
Universität Karlsruhe (TH)
Institut für Praktische Mathematik
Prof. Dr. C. Wieners

Parallele Finite Elemente

Skript zur Vorlesung im Sommersemester 2004

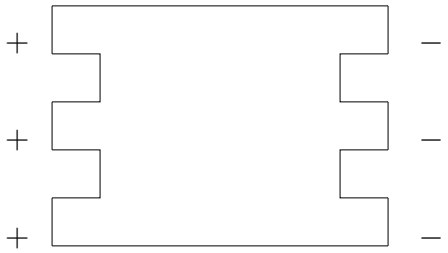
Version vom 24. April 2006

Inhaltsverzeichnis

1 Einführung

Wir entwickeln ein paralleles Programm für ein einfaches Modellproblem.

Dazu betrachten wir ein Polygongebiet $\Omega \subset \mathbf{R}^2$, und $\Gamma \subset \partial\Omega$ sei ein Teil vom Rand.



Wir betrachten die folgende Aufgabe: Berechne ein Potential

$$u\colon \bar{\Omega} \to \mathbf{R}$$

zu gegebenen Randwerten

$$d\colon \Gamma \to \mathbf{R}\,,$$

das die partielle Differentialgleichung

$$\begin{aligned} \Delta u(x) &= \frac{\partial^2}{\partial x_1^2}u(x) + \frac{\partial^2}{\partial x_2^2}u(x) = 0 && x \in \Omega \\ u(x) &= d(x) && x \in \Gamma \\ \nabla u(x) \cdot n &= \frac{\partial}{\partial x_1}u(x)n_1 + \frac{\partial}{\partial x_2}u(x)n_2 = 0 && x \in \partial\Omega \setminus \Gamma \end{aligned}$$

löst. Dabei sei n der äußere Normalenvektor auf dem Rand $\partial\Omega$. Derartige physikalische Gleichungen beschreiben z. B. ein elektrisches Potential, eine Temperatur- oder Druckverteilung etc.

Die numerische Approximation von u durch stückweise lineare Approximationen $u_h \approx u$ führt auf folgendes Problem: Für lineare Funktionen gilt $\Delta u_h = 0$!

Daher betrachten wir schwache Lösungen, d.h. Lösungen der schwachen Formulierung.

1. Schritt: Hauptsatz der Variationsrechnung

$$\Delta u = 0 \quad \Longleftrightarrow \quad \int_\Omega \Delta u\, v\, dx = 0 \text{ für alle } v$$

2. Schritt: Satz von Gauß für $q = \nabla u\, v\colon \Omega \to \mathbf{R}^2$

$$\int_\Omega \operatorname{div} q\, dx = \int_{\partial\Omega} q \cdot n\, ds$$

(Es gilt $\operatorname{div} q = \Delta u\, v + \nabla u \cdot \nabla v$.)

$$\nabla u \cdot \nabla v := \frac{\partial}{\partial x_1} u \cdot \frac{\partial}{\partial x_1} v + \frac{\partial}{\partial x_2} u \cdot \frac{\partial}{\partial x_2} v$$

$$\Longrightarrow \quad \int_\Omega \Delta u\, v\, dx + \int_\Omega \nabla u \cdot \nabla v\, dx = \int_{\partial\Omega} v \nabla\, u \cdot n \quad ds$$

$$\Longrightarrow \quad \int_\Omega \nabla\, u \cdot \nabla\, v\, dx = 0$$

für $\Delta u = 0$ in Ω

$\nabla u \cdot n = 0$ auf $\partial\Omega \setminus \Gamma$

$v = 0$ auf Γ

Ziel

1) Approximiere u in einem geeigneten endlich dimensionalen Vektorraum (*Finite Elemente*)
2) Stelle ein lineares Gleichungssystem für das Problem auf (*Assemblieren*)
3) Löse das System (*parallele Iterationsverfahren*)

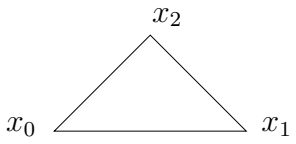
zu 1) Eine zuverlässige Triangulierung von Ω ist ein Tupel $\mathcal{M} = (\mathcal{V}, \mathcal{C})$ (*mesh*) mit

$\mathcal{V} \subset \bar{\Omega}$ *(vertex)*

$\mathcal{C} \subset \mathcal{V} \times \mathcal{V} \times \mathcal{V}$ *(cell)*

und

a) Für alle $C = (x_0, x_1, x_2) \in \mathcal{C}$ gilt:
die drei Punkte x_0, x_1, x_2 bilden ein Dreieck $\bar{\Omega}_C := \operatorname{conv}(C)$ der Fläche $|\Omega_C| > 0$.
Es ist dabei Konvention: $\det(x_1 - x_0, x_2 - x_0) > 0$



$$x \in C \iff x \in \{x_0, x_1, x_2\}$$

d. h. der Vektor $C = (x_0, x_1, x_2)$ wird auch als Menge $C = \{x_0, x_1, x_2\}$ aufgefasst.

b) Für $C, C' \in \mathcal{C}$ gilt

$$\operatorname{conv}(C) \cap \operatorname{conv}(C') = \operatorname{conv}(C \cap C')$$

d.h. der Durchschnitt für $C \neq C'$ ist

- entweder leer
- oder eine Ecke x_j
- oder eine Kante/Seite $\operatorname{conv}\{x_j, x_k\}$

c) $\mathcal{V} = \bigcup_{C \in \mathcal{C}} C$, $\bar{\Omega}_{\mathcal{C}} := \bigcup_{C \in \mathcal{C}} \bar{\Omega}_C$ approximiert Ω
Hier: Wir betrachten nur Polygongebiete $\bar{\Omega} = \bar{\Omega}_{\mathcal{C}}$.

Damit wird der Finite-Elemente-Raum

$$V_{\mathcal{C}} := \{v_h \in C(\bar{\Omega}) : \quad v_h|_{\Omega_C} \text{ linear } \forall\, C \in \mathcal{C}\}$$

definiert ($h := \max\{|x_j - x_k| : \quad x_j \neq x_k,\ x_j, x_k \in C \in \mathcal{C}\}$).
Setze $V_{\mathcal{C}}(d) := \{u_h \in V_{\mathcal{C}} : \quad u_h(x) = d(x)$ für alle $x \in \mathcal{V} \cap \Gamma\}$ und löse

$$u_h \in V_{\mathcal{C}}(d) : \quad \int_\Omega \nabla u_h \cdot \nabla v_h \, dx = 0 \quad \forall\, v_h \in V_{\mathcal{C}}(0)$$

Satz

$u_h \in V_{\mathcal{C}}$ ist eindeutig durch $u_h|_{\mathcal{V}} : \mathcal{V} \longrightarrow \mathbf{R}$ bestimmt.
Definiere Knotenbasis $\varphi_x \in V_{\mathcal{C}}$ mit

$$\varphi_x(y) = \begin{cases} 1 & x = y \\ 0 & x \neq y \end{cases} \qquad x, y \in \mathcal{V} \qquad \Longrightarrow \qquad u_h(y) = \sum_{x \in \mathcal{V}} u_h(x)\, \varphi_x(y) \qquad y \in \bar{\Omega}\,.$$

Wähle $u_h^0 \in V_{\mathcal{C}}(d)$ und bestimme $u_h = c_h + u_h^0$ mit

$$c_h \in V_{\mathcal{C}}(0) : \quad \int_\Omega \nabla c_h \cdot \nabla \varphi_y \, dx = -\int_\Omega \nabla u_h^0 \cdot \nabla \varphi_y \, dx \qquad \forall y \in \mathcal{V} \setminus \Gamma$$

zu 2) Sei $\mathcal{P} = \{0, ..., P-1\}$ $\quad P = |\mathcal{P}|$ die Prozessormenge.
Wähle $\operatorname{dest}(C) \in \mathcal{P}$ $\quad C \in \mathcal{C}$, $\mathcal{C}_p = \{C \in \mathcal{C} : \operatorname{dest}(C) = p\}$
$\Longrightarrow$ $\mathcal{C} = \bigcup_{p \in \mathcal{P}} \mathcal{C}_p$ disjunkte Zerlegung

$$\bar{\Omega}_p = \bigcup_{c \in \mathcal{C}_p} \bar{\Omega}_C$$

$$\mathcal{V} = \bigcup_{p \in \mathcal{P}} \mathcal{V}_p$$

überlappende Zerlegung mit $\mathcal{V}_p = \bigcup_{C \in \mathcal{C}_p} C$.

bestimme $c_h = (c_h^p)_{p \in \mathcal{P}}$ mit $c_h^p : \bar{\Omega}_p \to \mathbf{R}$ und
$c_h^p(x) = c_h^q(x)$ für $x \in \bar{\Omega}_p \cap \bar{\Omega}_q$ und

$$\sum_{p \in \mathcal{P}} \int_{\Omega_p} \nabla c_h^p \cdot \nabla \varphi_y \, dx = -\sum_{p \in \mathcal{P}} \int_{\Omega_p} \nabla u_h^0 \cdot \varphi_y \, dx \qquad y \in \mathcal{V} \setminus \Gamma$$

Wähle Nummerierung $G_p \colon \mathcal{V}^p \longrightarrow \{0, ..., N^p - 1\}$, wobei $N^p = |\mathcal{V}^p|$.
Zu $\underline{u} = (\underline{u}_p) \in \prod_{p \in \mathcal{P}} \mathbf{R}^{N^p}$ definiere

$$\underline{u}_p(x) := \begin{cases} \underline{u}_p[G_p(x)] & \text{für } x \in \mathcal{V}^p \\ 0 & \text{für } x \notin \mathcal{V}^p \end{cases} .$$

Bestimme $\underline{c} = (\underline{c}_p)$ mit

$$\Big(\sum_{p \in \mathcal{P}} \underline{A}_p \underline{c}_p\Big)(x) = \sum_{p \in \mathcal{P}} \underline{b}_p(x) \text{ für alle } x \in \mathcal{V}, \text{ wobei } \underline{A}_p \in \mathbf{R}^{N^p, N^p} \text{ und } \mathbf{R}^{N^p} \text{und}$$

$$\underline{A}_p[G_p(x), G_p(y)] = \begin{cases} \int\limits_{\Omega_p} \nabla \varphi_x \cdot \nabla \varphi_y dx & \text{für } x, y \in \mathcal{V}^p \setminus \Gamma \\ 1 & \text{für } x = y \in \mathcal{V}^p \cap \Gamma \\ 0 & \text{sonst} \end{cases} .$$

$$\underline{b}_p[G_p(x)] = \begin{cases} -\int\limits_{\Omega_p} \nabla u^0 \cdot \nabla \varphi_y dx & \text{für } y \in \mathcal{V}^p \setminus \Gamma \\ 0 & \text{sonst} \end{cases} .$$

zu 3) parallele Realisation von

$\underline{r} = \underline{b}$
solange $|\underline{r}| > \varepsilon$
$\underline{c} = \underline{Br}$ Vorkonditionieren
$\underline{u} = \underline{u} + \underline{c}$
$\underline{r} = \underline{r} - \underline{Ac}$

1.1 Paralleles Programm 1

1.1.1 Beschreibung

In dieser ersten Version des parallelen Programms wird ein paralleles Programmiermodell realisiert, das als Grundlage für alle weiteren parallelen Programme dient. Dazu wird die Bibliothek *MPI* verwendet. Auf diese Weise kann mit nur wenigen Zeilen Quelltext ein voll parallelisierbares Programm geschrieben werden. Das soll hier demonstriert werden.

In der ersten Version des Programmes wird also ein paralleles Programmiermodell definiert und jeder Prozessor gibt die Anzahl der beteiligten Prozessoren im Standardoutput aus.

1.1.2 Anmerkung

Die Dokumentation der parallelen Programme erfolgt stets nach dem folgenden Schema: Zunächst werden in einer kurzen Beschreibung die wesentlichen Aspekte jeder Programmversion genannt. Danach erfolgt eine Auflistung der Quelltext-Dateien.
Anschließend werden die verwendeten Strukturen genauer erläutert. Das geschieht durch eine Auflistung aller definierten Funktionen und aller globalen Methoden, die für einzelne Klassen definiert wurden. Die Beschreibung einer Klassenmethode wird dabei durch ein schwarzes Quadrat (■) gekennzeichnet. Die Beschreibung einer globalen Funktion, Variable oder Konstante wird mit einem weißen Quadrat (□) gekennzeichnet.
Relevante Quelltexte werden mit einer Zeilennummerierung angezeigt. Ein Bezug auf die Zeilennummern wird in der Form $_{(123)}$ oder $_{(123–456)}$ angegeben.
Vielfach werden bereits bekannte Abschnitte aus den Quelltexten entfernt. Wo dies geschehen ist, erscheinen drei Punkte (. . .).

1.1.3 Dateien

```
PFEM/01/Parallel.h
PFEM/01/Parallel.C
PFEM/01/Main.C
PFEM/01/Makefile
```

Die Datei `Makefile` ist die Make-Datei des Programms. Sie dient dem Programm *make*, das die Kompilierung des Programms steuert, als Eingabedatei. Um die erste Version des Programms zu kompilieren, begibt man sich zunächst in das Verzeichnis `PFEM/01/`, und ruft dann den Befehl

```
make
```

in der Konsole auf. Der C-Compiler der *MPI*-Bibliothek *mpiCC* erzeugt daraufhin die ausführbare Datei

```
PFEM/01/p01
```

Das parallele Programm kann dann von der Konsole aus mit dem Befehl

```
mpirun -np N p01
```

gestartet werden, wobei `N` die Anzahl der parallelen Prozesse angibt, die zur Abarbeitung des Programms verwendet werden sollen. Das hier beschriebene Vorgehen zur Kompilierung und Ausführung des parallelen Programms gilt im folgenden für alle Programme.

1.1.4 Das parallele Programmiermodell

```
1  // file:    PFEM/01/Parallel.h
2
3  #ifndef _PARALLEL_H_
4  #define _PARALLEL_H_
5
6  #include <iostream>
7  using namespace std;
8
9  class ParallelProgrammingModel {
10     int p;
11     int N;
12  public:
13     short proc () const { return p; }
14     short size () const { return N; }
15     bool master () const { return (p == 0); }
16  public:
17     ParallelProgrammingModel (int*, char ***);
18     ~ParallelProgrammingModel ();
19 };
20 extern ParallelProgrammingModel* PPM;
21
22 #define pout cout << PPM->proc() << ": "
23 #endif
```

```
1  // file:    PFEM/01/Parallel.C
2
3  #include "Parallel.h"
4
5  #include "mpi.h"
6
```

```
7  ParallelProgrammingModel* PPM = 0;
8
9  ParallelProgrammingModel::ParallelProgrammingModel (int* argc, char ***argv) {
10     MPI_Init(argc,argv);
11     MPI_Comm_size(MPI_COMM_WORLD,&N);
12     MPI_Comm_rank(MPI_COMM_WORLD,&p);
13 }
14 ParallelProgrammingModel::~ParallelProgrammingModel () {  MPI_Finalize(); }
```

Die Klasse **ParallelProgrammingModel** bildet für den Programmierer die Schnittstelle zur *MPI*-Bibliothek. Die öffentlichen Methoden der Klasse `ParallelProgrammingModel` sind:

- `ParallelProgrammingModel::ParallelProgrammingModel (int*, char ***)`
  Konstruktor: Initialisiert das parallele Programm und bestimmt für jeden Prozess die Prozessnummer `p` sowie die Anzahl aller parallelen Prozesse `N`. Als Argumente werden dem Konstruktor die Kommandozeilenparameter in einer verketteten Liste übergeben.
- `ParallelProgrammingModel::~ParallelProgrammingModel ()`
  Destruktor: Beendet das parallele Programm.
- `short ParallelProgrammingModel::proc ()`
  Gibt die Prozessnummer des aufrufenden Prozesses zurück.
- `short ParallelProgrammingModel::size ()`
  Gibt die Anzahl aller parallelen Prozesse zurück.
- `bool ParallelProgrammingModel::master ()`
  Gibt den Wahrheitswert `TRUE` zurück, wenn es sich bei dem aufrufenden Prozess um den Master-Prozess handelt. Der Master-Prozess hat dabei stets die Prozessnummer 0.

Ein Zeiger auf das parallele Programmiermodell wird in jedem parallelen Prozess unter der folgenden globalen Zeigervariablen gespeichert:

- ☐ `extern ParallelProgrammingModel* PPM`
  Zeiger auf das parallele Programmiermodell.

1.1.5 Das Hauptprogramm

```
1  // file:    PFEM/01/Main.C
2
```

```
3  #include "Parallel.h"
4
5  int main(int argv, char** argc)
6  {
7      PPM = new ParallelProgrammingModel(&argv,&argc);
8      pout << PPM->size() << " processors" << endl;
9      delete PPM;
10     return 0;
11 }
```

2 Ein paralleles Programmiermodell auf verteiltem Speicher

Drei Gründe für paralleles Rechnen

— Rechenleistung (flops: floating point operations per second)

— Speicher (z.B. Suchen in riesigen Datenmengen)

— Kosten (2 kleine Rechner sind billiger als ein großer)

Definition

a) Ein sequentielles Programm ist eine Folge von Anweisungen. Ein Prozessor arbeitet diese Anweisungen der Reihe nach ab.

b) Ein sequentieller Prozess ist die Abarbeitung eines Programms und hat jederzeit einen klar definierten Zustand.

c) Ein paralleles Programm ist eine Menge interagierender sequentieller Programme. In der Regel wird jeder parallele Prozess auf einem anderen Prozessor abgearbeitet (oder — zum Entwickeln — in kurzen Abständen abwechselnd auf einem Prozessor).

Wir betrachten nur das parallele Programmiermodell

MIMD = multiple instruction and multiple data

Ein paralleles Programm startet P verschiedene Prozesse

$$p \in \mathcal{P} = \{0, \ldots, P-1\}$$

(gleiche Bezeichnung für Prozess und Prozessnummer)

mit disjunktem Speicher (keine gemeinsamen Variablen), die im Wesentlichen dieselbe Befehlsfolge abarbeiten (bis auf wenige Abfragen nach der eigenen Prozessnummer).

Datenaustausch von Prozess p nach Prozess q erfolgt durch

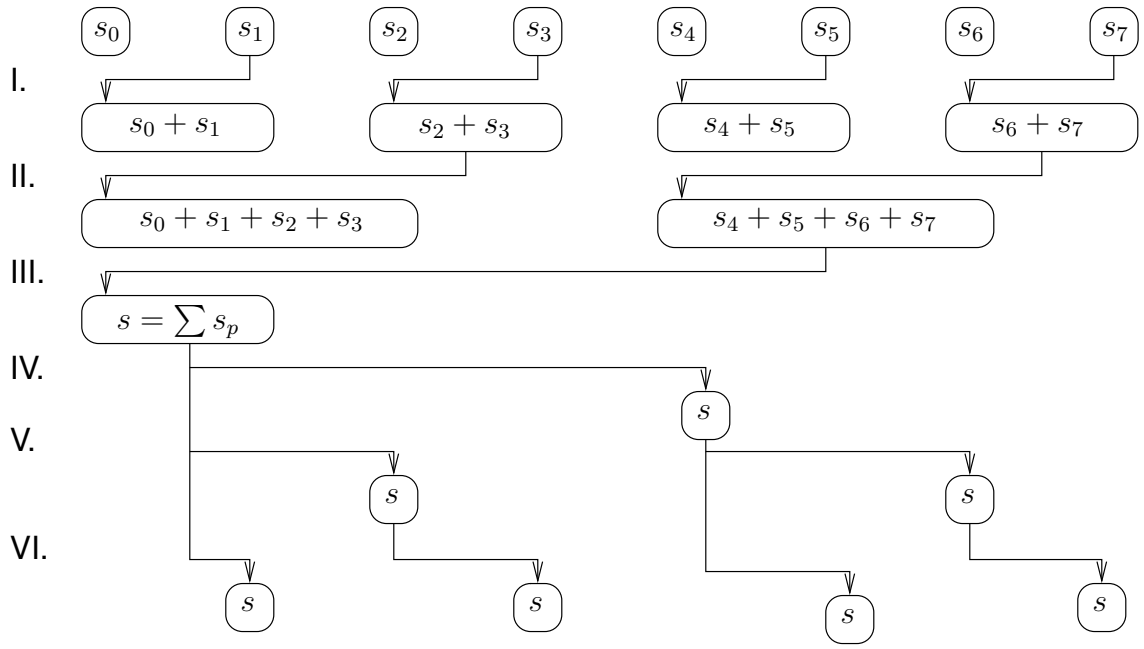
send (q, data) auf Prozess p

receive (p, data) auf Prozess q

Beispiel

Berechne $|\Omega| = \sum_{C \in \mathcal{C}} |\Omega_C| = \sum_{p \in \mathcal{P}} \sum_{C \in \mathcal{C}_p} |\Omega_C|$ mit $|\Omega_C| = \det(x_1 - x_0,\ x_2 - x_o) \qquad C = (x_0, x_1, x_2)$ Paralleles Programm: $s_p = |\Omega_p| = \sum_{C \in \mathcal{C}_p} |\Omega_C|$

$s =$ GlobalSum(s_p) Kommunikation von Global Sum für $P = 8 = 2^3$:



$\Longrightarrow$ 6 Kommunikationszykel

allgemein für $P = 2^L$:

GlobalSum (s_p) $\{s = s_q$ for $\ell = 1, \ldots, L$

if p mod $2^\ell = 2^{\ell-1}$ send $(p - 2^{\ell-1}, s)$

if p mod $2^\ell = 0$ receive $(p + 2^{\ell-1}, t)$

$s = s + t$

Broadcast (s)

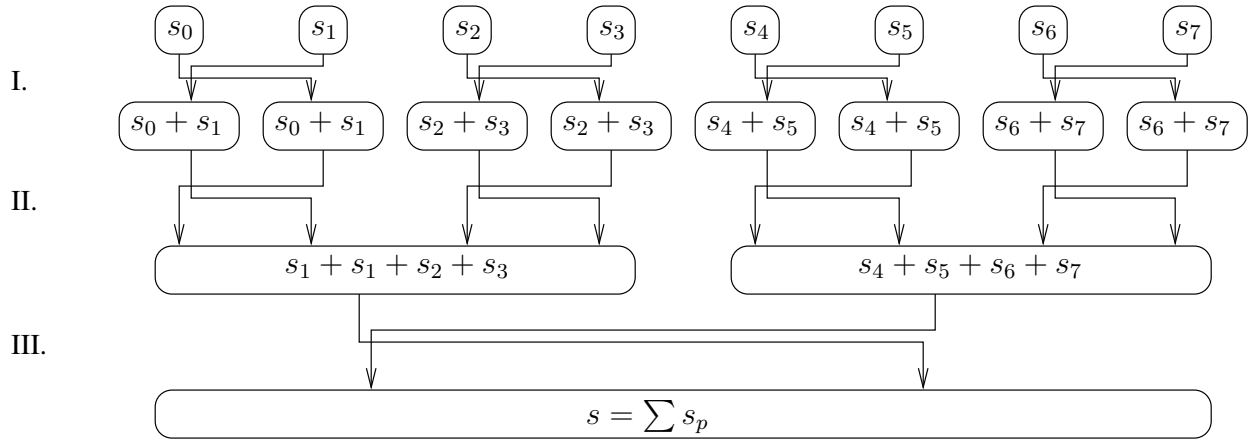
$\}$

Broadcast (s) for $\ell = L, \ldots, 1$

if $p \mod 2^\ell = 0$ send $(p + 2^{\ell-1}, s)$

if $p \mod 2^\ell = 2^{\ell-1}$ receive $(p - 2^{\ell-1}, s)$

Nicht optimal!



$\Longrightarrow$ 3 Kommunikationszykel

aber: – erfordert kreuzungsfreie Netzverbindung
– Ergebnis evtl. nicht identisch (Rundungsfehler, IEEE Arithmetik)

also: Verwende Global Sum / Broadcast einer parallelen Bibliothek
(hier: MPI = message passing interface), passend konfiguriert / compiliert
für die Zielmaschine.

Laufzeit $T_P(M) = \frac{M}{P} T_{\text{cal}} + \ln(P) \cdot T_{\text{com}}$

$P = |\mathcal{P}|$

$M = |\mathcal{C}|$

T_{cal} Rechenzeit pro C (Berechnung von $|\Omega_C|$)

T_{com} Kommunikationszeit für einen Datenaustausch

Definition

a) paralleler Speedup $S(M,P) = \dfrac{T_1(M)}{T_P(M)} \quad (\leq P)$

b) parallele Effizienz $E(M,P) = \dfrac{S(M,P)}{P} \quad (\leq 1)$

c) parallele Skalierbarkeit $D(M,P) = \dfrac{T_{2P}(2M)}{T_P(M)}$

d) Ein Algorithmus ist asymptotisch effizient, wenn für feste Prozessorzahl P gilt:

$$\lim_{M \to \infty} E(M,P) = 1$$

e) Ein Algorithmus heißt gut skalierbar, wenn für $M > M_0$ gilt:

$$\lim_{P\to\infty} D(PM, P) = 1$$

Beispiel Global Sum:

$$S(M,P) = \frac{M\,T_{\text{cal}}}{\frac{M}{P}T_{\text{cal}} + \ln P\,T_{\text{com}}} = \frac{P}{1 + \frac{P \ln P\,T_{\text{com}}}{M T_{\text{cal}}}} \to P \qquad \text{für } M \to \infty,\ P \text{ fest}$$

$$\Longrightarrow E(M,P) \to 1 \qquad \text{für } M \to \infty,\ P \text{ fest}$$

$$D(M,P) = \frac{\frac{2M}{2P}\,T_{\text{cal}} + \ln 2\,P\,T_{\text{com}}}{\frac{M}{P}T_{\text{cal}} + \ln P\,T_{\text{com}}} = 1 + \frac{\ln 2\,T_{\text{com}}}{\frac{M}{P}T_{\text{cal}} + \ln P\,T_{\text{com}}}$$

$$\Longrightarrow \lim_{P\to\infty} D(PM, P) = 1$$

aber: $\lim_{P\to\infty} E(PM,P) = \frac{1}{P}\,\frac{PM\,T_{\text{cal}}}{\frac{PM}{P}\,T_{\text{cal}} + \ln P\,T_{\text{com}}} = \frac{1}{1 + \frac{\ln P\,T_{\text{com}}}{M\,T_{\text{cal}}}} = 0$

Satz

a) Falls $D(M,P) \geq d > 1$ für $P > P_0$ und festes M, dann gilt $\lim_{P\to\infty} E(PM,P) = 0$
(genauer $\leq d^{\ln P}$)
also: Auch bei beliebig vielen Prozessoren und beliebiger Problemgröße wird die Effizienz immer schlechter (für $d < 2$ die Gesamtlaufzeit kleiner!).

b) Falls $E(PM,P) > 0$ für alle P, M fest
$\Longrightarrow \lim_{P\to\infty} D(PM,P) = 1$ gut skaliert
also: Wähle Verhältnis von M und P mit möglichst hoher Effizienz.

c) Wenn ein fester Anteil der Rechenzeit nicht parallelisierbar ist, d.h.

$$T_P(M) = \frac{T_{\text{par}}}{P} + T_{\text{seq}},$$

dann gilt $S(M,P) = \frac{P(T_{\text{par}}+T_{\text{seq}})}{T_{\text{par}}+P\,T_{\text{seq}}}$ (M fest)
$\Longrightarrow \lim_{P\to\infty} S(M,P) = \frac{T_{\text{par}}}{T_{\text{seq}}} + 1$

Beispiel

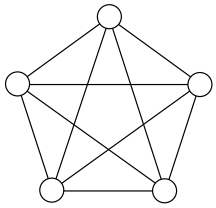
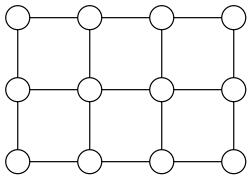
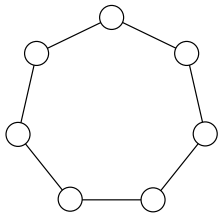
Wenn 10% des Algorithmus nicht parallelisierbar ist, dann lassen sich nicht mehr als 11 Prozesse effizient nutzen!

Die einzelne Kommunikation von m Byte von Prozessor p nach Prozessor q benötigt die Zeit

$$T(m) = T_{\text{init}} + m\,T_{\text{trans}}$$

mit T_{init} Aufsetzzeit (hier $\approx 1000 \cdot T_{\text{trans}}$)
T_{trans} Transferzeit (Bandbreite z.B. $1Gbit/s$)

$T(m)$ ist abhängig vom Protokoll (Ethernet / Myrinet) und der Netzwerktopologie:

	Volle Verbindung (Hyperebene)	$2d$-Feld	Ring
			
Knotengrad (Zahl der direkten Verbindungen pro. Prozessor)	$P-1$	4	2
Durchmesser (längste Verbindung von beliebigen Knoten)	1	$2(\sqrt{P}-1)$	$\frac{P}{2}$
Zahl der Verbindungen	$\binom{P}{2}$	$2P$	P
Bisektionsbreite	$2(P-2)$	$\sqrt{P}$	2

(Bisektionsbreite = minimale Zahl von Knoten, ohne die das Netzwerk in zwei Teile zerfällt; Maß für die Ausfallsicherheit eines Netzwerks)

Lokale Kommunikation

Neben Broadcast und Global Sum verwenden wir nur ein Modul zum Datenaustausch:

a) Auf jedem Prozessor $p \in \mathcal{P}$ sammle alle Daten, die an Prozessor $q \in \mathcal{P}$ geschickt werden, in einem Pufferfeld $B(p,q)$; bestimme die Puffergröße $S(p,q) = |B(p,q)|$ und die Anzahl der nicht–leeren Nachrichtenpuffer
$M(p) = |\{S(p,q) \neq 0 \colon q \in \mathcal{P}\}|$.
Bei lokaler Kommunikation gilt $M(p) = O(1)$.

b) Bestimme den (gewichteten) Kommunikationsgraphen

$$\mathcal{G} = \{(p,q) \colon |S(p,q)| \neq 0\} \colon$$

1) definiere $m \in \mathbf{N}^P$, $m = 0$ und setze $m(p) = M(p)$ auf Prozessor p

2) Global Sum von m ($\hat{=}$ Austausch von $m(p)$)

$$M = \sum_{p \in \mathcal{P}} m(p),\ n(p) = \sum_{q=0}^{P-1} m(q)\ (n(0) = 0)$$

3) definiere $d, s \in \mathbf{R}^M$, $d, s = 0$ und setze auf Prozessor p

$$\begin{aligned} \{q \in \mathcal{P} \colon |s(p,q)| \neq 0\} = &\quad \{q_0, \ldots, q_{m(p)-1}\} \\ d((n(p)+j) = q_j &\quad j = 0, \ldots, m(p)-1 \\ s(n(p)+j) = S(p, q_j) & \end{aligned}$$

Global Sum von d, s

Bei lokaler Kommunikation gilt $M = 0(P)$.

c) Nun kennt jeder Prozessor p den gesamten Kommunikationsgraphen:

$$\begin{aligned} \mathcal{G} = &\quad \{(p, d(n(p)+j)) \colon p \in \mathcal{P},\ j = 0, \ldots, m(p)-1\} \\ &\quad \text{mit Gewichten} \quad s(n(p)+j) \end{aligned}$$

Datenaustausch auf Prozessor p:
sende $B(p,q)$ nach $q = d(n(p)+j) \quad j = 0, \ldots, m(p)-1$
empfange $B(q,p)$ der Größe $s(n(q)+j)$ von Prozessor p mit $q = d(n(p)+j)$

Problem

Wenn gleichzeitig

Prozessor 0 sendet $B(0,1)$ an Prozessor 1
Prozessor 1 sendet $B(1,0)$ an Prozessor 0

dann warten beide, bis die Nachricht empfangen wird; da sie aber beide senden, kann keiner empfangen.

