Parallelisierung numerischer Verfahren

PD Dr. Volker John

30. Juni 2003

Inhaltsverzeichnis

1 Motivation

1.1 Rechnertypen und Einsatzgebiete

Es gibt verschiedene Arten von Computern, die sich in ihrer Hardware, ihren Anschaffungskosten und ihren Anwendungsgebieten unterscheiden:

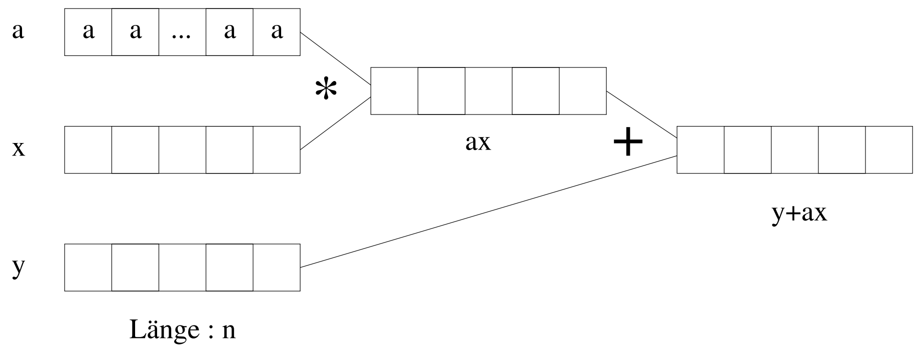
- PC 2003 (2.66 Ghz, 512 MB RAM, 1200 Euro (Aldi))
- Workstations 2002 (625 MHz, 2 GB RAM, ≈ 10000 Euro (Hewlett Packard))
- Vektorrechner. Diese Rechner können eine Operation nicht nur mit skalaren Größen durchführen sondern mit ganzen Arrays (Vektoren).

  Beispiel 1.1 Vektorrechner. Es ist $\mathtt{y} := \mathtt{y} + \mathtt{ax}$ zu berechnen, wobei $\mathtt{x}, \mathtt{y}$ n-dimensionale Arrays sind und a eine Zahl ist.

  Standardrechner:

  ```
  for(i = 0; i < n; i++)
    y[i] += a * x[i];
  ```

  Vektorrechner mit Registerlänge n.

  Die Operationen $*$ und $+$ werden für alle Einträge der Arrays gleichzeitig durchgeführt. Das ist eine Form der Parallelität auf Maschinenebene.

  Vektorrechner verlieren zunehmend an Bedeutung. Sie werden in dieser Vorlesung nicht weiter behandelt.

- **Parallelrechner**
  - **Massiv-Parallelrechner**, siehe Abbildung 1.1
    - 1995 Parsitec GC (80 MHz, 1.5 GB RAM, 9.2 MFlops / Prozessor)
    - 2000 HP N-Class (3 Stück, gesamt 24 Prozessoren, 48 GB RAM, 42 GFlops peak performance)
    - 2002 HP-Superdome (128 GB RAM, 192 GFlops peak performance)
    - **Cluster von Workstations** (oder PCs)

Aktuelle Übersichten über die Leistungsfähigkeit von Parallelrechnern findet man unter
`http://www.top500.org`
`http://www.specbench.org` (Standard Performance Evaluation Corporation), siehe auch Abbildung 1.2.

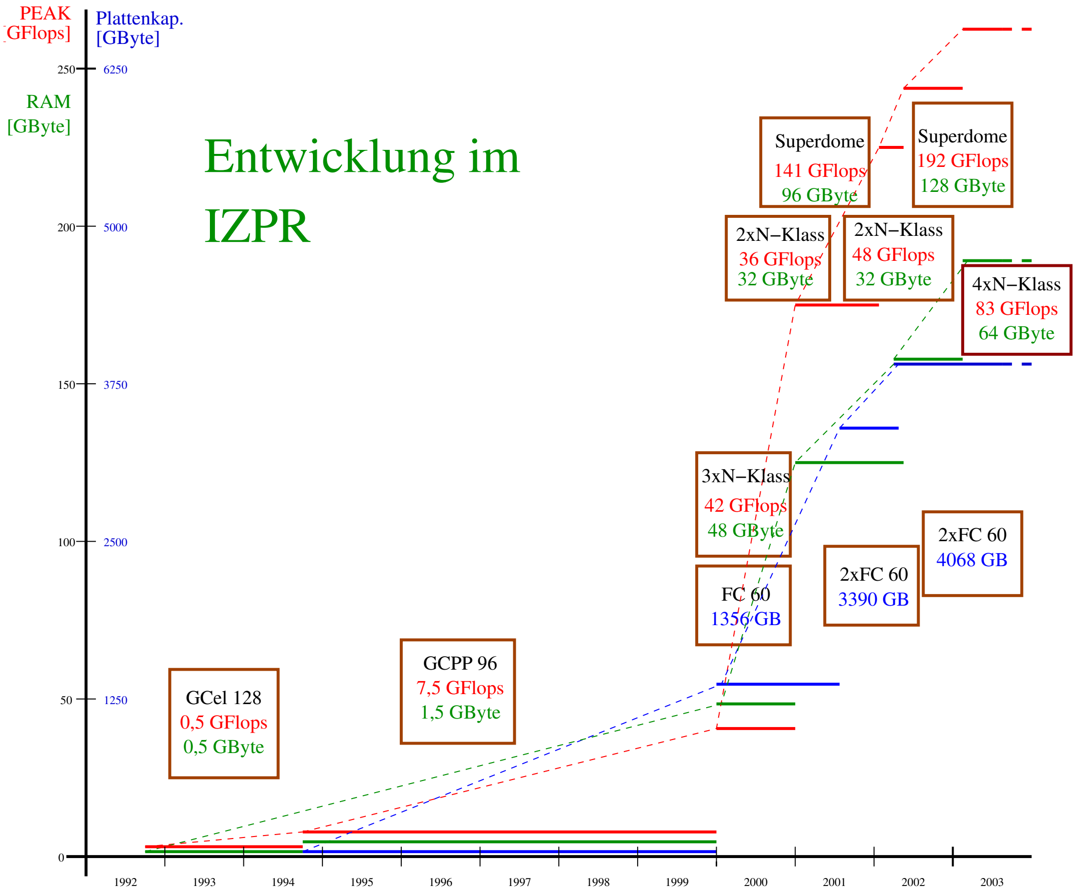
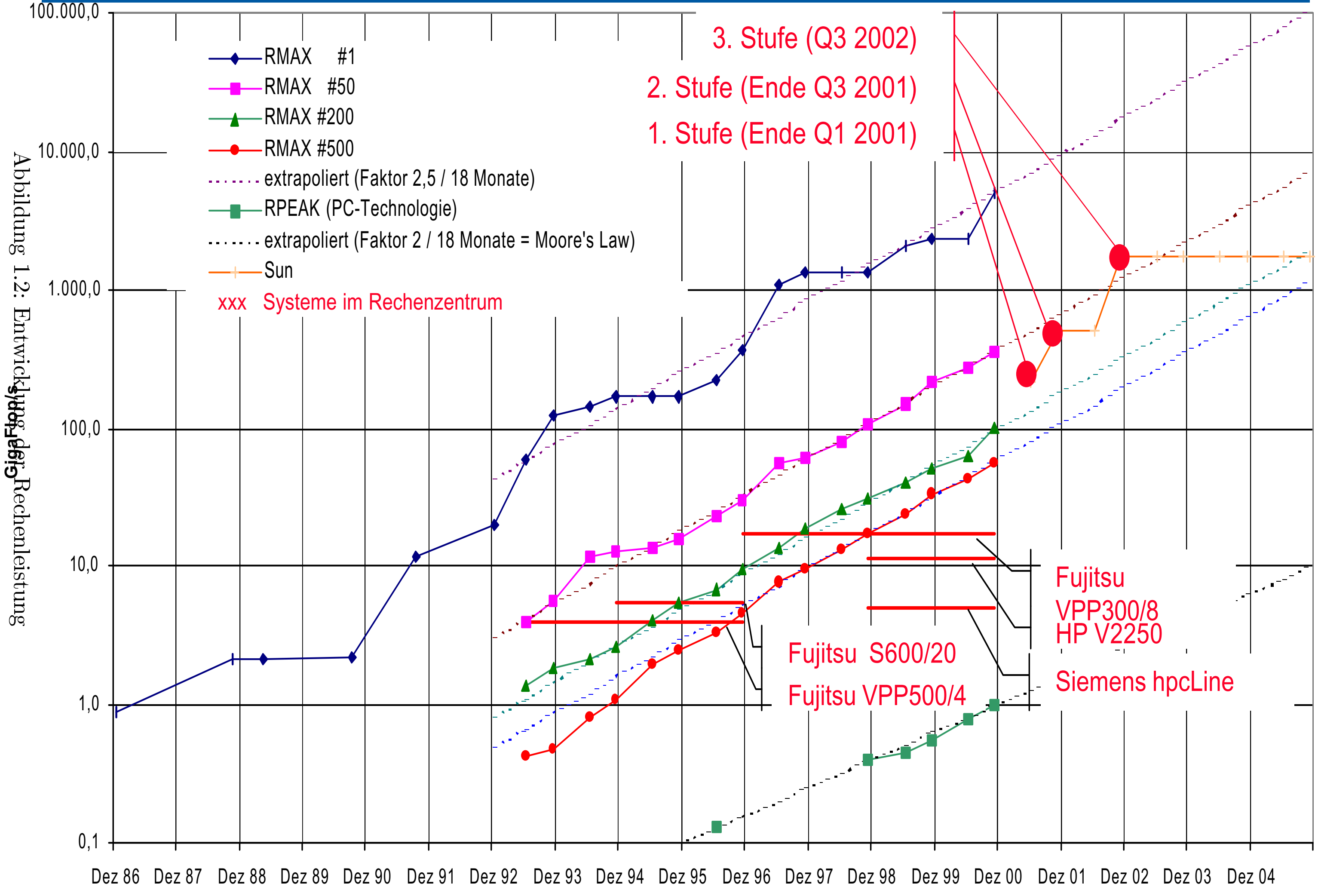


Abbildung 1.1: Entwicklung der Hardware im IZPR

Entwicklung der Rechenleistung im internat. Vergleich

(Liste der 500 größten Rechensysteme)



C.Bischof/K.Brühl 6. Februar 2001

Abbildung 1.2: Entwicklung der Rechenleistung

Die verschiedenen Rechnertypen besitzen unterschiedliche *Einsatzgebiete*:
- PC : Wirtschaft (Textverarbeitung, Tabellenkalkulation), Schule, Privathaushalte (Spiele)
- Workstations, Vektorrechner, Parallelrechner : Forschung (Universitäten, Wirtschaft), Anwendung der Forschung (Rechenzentren, Wirtschaft, Universitäten), Verwaltung (Lohnabrechnung in der Wirtschaft)

Einige Forschungsgebiete und Anwendungen, die Parallelrechner verwenden sind
- Simulation von Naturprozessen, z. Bsp. *Strömungen*, chemische Reaktionen
  Anwendungen : Flugzeug- und Kraftfahrzeugdesign (Aerodynamik, Strömungsverhältnisse im Kraftfahrzeug), Ausbreitungen von Umweltverschmutzungen, Wetter- und Klimavorhersage, Computermedizin
- Simulation von industriellen Prozessen, z. Bsp. Schüttguttransport, Brückenkonstruktionen
- Optimierungs- und Suchverfahren, die im wesentlichen auf Probieren (Vergleichen) beruhen
- Bildverarbeitung

Die Simulation von Naturprozessen führt im Kern oft auf die Lösung linearer Gleichungssysteme mit schwach besetzten Matrizen. Die Parallelisierung von Verfahren zur Lösung solcher Gleichungssysteme ist ein Schwerpunkt dieser Vorlesung.

Beispiel 1.2 Navier-Stokes Gleichungen. Inkompressible Strömungen in einem Gebiet Ω werden durch die (stationären) Navier-Stokes Gleichungen beschrieben

$$\begin{aligned} -\nu\Delta u + (u\cdot\nabla)u + \nabla p &= 0 \quad \text{in } \Omega \quad \text{(Impulserhaltung)} \\ \nabla\cdot u &= 0 \quad \text{in } \Omega \quad \text{(Massenerhaltung)}, \end{aligned}$$

wobei u die Geschwindigkeit, p der Druck und ν die kinematische Viskosität ist. Die Gleichungen müssen noch mit Randbedingungen ausgestattet werden. Wegen der Nichtlinearität in der Impulserhaltungsgleichung werden die Navier-Stokes Gleichungen iterativ gelöst. In jedem Iterationsschritt hat man ein lineares System von Gleichungen mit schwach besetzten Matrizen zu lösen. In einer konkreten Realisierung erhält man folgende Ergebnisse
- Rechner: HP Superdome
- Kommunikation: MPI
- Unbekannte im Gleichungssystem : ≈ 1.58 Mill.

Prozessoren	1	2	4	8	16	32
Zeit (s)	6965	3810	1809	891	432	230
Kommunikation (%)	0	1.4	2.8	2.4	4.1	6.6
parallel Effizienz		0.91	0.96	0.97	1.00	0.94

Die parallele Effizienz ist der Quotient der Zeit, die auf einem Prozessor benötigt wurde durch das Produkt der Zeit auf p Prozessoren mit der Anzahl p der Prozessoren, siehe Definition 5.10.

1.2 Vorteile, Nachteile und Herausforderungen von Parallelrechnern

Parallelrechner bieten folgende *Vorteile*:
- Mehrere Prozessoren können zur Lösung eines Problems benutzt werden (*Hardwareparallelität*):
  - Problem muß vom Programmierer (oder durch entsprechende Software) in Teilprobleme zerlegt werden,
  - auf jedem Prozessor läuft ein (eigenes) Programm, das eine Anzahl Teilprobleme löst,
  - zur Lösung benötigte Daten, die sich auf anderen Prozessoren befinden, müssen sich durch Datenaustausch (Kommunikationen) beschafft werden

- i.a. besitzen Parallelrechner viel mehr Speicher als ein einzelner Rechner $\Longrightarrow$ große Probleme können gelöst werden.

Es gibt jedoch auch einige *Nachteile*. Der Einfluß einiger Nachteile kann durch adequate Auswahl von numerischen Verfahren und gute Softwareentwicklung klein gehalten werden.

- Kommunikationen beeinflussen die Effizienz der verwendeten Verfahren negativ; oft ist Kommunikation langsam im Vergleich zur Rechengeschwindigkeit $\Longrightarrow$ so wenig wie nötig und so geschickt wie möglich kommunizieren !
- vorhandene Software für Einzelrechner muß an den Stellen erweitert werden, an denen Kommunikationsbedarf besteht und neue Software muß unter dem parallelen Blickwinkel konzipiert werden (*algorithmische Parallelität*):
  - bei der Auswahl der verwendeten Verfahren ist neben deren numerischen Verhalten jetzt auch auf Parallelisierbarkeit und parallele Effizienz zu achten,
  - die ausgewählten Verfahren müssen unter dem Blickwinkel eines minimalen Kommunikationsbedarfes analysiert werden,
  - Kommunikation muß programmiert werden.
- Massiv-Parallelrechner
  - benötigen viel Ausstattung (klimatisierte Räume, hochbelastbare Fußböden, unterbrechungsfreie Stromversorgung, ...),
  - keine einheitlichen Betriebssysteme und vom Hersteller entwickelten Routinen zur Kommunikationssteuerung (aber: es gibt portable Bibliotheken, mit denen man auf den meisten Parallelrechnern die Kommunikation steuern kann, z. Bsp. *MPI*).
  - früher vergleichsweise anfällig für Hard- und Softwarefehler, hat sich in den letzten Jahren verbessert

Eine Einführung in MPI ist Bestandteil der Vorlesung.

Bevor man ein Programm parallelisiert, muß man sich überlegen, wie die in dem Programm genutzten numerischen Verfahren parallelisiert werden können. Es stellt sich heraus, daß es Verfahren gibt, die sich gut parallelisieren lassen und Verfahren, bei denen eine effiziente Parallelisierung schwierig ist.

Beispiel 1.3 Gut und schlecht parallelisierbare Verfahren. Zu lösen ist das lineare Gleichungssystem

$$Ax = b. \tag{1.1}$$

1. Die LU-Zerlegung (Gauß-Verfahren) mit Spaltenpivotsuche zur Lösung von (1.1) ist für kleine Gleichungssysteme (mit bis zu mehreren Hundert Unbekannten) ein numerisch stabiles und effizientes Verfahren. Dieses Verfahren läßt sich jedoch nur schwierig und mit hohen Kommunikationsverlusten parallelisieren.
2. Das Jacobi-Verfahren zur Lösung von (1.1)

$$x^{n+1} = x^n - \operatorname{diag}(A)^{-1}(Ax^n - b).$$

ist ein Iterationsverfahren, das falls es konvergiert, sehr langsam konvergiert. Es läßt sich aber einfach und mit wenig Kommunikationsverlusten parallelisieren.

Die im Mittelpunkt der Vorlesung stehenden großen Gleichungssysteme mit schwach besetzten Matrizen werden im allgemeinen iterativ gelöst. Die verwendeten Iterationsverfahren lassen sich relativ gut parallelisieren.

1.3 Empfohlene Literatur

Es werden die Bücher von Frommer [Fro90], Haase [Haa99] und Saad [Saa96] empfohlen. Das Buch von Saad findet man elektronisch auf seiner Homepage

`http://www-users.cs.umn.edu/~saad/books.html`

2 Einführung in MPI

MPI (message passing interface) ist eine Bibliothek, die Kommandos für eine Rechner-unabhängige Parallelisierung von Programmen enthält. Ein Programm, welches mit MPI auf irgendeiner Computerplattform parallelisiert wurde, läuft sofort auch auf anderen Computerplattformen. Für jede Computerplattform existiert eine spezielle MPI-Bibliothek. Diese sind frei erhältlich (public domain). Die Implementierung der in MPI verfügbaren Kommandos basiert auf den speziellen Kommandos der jeweiligen parallelen Computer. Somit hat man einen zusätzlichen Aufwand der zu Effizienzverlusten führt. Das ist der Preis für die Flexibilität.

MPI ist im Interdisziplinären Zentrum für paralleles Rechnen (IZPR) und im IAN auf folgenden Plattformen erhältlich

- massiv parallele Rechner (neu): HP N-Classes, HP Superdome (green, ritz, paulus),
- Workstations,
- massiv parallele Rechner (alt): Parsytec GC Power Plus.

Compilieren von MPI-Programmen

Ein Programm wird wie folgt **compiliert**:

```
mpicc +O3 filename -o executable
```

Mehr Optionen findet man mit `man mpicc`

Starten von MPI-Programmen

Ein Programm wird wie folgt **gestartet**:

- Für einen massiv Parallelrechner nutzt man:
  ```
  mpirun -np #1 executable
  ```
  Hierbei ist `#1` die Anzahl der Prozessoren.
- Auf den Workstationclustern des IAN gibt es zwei Versionen von `mpirun`. Bei der neuen Version nutzt man:
  ```
  mpirun -f appfile
  ```
  wobei `appfile` die Befehle enthält, zum Beispiel
  ```
  \hspace*{2em} -h fest -np 7 /homes/john/VORLESUNGEN/PAR_NUM/MPI/a.out
  \hspace*{2em} -h bessel -np 5 /homes/john/VORLESUNGEN/PAR_NUM/MPI/a.out
  ```
  Damit würde das Program `a.out` mit zwölf Prozessen laufen, davon sieben auf fest und 5 auf bessel. Genauso startet man auch ein Programm auf den HP N-Classes, daß Prozessoren von mehreren N-Classes oder auch vom HP Superdome nutzen soll
  ```
  \hspace*{2em} -h paulus -np 7 /homes/john/VORLESUNGEN/PAR_NUM/MPI/a.out
  \hspace*{2em} -h green -np 5 /homes/john/VORLESUNGEN/PAR_NUM/MPI/a.out
  ```
  Bei der alten Version von mpirun nutzt man:
  ```
  mpirun -np #1 -machinefile filename executable
  ```
  Dabei ist `filename` eine ASCII-Datei die die Namen der Computer enthält, auf denen das Programm laufen soll, zum Beispiel

  leibniz
  newton
  leibniz
  newton

  Der Zugriff auf diese Rechner muss in der Datei `$HOME/.rhosts` erlaubt sein.

Informationen über MPI findet man im Internet unter
`www-unix.mcs.anl.gov/mpi/index.html`
Die MPI-Bibliothek enthält mehr als 100 Kommandos. Im Prinzip sind jedoch 6 grundlegende Kommandos ausreichend. Diese werden zunächst eingeführt. Andere nützliche Kommandos werden im Anschluss vorgestellt.

Starten und Beenden von MPI

- `MPI_Init` - initialisiert die MPI-Umgebung, das ist immer das erste MPI-Kommando
- `MPI_Finalize` - schließt die MPI-Umgebung, das ist immer das letzte MPI-Kommando

Informationen über die MPI-Umgebung

Jeder Prozeß sollte wissen, mit wie vielen anderen Prozessen er zusammenarbeitet und welche interne Prozeßnummer er besitzt. Diese Informationen erhält er mit
- `MPI_Comm_size` - gibt die Anzahl der Prozesse an. Falls diese Anzahl p ist, werden diese in MPI mit $0, \ldots, p-1$ numeriert.
- `MPI_Comm_rank` - gibt die interne Nummer des aufrufenden Prozesses zurück.

Punkt- zu Punkt-Kommunikationen

The Basisform der Kommunikation besteht darin, dass Prozess `i` Daten zu Prozess `j` sendet und Prozess `j` diese Daten empfängt. Das geschieht mit den Kommandos
- `MPI_Send` - senden von Daten,
- `MPI_Recv` - (receive) empfangen von Daten.

Ein Datenaustausch zwischen zwei Prozessen kann auch mit Hilfe des Kommandos `MPI_Sendrecv` erfolgen.

Beispiel 2.1 Paralle Berechnung von Mittelwert und Maximum von n Zahlen. Gegeben ist ein Vektor der Länge n. Von den Einträgen des Vektors soll das arithmetische Mittel

$$\frac{1}{n}\sum_{i=1}^{n} a_i$$

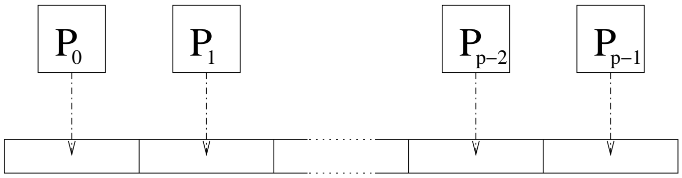
sowie die größte Komponente bestimmt werden. Ein paralleles Programm zur Lösung dieser Aufgabe soll wie folgt ablaufen:
- Prozess `0` liest die Länge n des Vektors ein.
- Der Vektor wird in p (Anzahl der Prozessoren) gleich lange Teile zerlegt (dabei wird n so aufgerundet, dass n durch p teilbar ist)).
- Prozess `0` füllt den Vektor mit Zufallszahlen.
- Prozess `0` schickt die Information über die Länge der Teilvektoren und danach die jeweiligen Teilvektoren an die anderen Prozesse.
- jeder Prozess berechnet die Summe der Komponenten sowie die größte Komponente (lokales Maximum).
- diese Informationen werden an Prozess `0` gesandt welcher das globale Maximum sowie den Mittelwert berechnet.
- Prozess `0` gibt die Ergebnisse aus.

Eine Lösung dieser Aufgabe findet man auf Seite 49.

Hausaufgabe: Man lade das Programm von
`http://www-ian.math.uni-magdeburg.de/~home/john/`
versuche das Programm zum Laufen zu bringen und den Inhalt und den Syntax des Programmes zu verstehen.

Globaler Datenaustausch, gemeinschaftliche (collective) Kommunikation

- `MPI_Bcast` - (broad cast) verteilt eine Botschaft vom Prozess `root` auf alle anderen Prozesse, wobei ein Baum-Topologie benutzt wird, siehe Seite 16
- `MPI_Reduce` - reduziert Daten von allen Prozessen auf ein einzelnes Datum auf dem `root`-Prozess, die Art der Reduktion (Summe, Maximum, ...) wird durch `MPI_Op op` angegeben, eine Baum-Topologie wird benutzt
- `MPI_Allreduce` - erster Schritt wie `MPI_Reduce`, danach wird das erhaltene Ergebnis wieder auf alle Prozessoren verteilt (wie `MPI_Bcast`)
- `MPI_Barrier` - blockiert alle Prozesse so lange, bis jeder Prozess eine vorgegebene Programmstelle (das ist die Barriere) erreicht hat, als Ergebnis ist das Programm synchronisiert
- `MPI_Gather` - der `root`-Prozess empfängt (sammelt) Informationen von allen anderen Prozessen und verknüpft diese Informationen zu einem Array gemäß der Nummern der anderen Prozesse



Array auf Prozess root (hier Prozess 0)

- `MPI_Scatter` - der `root`-Prozess verteilt die Daten eines Arrays auf die anderen Prozesse, der Array wird in p gleiche Teile zerlegt, Teil i wird zu Prozess `i` gesandt

Zeitmessung

- `MPI_Wtime` - gibt die aktuelle Zeit zurück

Beispiel 2.2 Fortsetzung von Beispiel 2.1. Die Aufgabenstellung sei wie im Beispiel 2.1. Nun nutze man die Befehle zur gemeinschaftlichen Kommunikation um den Mittelwert und das Maximum zu berechnen. Eine Lösung dieser Aufgabe findet man auf Seite 52.

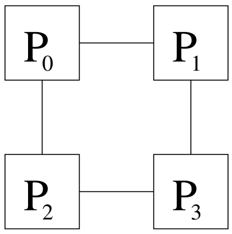
3 Topologien

Die Prozesse eines parallel ablaufenden Programmes müssen miteinander verbunden sein um zu kommunizieren.

Definition 3.1 *Ein **Link** ist die Verbindung zweier Prozesse (Knoten). Das Netzwerk aller Prozesse wird **Topologie** genannt.*

Die Prozessoren eines parallelen Computers sind physisch vernetzt (physikalische Topologie). Die Daten- oder Kommunikationsstruktur eines Programmes definiert eine logische or virtuelle Topologie. Die logische Topologie wird durch das Betriebssystem auf die physikalische Topologie abgebildet.

Beispiel 3.2 P_0 will eine Nachricht an P_3 senden, das heisst es muss ein logischer Link existieren. Wird das Kommando zum Senden der Nachricht aufgerufen, so organisiert das Betriebssystem ob die Nachricht über P_1 oder P_2 gesandt wird, weil keine direkte Verbindung von P_0 und P_3 existiert.

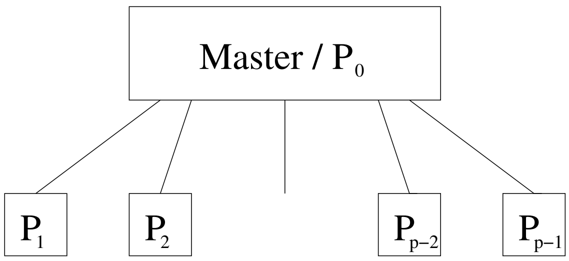


Am effizientesten ist es, wenn physikalische und logische Topologie übereinstimmen. Das kann im allgemeinen jedoch nicht erreicht werden, zum Beipiel wegen der Datenstrukturen im Programm. Ausserdem würde diese Eigenschaft immer nur für einen speziellen Parallelcomputer gelten und würde beim nächsten schon nicht mehr gelten. Ein charakteristisches Merkmal einer Topologie ist die maximale Weglänge.

Definition 3.3 *Die **maximale Weglänge** ist die maximale Anzahl von Links welche eine Nachricht benötigt von einem Knoten der Topologie zu einem anderen Knoten.*

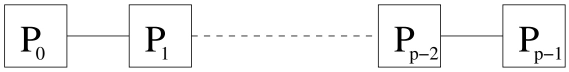
Im folgenden wird ein Überblick über einige logische Topologien gegeben. In der Literatur sind weitere zu finden.

Master-Worker oder Master-Slave



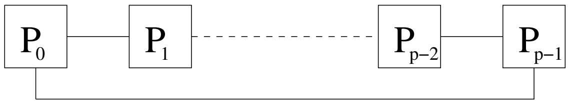
Die maximale Weglänge ist 2. Der Master ist jedoch an jeder Kommunikation beteiligt. Diese Topologie wurde im Beispiel 2.1 verwendet.

