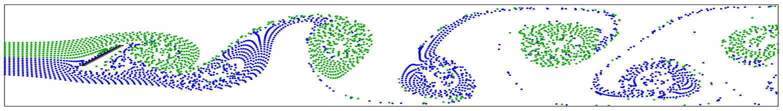
Institut für Angewandte Mathematik der Universität Bonn
Abteilung Wissenschaftliches Rechnen und Numerische Simulation
Professor Dr. Michael Griebel

Praktikum Sommersemester 2001

Numerische Simulation und Visualisierung

- Leitung: *Prof. Dr. M. Griebel*, Zi. 604, Tel. 73 3437
- Mitarbeiter: *F. Koster*, Wegelerstr. 4, Tel. 73 3173



Blatt 3

1. Zur Parallelisierung

Im folgenden wenden wir uns der Parallelisierung unseres Programms zur Strömungssimulation zu. Dabei wird eine Verkürzung der Gesamtrechenzeit dadurch erreicht, daß die zu leistenden Berechnungen auf mehrere Prozessoren verteilt werden und somit, zumindest zu einem bestimmten Grad, gleichzeitig ausführbar werden. Darüberhinaus steht einem bei Parallelrechnern meist ein wesentlich größerer Hauptspeicher zur Verfügung als bei sequentiellen Maschinen.
Vor jeglicher Parallelisierung sollte zuerst einmal ein möglichst effizientes sequentielles Verfahren erstellt werden, welches dann durch Parallelisierung weiter beschleunigt werden kann. Eine solche rein numerische Effizienzsteigerung wäre etwa durch den Einsatz von Mehrgitterverfahren bei der Lösung der Druckgleichung, durch die Implementierung einer (semi-)impliziten Behandlung der Impulsgleichungen oder durch den Einsatz verbesserter Diskretisierungsverfahren (Ansätze höherer Ordnung, adaptive Verfeinerung) zu erreichen.
Die Verbesserung unseres Algorithmus sowie deren anschließende Parallelisierung ist jedoch keine leichte Aufgabe und übersteigt den Rahmen dieses Praktikums. Deshalb betrachten wir hier der Einfachheit halber exemplarisch die Parallelisierung des bestehenden Algorithmus von Blatt 1 und Blatt 2, bei dem zur Lösung der Druckgleichung lediglich die SOR-Iteration eingesetzt wird.

2. Parallelrechner und Programmierumgebungen

Neben den Vektorrechnern (etwa Cray's YMP, Fujitsu's S600 oder der SX-3R von NEC), bei denen die Steigerung der Leistungsfähigkeit durch das pipelineartige Abarbeiten gleichartiger arithmetischer Befehle für die in einem Vektor gespeicherten Daten erreicht wird, haben sich in jüngster Zeit immer mehr „echte“ *Parallelrechner* mit einigen Dutzend bis hin zu über

Tausend leistungsfähigen Prozessoren (meist vom RISC-Typus) durchgesetzt, die gleichzeitig rechnen können und zudem miteinander kommunizieren müssen.
Die Rechner der sogenannten MIMD-Klasse sind durch den sogenannten **m**_ultiple_ **i**_nstruction stream_ sowie den sogenannten **m**_ultiple_ **d**_ata stream_ gekennzeichnet, d.h. die Menge der auszuführenden Operationen ist verteilt auf mehrere „Ströme“, wird also auf die vorhandenen Prozessoren aufgeteilt.[1] Genauso wird mit der Menge der zu verarbeitenden Daten verfahren. Auch sie sind auf mehrere „Ströme“ und somit auf die vorhandenen Prozessoren verteilt. In Bezug auf den physikalischen Speicher haben sich in der Vergangenheit zwei Typen von MIMD-Rechnern am Markt etabliert. Man unterscheidet zwischen Rechnern mit verteiltem Speicher (MIMD distributed address space machines), auch „message passing machines“ genannt, und Rechnern mit globalem Speicher (MIMD global address space machines), auch „shared memory machines“ genannt.[2] [3]
Beispiele für *distributed memory machines* sind etwa Intel's iPSC/860, die Paragon XP/S, der nCube-2, Transputersysteme wie Parsytec's GC, Meiko's CS-2, IBM's SP1 und SP2 oder die CM-5 der mittlerweile nicht mehr existierenden Firma Thinking Machines. Aber auch Cluster von Arbeitsplatzrechnern, die durch Netzwerke (Ethernet, HIPPI, FDDI oder ATM-Switch) miteinander verbunden sind, zählen hierzu. Diesen Maschinen gemeinsam ist die verteilte Programmierung, d.h. die parallel auszuführenden Programme werden als Prozesse oder Tasks auf jeden Prozessor geladen. Die Datenallokierung geschieht lokal auf jedem Prozessor, und der Datenaustausch zwischen den einzelnen Prozessen ist durch explizite Befehle (send, receive) zu programmieren, die damit, ähnlich wie beim in der Informatik bekannten Erzeuger-Verbraucher-Problem, neben dem Datenaustausch zugleich die Synchronisation der Prozesse bewirken.
In diesem Zusammenhang interessieren uns insbesondere Netzwerke von Arbeitsplatzrechnern, die über Ethernet gekoppelt sind. Mit Hilfe spezieller Kommunikationsplattformen wie PVM, Parmacs, Linda, Express, P4 oder MPI lassen sich auch solche Netze als ein paralleles Rechensystem nutzen und bieten aufgrund ihrer allgemeinen Verfügbarkeit und ihres guten Preis-Leistungsverhältnisses mittlerweile eine echte Alternative zu konventionellen MIMD-Rechnern. Dabei ist es nicht nur möglich Rechner gleicher Bauart zusammenzuschließen. Die Rechner können die verschiedensten Architekturen haben. Außerdem ist es nicht nötig, daß alle Rechner an einem lokalen Netz hängen (LAN = Local Area Network), sondern sie dürfen weit verstreut stehen (WAN = Wide Area Network). Mit den sich in den nächsten Jahren entwickelnden Datenautobahnen kann es möglich werden, Rechner aus der ganzen Welt zu einem virtuellen Parallelrechner zusammenzuschließen. So gibt es heute schon Versuche, Rechner über das Internet zu verknüpfen.

