Parallelisierung am Beispiel ausgewählter Probleme aus der Computational Physics I und II.

Martin Hasenbusch

NIC/DESY-Zeuthen
Platanenallee 6, D-15738, Germany

e-mail: Martin.Hasenbusch@desy.de

Zusammenfassung

MPP-Rechner, PC-Cluster
Einführung in "Message Passing" mit MPI
Laplace- und Poissongleichung mit Überrelaxation
Monte-Carlo-Simulation des Isingmodells

1 Einleitung

Manche numerischen Probleme in der Physik benötigen mehr Rechenleistung und Speicherplatz als von PCs oder Workstations geliefert wird.

Ein Beispiel ist das Arbeitsgebiet des Lehrstuhls Computational Physics, die numerische Behandlung der Quantenchromodynamik. Aber auch die Untersuchung von komplexeren Problemen aus der klassischen Physik verlangt nach hoher Rechenleistung und viel Hauptspeicher.

Als Übungsbeispiel nehmen wir die Elektrostatik, die schon zum Ende der CP I behandelt wurde. Als zweites Beispiel betrachen wir das Isingmodell.

Der klassische Typ des Hochleistungsrechners ist der "Vektorrechner". Im letzten Jahrzehnt ist dieser Rechnertyp zum Teil von sogenannten massive parallelen Rechnern abgelöst worden.

Während bei dem Vektorrecher eine (oder wenige) sehr schnelle CPUs zum Einsatz kommt, ist die grundlegende Idee bei einem MPP-Rechner sehr viele CPUs mit einem leistungsfähigen Netzwerk zu verkoppeln. Die CPUs sind die gleichen, wie sie in Workstations und Servern zum Einsatz kommen.

Der am meisten verbreitete Rechner dieser Art ist die CRAY T3E (siehe http://www.cray.com/) Basis ist der 21164A Alpha Prozessor, der auch in Digital Workstations und Servern verwendet wurde. Die CRAY T3E hat bis zu 2176 CPUs. Im Zuse Rechenzentrum ist z.B. ein Recher mit 544 Prozessor installiert.

Mittlerweile haben PCs von Intel und AMD die RISC CPUs von Compaq, HP, SUN und IBM in der Geschwindigkeit eingehohlt. Damit sind PCs eine interessante Basis für den Bau eines preisgünstigen Parallelrechners.

Als Netzwerk bieten sich Fast und Gigabit Ethernet an. Als schnellere Alternativen gibt es Myrinet (http://www.myrinet.com) oder SCI (http://www.scali.com) oder (?). Einen solchen PC-cluster baut man selbst zusammen oder er wird komplett von einer Firma geliefert (z.B. http://www.fujitsu-siemens.de/loboea/hpc, http://www.cluster.megware.de/ ...).

Leider läuft ein Programm das wir für den PC oder eine Workstation geschrieben haben nicht (oder nicht schneller) auf dem Parallelrechner.

Bei gewöhnlichen C oder Fortranprogrammen laufen die einzelnen Rechenschritte in einer festen Reihenfolge ab. Um mit dem Parallelrechner effizient zu arbeiten, müssen aber viele Rechenschritte gleichzeitig ablaufen können.

Zum einen muss das Problem bzw. der verwendete Algorithmus dies erlauben, und zum anderen brauchen wir spezielle Software um die Parallelverarbeitung zu ermöglichen. Deshalb auch diese spezielle Vorlesung.

1.1 Message Passing

Eine sehr flexible Art der Parallelisierung ist das sogenannte Message Passing. Auf jedem Knoten des Rechner läuft eine Kopie des Programms. Die einzelnen Kopien kommunizieren, indem sie Datenpakete austauschen.

- PVM (Parallel Virtual Maschine) http://www.epm.ornl.gov/pvm/pvm_home.html
- MPI (Message Passing Interface) MPICH: http://www-unix.mcs.anl.gov/mpi/mpich/

Man bindet Funktionen bzw. Subroutinen aus der MPICH Bibliothek in ein C oder ein Fortranprogramm ein.