Kette (pipe)



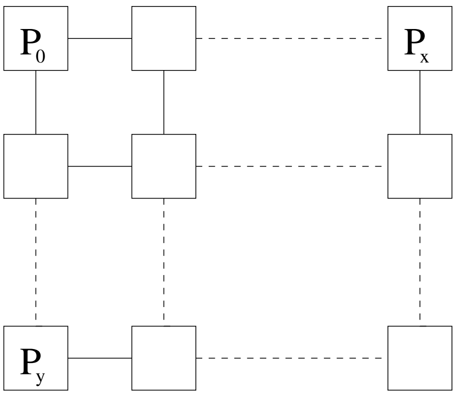
Die maximale Weglänge ist $p-1$.

Ring



Die maximale Weglänge ist $(p-1)/2$.

Array (2d)



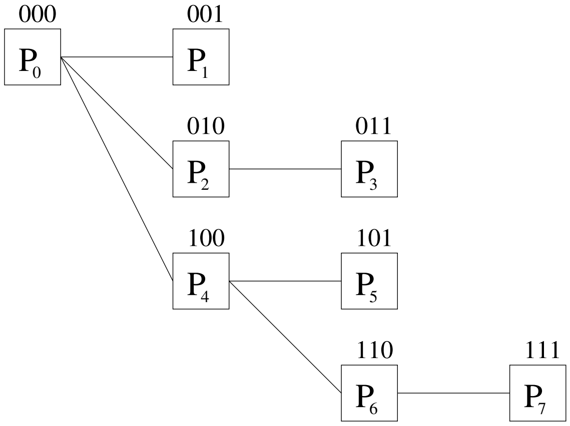
Die maximale Weglänge ist $x+y$.

Array (3d)

Die maximale Weglänge ist $x+y+z$.

(Optimaler) Baum (tree)

Sei p die Anzahl der Prozesse mit $2^{d-1} < p \leq 2^d$. Ein Baum wird optimal genannt, falls er die Tiefe d besitzt (Wurzel = Tiefe 0) und maximal d Links pro Knoten vorhanden sind. Der Prozess P_0 wird Wurzel genannt.

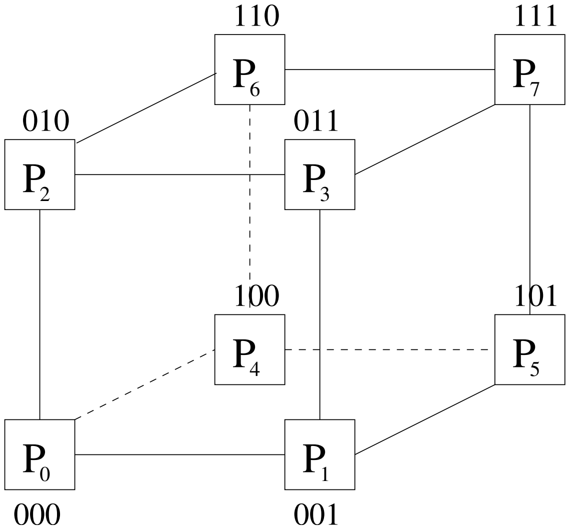


Die maximale Weglänge ist dann $2d - 1$ ($P_7 \rightarrow P_3$). Diese logische Topologie wird von `MPI_Bcast`, `MPI_Reduce` und `MPI_Allreduce` verwendet.

Bei einem fetten Baum (fat tree) ist ein Prozessor desto leistungsstärker, je näher er an der Wurzel ist.

Hypercube

Sei p die Anzahl der Prozesse mit $p = 2^d$. Ein Hypercube besitzt genau d Links pro Knoten. Die Nummer benachbarter Knoten unterscheidet sich in der Binärdarstellung genau um ein Bit.



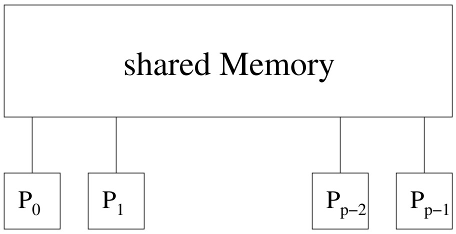
Die maximale Weglänge ist d.

Beispiel 3.4 Gegeben seien p Prozesse und jeder Prozess kenne eine Zahl. Das arithmetische Mittel dieser p Zahlen ist unter Verwendung der Topologien Master-Worker, Kette, Ring, Baum und Hypercube zu berechnen. Zum Schluss soll jeder Prozess dieses arithmetische Mittel kennen. Eine Lösung dieser Aufgabe findet man auf Seite 54.

4 Einteilung von Parallelrechnern nach Speicherzugriff

Shared Memory (Gemeinsamer Speicher)

Alle Prozessoren nutzen denselben Speicher. Die Kommunikation zwischen den Prozessoren findet nur innerhalb des gemeinsamen Speichers statt.



Vorteile:
- Jeder Prozessor hat auf alle Daten Zugriff. Daraus resultiert eine schnelle Kommunikation.
- Man kann auch Programme, die nicht parallelisiert sind, auf einem Prozessor laufen lassen und den gesamten Speicher nutzen.

Nachteil: Zugriffskonflikte müssen verhindert werden. Zugriffskonflikte entstehen zum Beispiel, wenn ein Prozessor ein Datum lesen will und ein anderer zu gleichen Zeit das Datum beschreiben will. Welcher Prozessor darf nun seine Operation zuerst ausführen ? Eine leistungsfähige Verwaltung ist zur Steuerung des Speichers notwendig. Allerdings wird durch einen großen Verwaltungsaufwand die Effizienz des parallelen Systems vermindert. Da der Verwaltungsaufwand mit wachsender Prozessorzahl sich immer mehr erhöht, ist die Nutzung von shared Memory nur bei einer relativ kleinen Anzahl von Prozessoren sinnvoll.

Distributed Memory (Verteilter Speicher)

Nur Prozessor i, $0 \leq i \leq p-1$, hat Zugriff auf den Speicher i. Die verschiedenen lokalen Speicher sind miteinander verbunden (physikalische Topologie). Falls ein Prozessor Daten benötigt, die auf einem anderen Prozessor vorhanden sind, müssen diese Daten durch eine explizite Kommunikation gesendet werden.

Vorteile:
- keine Zugriffskonflikte,
- kann mit preiswerter Hardware (PC) realisiert werden.

Nachteil: Ein Datenaustausch erfolgt über physikalische Datenkanäle. Die dafür benötigte Zeit kann im allgemeinen nicht vernachläsigt werden. Die Geschwindigkeit des Datennetzwerkes ist von großer Bedeutung um die Kommunikationsverluste gering zu halten.

Beispiel 4.1 *Kommunikationsverluste bei der parallelen Lösung der Navier-Stokes Gleichungen auf dem Parsytec GC Power Plus (verteilter Speicher).* Dieses Beispiel soll ein Gefühl für die Größe von Kommunikationsverlusten geben. Es wurden zwei Verfahren getestet. Der Löser 1 beruht vorwiegend auf der Lösung vieler kleiner (lokaler) Systeme während der Löser 2 wenige große (globale) Systeme löst. Das benutzte Programm wurde mit speziellen Befehlen des Parsytec GC Power Plus programmiert und nicht mit MPI.

Prozessoren	Löser 1 Kommunikationanteil an Gesamtzeit in %	Quelle	Löser 2 Kommunikationanteil an Gesamtzeit in %	Quelle
4	4 - 13	[Joh98]		
8	2 - 5	[JT00]	7 - 30	[JT00]
16	3 - 15	[Joh98]	15	[Joh99]
	2 -5	[Joh99]		
32	12	[Joh00]	24	[Joh00]

Dieses Beispiel unterstreicht bereits eine allgemeine Gesetzmäßigkeit für die effiziente Parallelisierbarkeit von Verfahren: ”lokale” Verfahren sind besser zu parallelisieren als Verfahren, die globale Informationen benötigen.

Bemerkung 4.2 Man versucht den Speicher so zu organisieren, dass die Vorteile beider Speichertypen genutzt werden können.

- Virtuelles shared Memory: Ein shared Memory wird durch Software auf einem distributed memory System simuliert.
- HPUX-N Class: Eine N-Class besteht aus 8 Prozessoren mit shared Memory. Der Parallelcomputer setzt sich aus mehreren N-Classes zusammen, die wiederum miteinander verknüpft sind. Aus Sicht der N-Classes hat man nun ein distributed Memory System.

5 Synchronisation und Bewertung paralleler Algorithmen

5.1 Synchronisation paralleler Algorithmen

In diesem Abschnitt werden einige Begriffe eingeführt, die mit der Synchronisation paralleler Algorithmen verbunden sind.

Die Granularität misst die Größe der Programmabschnitte, die ohne Kommunikation mit anderen Prozessen ausführbar sind.

Definition 5.1 *Das Verhältnis von Rechenkosten zu Kommunikationskosten wird Granularität genannt. Algorithmen, die lange Programmabschnitte besitzen, die unabhängig voneinander parallel ausgeführt werden können, besitzen eine grobe Granularität (grobkörnige Algorithmen). Algorithmen, die nach wenigen Arithmetikoperationen wieder Daten von anderen Prozessen benötigen, haben eine feine Granularität (feinkörnige Algorithmen).*

Im allgemeinen ist die Kommunikationsgeschwindigkeit eines Parallelrechners viel langsamer als der Zugriff auf den Hauptspeicher oder gar die Rechengeschwindigkeit. Darum ist es günstig grobkörnige Algorithmen auf Parallelrechnern zu verwenden. Manchmal ist es möglich, die Granularität existierender Algorithmen zu vergröbern, indem man einige Details ändert. Ein Beispiel ist die Arbeit zur Lösung des Grobgittersystem beim Mehrgitter W-Zyklus die man spendiert, siehe [Joh98]. Die Theorie sagt, dass man das Grobgittersystem exakt lösen muss. Das ist jedoch manchmal so groß, dass man ein iteratives Verfahren zur Lösung nehmen sollte. Auf einem Parallelrechner sollte man weniger Iterationen durchführen als auf einem seriellen Rechner.

Die Zeit für eine Kommunikation setzt sich aus zwei Anteilen zusammen, der Zeit für die Initialisierung der Kommunikation (Latency) und der Zeit für die Übertragung

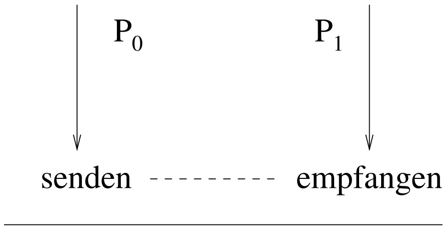
$$t_{\text{Kommunikation}} = t_{\text{Initialisierung}} + t_{\text{Übertragung}}$$

Nach dieser Formel und auch nach praktischen Erfahrungen ist das Senden einer langen Nachricht besser als das Senden mehrerer entsprechend kurzer Nachrichten. Man spart Initialisierungszeiten. Die Initialisierungszeiten liegen im allgemeinen im Mikrosekundenbereich.

Die Synchronisation paralleler Prozesse is notwendig, um undefinierte Zustände zu verhindern. Ein undefinierter Zustand ist gegeben, falls das Ergebnis eines Teilprogrammes von der Geschwindigkeit der einzelnen Prozesse abhängt und das Ergebnis nicht eindeutig bestimmt ist. Solch ein Zustand kann zum Beispiel beim Zugriff auf gemeinsamen Speicher auftreten. Man unterscheidet synchrone und asynchrone Kommunikationen.

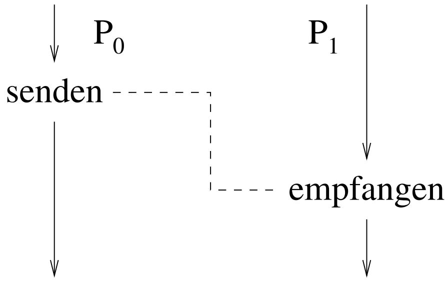
Definition 5.2 *Synchrone Kommunikation. Alle Prozesse warten solange bis alle Prozesse signalisert haben, dass sie zum Datenaustauch bereit sind.*

Beispiel 5.3 Der Prozess P_0 ist zum senden bereit. Er wartet solange, bis der Prozess P_1 bereit ist zum Empfangen. Der empfangende Prozess P_1 muss ohnehin solange warten, bis er Daten von P_0 erhält.



Definition 5.4 *Asynchrone Kommunikation. Die Prozesse senden und empfangen die Daten unabhängig vom Zustand der anderen Prozesse.*

Beispiel 5.5 Der sendende Prozess P_0 fragt nicht nach ob P_1 bereit zum Empfangen ist, sondern er sendet seine Daten und kümmert sich auch nicht, ob P_1 diese empfangen hat. Nach dem Senden arbeitet P_0 sein Programm weiter ab. Falls der empfangende Prozess tatsächlich noch nicht bereit ist, die Daten von P_0 zu empfangen, so werden diese in einem Puffer abgespeichert bis P_1 empfangsbereit ist.



Im allgemeinen ist die asynchrone Kommunikation effizienter als die synchrone Kommunikation.

5.2 Bewertung paralleler Algorithmen

Dieser Abschnitt beschäftigt sich mit der Frage, wie man parallele Algorithmen miteinander vergleichen kann und wie der Gewinn gemessen werden kann, den man bei der Nutzung eines parallelen Algorithmus hat. Die Grundlage der Vergleiche ist immer die Rechenzeit, die auch das wichtigste Maß in Anwendungen ist.

Speedup

In der Literatur sind mindestens fünf verschiedene Arten von Speedup defininiert. Here sollen die zwei gebräuchlisten eingeführt werden.

Definition 5.6 *Es sei ein Problem $\mathbb{P}$ gegeben der Größe n welches auf p Prozessoren mit dem Algorithmus A gelöst wird. Der relative Speedup ist definiert als*

$$\textit{relativer Speedup}(n,p) = \frac{\textit{Zeit um } \mathbb{P} \textit{ mit } A \textit{ auf einem Prozessor zu lösen}}{\textit{Zeit um } \mathbb{P} \textit{ mit } A \textit{ auf } p \textit{ Prozessoren zu lösen}}.$$

Der relative Speedup ist keine feste Zahl. Er hängt von der Problemgröße n und der Anzahl der Prozessoren p ab. Für eine feste Problemgröße n verhält sich der relative Speedup im allgemeinen wie in Abb. 5.1. Man sieht, dass der erreichbare relative Speedup von der Problemgröße abhängt. Er ist im allgemeinen kleiner als p.

Definition 5.7 *Es seien ein Problem $\mathbb{P}$ gegeben der Größe n und ein paralleler Algorithmus A. Beim realen Speedup wird die parallele Rechenzeit mit der Zeit verglichen, die man braucht, um*

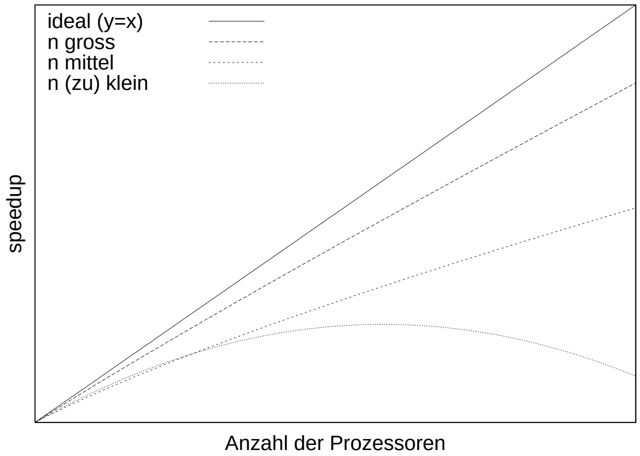


Abbildung 5.1: Relative Speedups.

$\mathbb{P}$ mit dem schnellsten verfügbaren Algorithmus auf einem Prozessor (des gleichen Computers) zu lösen:

$$\begin{aligned}&\textit{relativer Speedup}(n,p)\\ =\;& \frac{\textit{Zeit um } \mathbb{P} \textit{ mit dem besten seriellen Algorithmus auf einem Prozessor zu lösen}}{\textit{Zeit um } \mathbb{P} \textit{ mit } A \textit{ auf } p \textit{ Prozessoren zu lösen}}.\end{aligned}$$

Da für viele Probleme der schnellste Algorithmus nicht bekannt ist oder für andere Probleme kein Algorithmus der schnellste bei allen Parameterkonstellationen ist, nimmt man die Laufzeit des sequentiellen Algorithmus den man "in der Praxis nutzt" anstelle der Laufzeit des schnellsten Algorithmus.

Scaleup

Oft ist es nicht möglich, das gegebene Problem auf einem Prozessor zu lösen weil nicht genügend Speicherplatz zur Verfügung steht. Andererseits kann das Problem, welches man auf einem Prozessor lösen kann, zu klein sein um einen vernünftigen Speedup zu geben. Diese Punkte führen zu der Idee, dass man nicht nur die Anzahl der Prozessoren erhöht um parallele Algorithmen zu bewerten sondern auch die Problemgröße.

Definition 5.8 *Es sei ein Problem $\mathbb{P}$ gegeben der Größe n welches auf einem Prozessor mit dem Algorithmus A gelöst wird. Der Scaleup ist definiert als*

$$\textit{Scaleup}(n,p) = \frac{\textit{Zeit um } \mathbb{P} \textit{ der Größe } n \textit{ mit } A \textit{ auf einem Prozessor zu lösen}}{\textit{Zeit um } \mathbb{P} \textit{ der Größe } np \textit{ mit } A \textit{ auf } p \textit{ Prozessoren zu lösen}}.$$

Beispiel 5.9 Ein Arbeiter kann eine Grube von 1 m^3 in einer Stunde graben.
- Speedup-Arbeiter: 1000 Arbeiter sollen diese Grube graben. Sie behindern sich gegenseitig und der Speedup ist sehr schlecht.
- Scaleup-Arbeiter: 1000 Arbeiter sollen jeder eine Grube von 1 m^3 graben. Der Scaleup ist perfekt.

Relative und reale Effizienz

Die relative und reale Effizienz sind eng mit dem Speedup verbunden.

Definition 5.10 *Es sei ein Problem $\mathbb{P}$ gegeben der Größe n welches auf p Prozessoren mit dem Algorithmus A gelöst wird. Die relative (reale) Effizienz ist*

$$\textit{relative (reale) Effizienz}(n,p) = \frac{\textit{relativer (realer) Speedup}}{p}.$$

Skalierbarkeit

Die Eigenschaft der Skalierbarkeit eines parallelen Systems wird benutzt um die Änderung in der Leistung eines parallelen Verfahrens zu beschreiben, wenn die Problem- und die Computergröße anwachsen.

Definition 5.11 *Ein paralleler Algorithmus wird skalierbar genannt, falls seine Leistung sich ständig verbessert wenn sowohl die Problemgröße als auch die Anzahl der Prozessoren erhöht werden. Mit anderen Worten, wenn sich der Scaleup nicht beträchtlich verschlechtert.*

6 Das Modellproblem

Das in der Vorlesung betrachtete Modellproblem ist die Laplace-Gleichung im Einheitsquadrat mit Dirichlet-Randbedingungen

$$\begin{aligned} -\Delta u &= f \quad \text{in } (0,1)^2 \\ u &= g \quad \text{auf } \Gamma. \end{aligned} \tag{6.1}$$

Hierbei ist Δ der Laplace-Operator

$$\Delta u(x,y) = u_{xx}(x,y) + u_{yy}(x,y) = \frac{\partial^2 u}{\partial x^2}(x,y) + \frac{\partial^2 u}{\partial y^2}(x,y)$$

und Γ ist der Rand des Quadrates $(0,1)^2$. Der Gradient von u wird mit ∇u bezeichnet:

$$\nabla u = \begin{pmatrix} u_x \\ u_y \end{pmatrix}.$$

Es gilt $\Delta u = \nabla \cdot \nabla u$. Die Funktionen f und g seien hinreichend glatt (stetig oder differenzierbar).

Die Gleichung (6.1) beschreibt Diffusionsvorgänge (Ausbreitung durch Teilchenbewegung).

6.1 Schwache Lösung der Laplace-Gleichung

Eine Lösung von (6.1), die zweimal stetig differenzierbar ist und die samt ihren partiellen Ableitungen bis zur zweiten Ordnung stetig auf Γ fortgesetzt werden kann, wird klassische oder starke Lösung von (6.1) genannt. Für die Existenz einer klassischen Lösung müssen die rechte Seite f und die Randbedingung g gewisse Differenzierbarkeitseigenschaften besitzen. In Anwendungen sind diese Eigenschaften oft nicht gegeben. Deshalb benötigt man einen schwächeren Lösungsbegriff.

Zunächst werden einige Funktionenräume auf einem Gebiet Ω (zum Beispiel $\Omega = (0,1)^2$) eingeführt:

$$\begin{aligned} L^2(\Omega) &= \left\{ v \ ; \ \int_\Omega v^2 d\mathbf{x} < \infty \right\}, \quad \|v\|_{L^2} = \left(\int_\Omega v^2 d\mathbf{x} \right)^{1/2}, \\ H^1(\Omega) &= \left\{ v \ ; \ \int_\Omega v^2 + \nabla v \cdot \nabla v \ d\mathbf{x} < \infty \right\}, \quad \|v\|_{H^1} = \left(\int_\Omega v^2 + \nabla v \cdot \nabla v \ d\mathbf{x} \right)^{1/2}, \\ H_0^1(\Omega) &= \left\{ v \ ; \ v \in H^1(\Omega), v|_\Gamma = 0 \text{ (im Sinne der Spur) } \right\}, \\ H_g^1(\Omega) &= \left\{ v \ ; \ v \in H^1(\Omega), v|_\Gamma = g \text{ (im Sinne der Spur) } \right\}. \end{aligned}$$

Ausserdem benötigen wir noch die Bilinearform in $L^2(\Omega)$

$$(v,w) := \int_\Omega vw d\mathbf{x}.$$

Es folgt $\|v\||_{L^2}^2 = (v,v)$.

Seien $V = H_0^1(\Omega)$ und $v \in V$. Man multipliziert (6.1) mit v und integriert über Ω:

$$(-\Delta u, v) = (-\nabla \cdot \nabla u, v) = (f,v).$$

Nun integriert man die linke Seite partiell (Greensche Formel):

$$(\nabla u, \nabla v) - \int_\Gamma (\nabla u \cdot \mathbf{n}) v ds = (f, v).$$

Hierbei ist $\mathbf{n}$ die nach aussen gerichtete Einheitsnormale an Γ. Das Randintegral verschwindet, da die Funktion v auf dem gesamten Rand verschwindet, also

$$(\nabla u, \nabla v) = (f, v). \tag{6.2}$$

Definition 6.1 *Die Funktion $u \in H_g^1(\Omega)$ wird schwache Lösung der Laplace-Gleichung (6.1) genannt, falls sie die Gleichung (6.2) für alle Testfunktionen $v \in V$ erfüllt.*

Sei $W = H_g^1(\Omega)$. Wir statten die Räume V und W mit abzählbaren Basen aus:

$$V = \operatorname{span}\{v_i, i = 1, 2, 3, \ldots\}, \quad W = \operatorname{span}\{w_i, i = 1, 2, 3, \ldots\}.$$

Das geht, da diese Räume separabel sind. Dann hat die unbekannte Lösung u die folgende Darstellung

$$u = \sum_{j=1}^{\infty} u_j w_j \tag{6.3}$$

mit unbekannten reellen Koeffizienten u_j. Da (6.2) bezüglich der Testfunktion linear ist, reicht es, dass (6.2) für alle Basisfunktionen v_i erfüllt ist. Setzt man (6.3) in (6.2) ein, erhält man

$$\sum_{j=1}^{\infty} u_j (\nabla w_j, \nabla v_i) = (f, v_i), \quad i = 1, 2, 3, \ldots. \tag{6.4}$$

Das ist ein unendlich-dimensionales lineares Gleichungssystem zur Bestimmung der Koeffizienten u_j.

6.2 Grundidee der Finite-Elemente-Methode (FEM)

Die Grundidee der FEM besteht einfach darin, (6.4) durch ein geeignetes endlich-dimensionales System zu ersetzen. Das geschieht, indem man die unendlich-dimensionalen Räume V und W durch geeignete endlich-dimensionale Approximationen $V^h \sim V$ und $W^h \sim W$ ersetzt (oft $V^h \subset V$, $W^h \subset W$, ist aber nicht notwendig).

Das endlich-dimensionale (diskrete) Problem lautet:
Finde $u^h \in W^h$, so dass

$$(\nabla u^h, \nabla v^h) = (f, v^h) \quad \text{für alle } v^h \in V^h. \tag{6.5}$$

Seien $V^h = \operatorname{span}\{v_i^h, i = 1, 2, \ldots, n\}$, $W^h = \operatorname{span}\{w_i^h, i = 1, 2, \ldots, n\}$, dann hat u^h die Darstellung

$$u^h = \sum_{j=1}^{n} u_j^h w_j^h.$$

Es reicht, wenn (6.5) für alle Basisfunktionen von V^h erfüllt ist. Man erhält folgende Bestimmungsgleichungen für die unbekannten reellen Koeffizienten u_j^h:

$$\sum_{j=1}^{n} u_j^h (\nabla w_j^h, \nabla v_i^h) = (f, v_i^h), \quad i = 1, 2, \ldots, n. \tag{6.6}$$

Das ist ein lineares Gleichunssystem der Größe $n \times n$, welches man auf dem Computer lösen kann:

$$Au = b, \quad A \in \mathbb{R}^{n \times n}, \ u = (u_1^h, \ldots, u_n^h) \in \mathbb{R}^n, \ b \in \mathbb{R}^n. \tag{6.7}$$

6.3 Lineare Finite Elemente in 2d

Das Modellgebiet $(0,1)^2$ wird mit einem Gitter wie in Abbildung 6.1 zerlegt, d.h. es gibt nur Gitterlinien, die parallel zu den Achsen sind und die entstehenden Vierecke werden mit der Diagonalen von links unten nach rechts oben in zwei Dreiecke zerlegt.

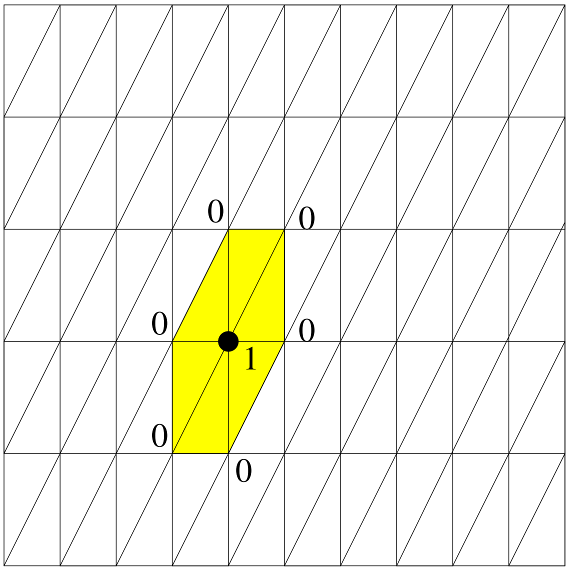


Abbildung 6.1: Zerlegung von $(0,1)^2$ und schematische Darstellung einer Basisfunktion

Die folgende Beschreibung der Finite-Elemenent-Methode mit linearen Elementen gilt für Triangulierungen beliebiger Gebiete $\Omega \subset \mathbb{R}^2$. Die Anzahl der inneren Dreiecksecken sei N. Dann ist

$$\begin{aligned} V^h &= \operatorname{span}\{v_i^h \ : \ v_i^h = 1 \text{ in der Ecke } i, \ v_i^h = 0 \text{ in allen anderen Ecken}, \\ &\qquad v_i^h \text{ ist stetig}, v_i^h \text{ ist auf jedem Dreieck linear}, v_i^h|_\Gamma = 0, i = 1,\ldots,N\}. \end{aligned}$$

Der Raum W^h wird ähnlich definiert, nur bezüglich des Randes sind Änderungen zu V^h nötig. Sei n die Anzahl aller Dreiecksecken, dann ist

$$\begin{aligned} W^h &= \operatorname{span}\{w_i^h \ : \ w_i^h = 1 \text{ in der Ecke } i, \ w_i^h = 0 \text{ in allen anderen Ecken}, \\ &\qquad w_i^h \text{ ist stetig}, w_i^h \text{ ist auf jedem Dreieck linear}, i = 1,\ldots,n\}. \end{aligned}$$

Die Basisfunktionen werden wegen ihrer Gestalt auch Hütchenfunktionen genannt, siehe Abbildung 6.1 für eine schematische Darstellung. Die Triangulierung wird mit $\mathcal{T}^h$ und ein Dreieck mit K bezeichnet.