[1] Daneben existieren auch Rechner des SIMD-Typs (single instruction multiple data). Sie bestehen aus array-artig angeordneten, synchron getakteten, schwachen Prozessoren, deren Befehlsvorrat sehr klein ist, die aber einzeln auch sehr billig sind. Dabei bringt die Masse der Prozessoren die relativ große Rechenleistung. Ein Beispiel hierfür ist etwa der Rechner der Firma MasPar, mit bis zu 65536 Prozessoren.

[2] Die verschiedenen Prozessoren arbeiten dann parallel auf einem gemeinsamen Speicher. Ein Beispiel hierfür ist der Power Challenge der Firma Silicon Graphics.

[3] Eine Klassifikation der verschiedenen Parallelrechner-Typen kann mittels des Erlanger Klassifikations-Schemas durchgeführt werden, vgl. Bode, Händler, *Rechnerarchitektur I,II*, Springer, 1983

Die zur Zeit leistungsfähigsten Parallelrechner[4] sind aber parallel geschaltete Vektorrechner, wie der *Numerische Windtunnel* des National Aerospace Laboratory in Japan, der aus 140 Prozessoreinheiten VPP500 mit je 1.7 GFlops Peak Performance und 256 MByte Hauptspeicher besteht. Es ergibt sich dann ein Rechner mit einer Leistung von 238 GFlops und einem Hauptspeicher von 36 GByte.
In jüngster Zeit wird mit dem Ansatz des sogenannten *virtual shared memory*-Konzepts versucht, die explizit zu programmierende Datenverteilung auf mehrere Prozessoren und die Notwendigkeit des explizit zu programmierenden Datenaustausches vom Benutzer abzuschirmen. Die Speicher aller Prozessoren werden dem Benutzer als ein zusammenhängender und global adressierbarer Datenraum angeboten, also als ein großer Speicher, auf den alle Prozessoren des Rechners zeitgleich zugreifen können. Beispiele hierfür sind die KSR-1 und deren Nachfolgemodell KSR-2 der nicht mehr existierenden Firma Kendall Square, die MPP-1 von Convex oder die T3D von Cray. Dabei sorgt das Betriebssystem gemäß Benutzerdirektiven für die automatische Datenverteilung auf die Speicherstücke der verschiedenen Prozessoren. Die explizite Programmierung der Zuordnung von Prozessen auf Prozessoren, die lokale Speicherallokation und die Kommunikation zwischen den Prozessen in zu programmierenden Datenaustauschschritten entfällt dabei weitgehend. Diese Bequemlichkeit bei der parallelen Programmierung wird allerdings mit einem gewissen Verlust der Transparenz und Effizienz des Berechnungsablaufs erkauft. Die Effizienz ist dabei stark von den verwendeten Caching-Mechanismen abhängig, deren Entwicklungsphase noch nicht abgeschlossen ist. Handoptimierte Programme sind im allgemeinen schneller.
Ein bisher großes Problem im Bereich des Parallelrechnens ist die Vielzahl der Programmierumgebungen und -sprachen, die fast so zahlreich wie die Parallelrechner selbst sind. Während sequentielle Programme nach dem Übersetzen meist auch auf anderen Maschinen laufen, ist es sehr aufwendig, ein paralleles Programm von einem Rechner auf den anderen zu portieren. Die immer schneller werdende Entwicklung neuer Parallelrechner und die große Fluktuation auf dem Herstellermarkt hat dieses Problem noch verstärkt. Außerdem sind die parallelen Programme häufig auf die speziellen Architekturen der Parallelrechner zugeschnitten.
Es ist daher unerläßlich, Standards, hauptsächlich im Bereich des Message Passing, festzulegen. Dazu wurde in letzter Zeit an MPI (Message Passing Interface) gearbeitet, das auf bekannten Kommunikationsbibliotheken wie PVM, Parmacs und Express basiert.

