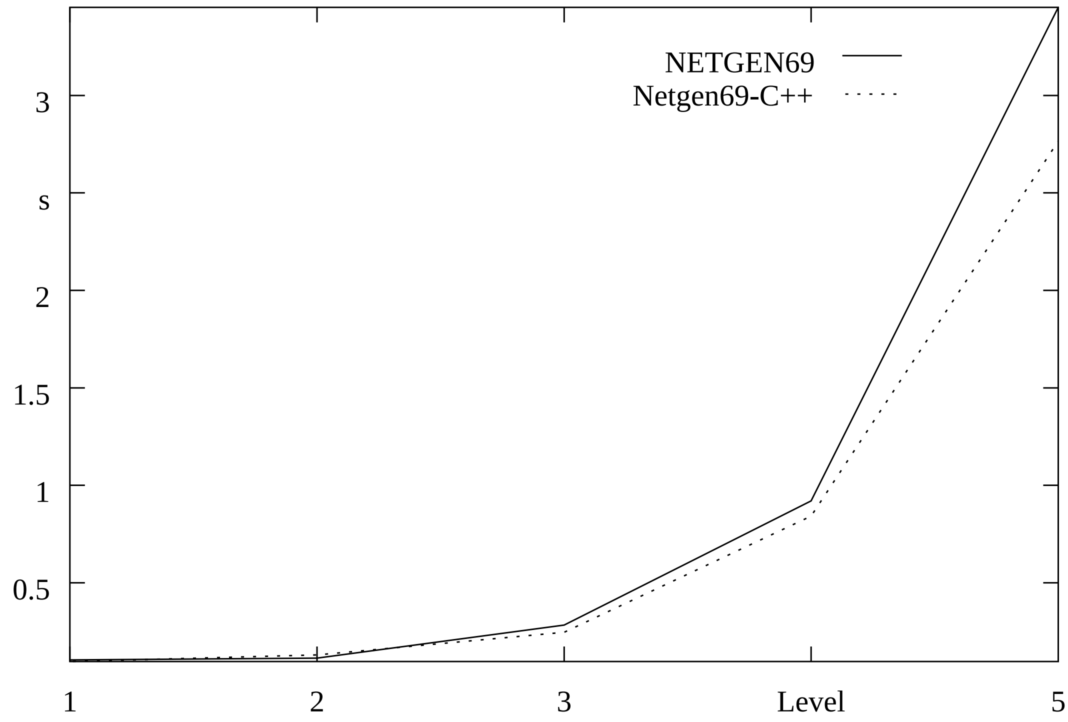

Abbildung 9: Performancevergleich `Netgen69-C++` – NETGEN69.

Dieses Erfordernis besteht bei `Netgen69-C++` nicht mehr. Er ist in der Lage, jede Speicheranforderung zu erfüllen, die das darunterliegende Rechnersystem zuläßt. Gleichzeitig wird zu keinem Zeitpunkt mehr Speicher angefordert, als unbedingt benötigt wird. Dadurch können auf demselben System gleichzeitig auch weitere Anwendungen den verfügbaren Platz mit der FEM – Anwendung teilen.[11]

Inwieweit der `C++` – Code dem Performancevergleich mit dem FORTRAN – Code standhält, soll folgende Messung zeigen. Sie wurde auf einem Dual-Pentium-II mit 266 MHz und 128 MB Hauptspeicher vorgenommen. Es liegen jeweils die Daten eines verwendeten Prozessors zugrunde.[12] NETGEN69 wurde mit dem im `egcs-1.0.2` enthaltenen `g77` compiliert, `Netgen69-C++` mit dem `egcs-1.0.2` selbst.

Zur besseren Anschaulichkeit wurden die einzelnen Meßpunkte im Diagramm (s. Abb. 9) durch gerade Linien verbunden.

Aus Abbildung 9 kann man zwei Tatsachen ableiten:

[11] Dies wird aus Performancegründen nicht unbedingt wünschenswert sein; dennoch ergibt sich diese Möglichkeit automatisch.