Die Berechnung der Integrale in (6.6) wird auf Integrale über die Dreiecke zurückgeführt:

$$\begin{aligned} (\nabla w_j^h, \nabla v_i^h) &= \int_\Omega \nabla w_j^h \cdot \nabla v_i^h d\mathbf{x} = \sum_{K\in\mathcal{T}^h} \int_K \nabla w_j^h \cdot \nabla v_i^h d\mathbf{x}, \\ (f, v_i^h) &= \sum_{K\in\mathcal{T}^h} \int_K f v_i^h d\mathbf{x}. \end{aligned}$$

Wir betrachten nun ein beliebiges Dreieck K. Die Integrale auf den Dreiecken werden mit Hilfe der Seitenmittelpunktregel numerisch approximiert. Seien Q_0, Q_1, Q_2 die Mittelpunkte der Seiten von K und S_K der Schwerpunkt von K. Dann hat die Seitenmittelpunktregel die folgende Gestalt:

$$\int_K v d\mathbf{x} \approx \frac{|K|}{3}(v(Q_0) + v(Q_1) + v(Q_2)),$$

wobei $|K|$ der Flächeninhalt von K ist. Die Seitenmittelpunktregel ist exakt, falls v ein Polynom zweiten Grades ist. Jede Basisfunktion, die auf dem Dreieck K nicht verschwindet, nimmt in zwei Seitenmittelpunkten den Wert 0.5 an und im dritten Seitenmittelpunkt den Wert 0, siehe Abbildung 6.2. Für die rechte Seite folgt:

$$\begin{aligned}\int_K f(\mathbf{x}) v_i^h(\mathbf{x}) d\mathbf{x} &\approx \int_K f(S_K) v_i^h(\mathbf{x}) d\mathbf{x} = f(S_K) \int_K v_i^h(\mathbf{x}) d\mathbf{x} = f(S_K) \frac{|K|}{3}(0.5 + 0.5 + 0) \\ &= f(S_K) \frac{|K|}{3}.\end{aligned}$$

Die Gradienten der Basisfunktionen sind auf jedem Dreieck konstant. Damit folgt

$$\int_K \nabla w_j^h \cdot \nabla v_i^h d\mathbf{x} = \nabla w_j^h(S_K) \cdot \nabla v_i^h(S_K)|K|.$$

Seien $P_0 = (x_0, y_0), P_1 = (x_1, y_1), P_2 = (x_2, y_2)$ die Eckpunkte vom Dreieck K (im mathematisch positiven Sinn numeriert) und sei v_i^h die Basisfunktion, die in P_0 den Wert 1 annimmt. Man rechnet nach, dass

$$\nabla v_i^h = -\frac{1}{2|K|} \begin{pmatrix} y_2 - y_1 \\ x_1 - x_2 \end{pmatrix}.$$

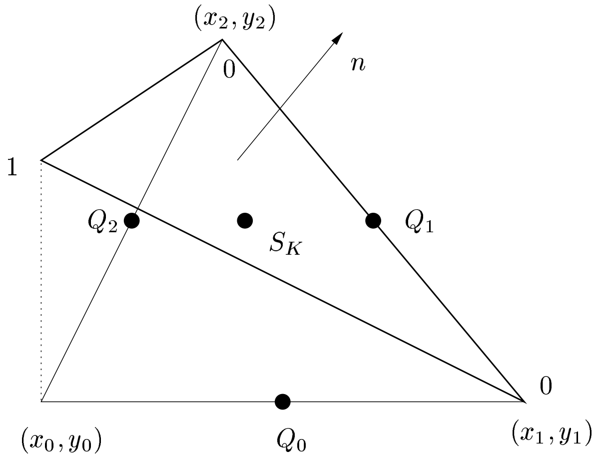


Abbildung 6.2: Basisfunktion auf einem Dreieck

Der Träger einer Funktion (support) ist der Abschluss der Menge aller Werte des Definitionsbereiches, in denen die Funktion nicht verschwindet:

$$\operatorname{supp}(v) = \overline{\{\mathbf{x} \ : \ v(\mathbf{x}) \neq 0\}}.$$

Der Träger der in Abbildung 6.1 dargestellten Basisfunktion ist beispielsweise die Vereinigung der sechs farbig dargestellten abgeschlossenen Dreiecke.

Die Einträge der Matrix A sowie der rechten Seite b in 6.7 erhält man durch Addition der Beiträge von den einzelnen Dreiecken:

$$\begin{aligned} a_{ij} &= \sum_{K \in \mathcal{T}^h | \operatorname{supp}(w_j^h) \cap \operatorname{supp}(v_i^h) \cap K| > 0} \nabla w_j^h(S_K) \cdot \nabla v_i^h(S_K)|K|, \qquad (6.8) \\ b_i &= \sum_{K \in \mathcal{T}^h, |\operatorname{supp}(v_i^h) \cap K| > 0} f(S_K) \frac{|K|}{3}. \end{aligned}$$

6.4 Die Konfiguration des Modellproblems auf dem Parallelrechner

Das Einheitsquadrat $(0,1)^2$ wird durch einen 2D-Array von Prozessen gemäß Abbildung 6.3 überdeckt. Die Überdeckungsgebiete seien für alle Prozesse kongruent. Jeder Prozess verwaltet die gleiche Anzahl von Gitterzellen. Die Anzahl der Prozesse in x-Richtung sei n_x und in y-Richtung n_y. Das Programm muss mit genau $n_x n_y$ Prozessen gestartet werden.

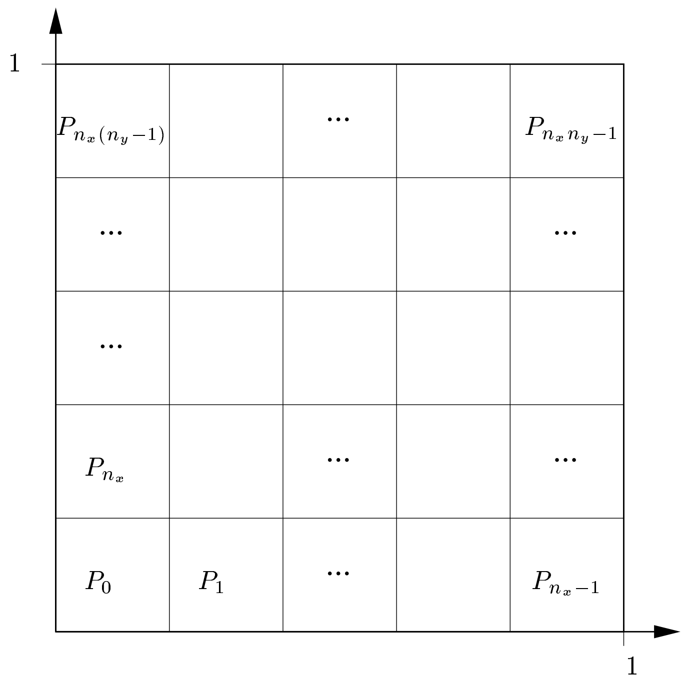


Abbildung 6.3: Überdeckung von $(0,1)$ durch die Prozesse

Die Gitterfeinheit sei auf jedem Prozess $n_{x,loc} \times n_{y,loc}$, so dass insgesamt $2n_{x,loc}n_{y,loc}$ Gitterzellen auf jedem Prozess vorhanden sind. Die linke untere Ecke von Prozess `rank` habe die Koordinaten (x_0, y_0). Diese Werte lassen sich wie folgt berechnen:

$\texttt{row} = (\texttt{int})\texttt{rank}/\texttt{n}_\texttt{x}$
$\texttt{y}_0 = \texttt{row} * 1.0/\texttt{n}_\texttt{y}$
$\texttt{column} = (\texttt{int})\texttt{rank}\%\texttt{n}_\texttt{x}$ (% ist modulo)
$\texttt{x}_0 = \texttt{column} * 1.0/\texttt{n}_\texttt{x}$

siehe Seite 60.

Die Koordinaten einer beliebigen Ecke (x_i, y_i) lassen sich wie folgt berechnen:

$\texttt{row} = (\texttt{int})\texttt{i}/\texttt{n}_{\texttt{x,loc}}$
$\texttt{y}_\texttt{i} = \texttt{y}_0 + \texttt{row} * 1.0/(\texttt{n}_{\texttt{y,loc}} * \texttt{n}_\texttt{x})$
$\texttt{column} = (\texttt{int})\texttt{i}\%\texttt{n}_{\texttt{x,loc}}$
$\texttt{x}_\texttt{i} = \texttt{x}_0 + \texttt{column} * 1.0/(\texttt{n}_{\texttt{x,loc}} * \texttt{n}_\texttt{x})$

siehe Seite 61.

Die lokalen Freiheitsgrade werden von links unten beginnend von links nach rechts und dann von unten nach oben numeriert.

7 Iterative Löser für lineare Gleichungssysteme

In diesem Kapitel wird die Parallelisierung von iterativen Lösern für ein lineares Gleichungssystem der Gestalt

$$Ax = b, \quad A \in \mathbb{R}^{n \times n}, x, b \in \mathbb{R}^n \tag{7.1}$$

betrachtet.

7.1 Grundlegende Matrix-Vektor-Operationen

Ein Vektor oder eine Matrix kann auf einem Parallelrechner auf zwei verschiedene Arten gespeichert werden.

Definition 7.1 *Ein Wert x auf dem Prozess P_i wird mit $P_i \to x$ bezeichnet. Dieser Wert wird* konsistent gespeichert *oder* überlappend gespeichert *genannt, falls*

$$x = P_i \to x$$

für alle Prozesse P_i auf denen x vorhanden ist. Er wird inkonsistent gespeichert *oder* additiv gespeichert *genannt, falls*

$$x = \sum_i P_i \to x.$$

Beispiel 7.2 Der Wert x solle auf den Prozessen P_0, P_1 und P_2 vorhanden sein und der Wert von x sei 17.

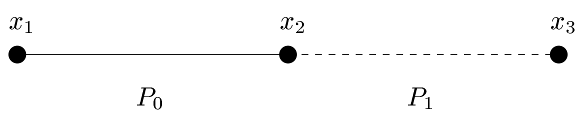
	P_0	P_1	P_2	Speicherart
Wert von $P_i \to x$	17	17	17	konsistent
Wert von $P_i \to x$	10	10	-3	inkonsistent
Wert von $P_i \to x$	17	0	0	inkonsistent

Die inkonsistente Speicherung eines Wertes ist nicht eindeutig bestimmt.

Ein Vektor beziehungsweise eine Matrix werden (in)konsistent gespeichert genannt, falls alle Komponenten (in)konsistent gespeichert sind.

Die iterative Lösung großer linearer System erfordert Matrix-Vektor-Operationen. Um diese Operationen korrekt ausführen zu können, müssen die Speicherarten der beteiligten Objekte zueinander passen. Um die Zusammenhänge zu illustrieren, betrachten wir das folgende Beispiel.

Beispiel 7.3 Gegeben sei folgende Situation:



Vektor	P_0-links	P_0-rechts	P_1-links	P_1-rechts	Speicherart
u = (-1,1,3)	-1	1	1	3	konsistent
v = (1,-2,1)	1	-2	-2	1	konsistent
w = (3,2,1)	3	1	1	1	inkonsistent
r = (-2,1,-1)	-2	-2	3	-1	inkonsistent

Multiplikation mit einem Skalar

Das Produkt eines (in)konsistent gespeicherten Vektors mit einem konsistenten Skalar ist ein (in)konsistent gespeicherter Vektor.

Vektor	P_0-links	P_0-rechts	P_1-links	P_1-rechts	Speicherart
$\alpha u = (-\alpha, \alpha, 3\alpha)$	$-\alpha$	α	α	3α	konsistent
$\alpha w = (3\alpha, 2\alpha, \alpha)$	3α	α	α	α	inkonsistent

Addition zweier Vektoren

Man darf nur Vektoren addieren, die vom gleichen Speichertyp sind ! Anderenfalls erhält man ein falsches Ergebnis. Die Summe zweier (in)konsistent gespeicherter Vektoren ist ein konsistent (in)gespeicherter Vektor.

Vektor	P_0-links	P_0-rechts	P_1-links	P_1-rechts	Speicherart
$u + v = (0, -1, 4)$	0	-1	-1	4	konsistent
$w + r = (1, 3, 0)$	1	-1	4	0	inkonsistent
$u + w = (2, 3, 4)$	2	2	2	4	undefiniert

Skalarprodukt zweier Vektoren

Zur Berechnung des Skalarproduktes zweier Vektoren x und y, muss einer der Vektoren konsistent und der andere inkonsistent gespeichert sein. Seinen beispielsweise $x \in \mathbb{R}^n$ konsistent und $y \in \mathbb{R}^n$ inkonsistent gespeichert. Dann gilt

$$\begin{aligned} x^T y &= \sum_{j=1}^{n} x_j y_j = \sum_{j=1}^{n} x_j \left(\sum_{i=0}^{p-1} (P_i \to y_j) \right) = \sum_{i=0}^{p-1} \left(\sum_{j=1}^{n} x_j (P_i \to y_j) \right) \\ &= \sum_{i=0}^{p-1} \left(\sum_{j=1}^{n} (P_i \to x_j)(P_i \to y_j) \right) . \end{aligned}$$

Die erste Summe läuft über alle Prozesse und die zweite Summe kann lokal vom Prozess i berechnet werden. Das Ergebnis ist zunächst inkonsistent gespeichert. Zur Berechnung des konsistent gespeicherten Wertes sind noch Kommunikationen erforderlich.

	P_0	P_1	$P_0 + P_1$	Wahrheitswert
$u^T w = 2$	$u_1 w_1 + u_2 w_2 = -2$	$u_2 w_2 + u_3 w_3 = 4$	2	richtig
$u^T v = 0$	$u_1 v_1 + u_2 v_2 = -3$	$u_2 v_2 + u_3 v_3 = 1$	-2	falsch
$w^T r = -5$	$w_1 r_1 + w_2 r_2 = -8$	$w_2 r_2 + w_3 r_3 = 2$	-6	falsch

Vor der Kommunikation ist der Wert inkonsistent gespeichert.

Matrix-Vektor-Multiplikation

Bei einer Matrix-Vektor-Multiplikation wird nichts anderes als eine Anzahl von Skalarprodukten zweier Vektoren berechnet. Somit muss entweder die Matrix inkonsistent und der Vektor konsistent gespeichert sein oder umgekehrt. Das Ergebnis dieser Operation ist ein inkonsistent gespeicherter Vektor.

Falls ein Objekt nicht auf die für eine Operation geeignete Weise gespeichert ist, muss seine Speicherart vor der Operation geändert werden.

Beispiel 7.4 Fortsetzung von Beispiel 7.3.
Den Vektor w konsistent machen. P_0 sendet seinen Wert von w_2 zu P_1 und P_1 seinen Wert zu P_0. Beide Prozesse addieren ihre Werte zu den eigenen Werten um den konsistenten Wert zu erhalten.

Den Vektor u inkonsistent machen. Eine Möglichkeit ist wie folgt. Alle Prozesse, die u_2 besitzen setzen diesen Wert zu Null, nur der Prozess mit der kleinsten Nummer tut dies nicht.

Beispiel 7.5 Norm eines Vektors. Der Vektor u sei auf p Prozessen verteilt und die Norm von u, $\|u\| = \sqrt{u^T u}$ soll berechnet werden. Die Operation $u^T u$ ist weder korrekt definiert wenn u konsistent gespeichert ist noch wenn u inkonsistent gespeichert ist. Sei u inkonsistent gespeichert. Dann muss man folgende Vorgehensweise zur Berechnung der Norm anwenden:

- kopiere u nach v (keine Kommunikation)
- mache u konsistent (Kommunikation)
- berechne inkonsistentes Ergebnis $u^T v$
- mache $u^T v$ konsistent (Kommunikation)
- berechne $\sqrt{u^T v}$

Ein Überblick über Speicherarten für wohldefinierte Operationen ist in Tabelle 7.1 gegeben.

Tabelle 7.1: Speicherarten für wohldefinierte Operationen

Operation	Vektor/Matrix 1	Vektor/Matrix 2	Ergebnis
Mult. mit konsistentem Skalar	konsistent	-	konsistent
Mult. mit konsistentem Skalar	inkonsistent	-	inkonsistent
Addition	konsistent	konsistent	konsistent
Addition	inkonsistent	inkonsistent	inkonsistent
Skalarprodukt	inkonsistent	konsistent	inkonsistent
Skalarprodukt	konsistent	inkonsistent	inkonsistent
Matrix-Vektor-Mult.	inkonsistent	konsistent	inkonsistent
Matrix-Vektor-Mult.	konsistent	inkonsistent	inkonsistent

7.2 Eigenschaften und Abspeicherung der Matrix A des Modellproblems aus Kapitel 6

Die Matrix A, welche bei der Assemblierung des Modellproblems generiert wird, besitzt folgende Eigenschaften:

- Die Anzahl der Spalten und Zeilen von A stimmt mit der Anzahl der Freiheitsgrade überein. Wir zählen auch alle Werte, die auf dem Rand von $(0, 1)$ liegen mit zu den Freiheitsgraden. Damit ist die Anzahl der Freiheitsgrade gleich der Anzahl aller Eckpunkte der gegebenen Triangulierung. Bei den Freiheitsgraden, von denen der Wert durch die Dirichlet-Randbedingung bereits bekannt ist, wird die entsprechende Zeile der Matrix durch eine Zeile ersetzt, die im Diagonalelement eine Eins und sonst Nullen hat, der Lösungvektor und der Vektor für die rechte Seite werden mit dem bekannten Wert belegt. Im Parallelen muss man dabei beachten, dass die Lösung konsistent gespeichert wird und die rechte Seite inkonsistent, siehe unten. Mit diesem Schritt wird natürlich die Symmetrie von A zerstört. Um diese wieder herzustellen, werden die bekannten Dirichletwerte gleich eingesetzt und das Ergebnis auf die rechte Seite gebracht. Damit steht auch in den Spalten, die diegleichen Indizes wie die Dirichletzeilen besitzen, in der Hauptdiagonale eine Eins und sonst Nullen. Die positive Definitheit von A wird dabei nicht zerstört (siehe Satz über positive Definitheit und Werte der Hauptunterdeterminanten). Eine Umsetzung dieses Verfahrens findet man auf Seite 68.
  Fazit: die Matrix A hat im allgemeinen eine hohe Dimension n.
- Die Matrix A ist schwach besetzt. Das heisst, der überwiegende Anteil der Einträge von A ist Null. Ein Eintrag a_{ij} kann höchstens ungleich Null sein, wenn die Basisfunktionen v_i^h und v_j^h einen gemeinsamen Träger besitzen, dessen Maß ungleich Null ist. Bei der regulären Triangulierung wie sie im Modellproblem verwendet wird, siehe Abbildung 6.1, können das für

gegebenes v_i^h höchstens sieben Basisfuktionen v_j^h sein (den Fall $i = j$ nicht vergessen). Damit ist die Anzahl der Nichtnulleinträge pro Zeile und pro Spalte in A höchstens 7, unabhängig von der Gesamtanzahl der Freiheitsgrade. Die Gesamtanzahl von Nichtnulleinträgen von A ist höchstens $7n$.

- Die Matrix A ist symmetrisch, das heisst $a_{ij} = a_{ji}$. Das folgt sofort aus der Berechnungsvorschrift (6.8) von a_{ij}.
- Die Matrix A ist positiv definit. Das heisst, für alle Vektoren $x \in \mathbb{R}^n$, ausser für den Nullvektor, gilt $x^T Ax > 0$. Bei symmetrischen Matrizen ist diese Eigenschaft äquivalent damit, dass alle Eigenwerte positiv sind.
- Da jede Gitterzelle der Triangulierung zu genau einem Prozess gehört und die Assemlierung von A als Summation über die Gitterzellen ausgeführt wird, siehe (6.8), ist die Matrix A nach der Assemblierung inkonsistent gespeichert. Dasselbe gilt für die rechte Seite b. Präziser, Einträge, die mit Unbekannten verbunden sind, die auf Prozessgrenzen liegen, sind, falls sie auf mehreren Prozessen vorhanden sind, inkonsistent gespeichert. Einträge, die mit anderen Unbekannten verbunden sind, sind nur auf einem Prozess vorhanden. Wegen der Wohldefiniertheit des Matrix-Vektor-Produktes ist es damit natürlich, dass Vektoren, die mit A multipliziert werden, zum Beispiel die Lösung x, konsistent gespeichert werden. Das Ergebnis ist dann ein inkonsistent gespeicherter Vektor, zum Beispiel die rechte Seite b. Das heisst, b ist auch schon auf passende Art gespeichert.

Die Eigenschaften der Symmetrie und der positiven Definitheit von A sind vorteilhaft für die numerische Lösung des linearen Gleichungssystems $Ax = b$ und für die Analysis der verwendeten numerischen Verfahren.

Aus Speicherplatzgründen wird man von A nur die Nichtnulleinträge speichern mit zugehörigen Informationen über die Zeilen- und Spaltenindizes. Eine oft genutzte Speichertechnik ist die sogenannte *compressed sparse row (CSR)*-Speichertechnik, siehe zum Beispiel Saad [Saa96]. In der CSR-Speichertechnik werden die Nichtnullelemente zeilenweise abgespeichert. Das ist günstig für die Durchführung von Matrix-Vektor-Produkten mit A. Man benötig einen `double`-Array `AA` für die Koeffizienten von A, einen `integer`-Array `JA` für die zugehörigen Spaltenindizes und einen `integer`-Array `IA`, der angibt, wo die Zeilen in `AA` und `JA` beginnen. Die Länge von `IA` ist $n + 1$ (das Ende von `AA` und `JA` muss auch angegeben werden), die Längen von `AA` und `JA` sind im Modellproblem höchstens $7n$. Damit ist der Speicherbedarf für große n sehr viel geringer als bei einer vollen Speicherung von A (dieser ist n^2).

Beispiel 7.6 *CSR-Speichertechnik.* Die Matrix

$$A = \begin{pmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{pmatrix}$$

wird wie folgt gespeichert (Numerierung beginnt bei 0):

AA	–	1	2	3	4	5	6	7	8	9	10	11	12
JA	–	0	3	0	1	3	0	2	3	4	2	3	4
IA	–	0	2	5	9	11	12						

7.3 Das gedämpfte Jacobi-Verfahren

7.3.1 Das serielle Verfahren

Das gedämpfte Jacobi-Verfahren (Carl Gustav Jacob Jacobi (1804 - 1851)) ist eines der einfachsten Verfahren zur iterativen Lösung von (7.1).

Algorithmus 7.7 Gedämpftes Jacobi-Verfahren. *Gegeben sind eine Anfangsnäherung x^0 und eine positive Zahl ω. Die Diagonale von A wird mit D bezeichnet ($D = \operatorname{diag}(A)$) und es wird vorausgesetzt, dass D invertierbar ist. Im gedämpften Jacobi-Verfahren wird die Iterierte x^{k+1} mittels*

$$x^{k+1} = x^k - \omega D^{-1}(Ax^k - b) \tag{7.2}$$

berechnet. Für $\omega = 1$ spricht man vom Jacobi-Verfahren.

Die Iteration (7.2) ist nichts weiter als eine Fixpunktiteration der Fixpunkt-Gleichung

$$x = x - \omega D^{-1}(Ax - b),$$

die man durch einfache Umformungen aus (7.1) erhält.

Falls A symmetrisch und positiv definit ist, so sind die Diagonaleinträge von A positiv und mithin ist D invertierbar. Das sieht man, indem man in der Bedingung für die positive Definitheit als Vektor x die kartesischen Einheitsvektoren e_i, $i = 1, \ldots, n$, wählt: $e_i^T A e_i > 0$ ist gleichbedeutend mit $a_{ii} > 0$.

Beispiel 7.8 Zu lösen sei

$$\begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ -5 \\ 5 \end{pmatrix} \quad \Longrightarrow \quad \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \\ 2 \end{pmatrix}.$$

Wir verwenden $\omega = 1$ und starten mit $x^0 = (0,0,0)^T$. Man erhält

$$\begin{aligned}
x^1 &= \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} - \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix}^{-1} \left[\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} - \begin{pmatrix} 3 \\ -5 \\ 5 \end{pmatrix} \right] \\
&= \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} - \begin{pmatrix} -3/2 \\ 5/2 \\ -5/2 \end{pmatrix} = \begin{pmatrix} 3/2 \\ -5/2 \\ 5/2 \end{pmatrix} \\
x^2 &= \begin{pmatrix} 3/2 \\ -5/2 \\ 5/2 \end{pmatrix} - \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix}^{-1} \left[\begin{pmatrix} 11/2 \\ -9 \\ 15/2 \end{pmatrix} - \begin{pmatrix} 3 \\ -5 \\ 5 \end{pmatrix} \right] = \begin{pmatrix} 0.25 \\ -0.5 \\ 1.25 \end{pmatrix}
\end{aligned}$$

und so weiter.

Man kann folgende allgemeine Konvergenzaussage für das gedämpfte Jacobi-Verfahren beweisen.

Satz 7.9 *Sei A eine positiv definite Matrix. Dann konvergiert das gedämpfte Jacobi-Verfahren für jeden Startwert x^0, falls die Matrix*

$$\frac{2}{\omega} D - A \tag{7.3}$$

positiv definit ist.

Beweis: Siehe, zum Beispiel, [Fro90, S. 148 f.]. ∎

Da die Matrix $\frac{2}{\omega}D - A$ symmetrisch ist, besagt die Bedingung (7.3), dass alle Eigenwerte von $\frac{2}{\omega}D - A$ positiv sein müssen. Da die Eigenwerte von D positiv sind (D ist eine Diagonalmatrix mit positiven Hauptdiagonaleinträgen), kann das immer erreicht werden, indem man ω hinreichend klein wählt. Ein kleiner Dämpfungsparameter ω führt im allgemeinen jedoch auch zu einer sehr langsamen Konvergenz des Verfahrens.

Eine serielle Implementierung der Iteration (7.2) sieht wie folgt aus.

Algorithmus 7.10 *Serielles gedämpftes Jacobi-Verfahren.* Gegeben sind A, x, b, ω, $\varepsilon > 0$ und $maxit > 0$. D sei die Diagonale von A.

$\texttt{for}\ (\mathtt{k}=0;\mathtt{k}<\mathtt{maxit};\mathtt{k}++)$
  $\mathtt{d}=\mathtt{Ax};$ `// d ist der Defekt oder Residuenvektor`
  $\mathtt{d}=\mathtt{d}-\mathtt{b};$
  $\texttt{if}\ (\mathtt{sqrt}(\mathtt{d}^{\mathtt{T}}\mathtt{d})<\varepsilon)$ `// Residuum klein genug`
    `break;`
  $\mathtt{d}=\omega\mathtt{D}^{-1}\mathtt{d};$ `//` $\mathtt{d}_{\mathtt{i}}=\omega\mathtt{d}_{\mathtt{i}}/\mathtt{a}_{\mathtt{ii}}$
  $\mathtt{x}=\mathtt{x}-\mathtt{d};$
`endfor`

7.3.2 Das parallele Verfahren

Für eine parallele Implementierung von Algorithmus 7.10 muss man zusätzlich noch die Speicherungsarten der beteiligten Matrix und Vektoren in Betracht ziehen und kontrollieren, ob alle Operationen wohldefiniert sind.

Algorithmus 7.11 *Paralleles gedämpftes Jacobi-Verfahren.* Im folgenden werden konsistent gespeicherte Größen dick gedruckt und inkonsistente dünn. Gegeben sind $A, \mathbf{x}, b, \boldsymbol{\omega}$, $\varepsilon > 0$ und $\mathbf{maxit} > 0$. $\mathbf{D}$ sei die Diagonale von A. Es stellt sich heraus, dass man im Verfahren die Diagonale von A als konsistenten Vektor benötigt.