2 Laplace und Possiongleichung auf dem Gitter

Hier wollen wir nochmal kurz die verwendeten Algorithmen wiederholen und ein standard C-programm für den Überrelaxationsalgorithmus schreiben.

Die Elektrostatik ist durch die Maxwellgleichungen

$$\vec{\nabla} \cdot \vec{E} = \frac{\rho}{\varepsilon_0}, \tag{2.1}$$

$$\vec{\nabla} \times \vec{E} = 0 \tag{2.2}$$

und Randbedingungen festgelegt mit den üblichen Bezeichnungen (in MKSA Einheiten). Mit dem Ansatz eines Potentials

$$\vec{E} = -\vec{\nabla}\Phi \tag{2.3}$$

wird (2.2) gelöst, und wir haben die Poisson Gleichung

$$-\triangle\Phi = \frac{\rho}{\varepsilon_0} \tag{2.4}$$

bzw. die Laplace–Gleichung

$$\triangle\Phi = 0 \tag{2.5}$$

im leeren Raum. In diesem Kurs wird es hauptsächlich darum gehen, die Laplace–Gleichung mit Randbedingungen zu lösen.

2.1 Elektrostatische Energie

In der Elektrodynamik wird gezeigt, daß die Energieerhaltung gilt, wenn man dem elektrischen Feld die Energie U,

$$U = \frac{\varepsilon_0}{2} \int \mathrm{d}V \vec{E}^2 = \frac{\varepsilon_0}{2} \int \mathrm{d}V (\vec{\nabla}\Phi)^2 \tag{2.6}$$

zuschreibt, wo das Integral über das betrachtete Volumen V geht. So wie ein Teilchen im *statischen*, d. h. zeitunabhängigen, Fall am Potentialminimum ruht, so kann man auch das $\vec{E}$-Feld in diesem Fall finden, indem man das Minimum der Energie sucht. Dann muß U stationär sein unter Variationen $\Phi \to \Phi + \delta\Phi$ im

Inneren von V. Am Rand ist Φ durch die Randbedingungen fixiert[1], und kann nicht variiert werden. Dann gilt

$$0 \stackrel{!}{=} \delta U = \int_V \mathrm{d}V (\vec{\nabla}\Phi) \cdot \vec{\nabla}\delta\Phi = -\int_V \mathrm{d}V \delta\Phi \triangle \Phi \tag{2.7}$$

Hier wurde partiell integriert, wobei wegen $\delta\Phi|_{\partial V} = 0$ kein Oberflächenterm auftritt. Da $\delta\Phi$ beliebig ist, folgt hieraus die Laplace–Gleichung im Inneren von V. Weil (2.6) ein positives quadratisches Funktional in Φ ist, ist das durch die Laplace–Gleichung gegebene Extremum ein Minimum. Dieser Zugang zur Laplace–Gleichung ist günstig für die Diskretisierung und spätere numerische Behandlung.

2.2 Diskretisierung der Laplace–Gleichung

Ein kontinuierliches Feld wie Φ und $\vec{E}$ entspricht selbst über einem endlichen Volumen unendlich vielen Parametern. Solche Probleme sind mit endlichen Computern im strikten Sinne nicht zu behandeln. Wir wissen aber, daß physikalische Felder stetig sind und Experimente endliche Fehler haben, so daß es genügt, Felder für endlich viele genügend dicht gelegene Punkte zu kennen.

Wir führen ein Rechteckgitter ein und beschränken uns darauf $\Phi_{i,j}$ zu berechnen, wobei die Verbindung durch

$$\Phi_{i,j} = \Phi(\vec{r}_{i,j}), \ \vec{r}_{i,j} = (ih_x, jh_y) \tag{2.8}$$

gegeben ist. h_x, h_y sind Schrittweiten und kontrollieren die Feinheit der Auflösung in $x-$ und $y-$Richtung. Um eine gute Näherung ans Kontinuum zu erhalten, müssen sie natürlich klein sein gegen physikalische Längen im Problem, z. B. Ausdehnung des Volumens und Längen, über die die gestellten Randbedingungen variieren.