$\Longrightarrow$ Verklemmung (dead lock)

Lösung 1:	Prozessor 0	Prozessor 1
	sende $B(0,1)$	empfange $B(0,1)$
	empfange $B(1,0)$	sende $B(1,0)$
Lösung 2:	asynchrone Kommunikation	asend, arecv, wait

Asynchrone Kommunikation

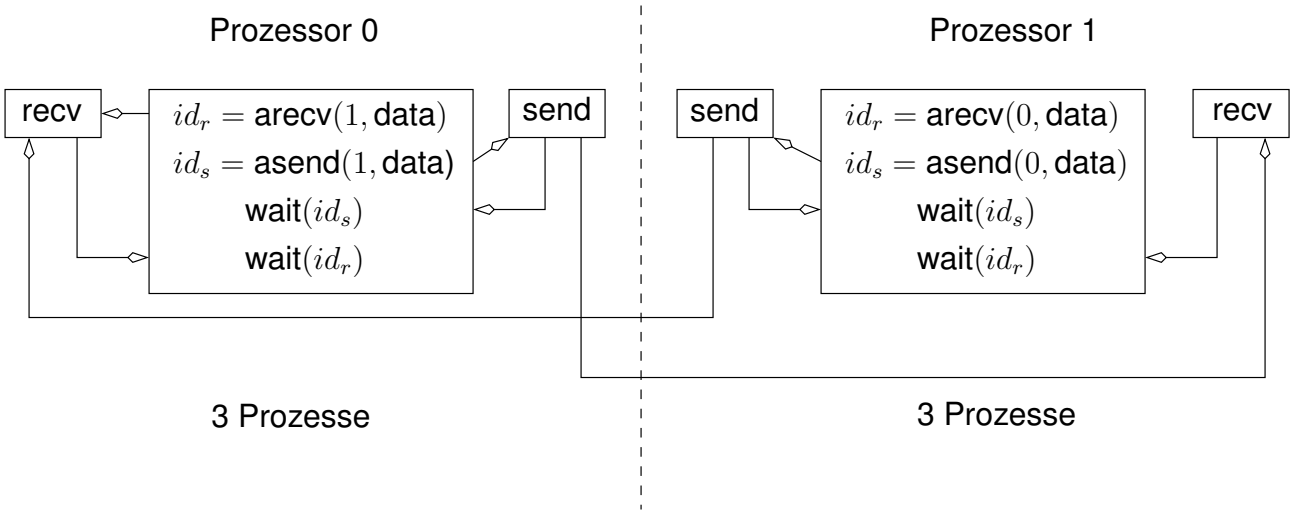
id = asend (q, data)
startet auf demselben Prozessor einen eigenen Prozess und übergibt ihm (die Adresse von) data. Das Programm läuft weiter, während der zweite Prozess die Daten verschickt.

id = arecv (p, data)
startet auf demselben Prozessor einen eigenen Prozess und übergibt ihm die Adresse von data. Das Programm läuft weiter, kann aber data noch nicht verwenden.

wait (id)
das Programm bleibt stehen und wartet auf ein Signal, dass der Prozess id erfolgreich beendet wurde, d.h. data verschickt bzw. empfangen wurde.

Vorsicht: Von data werden nur die Adressen verwendet, das Feld kann erst nach wait wieder verwendet werden!

Beispiel



2.1 Paralleles Programm 2

2.1.1 Beschreibung

In der zweiten Version des Programmes wird ein Netz aus Dreieckszellen eingelesen und an die beteiligten Prozessoren verteilt. Jeder Prozessor gibt die eigenen Zellen im Standardoutput aus und ermittelt die Fläche seines Netzes. Die Fläche des Gesamtnetzes wird errechnet, an alle Prozesse übermittelt und jeweils ausgegeben.

2.1.2 Dateien

```
PFEM/02/Parallel.h
PFEM/02/Parallel.C
PFEM/02/Main.C
PFEM/02/Makefile
PFEM/02/mesh
```

2.1.3 Die globale Kommunikation

```
1  // file:    PFEM/02/Parallel.h
2
3  #ifndef _PARALLEL_H_
4  #define _PARALLEL_H_
5
6  #include <iostream>
7  using namespace std;
8
9  class ParallelProgrammingModel {
10     int p;
11     int N;
12  public:
13     short proc () const { ... }
14     short size () const { ... }
15     bool master () const { ... }
16     void Broadcast (void*, int);
17     void Sum (int*, size_t);
18     void Sum (double*, size_t);
19     template <class C> void Broadcast(C& c) { Broadcast(&c,sizeof(c));}
20     template <class C> C Sum (C a) { Sum(&a,1); return a; }
21     ParallelProgrammingModel (int*, char ***);
22     ~ParallelProgrammingModel ();
23 };
24 extern ParallelProgrammingModel* PPM;
25
26 #define pout ...
27 #endif
```

```
1  // file:    PFEM/02/Parallel.C
2
3  #include "Parallel.h"
4
5  #include "mpi.h"
6
7  ParallelProgrammingModel* PPM = 0;
```

```
8
9  ParallelProgrammingModel::ParallelProgrammingModel (int* argc, char ***argv) { ... }
10
11 ParallelProgrammingModel::~ParallelProgrammingModel () { ... }
12
13 void ParallelProgrammingModel::Broadcast (void *data, int size) {
14     MPI_Bcast (data,size,MPI_BYTE,0,MPI_COMM_WORLD);
15 }
16 void ParallelProgrammingModel::Sum (int* a, size_t n) {
17     int* b = new int [n];
18     MPI_Allreduce(a,b,n,MPI_INT,MPI_SUM,MPI_COMM_WORLD);
19     memcpy(a,b,sizeof(int)*n);
20     delete[] b;
21 }
22 void ParallelProgrammingModel::Sum (double* a, size_t n) {
23     double* b = new double [n];
24     MPI_Allreduce(a,b,n,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
25     memcpy(a,b,sizeof(double)*n);
26     delete[] b;
27 }
```

Die Globale Kommunikation wird über die Klasse **ParallelProgrammingModel** realisiert, die in dieser zweiten Version des parallelen Programms entsprechend erweitert wurde. Die neuen Methoden der Klasse sind:

- `void ParallelProgrammingModel::Broadcast (void *data, int size)`
  Die Methode `Broadcast` versendet Daten an alle parallelen Prozesse. Das Argumente der Methode ist ein untypisierter Zeiger `data` auf die erste Speicheradresse, welche die zu versendenden Daten enthält. Das zweite Argument `size` gibt die Länge der zu versendenden Daten in Bytes an. Für das eigentliche Versenden der Daten ruft die Methode die MPI-Funktion `MPI_Bcast` auf. Nach dem Methodenaufruf enthalten alle Variablen, auf die der Zeiger `data` zeigt, den Wert der Variablen im Master-Prozess. Um beispielsweise den Wert einer Variablen `v` zu versenden, kann man `PPM->Broadcast(&v,sizeof(v));` aufrufen. Anstelle dieser Methode sollte jedoch stets die nachfolgend beschriebene Template-Methode `template <class C> void Broadcast(C& c)` verwendet werden.

- `template <class C> void ParallelProgrammingModel::Broadcast(C& c)`
  Mit dieser Template-Methode können Daten beliebigen Typs an alle parallelen Prozesse versendet werden. Das Argument der Methode ist ein Zeiger `c` auf die Variable, deren Wert versendet werden soll. Nach dem Aufruf enthalten alle Variablen, auf die der Zeiger `c` zeigt, den Wert der Variablen im Master-Prozess. Programmierer sollten immer diese Methode anstelle von `void Broadcast (void *data, int size)` verwenden (siehe oben).

- `void ParallelProgrammingModel::Sum (int* a, size_t n)`
  `void ParallelProgrammingModel::Sum (double* a, size_t n)`

Die beiden `Sum`-Methoden berechnen die globale Summe von `int`- bzw. `double`-wertigen Daten. Das erste Argument der Methode ist ein Zeiger `a` auf die erste Speicheradresse der zu summierenden Daten. Das zweite Argument `size` gibt die Anzahl der zu summierenden `int`- bzw. `double`-Werte an. Ist `arr` beispielsweise ein `int`-Array mit 4 Elementen, so wird mittels `PPM->Sum(arr,4);` elementweise die globale Summe von `arr` berechnet. Die Methode verwendet dazu die MPI-Funktion `MPI_Allreduce`. Nach dem Methodenaufruf enthalten alle Variablen, auf die der Zeiger `a` zeigt, die Summe der Variablenwerte in den einzelnen Prozessen. Zur Berechnung globaler Summen sollten diese beiden Methoden jedoch nicht direkt aufgerufen werden. Dazu ist die Template-Methode `template <class C> C Sum (C a)` vorgesehen (siehe unten).

- `template <class C> C ParallelProgrammingModel::Sum (C a)`
  Mit dieser Template-Methode kann die globale Summe einer `int`- oder `double`-wertigen Variablen berechnet werden. Dazu übergibt man der Methode die Variable `a`, deren Summe berechnet werden soll. Die Methode gibt dann die globale Summe zurück. Programmierer sollten immer diese Methode anstelle von `void Sum (int* a, size_t n)` oder `void Sum (double* a, size_t n)` verwenden (siehe oben).

2.1.4 Punkte und Dreiecke (Teil 1)

```
1  // file:    PFEM/02/Main.C
2
3  #include "Parallel.h"
4
5  #include <fstream>
6  using namespace std;
7
8  class Point {
9      double x[2];
10 public:
11     Point () {};
12     Point (const Point& y) { x[0] = y.x[0]; x[1] = y.x[1]; }
13     double operator [] (int i) const {return x[i]; }
14     Point& operator -= (const Point& y) { x[0]-=y[0];x[1]-=y[1]; return *this;}
15     friend istream& operator >> (istream& s, Point& y) {
16         double a,b;  s >> a >> b;
17         y.x[0] = a;  y.x[1] = b;
18         return s;
19     }
20 };
21 inline Point operator - (const Point& x, const Point& y) {
22     Point z = x; return z -= y;
23 }
```

```
24 inline double det(const Point& x,const Point& y) { return x[0]*y[1]-x[1]*y[0];}
25 inline ostream& operator << (ostream& s, const Point& x) {
26     return s << x[0] << " " << x[1];
27 }
28
29 class Cell {
30     Point x[3];
31 public:
32     Cell () {}
33     double area () const { return 0.5 * det(x[1]-x[0],x[2]-x[0]); }
34     const Point& operator [] (int i) const { return x[i]; }
35     friend istream& operator >> (istream& s, Cell& C) {
36         return s >> C.x[0] >> C.x[1] >> C.x[2];
37     }
38 };
39 inline ostream& operator << (ostream& s, const Cell& C) {
40     return s << "(" << C[0] << " | " << C[1] << " | " << C[2] << ")" << endl;
41 }
42
43 int main(int argv, char** argc)
44 {
45     PPM = new ParallelProgrammingModel(&argv,&argc);
46     int M;
47     double A = 0;
48     if (PPM->master()) {
49     ifstream s("mesh");
50     s >> M;
51     PPM->Broadcast(M);
52     for (int m=0; m<M; ++m) {
53         Cell C; s >> C; PPM->Broadcast(C);
54         if (m%PPM->size() == PPM->proc()) pout << C;
55         if (m%PPM->size() == PPM->proc()) A += C.area();
56     }
57     }
58     else {
59     PPM->Broadcast(M);
60     for (int m=0; m<M; ++m) {
61         Cell C; PPM->Broadcast(C);
62         if (m%PPM->size() == PPM->proc()) pout << C;
63         if (m%PPM->size() == PPM->proc()) A += C.area();
64     }
65     }
66     A = PPM->Sum(A); pout << "area " << A << endl;
67     delete PPM;
68     return 0;
69 }
```

Die Klasse **Point** repräsentiert einen Punkt in einem zweidimensionalen, karthesischen Koordinatensystem, der mit seinem Ortsvektor identifiziert werden kann. Für die Klasse `Point` sind folgende Methoden, Operatoren und Funktionen definiert:

- `Point::Point ()`
  Konstruktor: Erzeugt eine neue Instanz der Klasse `Point`. Die Koordinaten des so erzeugten Punktes sind dabei nicht definiert.

- `Point::Point (const Point& y)`
  Copy-Konstruktor: Erzeugt einen neuen Punkt mit denselben Koordinaten eines beeits vorhandenen Punktes `y`.

- `double Point::operator [] (int i)`
  Ist `x` ein `Point`-Objekt, so geben die Aufrufe von `x[0]` und `x[1]` die erste und die zweite Koordinate des Punktes `x` zurück.

- `Point& Point::operator -= (const Point& y)`
  Zuweisungsoperator: Sind `x` und `y` zwei Punkte, so führt der Befehl `x -= y` die Zuweisung `x = x-y` aus.

- `friend istream& Point::operator >> (istream& s, Point& y)`
  Eingabeoperator: Liest die Koordinaten eines Punktes `y` aus einem Eingabestrom `s` ein.

- `inline Point operator - (const Point& x, const Point& y)`
  Gibt die Differenz zweier Punkte `x` und `y` im Sinne der Ortsvektoren zurück.

- `inline double det(const Point& x,const Point& y)`
  Gibt die Determinante folgender Matrix zurück:

$$\begin{pmatrix} x[0] & y[0] \\ x[1] & y[1] \end{pmatrix}$$

- `inline ostream& operator << (ostream& s, const Point& x)`
  Ausgabeoperator: Gibt die Koordinaten eines Punktes `x` in einem Ausgabestrom `s` aus.

Die Klasse **Cell** repräsentiert ein Dreieck im zweidimensionalen euklidischen Raum, welches mit seinen Eckpunkten identifiziert wird. Die Eckpunkte eines `Cell`-Objektes werden von `Point`-Objekten repräsentiert. Für die Klasse `Cell` sind folgende Methoden und Operatoren definiert:

- `Cell::Cell ()`
  Konstruktor: Erzeugt ein neues Objekt der Klasse `Cell`. Die Eckpunkte des so erzeugten Dreiecks sind dabei undefiniert.

- `const Point& Cell::operator [] (int i)`
  Ist `C` ein `Cell`-Objekt, dann geben die Aufrufe von `C[0]`, `C[1]` und `C[2]` eine Referenz auf den ersten, den zweiten und den dritten Eckpunkt des Dreiecks `C` zurück.

- `double Cell::area ()`
  Gibt den Flächeninhalt des Dreiecks zurück.

- `friend istream& operator >> (istream& s, Cell& C)`
  Eingabeoperator: Liest die Koordinaten der drei Eckpunkte von `C` nacheinander aus einem Eingabestrom `s` ein.

- `inline ostream& operator << (ostream& s, const Cell& C)`
  Ausgabeoperator: Gibt die Koordinaten der drei Eckpunkte eines Dreiecks `C` in einen Ausgabestrom `s` aus.

2.1.5 Das Hauptprogramm

2.2 Paralleles Programm 3

2.2.1 Beschreibung

Die dritte Programmversion entwickelt die zweite Version weiter, indem jetzt zur Kommunikation anstelle der Broadcast-Funktion die Klasse **Exchangebuffer** verwendet wird. Wiederum gibt jeder Prozessor die eigenen Zellen und die Fläche des Gesamtnetzes aus.

2.2.2 Dateien

```
PFEM/03/Parallel.h
PFEM/03/Parallel.C
PFEM/03/Mesh.h
PFEM/03/Main.C
PFEM/03/Makefile
PFEM/03/mesh
```

Die Datei `Mesh.h` enthält in dieser Version des parallelen Programms die Definitionen der Klassen `Point` und `Cell`.

2.2.3 Die lokale Kommunikation

```
1  // file:    PFEM/03/Parallel.h
2
3  #ifndef _PARALLEL_H_
4  #define _PARALLEL_H_
5
6  #include <iostream>
7  #include <fstream>
8  #include <vector>
9  using namespace std;
10
11 class ExchangeBuffer;
12
13 class ParallelProgrammingModel {
14     int p;
15     int N;
16  public:
17     short proc () const { ... }
18     short size () const { ... }
19     bool master () const { ... }
20     void Broadcast (void*, int);
21     void Sum (int*, size_t);
22     void Sum (double*, size_t);
23     template <class C> void Broadcast(C& c) { ... }
24     template <class C> C Sum (C a) { ... }
25     void Communicate (ExchangeBuffer&);
26     ParallelProgrammingModel (int*, char ***);
27     ~ParallelProgrammingModel ();
28 };
29 extern ParallelProgrammingModel* PPM;
30
31 class Exchange {
32  protected:
33     int* n;
34     int* d;
35     int* s;
36     int n_send;
37     int n_recv;
38  public:
39     Exchange () {
40         n = new int [PPM->size()+1];
41         s = new int [PPM->size()];
42         for (int q=0; q<PPM->size(); ++q) s[q] = 0;
43         d = 0;
44     }
45     ~Exchange () {
46         delete[] n;
```

```
47          if (d!=0) delete[] d;
48          delete[] s;
49      }
50      void SendSize (int m, short q) { s[q] = m; }
51      size_t ReceiveSize (short q) const {
52          for (int k=n[q]; k<n[q+1]; ++k)
53              if (d[k] == PPM->proc()) return s[k];
54          return 0;
55      }
56      int SendMessages () const { return n_send; }
57      int RecvMessages () const { return n_recv; }
58      int Messages (int q) const { return n[q]; }
59      int MessageDest (int k) const { return d[k]; }
60      int MessageSize (int k) const { return s[k]; }
61      void CommunicateSize () {
62          for (int q=0; q<PPM->size(); ++q) n[q] = 0;
63          for (int q=0; q<PPM->size(); ++q) if (s[q]) ++(n[PPM->proc()]);
64          PPM->Sum(n,PPM->size()+1);
65          for (int q=PPM->size(); q>0; --q) n[q] = n[q-1];
66          n[0] = 0;
67          for (int q=0; q<PPM->size(); ++q) n[q+1] += n[q];
68          d = new int [n[PPM->size()]];
69          for (int k=0; k<n[PPM->size()]; ++k) d[k] = 0;
70          int* tmp = new int [n[PPM->size()]];
71          for (int k=0; k<n[PPM->size()]; ++k) tmp[k] = 0;
72          int k = n[PPM->proc()];
73          for (int q=0; q<PPM->size(); ++q)
74              if (s[q]) {
75                  tmp[k] = s[q];
76                  d[k++] = q;
77              }
78          delete[] s;
79          s = tmp;
80          PPM->Sum(s,n[PPM->size()]);
81          PPM->Sum(d,n[PPM->size()]);
82          n_send = n[PPM->proc()+1] - n[PPM->proc()];
83          n_recv = 0;
84          for (int k=0; k<n[PPM->size()]; ++k)
85              if (d[k] == PPM->proc()) ++n_recv;
86      }
87  };
88
89  inline ostream& operator << (ostream& s, const Exchange& E) {
90      s << " on " << PPM->proc() << " :"
91        << "send " << E.SendMessages() << " : ";
92      for (int k=E.Messages(PPM->proc()); k<E.Messages(PPM->proc()+1); ++k)
93      s << E.MessageDest(k) << "|" << E.MessageSize(k) << " ";
94      s << "recv " << E.RecvMessages() << " : ";
```

```
95      for (int q=0; q<PPM->size(); ++q)
96      for (int k=E.Messages(q); k<E.Messages(q+1); ++k)
97          if (E.MessageDest(k) == PPM->proc())
98          s << q << "|" << E.MessageSize(k) << " ";
99      return s << endl;
100 }
101
102 const size_t BufferSize = 128000;
103
104 class Buffer {
105     char* b;
106     char* p;
107     size_t n;
108  public:
109     Buffer (size_t m = 0) : n(m) { if (m) b = new char [m]; else b=0; p=b; }
110     ~Buffer () { delete[] b; }
111     void rewind () { p = b; }
112     size_t size () const { return size_t(p-b); }
113     size_t Size () const { return n; }
114     char* operator () () { return b; }
115     void resize (size_t m) {
116         char* tmp = new char [m];
117         memcpy(tmp,b,size());
118         p = tmp + size();
119         delete[] b;
120         b = tmp;
121         n = m;
122     }
123     template <class C> Buffer& operator << (const C& c) {
124         size_t m = sizeof(c);
125         while (size()+m>n) resize(n+BufferSize);
126         memcpy(p,&c,m);
127         p += m;
128         return *this;
129     }
130     template <class C> Buffer& operator >> (C& c) {
131         size_t m = sizeof(c);
132         memcpy(&c,p,m);
133         p += m;
134         return *this;
135     }
136 };
137
138 class ExchangeBuffer : public Exchange {
139     vector<Buffer> SendBuffers;
140     vector<Buffer> ReceiveBuffers;
141  public:
142     Buffer& Send (short q) { return SendBuffers[q]; }
```

```
143     Buffer& Receive (short q) { return ReceiveBuffers[q]; }
144     ExchangeBuffer() : SendBuffers(PPM->size()),
145         ReceiveBuffers(PPM->size()) {}
146     void Communicate () { PPM->Communicate(*this); }
147     void CommunicateSizeBuffer () {
148         for (short q=0; q<PPM->size(); ++q)
149             SendSize(Send(q).size(),q);
150         CommunicateSize();
151         for (short q=0; q<PPM->size(); ++q)
152             Receive(q).resize(ReceiveSize(q));
153         PPM->Communicate(*this);
154     }
155 };
156
157 #define pout cout << PPM->proc() << ": "
158 #define mout if (PPM->master()) cout
159
160 #endif
```

```
1  // file:    PFEM/03/Parallel.C
2
3  #include "Parallel.h"
4
5  #include "mpi.h"
6
7  ParallelProgrammingModel* PPM = 0;
8
9  ParallelProgrammingModel::ParallelProgrammingModel (int* argc, char ***argv) { ... }
10
11 ParallelProgrammingModel::~ParallelProgrammingModel () { ... }
12
13 void ParallelProgrammingModel::Broadcast (void *data, int size) { ... }
14
15 void ParallelProgrammingModel::Sum (int* a, size_t n) { ... }
16
17 void ParallelProgrammingModel::Sum (double* a, size_t n) { ... }
18
19 void ParallelProgrammingModel::Communicate (ExchangeBuffer& E) {
20     const int tag = 27;
21     MPI_Request* req_send = new MPI_Request [E.SendMessages()];
22     MPI_Request* req_recv = new MPI_Request [E.RecvMessages()];
23     MPI_Request* r = req_recv;
24     for (int q=0; q<PPM->size(); ++q)
25         for (int k=E.Messages(q); k<E.Messages(q+1); ++k)
26             if (E.MessageDest(k) == PPM->proc())
27                 MPI_Irecv(E.Receive(q)(),E.MessageSize(k),MPI_BYTE,
28                     q,tag,MPI_COMM_WORLD,r++);
29     r = req_send;
```

```
30      for (int k=E.Messages(PPM->proc()); k<E.Messages(PPM->proc()+1); ++k)
31          MPI_Isend(E.Send(E.MessageDest(k))(),E.MessageSize(k),MPI_BYTE,
32              E.MessageDest(k),tag,MPI_COMM_WORLD,r++);
33      MPI_Status st;
34      r = req_send;
35      for (int k=0; k<E.SendMessages(); ++k) MPI_Wait(r++,&st);
36      r = req_recv;
37      for (int k=0; k<E.RecvMessages(); ++k) MPI_Wait(r++,&st);
38      delete[] req_send;
39      delete[] req_recv;
40  }
```

Die Klasse **Exchange** dient dazu, einen lokalen Kommunikationszyklus zu koordinieren, d.h. sie kapselt den Kommunikationsgraphen. Für die Klasse `Exchange` sind folgende Methoden und Operatoren definiert:

- `Exchange::Exchange ()`
  Konstruktor: Erzeugt eine neue Instanz der Klasse `Exchange`. Intern werden die privaten Variablen `s` und `n` angelegt. Bevor die Methode `CommunicateSize` aufgerufen wird (siehe unten), gilt für das Array `s` folgende Semantik: Für eine Prozessnummer `q` enthält `s[q]` die Anzahl von Bytes, die vom aufrufenden Prozess an den Prozess `q` gesendet werden sollen.

- `Exchange::~Exchange ()`
  Destruktor: Gibt den von der Instanz belegten Speicherplatz frei.

- `void Exchange::SendSize (int m, short q)`
  Durch den Aufruf dieser Methode teilt der Programmierer `Exchange`-Objekt mit, dass an den Prozess mit der Prozessnummer `q` Daten der Länge `m` Bytes gesendet werden sollen. Die Methode darf nach einem Aufruf von `CommunicateSize` (siehe unten) nicht mehr verwendet werden.

- `void Exchange::CommunicateSize ()`
  Diese Methode wird aufgerufen, wenn jeder Prozess dem `Exchange`-Objekt mittels `void SendSize (int m, short q)` mitgeteilt hat, wie viele Daten er an einen anderen Prozess senden will. Die Methode `CommunicateSize` kommuniziert diese Sendeabsichten allen Prozessen und definiert eine Sendereihenfolge. Dazu wird die Semantik der privaten Klassenvariable `s` geändert.
  Zunächst setzt die Methode in der privaten Array-Variable `n` den Wert `n[q]` auf 1, wenn der aufrufende Prozess Daten an den Prozess `q` versenden will. Anschließend wird die globale Summe dieses Arrays gebildet. Der Wert von `n[q]` gibt dann die Anzahl der Nachrichten an, die an einen Prozess `q` gesendet werden sollen (64–66).
  Die Methode führt anschließend eine Nummerierung der zu versendenden Nachrichten durch. Dazu wird das Array `n` wie folgt verändert: Der Wert von `n[P]` gibt

die Anzahl aller zu versendenden Nachrichten an, wobei `P` die Anzahl aller parallelen Prozesse ist. Für jede Prozessnummer `q` gibt `n[q]` die Nummer der ersten Nachricht an, die vom Prozess `q` versendet wird. Der Prozess `q` versendet also die Nachrichten mit den Nummern von `n[q]` bis `n[q+1]-1` (68–69).
Nachdem die Nummerierung der Nachrichten vorgenommen wurde, legt die Methode die private Array-Variable `d` an und überschreibt `s`. Für eine Nachricht mit der Nummer `k` gibt `d[k]` die Nummer des empfangenden Prozesses an und `s[k]` die Länge der Nachricht in Bytes. Diese Informationen werden global kommuniziert. (70–83)
In jedem Prozess wird anschließend die Anzahl der zu versendenden Nachichten `n_send` und die Anzahl der zu empfangenden Nachrichten `n_recv` bestimmt (84–87)

Die folgenden Methoden der Klasse `Exchange` *dürfen nur dann verwendet werden, wenn die Methode* `CommunicateSize` *bereits aufgerufen wurde*.

- `size_t Exchange::ReceiveSize (short q)`
  Gibt die Länge der Daten in Bytes zurück, die der aufrufenden Prozess von einem Prozess mit der Nummer `q` empfängt.

- `int Exchange::SendMessages ()`
  Gibt die Anzahl der vom aufrufenden Prozess zu versendenden Nachrichten zurück.

- `int Exchange::RecvMessages ()`
  Gibt die Anzahl der vom aufrufenden Prozess zu empfangenden Nachrichten zurück.

- `int Exchange::Messages (int q)`
  Gibt die Nummer der ersten Nachricht zurück, die vom Prozess `q` versendet wird.

- `int Exchange::MessageDest (int k)`
  Gibt die Prozessnummer desjenigen Prozesses zurück, der die Nachricht mit der Nummer `k` empfängt.

- `int Exchange::MessageSize (int k)`
  Gibt die Länger der Nachricht mit der Nummer `k` in Bytes zurück.

- `inline ostream& operator << (ostream& s, const Exchange& E)`
  Ausgabeoperator: Schreibt die Daten eines `Exchange`-Objektes `E` in einen Ausgabestrom `s`.

Die Klasse **Buffer** definiert einen Datenpuffer, in dem die zu versendenden und zu empfangenden Daten eines Prozesses gespeichert werden können. Für die Klasse `Buffer` sind folgende Methoden, Operatoren und Konstanten definiert:

- `Buffer::Buffer (size_t m = 0)`
  Konstruktor: Erzeugt ein neues Objekt der Klasse `Buffer`. Intern wird Speicherplatz der Länge `m` Bytes allokiert. Die privaten Zeigervariablen `b` und `p` zeigen dabei auf die erste Speicheradresse des allokierten Speichers.

- `Buffer::~Buffer ()`
  Destruktor: Gibt den von der Instanz belegten Speicherplatz frei.

- `char* Buffer::operator () ()`
  Der Operator `()` gibt den Zeiger auf die erste Speicheradresse des vom Puffer verwalteten Speicherbereiches zurück.

- `void Buffer::resize (size_t m)`
  Die Methode `resize` allokiert einen Speicherbereich der Länge `m` Bytes und kopiert den Inhalt des Puffers in diesen neuen Speicherbereich. Der zuvor belegt Speicherbereich wird frei gegeben. Diese Methode wird vom Operator >> verwendet, um den Puffer bei Bedarf zu vergrößern.

- `template <class C> Buffer& Buffer::operator << (const C& c)`
  Mit dem Template-Operator << können Daten beliebigen Typs auf den Puffer geschrieben werden. Die Daten werden an der Stelle in den Speicherbereich eingefügt, auf die der private Zeiger `p` zeigt. Der Zeigen `p` wird daraufhin an das Ende der eingefügten Daten gesetzt. Ist der vom Puffer verwaltete Speicherbereich zu klein, um die neuen Daten aufzunehmen, wird dieser mittels `resize` so lange um `BufferSize` Bytes erweitert (siehe unten), bis die neuen Daten aufgenommen werden können.

- `template <class C> Buffer& Buffer::operator >> (C& c)`
  Mit dem Template-Operator >> werden Daten beliebigen Typs vom Puffer gelesen. Die private Zeigervariable `p` zeigt dabei auf die Speicheradresse, von der die Daten ausgelesen werden. Der Zeiger `p` wird danach an das Ende des ausgelesenen Speicherbereiches gesetzt.

- `void Buffer::rewind ()`
  Die Methode setzt die private Zeigervariable `p` zurück auf die erste Speicheradresse des vom Puffer verwalteten Speichers.

  Bei der Verwendung der Operatoren << und >> muss beachtet werden, dass beide Operatoren den Speicherbereich des Puffers von vorne nach hinten einbeziehungsweise auslesen.

- `size_t Buffer::size ()`
  Gibt die Länge der Daten, die sich im Puffer befinden, in Bytes zurück.

- `size_t Buffer::Size ()`
  Gibt die Länge des vom Puffer verwalteten Speicherbereiches zurück.

- `const size_t BufferSize = 128000`
  Gibt die Anzahl von Bytes an, um die der Speicherbereich eines `Buffer`-Objektes erweitert wird, falls dieses neue Daten nicht mehr aufnehmen kann.