$\mathtt{D}=\mathtt{diag}(\mathtt{A});$
$\mathbf{D}=\mathtt{mache\ D\ konsistent};$ `// Kommunikation`
$\texttt{for}\ (\mathbf{k}=\mathbf{0};\mathbf{k}<\mathbf{maxit};\mathbf{k}++)$
  $\mathtt{d}=\mathtt{Ax};$
  $\mathtt{d}=\mathtt{d}-\mathtt{b};$
  $\mathtt{r}=\mathtt{d};$
  $\mathbf{d}=\mathtt{mache\ d\ konsistent};$ `// Kommunikation`
  $\texttt{if}\ (\mathtt{sqrt}(\mathbf{d}^{\mathtt{T}}\mathtt{r})<\varepsilon)$ `// Kommunikation`
    `break;`
  $\mathbf{d}=\boldsymbol{\omega}\mathbf{D}^{-1}\mathbf{d};$
  $\mathbf{x}=\mathbf{x}-\mathbf{d};$
`endfor`

Beispiel 7.12 *Verhalten des parallelen gedämpften Jacobi-Verfahrens im Modellproblem.* Wir betrachten das Modellproblem (6.1) in welchem die rechte Seite und die Randbedingungen so gewählt werden, dass

$$u(x,y) = x + y$$

die Lösung ist. Das Einheitsquadrat $(0,1)^2$ wird mit n_x^2 Quadraten trianguliert, die dann jeweils noch in Dreiecke geteilt werden. Das Modellproblem wird mit dem Jacobi-Verfahren ($\omega = 1$) gelöst, die Anfangsnäherung ist $x^0 = 0$ und die Iteration wird abgebrochen, wenn die Euklidische Norm des Defektes kleiner als $\varepsilon = 1e-6$ ist. Man erhält folgende Rechenzeiten auf dem HP-Superdome:

		Prozessoren	1	4 (2×2)	9 (3×3)	16 (4×4)	36 (6×6)
$n_x = n_y$	Freiheitsgr.	Iterationen					
6	49	103	0.002	0.003	0.004	-	0.006
12	169	407	0.006	0.013	0.015	0.015	0.018
24	625	1561	0.072	0.062	0.068	0.066	0.072
48	2401	5933	0.96	0.44	0.35	0.32	0.33
96	9409	22451	14.81	4.54	2.71	1.97	1.63
192	37249	84638	1191.2	64.9	30.3	19.0	12.2

Man kann folgende Beobachtungen treffen:

- Erhöht sich die Anzahl der Unbekannten um den Faktor 4, so erhöht sich die Anzahl der benötigten Iterationen auch um den Faktor 4 (diese sind zudem teurer). Das ist eine schlechte Eigenschaft des Verfahrens. Man möchte ein Verfahren haben, bei dem die Anzahl der

benötigten Iterationen nach oben beschränkt werden kann für eine beliebige Dimension des Gleichungssystems und der Aufwand pro Iteration proportional zur Anzahl der Unbekannten ist (zum Beispiel wie bei Mehrgitterverfahren).
- Für kleine Systeme lohnt sich die Parallelisierung nicht.
- Für die extrem lange Rechenzeit für $n_x = 192$ auf einem Prozessor scheinen Speicherzugriffzeiten verantwortlich zu sein. Das System passt nicht mehr in den Cache.
- Der Scaleup des Verfahrens ist schlecht, da man für höherdimensionale Systeme mehr Iterationen braucht. Der Scaleup pro Iterationen ist dagegen ganz gut. Man erhält zum Beispiel:

$$\begin{aligned}\text{Scaleup/Iteration}(9409, 4) &= \frac{\text{Zeit}(2401, \text{ ein Prozess }) \times \text{ 22451 Iterationen}}{\text{Zeit}(9409, \text{ 4 Prozesse}) \times \text{ 5933 Iterationen}} \approx 0.8 \\ \text{Scaleup/Iteration}(37249, 16) &= \frac{\text{Zeit}(2401, \text{ ein Prozess }) \times \text{ 84638 Iterationen}}{\text{Zeit}(37249, \text{ 16 Prozesse}) \times \text{ 5933 Iterationen}} \approx 0.72.\end{aligned}$$

Insgesamt ist das gedämpfte Jacobi-Verfahren ein Verfahren, welches sich einfach parallelisieren lässt, dessen parallele Effizienz hoch ist (Scaleup/Iteration), dessen mathematische Eigenschaften (Konvergenzverhalten) jedoch ziemlich schlecht sind.

7.4 Das Gauß-Seidel-Verfahren und das SOR-Verfahren

7.4.1 Das serielle Verfahren

Das SOR-Verfahren (David M. Young (geb. 1923), veröffentlicht 1950 in seiner Dissertation, 1954 als Zeitschriftenartikel) hat die folgenden Gestalt (SOR - successive overrelaxation):

Algorithmus 7.13 SOR-Verfahren. *Gegeben sind eine Anfangsnäherung x^0 und eine positive Zahl ω. Im SOR-Verfahren wird die Iterierte x^{k+1} wie folgt berechnet:*

$$x_i^{k+1} = x_i^k - \frac{\omega}{a_{ii}} \left(\sum_{j=1}^{i-1} a_{ij} x_j^{k+1} + \sum_{j=i}^{n} a_{ij} x_j^k \right), \quad i = 1, \ldots, n, \quad k = 0, 1, 2, \ldots. \tag{7.4}$$

Für $\omega = 1$, wird das Verfahren (7.4) auch Gauß-Seidel-Verfahren genannt (Carl Friedrich Gauß (1777 - 1855), Philipp Ludwig von Seidel (1821 - 1896)).

Die Iteration (7.4) kann auch in Matrix-Vektor-Form geschrieben werden. Diese ist jedoch viel komplizierter, als beispielsweise die Matrix-Vektor-Form des gedämpften Jacobi-Verfahrens. Wir benötigen die Matrix-Vektor-Form im folgenden nicht. Man nutzt (7.4) zur Implementierung des SOR-Verfahrens.

Der Unterschied zum gedämpften Jacobi-Verfahren wird klar, wenn man das gedämpfte Jacobi-Verfahren in einer Komponentenschreibweise wie (7.4) schreibt:

$$x_i^{k+1} = x_i^k - \frac{\omega}{a_{ii}} \left(\sum_{j=1}^{i-1} a_{ij} x_j^k + \sum_{j=i}^{n} a_{ij} x_j^k \right), \quad i = 1, \ldots, n, \quad k = 0, 1, 2, \ldots.$$

Während man beim gedämpften Jacobi-Verfahren nur die alte Iterierte x^k zur Berechnung der neuen Iterierten x^{k+1} nutzt, verwendet man beim SOR-Verfahren für die Berechnung der i-ten Komponente von x^{k+1} die bereits berechneten erste bis $(i-1)$-te Komponente von x^{k+1}.

Beispiel 7.14 Zu lösen sei

$$\begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ -5 \\ 5 \end{pmatrix} \quad \Longrightarrow \quad \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \\ 2 \end{pmatrix}.$$

Wir verwenden $\omega = 1$ und starten mit $x^0 = (0,0,0)^T$. Man erhält

$$
\begin{aligned}
x_1^1 &= x_1^0 - \frac{1}{a_{11}}\left(\sum_{j=1}^{3} a_{1j}x_j^0 - b_1\right) = 0 - \frac{1}{2}(0-3) = \frac{3}{2} \\
x_2^1 &= x_2^0 - \frac{1}{a_{22}}\left(a_{21}x_1^1 + a_{22}x_2^0 + a_{23}x_3^0 - b_2\right) = 0 - \frac{1}{2}\left(-\frac{3}{2}+5\right) = -\frac{7}{4} \\
x_3^1 &= x_3^0 - \frac{1}{a_{33}}\left(a_{31}x_1^1 + a_{32}x_2^1 + a_{33}x_3^0 - b_3\right) = 0 - \frac{1}{2}\left(\frac{7}{4}-5\right) = \frac{13}{8}.
\end{aligned}
$$

Es gilt folgende Konvergenzaussage:

Satz 7.15 (Ostrowski (1954)). *Sei A symmetrisch und positiv definit. Dann konvergiert das SOR-Verfahren für jeden Startwert x^0 genau dann wenn $\omega \in (0,2)$.*

Beweis: Siehe [Fro90, S. 151]. ∎

Einige andere Eigenschaften des SOR-Verfahrens sind wie folgt:

- Das SOR-Verfahren hängt von der Numerierung der Unbekannten ab. Das ist ein Unterschied zum Jacobi-Verfahren, das unabhängig von der Numerierung der Unbekannten ist. Ändert man die Numerierung der Unbekannten im SOR-Verfahren, so ändert man im allgemeinen das Verfahren.

  Beispiel 7.16 *Fortsetzung von Beispiel 7.14.*

$$
\begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix}
\begin{pmatrix} x_3 \\ x_2 \\ x_1 \end{pmatrix} = \begin{pmatrix} 5 \\ -5 \\ 3 \end{pmatrix}
\quad \Longrightarrow \quad
\begin{pmatrix} x_3 \\ x_2 \\ x_1 \end{pmatrix} = \begin{pmatrix} 2 \\ -1 \\ 1 \end{pmatrix},
$$

  $\omega = 1$, $x^0 = (0,0,0)^T$. Nach dem ersten SOR-Schritt erhält man

$$
\begin{pmatrix} x_3^1 \\ x_2^1 \\ x_1^1 \end{pmatrix} = \begin{pmatrix} 5/2 \\ -5/4 \\ 7/8 \end{pmatrix}.
$$

  Es gibt Klassen von Gleichungen, zum Beispiel Konvektions-Diffusions-Gleichungen, bei denen das SOR-Verfahren ein sehr gutes Verfahren ist, wenn die Unbekannten geeignet durchnumeriert sind und ein sehr schlechtes Verfahren ist, wenn die Numerierung der Unbekannten nicht geeignet ist.
- Man kann für symmetrisch positiv definite Matrizen, bei denen zusätzlich $a_{ij} \le 0$, für $i \ne j$ gilt, zeigen, dass das Jacobi-Verfahren (mit $\omega = 1$) nicht schneller konvergiert als das Gauß-Seidel-Verfahren, siehe [Fro90, S. 152].
- Die Konvergenzrate des SOR-Verfahrens hängt von der Wahl von ω ab. Es kann gezeigt werden, dass für eine gewisse Klasse von Matrizen, zu denen auch die Matrizen gehören die wir bei der Assemblierung des Modellproblems erhalten, ein optimaler Wert ω_{opt} existiert und dass $\omega_{opt} \in (1,2)$. Daher kommt auch die Bezeichnung *over*relaxation. Die Berechnung von ω_{opt} ist in der Praxis jedoch schwierig. Für einige Modellprobleme ist ω_{opt} jedoch bekannt, zum Beispiel für das von uns betrachtete mit einer finiten Differenzendiscretisierung:

$$
\omega_{opt} = \frac{2}{1+\sin(\pi h)},
$$

  wobei h die Kathedenlänge der Dreiecke ist $h = 1/(n_x n_{x,loc}) = 1/(n_y n_{y,loc})$. Für feine Gitter ist h klein, also ist ω_{opt} nahe bei 2.

Eine serielle Implementierung der Iteration (7.4) sieht wie folgt aus.

Algorithmus 7.17 *Serielles SOR-Verfahren.* Gegeben sind A, x, b, ω, $\varepsilon > 0$ und $maxit > 0$.

1. $\mathtt{for}\ (\mathtt{k = 0; k < maxit; k++})$

```
 2.     d = Ax;    // d ist der Defekt oder Residuenvektor
 3.     d = d − b;
 4.     if (sqrt(dᵀd) < ε)     // Residuum klein genug
 5.       break;
 6.     for (i = 0; i < n; i + +)
 7.       sum = 0;
 8.       for (j = 0; j < n; j + +)
 9.         sum = sum + a_ij x_j;
10.       endfor
11.       x_i = x_i − ω(sum − b_i)/a_ii;
12.     endfor
13. endfor
```

Der einzige Unterschied zum Jacobi-Verfahren ist der Korrekturschritt (Zeilen 6 - 12).

7.4.2 Parallelisierte Varianten

Die Parallelisierung des SOR-Verfahrens ist, im Gegensatz zum Jacobi-Verfahren, kompliziert. Das hängt damit zusammen, dass beim Korrekturschritt der neue Wert von x_i von den neuen Werten von $x_1, \ldots, x_{i-1}$ abhängt. Das heißt, x_i kann nicht korrigiert werden, bevor nicht $x_1, \ldots, x_{i-1}$ korrigiert worden sind. Diese globale Abhängigkeit des neuen Wertes x_i von $x_1, \ldots, x_{i-1}$, die die Parallelisierung erschwert, ist jedoch auch für die besseren mathematischen Eigenschaften des SOR-Verfahrens im Vergleich zum Jacobi-Verfahren verantwortlich.

SOR-Verfahren mit Rot-Schwarz-Numerierung

Das SOR-Verfahren lässt sich nur in Spezialfällen effizient parallelisieren. Einer dieser Fälle ist gegeben, falls eine Einteilung der Unbekannten in eine Klasse *Rot* und eine Klasse *Schwarz* möglich ist, so daß jede rote Unbekannte i nur von sich selbst und von schwarzen Unbekannten j direkt abhängt, d.h. $a_{ij} \neq 0$ oder $a_{ji} \neq 0$, und jede schwarze Unbekannte i nur von sich selbst und von roten Unbekannten direkt abhängt. Alle anderen Matrixeinträge verschwinden. Eine solche Rot-Schwarz-Numerierung ist nur in Spezialfällen möglich. In dem von uns betrachteten Modellproblem ist sie beispielsweise nicht möglich.

Angenommen, eine Rot-Schwarz-Numerierung wäre möglich. Dann werden die Unbekannten so geordnet, dass zuerst die roten und dann die schwarzen kommen. Das Gleichungssystem (7.1) erhält eine spezielle Gestalt (A symmetrisch):

$$Ax = \begin{pmatrix} D_r & C \\ C^T & D_s \end{pmatrix} \begin{pmatrix} x_r \\ x_s \end{pmatrix} = \begin{pmatrix} b_r \\ b_s \end{pmatrix}.$$

Die Matrizen D_r und D_s sind Diagonalmatrizen.

Der Korrekturschritt des SOR-Verfahrens kann nun in zwei Teilschritte zerlegt werden und er hat in Matrix-Vektor-Notation folgende Gestalt:

Schritt 1: Korrektur der roten Unbekannten

$$x_r^{k+1} = (1-\omega)x_r^k - \omega D_r^{-1}(Cx_s^k - b_r) \tag{7.5}$$

Schritt 2: Korrektur der schwarzen Unbekannten

$$x_s^{k+1} = (1-\omega)x_s^k - \omega D_s^{-1}(C^T x_r^{k+1} - b_s) \tag{7.6}$$

Beide Schritte kann man als Jacobi-Schritte für die jeweiligen Unbekannten ansehen. Darin liegt auch das Prinzip dieser Herangehensweise: teile die Unbekannten in Klassen ein, so dass die Vertreter innerhalb jeder Klasse voneinander unabhängig sind. Dann ist ein SOR-Schritt innerhalb einer jeden Klasse nichts anderes als ein Jacobi-Schritt. Die Klassen werden hintereinander abgearbeitet.

Eine Parallelisierung dieses Verfahrens kann wie folgt vorgenommen werden:

1. Jeder Prozess besorgt sich die Werte x_s^k, von denen seine roten Unbekannten abhängen und die nicht auf ihm gespeichert sind. Das sind im allgemeinen wenige Werte. Kommunikation ist erforderlich.
2. Alle Prozesse führen in parallel die Korrektur der roten Unbekannten (7.5) durch.
3. Jeder Prozess besorgt sich die Werte x_r^{k+1}, von denen seine schwarzen Unbekannten abhängen und die nicht auf ihm gespeichert sind. Kommunikation ist erforderlich.
4. Alle Prozesse führen in parallel die Korrektur der schwarzen Unbekannten (7.6) durch.

Falls eine Rot-Schwarz-Numerierung nicht möglich ist, kann man versuchen, eine Numerierung mit mehr als zwei Farben zu finden (so wenig wie möglich), die die Eigenschaften der Rot-Schwarz-Numerierung besitzt.

Block-Jacobi-Verfahren mit inneren SOR-Iterationen (BJac-SOR)

Eine andere Parallelisierungsstrategie besteht darin, das SOR-Verfahren so zu modifizieren, dass man ein Verfahren erhält, welches man einfacher parallelisieren kann. Die Grundidee einer solchen Modifizierung besteht darin, dass man auf Informationen, die die effiziente Parallelisierung verhindern, verzichtet. Das heißt beim SOR-Verfahren, dass jeder Prozess auf Werte, die er zwar beim Verfahren braucht, die jedoch nicht auf ihm vorhanden sind, verzichtet. Jeder Prozess führt den Korrekturschritt nur mit den Unbekannten aus, die er besitzt. Diese Unbekannten bilden einen Block und bezüglich dieser Blöcke führt man ein Jacobi-Verfahren durch. Das entstehende Block-Jacobi-Verfahren wird im allgemeinen schlechtere mathematische Eigenschaften als das SOR-Verfahren besitzen, da man die globale Abhängigkeit der Unbekannten beseitigt. Die Eigenschaften werden desto schlechter sein, je mehr Blöcke man hat (bei konstantem n). Andererseits wird das Block-Jacobi-Verfahren bessere mathematische Eigenschaften als das Jacobi-Verfahren haben, da man innerhalb der Blöcke ein besseres Verfahren, nämlich das SOR-Verfahren, verwendet.

Bezüglich der berechneten Korrektur

$$x_i - \frac{\omega}{a_{ii}}\left(\sum_{j=1}^{n} a_{ij}x_j - b_i\right) \tag{7.7}$$

muss man zwischen zwei Klassen von Unbekannten unterscheiden. Die Werte x_j, $j = 1, \ldots, n$, sind konsistent gespeichert und die Werte a_{ii}, $i = 1, \ldots, n$, muss man vor der Iteration auf konsistent Art und Weise speichern.

1. Die Werte a_{ij}, $j = 1, \ldots, n$ und b_i liegen konsistent gespeichert vor. Dann ist der berechnete Wert konsistent und kann sofort von x_i subtrahiert werden, das heisst, x_i wird sofort korrigiert. Bei unserem Modellproblem hat man diesen Fall bei allen Freiheitsgraden, die nicht auf den Prozessgrenzen liegen. Für diese Freiheitsgrade berechnen sich a_{ij} und b_i aus Integralen über die Dreiecke, von denen der Freiheitsgrad ein Eckpunkt ist. Diese Dreiecke liegen beim Modellproblem alle auf demselben Prozess.
2. Die Werte a_{ij}, $j = 1, \ldots, n$ und b_i liegen inkonsistent gespeichert vor. Dann ist der zweite Term in (7.7) ein inkonsistent gespeicherter Wert, der nicht von x_i subtrahiert werden darf. Dieser Wert ist zunächst auf einer Hilfsvariablen zu speichern. Nachdem der Prozess den Korrekturschritt ausgeführt hat, wird diese Hilfsvariable konsistent gemacht und erst dann von x_i subtrahiert.
   Beim Modellproblem gehören zu dieser Klasse alle Unbekannten, die auf den Prozessgrenzen liegen.

Das parallelisierte Block-Jacobi-Verfahren mit inneren SOR-Iterationen hat folgenden Gestalt:

Algorithmus 7.18 *Parallelisierte Block-Jacobi-Verfahren mit inneren SOR-Iterationen.* Im folgenden werden konsistent gespeicherte Größen dick gedruckt und inkonsistente dünn. Gegeben sind $A, \mathbf{x}, b, \boldsymbol{\omega}$, $\varepsilon > 0$ und **maxit** > 0. **D** sei die Diagonale von A.

1. `D` $=$ `diag(A);`
2. $\mathbf{D}$ $=$ `mache D konsistent;    // Kommunikation`

```
belege den Vektor s mit Nullen
for (k = 0; k < maxit; k++)
   d = Ax;
   d = d − b;
   r = d;
   d = mache d konsistent;    // Kommunikation
   if (sqrt(d^T r) < ε)    // Kommunikation
     break;
   for (i = 0; i < n; i++)
     sum = 0;
     for (j = 0; j < n; j++)
       sum = sum + a_ij x_j;
     endfor
     s_i = ω(sum − b_i)
     if (innerer Freiheitsgrad)    // Wert von s_i ist nur auf einem Prozess,
                                      also konsistent
       x_i = x_i − s_i
       s_i = 0
     endif
   endfor
endfor
s = mache s konsistent;    // Kommunikation
x = x − s    // s[i] für innere Freiheitsgrade 0
```

Beispiel 7.19 *Verhalten des parallelen Block-Jacobi-Verfahrens mit inneren SOR-Iterationen (BJac-SOR) im Modellproblem.* Die Konfiguration des Beispiels ist analog zu Beispiel 7.12. Die Unbekannten, die sich auf einem Prozessor befinden, bilden einen Block, das heisst, je mehr Prozessoren verwendet werden, desto mehr Blöcke gibt es im Verfahren und desto mehr ist das ursprüngliche SOR-Verfahren modifiziert. Für $n_x = n_y = 192$, dass heisst 37249 Freiheitsgrade, erhält man folgende Iterationsanzahlen und Rechenzeiten auf dem HP-Superdome:

Prozesse (Prozesor-Array)	ω	Anzahl der Iterationen	Zeit
1	1.0	42349	1352.1
1	1.25	25419	811.8
1	1.5	14131	443.7
1	1.9	2223	66.5
4 (2×2)	1.0	43226	107.4
4 (2×2)	1.25	26311	65.1
4 (2×2)	1.5	divergent	
9 (3×3)	1.0	43679	49.5
9 (3×3)	1.25	26768	30.3
16 (4×4)	1.0	44122	29.4
16 (4×4)	1.25	27216	18.2
36 (6×6)	1.0	44997	16.2
36 (6×6)	1.25	28098	10.1

Man kann folgende Beobachtungen treffen:

- Das SOR-Verfahren auf einem Prozessor benötigt für alle gewählten Relaxationsparameter ω weniger Iterationen zur Konvergenz als das Jacobi-Verfahren, vergleiche mit Beispiel 7.12. Der Einfluss von ω auf die Konvergenzgeschwindigkeit ist deutlich zu erkennen. Für $\omega = 1.9$ berechnet das SOR-Verfahren die Lösung etwa 18-mal schneller als das Jacobi-Verfahren. Die besseren mathematischen Eigenschaften des SOR-Verfahrens machen sich bemerkbar.
- Für $n_x = n_y = 96$ benötigt das SOR-Verfahren mit $\omega = 1.9$ bis zum Erfülltsein des Abbruchkriteriums 564 Iterationen. Damit sieht man, dass sich beim SOR-Verfahren die Anzahl

der Iterationen vervierfacht wenn sich die Anzahl der Unbekannten vervierfacht. In dieser Beziehung ist das SOR-Verfahren nicht besser als das Jacobi-Verfahren.

- Beim BJac-SOR erkennt man, dass sich die Anzahl der benötigten Iterationen leicht erhöht, wenn sich die Anzahl der Blöcke erhöht. Das spiegelt die etwas schlechteren mathematischen Eigenschaften des Verfahrens bei einer Erhöhung der Blockanzahl wider. Durch die gute Parallelisierbarkeit des Verfahrens hat man jedoch unter dem Strich eine Verringerung der Rechenzeiten bei einer Erhöhung der Prozessoranzahl. Für den Speedup sollte man nicht die Rechenzeit auf einem Prozessor zu Grunde legen, da diese wahrscheinlich von Speicherzugriffszeiten dominiert ist. Nimmt man stattdessen die Zeiten für 4 Prozessoren als Basis erhält man ($\omega = 1.25$)

$$\begin{aligned}
\text{Speedup}(4,16) &= \frac{\text{Zeit auf 4 Prozessoren} \times 4 \text{ Prozessoren}}{\text{Zeit auf 16 Prozessoren} \times 16 \text{ Prozessoren}} \approx 0.89 \\
\text{Speedup}(4,36) &= \frac{\text{Zeit auf 4 Prozessoren} \times 4 \text{ Prozessoren}}{\text{Zeit auf 36 Prozessoren} \times 36 \text{ Prozessoren}} \approx 0.72.
\end{aligned}$$

  Diese Werte sind recht gut.
- Die schlechteren mathematischen Eigenschaften des BJac-SOR zeigen sich auch daran, dass das Verfahren für Relaxationsparameter ω, für die das SOR-Verfahren konvergiert, divergiert.

Insgesamt hat man durch die Modifizierung des SOR-Verfahrens ein brauchbares paralleles Verfahren erhalten.

7.4.3 Das symmetrische SOR-Verfahren (SSOR-Verfahren)

Das SSOR-Verfahren (symmetric SOR) ist eine symmetrische Variante des SOR-Verfahrens. Der Korrekturschritt des SSOR-Verfahrens setzt sich aus zwei Teilschritten zusammen. Der erste Teilschritt entspricht dem Korrekturschritt des SOR-Verfahrens. Im zweiten Teilschritt wird ebenfalls eine Korrektur wie beim SOR-Verfahren berechnet, jedoch werden die Unbekannten in umgekehrter Reihenfolge abgearbeitet.

Algorithmus 7.20 *Serielles SSOR-Verfahren.* Gegeben sind A, x, b, ω, $\varepsilon > 0$ und $maxit > 0$.

```
 1. for (k = 0; k < maxit; k++)
 2.    d = Ax;    // d ist der Defekt oder Residuenvektor
 3.    d = d - b;
 4.    if (sqrt(d^T d) < ε)     // Residuum klein genug
 5.      break;
 6.    for (i = 0; i < n; i++)
 7.      sum = 0;
 8.      for (j = 0; j < n; j++)
 9.        sum = sum + a_ij x_j;
10.      endfor
11.      x_i = x_i - ω(sum - b_i)/a_ii;
12.    endfor
13.    for (i = n - 1; i ≥ 0; i--)
14.      sum = 0;
15.      for (j = 0; j < n; j++)
16.        sum = sum + a_ij x_j;
17.      endfor
18.      x_i = x_i - ω(sum - b_i)/a_ii;
19.    endfor
20. endfor
```

Im Vergleich zum SOR-Verfahren hat man zusätzlich den zweiten Teil des Korrekturschrittes (Zeilen 13 - 19). Der Aufwand für einen SSOR-Schritt scheint zunächst doppelt so hoch wie für einen SOR-Schritt. Durch eine geschickte Implementierung kann man jedoch erreichen, dass der Aufwand für einen SSOR-Schritt im wesentlichen genauso hoch wie für einen SOR-Schritt ist, siehe [Hac91, S. 120]. Mathematische Eigenschaften des SSOR-Verfahrens findet man ebenfalls in [Hac91].

Die Parallelisierungsstrategien beim SSOR-Verfahren sind analog wie beim SOR-Verfahren.

7.5 Das vorkonditionierte Verfahren der konjugierten Gradienten

Das Verfahren heisst im Englischen *preconditioned conjugate gradient method* und wird mit *pcg* abgekürzt.

Wir wollen die Gleichung (7.1) lösen, wobei A eine symmetrische und positiv definite Matrix ist. Eine äquivalente Aufgabenstellung ist

$$\text{löse} \qquad \min_{x\in\mathbb{R}^n} \left\{ \frac{1}{2}(Ax,x) - (b,x) \right\}, \tag{7.8}$$

wobei $(\cdot,\cdot)$ das Skalarprodukt in $\mathbb{R}^n$ bezeichnet. Man kann zeigen, dass die eindeutige Lösung von (7.1) das eindeutige Minimum von (7.8) ist. Das folgt aus der notwendigen Bedingung für ein Minimum von (7.8):

$$\mathbf{0} = \nabla \left\{ \frac{1}{2}(Ax,x) - (b,x) \right\} =: \nabla f(x) = Ax - b =: -r. \tag{7.9}$$

Der Vektor r wird Residuum genannt und $f(x)$ ist eine skalare Funktion, die minimiert werden muss.

Die Standardherangehensweise zur Lösung solch eines Minimierungsproblems ist die Nutzung einer iterativen Methode, die in jedem Iterationsschritt nur eindimensionale Minimierungsprobleme löst:

Schritt 1: wähle eine Anfangsnäherung $x^0 \in \mathbb{R}^n$, setze $k = 0$.

Schritt 2: wähle eine Richtung $d^k \in \mathbb{R}^n$, löse das eindimensionale Minimierungsproblem

$$\tau^k = \min_{\tau\in\mathbb{R}} f(x^k + \tau d^k).$$

Schritt 3: setze $x^{k+1} = x^k + \tau^k k d^k$.

Schritt 4: $k := k+1$, gehe zu Schritt 2.