3. Gebietszerlegung als Parallelisierungsstrategie

Bei der Parallelisierung numerischer Algorithmen für partielle Differentialgleichungen bietet es sich an, das Gesamtgebiet Ω in Teilgebiete $\Omega_1 \dots \Omega_N$ zu zerlegen und auf jedem Prozessor ein Teilgebiet zu behandeln. Das erste sogenannte Gebietszerlegungsverfahren ("domain decomposition method") wurde 1870 von H.A. Schwarz entwickelt. Deswegen spricht man bei Gebietszerlegungsverfahren auch von Schwarzschen Methoden. Man kann die Gebietszerlegungsverfahren in überlappende und nicht–überlappende Varianten untergliedern, je nachdem ob sich die jeweiligen Teilgebiete überlagern oder nicht (vgl. Abbildung 3.1).

[4] Stand: 1995, aktuelle Übersicht unter *www.top500.org*

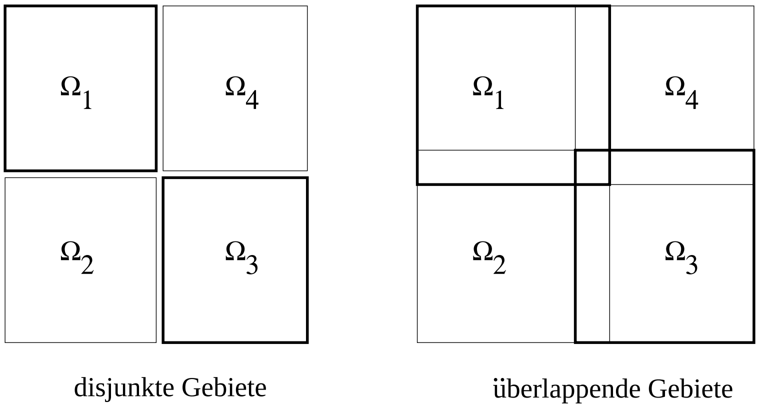


Abb. 3.1: Gebietszerlegungsmethoden

Jedes Teilgebiet wird nun einem Prozeß zugeordnet, der die Unbekannten, die in dem entsprechenden Gebiet liegen, berechnet. Jeder einzelne Prozeß muß also nicht mehr auf die ganzen Datenfelder zugreifen können, sondern nur noch auf einen Teil, und die Rechenoperationen eines iterativen Lösungsverfahrens werden auf die einzelnen Prozesse verteilt ausgeführt. Werden nun die Prozesse auf unterschiedliche Prozessoren verteilt, so kann der Algorithmus parallel abgearbeitet werden. Im Speicher der einzelnen Prozessoren liegen dann jeweils nur noch die Daten, die von den dort laufenden Prozessen benötigt werden. Neben der Beschleunigung der Rechenzeit durch die parallele Bearbeitung bringt die Parallelisierung meist einen weiteren Vorteil: Es steht wesentlich mehr Speicherplatz als bei der seriellen Berechnung auf einer Maschine zur Verfügung.
Um eine möglichst optimale Beschleunigung zu erzielen, muß jeder Prozessor annähernd gleich belastet werden. Dies gelingt in unserer Anwendung dann, wenn die Teilgebiete, in die das Gesamtgebiet zerlegt wurde, annähernd gleich gleich groß sind und somit etwa gleich viele Unbekannte beinhalten und wenn auf jedem Prozessor etwa gleich viele Prozesse laufen. Um die Konvergenz des Gesamtalgorithmus zu gewährleisten, ist zudem zu bestimmten Zeitpunkten der Austausch rechenrelevanter Daten zwischen Prozessen, die benachbarte Gebiete bearbeiten, nötig. Dabei ist darauf zu achten, daß nur *notwendige* Daten in möglichst *wenig* Kommunikationsschritten ausgetauscht werden, da der Verbindungsaufbau zwischen den Prozessen und der Datenaustausch auf den meisten Parallelrechnern relativ viel Zeit im Vergleich zur Zeit für die Ausführung von Rechenoperationen benötigt. Ein erhöhter Kommunikationsaufwand eines parallelisierten Algorithmus kann die durch die Parallelisierung der Berechnungen bewirkte Effizienzsteigerung in gewissem Maß reduzieren oder sogar vollständig zunichte machen.