Die Klasse **ExchangeBuffer** ist einer Erweiterung der Klasse `Exchange`, die nicht nur den Ablauf der lokalen Kommunikation organisieren, sondern auch die zu versendenden und zu empfangenden Daten verwalten kann. Dazu dienen im wesentlichen `Buffer`-Objekte sowie eine Reihe weiterer Methoden. Die Klasse übernimmt damit die Steuerung jeweils eines lokalen Kommunikationszyklus. Die Klasse `ExchangeBuffer` besitzt die folgenden Methoden:

- `ExchangeBuffer::ExchangeBuffer()`
  Konstruktor: Erzeugt ein neues Objekt der Klasse `ExchangeBuffer`. Intern werden die privaten Variablen `SendBuffers` und `ReceiveBuffers` als `Buffer`-wertige Vektoren angelegt.

- `Buffer& ExchangeBuffer::Send (short q)`
  Die Methode gibt für eine Prozessnummer `q` eine Referenz auf den Puffer zurück, der die vom Prozess `q` versendeten Daten speichert.

- `Buffer& ExchangeBuffer::Receive (short q)`
  Die Methode gibt für eine Prozessnummer `q` eine Referenz auf den Puffer zurück, der die Daten speichert, die vom Prozess `q` empfangen werden. Diese Methode sollte vom Programmierer jedoch erst dann aufgerufen werden, wenn zuvor die Methode `CommunicateSizeBuffer` aufgerufen wurde (siehe unten).

- `void ExchangeBuffer::Communicate ()`
  Ruft die Methode für das Versenden der Daten in den Sendepuffern im globalen `ParallelenProgrammingModel`-Objekt `PPM` auf. Um einen lokalen Kommunikationszyklus durchzuführen, sollte jedoch anstatt dieser Methode immer die Methode `CommunicateSizeBuffer` aufgerufen werden.

- `void ExchangeBuffer::CommunicateSizeBuffer ()`
  Diese Methode führt einen lokalen Kommunikationszyklus aus. Dazu wird zunächst mittels `CommunicateSize` der Kommunikationsgraph erstellt, anschließend werden die gepufferten Daten an die entsprechenden Prozesse versendet. Das Versenden der Daten wird vom globalen `ParallelenProgrammingModel`-Objekt `PPM` durchgeführt.

Die eigentliche lokale Kommunikation, also das Versenden und Empfangen von Daten, zwischen den einzelnen parallelen Prozessen wird von MPI durchgeführt. Die Schnittstelle zu MPI für den Programmierer ist die Klasse **ParallelProgrammingModel**, die zu diesem Zweck um die folgende Methode erweitert wurde:

- `void ParallelProgrammingModel::Communicate (ExchangeBuffer& E)`
  Diese Methode führt das Versenden und Empfangen von Daten unter den einzelnen parallelen Prozessen aus. Dazu werden die MPI-Funktionen `MPI_Isend`, `MPI_Irecv` und `MPI_Wait` verwendet, mit denen eine asynchrone Kommunikation realisiert werden kann. Das erste Argument der Funktion ist eine Referenz auf ein `ExchangeBuffer`- Objekt, in dem der Kommunikationsgraph und die zu versendenden Daten gespeichert sind.

2.2.4 Das Hauptprogramm

```
1  // file:    PFEM/03/Main.C
2
3  #include "Mesh.h"
4
5  int main (int argv, char** argc)
6  {
7      PPM = new ParallelProgrammingModel(&argv,&argc);
8      int M;
9      ExchangeBuffer E;
10     if (PPM->master()) {
11         ifstream s("mesh");
12         s >> M;
13         for (int m=0; m<M; ++m) {
14             Cell C; s >> C;
15             int dest = m % PPM->size();
16             E.Send(dest) << C;
17         }
18     }
19     E.CommunicateSizeBuffer();
20     double A = 0;
21     while (E.Receive(0).size() < E.Receive(0).Size()) {
22         Cell C; E.Receive(0) >> C;
23         pout << "received: " << C;
24         A += C.area();
25     }
26     A = PPM->Sum(A); pout << "area " << A << endl;
27     delete PPM;
28     return 0;
29 }
```

3 Parallele Interface und Lastverteilung

Eine Lastverteilung wird durch

$$\text{dest}\colon \mathcal{C} \longrightarrow \mathcal{P} = \{0, \ldots, P-1\}$$

bestimmt. Setze:

$\mathcal{C}_p = \{C \in \mathcal{C}\colon \operatorname{dest}(C) = p\}$	disjunkte Zerlegung von $\mathcal{C}$
$\mathcal{V}_p = \bigcup_{C \in \mathcal{C}_p} C\colon$	überlappende Zerlegung von $\mathcal{V}$
$\bar{\Omega}_p = \bigcup_{C \in \mathcal{C}_p} \bar{\Omega}_C$	Gebietszerlegung
$\Gamma_{pq} = \partial\Omega_p \cup \partial\Omega_q$	Interface zwischen Prozessor p und q
$\pi(x) = \{p\colon x \in \mathcal{V}_p\} \subset \mathcal{P}$	Prozessmenge für $x \in \mathcal{V}$

Interfacekommunikation

$S(p,q) = |\Gamma_{pq} \cup \mathcal{V}|$ Zahl der Interfacepunkte auf Γ_{pq}
$G = \big\{\{(p,q)\colon S(p,q) \neq 0\big\}$ gewichteter, gerichteter Kommunikationsgraph für den Datenaustausch auf dem Interface

mit

$$\begin{aligned}
S(p,\cdot) &= \sum_{q \in \mathcal{P}} S(p,q) && \text{Sendevolumen auf Prozessor } p \\
S(\cdot,q) &= \sum_{p \in \mathcal{P}} S(p,q) && \text{Empfangsvolumen auf Prozessor } q \\
M(p) &= |\{q \in \mathcal{P}\colon S(p,q) \neq 0\}| && \text{Zahl der Sendungen auf Prozessor } p \\
N(q) &= |\{p \in \mathcal{P}\colon S(p,q) \neq 0\}| && \text{Zahl der Empfänger auf Prozessor } q
\end{aligned}$$

Kommunikationszeit

$$T_p = (M(p) + N(p))\, T_{\text{init}} + (S(p,\cdot) + S(\cdot,p)) T_{\text{trans}}$$

Satz

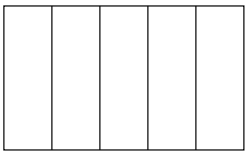
Die Aufgabe, eine Lastverteilung mit $|\mathcal{C}_p| = \frac{1}{p}|\mathcal{C}|$ zu bestimmen, so dass $\max_{p \in \mathcal{P}} T_p$ minimal ist, ist NP vollständig.

Einfache Lastverteilungen

a) Lineare Verteilung
wähle Nummerierung $\mathcal{C} = \{C_0 \ldots, C_{M-1}\}$ und setze

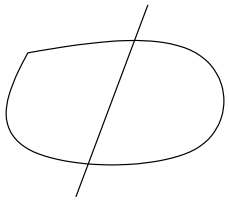
$$\mathrm{dest}(C_m) = \text{ int } \left(\frac{m}{N}\right), \; N = \text{ int } \left(\frac{M+P-1}{P}\right)$$

Nummerierung z.B. Sortierung nach Zellmittelpunkten $x_c = \frac{1}{3}(x_0 + x_1 + x_2)$.
Totalordnung auf $\mathbf{R}^2$, z.B. $\binom{y_1}{y_2} < \binom{z_1}{z_2} \iff y_1 < z_1$ oder $(y_1 = z_1$ und $y_2 < z_2)$



(Alternative: Space-Filling Curves)

b) RCB = Recursive Coordinate Bisection
(hier für $P = 2^L$)
sortiere $\mathcal{C} = \{C_0, \ldots, C_{M-1}\}$ nach $x - y$ Totalordnung
zerlege $\mathcal{C} = \mathcal{C}_1^{(1)} \cup \mathcal{C}_2^{(1)} = \{C_m \colon m < \frac{M}{2}\} \cup \{\mathcal{C}_m \colon m \geq \frac{M}{2}\}$
sortiere jedes $\mathcal{C}_k^{(1)}$ mit $y - x$ Totalordnung
zerlege $\mathcal{C}_k^{(1)} = \mathcal{C}_{2k}^{(2)} \cup \mathcal{C}_{2k+1}^{(2)}$
sortiere und zerlege $\mathcal{C}_k^{(2)}$ nach $x - y$ Totalordnung u.s.w.
$\Longrightarrow$ ordne jede Teilmenge $\mathcal{C}_k^{(L)}$ einem Prozessor zu.



c) RIB = Recursive Inertia Bisection
wähle Nummerierung $\mathcal{C} = \{C_0, \ldots, C_{M-1}\}$
bilde Nachbarschaftsgraph

$$\mathcal{N} = \{(m,n) \colon |C_m \cap C_n| = 2\} \quad \text{(gemeinsame Kante)}$$

$$\underline{A} \in \mathbf{R}^{M,M}\colon \underline{A}(m,n) = \begin{cases} k_n & m = n \\ -1 & (m,n) \in \mathcal{N},\ m \neq n \\ 0 & \text{sonst} \end{cases}$$

mit Knotengrad $k_n = |\{m\colon (n,m) \in \mathcal{N}\}|$

$\underline{A}\,\underline{u}_0 = 0$ für $\underline{u}_0 \equiv 1 \implies$ kleinster EW $\lambda_0 = 0$

bestimme $\underline{u}_1 \in \mathbf{R}^M$ mit $\underline{A}\,\underline{u}_1 = \lambda_1$ mit λ_1 zweitkleinster Eigenwert,

zerlege $\mathcal{C} = \{C_m\colon \underline{u}_1(m) < 0\} \cup \{C_m\colon \underline{u}_1(m) \geq 0\}$.

Aufbau der Prozessormenge

a) Bestimme die Lastverteilung $\mathrm{dest}\colon \mathcal{C} \longrightarrow \mathcal{P}$ auf Prozessor $p = 0$.

b) Setze $\pi(x) = \emptyset \qquad x \in \mathcal{V}$

für $C \in \mathcal{C}$ und $x \in C\colon \pi(x) := \pi(x) \cup \{\mathrm{dest}(C)\}$

(auf Prozessor $p = 0$)

c) setze $B(p,q) = \emptyset$

auf Prozessor $p = 0\colon\ B(0,q) = \{(x,\pi(x))\colon q \in \pi(x), |\pi(x)| > 1\} \cup \{C\colon\ \mathrm{dest}(C)\}$

d) Austausch von B

e) Auslesen von B;

auf Prozessor $q \neq 0\colon$ lese $(x,\pi) \in B(0,q)$, setze $\pi_p(x) = \pi$

lese $C \in B(0,q)$, setze $\mathcal{C}_q := \mathcal{C}_q \cup \{C\}$

auf Prozessor $q = 0\colon$ $\mathcal{C}_0 = \{C\colon\ \mathrm{dest}(C) = 0\}$

setze $\pi_0(x) = \emptyset$ für $x \in \mathcal{V}_0 = \bigcup_{C \in \mathcal{C}_0} C$

Speicherersparnis: für $x \in \mathcal{V}_p$ repräsentiere

$\pi_p(x) = \{p\}$ durch $\pi_p(x) = \emptyset$!

Gitteraufbau: $\mathcal{M}_p = (\mathcal{C}_p, \mathcal{V}_p)$

$\mathcal{C}_p = \emptyset,\ \mathcal{V}_p = \emptyset$

für alle neuen Zellen $C\colon$ $\mathcal{C}_p := \mathcal{C}_p \cup \{C\}$

$\mathcal{V}_p := \mathcal{V}_p \cup C$

Problem beim Entfernen von Zellen C auf $\mathcal{C}_p\colon$

Wie entscheidet man, ob $x \in C$ aus $\mathcal{V}_p$ entfernt werden kann?

Datenstrukturen zur Verwaltung von Mengen

Wir benötigen Operationen
insert $(\mathcal{V}, x) = \{\mathcal{V} := \mathcal{V} \cup \{x\}\}$
in $(\mathcal{V}, x) = (x \in \mathcal{V}\ ?)$

a) Vektor $\mathcal{V} = \boxed{x_0|x_1|x_2|\ldots|x_{N-1}}$

in $(\mathcal{V}, x)$: für alle $i = 0, \ldots, N-1$ wenn $x_i = x$ return true
return false

insert $(\mathcal{V}, x)$: resize (kopiere $\mathcal{V}$ in einem Vektor der Länge $|\mathcal{V}| + 1$), setze $x_{|\mathcal{V}|} = x$

b) Liste $\mathcal{V} =$ NULL $\leftarrow \boxed{x_0} \rightleftarrows \boxed{x_1} \ldots \rightleftarrows \boxed{x_{N-1}} \rightarrow$ NULL

jedes Element besteht aus

data	Wert
pred	Zeiger auf oben Vorgänger
succ	Zeiger auf den Nachfolger

Zeiger auf

in $(\mathcal{V}, x)$: : $p =$ Anfang der Liste

solange $p \neq$ NULL wenn p.data $= x$ return true

$p :=$ p.succ

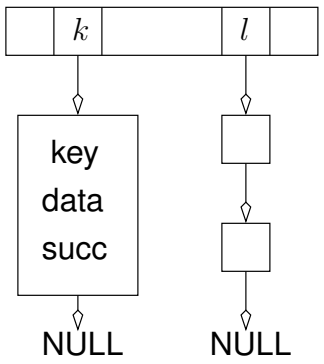
return false

insert $(\mathcal{V}, x)$: $p =$ Zeiger auf Ende der Liste

Erzeuge neuen Eintrag q

setze p.succ $= q$, q.pred $= p$, q.succ = NULL

c) Hash



Verwende Schlüsselfunktion key: $\mathcal{V} \longrightarrow \mathcal{N}$

Allokiere einen Vektor H der Länge K (Hash-Tabelle)

Initialisiere $H(k) =$ NULL leere Liste

in $(\mathcal{V}, x)$: $k =$ key (x) mod K return in $(H(k), x)$

insert $(\mathcal{V}, x)$: wenn $|\mathcal{V}| > c\,K$: resize

baue neue Hash-Tabelle der Länge $2K$ auf

$k =$ key (x) mod K insert $(H(k), x)$

Eigenschaften bei der Verwaltung einer Menge mit N Elementen

	Vektor	Liste	Hash
in	$O(N)$	$O(N)$	$O(1)$
insert	$O(N)$	$O(1)$	$O(1)$
Speicherbedarf	optimal	etwas mehr	mehr $(O(N))$
Vorteil	schneller Durchlauf über alle Elemente	schnelles Einfügen	schnelles Testen (in)
Nachteil	resize und suchen teuer	suchen teuer	mehr Speicher benötigt gute Hash-Funktion

z.B. key $(x) =$ int $(115421 \cdot (x[0] + \varepsilon_1) + 124323 \cdot (x[1] + \varepsilon_2)) + \text{const},\ \varepsilon_1 = 0.1,\ \varepsilon_2 = 0.3$

Iteratoren in `C++`

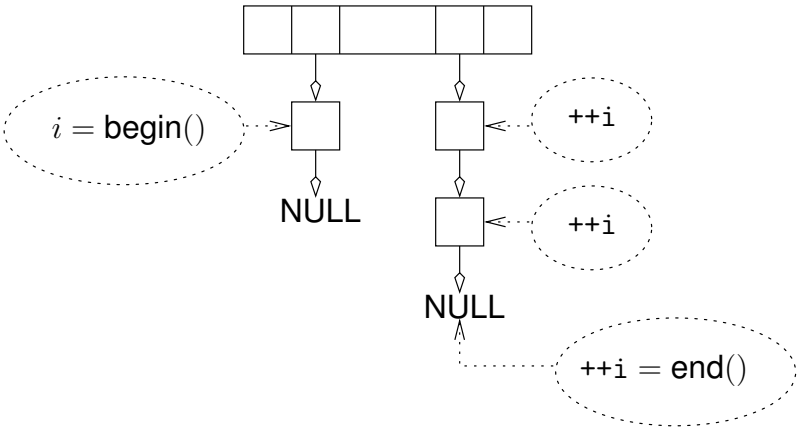
vector, list und hash-map sind Container zum Verwalten von Daten. Zu jedem Container wird ein Iterator definiert, um alle Objekte im Container (in der abgespeicherten Reihenfolge) zu durchlaufen.

Funktionalität

begin	Iterator, der einen Zeiger auf das erste Element im Container enthält
$*$	das zugehörige Element (bei list / vector)
$\longrightarrow$ first	der Schlüssel (bei hash-map)
$\longrightarrow$ second	das zugehörige Element (bei hash-map)
++	verschiebt im Iterator den Zeiger auf das nächste Element
end	Ergebnis: von `++` für das letzte Element

Wir unterscheiden Iteratoren, die das Element nicht ändern dürfen (const) und die es ändern dürfen.

Beispiel



3.1 Paralleles Programm 4

3.1.1 Beschreibung

Um die Kommunikation zwischen den Prozessoren zu minimieren, werden in dieser Programmversion die eingelesenen Zellen vor der Versendung sortiert. Jeder Prozessor erhält dadurch einen zusammenhängenden Teil des Gesamtnetzes.

3.1.2 Dateien

```
PFEM/04/Parallel.h
PFEM/04/Parallel.C
PFEM/04/Mesh.h
PFEM/04/Main.C
PFEM/04/Makefile
```

3.1.3 Punkte und Dreiecke (Teil 2)

```
1  // file:     PFEM/04/Mesh.h
2
3  #ifndef _MESH_H_
4  #define _MESH_H_
5
6  #include "Parallel.h"
7
8  const double GeometricTolerance = 1e-8;
9
10 class Point {
11     double x[2];
12 public:
13     Point () {};
14     Point (double a, double b) { x[0] = a; x[1] = b; }
15     Point (const Point& y) { ... }
16     double operator [] (int i) const { ... }
17     Point& operator += (const Point& y) { x[0]+=y[0];x[1]+=y[1]; return *this;}
18     Point& operator -= (const Point& y) { ... }
19     inline bool operator < (const Point& y) const {
20         if (x[0] < y[0] - GeometricTolerance) return true;
21         if (x[0] > y[0] + GeometricTolerance) return false;
22         if (x[1] < y[1] - GeometricTolerance) return true;
23         return false;
24     }
25     friend istream& operator >> (istream& s, Point& y) { ... }
26 };
27
28 inline Point operator + (const Point& x, const Point& y) {
29     Point z = x; return z += y;
30 }
31
32 inline Point operator - (const Point& x, const Point& y) { ... }
33
34 inline Point operator * (double a, const Point& x) {
35     return Point(a*x[0],a*x[1]);
36 }
37
38 inline double det(const Point& x,const Point& y) { ... }
39 inline ostream& operator << (ostream& s, const Point& x) { ... }
40
41
42 class Cell {
```

```
43      Point x[3];
44  public:
45      Cell () {}
46      double area () const { ... }
47      Point operator () () const { return (1.0/3.0) * (x[0]+x[1]+x[2]); }
48      bool operator < (const Cell& C) const { return ((*this)()<C()); }
49      const Point& operator [] (int i) const { ... }
50      friend istream& operator >> (istream& s, Cell& C) { ... }
51  };
52
53  inline ostream& operator << (ostream& s, const Cell& C) { ... }
54
55  inline istream& operator >> (istream& s, vector<Cell>& C) {
56      for (int m=0; m<C.size(); ++m) s >> C[m];
57      return s;
58  }
59  #endif
```

Um eine bessere Aufteilung der Dreiecke eines Gitters auf die einzelnen parallelen Prozesse zu realisieren, wurde die Klasse **Point** um folgende Methoden, Operatoren und Konstanten erweitert:

- `Point::Point (double a, double b)`
  Konstruktor: Erzeugt ein neues Objekt der Klasse `Point` mit der ersten und zweiten Koordinate `a` bzw. `b`.

- `Point& Point::operator += (const Point& y)`
  Zuweisungsoperator: Sind `x` und `y` zwei Punkte, so führt der Befehl `x += y` die Zuweisung `x = x+y` aus.

- `inline bool Point::operator < (const Point& y)`
  Vergleichsoperator: Sind `x` und `y` zwei `Point`-Objekte, so evaluiert der Ausdruck `x < y` zu `true` wenn die erste Koordinate von `x` kleiner ist als die erste Koordinate von `y`. Ansonsten evaluiert der Ausdruck zu `false`. Unterscheiden sich die ersten Koordinaten der beiden Punkte nur um den Wert `GeometricTolerance`, dann werden die zweiten Koordinaten vergleichen.

- `const double GeometricTolerance = 1e-8`
  Geometrische Toleranz: Unterscheiden sich die Koordinaten zweier Punkte um weniger als diese Konstante, so werden die Koordinaten als gleich angesehen.

- `inline Point operator + (const Point& x, const Point& y)`
  Gibt die Summe zweier Punkte `x` und `y` im Sinne der Ortsvektoren zurück.

- `inline Point operator * (double a, const Point& x)`
  Gibt den mit `a` skalierten Punkt `x` zurück.

Die Klasse **Cell** wurde in dieser Version des parallelen Programms ebenfalls erweitert. Die neuen Methoden dieser Klasse sind:

- `Point Cell::operator () ()`
  Gibt den Schwerpunkt des Dreiecks zurück.
- `bool Cell::operator < (const Cell& C)`
  Vergleichsoperator: Vergleicht zwei Dreiecke anhand ihrer Schwerpunkte.

Zusätzlich wurde ein Eingabeoperator für `Cell`-wertige Vektoren definiert:

- `inline istream& operator >> (istream& s, vector<Cell>& C)`
  Eingabeoperator: Liest nacheinander die Dreiecke in einem `Cell`-wertigen Vektor `C` aus einem Eingabestrom `s` ein.

3.1.4 Das Hauptprogramm

```
1  // file:     PFEM/04/Main.C
2
3  #include "Mesh.h"
4
5  #include <fstream>
6  #include <vector>
7
8  int main(int argv, char** argc)
9  {
10     PPM = new ParallelProgrammingModel(&argv,&argc);
11     ExchangeBuffer E;
12     if (PPM->master()) {
13         ifstream s("mesh");
14         int M; s >> M;
15         int N = (M + PPM->size() - 1) / PPM->size();
16         vector<Cell> C(M); s >> C;
17         sort(C.begin(),C.end());
18         for (int m=0; m<M; ++m) {
19             int dest = m / N;
20             E.Send(dest) << C[m];
21         }
22     }
23     E.CommunicateSizeBuffer();
24     double A = 0;
25     while (E.Receive(0).size() < E.Receive(0).Size()) {
26         Cell C; E.Receive(0) >> C;
27         pout << "received : " << C() << endl;
```

```
28          A += C.area();
29      }
30      A = PPM->Sum(A); pout << "area " << A << endl;
31      delete PPM;
32      return 0;
33  }
```

3.2 Paralleles Programm 5

3.2.1 Beschreibung

In dieser Programmversion wird zusätzlich für jeden Punkt der Interfaces zwischen den Teilgittern gespeichert, welche Prozessoren auf diesen Punkten arbeiten. Jeder Prozessor speichert also in seinem eigenen Container **ProcSets** die Prozessormengen seiner Interfacepunkte über die er mit den anderen Prozessoren kommunizieren muß. Ausgegeben werden jeweils die **ProcSets** und die Gesamtfläche des Netzes.

3.2.2 Dateien

```
PFEM/05/Parallel.h
PFEM/05/Parallel.C
PFEM/05/Point.h
PFEM/05/Mesh.h
PFEM/05/Main.C
PFEM/05/Makefile
PFEM/05/mesh
```

3.2.3 Die Prozessmengen

```
1   // file:     PFEM/05/Parallel.h
2
3   #ifndef _PARALLEL_H_
4   #define _PARALLEL_H_
5
6   #include "Point.h"
7
8   #include <vector>
9   #include <list>
10  #include <ext/hash_set>
11  using __gnu_cxx::hash_set;
```

```
12 #include <ext/hash_map>
13 using __gnu_cxx::hash_map;
14
15 class ExchangeBuffer;
16
17 class ParallelProgrammingModel { ... };
18
19 extern ParallelProgrammingModel* PPM;
20
21 class Exchange { ... };
22
23 inline ostream& operator << (ostream& s, const Exchange& E) { ... }
24 const size_t BufferSize = 128000;
25
26 class Buffer { ... };
27
28 class ExchangeBuffer : public Exchange { ... };
29
30 class ProcSet : public vector<short> {
31  public:
32     ProcSet () {};
33     void Add (short q) {
34         int m = size();
35         for (short i=0; i<m; ++i)
36             if ((*this)[i] == q) return;
37         resize(m+1);
38         if (q < (*this)[0]) {
39             (*this)[m] = (*this)[0];
40             (*this)[0] = q;
41         }
42         else (*this)[m] = q;
43     }
44     bool Erase () {
45         if (size() <= 1) return true;
46         for (int i=0; i<size(); ++i)
47             if ((*this)[i] == PPM->proc()) return false;
48         return true;
49     }
50 };
51
52 inline ostream& operator << (ostream& s, const ProcSet& P) {
53     for (short i=0; i<P.size(); ++i) s << " " << P[i];
54     return s;
55 }
56
57 class procset : public hash_map<Point,ProcSet,Hash>::const_iterator {
58     typedef hash_map<Point,ProcSet,Hash>::const_iterator Iterator;
59  public:
```

```
60      procset (hash_map<Point,ProcSet,Hash>::const_iterator P) : Iterator(P) {}
61      const Point& operator () () const { return (*this)->first; }
62      const ProcSet& operator * () const { return (*this)->second; }
63      int size () const { return (*this)->second.size(); }
64      short operator [] (int i) const { return (*this)->second[i]; }
65      friend ostream& operator << (ostream& s, const procset& P) {
66          return s << P->first << " : " << P->second << endl;
67      }
68  };
69
70  class ProcSets : public hash_map<Point,ProcSet,Hash> {
71   public:
72      ProcSets () {}
73      procset procsets () const { return procset(begin()); }
74      procset procsets_end () const { return procset(end()); }
75      void Add (const Point& x, short q) {
76          (*this)[x];
77          hash_map<Point,ProcSet,Hash>::iterator p = find(x);
78          p->second.Add(q);
79      }
80      void Copy (const Point& x, const Point& y) {
81          hash_map<Point,ProcSet,Hash>::iterator p = find(x);
82          if (p == end()) return;
83          (*this)[y] = p->second;
84      }
85      void Clean () {
86          for (hash_map<Point,ProcSet,Hash>::iterator p=begin(); p!=end();) {
87              hash_map<Point,ProcSet,Hash>::iterator q = p++;
88              if (q->second.Erase()) erase(q);
89          }
90      }
91  };
92
93  inline ostream& operator << (ostream& s, const ProcSets& PS) {
94      for (procset p=PS.procsets(); p!=PS.procsets_end(); ++p)
95      s << p() << " :" << *p << endl;
96      return s;
97  }
98
99  #define pout ...
100 #endif
```

Die Klasse **ProcSet** ist ein erweiterter Vektor, der `short`-Werte speichern kann. Diese Klasse wird dazu verwendet, die Prozessornummer derjenigen Prozesse abzuspeichern, die auf einem bestimmten Knotenpunkt eines Gitters operieren. Für die Klasse **ProcSet** sind folgende Methoden und Operatoren definiert:

- `ProcSet::ProcSet ()`
  Konstruktor: Erzeugt ein neues Objekt der Klasse `ProcSet`.

- `void ProcSet::Add (short q)`
  Fügt die Prozessnummer `q` in den Vektor ein, sofern diese dort noch nicht vorhanden ist.

- `bool ProcSet::Erase ()`
  Die Methode gibt den Wert `true`, wenn in dem Vektor nur eine einzelne Prozessnummer gespeichert ist, oder wenn sich die Prozessnummer des aufrufenden Prozesses nicht im Vektor befindet.

- `inline ostream& operator << (ostream& s, const ProcSet& P)`
  Ausgabeoperator: Gibt die in einem `ProcSet`-Objekt `P` gespeicherten Prozessnummern in einen Ausgabestrom `s` aus.

Die Klasse **procset** ist eine Iteratorklasse. Iteratoren sind ein Bestandteil der C++ Standard Template Library (STL). Mit einem Objekt der Klasse `procset` können die Einträge eines `ProcSet`-wertigen Hash mit `Point`-wertigen Schlüsseln durchlaufen werden. Dazu besitzt jede Iteratorklasse die Operatoren `--` und `++`, die den Iterator auf den vorangehenden bzw. den nachfolgenden Eintrag im Hash setzen. Darüber hinaus sind für die Klasse `procset` folgende Methoden und Operatoren definiert:

- `procset::procset (hash_map<Point,ProcSet,Hash>::const_iterator P)`
  Konstruktor: Erzeugt einen neuen Iterator. Als Argument wird dem Konstruktor ein Iterator `P` eines `ProcSet`-wertigen Hashes mit `Point`-wertigen Schlüsseln übergeben. Der von `P` bezeichnete Eintrag im Hash wird zum aktuellen Eintrag des neuen Iterators.

- `const Point& procset::operator () ()`
  Gibt eine Referenz auf den Schlüssel des aktuellen Eintrags im Hash zurück.

- `const ProcSet& procset::operator * ()`
  Gibt eine Referenz auf den Wert des aktuellen Eintrags im Hash zurück.

- `int procset::size ()`
  Gibt die Anzahl der Prozessnummern zurück, die im `ProcSet`-Objekt des aktuellen Eintrags im Hash gespeichert sind.

- `short procset::operator [] (int i)`
  Ist `p` ein `procset`-Iterator, so gibt `p[i]` die `i`-te Prozessnummer zurück, die im `ProcSet`-Objekt des aktuellen Eintrags im Hash gespeichet ist.

- `friend ostream& operator << (ostream& s, const procset& P)`
  Ausgabeoperator: Gibt den Schlüssel und den Wert des Eintrags im Hash, der von einem Iterator `P` bezeichnet wird, in einen Ausgabestrom `s` aus.

Die Klasse **ProcSets** implementiert schließlich die Funktion π, die jedem Knotenpunkt x eines Gitters die Menge $\pi(x)$ der Nummern derjenigen Prozesse zuordnet, die auf dem Punkt x operieren. Die Klasse `ProcSets` ist dazu als Hash definiert. Jeder Eintrag im Hash wird über einen `Point`-wertigen Schlüssel eindeutig identifiziert und erhält ein `ProcSet`-Objekt als Wert. Die Einträge im Hash können mit einem `procset`-Iterator durchlaufen werden. Zusätzlich sind folgende Methoden und Operatoren für die Klasse `ProcSets` definiert:

- `ProcSets::ProcSets ()`
  Konstruktor: Erzeugt ein neues Hash der Klasse `ProcSets`.
- `procset ProcSets::procsets ()`
  Gibt einen `procset`-Iterator zurück, der den ersten Eintrag im Hash bezeichnet.
- `procset ProcSets::procsets_end ()`
  Gibt einen `procset`-Iterator zurück, der das Ende des Hashes bezeichnet.
- `void ProcSets::Add (const Point& x, short q)`
  Diese Methode fügt der Prozessmenge eines Knotenpunkts `x` die Prozessnummer `q` hinzu, sofern diese noch nicht in der Prozessmenge enthalten ist.
- `void ProcSets::Copy (const Point& x, const Point& y)`
  Mit dieser Methode wird einem Knotenpunkt `y` die Prozessmenge eines Knotenpunktes `x` zugewiesen. Die Prozessmenge des Knotenpunkts `x` bleibt dabei erhalten. Nach einem Aufruf dieser Methode besitzen beiden Knotenpunkte `x` und `y` diesselbe Prozessmenge.
- `void ProcSets::Clean ()`
  In einem `ProcSets`-Hash werden nur die Prozessmengen solcher Knotenpunkte gespeichert, auf denen mindestens zwei Prozesse operieren. Außerdem speichert jeder parallele Prozess nur die Prozessmengen derjenigen Knotenpunkte ab, auf denen er selbst operiert. Auf diese Weise sind in dem `ProcSets`-Hash jedes einzelnen parallelen Prozesses nur solche Knotenpunkte berücksichtigt, für die der Prozess mit anderen Prozessen kommunizieren muss. Die Methode `Clean` entfernt dazu in jedem aufrufenden Prozess die überflüssigen Einträge aus dem Hash.
- `inline ostream& operator << (ostream& s, const ProcSets& PS)`
  Ausgabeoperator: Gibt nacheinander die Punkte und deren Prozessmengen in einem `ProcSets`-Hash `PS` in einen Ausgabestrom `s` aus.