Die Lösung des eindimensionalen Minimierungsproblems in Schritt 3 lautet

$$\tau^k = \frac{(d^k, r^k)}{(d^k, Ad^k)}, \quad r^k = b - Ax^k.$$

Das folgt unmittelbar aus der notwendigen Bedingung für ein Minimum

$$\begin{aligned} 0 &= \frac{d}{d\tau} f(x^k + \tau d^k) = \frac{d}{d\tau}\left\{ \frac{1}{2}(A(x^k + \tau d^k), x^k + \tau d^k) - (b, x^k + \tau d^k) \right\} \\ &= \frac{1}{2}(Ax^k, d^k) + \frac{1}{2}(Ad^k, x^k) + \tau(Ad^k, d^k) - (b, d^k) \\ &= \frac{1}{2}(Ax^k, d^k) + \frac{1}{2}(Ax^k, d^k) + \tau(Ad^k, d^k) - (b, d^k). \end{aligned}$$

7.5.1 Die Methode des steilsten Abstiegs, das Gradientenverfahren

Diese Methode nutzt als Minimierungsrichtung die Richtung des lokal stärksten Abstieges von $f(x^k)$, das heisst, die Richtung in welche der Wert von $f(x^k)$ sich am stärksten verringert. Das ist die Richtung des negativen Gradienten:

$$d^k = -\nabla f(x^k) = r^k.$$

Die letzte Gleichheit folgt aus (7.9).

Die Konvergenz dieses Verfahrens wird in der von A induzierten Norm gemessen:

$$\|x\|_A = (Ax, x)^{1/2}.$$

Das ist eine Norm, da A symmetrisch und positiv definit ist. Sei $\hat{x}$ die Lösung von (7.1) und sei $p(\varepsilon)$, $\varepsilon > 0$, die kleinste natürliche Zahl k mit

$$\|x^k - \hat{x}\|_A \leq \varepsilon \|x^0 - \hat{x}\|_A \quad \forall\, x^0 \in \mathbb{R}^n,$$

das heisst, $p(\varepsilon)$ ist die Anzahl der Iterationen um den Fehler in $\|\cdot\|_A$ um den Faktor ε zu verringern. Man kann zeigen, siehe zum Beispiel [AB84], dass

$$p(\varepsilon) \leq \frac{1}{2}\mathrm{cond}_2(A) \ln\left(\frac{1}{\varepsilon}\right) + 1.$$

Hierbei ist $\mathrm{cond}_2(A)$ die Spektral-Konditionszahl von A :

$$\mathrm{cond}_2(A) = \|A\|_2 \|A^{-1}\|_2 = \sqrt{\lambda_{max}(A^T A)}\sqrt{\lambda_{max}(A^{-T} A^{-1})} \geq 1.$$

Für symmetrisch und positiv definite Matrizen gilt

$$\mathrm{cond}_2(A) = \frac{\lambda_{max}(A)}{\lambda_{min}(A)}.$$

Im allgemeinen hat man also beim Gradientenverfahren $p(\varepsilon) \sim \mathrm{cond}_2(A)$, das heisst für schlecht konditionierte Matrizen ist die Konvergenz langsam. Der Ausweg besteht darin, bessere Minimierungsrichtungen zu verwenden.

7.5.2 Die Methode der konjugierten Gradienten (CG)

Dieses Verfahren ist eines der besten iterativen Lösungsverfahren für die Lösung linearer Systeme mit symmetrisch und positiv definiter Matrix. Es wurde von Hestenes und Stiefel im Jahre 1952 veröffentlicht [HS52].

Die Konstruktion des CG-Verfahrens beginnt damit, dass man sich einige Bedingungen für bessere Minimierungsrichtungen $\{d^0, d^1, d^2, \ldots\}$ überlegt:

1. $f(x)$ soll im k-ten Iterationsschritt in $\mathrm{span}\{d^0, d^1, d^2, \ldots, d^{k-1}\}$ minimiert werden. Diese Bedingung ist stärker als der obige Schritt 2.
2. Die Vektoren d^k sollen linear unabhängig sein. Es folgt $\mathrm{span}\{d^0, d^1, d^2, \ldots, d^{n-1}\} = \mathbb{R}^n$.
3. Die Vektoren d^k sollen einfach zu berechnen sein.

zu 1.: Man kann folgende Bedingung für die Richtungen herleiten:

$$(Ad^k, d^j) = 0 \quad \text{für } j = 0, \ldots, k-1.$$

Diese Eigenschaft heisst A-orthogonal oder (A)-konjugiert.

zu 2.: Man kann zeigen: Falls $\{d^0, d^1, d^2, \ldots, d^k\}$ konjugiert sind, dann sind sie linear unabhängig. Es folgt: Das Minimum von $f(x)$ in $\mathbb{R}^n$ wird (bei exakter Arithmetik) in nicht mehr als n Iterationen berechnet.

zu 3.: Man nutzt einen geeigneten Ansatz:

$$d^k = r^k + \sum_{j=1}^{k-1} \beta_{kj} d^j.$$

Aus der Eigenschaft, dass die Vektoren d^j konjugiert sind, erhält man

$$\beta_{kj} = 0 \quad j \leq k-2, \qquad \beta_{k,k-1} = \frac{(r^k, r^k)}{(r^{k-1}, r^{k-1})}.$$

Das bedeutet, dass man in der k-ten Iteration nur d^{k-1} benötigt. Dadurch, dass man sich $\{d^0, \ldots, d^{k-2}\}$ nicht merken muss, spart man Speicher.

Satz 7.21 *Das CG-Verfahren konvergiert in exakter Arithmetik nach spätestens n Iteratioen. Sei $p(\varepsilon)$ wie oben definiert, dann ist*

$$p(\varepsilon) \leq \frac{1}{2}\sqrt{\mathrm{cond}_2(A)}\ln\left(\frac{2}{\varepsilon}\right) + 1.$$

Beweis: Siehe [AB84, S.24 ff.]. ■

Die Abschätzung der Konvergenzrate kann für gewisse Verteilungen der Eigenwerte von A verbessert werden. Bilden die Eigenwerte zum Beispiel Cluster, so verbessert sich die Konvergenzrate im allgemeinen.

Es gibt verschiedene Möglichkeiten, dass CG-Verfahren effizient zu implementieren. Eine davon ist wie folgt.

Algorithmus 7.22 *Serielles CG-Verfahren.* Gegeben sind A, x, b, und $\varepsilon > 0$.

```
r = Ax − b;
d0 = (r, r);
if(sqrt(d0) < ε);      // Residuum klein genug
    stop;
d = −r;
while(1)
    h = Ad;
    τ = d0/(d, h);
    x = x + τd;
    r = r + τh;
    d1 = (r, r);
    if(sqrt(d1) < ε)    // Residuum klein genug
      break;
    β = d1/d0;
    d0 = d1;
    d = −r + βd
endwhile
```

Das parallelisierte Verfahren sieht wie folgt aus.

Algorithmus 7.23 *Paralleles CG-Verfahren.* Gegeben sind $A, \mathbf{x}, b$, und $\varepsilon > 0$.

```
r = Ax − b;
h = r;
h = mache h konsistent;     // Kommunikation
d0 = (h, r);
d0 = mache d0 konsistent;     // Kommunikation
if(sqrt(d0) < ε);      // Residuum klein genug
    stop;
d = −h;
while(1)
    h = Ad;
    τ = (d, h);
    τ = mache τ konsistent;     // Kommunikation
    τ = d0/τ;
    x = x + τd;
    r = r + τh;
    h = r;
    h = mache h konsistent;     // Kommunikation
    d1 = (h, r);
    d1 = mache d1 konsistent;     // Kommunikation
```

```
20.     if(sqrt(d1) < ε)    // Residuum klein genug
21.       break;
22.     β = d1/d0;
23.     d0 = d1;
24.     d = −h + βd
25. endwhile
```

Die parallele Implementierung des CG-Verfahrens benötigt innerhalb eines Iterationsschritts:

- zwei Akkumulationen von skalaren Größen (All-Reduce-Operationen) (Zeilen 12 und 19),
- eine Konvertierung eines inkonsistent gespeichertern Vektors in einen konsistent gespeicherten Vektor (Zeile 17).

7.5.3 Das vorkonditionierte Verfahren der konjugierten Gradienten (PCG)

Precondinitioned conjugate gradient method. Die Analysis des CG-Verfahrens zeigt, dass die Konvergenzrate von $\mathrm{cond}_2(A)$ abhängt. Die Motivation für eine Vorkonditionierung besteht darin, dss man an Stelle von (7.1) ein symmetrisches und positiv definites System

$$\tilde{A}y = \tilde{b} \tag{7.10}$$

löst, wobei

- $\mathrm{cond}_2(\tilde{A}) < \mathrm{cond}_2(A)$, das heisst, dass CG-Verfahren konvergiert im allgemeinen schneller für (7.10).
- x kann einfach von y berechnet werden.

Um ein geeignetes System (7.10), nimmt man eine symmetrische und positiv definite Matrix C, die in der Form $C = EE^T$ faktorisiert wird, wobei $C^{-1} \approx A^{-1}$. C wird Vorkonditionierer genannt. Sei $y = E^T x$. Anstelle von (7.8), betrachtet man nun

$$\text{löse} \qquad \min_{y \in \mathbb{R}^n} \left\{ \frac{1}{2}(\tilde{A}y, y) - (\tilde{b}, y) \right\}, \tag{7.11}$$

mit $\tilde{A} = E^{-1}AE^{-T}$, $\tilde{b} = E^{-1}b$. Die Matrix $\tilde{A}$ ist symmetrisch und positiv definit:

$$\tilde{A}^T = (E^{-1}AE^{-T})^T = E^{-1}A^TE^{-T} = (E^{-1}AE^{-T} = \tilde{A},$$

$$y^T\tilde{A}y = y^TE^{-1}AE^{-T}y = x^TAx > 0, \quad \forall\ x \neq 0.$$

Man kann also das CG-Verfahren auf (7.11) anwenden. Es wird nun vorkonditioniertes CG-Verfahren (PCG-Verfahren) genannt.

Wegen der engen Beziehung von (7.11) und (7.8) kann man des PCG-Verfahren mit Hilfe von A, x, b und C aufschreiben. Die Faktormatrix E taucht dabei im Algorithmus nicht auf.

Eine serielle Implementierung des PCG-Verfahrens sieht wie folgt aus:

Algorithmus 7.24 *Serielles PCG-Verfahren.* Gegeben sind A, C, x, b, und $\varepsilon > 0$.

```
r = Ax − b;
h = C⁻¹r;
d0 = (r, h);
if(sqrt(d0) < ε)    // Residuum klein genug
    stop;
d = −h;
while(1)
    h = Ad;
    τ = d0/(d, h);
    x = x + τd;
```

```
11.     r = r + τh;
12.     h = C^-1 r;
13.     d1 = (r, h);
14.     if(sqrt(d1) < ε)    // Residuum klein genug
15.       break;
16.     β = d1/d0;
17.     d0 = d1;
18.     d = −h + βd;
19. endwhile
```

Es gibt nur zwei zusätzliche Kommandos im Vergleich zum CG-Verfahren (Zeilen 2 und 12). Deshalb wollen wir uns bei der Beschreibung einer parallelen Implementierung des PCG-Verfahrens auf diese zwei Kommandos beschränken. Vom CG-Verfahren ist bekannt, dass r inkonsistent gespeichert ist.

1. Fall: $\mathbf{C^{-1}}$ ist konsistent gespeichert.

```
h = C^-1 r;
h = mache h konsistent;     // Kommunikation
d1 = h^T r;
```

In diesem Fall hat man im Programm nur die Lösung des Vorkonditionierungssystems hinzuzufügen. Ein einfaches Beispiel ist $\mathbf{C} = \mathrm{diag}(A)$, wobei $\mathbf{C}$ von der inkonsistent gespeicherten Diagonalen von A vor der Iteration berechnet wird.

2. Fall: $\mathtt{C}^{-1}$ ist inkonsistent gespeichert. In diesem Fall ist $\mathtt{C}^{-1}\mathbf{r}$ nicht wohl definiert. Da der Vektor $\mathbf{r}$ immer inkonsistent gespeichert gebraucht wird, macht es keinen Sinn, ihn zu konvertieren. Falls wir $\mathtt{C}^{-1}$ umwandeln, haben wir Fall 1. Ein alternativer Weg ist die Einführung eines Hilfsvektors g.

```
g = r;
g = mache g konsistent;     // Kommunikation
h = C^-1 g;
h = mache h konsistent;     // Kommunikation
```

In diesem Fall braucht man eine Kommunikation mehr als in Fall 1.

Beispiel 7.25 *Verhalten des parallelen CG- und PCG-Verfahrens mit Diagonalvorkonditionierer ($C = diag(A)$). im Modellproblem.* Die Konfiguration des Beispiels ist analog zu Beispiel 7.12. Für $n_x = n_y = 384$, dass heisst 148225 Freiheitsgrade, erhält man folgende Iterationsanzahlen und Rechenzeiten auf dem HP-Superdome:

	CG-Verfahren		PCG-Verfahren	
Prozesse	Anzahl der Iterationen	Zeit	Anzahl der Iterationen	Zeit
1	972	67.08	945	69.39
4 (2×2)	972	14.95	945	15.70
9 (3×3)	972	5.58	945	6.15
16 (4×4)	972	0.88	945	1.56
36 (6×6)	972	0.37	945	0.45

Man kann folgende Beobachtungen treffen:

- Die Rechenzeit auf einem Prozessor ist vergleichbar mit der Zeit des SOR-Verfahrens mit $\omega = 1.9$ für ein System, dessen Dimension viermal kleiner ist. Damit sind das CG- und das PCG-Verfahren deutlich besser als das SOR-Verfahren und erst recht als das Jacobi-Verfahren.
- Die Vorkonditionierung verringert in diesem Beispiel zwar die Anzahl der Iterationen, die Rechenzeit steigt aber insgesamt, da der Aufwand pro Iteration größer ist.
- Der Speedup der Verfahren ist teilweise deutlich höher als 100 %. Für wenig Prozessoren, also größere Blöcke auf den einzelnen Prozessoren, scheint wieder der Speicherzugriff die Rechenzeit

wesentlich zu bestimmen. Für viel Prozessoren hat man beim CG-Verfahren

$$\text{Speedup}(16,36) \quad = \quad \frac{\text{Zeit auf 16 Prozessoren} \ \times \ \text{16 Prozessoren}}{\text{Zeit auf 36 Prozessoren} \ \times \ \text{36 Prozessoren}} \approx 1.06$$

(Beachte: angegebene Rechenzeiten sind gerundet.)

- Für $n_x = n_y = 192$ benötigt das CG-Verfahren 492 Iterationen und das PCG-Verfahren mit Diagonalvorkonditionierer 478 Iterationen. Damit verdoppelt sich die Anzahl der Iterationen bei einer Vervierfachung der Anzahl der Unbekannten. Das ist zwar besser als beim Jacobi- und beim SOR-Verfahren, aber noch nicht optimal.

Bemerkung 7.26 Die Parallelisierung anderer sogenannter Krylov-Teilraum-Verfahren, die auch zur Lösung von Systemen angewandt werden können, deren Matrix nicht symmetrisch und positiv definit ist (CGS, BiCGStab, GMRES), ist ebenfalls relativ einfach wie die Parallelisierung des CG-Verfahrens. Im Gegensatz dazu ist die Parallelisierung von Mehrgitterverfahren wesentlich komplizierter.

A Lösungen der Übungsaufgaben

Lösung zu Beispiel 2.1

```
/*************************************************************/
/*                                                           */
/* compute average and maximum of a sequence of random numbers */
/*                                                           */
/*************************************************************/

#include <stdlib.h>
#include "mpi.h"

/* define the master process */
#define MASTER  0

/* start of main program */
main(int argc,char *argv[])
{
  int rank, size;
  int LENGTH, i, loclength;
  double locsum, locmax, sum, max, *sbuf, *rbuf, t1, t2;
  MPI_Status status;

  /* initialize MPI */
  MPI_Init(&argc,&argv);

  /* every process asks its own number */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /* every process asks the total number of processes */
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  /* every process generates an array of non-negative random numbers */
  if (rank==MASTER)
  {
    printf("length of array: (non-positive finishes the program): ");
    scanf("%d", &LENGTH);
    if (LENGTH<=0)
      /* kill all prozeses */
      MPI_Abort(MPI_COMM_WORLD, 1);

    /* change LENGTH in such a way that the length can be divided by
       the number of processes without rest */
    loclength = LENGTH/size;
    if (LENGTH % size > 0)
      loclength++;
    LENGTH = loclength*size;

    /* starting time */
    t1 = MPI_Wtime();

    /* allocate array for random numbers */
```

```
  if (!(sbuf=malloc(LENGTH*sizeof(*sbuf))))
    MPI_Abort(MPI_COMM_WORLD, 1);

  /* fill array with random numbers */
  for (i=0;i<LENGTH;i++)
    sbuf[i] = (double)rand()/RAND_MAX;

  /* MASTER process sends the length of the array to all processes */
  for (i=1;i<size;i++)
    MPI_Send(&LENGTH, 1, MPI_INT, i, 0,  MPI_COMM_WORLD);
}
else
  /* non MASTER processes receive the length of the array */
  MPI_Recv(&LENGTH, 1, MPI_INT, MASTER, 0,  MPI_COMM_WORLD, &status);

/* every process computes the length of the part of the array (local array)
   which it should investigate */
loclength = LENGTH/size;

/* every process allocates memory in order to store the local array */
if (!(rbuf=malloc(loclength*sizeof(*rbuf))))
  MPI_Abort(MPI_COMM_WORLD, 1);

/* the data are distributed */
if (rank==MASTER)
{
  /* the master process sends the i-th part of the global array to
     process i */
  for (i=1;i<size;i++)
    MPI_Send(sbuf+i*loclength, loclength, MPI_DOUBLE, i, 0,  MPI_COMM_WORLD);

  /* the master process copies the beginning of the global array to its
     own local array */
  for (i=0;i<loclength;i++)
    rbuf[i]=sbuf[i];
}
else
  /* the other processes receive their data */
  MPI_Recv(rbuf, loclength, MPI_DOUBLE, MASTER, 0,  MPI_COMM_WORLD, &status);

/* short form of this sequence of commands                               */
/*                                                                       */
/*  MPI_Scatter(sbuf, loclength, MPI_DOUBLE, rbuf, loclength,            */
/*              MPI_DOUBLE, MASTER, MPI_COMM_WORLD);                     */
/*                                                                       */

/* every process computes the sum of the entries of its local array
   (also MASTER) */
locsum = 0;
for (i=0;i<loclength;i++)
  locsum += rbuf[i];

/* computation of the global sum */
/* every process sends its lokal sum to the MASTER */
/* the MASTER receives the singel numbers and accumulated immediately
   the sum */
if (rank==MASTER)
{
```

```
    sum = locsum;
    for (i=1;i<size;i++)
    {
      MPI_Recv(&locsum, 1, MPI_DOUBLE, i, 0,  MPI_COMM_WORLD, &status);
      sum += locsum;
    }
  }
  else
    MPI_Send(&locsum, 1, MPI_DOUBLE, MASTER, 0,  MPI_COMM_WORLD);

  /* short form of this sequence of commands                          */
  /*                                                                  */
  /* MPI_Reduce(&locsum, &sum, 1, MPI_DOUBLE, MPI_SUM, MASTER,        */
  /*            MPI_COMM_WORLD);                                      */

  /* every process computes the maximum of its local array */
  locmax = 0;
  for (i=0;i<loclength;i++)
    if (rbuf[i] > locmax)
      locmax = rbuf[i];
  /* every process writes its local maximum */
  printf("process %d : local maximum = %g \n",rank,locmax);

  /* compute global maximum */
  /* each process sends its lokal maximum to the MASTER */
  /* the MASTER receives the individual numbers and computes immediately
     the global maximum */
  if (rank==MASTER)
  {
    max = locmax;
    for (i=1;i<size;i++)
    {
      MPI_Recv(&locmax, 1, MPI_DOUBLE, i, 0,  MPI_COMM_WORLD, &status);
      if (locmax > max)
max = locmax;
    }
  }
  else
    MPI_Send(&locmax, 1, MPI_DOUBLE, MASTER, 0,  MPI_COMM_WORLD);

  /* short form of this sequence of commands                          */
  /*                                                                  */
  /* MPI_Reduce(&locmax, &max, 1, MPI_DOUBLE, MPI_MAX, MASTER,        */
  /*            MPI_COMM_WORLD);                                      */

  /* final time */
  t2 = MPI_Wtime();

  /* the MASTER process writes the computed values */
  if (rank==MASTER)
    printf("length %d, average = %g, maximum %g, time %g \n",
   LENGTH, sum/LENGTH, max, t2-t1);

  /* final MPI command */
  MPI_Finalize();

  return(EXIT_SUCCESS);
}
```

Lösung zu Beispiel 2.2

```
/*************************************************************/
/*                                                           */
/* compute average and maximum of a sequence of random numbers */
/*                                                           */
/*************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"

/* define the master process */
#define MASTER  0


/* start of main program */
int main(int argc, char *argv[]) {
  int rank, size;
  int LENGTH, i, loclength;
  double locsum, locmax, sum, max, *sbuf, *rbuf, t1, t2;

  /* initialize MPI */
  MPI_Init(&argc,&argv);

  /* every process asks its own number */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /* every process asks the total number of processes */
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  /* every process generates an array of non-negative random numbers */
  if (rank==MASTER) {
    printf("length of array: (non-positive finishes the program): ");
    scanf("%d", &LENGTH);
    if (LENGTH<=0)
      /* kill all prozeses */
      MPI_Abort(MPI_COMM_WORLD, 1);

    /* change LENGTH in such a way that the length can be divided by
       the number of processes without rest */
    if (LENGTH % size > 0)
      LENGTH+= size- (LENGTH % size);

    /* starting time */
    t1=MPI_Wtime();

    /* allocate array for random numbers */
    if (!(sbuf=malloc(LENGTH*sizeof(*sbuf))))
      MPI_Abort(MPI_COMM_WORLD, 1);

    /* fill array with random numbers */
    for (i=0; i<LENGTH; i++)
      sbuf[i]=(double)rand()/RAND_MAX;
  }

  /* braodcast array length */
  MPI_Bcast(&LENGTH, 1, MPI_INT, MASTER, MPI_COMM_WORLD);
```

```
  /* every process computes the length of the part of the array (local array)
     which it should investigate */
  loclength=LENGTH/size;

  /* every process allocates memory in order to store the local array */
  if (!(rbuf=malloc(loclength*sizeof(*rbuf))))
    MPI_Abort(MPI_COMM_WORLD, 1);

  MPI_Scatter(sbuf, loclength, MPI_DOUBLE, rbuf, loclength,
              MPI_DOUBLE, MASTER, MPI_COMM_WORLD);

  /* every process computes the sum of the entries of its local array
     (also MASTER) */
  locsum=0;
  for (i=0; i<loclength; i++)
    locsum+=rbuf[i];

  /* computation of the global sum */
  /* every process sends its lokal sum to the MASTER */
  /* the MASTER receives the singel numbers and accumulated immediately
     the sum */
  MPI_Reduce(&locsum, &sum, 1, MPI_DOUBLE, MPI_SUM, MASTER, MPI_COMM_WORLD);

  /* every process computes the maximum of its local array */
  locmax=0;
  for (i=0; i<loclength; i++)
    if (rbuf[i]>locmax)
      locmax=rbuf[i];
  /* every process writes its local maximum */
  printf("process %d : local maximum=%g \n",rank,locmax);

  /* compute global maximum */
  /* each process sends its lokal maximum to the MASTER */
  /* the MASTER receives the individual numbers and computes immediately
     the global maximum */
  MPI_Reduce(&locmax, &max, 1, MPI_DOUBLE, MPI_MAX, MASTER, MPI_COMM_WORLD);

  /* final time */
  t2=MPI_Wtime();

  /* the MASTER process writes the computed values */
  if (rank==MASTER)
    printf("length %d, average=%g, maximum %g, time %g \n",
   LENGTH, sum/LENGTH, max, t2-t1);

  /* final MPI command */
  MPI_Finalize();

  return(EXIT_SUCCESS);
}
```

Lösung zu Beispiel 3.4

```
/**********************************************************/
/*                                                        */
/* using different logical topologies                     */
/* the average of a set of numbers is to compute          */
/* add the end, every process should know the averagea    */
/*                                                        */
/**********************************************************/

#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"
#include <math.h>

/* define the  master prozess */
#define MASTER  0

/* start of main program */
main(int argc,char *argv[])
{
  int rank, size;
  int topology, i, d, rank_neigh,bit,send_done, receive_done,sends;
  double number, average, t1, t2;
  MPI_Status status;

  /* initialize MPI */
  MPI_Init(&argc,&argv);

  /* every process asks its own number */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /* every process asks the total number of processes */
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  /* the process master asks the type of logical topology */
  if (rank==MASTER)
  {
    printf("topology : \n");
    printf("    1   -  master-worker\n");
    printf("    2   -  pipe\n");
    printf("    3   -  ring\n");
    printf("    4   -  tree\n");
    printf("    5   -  hypercube\n");
    printf("  else -  Programmende\n");
    scanf("%d", &topology);

    if ((topology<1) ||(topology>5))
      {
        printf("program is terminated /n");
          /* kill all prozeses */
          MPI_Abort(MPI_COMM_WORLD, 1);
      }
  }

  /* MASTER sends the type of topology to the other processes */
  /* they are waiting for this information */
  MPI_Bcast(&topology, 1, MPI_INT,MASTER,MPI_COMM_WORLD);
```

```
  /* every process computes a number */
  number = rank*((double)rand())/RAND_MAX+rank*(rank-1)*size;
  printf("rank %d number %g \n",rank, number);

  switch(topology)
  {
  case 1:
    /* master-worker */
    /* step 1 : every process sends its number to the master */
    MPI_Reduce(&number, &average, 1, MPI_DOUBLE, MPI_SUM, MASTER,
       MPI_COMM_WORLD);
    /* the master computes the average */
    average /= size;
    /* and sends the average back to all processes */
    MPI_Bcast(&average, 1, MPI_DOUBLE,MASTER,MPI_COMM_WORLD);
    break;

  case 2:
    /* pipe */
    /* process with the highest rank sends its number to
       the process with the second highest rank */
    if (rank==size-1)
    {
      MPI_Send(&number, 1, MPI_DOUBLE, rank-1, 0,  MPI_COMM_WORLD);
    }
    /* all other processes waiting first for a message from its
       neighbour which rank is higher by one (successor)
       they add their value to the received data
       they send the new data to their neighbour which rank is smaller
       by one (predecessor), without process 0
    */
    else
    {
      MPI_Recv(&average, 1, MPI_DOUBLE, rank+1, 0,  MPI_COMM_WORLD, &status);
      number += average;
      if (rank >0)
        MPI_Send(&number, 1, MPI_DOUBLE, rank-1, 0,  MPI_COMM_WORLD);
    }
    /* process 0 knows the sum of all numbers
       it computes the average and starts
       the back sending is done with the same principle
    */
    if (rank==0)
    {
      average = number/size;
      MPI_Send(&average, 1, MPI_DOUBLE, 1, 0,  MPI_COMM_WORLD);
    }
    /* all other processes receive average from their predecessor
       and pass it to their successor
    */
    else
    {
      MPI_Recv(&average, 1, MPI_DOUBLE, rank-1, 0,  MPI_COMM_WORLD, &status);
      if (rank < size -1)
MPI_Send(&average, 1, MPI_DOUBLE, rank+1, 0,  MPI_COMM_WORLD);
    }
    break;
  case 3:
```

```
    /* ring */
    average = number;
    for (i=0;i<size -1;i++)
    {
      /* send own value to successor */
      if (rank < size -1)
MPI_Send(&average, 1, MPI_DOUBLE, rank+1, 0,  MPI_COMM_WORLD);
      else
MPI_Send(&average, 1, MPI_DOUBLE, 0, 0,  MPI_COMM_WORLD);
      /* receive value from predecessor */
      if (rank==0)
MPI_Recv(&average, 1, MPI_DOUBLE, size-1, 0,  MPI_COMM_WORLD, &status);
      else
MPI_Recv(&average, 1, MPI_DOUBLE, rank-1, 0,  MPI_COMM_WORLD, &status);
      /* sum up */
      average += number;
    }
    /* now the sum is computed, compute average */
    average /= size;
    break;
  case 4:
    /* tree */
    /* depth of the tree */
    d = (int) (log(size)/log(2)+0.5);
    average = number;
    send_done = 0;
    if (rank==0)
      printf("depth of tree %d\n",d);
    for (i=0;i<d;i++)
    {
      /* the process has to do only something if it has not send
         its data yet */
      if (!send_done)
      {
        bit = (rank & (1 << i) );
        if (bit)
        {
          /* send the data to the node closer to the root */
          rank_neigh = rank-(1 << i);
          /*printf("send %d to %d \n",rank, rank_neigh);*/
          MPI_Send(&average, 1, MPI_DOUBLE, rank_neigh, 0,  MPI_COMM_WORLD);
          send_done = 1;
        }
        else
        {
          /* receive data from nodes farer away from the root */
          rank_neigh = rank+(1 << i);
          if (rank_neigh>=size)
            continue;
          /*printf("receive %d from %d\n",rank,rank_neigh);*/
          MPI_Recv(&number, 1, MPI_DOUBLE, rank_neigh, 0,  MPI_COMM_WORLD, &status);
          average += number;
        }
      }
    }
    /* the sum is now computed on the master process */
    /* compute the average */
    if (rank==0)
```

```
  {
    average /=size;
    receive_done=1;
  }
  else
    receive_done=0;

  /* compute how many sends the process has to do */
  sends = 0;
  for (i=0;i<d;i++)
  {
    if (rank & (1 << i) )
      break;
    else
      sends++;
  }

  for (i=0;i<d;i++)
  {
    bit = (rank & (1 << (d-i-1)) );
    if (bit)
    {
      /* receive data from nodes closer to the root */
      if (!receive_done)
      {
        rank_neigh = rank-(1 << sends);
        /*printf("recv %d from %d \n",rank, rank_neigh);*/
        MPI_Recv(&average, 1, MPI_DOUBLE, rank_neigh, 0,  MPI_COMM_WORLD, &status);
        receive_done = 1;
      }
    }
    else
    {
      /* send data further after having received them */
      if ((receive_done)&&(sends))
      {
        rank_neigh = rank+(1 << (d-i-1));
        if (rank_neigh>=size)
          continue;
        /*printf("send %d to %d\n",rank,rank_neigh);*/
        MPI_Send(&average, 1, MPI_DOUBLE, rank_neigh, 0,  MPI_COMM_WORLD);
        sends--;
      }
    }
  }

  break;
case 5:
  /* Hypercube */
  /* dimension of the hypercube */
  d = (int) (log(size)/log(2)+0.5);
  average = number;
  if (rank==0)
    printf("dimension of hypercube %d\n",d);
  /* send and receive date from the process whose rank is different
     from my rank only in bit i */
  for (i=0;i<d;i++)
  {
```

```
      bit = (rank & (1 << i) );
      /*printf("rank %d  i %d bit %d\n",rank,i,bit);*/
      if (bit)
        rank_neigh = rank-(1 << i);
      else
        rank_neigh = rank+(1 << i);
      /*printf("rank %d  i %d neigh %d\n",rank,i,rank_neigh);*/
      MPI_Send(&average, 1, MPI_DOUBLE, rank_neigh, 0,  MPI_COMM_WORLD);
      MPI_Recv(&number, 1, MPI_DOUBLE, rank_neigh, 0,  MPI_COMM_WORLD, &status);
      average += number;
      }
    /* now the sum is computed, compute average */
    average /= size;
    break;
  }

  /* every process computes the result */
  printf("rank %d average %g \n",rank, average);

  /* final MPI command */
  MPI_Finalize();
  return(EXIT_SUCCESS);
}
```