4. Parallelisierung des Strömungsprogramms

Wir wollen uns jetzt der Parallelisierung des in den Blättern 1 und 2 beschriebenen sequentiellen Programms zuwenden. Dazu teilen wir das Grundgebiet $\mathcal{R}$ für die Strömungsberechnung entlang der Gitterlinien in i_{proc} Teile in x-Richtung und j_{proc} Teile in y-Richtung auf, so daß wir insgesamt $i_{proc} \cdot j_{proc}$ rechteckige Teilgebiete $\Omega_{ip,jp}$, $ip = 1, \ldots, i_{proc}$, $jp = 1, \ldots, j_{proc}$ erhalten. In Abbildung 4.1 ist diese Aufteilung dargestellt, wobei die Gitterlinien des gestaffelten

Gitters gestrichelt und die Grenzen der Gebiete mit durchgehenden Linien gekennzeichnet sind. In jedem Prozeß werden dann die Druck- und Geschwindigkeitswerte im Inneren des zugeordneten Teilgebietes berechnet. Um einen zusätzlichen Kommunikationsschritt zu sparen, werden die Geschwindigkeitswerte auf den Teilgebietsgrenzen jeweils von beiden Prozessen berechnet.

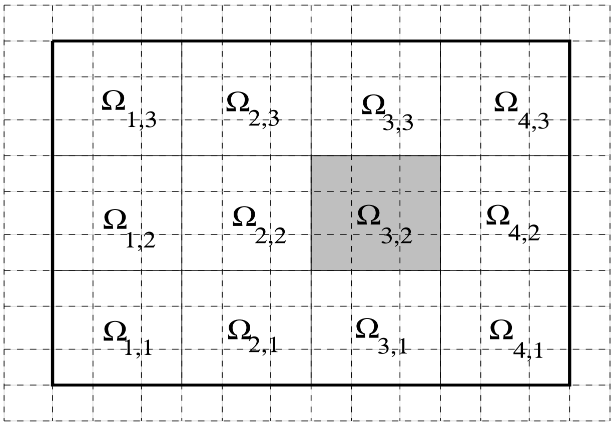


Abb.4.1: Aufteilung des Gebiets Ω (`iproc`=4, `jproc`=3)

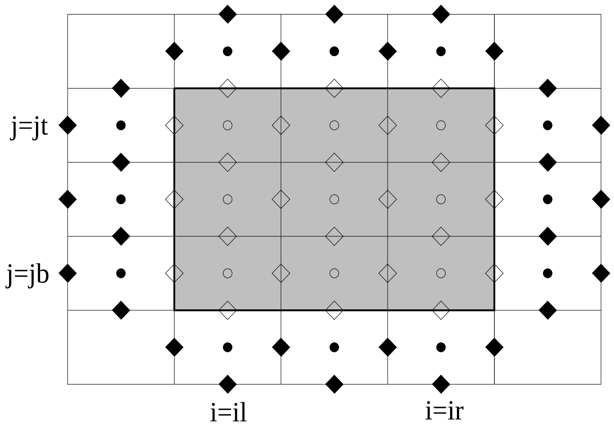


Abb. 4.2: Unbekannte und Randwerte für das Teilgebiet $\Omega_{3,2}$

Der Prozeß, der das Teilgebiet $[(il-1)\,\delta x, ir\,\delta x] \times [(jb-1)\,\delta y, jt\,\delta y]$ behandelt, benötigt dann die in Abbildung 4.2 gekennzeichneten Werte. Die an den weißen Punkten liegenden Werte müssen von dem jeweiligen Prozeß berechnet werden, während die Werte an den schwarzen Punkten als "Randwerte" nur zur Bestimmung der Werte im Inneren und auf dem Teilgebietsrand dienen. Diese "Randwerte" werden, sofern sie nicht jenseits des Randes des Gesamtgebietes $\mathcal{R}$ liegen, in den Prozessen berechnet, die die benachbarten Teilgebiete behandeln. Sie müssen also in geeigneten Kommunikationsschritten in der Zeitschleife von den benachbarten Prozessen versendet werden.
Nach jedem SOR–Schritt der Druckiteration müssen die Druckwerte der Randbordüren, wie in Abbildung 4.3 dargestellt, ausgetauscht werden, damit in den Teilgebieten mit den aktuellsten Randwerten weitergerechnet werden kann. Damit nicht zu viele Daten das Netz gleichzeitig belasten, wird der Datenaustausch in vier Teilschritten (nach links, nach rechts, nach unten, nach oben) durchgeführt. Nach dem Datenaustausch können dann auf den jeweiligen Prozessen die Teilsummen des Residuums berechnet werden, die anschließend alle an einen Prozeß (z.B. an den Hauptprozeß) gesendet und dort addiert werden. Dieser Prozeß entscheidet dann ob die Druckiteration abgebrochen wird (Toleranzgrenze ε unterschritten oder maximale Iterationszahl `itmax` erreicht) und verschickt eine Nachricht an alle anderen Prozesse, ob der nächste Iterationsschritt durchgeführt werden soll, oder die Druckiteration beendet wird.