3.2.4 Punkte und Dreiecke (Teil 3)

```
1 // file:     PFEM/05/Point.h
2
```

```
3  #ifndef _POINT_H_
4  #define _POINT_H_
5
6  #include <iostream>
7  using namespace std;
8
9  const double GeometricTolerance = 1e-8;
10
11 class Point {
12     double x[2];
13 public:
14     Point () {};
15     Point (const Point& y) { ... }
16     Point (double a, double b) { ... }
17     double operator [] (int i) const { ... }
18     Point& operator -= (const Point&y) { ... }
19     Point& operator += (const Point&y) { ... }
20     inline bool operator < (const Point& z) const { ... }
21     inline bool operator == (const Point& z) const {
22         if (x[0] < z.x[0] - GeometricTolerance) return false;
23         if (x[0] > z.x[0] + GeometricTolerance) return false;
24         if (x[1] < z.x[1] - GeometricTolerance) return false;
25         if (x[1] > z.x[1] + GeometricTolerance) return false;
26         return true;
27     }
28     friend istream& operator >> (istream& s, Point& y) { ... }
29 };
30
31 inline ostream& operator << (ostream& s, const Point& y) { ... }
32 inline Point operator - (const Point& y, const Point& z) { ... }
33 inline Point operator + (const Point& y, const Point& z) { ... }
34 inline Point operator * (double a, const Point& z) { ... }
35 inline double det(const Point& y,const Point& z) { ... }
36
37 class Hash {
38 public:
39     size_t operator () (const Point& y) const {
40     return size_t(115421.1*y[0]+124323.3*y[1]+345453*GeometricTolerance);
41     }
42 };
43 #endif
```

Die Klasse **Point** wurde in dieser Version des parallelen Programms um einen Vergleichsoperator erweitert:

- `inline bool Point::operator == (const Point& z)`
  Vergleichsoperator: Sind `x` und `y` zwei `Point`-Objekte, dann evaluiert der Aus-

druck `x == y` zu `true`, wenn sich die Koordinaten beider Punkte um weniger als den Toleranzwert `GeometricTolerance` unterscheiden. Diese Methode wird beim Durchsuchen eines `ProcSets`-Hash benötigt.

Die Klasse **Hash** definiert eine Abbildung, die einem Knotenpunkt x eines Gitters eine natürliche Zahl zuordnet. Diese Klasse wird zur Verwaltung einer `ProcSets`-Hash-Tabelle benötigt. Der einzige Operator dieser Klasse ist:

- `size_t Hash::operator () (const Point& y)`
  Gibt zu einem `Point`-Objekt `y` eine natürliche Zahl aus, die die Position des Punktes in einer `ProcSets`- Hash-Tabelle bestimmt.

3.2.5 Das Gitter

```
1  // file:     PFEM/05/Mesh.h
2
3  #ifndef _MESH_H_
4  #define _MESH_H_
5
6  #include "Parallel.h"
7
8  class Mesh {
9   public:
10     class Cell {
11         ...
12     };
13   private:
14     list<Cell> Cells;
15     hash_set<Point,Hash> Vertices;
16     void Insert (const Point& y) { Vertices.insert(y); }
17   public:
18     Mesh () {}
19     void Insert (const Cell& C) {
20         Cells.push_back(C);
21         Insert(C[0]);
22         Insert(C[1]);
23         Insert(C[2]);
24     }
25     double area () {
26         double A = 0;
27         for (list<Cell>::const_iterator c=Cells.begin();c!=Cells.end();++c)
28             A += (*c).area();
29         return A;
```

```
30      }
31      bool Vertex (const Point& y) {
32          return (Vertices.find(y) != Vertices.end());
33      }
34  };
35
36  inline istream& operator >> (istream& s, vector<Mesh::Cell>& C) {
37      for (int m=0; m<C.size(); ++m) s >> C[m];
38      return s;
39  }
40  #endif
```

Die Klasse **Mesh** kapselt Finite-Element-Gitter. Ein Gitter ist dabei als eine Menge von Knotenpunkten und eine Menge von Dreiecken zwischen den Knotenpunkten definiert, die von der Klasse `Mesh` verwaltet werden. Die Klasse `Cell` wurde in dieser Version des parallelen Programms als öffentliche Unterklasse `Mesh::Cell` des Gitters definiert. Für die Klasse `Mesh` wurden die folgenden öffentlichen Methoden und Operatoren definiert:

- `Mesh::Mesh ()`
  Konstruktor: Erzeugt ein neues Objekt der Klasse `Mesh`. Jedes `Mesh`-Objekt besitzt als private Variablen eine Liste `Cells` und ein Hash `Vertices` in denen die Dreiecke bzw. die Punkte des Gitters abgespeichert werden.

- `void Mesh::Insert (const Cell& C)`
  Fügt ein Dreieck `C` in das Gitter ein. Das übergebene `Cell`-Objekt wird in die private Liste `Cells` eingefügt. Die Eckpunkte des Dreiecks werden danach in das private Hash `Vertices` eingefügt, sofern die Punkte noch nicht im Hash vorhanden sind.

- `double Mesh::area ()`
  Gibt die Fläche des Gitters zurück, die sich als der Summe der Flächeninhalte aller darin vorhandenen Dreiecke berechnet.

- `bool Mesh::Vertex (const Point& y)`
  Gibt den Wert `true` zurück, wenn der übergebene Punkt `y` ein Knotenpunkt des Gitters ist.

- `inline istream& Mesh::operator >> (istream& s,vector<Mesh::Cell>& C)`
  Eingabeoperator: Liest nacheinander einzelne Dreiecke aus einem Eingabestrom `s` in einen `Mesh::Cell`-wertigen Vektor `C` ein.

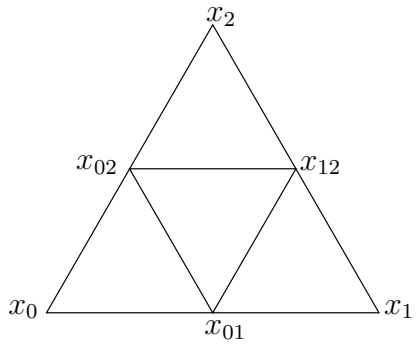
3.2.6 Das Hauptprogramm

```
1  // file:     PFEM/05/Main.C
2
3  #include "Mesh.h"
4
5  #include <fstream>
6  #include <vector>
7
8  int main(int argv, char** argc)
9  {
10     PPM = new ParallelProgrammingModel(&argv,&argc);
11     Mesh mesh;
12     ProcSets PS;
13     ExchangeBuffer E;
14     if (PPM->master()) {
15         ifstream s("mesh");
16         int M; s >> M;
17         int N = (M + PPM->size() - 1) / PPM->size();
18         vector<Mesh::Cell> C(M); s >> C;
19         sort(C.begin(),C.end());
20         for (int m=0; m<M; ++m) {
21             int dest = m / N;
22             if (dest == PPM->proc()) mesh.Insert(C[m]);
23             for (int k=0; k<3; ++k) PS.Add(C[m][k],dest);
24         }
25         for (procset p=PS.procsets(); p!=PS.procsets_end(); ++p) {
26             for (int i=0; i<p.size(); ++i) {
27                 int dest = p[i];
28                 if (dest == PPM->proc()) continue;
29                 if (p.size() == 1) continue;
30                 E.Send(dest) << short(p.size());
31                 E.Send(dest) << p();
32                 for (int j=0; j<p.size(); ++j) E.Send(dest) << short(p[j]);
33             }
34         }
35         for (short q=1; q<PPM->size(); ++q) E.Send(q) << short(0);
36         for (int m=0; m<M; ++m) {
37             int dest = m / N;
38             if (dest != 0) E.Send(dest) << C[m];
39         }
40     }
41     E.CommunicateSizeBuffer();
42     if (PPM->master())
43         PS.Clean();
44     else {
45         short s; E.Receive(0) >> s;
46         while (s > 0) {
```

```
47              Point P;
48              E.Receive(0) >> P;
49              for (int i=0; i<s; ++i) {
50                  short dest;
51                  E.Receive(0) >> dest;
52                  PS.Add(P,dest);
53              }
54              E.Receive(0) >> s;
55          }
56          while (E.Receive(0).size() < E.Receive(0).Size()) {
57              Mesh::Cell C; E.Receive(0) >> C;
58              mesh.Insert(C);
59          }
60      }
61      pout << "ProcSets" << endl << PS << endl;
62      pout << "area " << PPM->Sum(mesh.area()) << endl;
63      delete PPM;
64      return 0;
65  }
```

4 Parallele Verfeinerung

Um den Fehler der numerischen Lösung zu verkleinern, können wir beispielsweise zu einem feineren Gitter des Gebietes Ω übergehen. Für die parallele Realisierung betrachten wir nur den Spezialfall der uniformen Verfeinerung.



$C = (x_0, x_1, x_2) \quad \longrightarrow$

$C_0 = (x_0,\, x_{01},\, x_{02})$

$C_1 = (x_1,\, x_{12},\, x_{01})$

$C_2 = (x_2,\, x_{02},\, x_{12})$

$C_3 = (x_{01},\, x_{12},\, x_{02})$

wobei $x_{ij} = \frac{1}{2}(x_i + x_j)$

setze $\pi(C_k) = \pi(C) = \{\mathrm{dest}(C)\}$

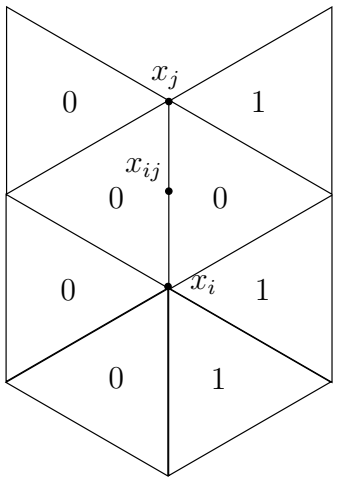
Problem

Bestimme $\pi(x_{ij}) = \{\mathrm{dest}(C) \colon x_{ij} \in \bar{\Omega}_C\}$

Lemma

Es gilt für die Kantenmittelpunkte $x_{ij}\colon\ \pi(x_{ij}) \subset \pi(x_i) \cap \pi(x_j)$.
Aber i. A. gilt: $\pi(x_{ij}) \neq \pi(x_i) \cap (x_j)$.

Beispiel



$\pi(x_{ij}) = \{0\} \neq \pi(x_i) = \pi(x_j) = \{0, 1\}$

Lösung

a) Einlesen von $\mathcal{C}$ auf Prozessor $p = 0$.

b) Aufbau der Vertexmenge: $\mathcal{V} = \emptyset$

für $C = (x_0, x_1, x_2) \in \mathcal{C}$: $\mathcal{V} := \mathcal{V} \cup C \cup \{x_{01}, x_{02}, x_{12}\}$

c) Bestimmung der Lastverteilung $\mathrm{dest}\colon \mathcal{C} \to \mathcal{P}$

d) Aufbau der Prozessormengen: $\pi(x) = \emptyset$ für alle $x \in \mathcal{V}$

für $C = (x_0, x_1, x_2) \in \mathcal{C}$: $\pi(x_j) := \pi(x_j) \cup \{\mathrm{dest}(C)\}$
$\pi(x_{ij}) := \pi(x_{ij}) \cup \{\mathrm{dest}(C)\}$

e) Datenaustausch von $\mathcal{C}$, π

f) Verfeinerung von $\mathcal{C}^{(0)} \to \mathcal{C}^{(1)}$ durch $C \to (C_0, C_1, C_2, C_3)$

g) Prozessormenge für Knoten schon fertig.
Für neue Kantenmittelpunkte auf $\Omega_C \cap \Omega_{C_k}$: $\pi(\frac{1}{2}(x_i + x_{ij})) = \pi(x_{ij})$.

Randidentifikation

Kantenmittelpunkte $\mathcal{E} = \{x_{ij} = \frac{1}{2}(x_i + x_j)\colon x_i, x_j \in C,\ x_i \neq x_j\}$

Randpunkte $\mathcal{B} = \mathcal{E} \cap \partial\,\Omega_{\mathcal{C}}$

Möglichkeiten zur Bestimmung von $\mathcal{B}$ für $P = 1$:

1) Einlesen von (x, id) für $x \in \mathcal{B}$ (mit Kennzahl id) für Randsegmente zur Zuordnung von Randbedingungen)

2) Setze $\mathcal{B} = \emptyset$ für alle C.
Wenn $x_{ij} \notin \mathcal{B}$: $\mathcal{B} := \mathcal{B} \cup \{x_{ij}\}$,
wenn $x_{ij} \in \mathcal{B}$: $\mathcal{B} := \mathcal{B} \setminus \{x_{ij}\}$.

3) Nachbarschaften ($\mathcal{F}$ Face)
(setze $x_C := \frac{1}{3}(x_0 + x_1 + x_2)$ Zellmittelpunkt)
$\mathcal{F} = \emptyset$, für alle C:
wenn $(x_{ij}, *, *) \notin \mathcal{F}$: $\mathcal{F} := \mathcal{F} \cup \{(x_{ij}, x_C, \infty)\}$
wenn $(x_{ij}, y, \infty) \in \mathcal{F}$: $\mathcal{F} := (\mathcal{F} \setminus \{x_{ij}, y, \infty\}) \cup \{(x_{ij}, y, x_C)\}$
$\mathcal{B} = \emptyset$, für $(x, y, z) \in \mathcal{F}$: wenn $z = \infty$: $\mathcal{B} := \mathcal{B} \cup \{x\}$.

Paralleles Verfeinern:

a) Bestimmen von $\mathcal{B}$ auf Prozess $p = 0$

b) Verteilen von $\mathcal{B} = \bigcup_{p \in \mathcal{P}} \mathcal{B}_p$ (disjunkt)

c) Verfeinern von $\mathcal{C}$:
wenn $x_{ij} \in \mathcal{B}$, setze $B := \mathcal{B} \cup \{\frac{1}{2}(x_i + x_{ij}), \frac{1}{2}(x_j + x_{ij})\}$

d) $\pi(x)$ für $\mathcal{B} \subset \mathcal{V}$ wie bisher.

4.1 Paralleles Programm 6

4.1.1 Beschreibung

Die sechste Programmversion erweitert die früheren Versionen, so dass jetzt das eingelesene Netz nach der Verteilung schrittweise uniform verfeinert werden kann. Für jeden Verfeinerungsschritt werden die Anzahl der Zellen, die Größe des **ProcSets** und die Gesamtfläche des Netzes von den Prozessoren ausgegeben.

4.1.2 Dateien

```
PFEM/06/Parallel.h
PFEM/06/Parallel.C
PFEM/06/Point.h
PFEM/06/Mesh.h
PFEM/06/Main.C
PFEM/06/Makefile
PFEM/06/mesh
```

4.1.3 Lastverteilung und Gitterverfeinerung

```
1  // file:    PFEM/06/Mesh.h
2
3  #ifndef _MESH_H_
4  #define _MESH_H_
5
6  #include "Parallel.h"
```

```
7
8  #include <fstream>
9  #include <vector>
10
11 class Mesh {
12  public:
13     class Cell { ... };
14  private:
15     list<Cell> Cells;
16     hash_set<Point,Hash> Vertices;
17     void Insert (const Point& y) { ... }
18  public:
19     Mesh () {}
20     void Insert (const Cell& C) { ... }
21     void Refine (const Cell& C, ProcSets& PS) {
22         Point x01 = 0.5*(C[0]+C[1]);
23         Point x12 = 0.5*(C[1]+C[2]);
24         Point x02 = 0.5*(C[0]+C[2]);
25         PS.Copy(x01,0.5*(C[0]+x01));
26         PS.Copy(x01,0.5*(C[1]+x01));
27         PS.Copy(x12,0.5*(C[1]+x12));
28         PS.Copy(x12,0.5*(C[2]+x12));
29         PS.Copy(x02,0.5*(C[0]+x02));
30         PS.Copy(x02,0.5*(C[2]+x02));
31         Insert(Mesh::Cell(C[0],x01,x02));
32         Insert(Mesh::Cell(C[1],x12,x01));
33         Insert(Mesh::Cell(C[2],x02,x12));
34         Insert(Mesh::Cell(x01,x12,x02));
35     }
36     void Refine (ProcSets& PS) {
37         for (list<Cell>::iterator c=Cells.begin();c!=Cells.end();) {
38             Refine(*c,PS); list<Cell>::iterator cc = c++;
39             Cells.erase(cc);
40         }
41     }
42     double area () { ... }
43     int size() { return Cells.size(); }
44     bool Vertex (const Point& y) { ... }
45 };
46
47 inline istream& operator >> (istream& s, vector<Mesh::Cell>& C) { ... }
48
49 inline void Distribute (Mesh& mesh, ProcSets& PS) {
50     ExchangeBuffer E;
51     if (PPM->master()) {
52         ifstream s("mesh");
53         int M; s >> M;
54         vector<Mesh::Cell> C(M); s >> C;
```

```
55          sort(C.begin(),C.end());
56          int N = (M + PPM->size() - 1) / PPM->size();
57          for (int m=0; m<M; ++m) {
58              int dest = m / N;
59              if (dest == 0) mesh.Insert(C[m]);
60              for (int k=0; k<3; ++k) PS.Add(C[m][k],dest);
61              PS.Add(0.5*(C[m][0]+C[m][1]),dest);
62              PS.Add(0.5*(C[m][1]+C[m][2]),dest);
63              PS.Add(0.5*(C[m][0]+C[m][2]),dest);
64          }
65          for (procset p=PS.procsets(); p!=PS.procsets_end(); ++p) {
66              for (int i=0; i<p->second.size(); ++i) {
67                  int dest = p->second[i];
68                  if (dest == 0) continue;
69                  if (p->second.size() == 1) continue;
70                  E.Send(dest) << short(p->second.size());
71                  E.Send(dest) << p->first;
72                  for (int j=0; j<p->second.size(); ++j)
73                      E.Send(dest) << short(p->second[j]);
74              }
75          }
76          for (short q=1; q<PPM->size(); ++q) E.Send(q) << short(0);
77          for (int m=0; m<M; ++m) {
78              int dest = m / N;
79              if (dest != 0) E.Send(dest) << C[m];
80          }
81      }
82      E.CommunicateSizeBuffer();
83      if (PPM->master())
84          PS.Clean();
85      else {
86          short s; E.Receive(0) >> s;
87          while (s > 0) {
88              Point y;
89              E.Receive(0) >> y;
90              for (int i=0; i<s; ++i) {
91                  short dest;
92                  E.Receive(0) >> dest;
93                  PS.Add(y,dest);
94              }
95              E.Receive(0) >> s;
96          }
97          while (E.Receive(0).size() < E.Receive(0).Size()) {
98              Mesh::Cell C; E.Receive(0) >> C;
99              mesh.Insert(C);
100         }
101     }
102 }
```

```
103 #endif
```

In den vorangegangenen Versionen des parallelen Programms, wurde die Aufteilung der Dreiecke eines Gitters stets im Hauptprogramm vorgenommen. In der sechsten Version wurde dazu folgende Funktion definiert:

- `inline void Distribute (Mesh& mesh, ProcSets& PS)`
  Diese Funktion nimmt die parallele Lastverteilung vor. Zunächst werden vom Master-Prozess die Dreiecke des Gitters aus der Datei „`mesh`“ in einen Vektor eingelesen. Die Dreiecke werden dann entsprechend ihrer Lage im Gitter sortiert und auf die parallelen Prozesse aufgeteilt.
  Der Master-Prozess fügt zunächst die ihm zugeteilten Dreiecke in sein lokales Teilgitter ein (59–60). Um eine spätere Verfeinerung der Dreiecke zu ermöglichen, werden für die Kantenmittelpunkte aller Dreieckskanten Prozessmengen berechnet und in das lokale `ProcSets`-Objekt des Master-Prozesses eingetragen (61–63).
  Anschließend werden die Prozessmengen der Punkte, auf denen mindestens zwei Prozesse operieren, an die entsprechenden Prozesse gesendet. Das geschieht in folgender Weise: Befindet sich eine Prozessnummer `q` in der Prozessmenge eines Knotenpunktes `x`, so werden an den Prozess `q` der Reihe nach die Anzahl der Prozessnummern in der Prozessmenge, der Punkt `x` und danach alle Prozessnummern in der Prozessmenge gesendet (65–74). Wurden alle Prozessmengen auf diese Weise versendet, wird an jeden Prozess der Wert `0` gesendet (76).
  Anschließend werden nacheinander alle Dreiecke an die entsprechenden Prozesse gesendet. Mit dem Aufruf von `E.CommunicateSizeBuffer();` wird der lokale Kommunikationszyklus durchgeführt (76–82).
  Der Master-Prozess bereinigt dann sein lokales `ProcSets`-Objekt. Alle übrigen Prozesse lesen zunächst die Prozessmengen der einzelnen Punkte aus ihrem Empfangspuffer aus, und bauen damit ihre lokalen `ProcSets`-Objekte auf. Dies geschieht so lange, bis die einzelnen Prozesse den Wert `0` aus ihrem Empfangspuffer auslesen. Danach liest jeder parallele Prozess die an ihn gesendeten Dreiecke aus dem Empfangspuffer und baut so sein lokales Teilgitter auf (86–104).

In dieser Version des parallelen Programms wird die Gitterverfeinerung realisiert. Ein Dreieck des Gitters wird dabei in vier neue Dreiecke unterteilt. Dadurch entstehen neue Dreiecke und neue Knotenpunkte. Die Verfeinerung wird in der Klasse **Mesh** implementiert, die dazu um die folgenden Methoden erweitert wurde:

- `void Mesh::Refine (const Cell& C, ProcSets& PS)`
  Verfeinert ein grobes Dreieck `C` des Gitters in drei verfeinerte Eckendreiecke und ein verfeinertes, inneres Dreieck. Zunächst werden die Kantenmittelpunkte des groben Dreiecks `C` berechnet (22–24). Die Prozessmengen dieser Kantenmittelpunkte müssen bereits im übergebenen `ProcSets`-Hash `PS` gespeichert sein. Dies ist

bei der ersten Verfeinerungsstufe durch die Funktion `Distribute` sicher gestellt (siehe oben).
Für die weiteren Verfeinerungsstufen müssen jedoch folgende Vorbereitungen getroffen werden: Jedem Mittelpunkt der Verbindungsstrecke zwischen einem Eckpunkt von `C` und einem Kantenmittelpunkt wird die Prozessmenge des Kantenmittelpunktes zugewiesen (25–30). Diese Mittelpunkte sind bei der nächsten Verfeinerungsstufe die Kantenmittelpunkte der Eckendreiecke. Auf den Kantenmittelpunkten des inneren Dreiecks operiert nur der aufrufende Prozess. Da Punkte, auf denen nur ein Prozess operiert, aus einem `ProcSets`-Hash wieder entfernt werden, werden die Kantenmittelpunkte des inneren Dreiecks nicht weiter berücksichtigt.
Die neuen Dreiecke werden abschließend in das Gitter eingefügt (31–34).

- `void Mesh::Refine (ProcSets& PS)`
  Verfeinert alle Dreiecke des Gitters uniform und entfernt die groben Dreiecke aus dem Gitter.

- `int Mesh::size()`
  Gibt die Anzahl aller Dreiecke im Gitter zurück.

4.1.4 Das Hauptprogramm

```
1  // file:     PFEM/06/Main.C
2
3  #include "Mesh.h"
4
5  const int levels = 1;
6
7  int main(int argv, char** argc) {
8      PPM = new ParallelProgrammingModel(&argv,&argc);
9      Mesh M;
10     ProcSets PS;
11     Distribute(M,PS);
12     pout << "number of coarse cells  " << PPM->Sum(M.size()) << endl;
13     pout << "ProcSets size " << PS.size() << endl;
14     pout << "area " << PPM->Sum(M.area()) << endl;
15     for (int l=0; l<levels; ++l) {
16         M.Refine(PS);
17         pout << "number of refined cells " << PPM->Sum(M.size()) << endl;
18         pout << "ProcSets size " << PS.size() << endl;
19         pout << "area " << PPM->Sum(M.area()) << endl;
20     }
21     delete PPM;
22     return 0;
23 }
```

5 Ausgabe und Visualisierung der Ergebnisse

Problem

Darstellung von über 1 Million Werten (8 MB) in 1 Million Zellen (48 MB) auf 100 Prozessoren auf einem Bildschirm (1 Million Pixel) $\longrightarrow$ Datenreduktion und Auswertung erforderlich $\longrightarrow$ Einsammeln erforderlich.

Low-Level Lösung (nur bedingt einsetzbar): wähle Visualisierungsprogramm, das Datenfiles einlesen kann (hier: gnuplot malt Linien in 2D / 3D) z.B. AVS, OpenDX, Tecplot, GMV

1) auf Prozessor p: $B(p,0) = \emptyset$ für alle $C \in \mathcal{C}_p$: $B(p,0) = B(p,0) \cup \big\{(c, u(x_0), u(x_1), u(x_2)\big\}$
2) Austausch von B
3) Auf Prozessor $p = 0$: lese $(x_0, x_1, x_2, u_0, u_1, u_2) \in B(0,q),\ q \in \mathcal{P}$ und schreibe Daten in geeignetem Fileformat heraus, z.B. in gnuplot für zwei Zellen C, C':

Netz		Graph von u im $\mathbf{R}^3$:		
	x_0		x_0	u_0
	x_1		x_1	u_1
	x_2		x_2	u_2
	x_0		x_0	u_0
	x_0'		x_0'	u_0'
	x_1'		x_1'	u_1'
	x_2'		x_2'	u_2'
	x_0'		x_0'	u_0'

Bemerkung

I. A. erlauben Visualisierungsprogramme doppelte Knoten

- Interfaces werden doppelt gespeichert, die Datenfiles werden größer
- jeder Prozessor q kann seine Daten unabhängig formatieren und in $B(q,0)$ schreiben; auf Prozessor $p = 0$ wird $B(q,0)$ direkt in Visualisierungsfile geschrieben.

5.1 Paralleles Programm 7

5.1.1 Beschreibung

5.1.2 Dateien

PFEM/07/Parallel.h
PFEM/07/Parallel.C
PFEM/07/Point.h
PFEM/07/Mesh.h
PFEM/07/Main.C
PFEM/07/Makefile
PFEM/07/mesh
PFEM/07/out.gnu

5.1.3 Punkte und Dreiecke (Teil 4)

```
1  // file:     PFEM/07/Point.h
2
3  #ifndef _POINT_H_
4  #define _POINT_H_
5
6  #include <iostream>
7  using namespace std;
8
9  const double GeometricTolerance = 1e-8;
10
11 class Point { ... };
12
13 inline ostream& operator << (ostream& s, const Point& y) { ... }
14 inline Point operator - (const Point& y, const Point& z) { ... }
15 inline Point operator + (const Point& y, const Point& z) { ... }
16 inline Point operator * (double a, const Point& z) { ... }
17 inline double det(const Point& y,const Point& z) { ... }
18
19 class Hash { ... };
20
21 inline double norm (const Point& y) {
22     return sqrt(y[0]*y[0]+y[1]*y[1]);
23 }
24 inline double dist (const Point& y, const Point& x) {
25     return norm(y-x);
26 }
27 #endif
```

In dieser Version des parallelen Programms wurden für die Klasse **Point** zusätzlich folgende Funktionen definiert:

- `inline double norm (const Point& y)`
  Berechnet zu einem Punkt `y` den Wert der euklidische Norm und gibt diesen zurück.
- `inline double dist (const Point& y, const Point& x)`
  Gibt den euklidischen Abstand zweier Punkte `x` und `y` zueinander zurück.