B Programm zum Modellproblem

Finden der benachbarten Prozessoren

```
void ComputeNeighbours(int rank, int x_subdoms, int y_subdoms,
                       int *left_neighbour, int *right_neighbour,
                       int *lower_neighbour, int *upper_neighbour)

{
  if ((rank % x_subdoms) == 0)
    *left_neighbour = -1;
  else
    *left_neighbour = rank-1;

  if ((rank % x_subdoms) == x_subdoms -1)
    *right_neighbour = -1;
  else
    *right_neighbour = rank+1;

  if (rank < x_subdoms)
    *lower_neighbour = -1;
  else
    *lower_neighbour = rank - x_subdoms;

  if (rank >   x_subdoms*(y_subdoms-1) -1)
    *upper_neighbour = -1;
  else
    *upper_neighbour = rank + x_subdoms;

  return;
}
```

Brechnung der geometrischen Position eines Prozesses

```
void  LocalLeftLowerCorner(int rank, int x_subdoms, int y_subdoms,
                           double *x, double *y)
{
  int i;

  /* y - coordinate */
  i = rank/x_subdoms;
  *y = i * 1.0 / y_subdoms;

  /* x - coordinate */
  i = rank % x_subdoms;
  *x = i * 1.0/x_subdoms;

  return;
}
```

Brechnung der Werte einer beliebigen Ecke

```
void ComputeCoordinates(double x0, double y0, int index,
                        int x_subdoms, int y_subdoms,
                        int x_int_loc, int y_int_loc,
                        double *x, double *y)
{
  int i;

  /* y - coordinate */
  i = index/(x_int_loc+1);
  *y = y0 + 1.0 * i /(y_subdoms * y_int_loc);

  /* x - coordinate */
  i = index % (x_int_loc+1);
  *x = x0 + i * 1.0 /(x_subdoms *  x_int_loc);

  return;
}
```

Berechnung der Matrixstruktur

```
void ComputeMatrixStructure(int dof_loc, int x_int_loc, int y_int_loc,
                            int *RowPtr, int *KCol)
{
  int i, n, ii, jj, kk, ind[3], index, start, ind_row, ind_col, nx;
  int skip;
  nx = x_int_loc+1;

  /* initialize column array */
  for (i=0;i<7*dof_loc; i++)
    KCol[i] = -1;

  /* initialize row ptr array */
  for (i=0;i<=dof_loc; i++)
    RowPtr[i] = 7*i;

  /* loop over all rectangles containing two triangles */
  n = 2*x_int_loc * y_int_loc;
  index = -2;
  for (i=0;i<n; i++)
  {
    /* first triangel, left lower */
    if ((i%2)==0)
    {
      index++;
      if (((i/2) % x_int_loc)== 0)
        index++;
      /*printf("%d \n",index);*/
      ind[0] = index;
      ind[1] = index+nx;
      ind[2] = index+nx+1;
    }
    else
    /* second triangel, right upper */
    {
      ind[0] = index;
      ind[2] = index+nx+1;
      ind[1] = index+1;
    }
    /* test all pairs of indizes if they are already
       contained in the matrix structure */
    for (ii=0;ii<3;ii++)
    {
      ind_row = ind[ii];
      for (jj=0;jj<3;jj++)
      {
        ind_col = ind[jj];
        start = RowPtr[ind[ii]];
        for (kk = 0; kk<7;kk++)
        {
           /* column already in KCol */
          if ((KCol[start+kk])==ind_col)
             break;
          /* new column */
          if ((KCol[start+kk])==-1)
          {
             KCol[start+kk] = ind_col;
             break;
```

```
          }
        }

      }
    }
  }

  /* dense matrix structure */
  /* compute how many entries are in each row */
  skip = 0;
  for (i=0;i<dof_loc; i++)
  {
    start = 7*i;
    for (ii=0;ii<7;ii++)
    {
      /* no column entry */
      if (KCol[start+ii]==-1)
        skip++;
    }
    RowPtr[i+1] -= skip;
  }

  /* dense the column array */
  for (i=0;i<dof_loc; i++)
  {
     start = RowPtr[i];
     index = start;
     for (ii=7*i;ii<7*(i+1);ii++)
     {
       if (KCol[ii]!=-1)
       {
          KCol[index] = KCol[ii];
          index++;
       }
     }
  }
  return;
}
```

Assemblierung der Matrix und der rechten Seite

```
#include "headers.h"

void Assemble(int dof_loc, int x_int_loc, int y_int_loc,
              double x0, double y0, int x_subdoms, int y_subdoms,
              double hx, double hy,
              int *RowPtr, int *KCol, double *matrix_entries,
              double *rhs)
{
  int i, n, ii, jj, kk, ll, ind[3], index, nx, start, end;
  double val, x[3], y[3], xs, ys, area;
  double vx_ii, vy_ii, vx_jj, vy_jj;

  nx = x_int_loc+1;
  /* loop over all rectangles containing two triangles */
  n = 2*x_int_loc * y_int_loc;
  /* area of the triangles */
  area = hx*hy/2;
  index = -2;
  for (i=0;i<n; i++)
  {
    /* first triangel, left lower */
    if ((i%2)==0)
    {
      index++;
      if (((i/2) % x_int_loc)== 0)
        index++;
      /*printf("%d \n",index);*/
      ind[0] = index;
      ind[1] = index+nx;
      ind[2] = index+nx+1;
    }
    else
    /* second triangel, right upper */
    {
      ind[0] = index;
      ind[2] = index+nx+1;
      ind[1] = index+1;
    }

    /*****************************************************************/
    /* assemble right hand side                                      */
    /*****************************************************************/
    /* compute barycentre of the triangle */
    xs = ys = 0;
    for (ii=0;ii<3;ii++)
    {
      ComputeCoordinates(x0, y0, ind[ii], x_subdoms, y_subdoms,
                         x_int_loc, y_int_loc, x+ii, y+ii);
      xs += x[ii];
      ys += y[ii];
    }
    xs /= 3;
    ys /= 3;
    val = RightHandSide(xs,ys);

    /* accumulate right hand side */
    for (ii=0;ii<3;ii++)
```

```
      rhs[ind[ii]] += val * area/3.0;

    /****************************************************************/
    /* assemble matrix                                              */
    /****************************************************************/

    for (ii=0;ii<3;ii++)
    {
      // correspondig row in matrix structure
      start = RowPtr[ind[ii]];
      end = RowPtr[ind[ii]+1];

      /* compute gradient of basis function ii */
      jj = (ii+1)%3;
      kk = (ii+2)%3;
      vx_ii = y[kk]-y[jj];
      vy_ii = x[jj]-x[kk];

      for (jj=0;jj<3;jj++)
      {
        /* compute gradient of basis function jj */
        ll = (jj+1)%3;
        kk = (jj+2)%3;
        vx_jj = y[kk]-y[ll];
        vy_jj = x[ll]-x[kk];

        /* compute update for matrix entry */
        val = vx_ii * vx_jj + vy_ii * vy_jj;
        val /= (4*area);

        /* update matrix entry */
        for (kk=start;kk<end;kk++)
        {
          if ((KCol[kk])==ind[jj])
          {
            matrix_entries [kk] += val;
            break;
          }
        }
      } /* end jj */
    } /* end ii */
  }

  return;
}
```

Matrix-Vektor-Operationen

```
/****************************************************************/
/* basic linear algebra routines                                */
/****************************************************************/
#include <math.h>

/****************************************************************/
/* A * x = y                                                    */
/****************************************************************/
void MatVec(int dof_loc,int *RowPtr,int *KCol,double *matrix_entries,
            double *x,double *y)
{
   int i, j, k, index;
   double s, value;

  j = RowPtr[0];
  for(i=0;i<dof_loc;i++)
  {
    s = 0;
    k = RowPtr[i+1];
    for(;j<k;j++)
    {
      index = KCol[j];
      value = matrix_entries[j];
      s += value * x[index];
    }
    y[i] = s;
  } // endfor i

return;
}

/****************************************************************/
/* d = y - A * x                                                */
/****************************************************************/
double MatVecDefect(int dof_loc,int *RowPtr,int *KCol,double *matrix_entries,
                  double *x,double *y, double *d)
{
  int i, j, k, index;
  double s, value,res;

  res = 0;
  j = RowPtr[0];
  for(i=0;i<dof_loc;i++)
  {
    s = y[i];
    k = RowPtr[i+1];
    for(;j<k;j++)
    {
      index = KCol[j];
      value = matrix_entries[j];
      s -= value * x[index];
    }
    d[i] = s;
    res += s*s;
  } // endfor i

  res = sqrt(res);
```

```
return(res);
}
```

Setzen der Dirichletbedingungen

```
#include "headers.h"

/****************************************************************/
/* set Dirichlet values                                         */
/* the matrix and the rhs will change, they have to be still    */
/* stored inconsistently after the manipulations !!!            */
/****************************************************************/

void SetDirichletValues(int left_neighbour,int right_neighbour,
                        int lower_neighbour, int upper_neighbour,
                        int x_int_loc, int y_int_loc,
                        int x_subdoms, int y_subdoms,
                        double x0, double y0,
                        int *RowPtr, int *KCol, double *matrix_entries,
                        double *sol, double *rhs)
{
   int i, j, k, start, end, ii, nx, ny, startj, endj, row, find, cond;
   double x, y;

   nx = x_int_loc+1;
   ny = y_int_loc+1;

   /*************************************************************/
   /* there is no neighbour on the lower side                   */
   /*************************************************************/
   if ((lower_neighbour==-1))
   {
      for (i=0;i<nx-1;i++)
      {
         start = RowPtr[i];
         end = RowPtr[i+1];
         for (j=start;j<end;j++)
         {
            if (KCol[j]==i)
               matrix_entries[j] = 1;
            else
               matrix_entries[j] = 0;
         }
         ComputeCoordinates(x0, y0, i, x_subdoms, y_subdoms,
                            x_int_loc, y_int_loc, &x, &y);

         rhs[i] = BoundCond(x,y);
         sol[i] = rhs[i];
      }

      /* special treatment for last point */
      /* if there is a right neighbour, then set matrix_entries and rhs */
      /* to zero to ensure inconsistent storage */
      /* sol has to be stored consistently */
      cond = 1;
      start = RowPtr[nx-1];
      end = RowPtr[nx];
      /* x - coordinate is 1 */
      if (right_neighbour==-1)
      {
         for (j=start;j<end;j++)
         {
```

```
         if (KCol[j]==nx)
            matrix_entries[j] = 1;
         else
            matrix_entries[j] = 0;
      }
      ComputeCoordinates(x0, y0, nx-1, x_subdoms, y_subdoms,
                         x_int_loc, y_int_loc, &x, &y);

      rhs[nx-1] = BoundCond(x,y);
      sol[nx-1] = rhs[nx-1];
      cond = 1;
   }
   else
   {
      for (j=start;j<end;j++)
         matrix_entries[j] = 0;
      rhs[nx-1] = 0;
      ComputeCoordinates(x0, y0, nx-1, x_subdoms, y_subdoms,
                         x_int_loc, y_int_loc, &x, &y);

      sol[nx-1] = BoundCond(x,y);
   }

   /* condensate Dirichlet values */
   for (i=0;i<nx-1+cond;i++)
   {
      start = RowPtr[i];
      end = RowPtr[i+1];
      /* for each column of row i */
      for (j=start;j<end;j++)
      {
         /* this is the row corresponding to the column */
         row = KCol[j];
         if (row==i)
            continue;
         startj = RowPtr[row];
         endj =  RowPtr[row+1];
         /* go through this row */
         for (k=startj;k<endj;k++)
         {
            find = 0;
            /* find matrix entry a(row,i) */
            if (KCol[k]==i)
            {
               rhs[row] -= matrix_entries[k] * sol[i];
               matrix_entries[k] = 0;
               find ++;
               break;
            }
         }
         if (!find)
            printf("not find1 %d !!! \n",cond);
      }
   }
}

/****************************************************************/
/* there is no neighbour on the upper side                     */
```

```
/**********************************************************/
 if ((upper_neighbour==-1))
 {
    for (i=nx*(ny-1);i<nx*ny-1;i++)
    {
       start = RowPtr[i];
       end = RowPtr[i+1];
       for (j=start;j<end;j++)
       {
          if (KCol[j]==i)
             matrix_entries[j] = 1;
          else
             matrix_entries[j] = 0;
       }
       ComputeCoordinates(x0, y0, i, x_subdoms, y_subdoms,
                          x_int_loc, y_int_loc, &x, &y);

       rhs[i] = BoundCond(x,y);
       sol[i] = rhs[i];
    }

    /* special treatment for last point */
    /* if there is a right neibour, then set matrix_entries and rhs */
    /* to zero to ensure inconsistent storage */
    cond = 1;
    start = RowPtr[nx*ny-1];
    end = RowPtr[nx*ny];
    /* x - coordinate is 1 */
    if (right_neighbour==-1)
    {
       for (j=start;j<end;j++)
       {
          if (KCol[j]==nx*ny-1)
             matrix_entries[j] = 1;
          else
             matrix_entries[j] = 0;
       }
       ComputeCoordinates(x0, y0, nx*ny-1, x_subdoms, y_subdoms,
                          x_int_loc, y_int_loc, &x, &y);

       rhs[nx*ny-1] = BoundCond(x,y);
       sol[nx*ny-1] = rhs[nx*ny-1];
       cond = 1;
    }
    else
    {
       for (j=start;j<end;j++)
          matrix_entries[j] = 0;
       rhs[nx*ny-1] = 0;
       ComputeCoordinates(x0, y0, nx*ny-1, x_subdoms, y_subdoms,
                          x_int_loc, y_int_loc, &x, &y);
       sol[nx*ny-1] = BoundCond(x,y);
   }

    /* condensate Dirichlet values */
    for (i=nx*(ny-1);i<nx*ny-1+cond;i++)
    {
       start = RowPtr[i];
```

```
      end = RowPtr[i+1];
      /* for each column of row i */
      for (j=start;j<end;j++)
      {
         /* this is the row corresponding to the column */
         row = KCol[j];
         if (row==i)
            continue;
         startj = RowPtr[row];
         endj =  RowPtr[row+1];
         /* go through this row */
         for (k=startj;k<endj;k++)
         {
            find = 0;
            /* find matrix entry a(row,i) */
            if (KCol[k]==i)
            {
               rhs[row] -= matrix_entries[k] * sol[i];
               matrix_entries[k] = 0;
               find ++;
               break;
         }
         }
         if (!find)
            printf("not find2 %d !!! \n",cond);
      }
   }
}
/*************************************************************/
/* there is no neighbour on the left side                    */
/*************************************************************/
if ((left_neighbour==-1))
{
   for (ii=0;ii<ny-1;ii++)
   {
      i = ii * nx;
      start = RowPtr[i];
      end = RowPtr[i+1];
      for (j=start;j<end;j++)
      {
         if (KCol[j]==i)
            matrix_entries[j] = 1;
         else
            matrix_entries[j] = 0;
      }
      ComputeCoordinates(x0, y0, i, x_subdoms, y_subdoms,
                         x_int_loc, y_int_loc, &x, &y);

      rhs[i] = BoundCond(x,y);
      sol[i] = rhs[i];
   }

   /* special treatment for last point */
   /* if there is an upper neibour, then set matrix_entries and rhs */
   /* to zero to ensure inconsistent storage */
   cond = 1;
   start = RowPtr[(ny-1)*nx];
   end = RowPtr[(ny-1)*nx+1];
```

```
/* x - coordinate is 1 */
if (upper_neighbour==-1)
{
   for (j=start;j<end;j++)
   {
      if (KCol[j]==(ny-1)*nx)
         matrix_entries[j] = 1;
      else
         matrix_entries[j] = 0;
   }
   ComputeCoordinates(x0, y0, (ny-1)*nx, x_subdoms, y_subdoms,
                      x_int_loc, y_int_loc, &x, &y);

   rhs[nx*(ny-1)] = BoundCond(x,y);
   sol[nx*(ny-1)] = rhs[nx*(ny-1)];
   cond = 1;
}
else
{
   for (j=start;j<end;j++)
      matrix_entries[j] = 0;
   rhs[nx*(ny-1)] = 0;
   ComputeCoordinates(x0, y0, (ny-1)*nx, x_subdoms, y_subdoms,
                      x_int_loc, y_int_loc, &x, &y);
   sol[nx*(ny-1)] = BoundCond(x,y);
}

/* condensate Dirichlet values */
for (ii=0;ii<ny-1+cond;ii++)
{
   i = ii * nx;
   start = RowPtr[i];
   end = RowPtr[i+1];
   /* for each column of row i */
   for (j=start;j<end;j++)
   {
      /* this is the row corresponding to the column */
      row = KCol[j];
      if (row==i)
         continue;
      startj = RowPtr[row];
      endj =  RowPtr[row+1];
      /* go through this row */
      for (k=startj;k<endj;k++)
      {
         find = 0;
         /* find matrix entry a(row,i) */
         if (KCol[k]==i)
         {
            rhs[row] -= matrix_entries[k] * sol[i];
            matrix_entries[k] = 0;
            find ++;
            break;
         }
      }
      if (!find)
         printf("not find3 !!! \n");
   }
```

```
   }
}

/************************************************************/
/* there is no neighbour on the right side                  */
/************************************************************/
if ((right_neighbour==-1))
{
   for (ii=0;ii<ny-1;ii++)
   {
      i = (ii+1) * nx -1 ;
      start = RowPtr[i];
      end = RowPtr[i+1];
      for (j=start;j<end;j++)
      {
         if (KCol[j]==i)
            matrix_entries[j] = 1;
         else
            matrix_entries[j] = 0;
      }
      ComputeCoordinates(x0, y0, i, x_subdoms, y_subdoms,
                         x_int_loc, y_int_loc, &x, &y);

      rhs[i] = BoundCond(x,y);
      sol[i] = rhs[i];
   }

   /* special treatment for last point */
   /* if there is an upper neibour, then set matrix_entries and rhs */
   /* to zero to ensure inconsistent storage */
   cond = 1;
   start = RowPtr[nx*ny-1];
   end = RowPtr[nx*ny];
   /* x - coordinate is 1 */
   if (upper_neighbour==-1)
   {
      for (j=start;j<end;j++)
      {
         if (KCol[j]==nx*ny-1)
            matrix_entries[j] = 1;
         else
            matrix_entries[j] = 0;
      }
      ComputeCoordinates(x0, y0, nx*ny-1, x_subdoms, y_subdoms,
                         x_int_loc, y_int_loc, &x, &y);

      rhs[nx*ny-1] = BoundCond(x,y);
      sol[nx*ny-1] = rhs[nx*ny-1];
      cond = 1;
   }
   else
   {
      for (j=start;j<end;j++)
         matrix_entries[j] = 0;
      rhs[nx*ny-1] = 0;
      ComputeCoordinates(x0, y0, nx*ny-1, x_subdoms, y_subdoms,
                         x_int_loc, y_int_loc, &x, &y);
```

```
         sol[nx*ny-1] = BoundCond(x,y);
      }

      /* condensate Dirichlet values */
      for (ii=0;ii<ny-1+cond;ii++)
      {
         i = (ii+1) * nx -1 ;
         start = RowPtr[i];
         end = RowPtr[i+1];
         /* for each column of row i */
         for (j=start;j<end;j++)
         {
            /* this is the row corresponding to the column */
            row = KCol[j];
            if (row==i)
               continue;
            startj = RowPtr[row];
            endj =  RowPtr[row+1];
            /* go through this row */
            for (k=startj;k<endj;k++)
            {
               find = 0;
            /* find matrix entry a(row,i) */
               if (KCol[k]==i)
               {
                  rhs[row] -= matrix_entries[k] * sol[i];
                  matrix_entries[k] = 0;
                  find ++;
                  break;
               }
            }
            if (!find)
               printf("not find4 !!! \n");
         }
      }
   }
  return;
}
```

Vektor konsistent machen

```
/****************************************************************/
/* make vector consistent                                       */
/****************************************************************/
#include <stdlib.h>
#include "mpi.h"

void  MakeVectorConsistent(int left_neighbour,int right_neighbour,
                           int lower_neighbour,int upper_neighbour,
                           int x_int_loc,int  y_int_loc, double *vec)
{
   int i, nx, ny, length;
   double *sbuf, *rbuf0, *rbuf1;
   MPI_Status status;

   nx = x_int_loc+1;
   ny = y_int_loc+1;

   /* compute maximum of nx and ny */
   if (nx>ny)
      length = nx;
   else
      length = ny;

   /* allocate buffers for sending and receiving data */
   if (!(sbuf=malloc(length*sizeof(*sbuf))))
      MPI_Abort(MPI_COMM_WORLD, 1);
   if (!(rbuf0=malloc(length*sizeof(*rbuf0))))
      MPI_Abort(MPI_COMM_WORLD, 1);
   if (!(rbuf1=malloc(length*sizeof(*rbuf1))))
      MPI_Abort(MPI_COMM_WORLD, 1);

   /*************************************************************/
   /* lower and upper boundaries                                */
   /*************************************************************/

   /* to lower neighbour */
   if ((lower_neighbour>-1))
   {
      for (i=0;i<nx;i++)
         sbuf[i] = vec[i];
      MPI_Send(sbuf, length, MPI_DOUBLE, lower_neighbour, 0,  MPI_COMM_WORLD);
   }
    /* from upper  neighbour */
   if ((upper_neighbour>-1))
      MPI_Recv(rbuf0, length, MPI_DOUBLE, upper_neighbour, 0,  MPI_COMM_WORLD, &status);

   /* to  upper neighbour */
   if ((upper_neighbour>-1))
   {
       for (i=0;i<nx;i++)
         sbuf[i] = vec[nx*(ny-1)+i];
       MPI_Send(sbuf, length, MPI_DOUBLE, upper_neighbour, 0,  MPI_COMM_WORLD);
   }
   /* from lower neighbour */
   if ((lower_neighbour>-1))
      MPI_Recv(rbuf1, length, MPI_DOUBLE, lower_neighbour, 0,  MPI_COMM_WORLD, &status);
```

```
   /* update values */
   if ((upper_neighbour>-1))
   {
      for (i=0;i<nx;i++)
         vec[nx*(ny-1)+i] += rbuf0[i];
   }

   if ((lower_neighbour>-1))
   {
      for (i=0;i<nx;i++)
         vec[i] += rbuf1[i];
   }

   /**************************************************************/
   /* left and right boundaries                                  */
   /**************************************************************/

  /* to left neighbour */
   if ((left_neighbour>-1))
   {
       for (i=0;i<ny;i++)
         sbuf[i] = vec[i*nx];
       MPI_Send(sbuf, length, MPI_DOUBLE, left_neighbour, 0,  MPI_COMM_WORLD);
   }
   /* from right neighbour */
   if ((right_neighbour>-1))
      MPI_Recv(rbuf0, length, MPI_DOUBLE, right_neighbour, 0,  MPI_COMM_WORLD, &status);

   /* to right neighbour */
   if ((right_neighbour>-1))
   {
       for (i=0;i<ny;i++)
         sbuf[i] = vec[(i+1)*nx-1];
       MPI_Send(sbuf, length, MPI_DOUBLE, right_neighbour, 0,  MPI_COMM_WORLD);
   }
   /* from left neighbour */
   if ((left_neighbour>-1))
      MPI_Recv(rbuf1, length, MPI_DOUBLE, left_neighbour, 0,  MPI_COMM_WORLD, &status);

   /* update values */
   if ((right_neighbour>-1))
   {
      for (i=0;i<ny;i++)
         vec[(i+1)*nx-1] += rbuf0[i];
   }
   if ((left_neighbour>-1))
   {
      for (i=0;i<ny;i++)
         vec[i*nx] += rbuf1[i];
   }


   free(sbuf);
   free(rbuf0);

   return;
}
```

Gedämpftes Jacobi-Verfahren

```
#include "mpi.h"
#include "headers.h"

#include <stdlib.h>
#include <string.h>
#include <math.h>
/************************************************************/
/* jacobi iteration                                         */
/************************************************************/
void jacobi(int rank,
    int left_neighbour,int right_neighbour,
    int lower_neighbour,int upper_neighbour,
    int x_int_loc,int  y_int_loc,
    int maxit, double eps, double damp,
    int dof_loc, int *KCol, int *RowPtr,
    double *matrix_entries, double *sol, double *rhs,
    double *defect)
{
  int i, j, k, index, start, end;
  double *r, *diag, residual, residual_cons, old_residual;


  r = malloc(dof_loc*sizeof(double));
  diag = malloc(dof_loc*sizeof(double));
  old_residual = 1;

  /* compute diagonal of matrix */
  /* loop over all rows of local matrix */
  for(i=0;i<dof_loc;i++)
  {
    start = RowPtr[i];
    end = RowPtr[i+1];
    /* loop over all columns belonging to this row */
    for(j=start;j<end;j++)
    {
       /* index of column */
      index = KCol[j];
      /* if the same as row index */
      if (index==i)
      {
         /* diagonal found */
         diag[i] = matrix_entries[j];
break;
      }
    }
  }

  /* make diag consistently */
  MakeVectorConsistent(left_neighbour, right_neighbour,
                       lower_neighbour, upper_neighbour,
                       x_int_loc, y_int_loc, diag);


  for (k=0;k<maxit;k++)
  {
    /* compute defect, stored inconsistently */
    MatVecDefect(dof_loc, RowPtr, KCol, matrix_entries, sol,
```

```
 rhs, defect);
    /* copy defect to r */
    memcpy(r,defect,dof_loc*sizeof(double));

    /* make defect consistent */
    MakeVectorConsistent(left_neighbour, right_neighbour,
 lower_neighbour, upper_neighbour,
 x_int_loc, y_int_loc, defect);
    /* compute inner product of r and defect, residual is stored
       inconsistently */
    residual = 0;
    for (i=0;i<dof_loc;i++)
       residual +=  defect[i] * r[i];

    /* make residual consistent */
    MPI_Allreduce(&residual, &residual_cons, 1, MPI_DOUBLE, MPI_SUM,
  MPI_COMM_WORLD);
    residual_cons= sqrt(residual_cons);

    /*if (rank==0)
      printf("ite %3d %8.6e rate %g\n",k, residual_cons,
     residual_cons/old_residual);
    */
    old_residual = residual_cons;
    /* stopping criterion fulfilled, stop iteration */
    if (residual_cons<eps)
      break;
    /* else compute update */
    for (i=0;i<dof_loc;i++)
    {
       /* printf("%g %g %g %g\n ",sol[i],defect[i], diag[i],damp);*/
      sol[i] += damp*defect[i]/diag[i];
    }

  }

  if (rank==0)
    printf("Jacobi: iterations %d residual %g\n", k, residual_cons);

  free(r);
  free(diag);

  return;
}
```