[12] Da im Netzgenerator keinerlei Kommunikation auftritt, ist es für den Vergleich nicht erforderlich, Messungen unter Multiprozessor-Bedingungen vorzunehmen.

1. Im Verfeinerungsbereich 2 ... 4 Level, der nach Aussage von Nutzern für die Verarbeitung in den FEM-Programmpaketen am praktikabelsten ist, erreicht `Netgen69-C++` die Verarbeitungsgeschwindigkeit von NETGEN69, bzw. übertrifft diese geringfügig.

2. Für höhere Levelzahlen[13] übertrifft die Geschwindigkeit des `Netgen69-C++` die von NETGEN69 in immer stärkerem Maße.

5 Zukünftige Entwicklungen

Künftige Weiterentwicklungen im Umfeld des `Netgen69-C++` beinhalten:

- Implementierung der `RFace` – Klasse zur Darstellung von Vierecksflächen. Diese Implementierung wurde zunächst zugunsten einer stabilen Implementierung des Gesamt-`Netgen69-C++` zurückgestellt, da nach Aussage von Nutzern in der momentanen Arbeit mit den FEM-Programmpaketen Dreiecksflächen von höherer Bedeutung sind.

- Verbesserung des `DistMesh::calc_own_share` – Algorithmus. Bislang wird hier der Gesamtvorrat an Grobnetzflächen einfach linear auf die vorhandenen Prozessoren aufgeteilt. Im Interesse der Adaptivität wäre es jedoch wünschenswert, wenn der Gesamtvorrat nach anderen Gesichtspunkten aufgeteilt würde.

- Objektorientierte Überarbeitung des gesamten FEM-Systems. Der Netzgenerator ist Ausgangspunkt eines OO-Designs des Gesamtsystems. Aufbauend auf die erfolgreiche Entwicklung von `Netgen69-C++` soll nun auch der Rest des FEM–Systems objektorientiert implementiert werden.

Literatur

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[2] L. Grabowsky, T. Ermer, and J. Werner. Nutzung von MPI für parallele FEM-Systeme. Preprint SFB393/97-08, TU Chemnitz-Zwickau, März 1997.

[3] C. Israel. NETGEN69 - Ein hierarchischer paralleler Netzgenerator. Preprint SPC 95_26, TU Chemnitz-Zwickau, August 1995.

[13]Mehr als 5 Level ließ die Hardwarekonfiguration des Testsystems aus Gründen des verfügbaren Hauptspeichers nicht zu.

[4] M. Meyer. `http://www.tu-chemnitz.de/~marme/programs/tokenizer/` WWW – Page.

[5] D. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison – Wesley, 1996.

[6] B. Stroustrup. *The C++ Programming Language*. Addison – Wesley, 3rd edition, 1997.

Other titles in the SFB393 series:

96-01 V. Mehrmann, H. Xu. Chosing poles so that the single-input pole placement problem is well-conditioned. Januar 1996.

96-02 T. Penzl. Numerical solution of generalized Lyapunov equations. January 1996.

96-03 M. Scherzer, A. Meyer. Zur Berechnung von Spannungs- und Deformationsfeldern an Interface-Ecken im nichtlinearen Deformationsbereich auf Parallelrechnern. March 1996.

96-04 Th. Frank, E. Wassen. Parallel solution algorithms for Lagrangian simulation of disperse multiphase flows. Proc. of 2nd Int. Symposium on Numerical Methods for Multiphase Flows, ASME Fluids Engineering Division Summer Meeting, July 7-11, 1996, San Diego, CA, USA. June 1996.

96-05 P. Benner, V. Mehrmann, H. Xu. A numerically stable, structure preserving method for computing the eigenvalues of real Hamiltonian or symplectic pencils. April 1996.