Die einfachste Näherung für den in der Energie auftretenden Gradienten von Φ ist nun

$$\begin{aligned} \partial_x \Phi &\simeq \frac{\Phi_{i+1,j} - \Phi_{i,j}}{h_x}, \\ \partial_y \Phi &\simeq \frac{\Phi_{i,j+1} - \Phi_{i,j}}{h_y}. \end{aligned} \tag{2.9}$$

Damit folgt für die Energie[2]

$$U = \frac{\varepsilon_0}{2} h_x h_y \sum_{i,j} \left\{ \left(\frac{\Phi_{i+1,j} - \Phi_{i,j}}{h_x} \right)^2 + \left(\frac{\Phi_{i,j+1} - \Phi_{i,j}}{h_y} \right)^2 \right\}, \tag{2.10}$$

[1] hier werden Dirichlet Randbedingungen betrachtet
[2] eigentlich Energie pro Längeneinheit in z-Richtung

wobei wir das Integral durch eine Riemann Summe genähert haben[3]. Der Summationsbereich muß passend zur Geometrie und den Rändern gewählt werden. Als einfaches Beispiel betrachten wir einen rechteckigen Bereich der Größe $L_x \times L_y$. Die *inneren* Gitterpunkte sind dann aufzuzählen mit

$$i = 1, \ldots, N_x,\ j = 1, \ldots, N_y,\ L_x = (N_x + 1)h_x,\ L_y = (N_y + 1)h_y, \tag{2.11}$$

und die Ränder haben $i \in \{0, N_x + 1\}$ oder $j \in \{0, N_y + 1\}$. Die Summe in (2.10) geht natürlicherweise über nächste Nachbar Paare, jedes einmal, einschließlich solcher, die einen Punkt auf dem Rand haben. In unserem Beispiel ist also zu summieren über $i = 0, \ldots, N_x$, $j = 0, \ldots, N_y$.

Nun können wir das wohldefinierte Problem betrachten, $\Phi_{i,j}$ für die inneren Punkte so zu wählen, daß U minimal wird bei festgehaltenen Rand-$\Phi_{i,j}$. Aus der Variationsableitung wird in der diskretisierten Form eine gewöhnliche partielle Ableitung, und wir erhalten eine Gleichung für jedes *innere* Paar (i, j),

$$\begin{aligned} 0 &= \frac{1}{\varepsilon_0 h_x h_y} \frac{\partial U}{\partial \Phi_{i,j}} = \qquad\qquad (2.12)\\ &= \frac{1}{h_x^2}(2\Phi_{i,j} - \Phi_{i+1,j} - \Phi_{i-1,j}) + \frac{1}{h_y^2}(2\Phi_{i,j} - \Phi_{i,j+1} - \Phi_{i,j-1}) =: (-\triangle\Phi)_{i,j}. \end{aligned}$$

Hier wurde die diskrete Version des Laplace–Operators definiert. Es ist leicht zu sehen, daß dies für glatte Funktionen und kleine h_x, h_y die Kontinuumsform nähert. Weiter zeigt Taylor Entwicklung, daß die Abweichung vom Kontinuum $\propto h_x^2, h_y^2$ geht (vgl. numerische Ableitung). In dieses System von $N_x N_y$ Gleichungen gehen auch die Randwerte (bis auf Eckpunkte) mit ein. Sie müssen schließlich die Lösung festlegen. Grundsätzlich handelt es sich um eine Gleichung vom linearen Typ

$$A\Phi = \Phi_R, \tag{2.13}$$

wobei man sich alle inneren $\Phi_{i,j}$ links als eine Spalte denkt und Φ_R durch Randwerte gegeben ist. Alle Verfahren zur Lösung linearer Gleichungen könnte man also verwenden. Die im folgenden diskutierten Verfahren sind jedoch für dieses Problem effektiver und nutzen aus, daß die Matrix A dünn besiedelt ist, d. h. nur nahe der Diagonalen von Null verschiedenene Matrixelemente hat. Das liegt daran, daß in (2.12) jedes Φ nur mit den 4 Nachbarn verknüpft wird.