5.1.4 Die Randerkennung

```
1  // file:     PFEM/07/Mesh.h
2
3  #ifndef _MESH_H_
4  #define _MESH_H_
5
6  #include "Parallel.h"
7
8  #include <fstream>
9  #include <vector>
10
11 class Mesh {
12  public:
13     class Cell { ... };
14     class cell : public list<Cell>::const_iterator {
15         typedef list<Cell>::const_iterator Iterator;
16      public:
17         cell(const Iterator& c) : Iterator(c) {};
18         const Point& operator [] (int i) const { return (*(*this))[i]; }
19         Point operator () () const { return (*(*this))(); }
20     };
21  private:
22     list<Cell> Cells;
23     hash_set<Point,Hash> Vertices;
24     hash_set<Point,Hash> Boundaries;
25     void Insert (const Point& y) { Vertices.insert(y); }
26  public:
27     Mesh () {}
28     cell cells () const { return cell(Cells.begin()); }
29     cell cells_end () const { return cell(Cells.end()); }
30     void Insert (const Cell& C) { ... }
31     double area () { ... }
32     double boundary () {
```

```
33          double L = 0;
34          for (cell c=cells(); c!=cells_end(); ++c) {
35              if (Boundary(0.5*(c[0]+c[1]))) L += dist(c[0],c[1]);
36              if (Boundary(0.5*(c[1]+c[2]))) L += dist(c[1],c[2]);
37              if (Boundary(0.5*(c[0]+c[2]))) L += dist(c[0],c[2]);
38          }
39          return L;
40      }
41      int size() const { ... }
42      bool Vertex (const Point& y) { ... }
43      bool Boundary (const Point& y) {
44          return (Boundaries.find(y)!=Boundaries.end());
45      }
46      short Boundary (const Cell& C) {
47          short s = 0;
48          if (Boundary(0.5*(C[0]+C[1]))) ++s;
49          if (Boundary(0.5*(C[1]+C[2]))) ++s;
50          if (Boundary(0.5*(C[0]+C[2]))) ++s;
51          return s;
52      }
53      void InsertBoundary (const Point& y) {
54          if (Boundaries.find(y)!=Boundaries.end())
55              Boundaries.erase(y);
56          else
57              Boundaries.insert(y);
58      }
59      void Refine (const Cell& C, ProcSets& PS) {
60          Point x01 = 0.5*(C[0]+C[1]);
61          Point x12 = 0.5*(C[1]+C[2]);
62          Point x02 = 0.5*(C[0]+C[2]);
63          PS.Copy(x01,0.5*(C[0]+x01));
64          PS.Copy(x01,0.5*(C[1]+x01));
65          PS.Copy(x12,0.5*(C[1]+x12));
66          PS.Copy(x12,0.5*(C[2]+x12));
67          PS.Copy(x02,0.5*(C[0]+x02));
68          PS.Copy(x02,0.5*(C[2]+x02));
69          Insert(Mesh::Cell(C[0],x01,x02));
70          Insert(Mesh::Cell(C[1],x12,x01));
71          Insert(Mesh::Cell(C[2],x02,x12));
72          Insert(Mesh::Cell(x01,x12,x02));
73          if (Boundary(x01)) {
74              InsertBoundary(0.5*(C[0]+x01));
75              InsertBoundary(0.5*(C[1]+x01));
76          }
77          if (Boundary(x12)) {
78              InsertBoundary(0.5*(C[1]+x12));
79              InsertBoundary(0.5*(C[2]+x12));
80          }
```

```
81          if (Boundary(x02)) {
82              InsertBoundary(0.5*(C[0]+x02));
83              InsertBoundary(0.5*(C[2]+x02));
84          }
85      }
86      void Refine (ProcSets& PS) { ... }
87  };
88
89  inline istream& operator >> (istream& s, vector<Mesh::Cell>& C) { ... }
90
91  inline void Distribute (Mesh& mesh, ProcSets& PS) {
92      ExchangeBuffer E;
93      if (PPM->master()) {
94          ifstream s("mesh");
95          int M; s >> M;
96          vector<Mesh::Cell> C(M); s >> C;
97          sort(C.begin(),C.end());
98          int N = (M + PPM->size() - 1) / PPM->size();
99          for (int m=0; m<M; ++m) {
100             int dest = m / N;
101             if (dest == 0) mesh.Insert(C[m]);
102             for (int k=0; k<3; ++k) PS.Add(C[m][k],dest);
103             PS.Add(0.5*(C[m][0]+C[m][1]),dest);
104             PS.Add(0.5*(C[m][1]+C[m][2]),dest);
105             PS.Add(0.5*(C[m][0]+C[m][2]),dest);
106             mesh.InsertBoundary(0.5*(C[m][0]+C[m][1]));
107             mesh.InsertBoundary(0.5*(C[m][1]+C[m][2]));
108             mesh.InsertBoundary(0.5*(C[m][0]+C[m][2]));
109         }
110         for (procset p=PS.procsets(); p!=PS.procsets_end(); ++p) {
111             for (int i=0; i<p->second.size(); ++i) {
112                 int dest = p->second[i];
113                 if (dest == 0) continue;
114                 if (p->second.size() == 1) continue;
115                 E.Send(dest) << short(p->second.size());
116                 E.Send(dest) << p->first;
117                 for (int j=0; j<p->second.size(); ++j)
118                     E.Send(dest) << short(p->second[j]);
119             }
120         }
121         for (short q=1; q<PPM->size(); ++q) E.Send(q) << short(0);
122         for (int m=0; m<M; ++m) {
123             int dest = m / N;
124             if (dest != 0) {
125                 E.Send(dest) << C[m];
126                 short s = mesh.Boundary(C[m]);
127                 E.Send(dest) << s;
128                 if (s) {
```

```
129                     Point y = 0.5*(C[m][0]+C[m][1]);
130                     if (mesh.Boundary(y)) E.Send(dest) << y;
131                     y = 0.5*(C[m][1]+C[m][2]);
132                     if (mesh.Boundary(y)) E.Send(dest) << y;
133                     y = 0.5*(C[m][0]+C[m][2]);
134                     if (mesh.Boundary(y)) E.Send(dest) << y;
135                 }
136             }
137         }
138     }
139     E.CommunicateSizeBuffer();
140     if (PPM->master())
141         PS.Clean();
142     else {
143         short s; E.Receive(0) >> s;
144         while (s > 0) {
145             Point y;
146             E.Receive(0) >> y;
147             for (int i=0; i<s; ++i) {
148                 short dest;
149                 E.Receive(0) >> dest;
150                 PS.Add(y,dest);
151             }
152             E.Receive(0) >> s;
153         }
154         while (E.Receive(0).size() < E.Receive(0).Size()) {
155             Mesh::Cell C; E.Receive(0) >> C;
156             mesh.Insert(C);
157             short s;
158             E.Receive(0) >> s;
159             for (short i=0; i<s; ++i) {
160                 Point y;
161                 E.Receive(0) >> y;
162                 mesh.InsertBoundary(y);
163             }
164         }
165     }
166 }
167
168 #endif
```

In der siebten Version des parallelen Programms wurde die Klasse `Mesh` erweitert. Zunächst definiert diese Klasse eine Iteratorklasse **Mesh::cell**, mit der die Einträge einer `Cell`-wertigen Liste durchlaufen werden können. Für die Iteratorklasse `Mesh::cell` sind die folgenden Methoden und Operatoren definiert:

- `Mesh::cell::cell(const Iterator& c)`

Konstruktor: Erzeugt einen neuen Iterator der Klasse `cell`. Der aktuelle Eintrag des erzeugten Iterators ist ein Eintrag in einer `Cell`-wertigen Liste, der von dem Iterator `c` bezeichnet wird.

- `const Point& Mesh::cell::operator [] (int i)`
  Ist `c` eine `Mesh::cell`-Iterator, so gibt `c[i]` den `i`-ten Eckpunkt des Dreiecks zurück, das dem aktuellen Eintrag in einer `Cell`-wertigen Liste entspricht.

- `Point Mesh::cell::operator () ()`
  Ist `c` eine `Mesh::cell`-Iterator, so gibt `c()` den Schwerpunkt des Dreiecks zurück, das dem aktuellen Eintrag in einer `Cell`-wertigen Liste entspricht.

Die Klasse **Mesh** besitzt in dieser Version des parallelen Programms ein weiteres privates Hash `Boundaries`, in dem die Kantenmittelpunkte der Randkanten eines Gitters gespeichert sind. Um die Randerkennung zu realisieren, wurden einige Methoden der Klasse verändert und andere neu definiert. Diese Methoden sind im einzelnen:

- `cell Mesh::cells ()`
  Gibt einen `Mesh::cell`-Iterator zurück, der den ersten Eintrag in der privaten Liste `Cells` bezeichnet.

- `cell Mesh::cells_end ()`
  Gibt einen `Mesh::cell`-Iterator zurück, der das Ende der privaten Liste `Cells` bezeichnet.

- `void Mesh::InsertBoundary (const Point& y)`
  Diese Methode nimmt die Randerkennung im Gitter vor. Sie fügt den Kantenmittelpunkt `y` einer Dreieckskante in das private Hash `Boundaries` ein, sofern dieser Punkt dort nocht nicht vorhanden ist. Ist der Punkt jedoch im Hash bereits vorhanden, so wird er wieder aus dem Hash entfernt. Dieser Vorgehensweise liegt die Idee zugrunde, dass innere Kanten des Gittes an genau zwei Dreiecken anliegen, während Randkanten nur an einem Dreieck anliegen. Wenn man also für alle Kantenmittelpunkte aller Dreiecke im Gitter diese Methode aufruft, so verbleiben nur die Kantenmittelpunkte der Randkanten im Hash `Boundaries`.

- `bool Mesh::Boundary (const Point& y)`
  Gibt zu einem Kantenmittelpunkt `y` eines Dreiecks den Wert `true` zurück, wenn es sich bei der Kante um eine Randkante handelt.

- `short Mesh::Boundary (const Cell& C)`
  Gibt zu einem Dreieck `C` des Gitters die Anzahl der Randkanten zurück.

- `double Mesh::boundary ()`
  Gibt den Umfang des Gitters, d.h. die Summe aller Randkantenlängen, zurück.

- `void Mesh::Refine (const Cell& C, ProcSets& PS)`
  Die Methode `Refine` wurde dahingehend verändert, dass die für die Kantenmittelpunkte der nächsten Verfeinerungsstufe die Methode `InsertBoundary` aufgerufen wird, wenn diese auf einer Randkante liegen.

Auch die Lastverteilungsfunktion `Distribute` wurde verändert:

- `inline void Distribute (Mesh& mesh, ProcSets& PS)`
  Die Funktion wurde wie folgt verändert: Nachdem die Gitterdreiecke vom Master-Prozess aus der Datei „`mesh`“ eingelesen und die Prozessmengen der Kantenmittelpunkte bestimmt wurden, wird die Randerkennung durchgeführt. Dazu wird für alle Kantenmittelpunkte aller Dreiecke die Methode `InsertBoundary` aufgerufen (108–110).
  Auch das Versenden und Empfangen der Dreiecke wurde verändert. Es werden nun der Reihe nach das Dreieck, die Anzahl der Randkanten des Dreiecks und danach die Kantenmittelpunkte der Randkanten des Dreiecks versendet (128–137). Mit diesen Daten baut jeder parallele Prozess anschließend sein lokales Teilgitter auf, in welchem dann auch die Information über die Randkanten vorhanden ist (159–165).

5.1.5 Die graphische Ausgabe (Teil 1)

```
1  // file:     PFEM/07/Main.C
2
3  #include "Mesh.h"
4
5  inline void Plot (Mesh& M) {
6      ExchangeBuffer E;
7      for (Mesh::cell c = M.cells(); c != M.cells_end(); ++c) E.Send(0) << *c;
8      E.CommunicateSizeBuffer();
9      if (PPM->master()) {
10         ofstream s("out");
11         Mesh::cell c = M.cells();
12         s << c[0] << endl;
13         for (int q=0; q<PPM->size(); ++q)
14             while (E.Receive(q).size() < E.Receive(q).Size()) {
15                 Mesh::Cell C; E.Receive(q) >> C;
16                 s << C[0] << endl << C[1] << endl
17                   << C[2] << endl << C[0] << endl << endl;
18             }
19     }
20 }
```

```
21
22 const int levels = 1;
23
24 int main(int argv, char** argc) {
25     PPM = new ParallelProgrammingModel(&argv,&argc);
26     Mesh M;
27     ProcSets PS;
28     Distribute(M,PS);
29     pout << "number of coarse cells  " << PPM->Sum(M.size()) << endl;
30     pout << "ProcSets size " << PS.size() << endl;
31     pout << "area " << PPM->Sum(M.area()) << endl;
32     pout << "boundary " << PPM->Sum(M.boundary()) << endl;
33     for (int l=0; l<levels; ++l) {
34     M.Refine(PS);
35     pout << "number of refined cells " << PPM->Sum(M.size()) << endl;
36     pout << "ProcSets size " << PS.size() << endl;
37     pout << "area " << PPM->Sum(M.area()) << endl;
38     pout << "boundary " << PPM->Sum(M.boundary()) << endl;
39     }
40     Plot(M);
41     delete PPM;
42     return 0;
43 }
```

Neben der Randerkennung wird in dieser Version des parallelen Programms auch eine graphische Ausgabe des Gitters realisiert. Dazu wurde folgende Funktion definiert:

- `inline void Plot (Mesh& M)`
  Die Funktion `Plot` erzeugt eine Textdatei mit dem Dateinamen `out`. Das Programm *gnuplot* verwendet diese Datei, um die Dreiecke des Gitters zu zeichnen. Zunächst senden alle parallelen Prozesse die Dreiecke ihrer lokalen Teilgitter an den Master-Prozess (9–10). Der Master-Prozess erzeugt danach die Ausgabedatei (11–21). Ein Dreieck wird dabei durch einen Kantenzug repräsentiert, der von einem Eckpunkt ausgehend die übrigen Eckpunkte durchläuft und wieder zum Ausgangspunkt zurück verbindet. In der Datei `out` werden deshalb pro Dreieck die Koordinaten von vier Punkten gespeichert.

In der Datei `out.gnu` werden die Parameter für das Programm *gnuplot* definiert. Durch den Kommandozeilenaufruf

```
gnuplot out.gnu -
```

wird *gnuplot* gestartet.

6 Matrixgraphen

Zu einer Matrix $A \in \mathbf{R}^{N,N}$ ist

$$G(A) = \{(i,j)\colon\ A_{ij} \neq 0\} \subset \{0,\ldots,N-1\} \times \{0,\ldots,N-1\}$$

der zugehörige Matrixgraph.
Zu Finiten Elementen mit Knotenfreiheitsgraden

$$u\colon \mathcal{V} \longrightarrow \mathbf{R} \qquad (\mathcal{V} = \bigcup_{C\in\mathcal{C}} C)$$

betrachten wir Steifigkeitsmatrizen

$$A\colon \mathcal{V} \times \mathcal{V} \longrightarrow \mathbf{R}$$

mit Graphen

$$G(A) = \{(x,y)\colon\ A(x,y) \neq 0\}$$
$$G(A) \subset G(\mathcal{C}) := \big\{(x,y)\colon\ x,y \in C,\, C \in \mathcal{C}\big\} = \bigcup_{C\in\mathcal{C}} C \times C$$

Aufbau von $G(\mathcal{C}) = \bigcup_{x\in\mathcal{V}} G(x)$ mit $G(x) = \big\{y \in \mathcal{V}\colon\ (x,y) \in G(\mathcal{C})\big\}$
setze $G(x) = \emptyset$ für alle $x \in \mathcal{V}$

$$\text{für } C \in \mathcal{C}\colon \qquad \begin{aligned} G(x) &= G(x) \cup \{y\} \\ G(y) &= G(y) \cup \{x\} \end{aligned} \qquad x,y \in G$$

Eigenschaften

a) $(x,x) \in G(\mathcal{C}) \Longrightarrow x \in G(x)$ (Konvention: x in $G(x)$ nicht abspeichern)

b) $\deg(x) := |G(x)| = O(1)$ (Steifigkeitsmatrix dünn besetzt)

CRS = compressed row storage

Abspeichern einer dünn besetzten Matrix in einem Vektor A.data der Länge $|G(\mathcal{C})|$.
Zugriff durch

A.diag	integer Vektor der Länge $\|\mathcal{V}\| + 1$
A.col	integer Vektor der Länge $\|G(\mathcal{C})\|$

Beispiel

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & 0 \\ a_{10} & a_{11} & 0 & a_{13} \\ a_{20} & 0 & a_{22} & 0 \\ 0 & a_{31} & 0 & a_{33} \end{pmatrix}$$

$A.\text{data} = (\ a_{00}\ \ a_{01}\ \ a_{02}\ \ a_{11}\ \ a_{10}\ \ a_{13}\ \ a_{22}\ \ a_{20}\ \ a_{22}\ \ a_{31}\ )$

$A.\text{col} = (\ 0\ \ 1\ \ 2\ \ 1\ \ 0\ \ 3\ \ 2\ \ 0\ \ 3\ \ 1\ )$

$A.\text{diag} = (\ 0\ \ 3\ \ 6\ \ 8\ \ 10\ )$

$A.\text{row} = (\ 0\ \ 0\ \ 0\ \ 1\ \ 1\ \ 1\ \ 2\ \ 2\ \ 3\ \ 3\ )$

Es gilt für $k = \text{A.diag}(i), ..., \text{A.diag}(i+1) - 1$: $\text{A.row}(k) = i$.

Aufbau der Datenstruktur

1) Nummerierung von $\mathcal{V}$ $(N = |\mathcal{V}|)$:

   $g\colon \mathcal{V} \longrightarrow \{0, \ldots, N-1\}$ bijektiv

2) Aufbau der Mengen $G(x)$
   definiere deg $\in \mathbf{Z}^N$ mit $\deg(g(x)) = |G(x)|$
   definiere $\text{diag} \in \mathbf{Z}^{N+1}$ mit $\text{diag}(0) = 0$ und
   $\text{diag}(i+1) = \text{diag}(i) + \deg(i) \quad i = 0, \ldots, N$

3) Nummerierung von $G(x)$:

   $g(x, \cdot)\colon G(x) \longrightarrow \{\text{diag}(g(x)), \ldots, \text{diag}(g(x)+1) - 1\}$ bijektiv

   Konvention: $g(x,x) = \text{diag}(g(x))$
   setze $\text{col}(g(x,y)) = g(y)$ für $x \in \mathcal{V},\ y \in G(x)$
   Konvention: $g(x,y) = -1$ für $(x,y) \notin G(\mathcal{C})$ (d.h. $y \notin G(x)$)

Operationen

A) Zugriff auf A

$$A(x,y) = \text{A.data } (g(x,y))$$

B) Matrix-Vektor-Multiplikation $f = A\,u$

$f \equiv 0$
für $x \in \mathcal{V}$
  für $y \in G(x)$
    $f(\mathrm{col}(g(x,y)))+= \text{ A.data } (g(x,y))\,u(g(x))$

$\Longleftrightarrow$

$f \equiv 0$
für $i = 0, \ldots, N-1$
  für $e = \mathrm{diag}(i), \ldots, \mathrm{diag}(i+1) - 1$
    $j = \mathrm{col}(e)$
    $f(j)+= \text{A.data } (e) \cdot u(i)$

Alternativen

– Element-Node-Tabelle
  Nummerierung von $\mathcal{V} = \{x_0, \ldots, x_{N-1}\}$ und
  Nummerierung von $\mathcal{C} = \{C_0, \ldots, C_{M-1}\}$
  definiere: $c(m,j)$ mit $m = 0, \ldots, M-1, \quad j = 0,1,2$
  durch $\mathcal{C} = \{(x_{c(m,0)},\, x_{c(m,1)},\, x_{c(m,2)}) \colon \quad m = 0, \ldots, M-1\}$
  Zugriff auf A_{ij} :

  suche $e \in \{\mathrm{diag}(i), \ldots,\ \mathrm{diag}(i+1) - 1\}$ mit $\mathrm{col}(e) = j$
  $\Longrightarrow \quad A_{ij} = \text{A.data } (e)$

– Assemblierungstabelle $e(m,j,k) \quad j,k = 0,1,2$
  mit $A(x_{c(m,j)},\, x_{c(m,k)}) = \text{A.data } e(m,j,k))$

6.1 Paralleles Programm 8

6.1.1 Beschreibung

6.1.2 Dateien

PFEM/08/Parallel.h
PFEM/08/Parallel.C
PFEM/08/Point.h
PFEM/08/Mesh.h
PFEM/08/Plot.h
PFEM/08/Main.C
PFEM/08/Makefile
PFEM/08/mesh
PFEM/08/out.gnu

In der Datei „Plot.h“ ist die Funktion Plot für die graphische Ausgabe des Gitters definiert.

6.1.3 Matrixgraphen (Teil 1)

```
1  // file:     PFEM/08/Main.C
2
3  #include "Mesh.h"
4  #include "Plot.h"
5
6  const int levels = 0;
7
8  class MatrixGraph {
9      class Row : public hash_map<Point,int,Hash> {
10         int row;
11      public:
12         Row () : row(-1) {}
13         Row (int r) : row(r) {}
14         int operator () () const { return row; }
15     };
16     hash_map<Point,Row,Hash> Rows;
17     vector<int> diag;
18     vector<int> column;
19     void Insert (const Point& x, const Point& y) {
20         hash_map<Point,Row,Hash>::iterator r = Rows.find(x);
21         if (r == Rows.end()) {
22             int i = Rows.size();
23             Rows[x] = Row(i);
24         }
```

```
25          Rows[x][y] = -1;
26          r = Rows.find(y);
27          if (r == Rows.end())  {
28              int i = Rows.size();
29              Rows[y] = Row(i);
30          }
31          Rows[y][x] = -1;
32      }
33   public:
34      MatrixGraph (const Mesh& M) {
35          if (M.size()== 0) return;
36          for (Mesh::cell c=M.cells(); c!=M.cells_end(); ++c) {
37              Insert(c[0],c[1]);
38              Insert(c[1],c[2]);
39              Insert(c[0],c[2]);
40          }
41          diag.resize(Rows.size()+1);
42          for (hash_map<Point,Row,Hash>::const_iterator r=Rows.begin();
43            r!=Rows.end(); ++r) diag[1+r->second()] = r->second.size();
44          diag[0] = 0;
45          for (int i=0; i<Rows.size(); ++i) diag[i+1] += diag[i]+1;
46          column.resize(diag[Rows.size()],-1);
47          for (int i=0; i<Rows.size(); ++i) column[diag[i]] = i;
48          for (hash_map<Point,Row,Hash>::iterator r=Rows.begin();
49            r!=Rows.end();++r) {
50              int e = diag[r->second()] + 1;
51              for (Row::iterator c=r->second.begin(); c!=r->second.end(); ++c) {
52                  c->second = e;
53                  column[e++] = Rows.find(c->first)->second();
54              }
55          }
56      }
57      int size () const { return Rows.size(); }
58      int Diag (int i) const { return diag[i]; }
59      int Column (int i) const { return column[i]; }
60  };
61
62  int main(int argv, char** argc) {
63      PPM = new ParallelProgrammingModel(&argv,&argc);
64      Mesh M;
65      ProcSets PS;
66      Distribute(M,PS);
67      int m = PPM->Sum(M.size());
68      mout << m << " coarse cells on " << PPM->size() << " processors" << endl;
69      for (int l=0; l<levels; ++l) M.Refine(PS);
70      m = PPM->Sum(M.size());
71      mout << m << " fine cells on " << PPM->size() << " processors" << endl;
72      MatrixGraph G(M);
```

```
73      pout << G.size() << " unknowns on proc " << PPM->proc() << endl;
74      pout << G.Diag(G.size()) << " matrix entries on proc "
75       << PPM->proc() << endl;
76      Plot(M);
77      delete PPM;
78      return 0;
79  }
```

In dieser Version des parallelen Programms werden Matrixgraphen eingeführt. Dazu wird zunächst die Klasse `MatrixGraph::Row` definiert, die ein `int`-wertiges Hash mit `Point`-wertigen Schlüsseln ist. Die Methoden und Operatoren dieser Klasse sind:

- `MatrixGraph::Row::Row ()`
  Konstruktor: Erzeugt ein neues Objekt der Klasse `MatrixGraph::Row` und weist der so erzeugten Matrixzeile die Zeilennummer `-1` zu.
- `MatrixGraph::Row::Row (int r)`
  Konstruktor: Erzeugt ein neues Objekt der Klasse `MatrixGraph::Row` und weist der so erzeugten Matrixzeile die Zeilennummer `r` zu.
- `int MatrixGraph::Row::operator () ()`
  Gibt die Zeilennummer der Matrixzeile zurück.

Die Klasse **MatrixGraph** implementiert den Matrixgraphen zu einem Finite-Element-Gitter in der CRS-Variante. Ein Matrixgraph weist jedem Punktepaar des Gitters, das über eine Dreieckskante verbunden ist, eine Zeilennummer und Spaltennummer in einer Steifigkeitsmatrix zu. Dazu sind folgende Methoden für die Klasse `MatrixGraph` definiert:

- `MatrixGraph::MatrixGraph (const Mesh& M)`
  Konstruktor: Erzeugt ein neues `MatrixGraph`-Objekt zu einem Gitter `M`. Die Methode durchläuft zuerst alle Dreiecke des Gitters, und ruft für je zwei Eckpunkte `x` und `y` eines Dreiecks die private Methode `Insert` auf (38–42). Diese Methode weist `x` ein `MatrixGraph::Row`-Hash `Rows[x]` zu. In dem gewählten Hash wird dann für `y` ein Eintrag `Rows[x][y]` bestimmt, der mit dem Initialwert -1 belegt wird (23–27).
  In gleicher Weise wird ein Eintrag `Rows[y][x]` bestimmt. Dadurch wird für jede Kante zwischen zwei Knotenpunkten im Gitter ein solcher Eintrag erzeugt (28–33).
  In dem Vektor `column` wird zu jedem Eintrag `Rows[x][y]` ein Element allokiert, sowie ein zusätzliches Element pro Zeile `Rows[x]`. Die Einträge des Vektors `column` speichern die Spaltennummern Einträge `Rows[x][y]` in der Steifigkeitsmatrix. Der Vektor `diag` speichert zu einer Zeilennummer `i` in der Steifigkeitsmatrix die Position `diag[i]` im Vektor `column` ab, an der die Spaltennummer des `i`-ten Diagonalelements gespeichert ist (43-47). Nach Konstruktion enthalten die Elemente

an den Positionen `diag[i]` bis `diag[i+1]-1` im Vektor `column` die Spaltennummern der Nichtnulleinträge in der `i`-ten Zeile der Steifigkeitsmatrix. Der Eintrag `diag[N]` enthält die Anzahl aller Nichtnullelemente in der Steifigkeitsmatrix, wobei `N` die Anzahl der Zeilen in der Steifigkeitsmatrix ist.
Nun wird jede Zeile `Rows[x]` durchlaufen. Jedem Eintrag `Rows[x][y]` wird eine Position `i` im Vektor `column` zugewiesen. Dem Element `column[i]` wird dann die Zeilennummer der Zeile `Rows[y]` zugewiesen. Dies ist die Spaltennummer des Eintrags `Rows[x][y]` (48–57).
Das Ergebnis dieser Methode ist, dass jeder Kante zwischen zwei Knotenpunkten `x` und `y` des Gitters durch `Rows[x]()` die Zeilennummer und durch `column[ Rows[x][y]]` die Spaltennummer eines Elements der Steifigkeitsmatrix zugeordnet ist. Jedem Knotenpunkt `x` des Gitters ist durch `Rows[x]()` die Zeilen- und Spaltennummer eines Diagonalelements in der Steifigkeitsmatrix zugeordnet.

- `int MatrixGraph::size ()`
  Gibt die Anzahl der Zeilen in der Steifigkeitsmatrix zurück.

- `int MatrixGraph::Diag (int i)`
  Gibt den `i`-ten Eintrag des privaten Vektors `diag` zurück.

- `int MatrixGraph::Column (int i)`
  Gibt den `i`-ten Eintrag des privaten Vektors `column` zurück.

7 Paralleles Assemblieren

Sei $\mathcal{V} = \bigcup \mathcal{V}_p$, $N_p = |\mathcal{V}_p|$, Nummerierung $g_p \colon \mathcal{V}_p \to \{0, \ldots, N_p - 1\}$.
Ein parallel verteilter Vektor

$$(\underline{u}_p)_{p\in\mathcal{P}} \in \prod_{p\in\mathcal{P}} \mathbf{R}^{N_p}$$

mit der parallelen Konsistenzbedingung

$$\underline{u}_p(x) = \underline{u}_q(x) \qquad p, q \in \pi(x)$$

definiert eine (Knoten-) Funktion

$$u \colon \mathcal{V} \ \to \ \mathbf{R}$$

mit $u(x) = \underline{u}_p(g_p(x))$ (mit $p \in \pi(x)$ beliebig).
Sie lässt sich stückweise linear fortsetzen zu

$$v \in V_{\mathcal{C}} := \{w \in C(\bar{\Omega}_{\mathcal{C}}) \colon \quad w|_C \text{ linear für alle } C \in \mathcal{C}\}.$$

Die Fortsetzung v stimmt natürlich an den Knoten mit u überein.

Theorem 1 *$v \in V_{\mathcal{C}}$ ist eindeutig durch die Einschränkung $v|_{\mathcal{V}}$ definiert, und es gilt*

$$v(y) = \sum_{x\in\mathcal{V}} v(x)\varphi_x(y) \qquad y \in \bar{\Omega}_{\mathcal{C}} = \bigcup_{C\in\mathcal{C}} \bar{\Omega}_C$$

mit der Knotenbasis $\varphi_x \in V_{\mathcal{C}},\ \varphi_x(y) = \begin{cases} 1 & y = x \\ 0 & y \neq x \end{cases} \qquad x, y \in \mathcal{V}.$

$v \in V_{\mathcal{C}}$ wird eindeutig durch $(\underline{v}_p)_{p\in\mathcal{P}}$ mit $\underline{v}_p(x) = \underline{v}_q(x)$ für alle $p, q \in \pi(x)$ repräsentiert.

Im Folgenden beschränken wir uns auf Dirichletrandbedingungen.
Sei $\Gamma = \partial\Omega$ (i.A.: $\Gamma \subset \partial\Omega$). Definiere zu $d \in C(\Gamma)$

$$V_{\mathcal{C}}(d) := \{v \in V_{\mathcal{C}} \colon \ v(x) = d(x),\ x \in \Gamma \cap \mathcal{V}\}$$

Speziell zu $d = 0$ definiere den Dualraum

$$V'_{\mathcal{C}} := L(V_{\mathcal{C}}(0), \mathbf{R}) := \{\lambda \colon V_{\mathcal{C}}(0) \ \to \ \mathbf{R} \ \text{ linear } \}$$

Theorem 2 *$\lambda \in V'_{\mathcal{C}}$ ist eindeutig durch $(\lambda(\varphi_x))_{x\in\mathcal{V}\setminus\Gamma}$ bestimmt.*
Die Einschränkungen $\lambda|_{C(\bar{\Omega}_p)}$ mit $\bar{\Omega}_p = \bigcup_{C\in\mathcal{C}_p} \bar{\Omega}_C$ definieren

$$(\underline{r}_p)_{p\in\mathcal{P}} \in \prod_{p\in\mathcal{P}} \mathbf{R}^{N_p} \ \text{ mit } \ \lambda|_{C(\bar{\Omega}_p)}(\varphi_x|_{C(\bar{\Omega}_p)}) = \underline{r}_p(g_p(x)),$$

d.h. $\lambda(\varphi_x) = \sum\limits_{p\in\mathcal{P}} \underline{r}_p(x)$.

Also definieren $(\underline{r}_p)$, $(\tilde{\underline{r}}_p) \in \prod\limits_{p\in\mathcal{P}} \mathbf{R}^{N_p}$ *genau dann dieselbe Linearform* λ*, wenn gilt:*

$$\sum_{p\in\mathcal{P}} \underline{r}_p(x) = \sum_{p\in\mathcal{P}} \tilde{\underline{r}}_p(x).$$

Finite-Elemente-Problem

Finde $u \in V_{\mathcal{C}}(d)$, so dass

$$\int_{\bar{\Omega}_{\mathcal{C}}} \nabla u \cdot \nabla v \, dx = 0 \text{ für alle } v \in V_{\mathcal{C}}(0) \quad \Longleftrightarrow \quad u \in V_{\mathcal{C}}(d): \quad F(u) = 0,$$

wobei $F\colon V_{\mathcal{C}}(d) \longrightarrow V'_{\mathcal{C}}$ mit $u \longmapsto \big(v \longmapsto F(u)[v] := \int\limits_{\bar{\Omega}_{\mathcal{C}}} \nabla u \cdot \nabla v \, dx\big)$.

Sequentielle Lösung

1) Wähle $u \in V_{\mathcal{C}}(d)$ (z.B. $u(x) = d(x) \quad x \in \Gamma \cap \mathcal{V}$
$u(x) = 0 \qquad x \in \mathcal{V} \setminus \Gamma$)

2) berechne $r\colon \mathcal{V} \longrightarrow \mathbf{R}$ mit
$r(x) = F(u)[\varphi_x] \qquad x \in \mathcal{V} \setminus \Gamma$
$r(x) = 0 \qquad x \in \mathcal{V} \cap \Gamma$

3) falls $||r|| := \big(\sum\limits_{x\in\mathcal{V}} r(x)^2\big)^{1/2}$ klein genug: STOP

4) bestimme $A\colon \mathcal{V} \times \mathcal{V} \longrightarrow \mathbf{R}$ mit

$$A(x,y) = \begin{cases} DF(u)[\varphi_x, \varphi_y] & x, y \in \mathcal{V} \setminus \Gamma \\ 0 & \text{sonst} \end{cases}$$

und $DF(u)[v,w] = \int\limits_{\bar{\Omega}_{\mathcal{C}}} \nabla v \cdot \nabla w \, dx \quad v, w \in V_{\mathcal{C}}(0)$.