96-06 P. Benner, R. Byers, E. Barth. HAMEV and SQRED: Fortran 77 Subroutines for Computing the Eigenvalues of Hamiltonian Matrices Using Van Loans's Square Reduced Method. May 1996.

96-07 W. Rehm (Ed.). Portierbare numerische Simulation auf parallelen Architekturen. April 1996.

96-08 J. Weickert. Navier-Stokes equations as a differential-algebraic system. August 1996.

96-09 R. Byers, C. He, V. Mehrmann. Where is the nearest non-regular pencil? August 1996.

96-10 Th. Apel. A note on anisotropic interpolation error estimates for isoparametric quadrilateral finite elements. November 1996.

96-11 Th. Apel, G. Lube. Anisotropic mesh refinement for singularly perturbed reaction diffusion problems. November 1996.

96-12 B. Heise, M. Jung. Scalability, efficiency, and robustness of parallel multilevel solvers for nonlinear equations. September 1996.

96-13 F. Milde, R. A. Römer, M. Schreiber. Multifractal analysis of the metal-insulator transition in anisotropic systems. October 1996.

96-14 R. Schneider, P. L. Levin, M. Spasojević. Multiscale compression of BEM equations for electrostatic systems. October 1996.

96-15 M. Spasojević, R. Schneider, P. L. Levin. On the creation of sparse Boundary Element matrices for two dimensional electrostatics problems using the orthogonal Haar wavelet. October 1996.

96-16 S. Dahlke, W. Dahmen, R. Hochmuth, R. Schneider. Stable multiscale bases and local error estimation for elliptic problems. October 1996.

96-17 B. H. Kleemann, A. Rathsfeld, R. Schneider. Multiscale methods for Boundary Integral Equations and their application to boundary value problems in scattering theory and geodesy. October 1996.

96-18 U. Reichel. Partitionierung von Finite-Elemente-Netzen. November 1996.

96-19 W. Dahmen, R. Schneider. Composite wavelet bases for operator equations. November 1996.

96-20 R. A. Römer, M. Schreiber. No enhancement of the localization length for two interacting particles in a random potential. December 1996. to appear in: Phys. Rev. Lett., March 1997

96-21 G. Windisch. Two-point boundary value problems with piecewise constant coefficients: weak solution and exact discretization. December 1996.

96-22 M. Jung, S. V. Nepomnyaschikh. Variable preconditioning procedures for elliptic problems. December 1996.

97-01 P. Benner, V. Mehrmann, H. Xu. A new method for computing the stable invariant subspace of a real Hamiltonian matrix or Breaking Van Loan's curse? January 1997.

97-02 B. Benhammouda. Rank-revealing 'top-down' ULV factorizations. January 1997.

97-03 U. Schrader. Convergence of Asynchronous Jacobi-Newton-Iterations. January 1997.

97-04 U.-J. Görke, R. Kreißig. Einflußfaktoren bei der Identifikation von Materialparametern elastisch-plastischer Deformationsgesetze aus inhomogenen Verschiebungsfeldern. March 1997.

97-05 U. Groh. FEM auf irregulären hierarchischen Dreiecksnetzen. March 1997.

97-06 Th. Apel. Interpolation of non-smooth functions on anisotropic finite element meshes. March 1997

97-07 Th. Apel, S. Nicaise. The finite element method with anisotropic mesh grading for elliptic problems in domains with corners and edges.

97-08 L. Grabowsky, Th. Ermer, J. Werner. Nutzung von MPI für parallele FEM-Systeme. March 1997.

97-09 T. Wappler, Th. Vojta, M. Schreiber. Monte-Carlo simulations of the dynamical behavior of the Coulomb glass. March 1997.

97-10 M. Pester. Behandlung gekrümmter Oberflächen in einem 3D-FEM-Programm für Parallelrechner. April 1997.

97-11 G. Globisch, S. V. Nepomnyaschikh. The hierarchical preconditioning having unstructured grids. April 1997.