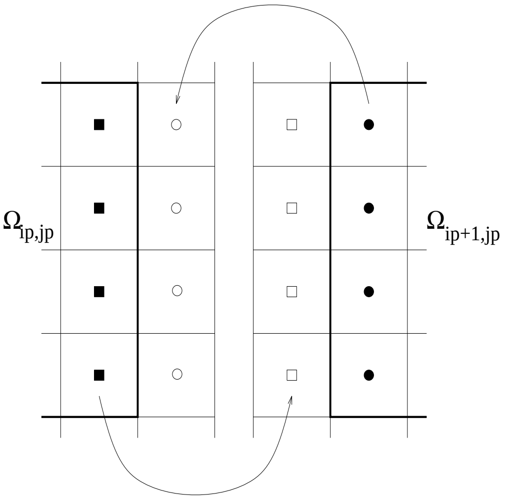


Abb. 4.3: Austausch der Druckwerte

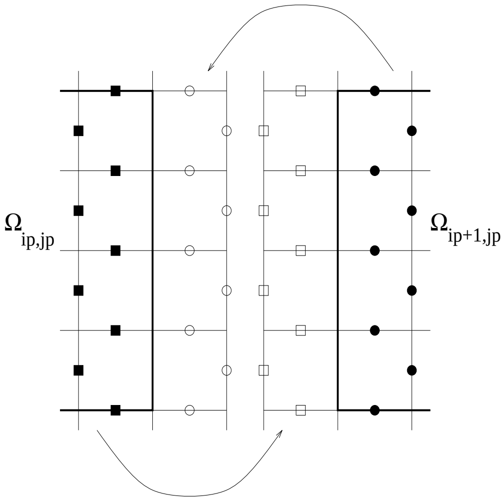


Abb. 4.4: Austausch der Geschwindigkeitswerte

Da nach Ende der Druckiteration in der Randbordüre die aktuellen Druckwerte stehen, können in `ADAP_UV` auch die Geschwindigkeitswerte auf dem Teilgebietsrand ohne einen neuen Kommunikationsschritt berechnet werden. Nach `ADAP_UV` müssen allerdings, wie in Abbildung 4.4 veranschaulicht, die Geschwindigkeitswerte zur Berechnung von F und G im nächsten Zeitschritt ausgetauscht werden.
Insgesamt erhalten wir (ohne freie Randwertprobleme und Zeitschrittweitensteuerung) den folgenden Algorithmus

Setze $t := 0$, $n := 0$
Belege u, v, p mit Anfangswerten
Solange $t < t_{end}$
    Setze die Randwerte für u und v
    Berechne $F^{(n)}$ und $G^{(n)}$ gemäß (Blatt1,(9),(10),(17))
    Berechne die rechte Seite der Druckgleichung (Blatt1,(11))
    Setze $it := 0$
    Solange $it < itmax$ und $\|r^{it}\| > eps$
        Führe einen SOR-Zyklus gemäß (Blatt1,(12)) durch
        Tausche die Druckwerte der Randbordüren aus
        Berechne die Teilresiduen und versende sie an den Hauptprozeß
        Der *Hauptprozeß* berechnet die Norm des Residuums der Druckgleichung $\|r^{it}\|$ und versendet diese an alle anderen Prozesse
        $it := it + 1$
    Berechne $u^{(n+1)}$ und $v^{(n+1)}$ gemäß (Blatt1,(7),(8))
    Tausche die Geschwindigkeitswerte der Randbordüren aus
    $t := t + \delta t$
    $n := n + 1$

5. Realisierung auf einem Netzwerk von Arbeitsplatzrechnern unter MPI

Wir werden zur Parallelisierung die Kommunikationsbibliothek MPI (`Message Passing Interface`) auf einem Netz gleichartiger Workstations verwenden. Wir haben damit ein homogenes Rechensystem mit verteiltem Speicher zur Verfügung, weswegen die Menge der Daten geeignet auf die einzelnen Prozessoren verteilt werden muß. Der nötige Datenaustausch zwischen den Prozessen geschieht in geeigneten Kommunikationsschritten.
Nachfolgend geben wir eine Kurzbeschreibung von MPI. Ausführliche Dokumentationen findet man unter der URL

`www.mpi.nd.edu/lam/`

Der Aufruf eines mit MPI parallelisierten Programms erfolgt durch:

```
mpirun -np <AnzahlProzesse> -machinefile <machines> <executable>
```

Die Datei `machines` ist vorher anzulegen und dort sind der Reihe nach (zeilenweise) alle Maschinen anzugeben auf denen Prozesse gestartet werden können. Die Benutzung der MPI-Routinen wird durch das Einfügen der Headerdatei `mpi.h` und durch das Compilieren und Linken unserer Objektdateien mit `mpicc` ermöglicht. Dazu ist im makefile `CC=mpicc` zu setzen. Das Skript `mpicc` ruft im Wesentlichen den `gcc` Compiler auf, siehe `mpicc -show`.
Anders als bei PVM müssen wir die jeweiligen Prozesse nicht selber anstoßen, sondern durch den Aufruf über

```
mpirun -np AnzahlProzesse -machinefile machines executable
```

wird auf einigen der in der Datei `machines` aufgeführten Maschinen jeweils das Programm `executable` gestartet. Dazu wird auf den Zielrechnern mittels `ssh` (`ssh2`) eine Verbindung aufgebaut. Je nach persöhnlicher Konfiguration der `ssh` kann deshalb eine mehrfache Passwortabfrage notwendig sein, siehe dazu auch S.20 des Installation Guide der bei uns benutzten MPICH Implementierung von MPI (*www.mcs.anl.gov/mpi/mpich/download.html*).
Weiterhin müssen die Prozesse anfänglich initalisiert und am Ende abgemeldet werden.