Block-Jacobi-Verfahren mit inneren SOR-Iterationen

```
#include "mpi.h"
#include "headers.h"

#include <stdlib.h>
#include <string.h>
#include <math.h>
/************************************************************/
/* block Jacobi iteration with inner sor iteration            */
/************************************************************/
void sor(int rank,
         int left_neighbour,int right_neighbour,
         int lower_neighbour,int upper_neighbour,
         int x_int_loc,int  y_int_loc,
         int maxit, double eps, double damp,
         int dof_loc, int *KCol, int *RowPtr,
         double *matrix_entries, double *sol, double *rhs,
         double *defect)
{
  int i, j, k, index, start, end, nx, ny;
  double *r, *diag, residual, residual_cons, old_residual, sum;
  double *update;

  r = malloc(dof_loc*sizeof(double));
  diag = malloc(dof_loc*sizeof(double));
  update =  malloc(dof_loc*sizeof(double));
  old_residual = 1;
  nx = x_int_loc+1;
  ny = y_int_loc+1;
  /* compute diagonal of matrix */
  /* loop over all rows of local matrix */
  for(i=0;i<dof_loc;i++)
  {
    start = RowPtr[i];
    end = RowPtr[i+1];
    /* loop over all columns belonging to this row */
    for(j=start;j<end;j++)
    {
       /* index of column */
      index = KCol[j];
      /* if the same as row index */
      if (index==i)
      {
         /* diagonal found */
         diag[i] = matrix_entries[j];
         break;
      }
    }
  }

  /* make diag consistently */
  MakeVectorConsistent(left_neighbour, right_neighbour,
                       lower_neighbour, upper_neighbour,
                       x_int_loc, y_int_loc, diag);

  for (k=0;k<maxit;k++)
  {
    /* compute defect, stored inconsistently */
```

```
   MatVecDefect(dof_loc, RowPtr, KCol, matrix_entries, sol,
rhs, defect);
   /* copy defect to r */
   memcpy(r,defect,dof_loc*sizeof(double));

   /* make defect consistent */
   MakeVectorConsistent(left_neighbour, right_neighbour,
lower_neighbour, upper_neighbour,
x_int_loc, y_int_loc, defect);
   /* compute inner product of r and defect, residual is stored
      inconsistently */
   residual = 0;
   for (i=0;i<dof_loc;i++)
      residual +=  defect[i] * r[i];

   /* make residual consistent */
   MPI_Allreduce(&residual, &residual_cons, 1, MPI_DOUBLE, MPI_SUM,
 MPI_COMM_WORLD);
   residual_cons= sqrt(residual_cons);

   old_residual = residual_cons;
   /* stopping criterion fulfilled, stop iteration */
   if (residual_cons<eps)
     break;
   if (residual_cons > 1e10)
      break;
   /* else compute update */
   /* the unknowns on the other processes are not available */
   for (i=0;i<dof_loc;i++)
   {
      sum = 0;
      /* compute inner product with i-th row */
      start = RowPtr[i];
      end = RowPtr[i+1];
      /* loop over all columns belonging to this row */
      for(j=start;j<end;j++)
      {
         /* index of column */
         index = KCol[j];
         sum += sol[index] * matrix_entries[j];
      }
      /* sum is stored inconsistently, rhs too
         these values cannot be added to consistently stored sol */
      update[i] = damp*(sum - rhs[i])/diag[i];
      /* update inner d.o.f. */
      /* the update on the boundary of the domain is zero */
      if ((i<nx)||(i>=nx*(ny-1)) || (i%(nx) ==0) ||((i+1)%nx==0))
         continue;
      else
      {
         sol[i] -= update[i];
         update[i] = 0;
      }
   }
   /* make the update consistent */
   MakeVectorConsistent(left_neighbour, right_neighbour,
                        lower_neighbour, upper_neighbour,
                        x_int_loc, y_int_loc, update);
```

```
    /* update current iterate */
    for (i=0;i<dof_loc;i++)
       sol[i] -= update[i];

  }
  if (rank==0)
    printf("Block Jacobi with inner SOR: iterations %d residual %g\n", k, residual_cons);

  free(r);
  free(diag);
  free(update);

  return;
}
```

Block-Jacobi-Verfahren mit inneren SSOR-Iterationen

```
#include "mpi.h"
#include "headers.h"

#include <stdlib.h>
#include <string.h>
#include <math.h>
/************************************************************/
/* block Jacobi iteration with inner ssor iteration          */
/************************************************************/
void ssor(int rank,
          int left_neighbour,int right_neighbour,
          int lower_neighbour,int upper_neighbour,
          int x_int_loc,int  y_int_loc,
          int maxit, double eps, double damp,
          int dof_loc, int *KCol, int *RowPtr,
          double *matrix_entries, double *sol, double *rhs,
          double *defect)
{
  int i, j, k, index, start, end, nx, ny;
  double *r, *diag, residual, residual_cons, old_residual, sum;
  double *update;


  r = malloc(dof_loc*sizeof(double));
  diag = malloc(dof_loc*sizeof(double));
  update =  malloc(dof_loc*sizeof(double));
  old_residual = 1;
  nx = x_int_loc+1;
  ny = y_int_loc+1;

  /* compute diagonal of matrix */
  /* loop over all rows of local matrix */
  for(i=0;i<dof_loc;i++)
  {
    start = RowPtr[i];
    end = RowPtr[i+1];
    /* loop over all columns belonging to this row */
    for(j=start;j<end;j++)
    {
       /* index of column */
      index = KCol[j];
      /* if the same as row index */
      if (index==i)
      {
         /* diagonal found */
         diag[i] = matrix_entries[j];
break;
      }
    }
  }

  /* make diag consistently */
  MakeVectorConsistent(left_neighbour, right_neighbour,
                       lower_neighbour, upper_neighbour,
                       x_int_loc, y_int_loc, diag);
```

```
 for (k=0;k<maxit;k++)
 {
   /* compute defect, stored inconsistently */
   MatVecDefect(dof_loc, RowPtr, KCol, matrix_entries, sol,
rhs, defect);
   /* copy defect to r */
   memcpy(r,defect,dof_loc*sizeof(double));

   /* make defect consistent */
   MakeVectorConsistent(left_neighbour, right_neighbour,
lower_neighbour, upper_neighbour,
x_int_loc, y_int_loc, defect);
   /* compute inner product of r and defect, residual is stored
      inconsistently */
   residual = 0;
   for (i=0;i<dof_loc;i++)
      residual +=  defect[i] * r[i];

   /* make residual consistent */
   MPI_Allreduce(&residual, &residual_cons, 1, MPI_DOUBLE, MPI_SUM,
 MPI_COMM_WORLD);
   residual_cons= sqrt(residual_cons);

   if (rank==0)
     printf("ite %3d %8.6e rate %g\n",k, residual_cons,
     residual_cons/old_residual);

   old_residual = residual_cons;
   /* stopping criterion fulfilled, stop iteration */
   if (residual_cons<eps)
     break;
   /* else compute update */
   /* the unknowns on the other processes are not available */
   for (i=0;i<dof_loc;i++)
   {
      sum = 0;
      /* compute inner product with i-th row */
      start = RowPtr[i];
      end = RowPtr[i+1];
      /* loop over all columns belonging to this row */
      for(j=start;j<end;j++)
      {
         /* index of column */
         index = KCol[j];
         sum += sol[index] * matrix_entries[j];
      }
      /* sum is stored inconsistently, rhs too
         these values cannot be added to consistently stored sol */
      update[i] = damp*(sum - rhs[i])/diag[i];
      /* update inner d.o.f. */
      /* the update on the boundary of the domain is zero */
      if ((i<nx)||(i>=nx*(ny-1)) || (i%(nx) ==0) ||((i+1)%nx==0))
         continue;
      else
      {
         sol[i] -= update[i];
         update[i] = 0;
      }
```

```
    }
    /* make the update consistent */
    MakeVectorConsistent(left_neighbour, right_neighbour,
                         lower_neighbour, upper_neighbour,
                         x_int_loc, y_int_loc, update);
    /* update current iterate */
    for (i=0;i<dof_loc;i++)
       sol[i] -= update[i];
    /* backward loop */
    for (i=dof_loc-1;i>=0;i--)
    {
       sum = 0;
       /* compute inner product with i-th row */
       start = RowPtr[i];
       end = RowPtr[i+1];
       /* loop over all columns belonging to this row */
       for(j=start;j<end;j++)
       {
          /* index of column */
          index = KCol[j];
          sum += sol[index] * matrix_entries[j];
       }
       /* sum is stored inconsistently, rhs too
          these values cannot be added to consistently stored sol */
       update[i] = damp*(sum - rhs[i])/diag[i];
       /* update inner d.o.f. */
       /* the update on the boundary of the domain is zero */
       if ((i<nx)||(i>=nx*(ny-1)) || (i%(nx) ==0) ||((i+1)%nx==0))
          continue;
       else
       {
          sol[i] -= update[i];
          update[i] = 0;
       }
    }
    /* make the update consistent */
    MakeVectorConsistent(left_neighbour, right_neighbour,
                         lower_neighbour, upper_neighbour,
                         x_int_loc, y_int_loc, update);
    /* update current iterate */
    for (i=0;i<dof_loc;i++)
       sol[i] -= update[i];
  }

  if (rank==0)
    printf("Block Jacobi with inner SSOR: iterations %d residual %g\n", k, residual_cons);

  free(r);
  free(diag);
  free(update);

  return;
}
```

Das vorkonditionierte Verfahren der konjugierten Gradienten

```
#include "mpi.h"
#include "headers.h"

#include <stdlib.h>
#include <string.h>
#include <math.h>
/***********************************************************/
/* preconditioned conjugate gradient iteration             */
/* with diagonal preconditioner                            */
/***********************************************************/
void pcg(int rank,
         int left_neighbour,int right_neighbour,
         int lower_neighbour,int upper_neighbour,
         int x_int_loc,int  y_int_loc,
         int maxit, double eps, double damp,
         int dof_loc, int *KCol, int *RowPtr,
         double *matrix_entries, double *sol, double *rhs,
         double *defect)
{
  int i, j, k, start, end, index;
  double *r, *d, *diag, residual, tau, d0, d1, beta;

  // allocate arrays
  if (!(r=malloc(dof_loc*sizeof(double))))
     MPI_Abort(MPI_COMM_WORLD, 1);
  if (!(d=malloc(dof_loc*sizeof(double))))
     MPI_Abort(MPI_COMM_WORLD, 1);
  if (!(diag=malloc(dof_loc*sizeof(double))))
     MPI_Abort(MPI_COMM_WORLD, 1);

  /* compute diagonal of matrix */
  /* loop over all rows of local matrix */
  for(i=0;i<dof_loc;i++)
  {
    start = RowPtr[i];
    end = RowPtr[i+1];
    /* loop over all columns belonging to this row */
    for(j=start;j<end;j++)
    {
       /* index of column */
      index = KCol[j];
      /* if the same as row index */
      if (index==i)
      {
         /* diagonal found */
         diag[i] = matrix_entries[j];
break;
      }
    }
  }

  /* make diag consistently */
  MakeVectorConsistent(left_neighbour, right_neighbour,
                       lower_neighbour, upper_neighbour,
                       x_int_loc, y_int_loc, diag);

  k = 0;
```

```
/* compute defect, stored inconsistently */
MatVecDefect(dof_loc, RowPtr, KCol, matrix_entries, sol,
             rhs, r);
for (i=0;i<dof_loc;i++)
   r[i] = -r[i];
/* defect  = diag(A)^{-1} r */
for (i=0;i<dof_loc;i++)
{
   r[i] = -r[i];
   defect[i] = r[i]/diag[i];
}

/* make defect consistent */
MakeVectorConsistent(left_neighbour, right_neighbour,
                     lower_neighbour, upper_neighbour,
                     x_int_loc, y_int_loc, defect);
/* compute inner product of r and defect, residual is stored
   inconsistently */
residual = 0;
for (i=0;i<dof_loc;i++)
   residual +=  r[i] * defect[i];
/* make residual consistent */
MPI_Allreduce(&residual, &d0, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

/* stopping criterion fulfilled */
if (sqrt(d0) < eps)
   return;

/*if (rank==0)
  printf("ite %3d %8.6e\n",k, d0);*/

for (i=0;i<dof_loc;i++)
   d[i] = -defect[i];

/* loop */
while(1)
{
   k++;
   /*  r= Ad, r stored inconsistently */
   MatVec(dof_loc, RowPtr, KCol, matrix_entries, d, defect);
   /* (defect,d) */
   residual = 0;
   for (i=0;i<dof_loc;i++)
      residual += defect[i] * d[i];
   /* make inner product consistently */
   MPI_Allreduce(&residual, &tau, 1, MPI_DOUBLE, MPI_SUM,
                 MPI_COMM_WORLD);
   tau = d0/tau;
   /* compute updates, r is after this stored consistently */
   for (i=0;i<dof_loc;i++)
   {
      sol[i] += tau * d[i];
      r[i] += tau * defect[i];
      defect[i] = r[i]/diag[i];
   }
   /* make defect consistent */
   MakeVectorConsistent(left_neighbour, right_neighbour,
                        lower_neighbour, upper_neighbour,
```

```
                           x_int_loc, y_int_loc, defect);
     residual = 0;
     for (i=0;i<dof_loc;i++)
        residual +=  r[i] * defect[i];
     MPI_Allreduce(&residual, &d1, 1, MPI_DOUBLE, MPI_SUM,
                   MPI_COMM_WORLD);
     /*if (rank==0)
       printf("ite %3d %8.6e rate %g\n",k, sqrt(d1), sqrt(d1)/sqrt(d0));*/
     /* stopping criterion fulfilled */
     if (sqrt(d1) < eps)
        break;
     if (k>maxit)
        break;
     beta = d1/d0;
     d0 = d1;
     /* update d */
     for (i=0;i<dof_loc;i++)
        d[i] = -defect[i] + beta * d[i];
  }

  if (rank==0)
    printf("PCG: iterations %d residual %g\n", k, sqrt(d1));

  free(r);
  free(d);
  free(diag);

}
```

Hauptprogramm

```
/************************************************************/
/*                                                          */
/* 2d pde problem in the unit square with linear finite     */
/* elements                                                 */
/*                                                          */
/************************************************************/

#include "mpi.h"
#include "headers.h"

#include <stdlib.h>
#include <stdio.h>
#include <strings.h>
#include <math.h>

#include "example_ansatz.h"


/* define the master process */
#define MASTER  0

/* start of main program */
int main(int argc, char *argv[])
{
  int rank, size;
  int x_subdoms,  y_subdoms, i, *info, x_int_loc, y_int_loc;
  int x_int_glob, y_int_glob, dof_glob, dof_loc;
  double t1, t2, h_x, h_y, x0_loc, y0_loc, x, y;
  int left_neighbour, right_neighbour, lower_neighbour, upper_neighbour;
  int *KCol, *RowPtr, j, start, end;
  int solver_type, maxite = 100000;
  double *matrix_entries, *rhs, *sol, *defect, res, damp, eps=1e-6;

  /* initialize MPI */
  MPI_Init(&argc,&argv);

  /* every process asks its own number */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /* every process asks the total number of processes */
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  /* allocate integer array for data transfer */
  if (!(info=malloc(4*sizeof(*info))))
    MPI_Abort(MPI_COMM_WORLD, 1);

  /* the unit square will be divided in subdomains */
  /* the number of subdomains in x-direction will be read */
  /* the number of subdomains in y-direction will be computed */
  /* by number of processes/number of subdomains in x-direction */

  if (rank==MASTER)
  {
    printf("number of subdomains in x-direction: (non-positive finishes the program): ");
    scanf("%d", &x_subdoms);
    if (x_subdoms<=0)
      /* kill all prozeses */
```

```
    MPI_Abort(MPI_COMM_WORLD, 1);

  /* compute number of subdomains in y-direction */
  y_subdoms = size/x_subdoms;

  if (y_subdoms<=0)
    /* kill all prozeses */
    MPI_Abort(MPI_COMM_WORLD, 1);

  // determine the grid size
  printf("number of intervals in x-direction in each subdomain: ");
  printf("(non-positive finishes the program): ");
  scanf("%d", &x_int_loc);
  if (x_int_loc<=0)
    /* kill all prozeses */
    MPI_Abort(MPI_COMM_WORLD, 1);
  printf("number of intervals in y-direction in each subdomain: ");
  printf("(non-positive finishes the program): ");
  scanf("%d", &y_int_loc);
  if (y_int_loc<=0)
    /* kill all prozeses */
    MPI_Abort(MPI_COMM_WORLD, 1);

  /* fill array for transfering information to the other processes */
  info[0] = x_subdoms;
  info[1] = y_subdoms;
  info[2] = x_int_loc;
  info[3] = y_int_loc;
}

/* braodcast information */
MPI_Bcast(info, 4, MPI_INT, MASTER, MPI_COMM_WORLD);

/* store information to the local variables */
x_subdoms = info[0];
y_subdoms = info[1];
x_int_loc = info[2];
y_int_loc = info[3];

/**************************************************/
/* stop if there are processors which are not      */
/* involved in solving the problem (this makes the */
/* coding easier if there are no such processors   */
/**************************************************/
if (rank >= x_subdoms * y_subdoms)
{
   printf("number of subdomains %d does not match number of processors %d\n",
          x_subdoms * y_subdoms,size);
    MPI_Abort(MPI_COMM_WORLD, 1);
}
/**************************************************/
/* compute local and global parameters of the mesh */
/**************************************************/
/* number of local d.o.f. */
dof_loc = (x_int_loc+1)*(y_int_loc+1);
/* number of intervals in x-dirction (globally) */
x_int_glob = x_int_loc*x_subdoms;
/* number of intervals in y-dirction (globally) */
```

```
y_int_glob = y_int_loc*y_subdoms;
/* number of global d.o.f. */
dof_glob = (x_int_glob+1)*(y_int_glob+1);
if (rank==0)
{
   printf("number of global degrees of freedom : %d ",dof_glob);
   printf("(inner d.o.f.: %d )\n ",(x_int_glob-1)*(y_int_glob-1));
}
/* mesh width in x-direction */
h_x = 1.0/x_int_glob;
/* mesh width in y-direction */
h_y = 1.0/y_int_glob;

/* compute neighbours */
ComputeNeighbours(rank, x_subdoms, y_subdoms,
                  &left_neighbour, &right_neighbour,
                  &lower_neighbour, &upper_neighbour);

/* coordinates of local left lower corner */
LocalLeftLowerCorner(rank, x_subdoms, y_subdoms,&x0_loc,&y0_loc);

/************************************************/
/* compute matrix structure, allocte arrays for  */
/* solution and rhs                              */
/************************************************/

/* allocate memory */
/* not more than 7*dof_loc entries possible */
if (! (RowPtr = malloc(dof_loc*sizeof(int)+1)))
   MPI_Abort(MPI_COMM_WORLD, 1);
if (! (KCol = malloc(7*dof_loc*sizeof(int))))
   MPI_Abort(MPI_COMM_WORLD, 1);
ComputeMatrixStructure(dof_loc, x_int_loc, y_int_loc, RowPtr,
                       KCol);

/* allocate matrix entries and initialize */
if (! (matrix_entries = malloc(7*dof_loc*sizeof(double))))
    MPI_Abort(MPI_COMM_WORLD, 1);
memset(matrix_entries,0,7*dof_loc*sizeof(double));

/* allocate right hand side and initialize */
if (! (rhs = malloc(dof_loc*sizeof(double))))
   MPI_Abort(MPI_COMM_WORLD, 1);
memset(rhs,0,dof_loc*sizeof(double));

/* allocate solution and initialize */
if (! (sol = malloc(dof_loc*sizeof(double))))
   MPI_Abort(MPI_COMM_WORLD, 1);
memset(sol,0,dof_loc*sizeof(double));

/*********************************/
/* assembling of matrices and rhs */
/*********************************/
Assemble(dof_loc, x_int_loc, y_int_loc, x0_loc, y0_loc, x_subdoms, y_subdoms,
         h_x, h_y, RowPtr, KCol, matrix_entries, rhs);

SetDirichletValues(left_neighbour, right_neighbour,
                   lower_neighbour, upper_neighbour,
```

```
                    x_int_loc, y_int_loc, x_subdoms, y_subdoms,
                    x0_loc , y0_loc, RowPtr, KCol, matrix_entries,
                    sol, rhs);
  if (rank==0)
  {
     for (i=0;i<dof_loc;i++)
     {
         printf("u(%d) =  %g ; \n ",i+1,sol[i]);
     }
  }

  /********************/
  /* choose the solver */
  /********************/

  if (rank==0)
  {
     /* read solver type */
     printf("solver type:\n");
     printf("  1 - damped Jacobi \n");
     printf("  2 - Block Jacobi with inner SOR \n");
     printf("  3 - Block Jacobi with inner SSOR \n");
     printf("  4 - Conjugate gradient \n");
     printf("  5 - Preconditioned conjugate gradient with diagonal preconditioner \n");
     scanf("%d", &solver_type);
     /*wrong solver type */
     if ((solver_type < 1)|| (solver_type>5))
     {
printf("invalid solver type !!!\n");
MPI_Abort(MPI_COMM_WORLD, 1);
     }
  }
  /* send information to the other processors */
  MPI_Bcast(&solver_type, 1, MPI_INT, MASTER, MPI_COMM_WORLD);

  /* allocate defect */
  if (! (defect = malloc(dof_loc*sizeof(double))))
     MPI_Abort(MPI_COMM_WORLD, 1);
  memset(defect,0,dof_loc*sizeof(double));

  /* solve system */
  switch(solver_type)
  {
    case 1: /* damped Jacobi */
       /* read damping parameter */
       if (rank==0)
       {
  printf("damping parameter: ");
  scanf("%lf", &damp);
       }
       MPI_Bcast(&damp, 1, MPI_DOUBLE, MASTER, MPI_COMM_WORLD);
       /* starting time */
       t1 = MPI_Wtime();
       /* call Jacobi iteration */
       jacobi(rank, left_neighbour, right_neighbour,
              lower_neighbour, upper_neighbour,
              x_int_loc, y_int_loc,
              maxite, eps, damp, dof_loc, KCol, RowPtr,
```

```
             matrix_entries, sol, rhs, defect);
      /* final time */
      t2 = MPI_Wtime();
      break;
    case 2: /* SOR */
      /* read damping parameter */
      if (rank==0)
      {
  printf("damping parameter: ");
  scanf("%lf", &damp);
      }
      MPI_Bcast(&damp, 1, MPI_DOUBLE, MASTER, MPI_COMM_WORLD);
      /* starting time */
      t1 = MPI_Wtime();
      /* call SOR iteration */
      sor(rank, left_neighbour, right_neighbour,
          lower_neighbour, upper_neighbour,
          x_int_loc, y_int_loc,
          maxite, eps, damp, dof_loc, KCol, RowPtr,
          matrix_entries, sol, rhs, defect);
      /* final time */
      t2 = MPI_Wtime();
      break;
    case 3: /* SSOR */
      /* read damping parameter */
      if (rank==0)
      {
  printf("damping parameter: ");
  scanf("%lf", &damp);
      }
      MPI_Bcast(&damp, 1, MPI_DOUBLE, MASTER, MPI_COMM_WORLD);
      /* starting time */
      t1 = MPI_Wtime();
      /* call SSOR iteration */
      ssor(rank, left_neighbour, right_neighbour,
           lower_neighbour, upper_neighbour,
           x_int_loc, y_int_loc,
           maxite, eps, damp, dof_loc, KCol, RowPtr,
           matrix_entries, sol, rhs, defect);
      /* final time */
      t2 = MPI_Wtime();
      break;
    case 4: /* CG */
      /* starting time */
      t1 = MPI_Wtime();
      /* call SSOR iteration */
      cg(rank, left_neighbour, right_neighbour,
         lower_neighbour, upper_neighbour,
         x_int_loc, y_int_loc,
         maxite, eps, damp, dof_loc, KCol, RowPtr,
         matrix_entries, sol, rhs, defect);
      /* final time */
      t2 = MPI_Wtime();
      break;
    case 5: /* PCG */
      /* starting time */
      t1 = MPI_Wtime();
      /* call SSOR iteration */
```

```
       pcg(rank, left_neighbour, right_neighbour,
           lower_neighbour, upper_neighbour,
           x_int_loc, y_int_loc,
           maxite, eps, damp, dof_loc, KCol, RowPtr,
           matrix_entries, sol, rhs, defect);
       /* final time */
       t2 = MPI_Wtime();
       break;
 }

  if (rank==MASTER)
     printf("solver time  %g \n",t2-t1);
  /* final MPI command */
  MPI_Finalize();

  return(EXIT_SUCCESS);
}
```

C MPI-Befehle

MPI_Abort

Terminates MPI execution environment

Synopsis

```
include "mpi.h"
int MPI_Abort( MPI_Comm comm, int errorcode )
```

Input Parameters

comm communicator of tasks to abort
errorcode error code to return to invoking environment

Notes

Terminates all MPI processes associated with the communicator comm; in most systems (all to date), terminates all processes.

Notes for Fortran

All MPI routines in Fortran (except for MPI_WTIME and MPI_WTICK) have an additional argument ierr at the end of the argument list. ierr is an integer and has the same meaning as the return value of the routine in C. In Fortran, MPI routines are subroutines, and are invoked with the call statement.

All MPI objects (e.g., MPI_Datatype, MPI_Comm) are of type INTEGER in Fortran.

Location: abort.c

MPI_Allreduce

Combines values from all processes and distribute the result back to all processes
Synopsis

```
#include "mpi.h"
int MPI_Allreduce ( void *sendbuf, void *recvbuf, int count,
                    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )
```

Input Parameters

sendbuf	starting address of send buffer (choice)
count	number of elements in send buffer (integer)
datatype	data type of elements of send buffer (handle)
op	operation (handle)
comm	communicator (handle)

Output Parameter

recvbuf	starting address of receive buffer (choice)

Notes for Fortran

All MPI routines in Fortran (except for MPI_WTIME and MPI_WTICK) have an additional argument ierr at the end of the argument list. ierr is an integer and has the same meaning as the return value of the routine in C. In Fortran, MPI routines are subroutines, and are invoked with the call statement.
All MPI objects (e.g., MPI_Datatype, MPI_Comm) are of type INTEGER in Fortran.

Notes on collective operations

The reduction functions (MPI_Op) do not return an error value. As a result, if the functions detect an error, all they can do is either call MPI_Abort or silently skip the problem. Thus, if you change the error handler from MPI_ERRORS_ARE_FATAL to something else, for example, MPI_ERRORS_RETURN, then no error may be indicated.
The reason for this is the performance problems in ensuring that all collective routines return the same error value.

Errors

All MPI routines (except MPI_Wtime and MPI_Wtick) return an error value; C routines as the value of the function and Fortran routines in the last argument. Before the value is returned, the current MPI error handler is called. By default, this error handler aborts the MPI job. The error handler may be changed with MPI_Errhandler_set; the predefined error handler MPI_ERRORS_RETURN may be used to cause error values to be returned. Note that MPI does not guarentee that an MPI program can continue past an error.