97-12 R. V. Pai, A. Punnoose, R. A. Römer. The Mott-Anderson transition in the disordered one-dimensional Hubbard model. April 1997.

97-13 M. Thess. Parallel Multilevel Preconditioners for Problems of Thin Smooth Shells. May 1997.

97-14 A. Eilmes, R. A. Römer, M. Schreiber. The two-dimensional Anderson model of localization with random hopping. June 1997.

97-15 M. Jung, J. F. Maitre. Some remarks on the constant in the strengthened C.B.S. inequality: Application to h- and p-hierarchical finite element discretizations of elasticity problems. July 1997.

97-16 G. Kunert. Error estimation for anisotropic tetrahedral and triangular finite element meshes. August 1997.

97-17 L. Grabowsky. MPI-basierte Koppelrandkommunikation und Einfluß der Partitionierung im 3D-Fall. August 1997.

97-18 R. A. Römer, M. Schreiber. Weak delocalization dueto long-range interaction for two electrons in a random potential chain. August 1997.

97-19 A. Eilmes, R. A. Römer, M. Schreiber. Critical behavior in the two-dimensional Anderson model of localization with random hopping. August 1997.

97-20 M. Meisel, A. Meyer. Hierarchically preconditioned parallel CG-solvers with and without coarse-matrix-solvers inside FEAP. September 1997.

97-21 J. X. Zhong, U. Grimm, R. A. Römer, M. Schreiber. Level-Spacing Distributions of Planar Quasiperiodic Tight-Binding Models. October 1997.

97-22 W. Rehm (Ed.). Ausgewählte Beiträge zum 1. Workshop Cluster-Computing. TU Chemnitz, 6./7. November 1997.

97-23 P. Benner, Enrique S. Quintana-Ortí. Solving stable generalized Lyapunov equations with the matrix sign function. October 1997

97-24 T. Penzl. A Multi-Grid Method for Generalized Lyapunow Equations. October 1997

97-25 G. Globisch. The hierarchical preconditioning having unstructured threedimensional grids. December 1997

97-26 G. Ammar, C. Mehl, V. Mehrmann. Schur-like forms for matrix Lie groups, Lie algebras and Jordan algebras. November 1997

97-27 U. Elsner. Graph partitioning - a survey. December 1997.

97-28 W. Dahmen, R. Schneider. Composite Wavelet Bases for Operator Equations. December 1997.

97-29 P. L. Levin, M. Spasojević, R. Schneider. Creation of Sparse Boundary Element Matrices for 2-D and Axi-symmetric Electrostatics Problems Using the Bi-orthogonal Haar Wavelet. December 1997.

97-30 W. Dahmen, R. Schneider. Wavelets on Manifolds I: Construction and Domain Decomposition. December 1997.

97-31 U. Elsner, V. Mehrmann, F. Milde, R. A. Römer, M. Schreiber. The Anderson Model of Localization: A Challenge for Modern Eigenvalue Methods. December 1997.

98-01 B. Heinrich, S. Nicaise, B. Weber. Elliptic interface problems in axisymmetric domains. Part II: The Fourier-finite-element approximation of non-tensorial singularities. January 1998.

98-02 T. Vojta, R. A. Römer, M. Schreiber. Two interfacing particles in a random potential: The random model revisited. February 1998.

98-03 B. Mehlig, K. Müller. Non-universal properties of a complex quantum spectrum. February 1998.

98-04 B. Mehlig, K. Müller, B. Eckhardt. Phase-space localization and matrix element distributions in systems with mixed classical phase space. February 1998.

98-05 M. Bollhöfer, V. Mehrmann. Nested divide and conquer concepts for the solution of large sparse linear systems. to app.: April 1998.

98-06 T. Penzl. A cyclic low rank Smith method for large, sparse Lyapunov equations with applications in model reduction and optimal control. March 1998.

The complete list of current and former preprints is available via
`http://www.tu-chemnitz.de/sfb393/preprints.html`.