- `int MPI_Init(int *argc,char **argv[])`:
  Initalisert den Prozess mit den Übergabeparametern der Kommandozeile.
  Achtung: die Reihenfolge von Übergabeparametern kann verändert werden und es werden MPI-eigene Parameter herausgefiltert.

- `int MPI_Finalize(void)`:
  Bereitet den Prozess zur Terminierung vor.

Diese einzelnen Prozesse werden in MPI vermöge eines Gruppenkonzepts zusammengefaßt. Alle gestarteten Prozesse sind über eine *Communicator-Handle* `MPI_COMM_WORLD` ansprechbar. Aus dieser Gruppe von Prozessen kann jeder Prozess mittels seines eindeutigen Rangs identifiziert werden.

- `int MPI_Comm_size(MPI_Comm Comm_Handle,int *numberoftasks)`:
  legt die Anzahl der Prozesse in der Gruppe `Comm_Handle` in `*numberoftasks` ab

- `int MPI_Comm_rank(MPI_Comm Comm_Handle,int *myrank)`:
  legt den Rang des anfragenden Prozesses in `*myrank` ab, der Rang kann die Werte 0 bis einschließlich `numberoftasks`-1 annehmen

Anders als bei PVM, muß die Kommunikation nicht erst angemeldet und dann durchgeführt werden, sondern die Daten können ohne Ein- und Auspacken gesendet bzw. empfangen werden.

- `int MPI_Send(void *sendbuffer,int count,MPI_Datatype datatype, int destination,int tag, MPI_Comm Comm_Handle)`:
  sendet `count` Elemente des Typs `datatype` (durch Konstanten beschrieben : `MPI_INT`, `MPI_DOUBLE`, etc.) angefangen von der Adresse `sendbuffer` an den Prozess mit Rang `destination`

- `int MPI_Recv(void *recvbuffer,int count,MPI_Datatype datatype,int source, int tag, MPI_Comm Comm_Handle,MPI_Status *status)`:
  empfängt `count` Elemente des Typs `datatype` von dem Prozess mit Rang `source`, schreibt diese nach `recvbuffer`

Je nach Implementierung der Kommunikation in eurem Programm kann es sinnvoll sein, die Funktionen `MPI_ISend`, `MPI_IRecv` und `MPI_Wait` zu benutzen. Während bei `MPI_Send`, `MPI_Recv` gewartet wird, bis die Operation fertig ist, kehren die Funktionen `MPI_ISend`, `MPI_IRecv` auch dann zurück, wenn die eigentliche Kommunikationsoperation noch nicht beendet ist und im Hintergrund auf ihre Erledigung wartet (z.B. weil der empfangende Prozess gerade mit einer anderen Kommunikationsoperation blockiert ist). Das hat den Vorteil, daß entstehende Wartezeiten verringert werden können und daß auch *deadlocks* vermieden werden. Eine gestartete `MPI_ISend`, `MPI_IRecv` Operation *muß* mit `MPI_Wait` beendet werden.

Vermöge des Gruppenkonzeptes können auch komplexere Kommunikationen als *one-to-one* Prozesskommunikationen einfach durchgeführt werden. Beispielsweise kann ein Prozess an alle anderen in seiner Gruppe existierenden Prozesse senden (*one-to-many*), oder es können alle Prozesse einer Gruppe an einen bestimmten der Gruppe senden (*many-to-one*).

- `int MPI_Bcast(void *buffer,int count,MPI_Datatype datatype,int root, MPI_Comm Comm_Handle)`:
  alle Mitglieder der Gruppe `Comm_Handle` empfangen nach `buffer` `count` Elemente des Typs `datatype` von dem Prozess mit Rang `root`

- `int MPI_Reduce(void *sendbuffer,void *recvbuffer,int count, MPI_Datatype datatype,MPI_op operation,int root,MPI_Comm Comm_Handle)`:
  alle Mitglieder der Gruppe `Comm_Handle` senden beginnend mit `buffer` `count` Elemente des Typs `datatype` an den Prozess mit Rang `root`, dieser führt die Operation `operation` (durch Konstanten beschrieben : `MPI_SUM`, etc.) auf diesen Daten aus und schreibt diese nach `*recvbuffer`

Desweiteren bietet MPI die Verteilung der Prozesse einer Gruppe auf ein kartesisches Gitter, was unserem Ansatz der Gebietszerlegung entspricht, an. Damit vereinfacht sich die Bestimmung der Nachbarprozesse und somit die Auswahl der Prozesse, welche miteinander kommunzieren müssen.