2.3 Gauß-Seidel-Iteration

Die Idee des Gauß-Seidel-Verfahrens besteht darin, immer wieder U als Funktion *eines* $\Phi_{i,j}$ zu minimieren. Man hat eine Feldkonfiguration im Speicher und einen zugehörigen U-Wert, besucht alle inneren Gitterplätze und setzt das jeweilige $\Phi_{i,j}$

[3] vgl. Definition Integral als Limes solcher Summen

auf den Wert, der U minimiert als Funktion dieser Variablen bei festgehaltenen übrigen.

Wir setzen nun, um einfachere Formeln zu haben, $h_x = h_y = h$. Dann kann man für einen beliebigen, aber fest gewählten inneren Gitterpunkt (i, j) die Terme in (2.10) folgendermaßen organisieren,

$$U = 2\varepsilon_0(\Phi_{i,j} - \bar{\Phi}_{i,j})^2 + \tilde{U}_{i,j}, \tag{2.14}$$

wobei $\tilde{U}_{i,j}$ nur Terme enthält, die von dem betrachteten $\Phi_{i,j}$ nicht abhängen, und

$$\bar{\Phi}_{i,j} = \frac{1}{4}(\Phi_{i+1,j} + \Phi_{i-1,j} + \Phi_{i,j+1} + \Phi_{i,j-1}) \tag{2.15}$$

über die Nachbarn mittelt. Das Gauß-Seidel (GS)-Verfahren setzt einfach jeweils

$$\Phi_{i,j} \to \bar{\Phi}_{i,j} \tag{2.16}$$

Das Verfahren ist sehr opportunistisch, jeder Feldwert setzt sich auf den Mittelwert seiner Nachbarn! Das konvergiert nicht einfach exakt nach einem Durchgang durchs Gitter, da ja dann die Nachbarn auch nicht mehr sind, was sie mal waren und (2.12) *nicht* überall erfüllt ist. Dennoch konvergiert das Verfahren asymptotisch, was dadurch plausibel ist, daß U während der Iteration monoton fällt. Gestartet wird die Iteration von einer im Prinzip beliebigen Anfangs-Feldkonfiguration, die die Randbedingungen erfüllt, wobei aber eine Schätzung des Resultats, soweit erhältlich, zu schnellerer Konvergenz führt und eingesetzt werden sollte.

2.4 Jacobi–Verfahren

Das Jacobi-Verfahren unterscheidet sich von Gauß-Seidel dadurch, daß man zwei komplette Konfigurationen im Speicher hält, Φ^{alt} und Φ^{neu}. Dann wird wieder die Gleichung (2.16) benutzt, wobei auf dem ganzen Gitter links Φ_{neu} konstruiert wird und rechts Φ_{alt} verwendet wird, also

$$\Phi^{neu}_{i,j} = \frac{1}{4}(\Phi^{alt}_{i+1,j} + \Phi^{alt}_{i-1,j} + \Phi^{alt}_{i,j+1} + \Phi^{alt}_{i,j-1}), \tag{2.17}$$

für alle inneren (i, j). Danach nimmt man Φ^{neu} als neues Φ^{alt} und iteriert weiter.