5) Bestimme $v\colon \mathcal{V} \longrightarrow \mathbf{R}$ mit

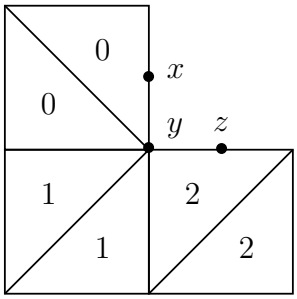
$$\sum_{y\in\mathcal{V}} A(x,y)v(y) = r(x) \qquad x \in \mathcal{V} \setminus \Gamma$$
$$v(x) = 0 \qquad x \in \mathcal{V} \cap \Gamma$$

6) Update setze $u := u - v$, gehe zu 2)

Parallele Lösung

Wir definieren $\mathcal{D}_p := \mathcal{V}_p \cap \Gamma$ als die Menge der Knoten auf dem Dirichletrand des Prozessors p.

Problem: Γ wird durch $\mathcal{B} \subset \Gamma$ identifiziert, aber $\mathcal{B} \cap \bar{\Omega}_p = \emptyset$ und $\mathcal{D}_p \neq \emptyset$ möglich!



$x \in \mathcal{B}_0 \implies y \in \mathcal{D}_0$, denn y liegt auf der durch x identifzierten Kante ,

$z \in \mathcal{B}_2 \implies y \in \mathcal{D}_2$, denn y liegt auf der durch z identifzierten Kante .

Aber: $y \in \mathcal{D}_p$ für alle $p \in \pi(x)$ obwohl es keine Randkante in $\bar{\Omega}_1$ gibt.

Die Dirichletwerte müssen somit kommuniziert werden.

1) $B(p,q) = \emptyset$, $\mathcal{D}_p = \emptyset$

Für alle $C = (x_0, x_1, x_2) \in \mathcal{C}_p$ auf Prozessor p:

wenn $x_{ij} := \frac{1}{2}(x_i + x_j) \in \mathcal{B}$:

$$B(p,q) := B(p,q) \cup \{(x_i,\, d(x_i))\} \quad q \in \pi(x_i)$$
$$B(p,q) := B(p,q) \cup \{(x_j,\, d(x_j))\} \quad q \in \pi(x_j)$$

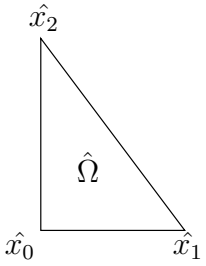
Austausch von $B(p,q)$

auf Prozessor q: für $(x,d) \in B(q,p)$ setze

$\underline{u}_q(g_q(x)) = d$, $\mathcal{D}_q := \mathcal{D}_q \cup \{x\}$

2) es gilt $r(y) = \int\limits_{\bar{\Omega}_{\mathcal{C}}} \nabla u \cdot \nabla \varphi_y \, dx = \sum\limits_{\substack{C \in \mathcal{C} \\ y \in C}} \int\limits_{\bar{\Omega}_C} \nabla u \cdot \nabla \varphi_y \, dx$

Berechnung im Referenzdreieck



$\hat{\Omega} = \operatorname{conv} \hat{C}$, $\hat{C} = (\hat{x}_0, \hat{x}_1, \hat{x}_2)$ $\quad \hat{x}_0 = \binom{0}{0}$, $\hat{x}_1 = \binom{1}{0}$, $\hat{x}_2 = \binom{0}{1}$

def. $T_C \colon \hat{\Omega} \longrightarrow \bar{\Omega}_C$ affin linear mit $T_C(\hat{x}_i) = x_i$

$\implies T_C(\hat{y}) = x_0 + DT_C(\hat{y} - \hat{x}_0), \quad DT_C = (x_1 - x_0,\, x_2 - x_0)$

Es gilt
$$\hat{\varphi}_0\binom{\hat{z}_1}{\hat{z}_2} = 1-\hat{z}_1-\hat{z}_2 \qquad \Longrightarrow \qquad \nabla\hat{\varphi}_0 = \binom{-1}{-1}$$
$$\hat{\varphi}_1\binom{\hat{z}_1}{\hat{z}_2} = \hat{z}_1 \qquad\qquad\qquad\quad \nabla\hat{\varphi}_1 = \binom{1}{0}$$
$$\hat{\varphi}_2\binom{\hat{z}_1}{\hat{\varphi}_2} = \hat{z}_2 \qquad\qquad\qquad\quad \nabla\hat{\varphi}_2 = \binom{0}{1}$$

$$\varphi_j(T_C\hat{z}) = \hat{\varphi}_j(\hat{z}) \qquad \hat{z}\in\hat{\Omega}, \qquad z = T_C(\hat{z})$$

$$\Longrightarrow \qquad D\varphi_j(T\hat{z})\cdot DT_C = D\hat{\varphi}_j(\hat{z})$$

$$\Longrightarrow \qquad (DT_C)^T \quad \nabla\varphi_j(z) = \nabla\hat{\varphi}_j(\hat{z})$$

$$\Longrightarrow \qquad \nabla\varphi_j(z) = (DT_C)^{-T}\,\nabla\hat{\varphi}_j(\hat{z}) =: \eta_j \text{ konstant auf } \hat{\Omega}$$

$$\Longrightarrow \qquad \int\limits_{\Omega_C}\nabla u\cdot\nabla\varphi_i\,dz = \int\limits_{\hat{\Omega}} |\det DT_C| \sum_{j=0}^{2} u(x_j)\eta_j\cdot\eta_i\,d\hat{z}$$
$$= \tfrac{1}{2}\ \det\,(x_1-x_0,\,x_2-x_0)\sum_{j=0}^{2}u(x_j)\eta_j\cdot\eta_i$$

Assemblieren von $(\underline{r}_p)_{p\in\mathcal{P}}$:

$\underline{r}_p = 0$

für alle $C=(x_0,\,x_1,\,x_2)\in\mathcal{C}_p$:

$$\delta = \det\,(x_1-x_0,\,x_2-x_0)$$
$$T = (DT_C)^{-T}$$
$$\eta_j = T\cdot\nabla\hat{\varphi}_j \qquad j=0,1,2$$
$$\eta = \sum_{j=0}^{2}\underline{u}_p(g(x_j))\eta_j$$
$$\underline{r}_p(g(x_i)) = \underline{r}_p(g(x_i)) + \tfrac{1}{2}\,\delta\,\eta\cdot\eta_i \qquad i=0,1,2$$

Für alle $x\in\mathcal{D}_p$: setze $\underline{r}_p(g_p(x)) = 0$.

3) Einsammeln der Residuenwerte (Collect)

Setze $\mu(x) = \min\pi(x)\in\mathcal{P}$.

$B(p,q) = \emptyset$

auf Prozessor p: für alle $x\in\mathcal{V}_p$ mit $p\neq\mu(x)$:

$$B(p,\,\mu(x)) := B(p,\,\mu(x))\cup\left\{(x,\,\underline{r}_q(g_p(x)))\right\}$$

Austausch von $B(p,q)$ $\qquad \underline{r}_p(g_p(x)) = 0$

auf Prozessor p: für $(x,\rho)\in B(q,p)$ setze $\underline{r}_p(g_p(x)) := \underline{r}_p(g_p(x)) + \rho$.

3) Nun gilt

$$||r||^2 = \sum_{p\in\mathcal{P}}\underline{r}_p^T\,\underline{r}_p$$

4) $\underline{A}_p = 0$

für $C = (x_0,\, x_1,\, x_2) \in \mathcal{C}_p$:

$$\rho = \text{ det } (x_1 - x_0,\, x_2 - x_0)$$
$$T = (DT_c)^{-T}$$
$$\eta_j = T \cdot \nabla\, \hat{\varphi}_j$$
$$\underline{A}_p(g_p(x_i,\, x_j)) = \underline{A}(g_p(x_i, x_j)) + \tfrac{1}{2}\, \rho\, \eta_i \cdot \eta_j$$

für $x \in \mathcal{D}_p$

$$i = g_p(x)$$
$$d = \mathrm{diag}_p(i)$$
$$\underline{A}_p(d) = 1$$
$$\text{für } \; e = d+1, \ldots, \mathrm{diag}_p(i+1) - 1$$
$$\underline{A}_p(e) = 0$$

5) Löse

$$\underline{A}\, \underline{v} = \underline{r}, \;\; \text{setze } \; \underline{u} := \underline{u} - \underline{v}$$

Theorem 3 *Sei* $(\underline{A}_p)$ *eine additive Darstellung von* A*, d.h.*

$$A(x,y) = \sum_{p \in \pi(x) \cap \pi(y)} \underline{A}_p(g_p(x,y)),$$

und sei $(\underline{v}_p)$ *konsistent, d.h.* $v(x) = \underline{v}_p(g_p(x)) \quad (p \in \pi(x))$.
Dann gilt: $(\underline{b}_p)$ *mit* $\underline{b}_p := \underline{A}_p\, \underline{v}$ *ist eine*
additive Darstellung von $b(x) = \sum\limits_{y \in \mathcal{V}} A(x,y)\; v(y) = \sum\limits_{p \in \pi(x)} \underline{b}_p(g_p(x))$

Einschub: Expression Templates

Problem: Die Implementation von

$\underline{u} = \underline{v} + \underline{w}$ als Vector operator + (Vector, Vector)

$\underline{b} = \underline{A} * \underline{u}$ als Vector operator $*$ (Matrix, Vector)

erfordert bei der Berechnung einen Hilfsvektor und bei der Zuweisung die Kopie eines Vektors.

Lösung Definiere Platzhalter (für Pointer auf die Objekte)

$VpV = \underline{v} + \underline{w}$ als $V\, p\, V$ operator + (Vector, Vector)

$MmV = \underline{A} * \underline{v}$ als $V\,p\,V$ operator $*$ (Matrix, Vector)
und Zuweisungen

$$\underline{u} = V\,p\,V \quad \text{als} \quad \text{Vector operator } = (V\,p\,V)\,\{$$
$$\text{for} \quad i = 0, \ldots, N-1$$
$$\text{this } [i] = V\,p\,V.\text{ first } [i]$$
$$+V_pV.\text{ second } [i];\}$$

$$\underline{b} = M\,m\,V \quad \text{als} \quad \text{Vector operator } = (M\,m\,V)\,\{$$
$$\text{for} \quad i = 0, \ldots, N-1$$
$$b = 0$$
$$\text{for } e = \mathrm{diag}(i), \ldots, \mathrm{diag}(i+1) - 1$$
$$j = \mathrm{column}(e)$$
$$b = b + M\,m\,V.\text{ first } (e) \cdot M\,m\,V.\text{ second}(j)$$
$$\text{this } (i) = b\}$$

Beachte: Die Realisation des Platzhalters als Template ermöglicht dem Prozessor die gesamte Berechnung in die Zuweisung zu verlagern, kein Rechenzeitverlust.

7.1 Paralleles Programm 9

7.1.1 Beschreibung

7.1.2 Dateien

```
PFEM/09/Parallel.h
PFEM/09/Parallel.C
PFEM/09/Point.h
PFEM/09/Mesh.h
PFEM/09/Algebra.h
PFEM/09/Plot.h
PFEM/09/Main.C
PFEM/09/Makefile
PFEM/09/mesh
PFEM/09/out.gnu
```

7.1.3 Matrixgraphen (Teil 2)

```
1  // file:      PFEM/09/Algebra.h
2
3  #ifndef _ALGEBRA_H_
4  #define _ALGEBRA_H_
5
6  #include "Mesh.h"
7
8  #include <valarray>
9
10 class MatrixGraph {
11     class Row : public hash_map<Point,int,Hash> { ... };
12  public:
13     class row : public hash_map<Point,Row,Hash>::const_iterator {
14         typedef hash_map<Point,Row,Hash>::const_iterator Iterator;
15       public:
16         row (const Iterator& r) : Iterator(r) {};
17         const Point& operator () () const { return (*this)->first; }
18     };
19  private:
20     const ProcSets& PS;
21     hash_map<Point,Row,Hash> Rows;
22     vector<bool> dirichlet;
23     vector<int> diag;
24     vector<int> column;
25     void Insert (const Point& x, const Point& y) { ... }
26  public:
27     MatrixGraph (const Mesh& M, const ProcSets& ps) : PS(ps) {
28         if (M.size()== 0) return;
29         for (Mesh::cell c=M.cells(); c!=M.cells_end(); ++c) {
30             Insert(c[0],c[1]);
31             Insert(c[1],c[2]);
32             Insert(c[0],c[2]);
33         }
34         diag.resize(Rows.size()+1);
35         for (hash_map<Point,Row,Hash>::const_iterator r=Rows.begin();
36           r!=Rows.end(); ++r) diag[1+r->second()] = r->second.size();
37         diag[0] = 0;
38         for (int i=0; i<Rows.size(); ++i) diag[i+1] += diag[i]+1;
39         column.resize(diag[Rows.size()],-1);
40         for (int i=0; i<Rows.size(); ++i) column[diag[i]] = i;
41         for (hash_map<Point,Row,Hash>::iterator r=Rows.begin();
42           r!=Rows.end();++r) {
43             int e = diag[r->second()] + 1;
44             for (Row::iterator c=r->second.begin(); c!=r->second.end(); ++c) {
45                 c->second = e;
46                 column[e++] = Rows.find(c->first)->second();
```

```
47              }
48          }
49          dirichlet.resize(Rows.size());
50          for (int i=0; i<Rows.size(); ++i) dirichlet[i] = false;
51      }
52      int size () const { return Rows.size(); }
53      int Size () const { return diag[Rows.size()]; }
54      int Diag (int i) const { return diag[i]; }
55      bool Dirichlet (int i) const { return dirichlet[i]; }
56      bool Dirichlet (const Point& x) const {
57          return dirichlet[Rows.find(x)->second()];
58      }
59      void SetDirichlet (const Point& x) {
60          dirichlet[Rows.find(x)->second()] = true;
61      }
62      int Column (int e) const { ... }
63      int operator () (const Point& x) const { return Rows.find(x)->second(); }
64      int operator () (const Point& x, const Point& y) const {
65          if (x == y) return diag[Rows.find(x)->second()];
66          return Rows.find(x)->second.find(y)->second;
67      }
68      row rows () const { return row(Rows.begin()); }
69      row rows_end () const { return row(Rows.end()); }
70      row find_row (const Point& x) const { return row(Rows.find(x)); }
71      const ProcSets& GetProcSets() const { return PS; }
72  };
73
74  class Vector : public valarray<double> {
75      const MatrixGraph& G;
76   public:
77      Vector (const MatrixGraph& g) : valarray<double>(g.size()), G(g) {}
78      double& operator () (const Point& x) { return operator [] (G(x)); }
79      double operator () (const Point& x) const { return operator [] (G(x)); }
80      MatrixGraph::row rows () const { return G.rows(); }
81      MatrixGraph::row rows_end () const { return G.rows_end(); }
82      MatrixGraph::row find_row (const Point& x) const { return G.find_row(x); }
83      Vector& operator = (double a) { for (int i=0; i<size(); ++i) (*this)[i]=a;}
84      const ProcSets& GetProcSets() const { return G.GetProcSets(); }
85      void ClearDirichlet () {
86          for (int i=0; i<size(); ++i)
87              if (G.Dirichlet(i)) (*this)[i] = 0;
88      }
89  };
90
91  inline double operator * (const Vector& u, const Vector& v) {
92      double s = 0;
93      for (int i=0; i<u.size(); ++i) s += v[i] * u[i];
94      return PPM->Sum(s);
```

```
95  }
96
97  inline double norm (const Vector& u) { return sqrt(u*u); }
98
99  inline Vector& operator << (Vector& u, double F (const Point&)) {
100     for (MatrixGraph::row r=u.rows(); r!=u.rows_end(); ++r) u(r()) = F(r());
101     return u;
102 }
103
104 void Collect (Vector& r) {
105     const ProcSets& PS = r.GetProcSets();
106     ExchangeBuffer E;
107     for (procset p = PS.procsets(); p!=PS.procsets_end(); ++p) {
108         int q = p[0];
109         if (q == PPM->proc()) continue;
110         if (r.find_row(p()) == r.rows_end()) continue;
111         E.Send(q) << p() << r(p());
112         r(p()) = 0;
113     }
114     E.CommunicateSizeBuffer();
115     for (int q=0; q<PPM->size(); ++q)
116         while (E.Receive(q).size() < E.Receive(q).Size()) {
117             Point x;
118             double a;
119             E.Receive(q) >> x >> a;
120             r(x) += a;
121         }
122 }
123
124 void DistributeDirichlet (MatrixGraph& G, Vector& u) {
125     ExchangeBuffer E;
126     const ProcSets& PS = u.GetProcSets();
127     for (procset p = PS.procsets(); p!=PS.procsets_end(); ++p) {
128         if (u.find_row(p()) == u.rows_end()) continue;
129         if (!G.Dirichlet(p())) continue;
130         for (int i=0; i<p.size(); ++i) {
131             int q = p[i];
132             if (q == PPM->proc()) continue;
133             E.Send(q) << p() << u(p());
134         }
135     }
136     E.CommunicateSizeBuffer();
137     for (int q=0; q<PPM->size(); ++q)
138         while (E.Receive(q).size() < E.Receive(q).Size()) {
139             Point P;
140             double a;
141             E.Receive(q) >> P >> a;
142             u(P) = a;
```

```
143                 G.SetDirichlet(P);
144             }
145 }
146 #endif
```

In der vorliegenden Version des parallelen Programms wurde die Klasse `MatrixGraph` um die Iteratorklasse **MatrixGraph::row** erweitert. Mit einem Objekt dieser können die Einträge im Hash `Rows` in der Klasse `MatrixGraph` durchlaufen werden. Für die Iteratorklasse `MatrixGraph::row` sind folgende Methoden und

- `MatrixGraph::row::row (const Iterator& r)`
  Konstruktor: Erzeugt einen neuen Iterator. Als Argument wird dem Konstruktor ein Iterator `r` eines `Row`-wertigen Hashes mit `Point`-wertigen Schlüsseln übergeben. Der von `r` bezeichnete Eintrag im Hash wird zum aktuellen Eintrag des neuen Iterators.

- `const Point& MatrixGraph::row::operator () ()`
  Gibt eine Referenz auf den Schlüssel des aktuellen Eintrags im Hash zurück.

Die Klasse **MatrixGraph** selbst wurde ebenfalls um eine weitere Funktionalität erweitert, durch die es möglich ist, den Randpunktes des Gitters Dirichlet-Bedingungen zuzuweisen. Dazu wurden folgende Methoden neu definiert oder verändert:

- `MatrixGraph::MatrixGraph (const Mesh& M, const ProcSets& ps)`
  Der Konstruktor wurde wie folgt erweitert: Nachdem die Vektoren das Hash `Rows` und die Vektoren `diag` und `column` angelegt wurden, wird im Vektor `dirichlet` für jede Zeile der Steifigkeitsmatrix ein `boolean`-Wert allokiert und mit `false` initialisiert (51,52). Für eine Zeilennummer `i` speichert der Eintrag `dirichlet[i]` dann die Information, ob es sich bei dem Gitterpunkt, dem die `i`-te Zeile der Steifigkeitsmatrix entspricht, um einen Punkt handelt, für den eine Dirichlet-Bedingung gilt.

- `int MatrixGraph::Size ()`
  Gibt die Anzahl der Nichtnullelemente der Steifigkeitsmatrix zurück.

- `bool MatrixGraph::Dirichlet (int i)`
  Gibt für eine Zeilennummer `i` der Wert `true` zurück, wenn es sich bei dem Gitterpunkt, dem die `i`-te Zeile in der Steifigkeitsmatrix entspricht, um einen Punkt mit einer Dirichlet-Bedingung handel.

- `bool MatrixGraph::Dirichlet (const Point& x)`
  Gibt für einen Gitterpunkt `x` den Wert `true` zurück, wenn für den Punkt eine Dirichlet-Bedingung gilt.

- `void MatrixGraph::SetDirichlet (const Point& x)`
  Mit dieser Methode wird dem `MatrixGraph`-Objekt mitgeteilt, dass für einen Gitterpunkt `x` eine Dirichlet-Bedingung gelten soll.

- `int MatrixGraph::operator () (const Point& x)`
  Gibt für einen Gitterpunkt `x` die zugehörige Zeilenummer in der Steifigkeitsmatrix zurück.

- `int MatrixGraph::operator () (const Point& x, const Point& y)`
  Gibt für zwei Gitterpunkte `x` und `y` die Spaltennummer desjenigen Eintrags in der Steifigkeitsmatrix zurück, der dem Punkte Paar (`x`,`y`) entspricht.

- `row MatrixGraph::rows ()`
  Gibt einen Iterator zurück, der auf den ersten Eintrag im Hash `Rows` zeigt.

- `row MatrixGraph::rows_end ()`
  Gibt einen Iterator zurück, der auf den letzen Eintrag im Hash `Rows` zeigt.

- `row MatrixGraph::find_row (const Point& x)`
  Gibt für einen Gitterpunkt `x` einen Iterator zurück, der auf den zugehörigen Eintrag im Hash `Rows` zeigt, d.h. der auf die zugehörige Zeile in der Steifigkeitsmatrix zeigt.

- `const ProcSets& MatrixGraph::GetProcSets()`
  Gibt eine Referenz auf das `ProcSet`-Objekt des Matrixgraphen zurück.

7.1.4 Vektoren und Matrizen (Teil 1)

Die Klasse **Vector** stellt eine Datenstruktur zur Verfügung, die zu einem gegebenen Matrixgraphen einen reellwertigen, m-dimensionalen Vektor erstellt, wobei m die Zeilendimension der Steifigkeitsmatrix sei, die vom Matrixgraphen repräsentiert wird. Diese Klasse wird dazu verwendet, den Lastvektor und den Lösungsvektor des entstehenden linearen Gleichungssystems zu repräsentieren. Für die Klasse `Vector` wurden die folgenden Methoden, Operatoren und Funktionen definiert:

- `Vector::Vector (const MatrixGraph& g)`
  Konstruktor: Erzeugt ein neues `Vector`-Objekt. Die Dimension des erzeugten Vektors gleicht der Anzahl der Zeilen im referenziell übergebenen `MatrixGraph`-Objekt `g`. Eine Referenz auf den Matrixgraphen wird in der privaten Variabel `G` abgespeichert.

- `double& Vector::operator () (const Point& x)`
  Gibt zu einem Gitterpunkt `x` eine Referenz auf den zugehörigen Wert im Lastvektor zurück. Diese Methode kann daher für Schreibzugriffe auf Vektorkomponenten verwendet werden.

- `double Vector::operator () (const Point& x)`
  Gibt zu einem Gitterpunkt `x` den zugehörigen Wert im Lastvektor zurück.

■ `MatrixGraph::row Vector::rows ()`
Gibt einen Iterator zurück, der auf die erste Zeile im Zeilen-Hash des Matrixgraphen `G` zeigt.

■ `MatrixGraph::row Vector::rows_end ()`
Gibt einen Iterator zurück, der auf die letzte Zeile im Zeilen-Hash des Matrixgraphen `G` zeigt.

■ `MatrixGraph::row Vector::find_row (const Point& x)`
Gibt einen Iterator zurück, der für einen Gitterpunkt `x` auf die zugehörige Zeile im Zeilen-Hash des Matrixgraphen `G` zeigt.

■ `Vector& Vector::operator = (double a)`
Zuweisungsoperator: Der Operator = weist allen Komponenten des Vektors den übergebenen `double`-Wert `a` zu.

■ `const ProcSets& Vector::GetProcSets()`
Gibt eine Referenz auf das `ProcSets`-Objekt des Matrixgraphen `G` zurück.

■ `void ClearDirichlet ()`
Die Methode `ClearDirichlet` weist jeder Komponente des Vektors den Wert Null zu, wenn für den entsprechenden Gitterpunkt eine Dirichlet-Bedingung definiert wurde.

□ `inline double operator * (const Vector& u, const Vector& v)`
Der Operator `*` implementiert das global Standard-Skalarprodukt zweier Vektoren `u` und `v`.

□ `inline double norm (const Vector& u)`
Gibt den Wert der Euklid-Norm eines Vektors `u` zurück.

□ `inline Vector& operator << (Vector& u, double F (const Point&))`
Zuweisungsoperator: Weist den Komponenten des Vektors `u` die Werte einer `double`-wertigen Funktion `F` an den zugehörigen Gitterpunkten zu.

□ `void Collect (Vector& r)`
Die Funktion `Collect` dient dazu, einen globalen Vektor `r` in seine *additiv eindeutige* Repräsentation zu überführen. Das heißt, dass für jeden Gitterpunkt die entsprechende Vektorkomponente in genau einem Prozess gespeichert wird, und zwar in dem dem Gitterpunkt zugeordneten Master-Prozess.
Dazu durchläuft jeder aufrufende Prozess die einzelnen Komponenten des seines lokalen Teilvektors. Ist der aufrufende Prozess nicht der Master-Prozess des zugehörigen Gitterpunktes, so schickt er den Punkt und den Wert der Komponente an den Master-Prozess und setzt die entsprechende Komponente auf den Wert Null (122–129). Anschließend liest jeder Prozess die empfangenen Punkte und Werte aus, und addiert diese Werte zu den entsprechenden, eigenen Vektorkomponenten hinzu (130–136).

□ `void DistributeDirichlet (MatrixGraph& G, Vector& u)`
Die Funktion `DistributeDirichlet` verteilt die Information über Dirichlet-Punkte unter den parallelen Prozessen. Jeder Prozess durchläuft das Prozessmengen-Hash seines lokalen MatrixGraphen `G`. Besitzt der aufrufende Prozess einen Gitterpunkt in seinem lokalen Gitter, für den eine Dirichlet-Bedingung definiert ist, so versendet er diesen Punkt und den Wert der zugehörigen Vektorkomponente des Vektors `u` an alle Prozesse, die zur Prozessmenge des Punktes zählen (142–150). Anschließend liest jeder Prozess die empfangenen Punkte und Wert aus. Er weist der ensprechenden Vektorkomponente den empfangenen Wert zu und definiert in seinem lokalen MatrixGraphen eine Dirichlet-Bedingung für den empfangenen Punkt (152–159). Auf diese Weise besitzen alle parallelen Prozesse nach dem Aufruf dieser Funktion diesselben Dirichlet-Punkte.

Die Klasse **Matrix** kapselt zu einem gegebenen Matrixgraphen die eigentliche Steifigkeitsmatrix, d.h. sie speichert die Werte der einzelnen Komponenten der Steifigkeitsmatrix. Dazu wurden folgende Methoden definiert:

- `class Matrix::Matrix (const MatrixGraph& g)`
  Konstruktor: Erzeugt ein neuen Objekt der Klasse `Matrix`. Dazu wird ein Array mit N Einträgen angelegt, wobei N der Anzahl der Nichtnullelemente im referenziell übergebenen Matrixgraphen `g` entspricht. Die Refrenz auf den Matrixgraphen wird in der privaten Variablen `G` gespeichert.

- `double& Matrix::operator () (const Point& x,const Point& y)`
  Gibt zu einem Punktepaar (`x`,`y`) des Gitters eine Referenz auf den zugehörigen Wert in der Steifigkeitsmatrix zurück. Diese Methode kann daher für Schreibzugriffe auf Matrixkomponenten verwendet werden.

- `double Matrix::operator () (const Point& x,const Point& y)`
  Gibt zu einem Punktepaar (`x`,`y`) des Gitters den zugehörigen Wert in der Steifigkeitsmatrix zurück.

- `int Matrix::Diag (int i)`
  Gibt zu einer Zeilennummer `i` die Position des Diagonalelements in der `i`-ten Zeile der Steifigkeitsmatrix im `column`-Vektor des Matrixgraphen `G` zurück.

7.1.5 Die graphische Ausgabe (Teil 2)

```
1  // file:      PFEM/09/Plot.h
2
3  #ifndef _PLOT_H_
4  #define _PLOT_H_
5
```

```
6  #include "Mesh.h"
7  #include "Algebra.h"
8
9  inline void Plot (const Mesh& M, const Vector& u) {
10     ExchangeBuffer E;
11     for (Mesh::cell c = M.cells(); c != M.cells_end(); ++c)
12         E.Send(0) << *c << u(c[0]) << u(c[1]) << u(c[2]);
13     E.CommunicateSizeBuffer();
14     if (PPM->master()) {
15         ofstream s("out");
16         Mesh::Cell C;
17         double u0,u1,u2;
18         for (int q=0; q<PPM->size(); ++q)
19             while (E.Receive(q).size() < E.Receive(q).Size()) {
20                 E.Receive(q) >> C >> u0 >> u1 >> u2;
21                 s << endl
22                   << C[0] << " " << u0 << endl
23                   << C[1] << " " << u1 << endl
24                   << C[2] << " " << u2 << endl
25                   << C[0] << " " << u0 << endl;
26             }
27         s << C[0] << " " << u0 << endl;
28     }
29 }
30 #endif
```

Die graphische Ausgabe wurde in der neunten Version des parallelen Programms erweitert. Es ist dadurch möglich, nicht nur das Gitter selbst, sondern auch die Lösung der Finiten-Element-Methode zu visualisieren. Dazu wurde die Funktion **Plot** wir folgt verändert:

- `inline void Plot (const Mesh& M, const Vector& u)`
  Die Funktion `Plot` erzeugt wie bisher eine Datei `out`, welche dem Programm *gnuplot* als Eingabedatei dient. Die resultierende graphische Ausgabe ist dabei ein dreidimensionales Netz, in dem jedem Gitterpunkt der entsprechende Wert der zugehörigen Vektorkomponente von `u` als Höhe zugeweisen wird.