- `int MPI_Cart_create(MPI_Comm Group_Handle,int cartdims,int *dims, int *periods,int reorder, MPI_Comm *Cart_Handle)`:
  verteilt die Prozesse der Gruppe `Group_Handle` auf ein `cartdims` dimensionales kartesisches Gitter, wobei jeweils in der `i`-ten Richtung `dims[i]` Prozesse eingeteilt werden, der Parameter `periods` beschreibt, ob die Prozesse auch periodisch vernetzt sein sollen, der Parameter `reorder` gestattet eine Änderung der Rangfolge der Prozesse, so daß mögliche Kommunikationswege optimal sind, die neue Gruppenbeschreibung wird in `*Cart_Handle` abgelegt

- `int MPI_Cart_coords(MPI_Comm Cart_Handle,int rank,int cartdims, int *coords)`:
  bestimmt die kartesischen Koordinaten `coords` des Prozesses mit dem Rang `rank` in der Gruppe `Cart_Handle`, `cartdims` bezeichnet die Länge von `coords`

- `int MPI_Cart_rank(MPI_Comm Cart_Handle,int *coords,int *rank)`:
  bestimmt den Rang des Prozesse mit den kartesischen Koordinaten `coords` in der Gruppe `Cart_Handle`

Als 'Nachteil' hierbei ergibt sich, daß die Anzahl der von uns gestarteten Prozesse mit der Anzahl der Zellen dieses Gitters übereinstimmen 'muß', d.h. `numberoftaksks` $= \prod_i$`dims[i]` soll gelten.

Änderungen im Code

Bei der Programmentwicklung muß nicht für jeden Prozeß ein eigenes Programm geschrieben werden. Vielmehr gibt es nur *ein* Programm mit variablen Gebietsgrenzen `il,ir,jb,jt`. Bei der Parallelisierung des Strömungsprogramms folgen wir dem in Abschnitt 4 beschriebenen Prinzip. Auf dem Prozessor, der das Teilgebiet $[(il-1)\,\delta x, ir\,\delta x]\times[(jb-1)\,\delta y, jt\,\delta y]$ behandelt, müssen dann die Felder die folgenden Dimensionen haben:

- `P` `[il-1,ir+1]`$\times$`[jb-1,jt+1]`
- `U` `[il-2,ir+1]`$\times$`[jb-1,jt+1]`
- `V` `[il-1,ir+1]`$\times$`[jb-2,jt+1]` $(*)$
- `F` `[il-2,ir+1]`$\times$`[jb-1,jt+1]`
- `G` `[il-1,ir+1]`$\times$`[jb-2,jt+1]`
- `RHS` `[il,ir]`$\times$`[jb,jt]`

Aufgabenstellung

1. Schreiben Sie die folgenden Prozeduren:

   - `void DRUCK_KOMM (REAL **P,int imax,int jmax,int il,int ir,int jb, int jt,int tl,int tr,int tb,int tt,REAL *bufP)`
     Die Druckwerte werden zwischen Prozessoren, die benachbarte Teilgebiete behandeln, ausgetauscht (siehe Abb. 4.3). Die folgende oder eine ähnliche Reihenfolge sollte eingehalten werden:
     nach links versenden — von rechts empfangen,
     nach rechts versenden — von links empfangen,
     nach oben versenden — von unten empfangen,
     nach unten versenden — von oben empfangen.
     Der Vektor `bufP` kann verwendet werden, wenn Daten, die nicht in einem Vektor stehen, in den Nachrichtenpuffer geschrieben werden müssen. Bemerkung: bei echten mehrdimensionalen Arrays, die einen einzigen großen Speicherblock belegen und z.B. mittels `U[i*128+j]` adressiert werden, gibt es auch die Möglichkeit sogenannter `Strided Vector Datatypes` bei denen das Umkopieren entfällt. Generell ist eines der Ziele bei der Entwicklung von MPI gewesen, eigentlich unnötige Kopieroperationen zu vermeiden.

     Der Aufruf erfolgt in `POISSON`, jeweils vor der Berechnung des Residuums.

   - `void UV_KOMM (REAL **U,REAL **V,int imax,int jmax,int il,int ir, int jb,int jt,int tl,int tr,int tb,int tt, REAL *bufU,REAL *bufV)`
     Die Geschwindigkeitswerte werden zwischen Prozessen, die benachbarte Teilgebiete behandeln, ausgetauscht (siehe Abb. 4.4). Wie in `DRUCK_KOMM` sollte der Datenaustausch in vier Schritten erfolgen und `bufU` sowie `bufV` haben die gleiche Funktion wie `bufP`.
     Der Aufruf erfolgt am Ende von `ADAP_UV`.