MPI_ERR_BUFFER Invalid buffer pointer. Usually a null buffer where one is not valid.

MPI_ERR_COUNT Invalid count argument. Count arguments must be non-negative; a count of zero is often valid.

MPI_ERR_TYPE Invalid datatype argument. May be an uncommitted MPI_Datatype (see MPI_Type_commit).

MPI_ERR_OP Invalid operation. MPI operations (objects of type MPI_Op) must either be one of the predefined operations (e.g., MPI_SUM) or created with MPI_Op_create.

MPI_ERR_COMM Invalid communicator. A common error is to use a null communicator in a call (not even allowed in MPI_Comm_rank).

Location: allreduce.c

MPI_Barrier

Blocks until all process have reached this routine. **Synopsis**

```
#include "mpi.h"
int MPI_Barrier (MPI_Comm comm )
```

Input Parameters

comm communicator (handle)

Notes

Blocks the caller until all group members have called it; the call returns at any process only after all group members have entered the call.

Algorithm

If the underlying device cannot do better, a tree-like or combine algorithm is used to broadcast a message wto all members of the communicator. We can modifiy this to use "blocks" at a later time (see MPI_Bcast).

Notes for Fortran

All MPI routines in Fortran (except for MPI_WTIME and MPI_WTICK) have an additional argument ierr at the end of the argument list. ierr is an integer and has the same meaning as the return value of the routine in C. In Fortran, MPI routines are subroutines, and are invoked with the call statement.
All MPI objects (e.g., MPI_Datatype, MPI_Comm) are of type INTEGER in Fortran.

Errors

All MPI routines (except MPI_Wtime and MPI_Wtick) return an error value; C routines as the value of the function and Fortran routines in the last argument. Before the value is returned, the current MPI error handler is called. By default, this error handler aborts the MPI job. The error handler may be changed with MPI_Errhandler_set; the predefined error handler MPI_ERRORS_RETURN may be used to cause error values to be returned. Note that MPI does not guarentee that an MPI program can continue past an error.

MPI_SUCCESS No error; MPI routine completed successfully.

MPI_ERR_COMM Invalid communicator. A common error is to use a null communicator in a call (not even allowed in MPI_Comm_rank).

Location: barrier.c

MPI_Bcast

Broadcasts a message from the process with rank "root" to all other processes of the group.
Synopsis

```
#include "mpi.h"
int MPI_Bcast ( void *buffer, int count, MPI_Datatype datatype, int root,
               MPI_Comm comm )
```

Input/output Parameters

buffer	starting address of buffer (choice)
count	number of entries in buffer (integer)
datatype	data type of buffer (handle)
root	rank of broadcast root (integer)
comm	communicator (handle)

Algorithm

If the underlying device does not take responsibility, this function uses a tree-like algorithm to broadcast the message to blocks of processes. A linear algorithm is then used to broadcast the message from the first process in a block to all other processes. MPIR_BCAST_BLOCK_SIZE determines the size of blocks. If this is set to 1, then this function is equivalent to using a pure tree algorithm. If it is set to the size of the group or greater, it is a pure linear algorithm. The value should be adjusted to determine the most efficient value on different machines.

Notes for Fortran

All MPI routines in Fortran (except for MPI_WTIME and MPI_WTICK) have an additional argument ierr at the end of the argument list. ierr is an integer and has the same meaning as the return value of the routine in C. In Fortran, MPI routines are subroutines, and are invoked with the call statement.
All MPI objects (e.g., MPI_Datatype, MPI_Comm) are of type INTEGER in Fortran.

Errors

All MPI routines (except MPI_Wtime and MPI_Wtick) return an error value; C routines as the value of the function and Fortran routines in the last argument. Before the value is returned, the current MPI error handler is called. By default, this error handler aborts the MPI job. The error handler may be changed with MPI_Errhandler_set; the predefined error handler MPI_ERRORS_RETURN may be used to cause error values to be returned. Note that MPI does not guarentee that an MPI program can continue past an error.

MPI_SUCCESS No error; MPI routine completed successfully.

MPI_ERR_COMM Invalid communicator. A common error is to use a null communicator in a call (not even allowed in MPI_Comm_rank).

MPI_ERR_COUNT Invalid count argument. Count arguments must be non-negative; a count of zero is often valid.

MPI_ERR_TYPE Invalid datatype argument. May be an uncommitted MPI_Datatype (see MPI_Type_commit).

MPI_ERR_BUFFER Invalid buffer pointer. Usually a null buffer where one is not valid.

MPI_ERR_ROOT Invalid root. The root must be specified as a rank in the communicator. Ranks must be between zero and the size of the communicator minus one.

Location: bcast.c

MPI_Comm_rank

Determines the rank of the calling process in the communicator

Synopsis

```
#include "mpi.h"
int MPI_Comm_rank ( MPI_Comm comm, int *rank )
```

Input Parameters

comm communicator (handle)

Output Parameter

rank rank of the calling process in group of comm (integer)

Notes for Fortran

All MPI routines in Fortran (except for MPI_WTIME and MPI_WTICK) have an additional argument ierr at the end of the argument list. ierr is an integer and has the same meaning as the return value of the routine in C. In Fortran, MPI routines are subroutines, and are invoked with the call statement.
All MPI objects (e.g., MPI_Datatype, MPI_Comm) are of type INTEGER in Fortran.

Errors

All MPI routines (except MPI_Wtime and MPI_Wtick) return an error value; C routines as the value of the function and Fortran routines in the last argument. Before the value is returned, the current MPI error handler is called. By default, this error handler aborts the MPI job. The error handler may be changed with MPI_Errhandler_set; the predefined error handler MPI_ERRORS_RETURN may be used to cause error values to be returned. Note that MPI does not guarentee that an MPI program can continue past an error.

MPI_SUCCESS No error; MPI routine completed successfully.

MPI_ERR_COMM Invalid communicator. A common error is to use a null communicator in a call (not even allowed in MPI_Comm_rank).

Location: comm_rank.c

MPI_Comm_size

Determines the size of the group associated with a communictor
Synopsis

```
#include "mpi.h"
int MPI_Comm\_size ( MPI_Comm comm, int *size )
```

Input Parameter

comm communicator (handle)

Output Parameter

size number of processes in the group of comm (integer)

Notes

MPI_COMM_NULL is not considered a valid argument to this function.

Notes for Fortran

All MPI routines in Fortran (except for MPI_WTIME and MPI_WTICK) have an additional argument ierr at the end of the argument list. ierr is an integer and has the same meaning as the return value of the routine in C. In Fortran, MPI routines are subroutines, and are invoked with the call statement.
All MPI objects (e.g., MPI_Datatype, MPI_Comm) are of type INTEGER in Fortran.

Errors

All MPI routines (except MPI_Wtime and MPI_Wtick) return an error value; C routines as the value of the function and Fortran routines in the last argument. Before the value is returned, the current MPI error handler is called. By default, this error handler aborts the MPI job. The error handler may be changed with MPI_Errhandler_set; the predefined error handler MPI_ERRORS_RETURN may be used to cause error values to be returned. Note that MPI does not guarentee that an MPI program can continue past an error.

MPI_SUCCESS No error; MPI routine completed successfully.

MPI_ERR_COMM Invalid communicator. A common error is to use a null communicator in a call (not even allowed in MPI_Comm_rank).

MPI_ERR_ARG Invalid argument. Some argument is invalid and is not identified by a specific error class (e.g., MPI_ERR_RANK).

Location: comm_size.c

MPI_Finalize

Terminates MPI execution environment

Synopsis

```
#include "mpi.h"
int MPI_Finalize()
```

Notes

All processes must call this routine before exiting. The number of processes running after this routine is called is undefined; it is best not to perform much more than a return rc after calling MPI_Finalize.
Notes for Fortran

All MPI routines in Fortran (except for MPI_WTIME and MPI_WTICK) have an additional argument ierr at the end of the argument list. ierr is an integer and has the same meaning as the return value of the routine in C. In Fortran, MPI routines are subroutines, and are invoked with the call statement.

All MPI objects (e.g., MPI_Datatype, MPI_Comm) are of type INTEGER in Fortran.

Location:finalize.c

MPI_Gather

Gathers together values from a group of processes
Synopsis

```
#include "mpi.h"
int MPI_Gather ( void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int root, MPI_Comm comm )
```

Input Parameter

sendbuf	starting address of send buffer (choice)
sendcount	number of elements in send buffer (integer)
sendtype	data type of send buffer elements (handle)
recvcount	number of elements for any single receive (integer, significant only at root)
recvtype	data type of recv buffer elements (significant only at root) (handle)
root	rank of receiving process (integer)
comm	communicator (handle)

Output Parameter

recvbuf	address of receive buffer (choice, significant only at root)

Notes for Fortran

All MPI routines in Fortran (except for MPI_WTIME and MPI_WTICK) have an additional argument ierr at the end of the argument list. ierr is an integer and has the same meaning as the return value of the routine in C. In Fortran, MPI routines are subroutines, and are invoked with the call statement.
All MPI objects (e.g., MPI_Datatype, MPI_Comm) are of type INTEGER in Fortran.

Errors

All MPI routines (except MPI_Wtime and MPI_Wtick) return an error value; C routines as the value of the function and Fortran routines in the last argument. Before the value is returned, the current MPI error handler is called. By default, this error handler aborts the MPI job. The error handler may be changed with MPI_Errhandler_set; the predefined error handler MPI_ERRORS_RETURN may be used to cause error values to be returned. Note that MPI does not guarentee that an MPI program can continue past an error.

MPI_SUCCESS No error; MPI routine completed successfully.

MPI_ERR_COMM Invalid communicator. A common error is to use a null communicator in a call (not even allowed in MPI_Comm_rank).

MPI_ERR_COUNT Invalid count argument. Count arguments must be non-negative; a count of zero is often valid.

MPI_ERR_TYPE Invalid datatype argument. May be an uncommitted MPI_Datatype (see MPI_Type_commit).

MPI_ERR_BUFFER Invalid buffer pointer. Usually a null buffer where one is not valid.

Location: gather.c

MPI_Init

Initialize the MPI execution environment

Synopsis

```
#include "mpi.h"
int MPI_Init(int *argc, char ***argv)
```

Input Parameters

argc	Pointer to the number of arguments
argv	Pointer to the argument vector

Command line arguments

MPI specifies no command-line arguments but does allow an MPI implementation to make use of them.

-mpiqueue	print out the state of the message queues when MPI_FINALIZE is called. All processors print; the output may be hard to decipher. This is intended as a debugging aid.
-mpiversion	print out the version of the implementation (not of MPI), including the arguments that were used with configure.
-mpinice nn	Increments the nice value by nn (lowering the priority of the program by nn). nn must be positive (except for root). Not all systems support this argument; those that do not will ignore it.
-mpedbg	Start a debugger in an xterm window if there is an error (either detected by MPI or a normally fatal signal). This works only if MPICH was configured with -mpedbg. CURRENTLY DISABLED. If you have TotalView, -mpichtv or mpirun -tv will give you a better environment anyway.
-mpimem	If MPICH was built with -DMPIR_DEBUG_MEM, this checks all malloc and free operations (internal to MPICH) for signs of injury to the memory allocation areas.
-mpidb	options Activate various debugging options. Some require that MPICH have been built with special options. These are intended for debugging MPICH, not for debugging user programs. The available options include:

mem	- Enable dynamic memory tracing of internal MPI objects
memall	- Generate output of all memory allocation/deallocation
ptr	- Enable tracing of internal MPI pointer conversions
rank n	- Limit subsequent -mpidb options to on the process with the specified rank in MPI_COMM_WORLD. A rank of -1 selects all of MPI_COMM_WORLD.
ref	- Trace use of internal MPI objects
reffile filename	- Trace use of internal MPI objects with output to the indicated file
trace	- Trace routine calls

Notes

Note that the Fortran binding for this routine has only the error return argument (MPI_INIT(ierror))
Because the Fortran and C versions of MPI_Init are different, there is a restriction on who can call MPI_Init. The version (Fortran or C) must match the main program. That is, if the main program is in C, then the C version of MPI_Init must be called. If the main program is in Fortran, the Fortran version must be called.
On exit from this routine, all processes will have a copy of the argument list. This is not required by the MPI standard, and truely portable codes should not rely on it. This is provided as a service by this implementation (an MPI implementation is allowed to distribute the command line arguments but is not required to).
Command line arguments are not provided to Fortran programs. More precisely, non-standard Fortran routines such as getarg and iargc have undefined behavior in MPI and in this implementation.
The MPI standard does not say what a program can do before an MPI_INIT or after an MPI_FINALIZE. In the MPICH implementation, you should do as little as possible. In particular, avoid anything that changes the external state of the program, such as opening files, reading standard input or writing to standard output.

Signals used

The MPI standard requires that all signals used be documented. The MPICH implementation itself uses no signals, but some of the software that MPICH relies on may use some signals. The list below is partial and should be independantly checked if you (and any package that you use) depend on particular signals.

IBM POE/MPL for SP2

SIGHUP, SIGINT, SIGQUIT, SIGFPE, SIGSEGV, SIGPIPE, SIGALRM, SIGTERM, SIGIO

-mpedbg switch

SIGQUIT, SIGILL, SIGFPE, SIGBUS, SIGSEGV, SIGSYS

Meiko CS2

SIGUSR2

ch_p4 device

SIGUSR1

The ch_p4 device also catches SIGINT, SIGFPE, SIGBUS, and SIGSEGV; this helps the p4 device (and MPICH) more gracefully abort a failed program.

Intel Paragon (ch_nx and nx device)

SIGUSR2

Shared Memory (ch_shmem device)

SIGCHLD

Note that if you are using software that needs the same signals, you may find that there is no way to use that software with the MPI implementation. The signals that cause the most trouble for applications include SIGIO, SIGALRM, and SIGPIPE. For example, using SIGIO and SIGPIPE may prevent X11 routines from working.

Errors

All MPI routines (except MPI_Wtime and MPI_Wtick) return an error value; C routines as the value of the function and Fortran routines in the last argument. Before the value is returned, the current MPI error handler is called. By default, this error handler aborts the MPI job. The error handler may be changed with MPI_Errhandler_set; the predefined error handler MPI_ERRORS_RETURN may be used to cause error values to be returned. Note that MPI does not guarentee that an MPI program can continue past an error.

MPI_SUCCESS

No error; MPI routine completed successfully.

MPI_ERR_OTHER

This error class is associated with an error code that indicates that an attempt was made to call MPI_INIT a second time. MPI_INIT may only be called once in a program.

Location:init.c

MPI_Recv

Basic receive

Synopsis

```
#include "mpi.h"
int MPI_Recv( void *buf, int count, MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm, MPI_Status *status )
```

Input Parameters

count	maximum number of elements in receive buffer (integer)
datatype	datatype of each receive buffer element (handle)
source	rank of source (integer)
tag	message tag (integer)
comm	communicator (handle)

Output Parameters

buf	initial address of receive buffer (choice)
status	status object (Status)

Notes

The count argument indicates the maximum length of a message; the actual number can be determined with MPI_Get_count.

Notes for Fortran

All MPI routines in Fortran (except for MPI_WTIME and MPI_WTICK) have an additional argument ierr at the end of the argument list. ierr is an integer and has the same meaning as the return value of the routine in C. In Fortran, MPI routines are subroutines, and are invoked with the call statement.
All MPI objects (e.g., MPI_Datatype, MPI_Comm) are of type INTEGER in Fortran.

Errors

All MPI routines (except MPI_Wtime and MPI_Wtick) return an error value; C routines as the value of the function and Fortran routines in the last argument. Before the value is returned, the current MPI error handler is called. By default, this error handler aborts the MPI job. The error handler may be changed with MPI_Errhandler_set; the predefined error handler MPI_ERRORS_RETURN may be used to cause error values to be returned. Note that MPI does not guarentee that an MPI program can continue past an error.

MPI_SUCCESS No error; MPI routine completed successfully.

MPI_ERR_COMM Invalid communicator. A common error is to use a null communicator in a call (not even allowed in MPI_Comm_rank).

MPI_ERR_TYPE Invalid datatype argument. May be an uncommitted MPI_Datatype (see MPI_Type_commit).

MPI_ERR_COUNT Invalid count argument. Count arguments must be non-negative; a count of zero is often valid.

MPI_ERR_TAG Invalid tag argument. Tags must be non-negative; tags in a receive (MPI_Recv, MPI_Irecv, MPI_Sendrecv, etc.) may also be MPI_ANY_TAG. The largest tag value is available through the the attribute MPI_TAG_UB.

MPI_ERR_RANK Invalid source or destination rank. Ranks must be between zero and the size of the communicator minus one; ranks in a receive (MPI_Recv, MPI_Irecv, MPI_Sendrecv, etc.) may also be MPI_ANY_SOURCE.

Location: recv.c

MPI_Reduce

Reduces values on all processes to a single value
Synopsis

```
#include "mpi.h"
int MPI_Reduce ( void *sendbuf, void *recvbuf, int count,
                 MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm )
```

Input Parameters

sendbuf	address of send buffer (choice)
count	number of elements in send buffer (integer)
datatype	data type of elements of send buffer (handle)
op	reduce operation (handle)
root	rank of root process (integer)
comm	communicator (handle)

Output Parameter

recvbuf	address of receive buffer (choice, significant only at root)

Algorithm

This implementation currently uses a simple tree algorithm.

Notes for Fortran

All MPI routines in Fortran (except for MPI_WTIME and MPI_WTICK) have an additional argument ierr at the end of the argument list. ierr is an integer and has the same meaning as the return value of the routine in C. In Fortran, MPI routines are subroutines, and are invoked with the call statement.
All MPI objects (e.g., MPI_Datatype, MPI_Comm) are of type INTEGER in Fortran.

Notes on collective operations

The reduction functions (MPI_Op) do not return an error value. As a result, if the functions detect an error, all they can do is either call MPI_Abort or silently skip the problem. Thus, if you change the error handler from MPI_ERRORS_ARE_FATAL to something else, for example, MPI_ERRORS_RETURN, then no error may be indicated.
The reason for this is the performance problems in ensuring that all collective routines return the same error value.

Errors

All MPI routines (except MPI_Wtime and MPI_Wtick) return an error value; C routines as the value of the function and Fortran routines in the last argument. Before the value is returned, the current MPI error handler is called. By default, this error handler aborts the MPI job. The error handler may be changed with MPI_Errhandler_set; the predefined error handler MPI_ERRORS_RETURN may be used to cause error values to be returned. Note that MPI does not guarentee that an MPI program can continue past an error.

MPI_SUCCESS No error; MPI routine completed successfully.

MPI_ERR_COMM Invalid communicator. A common error is to use a null communicator in a call (not even allowed in MPI_Comm_rank).

MPI_ERR_COUNT Invalid count argument. Count arguments must be non-negative; a count of zero is often valid.

MPI_ERR_TYPE Invalid datatype argument. May be an uncommitted MPI_Datatype (see MPI_Type_commit).

MPI_ERR_BUFFER Invalid buffer pointer. Usually a null buffer where one is not valid.

MPI_ERR_BUFFER This error class is associcated with an error code that indicates that two buffer arguments are aliased; that is, the describe overlapping storage (often the exact same storage). This is prohibited in MPI (because it is prohibited by the Fortran standard, and rather than have a separate case for C and Fortran, the MPI Forum adopted the more restrictive requirements of Fortran).

Location: reduce.c

MPI_Scatter

Sends data from one task to all other tasks in a group
Synopsis

```
#include "mpi.h"
int MPI_Scatter ( void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf,
                 int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm )
```

Input Parameters

sendbuf	address of send buffer (choice, significant only at root)
sendcount	number of elements sent to each process (integer, significant only at root)
sendtype	data type of send buffer elements (significant only at root) (handle)
recvcount	number of elements in receive buffer (integer)
recvtype	data type of receive buffer elements (handle)
root	rank of sending process (integer)
comm	communicator (handle)

Output Parameter

recvbuf	address of receive buffer (choice)

Notes for Fortran

All MPI routines in Fortran (except for MPI_WTIME and MPI_WTICK) have an additional argument ierr at the end of the argument list. ierr is an integer and has the same meaning as the return value of the routine in C. In Fortran, MPI routines are subroutines, and are invoked with the call statement.
All MPI objects (e.g., MPI_Datatype, MPI_Comm) are of type INTEGER in Fortran.

Errors

All MPI routines (except MPI_Wtime and MPI_Wtick) return an error value; C routines as the value of the function and Fortran routines in the last argument. Before the value is returned, the current MPI error handler is called. By default, this error handler aborts the MPI job. The error handler may be changed with MPI_Errhandler_set; the predefined error handler MPI_ERRORS_RETURN may be used to cause error values to be returned. Note that MPI does not guarentee that an MPI program can continue past an error.

MPI_SUCCESS No error; MPI routine completed successfully.

MPI_ERR_COMM Invalid communicator. A common error is to use a null communicator in a call (not even allowed in MPI_Comm_rank).

MPI_ERR_COUNT Invalid count argument. Count arguments must be non-negative; a count of zero is often valid.

MPI_ERR_TYPE Invalid datatype argument. May be an uncommitted MPI_Datatype (see MPI_Type_commit).

MPI_ERR_BUFFER Invalid buffer pointer. Usually a null buffer where one is not valid.

Location: scatter.c

MPI_Send

Performs a basic send
Synopsis

```
#include "mpi.h"
int MPI_Send( void *buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm )
```

Input Parameters

buf	initial address of send buffer (choice)
count	number of elements in send buffer (nonnegative integer)
datatype	datatype of each send buffer element (handle)
dest	rank of destination (integer)
tag	message tag (integer)
comm	communicator (handle)

Notes

This routine may block until the message is received.

Notes for Fortran

All MPI routines in Fortran (except for MPI_WTIME and MPI_WTICK) have an additional argument ierr at the end of the argument list. ierr is an integer and has the same meaning as the return value of the routine in C. In Fortran, MPI routines are subroutines, and are invoked with the call statement.
All MPI objects (e.g., MPI_Datatype, MPI_Comm) are of type INTEGER in Fortran.

Errors

All MPI routines (except MPI_Wtime and MPI_Wtick) return an error value; C routines as the value of the function and Fortran routines in the last argument. Before the value is returned, the current MPI error handler is called. By default, this error handler aborts the MPI job. The error handler may be changed with MPI_Errhandler_set; the predefined error handler MPI_ERRORS_RETURN may be used to cause error values to be returned. Note that MPI does not guarentee that an MPI program can continue past an error.

MPI_SUCCESS No error; MPI routine completed successfully.

MPI_ERR_COMM Invalid communicator. A common error is to use a null communicator in a call (not even allowed in MPI_Comm_rank).

MPI_ERR_COUNT Invalid count argument. Count arguments must be non-negative; a count of zero is often valid.

MPI_ERR_TYPE Invalid datatype argument. May be an uncommitted MPI_Datatype (see MPI_Type_commit).

MPI_ERR_TAG Invalid tag argument. Tags must be non-negative; tags in a receive (MPI_Recv, MPI_Irecv, MPI_Sendrecv, etc.) may also be MPI_ANY_TAG. The largest tag value is available through the the attribute MPI_TAG_UB.

MPI_ERR_RANK Invalid source or destination rank. Ranks must be between zero and the size of the communicator minus one; ranks in a receive (MPI_Recv, MPI_Irecv, MPI_Sendrecv, etc.) may also be MPI_ANY_SOURCE.

See Also MPI_Isend, MPI_Bsend

Location: send.c

MPI_Sendrecv

Sends and receives a message
Synopsis

```
#include "mpi.h"
int MPI_Sendrecv( void *sendbuf, int sendcount, MPI_Datatype sendtype,
                  int dest, int sendtag,
                  void *recvbuf, int recvcount, MPI_Datatype recvtype,
                  int source, int recvtag, MPI_Comm comm, MPI_Status *status )
```

Input Parameters

sendbuf	initial address of send buffer (choice)
sendcount	number of elements in send buffer (integer)
sendtype	type of elements in send buffer (handle)
dest	rank of destination (integer)
sendtag	send tag (integer)
recvcount	number of elements in receive buffer (integer)
recvtype	type of elements in receive buffer (handle)
source	rank of source (integer)
recvtag	receive tag (integer)
comm	communicator (handle)

Output Parameters

recvbuf	initial address of receive buffer (choice)
status	status object (Status). This refers to the receive operation.

Notes for Fortran

All MPI routines in Fortran (except for MPI_WTIME and MPI_WTICK) have an additional argument ierr at the end of the argument list. ierr is an integer and has the same meaning as the return value of the routine in C. In Fortran, MPI routines are subroutines, and are invoked with the call statement.
All MPI objects (e.g., MPI_Datatype, MPI_Comm) are of type INTEGER in Fortran.

Errors

All MPI routines (except MPI_Wtime and MPI_Wtick) return an error value; C routines as the value of the function and Fortran routines in the last argument. Before the value is returned, the current MPI error handler is called. By default, this error handler aborts the MPI job. The error handler may be changed with MPI_Errhandler_set; the predefined error handler MPI_ERRORS_RETURN may be used to cause error values to be returned. Note that MPI does not guarentee that an MPI program can continue past an error.

MPI_SUCCESS No error; MPI routine completed successfully.

MPI_ERR_COMM Invalid communicator. A common error is to use a null communicator in a call (not even allowed in MPI_Comm_rank).

MPI_ERR_COUNT Invalid count argument. Count arguments must be non-negative; a count of zero is often valid.

MPI_ERR_TYPE Invalid datatype argument. May be an uncommitted MPI_Datatype (see MPI_Type_commit).

MPI_ERR_TAG Invalid tag argument. Tags must be non-negative; tags in a receive (MPI_Recv, MPI_Irecv, MPI_Sendrecv, etc.) may also be MPI_ANY_TAG. The largest tag value is available through the the attribute MPI_TAG_UB.

MPI_ERR_RANK Invalid source or destination rank. Ranks must be between zero and the size of the communicator minus one; ranks in a receive (MPI_Recv, MPI_Irecv, MPI_Sendrecv, etc.) may also be MPI_ANY_SOURCE.

Location: sendrecv.c

MPI_Wtime

Returns an elapsed time on the calling processor
Synopsis

```
#include "mpi.h"
double MPI_Wtime()
```

Return value

Time in seconds since an arbitrary time in the past.

Notes

This is intended to be a high-resolution, elapsed (or wall) clock. See MPI_WTICK to determine the resolution of MPI_WTIME. If the attribute MPI_WTIME_IS_GLOBAL is defined and true, then the value is synchronized across all processes in MPI_COMM_WORLD.

Notes for Fortran
This is a function, declared as DOUBLE PRECISION MPI_WTIME() in Fortran.

See Also also: MPI_Wtick, MPI_Attr_get

Location: wtime.c

Literaturverzeichnis

[AB84] O. Axelsson and V.A. Barker. *Finite Element Solutions of Boundary Value Problems.* Academic Press, 1984.

[Fro90] A. Frommer. *Lösung linearer Gleichungssysteme auf Parallelrechnern.* Vieweg, Braunschweig, 1990.

[Haa99] G. Haase. *Parallelisierung numerischer Algorithmen für partielle Differentialgleichungen. (Parallelization of numerical algorithms for partial differential equations).* Teubner Verlag Stuttgart, 1999.

[Hac91] W. Hackbusch. *Iterative Lösung großer schwach besetzter Gleichungssysteme.* Teubner, Stuttgart, 1991.

[HS52] M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Standards*, 49:409–436, 1952.

[Joh98] V. John. On the parallel performance of coupled multigrid methods for the solution of incompressible Navier-Stokes equations. In M. Griebel, O.P. Iliev, S.D. Margenov, and P.S. Vassilevski, editors, *Large-Scale Scientific Computations of Engineering and Environmental Problems*, volume 62 of *Notes on Numerical Fluid Mechanics*, pages 269 – 280. Vieweg, 1998.

[Joh99] V. John. A comparison of parallel solvers for the incompressible Navier-Stokes equations. *Comput. Visual. Sci.*, 4(1):193 – 200, 1999.

[Joh00] V. John. On the performance of smoothers in coupled multigrid methods for the solution of the incompressible Navier-Stokes equations on parallel computers. In M. Rahman and C.A. Brebbia, editors, *Advances in Fluid Mechanics III*, pages 181 – 190. WIT Press, 2000.

[JT00] V. John and L. Tobiska. Smoothers in coupled multigrid methods for the parallel solution of the incompressible Navier-Stokes equations. *Int. J. Num. Meth. Fluids*, 33:453 – 473, 2000.

[Saa96] Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Company, 1996.

Index