7.1.6 Tensoren, Dirichlet-Werte und Residuenvektoren

```
1  // file:      PFEM/09/Main.C
2
3  #include "Mesh.h"
4  #include "Algebra.h"
5  #include "Plot.h"
```

```
6
7  const int levels = 0;
8
9  double f (const Point& x) { return x[0]*x[0]*x[1]; }
10
11 class Tensor {
12     double T[2][2];
13     double det;
14 public:
15     Tensor (const Mesh::cell& c) {
16         Point a = c[1]-c[0];
17         Point b = c[2]-c[0];
18         det = a[0]*b[1]-b[0]*a[1];
19         T[0][0] = b[1] / det;
20         T[0][1] = -a[1] / det;
21         T[1][0] = -b[0] / det;
22         T[1][1] = a[0] / det;
23     }
24     const double* operator [] (int i) const { return T[i]; }
25     double operator () () const { return det; }
26 };
27
28 Point operator * (const Tensor& T, const Point y) {
29     return Point(T[0][0]*y[0]+T[0][1]*y[1],T[1][0]*y[0]+T[1][1]*y[1]);
30 }
31
32 void Dirichlet (const Mesh& M, MatrixGraph& G, Vector& u) {
33     for (Mesh::cell c=M.cells(); c!=M.cells_end(); ++c) {
34         if (M.Boundary(0.5*(c[0]+c[1]))) {
35             G.SetDirichlet(c[0]);
36             G.SetDirichlet(c[1]);
37             u(c[0]) = f(c[0]);
38             u(c[1]) = f(c[1]);
39         }
40         if (M.Boundary(0.5*(c[1]+c[2]))) {
41             G.SetDirichlet(c[1]);
42             G.SetDirichlet(c[2]);
43             u(c[1]) = f(c[1]);
44             u(c[2]) = f(c[2]);
45         }
46         if (M.Boundary(0.5*(c[0]+c[2]))) {
47             G.SetDirichlet(c[0]);
48             G.SetDirichlet(c[2]);
49             u(c[0]) = f(c[0]);
50             u(c[2]) = f(c[2]);
51         }
52     }
53     DistributeDirichlet(G,u);
```

```
54 }
55
56 void Residual (const Mesh& M, const Vector& u, Vector& r){
57     const Point GradN[3] = { Point(-1,-1), Point(1,0), Point(0,1) };
58     r = 0.0;
59     for (Mesh::cell c=M.cells(); c!=M.cells_end(); ++c) {
60         Tensor T(c);
61         double area = 0.5 * T();
62         Point G0 = T * GradN[0];
63         Point G1 = T * GradN[1];
64         Point G2 = T * GradN[2];
65         Point G = u(c[0]) * G0 + u(c[1]) * G1 + u(c[2]) * G2;
66         r(c[0]) += area * G * G0;
67         r(c[1]) += area * G * G1;
68         r(c[2]) += area * G * G2;
69     }
70     r.ClearDirichlet();
71     Collect(r);
72 }
73
74 int main (int argv, char** argc) {
75     PPM = new ParallelProgrammingModel (&argv,&argc);
76     Mesh M;
77     ProcSets PS;
78     Distribute(M,PS);
79     int m = PPM->Sum(M.size());
80     mout << m << " coarse cells on " << PPM->size() << " processors" << endl;
81     for (int l=0; l<levels; ++l) M.Refine(PS);
82     m = PPM->Sum(M.size());
83     mout << m << " fine cells on " << PPM->size() << " processors" << endl;
84     MatrixGraph G(M,PS);
85     pout << G.size() << " unknowns on proc " << PPM->proc() << endl;
86     pout << G.Size() << " matrix entries on proc " << PPM->proc() << endl;
87     Vector u(G);
88     Dirichlet(M,G,u);
89     u << f;
90     Vector r(G);
91     Residual(M,u,r);
92     pout << norm(r) << " residual norm " << endl;
93     Plot(M,u);
94     delete PPM;
95     return 0;
96 }
```

In der Datei `Main.C` wurden neben dem Hauptprogramm noch eine Reihe weiterer Klassen und Funktionen definiert, welche es ermöglichen, konkrete Probleme mit dem Finite-Element-Gitter zu lösen.

Die Klasse **Tensor** ordnet dabei jedem Gitterdreick einen Tensor T zu, der die Vektoren

$$\begin{pmatrix} -1 \\ -1 \end{pmatrix}, \quad \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

auf die gewichteten, inneren Normalenvektoren der Dreieckskanten transformiert. Für die Klasse `Tensor` wurden dazu folgende Methoden und Operatoren definiert:

- `Tensor::Tensor (const Mesh::cell& c)`
  Konstruktor: Erzeugt zu einem Dreieck `c` einen neuen Tensor, der die oben genannten Transformationseigenschaften besitzt.
- `const double* Tensor::operator [] (int i)`
  Gibt zu einem `int`-Wert `i` von `0` oder `1` einen Zeiger auf das Array zurück, das die Komponenten der `i`-ten Zeile des Tensors speichert.
- `double Tensor::operator () ()`
  Gibt den doppelten Flächeninhalt des zu dem Tensor gehörenden Gitterdreiecks zurück.
- `Point operator * (const Tensor& T, const Point y)`
  Der Operator $*$ implementiert das Tensorprodukt eines Tensors `T`, der auf einen Punkt `y` angewendet wird.

Neben der Klasse `Tensor` werden in der Datei `Main.C` zwei globale Funktionen definiert. Im einzelnen sind das:

- `void Dirichlet (const Mesh& M, MatrixGraph& G, Vector& u)`
  Die Funktion `Dirichlet` durchläuft nacheinander alle Dreiecke eines Gitter `M` und weist denjenigen Eckpunkten eines Dreiecks eine Dirichlet-Bedingung zu, die an einer Randkante des Dreiecks liegen. Die auf diese Weise bestimmten Dirichlet-Punkte werden dem Matrixgraphen `G` übergeben. In den zugehörigen Komponenten des Vektors `u` werden die Funktionswerte der global definierten Funktion `f` an den Dirichlet-Punkten zugewiesen (35–54). Anschließend kommuniziert jeder parallele Prozess seine Dirichlet-Punkte allen anderen parallelen Prozessen (55).
- `void Residual (const Mesh& M, const Vector& u, Vector& r)`
  Die Funktion berechnet für ein Gitter `M` und für eine approximierten Lösung `u` auf diesem Gitter den Residuenvektor `r`. Dazu werden für jedes Dreieck die gewichteten, inneren Normalenvektoren der Dreieckskanten berechnet (62–66). Anschließend werden für jeden Dreieckspunkt entsprechende „Flüsse“ in Richtung der Kantennormalen in den entsprechenden Vektorkomponenten des Residuenvektors aufaddiert (67–70). Da auf den Dirichlet-Punkten keine Fehler entstehen, werden die entsprechenden Komponenten im Residuenvektor auf Null gesetzt (72). Der Residuenvektor wird danach in seine additiv eindeutige Darstellung gebracht (73).

8 Parallele Lineare Algebra

Eine Finite-Element-Funktion $u \in V_{\mathcal{C}}$ wird durch die Einschränkung $u\colon \mathcal{V} \to \mathbf{R}$ (als Knotenfunktion) eindeutig bestimmt, und im Parallelen durch **konsistente Vektoren**

$$(\underline{u}_p) \in V[\pi] := \Big\{(\underline{v}_p) \in \prod_{p\in\mathcal{P}} \mathbf{R}^{N_p} \colon \underline{v}_p(g_p(x)) = \underline{v}_q(g_q(x)) \text{ für alle } p,q \in \pi(x)\Big\}$$

repräsentiert.

Das Residuum $\lambda \in V_{\mathcal{C}}'$ mit $\lambda\colon V_{\mathcal{C}}(0) \to \mathbf{R}$ wird durch $(\lambda(\varphi_x))_{x\in\mathcal{V}\setminus\Gamma}$ eindeutig bestimmt und durch **additive Vektoren** (r_p) repräsentiert. Durch eine Interface-Kommunikation wird es in einen **additiv-eindeutigen Vektor**

$$(r_p) \in V(\pi) := \Big\{(s_p) \in \prod_{p\in\mathcal{P}} \mathbf{R}^{N_p} :\ r_p(x) = 0 \text{ für } p \neq \mu(x)\Big\} \qquad \big(\mu(x) := \min\ \pi(x)\big)$$

umgeformt (Collect).

Die additive Steifigkeitsmatrix $(\underline{A}_p)$ definiert einen Operator

$$\underline{A}: \quad V[\pi] \longrightarrow V(\pi)$$

durch $\underline{r}_p^{\text{add}} = \underline{A}_p\underline{u}_p$ (ohne Kommunikation)

und $(\underline{r}_p) =$ Collect $(\underline{r}_p^{\text{add}})$.

Ein Vorkonditionierer ist ein Operator

$$\underline{B}: \quad V(\pi) \longrightarrow V[\pi].$$

Beispiel

1) Jacobi-Vorkonditionierer $c = \operatorname{diag}(A)^{-1}r$
   definiert $\underline{d}_p(g_p(x)) := \sum\limits_{q\in\pi(x)} \underline{A}_q(\operatorname{diag}(g_q(x))$

$$\underline{c}^{\text{add}}(i) = \underline{d}_p(i)^{-1}\underline{r}_p(i) \qquad i = 0,\ldots,N_p-1$$
$$\underline{c}_p(g_p(x)) = \sum_{q\in\pi(x)} c_q^{\text{add}}(g_q(x)).$$

   Dabei wird zweimal die Routine **Consistent** $V(\pi) \longrightarrow V[\pi]$ aufgerufen:
   $B(p,q) = \emptyset$ auf Prozess p:
   für alle $x \in \mathcal{V}_p$ mit $|\pi_p(x)| > 1$:

$$B(p,q) := B(p,q) \cap \big\{(x, c_p^{\text{add}}(g_p(x))\big\}$$
$$\text{für alle } q \in \pi_p(x) \setminus \{p\}.$$

Austausch von $B(p,q)$

auf Prozess q: für alle p:

$$(x,\gamma) \in B(p,q) \,:\, \underline{c}_q(g_q(x)) = \underline{c}_q(g_q(x)) + \gamma$$

bis auf Rundungsfehler hat dann $(\underline{c}_q)$ konsistente Werte.

Alternative: Einsammeln auf $\mu(x)$, dann Verschicken an alle.

2) Block-Jacobi-Vorkonditionierer mit lokalem Gauß-Seidel Verfahren $c = \text{lower}\,(A)\,r$

$$\begin{aligned}
&\underline{c}_p^{\text{add}} = 0 \\
&\text{für } i = 0, \ldots, N_p - 1 \\
&\qquad \rho = \underline{r}_p(i) \\
&\qquad \text{für } e = \text{diag}(i) + 1, \ldots, \text{diag}(i+1) - 1 \\
&\qquad\qquad \rho = \rho - \underline{A}_p(e)\, \underline{c}_p(\text{col}_p(e)) \\
&\qquad \underline{c}_p^{\text{add}}(i) = \underline{d}_p(i)^{-1} \rho
\end{aligned}$$

Lineare Löser

Betrachte

$$\underline{A}\,\underline{u} = \underline{b}$$

in $\mathbf{R}^N$. Sei $\underline{u}^0$ ein Startvektor und

$$\underline{r}^0 = \underline{b} - \underline{A}\,\underline{u}^0 \qquad (\Longrightarrow\ \underline{r}^0 = \underline{A}(\underline{u} - \underline{u}^0)\,)$$

das Residuum, dann gilt:

$$\underline{A}\,\underline{c} = \underline{r}^0 \iff \underline{A}\,(\underline{u}^0 + \underline{c}) = \underline{b}$$

$\underline{c}$ heißt Korrektur.

Sei $\underline{B}$ ein Vorkonditionierer mit $\underline{B} \cdot \underline{A} \approx \text{id}$.

Berechne die (approximative) Korrektur

$$\underline{c}^0 = \underline{B}\,\underline{r}^0$$

und den Update $\underline{u}' = \underline{u}^0 + \underline{c}^0$.

Dann gilt:

$$\begin{aligned}
&\underline{r}' = \underline{b} - \underline{A}\,\underline{u}' = \underline{b} - \underline{A}\,\underline{u}^0 - \underline{A}\,\underline{c}^0 = \underline{r}^0 - \underline{A}\,\underline{c}^0 \\
&\underline{u}' - \underline{u} = \underline{u}^0 + \underline{u} + \underline{c} = \underline{u}^0 - \underline{u} + \underline{B}\,\underline{r}^0 = (\text{id} - \underline{B}\,\underline{A})(\underline{u}^0 - \underline{u})
\end{aligned}$$

Iterativer linearer Löser

gegeben $\underline{u}^0$, $\underline{A}$, $\underline{r}^0$

berechne $\varepsilon = \max\{$ eps , red $||\underline{r}^0||\}$

mit eps absolute Fehlerschranke

$k = 0$ red Reduktionsfaktor

solange	$\|\|r^k\\| > \varepsilon$	solange	$\|\|r\\| \geq \varepsilon$
	$\underline{c}^k = \underline{B}\,\underline{r}^k$		$\underline{c} = \underline{B}\,\underline{r}$
	$\underline{u}^{k+1} = \underline{u}^k + \underline{c}^k$		$\underline{u} := \underline{u} + \underline{c}$
	$\underline{r}^{k+1} = \underline{r}^k - \underline{A}\,\underline{c}^k$		$\underline{r} := \underline{r} - \underline{A}\,\underline{c}$

Dann gilt:

$$\underline{u}^k - \underline{u} = (\mathrm{id} - \underline{B}\,\underline{A})^k\,(\underline{u}^0 - \underline{u})$$

Theorem 4 *a) Wenn $\rho(\mathrm{id} - \underline{B}\,\underline{A}) < 1$ gilt, dann konvergiert $\underline{u}^k$ gegen $\underline{u}$.*

b) Wenn $\underline{A}$ symmetrisch positiv definit, dann gilt $\rho(\mathrm{id} - \underline{B}\,\underline{A}) < 1$

- *für den gedämpften Jacobi-Vorkonditionierer $\underline{B} = \Theta\,\mathrm{diag}(\underline{A})^{-1}$ mit $\Theta > 0$ genügend klein*
- *für den Gauß-Seidel-Vorkonditionierer $\underline{B} = (\mathrm{lower}\underline{A})^{-1}$*

c) Wenn $\underline{A}\colon V[\pi] \longrightarrow V(\pi)$

$\underline{B}\colon V(\pi) \longrightarrow V[\pi]$

$\underline{r}^0 \in V(\pi)$, $\underline{u}^0 \in V[\pi]$

dann gilt: $\underline{r}^k \in V(\pi)$, $\underline{u}^{(k)} \in V[\pi]$,

und für den (gedämpften) Jacobi-Vorkonditionierer ist die Iteration für alle P und jede Lastverteilung gleich!

Bemerkung Das Gauß-Seidel-Verfahren ist nicht ohne zusätzliche Parallelisierung durchführbar.

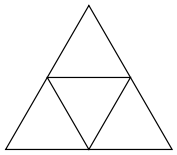
- es erfordert eine globale Nummerierung
- im einzelnen Schritt werden die schon aktualisierten Werte benötigt.

Das Block-Jacobi-Verfahren mit einem Gauß-Seidel-Schritt pro Prozess führt (je nach Lastverteilung und Nummerierung) zu unterschiedlichen Iterationen.

Kaskadisches Mehrgitter

Seien $\mathcal{C}, \ldots, \mathcal{C}^K$ geschachtelte Netze, die durch uniforme Verfeinerung ohne Umverteilung entstehen, d. h. für alle $C = (x_0,\, x_1,\, x_2) \in \mathcal{C}_p^k$ auf Prozess $p \in \mathcal{P}$ ex. $C_1,\, C_2,\, C_3,\, C_4$ mit

$$\bigcup_{j=1}^{4} C = C \cup \{x_{01},\, x_{12},\, x_{02}\} \quad \frac{1}{2}(x_i + x_j) = x_{ij}$$



Setze $V^k := V_{\mathcal{C}^k} = \{v \in C^0(\Omega)\colon v|_{\Omega_C}$ linear $C \in \mathcal{C}^k\}$.

Dann gilt: $V^{k-1} \subset V^k$

Zu $u \in V^{k-1}$ definiert $\underline{u}_p^{k-1} \in \mathbf{R}^{N_p^{k-1}}\colon \underline{u}_p^{k-1}(g_p^{k-1}(x)) = u(x)\ x \in \mathcal{V}_p^{k-1}$

$$\underline{u}_p^k \in \mathbf{R}^{N_p^k}\ \underline{u}_p^k(g_p^k(x)) = u(x)\ x \in \mathcal{V}_p^k$$

$$\Longrightarrow \qquad \underline{u}_p^k(g_p^k(x)) = \underline{u}^{k-1}(g_p^{k-1}(x)) \quad x \in \mathcal{V}_p^{k-1}$$

$$\underline{u}_p^k(g_p^k(x)) = \tfrac{1}{2}(\underline{u}^{k-1}(g_p^{k-1}(x_0)) + \underline{u}^{k-1}(g_p^{k-1}(x_1)))$$

$$x = \tfrac{1}{2}(x_0 + x_1) \in \mathcal{V}_p^k \setminus \mathcal{V}_p^{k-1},\ x_0,\, x_2 \in \mathcal{V}_p^{k-1}.$$

Es gilt: Wenn $(\underline{u}_p^{k-1}) \in V^{k-1}[\pi]$ konsistent ist, dann ist auch $(\underline{u}_p^k) \in V^k[\pi]$ konsistent.

Interpolate $(\underline{u}^{k-1},\, \underline{u}^k) \qquad \underline{u}^k = \underline{I}^k\, \underline{u}^{k-1}$

für $C = (x_0,\, x_1,\, x_2) \in \mathcal{C}^{k-1}$

$$\underline{u}_p^k(g_p^k(x_i)) = \underline{u}_p^{k-1}(g_p^{k-1}(x_i))$$

$$\underline{u}_p^k(g_p^k(\tfrac{1}{2}\, x_i + \tfrac{1}{2}\, x_j)) = \tfrac{1}{2}\, \underline{u}_p^{k-1}(g_p^{k-1}(x_i)) + \tfrac{1}{2}\, \underline{u}_p^{k-1}(g_p^{k-1}(x_j))$$

Idee

Sei

$$-\Delta\, u = 0 \qquad \text{in } \Omega$$

$$u = g \qquad \text{auf } \Gamma$$

$$u^j \in V^j \qquad \text{Galerkin-Lösung}$$

$$|\!|\!| v |\!|\!| := \qquad \left(\int_\Omega |\nabla\, v|^2\, dx\right)^{\frac{1}{2}} \text{ Energie-Norm}$$

$\Longrightarrow$ $u - u^j$ konvergiert mit $\|\!|u - u^j|\!\| \leq C\, 2^{-j}$

$$
\begin{array}{lll}
\Longrightarrow & \|\!|u - u^j|\!\| & \leq \quad \tfrac{1}{2}\|\!|u - u^{j-1}|\!\| \leq \cdots \leq 2^{-j}\|\!|u - u^0|\!\| \\
\Longrightarrow & \|\!|u^j - u^{j-1}|\!\| & \leq \quad \|\!|u^j - u|\!\| + \|\!|u^{j-1} - u|\!\| \\
 & & \leq \quad \underbrace{(2^j + 2^{j-1})}_{3\cdot 2^{j-1}}\, \|\!|u - u^0|\!\|
\end{array}
$$

Also: u^{j-1} wird immer bessere Startlösung für u^j!

Aber: Gauß-Seidel konvergiert immer langsamer für $k \to \infty$

Routinen:

Dirichlet $(\underline{u})$:	setze Randwerte
	paralleler Austausch von Randwerten
Residual $(\underline{u},\, \underline{r})$	berechne abhängig von der konsistenten
	aktuellen Lösung $\underline{u}$ das additive Residuum $\underline{r} = (\underline{r}_p)$
	paralleles Einsammeln ergibt additiv eindeutiges Residuum
	Löschen der Dirichleteinträge
Jacobi $(\underline{u},\, \underline{A})$	Berechnung der additiven Jacobimatrix $\underline{A} = (\underline{A}_p)$
	Ersetzen der Dirichletzeilen durch Einheitsvektoren
PC_Step $(\underline{u},\, \underline{r},\, \underline{A},\, \underline{B})$	Vorkonditionierungsschritt

$$\underline{c} = \underline{B}\,\underline{r}$$
$$\underline{u} := \underline{u} + \underline{c}$$
$$\underline{r} := \underline{r} - \underline{A}\,\underline{c}$$

Interpolate $(\underline{u}^{j-1},\, \underline{u}^j)$	Interpoliere Werte auf $\mathcal{C}^j$
	auf das nächste Netz $\mathcal{C}^{j+1}$

Algorithmus:	wähle Startwert $\underline{u}^0$, $\varepsilon > 0$, $\quad m > 0$
$j = 0$	Dirichlet $(\underline{u}^0)$
	Residual $(\underline{u}^0,\, \underline{r}^0)$
	Jacobi $(\underline{u}^0,\, \underline{A}^0)$
	solange $\|\|\underline{r}^0\|\| > \varepsilon$: PC_Step $(\underline{u}^0,\, \underline{r}^0,\, \underline{A}^0,\, \underline{B}^0)$

für $j = 1, \ldots, J$ Refine $(\mathcal{C}^{j-1}, \mathcal{C}^j)$

Interpolate $(\underline{u}^{j-1}, \underline{u}^j)$

Dirichlet $(\underline{u}^j)$

Residual $(\underline{u}^j, \underline{r}^j)$

Jacobi $(\underline{u}^j, \underline{A}^j)$

für $i = 1, m$ PC_Step $(\underline{u}^j, \underline{r}^j, \underline{A}^j, \underline{B}^j)$

Aufwand $\mathcal{O}(N^0 \cdot \cdots + N' \cdot m + \cdots + N^J \cdot m) = \mathcal{O}(m\, N^3)$!

Beobachtung:

Betrachte $\underline{A} \simeq$ Laplace-Diskretisierung auf dem Einheitsquadrat

$\underline{B}\,\underline{c} = \theta \underline{c}, \quad \theta < \frac{1}{\lambda_{\max}(\underline{A})}$

$\Longrightarrow \quad \underline{r}^k = (\mathrm{id} - \theta \underline{A})^k\, \underline{r}^0$

konvergent

dann gilt für die Eigenvektoren $\underline{w}_i$ von $\underline{A}$ mit $\underline{A}\,\underline{w}_i = \lambda_i\, \underline{w}_i$

a) $\underline{w}_i^T\, \underline{r}^k$ konvergiert schnell gegen 0 für große λ_i

b) $\underline{w}_i^T\, \underline{r}^k$ konvergiert langsam gegen 0 für kleine λ_i

große λ_i entsprechen hohen Frequenzen mit kurzwelligen $\underline{w}_i$

kleine λ_i entsprechen niedrigen Frequenzen mit langwelligen $\underline{w}_i$

Idee Reduziere die langwelligen Fehleranteile auf dem gröberen Netz!

Grobgitterkorrektur

Zur Näherung $u^j \in V_{\mathcal{C}^j}$ berechne

Residuum $\lambda^{j-1} \in \hat{V}_{\mathcal{C}^{j-1}}$ mit $\lambda^{j-1}(v) = \int\limits_\Omega \nabla u^j \cdot \nabla v\, dx$ für $v \in V_{\mathcal{C}^{j-1}}(0)$

Korrektur $c^{j-1} \in V_{\mathcal{C}^{j-1}}(0)$ mit $\int\limits_\Omega \nabla c^{j-1} \nabla v\, dx = \lambda^{j-1}(v) \qquad v \in V_{\mathcal{C}^{j-1}}(0)$

Update $u^j + c^{j-1} \in V_{\mathcal{C}^j}$

entspricht: zu $\underline{u}^j \in V^j[\pi]$ berechne

$\underline{r}^{j-1} \in V^{j-1}(\pi)$ mit $(\underline{r}^{j-1})\underline{v} = (\underline{r}^j)^T(\underline{I}^j\, \underline{v}) = \underline{b}^j - \underline{A}^j\, \underline{u}^j$

löse $\underline{c}^{j-1} \in V^{j-1}[\pi]$: $\underline{A}^{j-1}\, \underline{c}^{j-1} = \underline{r}^{j-1}$

Update $\underline{u}^j := \underline{u}^j + \underline{I}^j\, \underline{c}^{j-1}$

Restriktion

$$(\underline{I}^j)^T \colon V^j(\pi) \longrightarrow V^{j-1}(\pi)$$

$$\text{mit } (\underline{r}^j)^T(\underline{I}^j\,\underline{v}^{j-1}) = \left(\underbrace{(\underline{I}^j)^T\underline{r}^j}_{=\underline{r}^{j-1}\in V^{j-1}(\pi)}\right)^T \underline{v}^{j-1}$$

Restrict $\left((\underline{r}_p^j),(\underline{r}_p^{j-1})\right)$

$\underline{r}_p^{j-1} = 0$ für alle $x \in \mathcal{V}_p^{j-1}$: $\underline{r}_p^{j-1}(g_p^{j-1}(x)) \mathrel{+}= \underline{r}_p^j(g_p^j(x))$

$\underline{\text{tmp}}$ = $\underline{r}_p^j$ für alle $C = (x_0,\, x_1,\, x_2) \in \mathcal{C}_p$

$x_{ik} = \frac{1}{2}(x_i + x_k)$
$\underline{r}_p^{j-1}\left(g_p^{j-1}(x_i)\right) \mathrel{+}= \frac{1}{2}\ \underline{\text{tmp}}\ \ (g^j(x_{ik}))$
$\underline{r}_p^{j-1}\left(g_p^{j-1}(x_k)\right) \mathrel{+}= \frac{1}{2}\ \underline{\text{tmp}}\ \ (g^j(x_{ik}))$
$\underline{\text{tmp}}\ \ (g^j(x_{ik})) = 0$

für alle $x \in \mathcal{D}_p^{j-1}$: $\underline{r}_p^{j-1}\left(g_p^{j-1}(x)\right) = 0$

Collect $(\underline{r})$

Zweigitterverfahren

Gegeben: Startvektor $\underline{u}^j$, aktuelles Residuum $\underline{r}^j$

	$\underline{c}^j = \underline{B}^j\,\underline{r}^j$
Glätten	$\underline{u}^j \mathrel{+}= \underline{c}^j$
	$\underline{r}^j \mathrel{-}= \underline{A}^j\,\underline{c}^j$
	Restrict $(\underline{r}^j,\, \underline{r}^{j-1})$
Grobgitterkorrektor	$\underline{c}^{j-1} = 0$, Solve $(\underline{c}^{j-1},\, \underline{A}^{j-1},\, \underline{r}^{j-1})$
	Interpolate $(\underline{c}^{j-1},\, \underline{c}^j)$
	$\underline{u}^j \mathrel{+}= \underline{c}^j$
	$\underline{r}^j \mathrel{-}= \underline{A}^j\,\underline{c}^j$

Mehrgitterverfahren

lineares Iterationsverfahren

$\underline{u}^J := \underline{u}^J + \underline{C}^J(\underline{b}^J - \underline{A}^J\,\underline{u}^J)$,

wobei der Mehrgittervorkonditionierer $\underline{c}^j = \underline{C}^j\,\underline{r}^j$ rekursiv definiert ist: $\underline{C}^0 = (\underline{A}^0)^{-1}$ und für $j > 0$:

$\underline{c}^j = 0$

für $i = 1, \ldots, m$ $\quad \underline{w}^j = \underline{B}^j\,\underline{r}^j$

$\underline{c}^j \mathrel{+}= \underline{w}^j$

$\underline{r}^j \mathrel{-}= \underline{A}^j\,W^j$

$\underline{r}^{j-1} = (\underline{T}^j)^T\,\underline{r}^j$

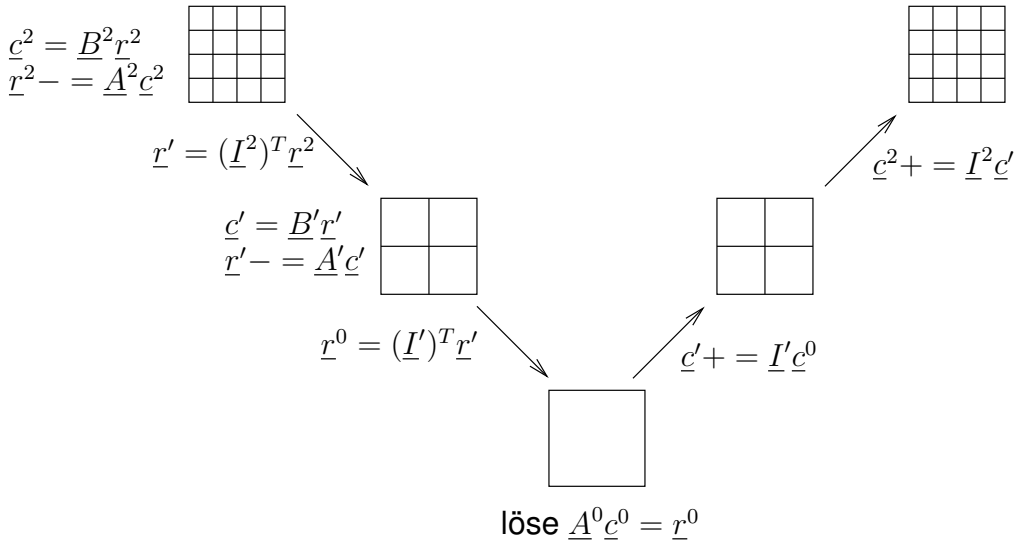
$\underline{c}^{j-1} = \underline{C}^{j-1}\,\underline{r}^{j-1}$

$\underline{w}^j = \underline{I}^j\,\underline{c}^{j-1}$

$\underline{c}^j \mathrel{+}= \underline{w}^j$

$\underline{r}^j \mathrel{-}= \underline{A}^j\,\underline{w}^j$

Beispiel $J = 2,\ m = 1$



cg–Verfahren für symmetrische Vorkonditionierer $\underline{C}$

Es gilt $\quad \underline{u}^{k+1} = \underline{u}^k + \underline{C}\,\underline{r}^k$

$\Longrightarrow \quad \underline{u}^1 = \underline{u}^0 + \underline{C}\,\underline{r}^0$

$\underline{u}^2 = \underline{u}^1 + \underline{C}\,\underline{r}^1 = \underline{u}^1 + \underline{C}(\underline{r}^0 - \underline{A}(\underline{u}^1 - \underline{u}^0))$

$= \underline{u}^0 + 2\underline{C}\,\underline{r}^0 - \underline{C}\,\underline{A}\,\underline{C}\,\underline{r}^0$ u. s. w.

$\Longrightarrow\ \underline{u}^{k+1} \in \underline{u}^0 + \underline{C}\,K^k$ mit

$K^k =$ span $\{(\underline{A}\,\underline{C})^i\,\underline{r}^0 \colon i = 0, \ldots, k\}$ Krylovraum

Das cg-Verfahren bestimmt $\underline{u}^{k+1} \in \underline{u}^0 + \underline{C}\,\underline{K}^k$, so dass der Fehler in einer geeigneten Norm minimal wird:

Gegeben	$\underline{u}$ und $\underline{r}$,
setze	$\underline{p} = 0,\ \rho_0 = 0,\ \varepsilon =$ red $\|\|\underline{r}\|\| +$ eps
solange	$\|\|r\|\| > \varepsilon$
	$\underline{c} = \underline{C}\,\underline{r}$
	$\rho_1 = \underline{c}^T\,\underline{r}$
	$\underline{p} = \frac{\rho_1}{\rho_0}\,\underline{p} + \underline{c}$
	$\rho_0 = \rho_1$
	$\underline{w} = \underline{A}\,\underline{p}$
	$\alpha = \frac{\rho_0}{\underline{c}^T\,\underline{w}}$
	$\underline{u} \mathrel{+}= \alpha\,\underline{p}$
	$\underline{r} \mathrel{-}= \alpha\,\underline{w}$

Satz es gilt für $\underline{u} = \underline{A}^{-1}\,\underline{b}$

$$\|\underline{u}^k - \underline{u}\| \leq Z \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^k \|\underline{u}^0 - \underline{u}\|, \quad \kappa = \text{cond}\,(\underline{C}\,\underline{A})$$

Rückblick

1) Basis der Vorlesung ist ein Modell zur Repräsentation verteilter Daten, die an Koordinaten in $\mathbf{R}^d$ gebunden sind.

2) Aufbauend auf dieses Modell wurde eine klassische Finite-Elemente-Diskretisierung für ein Modellproblem formuliert, d. h., der übliche sequentielle Algorithmus wurde auf verteilten Daten realisiert.

3) Der gesamte Simulationsprozess besteht aus

   - Preprocessing (Gitteraufbau . . .)
   - Berechnung (Assemblieren, Gleichungslösen)
   - Postprocessing (Visualisieren)

   Wir haben uns auf die Berechnung konzentriert.

4) Eine C++ Implementation als Prototyp wurde entwickelt:

   - Lernen von C++
   - Überprüfen des Modells: möglichst 1-1 Umsetzung

- kein Produktionscode!