   - `void OUTPUT (...)`
     Schreiben Sie die Funktion `OUTPUT` so um, daß die auf die verschiedenen Prozesse verteilten Daten in der *richtigen Reihenfolge* in eine Datei geschrieben werden, die dann zur Visualisierung mit MatLab verwendet werden kann. Eine gute Strategie ist es, daß nur der 0te Prozess die Datei öffnet und schreibt und daß die anderen Prozesse z.B. zeilenweise die Daten an den 0ten Prozess schicken. Bei vielen Parallelrechnern ist die MPI-Kommunikation sehr schnell, file-IO wird *intern* jedoch oft nur über einen Prozessor abgwickelt, so daß der Versuch alle Prozesse auf ein file schreiben zu lassen zu einem Flaschenhals führt. Zu vermeiden ist jedoch, daß der 0te Prozess *alle* Daten auf einmal speichern muss!

2. Führen Sie in den Prozeduren von Blatt 1 die folgenden Modifikationen durch:

   - In `main`:
     Bei der Speicherplatzallokierung haben die Felder die in $(*)$ angegebene Dimension.

- Alle Schleifen über i und j dürfen sich nur noch auf die jeweiligen Teilgebiete beziehen.
- In `SETBCOND` und `SETSPECBCOND`:
  Die entsprechenden Randwerte dürfen nur gesetzt werden, wenn der Teilgebietsrand mit dem Rand des Gesamtgebietes zusammenfällt oder ein Hindernis im Teilgebiet liegt.
- In `COMP_FG` und `POISSON`:
  Die Formeln für die Randwerte dürfen nur angewendet werden, wenn der Teilgebietsrand mit dem Rand des Gesamtgebietes zusammenfällt. In `POISSON` muß außerdem die Residuumsberechnung, wie in Abschnitt 4 erläutert, verändert werden.
- Fügen Sie den Aufruf der neu geschriebenen Prozeduren an den jeweils angegebenen Stellen ein.
- Testen Sie den parallelen Algorithmus am Beispiel der Driven Cavity mit den folgenden Parametern:

`imax = 100`	`jmax = 100`	`xlength = 10`	`ylength = 10`
`delt = 0.01`	`t_end = 1.0`	`tau = 0.5`	`del_vec = 2.0`
`eps = 0.01`	`omg = 1.7`	`alpha = 0.5`	`itermax = 100`
`GX = 0.0`	`GY = 0.0`	`Re = 10`	
`UI = 0.0`	`VI = 0.0`	`PI = 0.0`	

6. Leistungsmessung

Bei ausreichend feinem Gitter sowie einer genügend großen Zahl von Prozessoren lassen sich für das parallelisierte Programm beeindruckende Geschwindigkeitssteigerungen im Vergleich zum sequentiellen Programm erzielen. Diese Leistungssteigerung läßt sich mittels des sogenannten Speedups sowie der Effizienz ausdrücken. Diese Werte sind definiert als

- Speedup $S(p) := T(1)/T(p)$,
- parallele Effizienz $E(p) := T(1)/(p \cdot T(p)) = S(p)/p$,

wobei p die Anzahl der eingesetzten Prozessoren bezeichnet und $T(p)$ die benötigte Rechenzeit bei paralleler Berechnug auf p Prozessoren ist.[5]
Da üblicherweise das Workstationnetz nicht in *dediziertem* Modus, das bedeutet ausschließlich Ihrem Programm zur Verfügung steht, sondern abhängig von der Menge der aktuellen Benutzer mit anderen Prozessen um die Prozessoren konkurriert, wäre es notwendig, mehrere

[5] In C kann zur Zeitmessung der Befehl `int gettimeofday(struct timeval *tp, struct timezone *tzp)` (Definition des Prozedurkopfes in `time.h`) verwendet werden. Dabei besteht die Struktur `timeval` aus den zwei Komponenten `tv_sec`, der Anzahl der Sekunden seit dem 1. Januar 1970, und `tv_usec`, den zusätzlich seitdem vergangenen Mikrosekunden. In der Struktur `timezone` wird die lokale Zeitzone zurückgegeben. Die Funktion `gettimeofday` muß zur Zeitmessung einmal zu Beginn und einmal am Ende des zu messenden Programmteils aufgerufen werden, um anschließend die beiden gemessenen Zeiten voneinander abzuziehen.

Läufe zu absolvieren, u.U. auch mit dem `at`–Kommando nachts, um die Rechner möglichst allein benutzen zu können. Als Ergebnis nimmt man dann die schnellste erzielte Zeit, in der Hoffnung, daß in diesem zugehörigen Lauf das eigene Programm ungestört von anderen Prozessen abgearbeitet wurde.
Bei dem angegebenen Parametersatz handelt es sich um ein sehr kleines Problem, bei dem man durch die Parallelisierung noch keine Beschleunigung der Berechnung erzielt. Diese tritt erst bei Problemen ab etwa 25.000 Gitterzellen auf. Dann dauert die Berechnung auf wenigen Prozessoren aber so lange, daß sie den Rahmen des Praktikums übersteigt.