Grundsätze des Software-Engineering

- Formuliere ein Ziel
- Spezifiziere Schnittstellen und Modularisiere
- Formuliere Algorithmen
- Formuliere Tests
- dann erst: schreibe Prototypen
- Redesign

$\Longrightarrow$ Im Vordergrund steht immer das mathematische Problem!

Ausblick

1) Parallelisierungsverluste sind unvermeidlich, daher lohnt es sich nur für große, dreidimensionale, zeitabhängige, nichtlineare Systeme.
2) Parallelisiere immer die beste Methode, auch wenn die parallele Effizienz nicht optimal ist: Das ist immer besser als eine parallel effiziente Realisierung einer schlechteren Methode!
3) Erweiterungen
   - Systeme, 3D, andere Diskretisierungen (quadratisch)
   - Verfeinern und Mehrgitter auf wachsenden Prozessorzahlen
   - (Dynamisches) Umverteilen bei lokaler Verfeinerung
   - Parallele direkte Löser/Gebietszerlegungsverfahren für zu grobe Grobgitter.

8.1 Paralleles Programm 10

8.1.1 Beschreibung

8.1.2 Dateien

`PFEM/10/Parallel.h`
`PFEM/10/Parallel.C`
`PFEM/10/Point.h`
`PFEM/10/Mesh.h`

```
PFEM/10/Algebra.h
PFEM/10/Plot.h
PFEM/10/Main.C
PFEM/10/Makefile
PFEM/10/mesh
PFEM/10/out.gnu
```

8.1.3 Vektoren und Matrizen (Teil 2)

```
1  // file:      PFEM/10/Algebra.h
2
3  #ifndef _ALGEBRA_H_
4  #define _ALGEBRA_H_
5
6  #include "Mesh.h"
7
8  #include <valarray>
9
10 class MatrixGraph { ... };
11
12 template <class A, class B> class constAB {
13     const A& a;
14     const B& b;
15  public:
16     constAB (const A& _a, const B& _b) : a(_a), b(_b) {}
17     const A& first () const { return a; }
18     const B& second () const { return b; }
19 };
20
21 template <class A, class B>
22 constAB<A,B> operator * (const A& a, const B& b) { return constAB<A,B>(a,b); }
23
24 class Matrix;
25
26 class Vector : public valarray<double> { ... };
27
28 inline double operator * (const Vector& u, const Vector& v) { ... }
29
30 inline double norm (const Vector& u) { ... }
31
32 inline Vector& operator << (Vector& u, double F (const Point&)) { ... }
33
34 class Matrix : public valarray<double> {
35     const MatrixGraph& G;
36  public:
```

```
37      Matrix (const MatrixGraph& g) : valarray<double>(g.Size()), G(g) {}
38      double& operator () (const Point& x,const Point& y) { ... }
39      double operator () (const Point& x,const Point& y) const { ... }
40      int Diag (int i) const { ... }
41      Matrix& operator = (double a) { for (int i=0; i<size(); ++i) (*this)[i]=a;}
42      void ClearDirichlet () {
43          for (int i=0; i<G.size(); ++i)
44              if (G.Dirichlet(i)) {
45                  int e = Diag(i);
46                  operator [] (e) = 1;
47                  for (++e; e<Diag(i+1); ++e) operator [] (e) = 0;
48              }
49      }
50      void MultiplyMinus (const Vector& u, Vector& r) const {
51          for (int i=0; i<u.size(); ++i) {
52              int e = G.Diag(i);
53              r[i] -= operator [] (e) * u[i];
54              for (++e; e < G.Diag(i+1); ++e) {
55                  int j = G.Column(e);
56                  r[i] -= operator [] (e) * u[j];
57              }
58          }
59      }
60  };
61
62  void Collect (Vector& r) { ... }
63
64  void DistributeDirichlet (MatrixGraph& G, Vector& u) { ... }
65
66  Vector& Vector::operator -= (const constAB<Matrix,Vector>& MV) {
67      MV.first().MultiplyMinus(MV.second(),*this);
68      Collect(*this);
69      return *this;
70  }
71
72  #endif
```

In der zehnten Version des parallelen Programms wurde die Klasse **Matrix** um die folgenden Methoden und Operatoren erweitert:

- `Matrix& Matrix::operator = (double a)`
  Zuweisungsoperator: Der Operator = weist jeder Nichtnullkomponente der Steifigkeitsmatrix den Wert `a` zu.
- `void Matrix::ClearDirichlet ()`
  Die Methode bewirkt, dass die Steifigkeitsmatrix bei einer Multiplikation mit einem Vektor auf alle Vektorkomponenten, die Dirichlet-Punkten im Gitter entsprechen,

wie die Identität wirkt. Dazu wird zu jedem Dirichlet-Punkt des Gitters die entsprechende Diagonalkomponente der Steifigkeitsmatrix auf den Wert 1 gesetzt (46–48). Alle übrigen Einträge in derselben Zeile werden auf den Wert 0 gesetzt (49).

- `void Matrix::MultiplyMinus (const Vector& u, Vector& r)`
  Ist `A` ein `Matrix`-Objekt, und sind `u` und `r` zwei Vektoren, so führt der Aufruf von `A.MultiplyMinus(u,r)` zu der Zuweisung `r = r - Au`.

In dieser und in den folgenden Versionen des parallelen Programms müssen häufig Matrix-Vektor-Operationen der Form $r = r - Au$ vorgenommen werden. Um eine intuitive Schreibweise der Befehle (nämlich `r -= A * u`) zu ermöglichen, und *ein Speichern von Zwischenergebnissen* (nämlich `temp = A * u`) zu verhindern, wurde eine spezielle Programmiertechnik eingesetzt, die auf Templates beruht. Sie ermöglicht es, dass *alle Rechnungen erst bei der Zuweisung* erfolgen. Die Template-Klasse **constAB** spielt hierbei eine entscheidende Rolle. Für sie wurden folgende Methoden definiert:

- `template<class A,class B> constAB::constAB (const A& _a,const B& _b)`
  Konstruktor: Erzeugt ein neues Objekt der Klasse `constAB`. Die einzige Aufgabe dieses Objekts besteht darin, eine Referenz auf das übergebene Objekt `_a` der Klasse `A` sowie eine Referenz auf das Objekt `_b` der Klasse `B` zu speichern.

- `template<class A,class B> const A& constAB::first ()`
  Gibt eine Referenz auf das erste gespeicherte Objekt der Klasse `A` zurück.

- `template<class A,class B> const A& constAB::second ()`
  Gibt eine Referenz auf das zweite gespeicherte Objekt der Klasse `B` zurück.

- `template<class A,class B>`
  `constAB<A,B> operator * (const A& a, const B& b)`
  Gibt ein Objekt der Klasse `constAB<A,B>` zurück, das Refrenzen auf `a` und `b` speichert.

Wie bereits erwähnt, sollen alle Rechnungen erst bei der Zuweisung einer Matrix-Vektor-Multiplikation auf einen Ergebnisvektor durchgeführt werden. Zu diesem Zweck ist folgender Zuweisungsoperator definiert:

- `Vector& Vector::operator -= (const constAB<Matrix,Vector>& MV)`
  Zuweisungsoperator: Ist `A` eine Matrix und sind `r` und `u` Vektoren, so für der Befehl `r -= A * u` dazu, dass die Methode `A.MultiplyMinus(u,r)` ausgeführt wird. Das liegt daran, dass der Befehl `A * u` ein `constAB<Matrix,Vector>`-Objekt erzeugt, dass daraufhin dem Argument `MV` dieses Operators übergeben wird. Die eigentliche Berechung des Ergebnisses wird somit erst beim Aufrufen dieses Zuweisungsoperators durchgeführt. Ein Speichern von Zwischenergebnissen ist dadurch nicht erforderlich.

8.1.4 Der Jacobi-Vorkonditionierer

```
1  // file:      PFEM/10/Main.C
2
3  #include "Mesh.h"
4  #include "Algebra.h"
5  #include "Plot.h"
6
7  const int levels = 0;
8
9  double f (const Point& x) { return x[0] + x[1]; }
10
11 class Tensor { ... };
12
13 Point operator * (const Tensor& T, const Point y) { ... }
14
15 void Dirichlet (const Mesh& M, MatrixGraph& G, Vector& u) { ... }
16
17 void Residual (const Mesh& M, const Vector& u, Vector& r) { ... }
18
19 void Jacobian (const Mesh& M, const Vector& u, Matrix& A) {
20     const Point GradN[3] = { Point(-1,-1), Point(1,0), Point(0,1) };
21     A = 0.0;
22     for (Mesh::cell c=M.cells(); c!=M.cells_end(); ++c) {
23         Tensor T(c);
24         double area = 0.5 * T();
25         Point G[3];
26         for (int i=0; i<3; ++i) G[i] = T * GradN[i];
27         for (int i=0; i<3; ++i)
28             for (int j=0; j<3; ++j)
29                 A(c[i],c[j]) += area * G[i] * G[j];
30     }
31     A.ClearDirichlet();
32 }
33
34 int main(int argv, char** argc) {
35     PPM = new ParallelProgrammingModel (&argv,&argc);
36     Mesh M;
37     ProcSets PS;
38     Distribute(M,PS);
39     int m = PPM->Sum(M.size());
40     mout << m << " coarse cells on " << PPM->size() << " processors" << endl;
41     for (int l=0; l<levels; ++l) M.Refine(PS);
42     m = PPM->Sum(M.size());
43     mout << m << " fine cells on " << PPM->size() << " processors" << endl;
44     MatrixGraph G(M,PS);
45     pout << G.size() << " unknowns on proc " << PPM->proc() << endl;
46     pout << G.Size() << " matrix entries on proc " << PPM->proc() << endl;
```

```
47      Vector u(G);
48      Dirichlet(M,G,u);
49      u << f;
50      Vector r(G);
51      Residual(M,u,r);
52      Matrix A(G);
53      Jacobian(M,u,A);
54      Vector c(G);
55      c = 0;
56      u += c;
57      r -= A * c;
58      pout << norm(r) << " residual norm " << endl;
59      Plot(M,u);
60      delete PPM;
61      return 0;
62  }
```

In dieser Version des parallelen Programms wird das Lineare Gleichungssystem der Finite-Element-Methode mit dem Jacobi-Vorkonditionierer iterativ gelöst. Zur Berechnung des Vorkonditionierers wurde die folgende Funktion definiert:

- `void Jacobian (const Mesh& M, const Vector& u, Matrix& A)`
  Die Funktion berechnet zu einem Gitter `M` die Matrix `A` des Jacobi-Verfahrens.

8.2 Paralleles Programm 11

8.2.1 Beschreibung

8.2.2 Dateien

PFEM/11/Parallel.h
PFEM/11/Parallel.C
PFEM/11/Point.h
PFEM/11/Mesh.h
PFEM/11/Algebra.h
PFEM/11/Plot.h
PFEM/11/Main.C
PFEM/11/Makefile
PFEM/11/mesh
PFEM/11/out.gnu

8.2.3 Vektoren und Matrizen (Teil 3)

```
1  // file:      PFEM/11/Algebra.h
2
3  #ifndef _ALGEBRA_H_
4  #define _ALGEBRA_H_
5
6  #include "Mesh.h"
7
8  #include <valarray>
9
10 class MatrixGraph { ... };
11
12 template <class A, class B> class constAB { ... };
13
14 template <class A, class B> constAB<A,B> operator * (const A& a, const B& b) { ... }
15
16 class Matrix;
17
18 class Vector : public valarray<double> { ... };
19
20 inline double operator * (const Vector& u, const Vector& v) { ... }
21
22 inline double norm (const Vector& u) { ... }
23
24 class Matrix : public valarray<double> { ... };
25
26 inline Vector& operator << (Vector& u, double F (const Point&)) { ... }
27
28 inline ostream& operator << (ostream& s, const valarray<double>& v) {
29     for (int i=0; i<v.size(); ++i) s << " " << v[i];
30     return s << endl;
31 }
32
33 void Collect (Vector& r) { ... }
34
35 void DistributeDirichlet (MatrixGraph& G, Vector& u) { ... }
36
37 void Accumulate (Vector& c) {
38     ExchangeBuffer E;
39     const ProcSets& PS = c.GetProcSets();
40     for (procset p = PS.procsets();p!=PS.procsets_end(); ++p) {
41         if (c.find_row(p()) == c.rows_end()) continue;
42         for (int i=0; i<p.size(); ++i) {
43             int q = p[i];
44             if (q == PPM->proc()) continue;
45             E.Send(q) << p() << c(p());
46         }
```

```
47      }
48      E.CommunicateSizeBuffer();
49      for (int q=0; q<PPM->size(); ++q)
50          while (E.Receive(q).size() < E.Receive(q).Size()) {
51              Point x;
52              double a;
53              E.Receive(q) >> x >> a;
54              c(x) += a;
55          }
56  }
57
58  Vector& Vector::operator -= (const constAB<Matrix,Vector>& MV) { ... }
59
60  #endif
```

In der Datei `Algebra.h` wurden in dieser Version des parallelen Programms eine weitere Funktion sowie ein Operator definiert:

- `inline ostream& operator << (ostream& s, const valarray<double>& v)`
  Ausgabeoperator: Gibt die Werte eines `double`-wertigen Wertearrays `v` in einen Ausgabestrom `s` aus.

- `void Accumulate (Vector& c)`
  Die Funktion `Accumulate` überführt einen Vektor `c` in seine *konsistente* Darstellung. Das bedeutet, dass die einem Gitterpunkt zugeordnete Vektorkomponente in allen parallelen Prozessen, die sich in der Prozessmenge des Gitterpunktes befinden, gespeichert wird.
  Dazu durchläuft jeder aufrufende Prozess die einzelnen Komponenten des seines lokalen Teilvektors. Enthält die Prozessmenge eines zugehörigen Gitterpunktes weitere Prozesse, so sendet der aufrufende Prozess den Punkt und den Wert der Komponente an die diese Prozesse (42–50). Anschließend liest jeder Prozess die empfangenen Punkte und Werte aus, und addiert diese Werte zu den entsprechenden, eigenen Vektorkomponenten hinzu (51–57).

8.2.4 Der Gauß-Seidel-Vorkonditionierer

```
1  // file:      PFEM/11/Main.C
2
3  #include "Mesh.h"
4  #include "Algebra.h"
5  #include "Plot.h"
6
7  const int levels = 2;
8  const double ReductionFactor = 0.0001;
```

```
9  const int MaxIter = 1000;
10 double f (const Point& x) { return x[0]*x[0]*x[0]; }
11
12 class Tensor { ... };
13
14 Point operator * (const Tensor& T, const Point y) { ... }
15
16 void Dirichlet (const Mesh& M, MatrixGraph& G, Vector& u) { ... }
17
18 void Residual (const Mesh& M, const Vector& u, Vector& r) { ... }
19
20 void Jacobian (const Mesh& M, const Vector& u, Matrix& A) { ... }
21
22 void GaussSeidel (Vector& c, const Matrix& A, const Vector& r) {
23     Vector D = c;
24     for (int i=0; i<c.size(); ++i)
25     D[i] = A[A.Diag(i)];
26     Accumulate(D);
27     for (int i=0; i<c.size(); ++i) {
28     double R = r[i];
29     int e = A.Diag(i);
30     for (++e; e<A.Diag(i+1); ++e) {
31         int j = A.Column(e);
32         if (j < i)  R -= A[e] * c[j];
33     }
34     c[i] = R / D[i];
35     }
36     Accumulate(c);
37 }
38
39 int main(int argv, char** argc) {
40     PPM = new ParallelProgrammingModel (&argv,&argc);
41     Mesh M;
42     ProcSets PS;
43     Distribute(M,PS);
44     int m = PPM->Sum(M.size());
45     pout << m << " coarse cells of " << M.size() << " cells on "
46      << PPM->size() << " processors" << endl;
47     for (int l=0; l<levels; ++l) M.Refine(PS);
48     m = PPM->Sum(M.size());
49     mout << m << " fine cells on " << PPM->size() << " processors" << endl;
50     MatrixGraph G(M,PS);
51     pout << G.size() << " unknowns on proc " << PPM->proc() << endl;
52     pout << G.Size() << " matrix entries on proc " << PPM->proc() << endl;
53     Vector u(G);
54     Dirichlet(M,G,u);
55     u << f;
56     Vector r(G);
```

```
57      Residual(M,u,r);
58      double d = norm(r);
59      double eps = ReductionFactor * d;
60      Matrix A(G);
61      Jacobian(M,u,A);
62      Vector c(G);
63      for (int iter=0; iter<MaxIter; ++iter) {
64          mout << "|r(" << iter << ")| = " << d << endl;
65          if (d < eps) break;
66          GaussSeidel(c,A,r);
67          u += c;
68          r -= A * c;
69          d = norm(r);
70      }
71      Residual(M,u,r);
72      d = norm(r);
73      mout << "check r " << d << endl;
74      Plot(M,u);
75      delete PPM;
76      return 0;
77  }
```

In dieser Version des parallelen Programms wird das lineare Gleichungssystem der Finite-Element-Methode mit dem Gauß-Seidel-Vorkonditionierer iterativ gelöst. Zur Berechnung des Vorkonditionierers wurde die folgende Funktion definiert:

- `void GaussSeidel (Vector& c, const Matrix& A, const Vector& r)`
  Die Funktion berechnet zu einer Matrix `A` und zu einem Residuenvektor `r` das Gauss-Seidel-Inkrement `c`.

8.3 Paralleles Programm 12

8.3.1 Beschreibung

8.3.2 Dateien

PFEM/12/Parallel.h
PFEM/12/Parallel.C
PFEM/12/Point.h
PFEM/12/Mesh.h
PFEM/12/Algebra.h
PFEM/12/Assemble.h
PFEM/12/Plot.h
PFEM/12/Main.C

PFEM/12/Makefile
PFEM/12/mesh
PFEM/12/out.gnu

In dieser Version des parallelen Programms enthält die Header-Datei `Assemble.h` die Definitionen der Funktionen `Dirichlet`, `Residual` und `Jacobian` sowie die Definition der Klasse `Tensor`.

8.3.3 Mehrgitterverfahren (Teil 1)

```
1  // file:     PFEM/12/Main.C
2
3  #include "Mesh.h"
4  #include "Algebra.h"
5  #include "Plot.h"
6
7  double f (const Point& x) { return x[0]*x[0]*(0.5-x[0])*x[1]*(0.5-x[1]); }
8
9  #include "Assemble.h"
10
11 const int levels = 4;
12 const double ReductionFactor = 0.0001;
13 const double Epsilon = 0.0000001;
14 const int MaxIter = 1000;
15
16 void Interpolate (const Mesh& M, const Vector& w, Vector& u) {
17     for (Mesh::cell c=M.cells(); c!=M.cells_end(); ++c) {
18         u(c[0]) = w(c[0]);
19         u(c[1]) = w(c[1]);
20         u(c[2]) = w(c[2]);
21         u(0.5*(c[0]+c[1])) = 0.5 * (w(c[0])+w(c[1]));
22         u(0.5*(c[1]+c[2])) = 0.5 * (w(c[1])+w(c[2]));
23         u(0.5*(c[0]+c[2])) = 0.5 * (w(c[0])+w(c[2]));
24     }
25 }
26
27 void GaussSeidel (Vector& c, const Matrix& A, const Vector& r) { ... }
28
29 void Solve (Vector& u, const Matrix& A, Vector& r) {
30     double d = norm(r);
31     double eps = Epsilon + ReductionFactor * d;
32     Vector c(u);
33     for (int iter=0; iter<MaxIter; ++iter) {
34         mout << "|r(" << iter << ")| = " << d << endl;
35         if (d < eps) break;
36         GaussSeidel(c,A,r);
```

```
37          u += c;
38          r -= A * c;
39          d = norm(r);
40      }
41  }
42
43  class Meshes : public vector<Mesh> {
44   public:
45      Meshes (ProcSets& PS, int J) : vector<Mesh>(J+1) {
46          Distribute((*this)[0],PS);
47          for (int j=0; j<J; ++j) {
48              (*this)[j+1] = (*this)[j];
49              (*this)[j+1].Refine(PS);
50          }
51      }
52  };
53
54  class MatrixGraphs {
55      vector<MatrixGraph*> G;
56   public:
57      MatrixGraphs (const Meshes& M, const ProcSets& PS) : G(M.size()) {
58          for (int j=0; j<M.size(); ++j) G[j] = new MatrixGraph(M[j],PS);
59      }
60      ~MatrixGraphs () { for (int j=0; j<G.size(); ++j) delete G[j]; }
61      MatrixGraph& operator [] (int j) { return *G[j]; }
62  };
63
64  int main (int argv, char** argc) {
65      PPM = new ParallelProgrammingModel(&argv,&argc);
66      ProcSets PS;
67      Meshes M(PS,levels);
68      int m = PPM->Sum(M[0].size());
69      mout << m << " coarse cells on " << PPM->size() << " processors" << endl;
70      MatrixGraphs G(M,PS);
71      Vector u(G[0]);
72      u << f;
73      Dirichlet(M[0],G[0],u);
74      Vector r(G[0]);
75      Residual(M[0],u,r);
76      Matrix A(G[0]);
77      Jacobian(M[0],u,A);
78      Solve(u,A,r);
79      Vector old(u);
80      for (int j=0; j<levels; ++j) {
81          Vector u(G[j+1]);
82          Interpolate(M[j],old,u);
83          Dirichlet(M[j+1],G[j+1],u);
84          Vector r(G[j+1]);
```

```
85          Residual (M[j+1],u,r);
86          Matrix A(G[j+1]);
87          Jacobian(M[j+1],u,A);
88          Solve (u,A,r);
89          old = u;
90          if (j<levels-1) continue;
91          Residual(M[j+1],u,r);
92          double d = norm(r);
93          m = PPM->Sum(M[j+1].size());
94          mout << "check r " << d << " level " << j+1 << " cells " << m << endl;
95          Plot(M[j+1],u);
96      }
97      delete PPM;
98      return 0;
99  }
```

In der zwölften Version des parallelen Programms wird ein Mehrgitter-Verfahren zur Lösung des Finite-Element-Problems eingesetzt. Dazu wurden weitere Klassen und Funktionen definiert. Die Klasse **Meshes** ist dabei ein Vektor, der eine Folge von Gittern speichert, die jeweils durch uniforme Verfeinerung auseinander hervorgehen. Eine solche Gitterfolge bezeichnen wir als kaskadisches Mehrgitter. Für die Klasse `Meshes` sind dazu folgenden Methoden definiert:

- `Meshes::Meshes (ProcSets& PS, int J)`
  Konstruktor: Erzeugt einen neues Objekt der Klasse `Meshes`. Das so erzeugte kaskadische Mehrgitter besteht aus `(J+1)` ineinander verschachtelten Gittern, die jeweils durch eine `Refine`-Operation auseinander hervorgehen. Ist `M` so ein Mehrgitter, dann ist `M[0]` das gröbste Gitter, das mittels `Distribute` erzeugt wird (48). Nach dem Aufruf des Konstruktors enthält das referenziell übergebene `ProcSets`-Objekt `PS` die Prozessmengen der Punktes des feinsten Gitters `M[J]`.

Die ebenfalls neu definierte Klasse **MatrixGraphs** ist ein Vektor von `MatrixGraph`-Objekten. Die Klasse dient dazu, zu einem Mehrgitter die einzelnen Matrixgraphen zu speichern. Dafür wurden die folgenden Methoden definiert:

- `MatrixGraphs::MatrixGraphs (const Meshes& M, const ProcSets& PS)`
  Konstruktor: Erzeugt einen neues Objekt der Klasse `MatrixGraphs`. Der so erzeuget Vektor enthält zu einem referenzielle übergebenen Mehrgitter `M` die einzelnen Matrixgraphen. Ist `G` ein solches `MatrixGraph`-Objekt, so speichert für einen Index `i` das das `MartixGraph`-Objekt `M[i]` den Matrixgraphen des Gitters `M[i]`. Das `ProcSets`-Objekt `PS` muss dabei die Prozesmengen des feinsten Gitters enthalten.

Zusätzlich wurden für das Mehrgitter-Verfahren folgende Funktionen definiert:

- `void Interpolate (const Mesh& M, const Vector& w, Vector& u)`
  Die Funktion `Interpolate` führt eine lineare Interpolation von einem Gitter auf sein verfeinertes Gitter durch. Der Vektor `w` enthält dabei die Werte auf den Gitterpunkten des groben Gitters. Nach dem Aufruf der Funktion enthält der Vektor `u` auf den Gitterpunkten des groben Gitters diesselben Werte wie `w`. An den neuen Punktes des verfeinerten Gitters (d.h. auf den Kantenmittelpunkten des groben Gitters) enthält der Vektor `u` die linear interpolierten Werte bezüglich der beiden Nachbarpunkte.

- `void Solve (Vector& u, const Matrix& A, Vector& r)`
  Die Funktion `Solve` löst das lineare Gleichungssystem mit der Steifigkeitsmatrix `A` iterativ mit dem Gauß-Seidel-Vorkonditionierer. Die Funktion gibt die approximierte Lösung `u` sowie den Residuenvektor `r` zurück. Dabei endet die Iteration, wenn die Euklid-Norm des Residuenvektors um den global definierten Faktor `ReductionFactor` reduziert wurde. Durch die global definierte Konstante `Epsilon` wird dabei die Maschinengenauigkeit des Rechners berücksichtigt. Die Anzahl der Iteration ist außerdem durch die global definiert Konstante `MaxIter` beschränkt.

8.4 Paralleles Programm 13

8.4.1 Beschreibung

8.4.2 Dateien

```
PFEM/13/Parallel.h
PFEM/13/Parallel.C
PFEM/13/Point.h
PFEM/13/Mesh.h
PFEM/13/Algebra.h
PFEM/13/Assemble.h
PFEM/13/Plot.h
PFEM/13/Main.C
PFEM/13/Makefile
PFEM/13/mesh
PFEM/13/out.gnu
```

Die Definition der Klasse `Meshes` erfolgt in dieser Version des parallelen Programms in der Header-Datei `Mesh.h`. Die Klasse `MatrixGraphs` wird in der Datei `Algebra.h` definiert.

8.4.3 Mehrgitterverfahren (Teil 2)

```
1  // file:     PFEM/13/Main.C
2
3  #include "Mesh.h"
4  #include "Algebra.h"
5  #include "Plot.h"
6
7  double f (const Point& x) { return x[0]*x[0]*(0.5-x[0])*x[1]*(0.5-x[1]); }
8
9  #include "Assemble.h"
10
11 const int levels = 4;
12 const double ReductionFactor = 0.0001;
13 const double Epsilon = 0.0000001;
14 const int MaxIter = 1000;
15
16 void Interpolate (const Mesh& M, const Vector& w, Vector& u) { ... }
17
18 void Restrict (const Mesh& M, Vector o, Vector& r) {
19     r = 0;
20     for (Mesh::cell c=M.cells(); c!=M.cells_end(); ++c) {
21         r(c[0]) = o(c[0]);
22         r(c[1]) = o(c[1]);
23         r(c[2]) = o(c[2]);
24     }
25     for (Mesh::cell c=M.cells(); c!=M.cells_end(); ++c) {
26         r(c[0]) += 0.5 * o(0.5*(c[0]+c[1]));
27         r(c[1]) += 0.5 * o(0.5*(c[0]+c[1]));
28         o(0.5*(c[0]+c[1])) = 0;
29         r(c[1]) += 0.5 * o(0.5*(c[1]+c[2]));
30         r(c[2]) += 0.5 * o(0.5*(c[1]+c[2]));
31         o(0.5*(c[1]+c[2])) = 0;
32         r(c[0]) += 0.5 * o(0.5*(c[0]+c[2]));
33         r(c[2]) += 0.5 * o(0.5*(c[0]+c[2]));
34         o(0.5*(c[0]+c[2])) = 0;
35     }
36     r.ClearDirichlet();
37     Collect(r);
38 }
39
40 void GaussSeidel (Vector& c, const Matrix& A, const Vector& r) { ... }
41
42 void Solve (Vector& u, const Matrix& A, Vector& r) { ... }
43
44 class Matrices {
45     vector<Matrix*> A;
46 public:
```

```
47      Matrices (MatrixGraphs& G) : A(G.size()) {
48      for (int j=0; j<G.size(); ++j) A[j] = new Matrix (G[j]);
49      }
50      ~Matrices () { for (int j=0; j<A.size(); ++j) delete A[j]; }
51      Matrix& operator [] (int j) { return *A[j]; }
52      const Matrix& operator [] (int j) const { return *A[j]; }
53  };
54
55  void Cycle (int level, const Meshes& M, const MatrixGraphs& G,
56          Vector& u, const Matrices& A, Vector r) {
57      u = 0;
58      if (level == 0) {
59          Solve(u,A[level],r);
60          return;
61      }
62      Vector c(G[level]);
63      GaussSeidel(c,A[level],r);
64      u += c;
65      r -= A[level] * c;
66      Vector d(G[level-1]);
67      Vector w(G[level-1]);
68      Restrict(M[level-1],r,d);
69      Cycle(level-1,M,G,w,A,d);
70      Interpolate(M[level-1],w,c);
71      u += c;
72      r -= A[level] * c;
73      GaussSeidel(c,A[level],r);
74      u += c;
75      r -= A[level] * c;
76  }
77
78  void Multigrid (const Meshes& M, const MatrixGraphs& G,
79          Vector& u, const Matrices& A, Vector& r) {
80      int level = M.size()-1;
81      double d = norm(r);
82      double eps = Epsilon + ReductionFactor * d;
83      Vector c(u);
84      for (int iter=0; iter<MaxIter; ++iter) {
85          mout << "|r(" << iter << ")| = " << d << endl;
86          if (d < eps) break;
87          Cycle(level,M,G,c,A,r);
88          u += c;
89          r -= A[level] * c;
90          d = norm(r);
91      }
92  }
93
94  const int MaxLevels = 1;
```

```
95
96  int main (int argv, char** argc) {
97      PPM = new ParallelProgrammingModel(&argv,&argc);
98      int levels = MaxLevels;
99      if (argv>1) levels = atoi(argc[1]);
100     ProcSets PS;
101     Meshes M(PS,levels);
102     int m = PPM->Sum(M[0].size());
103     mout << m << " coarse cells on " << PPM->size() << " processors" << endl;
104     MatrixGraphs G(M,PS);
105     Matrices A(G);
106     for (int l=0; l<levels; ++l) {
107         Vector u(G[l]);
108         Dirichlet(M[l],G[l],u);
109         Jacobian(M[l],u,A[l]);
110     }
111     Vector u(G[levels]);
112     u << f;
113     Dirichlet(M[levels],G[levels],u);
114     Vector r(G[levels]);
115     Residual(M[levels],u,r);
116     Jacobian(M[levels],u,A[levels]);
117     Multigrid(M,G,u,A,r);
118     Plot(M[levels],u);
119     m = PPM->Sum(M[levels].size());
120     mout << m << " cells on level " << levels << endl;
121     delete PPM;
122     return 0;
123 }
```

Literatur

P. Bastian: Paralleles Rechnen I und II, Vorlesungsskript Heidelberg

G. Alefeld, I. Lenhardt, H. Obermaier: Parallele numerische Verfahren, Springer 2002

A. Frommer: Lösung linearer Gleichungssysteme auf Parallelrechnern, Vieweg 1990