2.5 Sukzessive Überrelaxation

Im GS Schritt (2.16) kann man einen Parameter einschmuggeln,

$$\Phi_{i,j} \to \Phi_{i,j} + \omega(\bar{\Phi}_{i,j} - \Phi_{i,j}). \tag{2.18}$$

Der zweite Term ist als Korrekturterm aufzufassen, und GS entspricht $\omega = 1$. Für die Lösung verschwindet die Korrektur und sie ist ein Fixpunkt für alle Werte ω. Die Energie ändert sich nun in

$$U \to U + 2\varepsilon_0 \left[(\omega - 1)^2 - 1\right] (\bar{\Phi}_{i,j} - \Phi_{i,j})^2. \tag{2.19}$$

Damit wird sie abgesenkt für alle Werte $0 < \omega < 2$. Interessant ist der Bereich der Überrelaxation (Successive Over-Relaxation = SOR), $1 < \omega < 2$. Hier wird "überkorrigiert" über das Minimum der Parabel (als Funktion eines $\Phi_{i,j}$) hinaus. Die lokale Energieabsenkung ist nicht die maximal mögliche. Dennoch ist SOR äußerst profitabel, wenn ω geeignet gewählt wird. Genauer gesagt kann man dann $\tau \propto h^{-1}$ (statt h^{-2}) erreichen, ein Gewinnfaktor von $\mathrm{O}(N)$ auf einem $N \times N$ Gitter! Leider ist diese Beschleunigung schwer plausibel zu machen. Die Überkorrektur berücksichtigt in gewisser Weise schon Verbesserungen der Lösung, die sonst im nächsten Schritt kämen und zieht auch die Nachbarn, wo ja der neue Wert dann eingeht, mit.

3 Parallelisierung

Die grobe Idee bei der Parallelisierung ist es, dass man das Gitter in gleich grosse (für den Fall dass alle Prozessoren die gleiche Rechenleistung haben) Teilgitter zerlegt. Jeder Knoten des Rechners bekommt ein Teilgitter zugeordnet.

Der Einfachheit halber wollen wir zunächst den Fall betrachten, dass das Gitter nur in einer Richtung unterteilt wird. (o.B.d.A. x-Richtung)

Die Zahl der Knoten sei p. Dann bekommt jeder Knoten $n_x \times N_y$ innere Punkte des Gitters. Dabei ist $n_x = N_x/p$. Für die Nummerierung in den Teilblöcken führen wir den Index $\tilde{i}$ ein. Für die inneren Punkte ist $\tilde{i} \in \{1, 2, ..., n_x\}$. Der globale Index ergibt sich aus $i = kn_x + \tilde{i}$, wobei die Knoten mit $k \in \{0, 1, ..., p-1\}$ nummeriert sind.

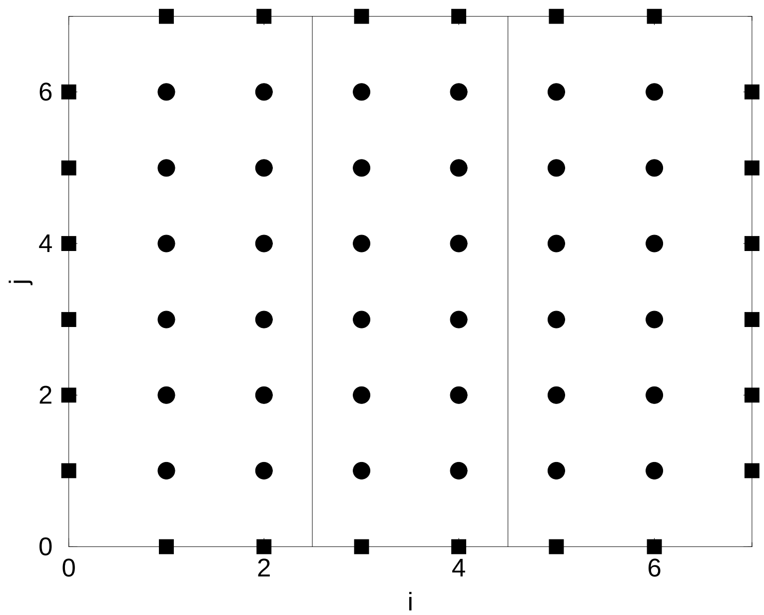
Zunächst wollen wir die Parallelisierung des Jacobi-Verfahrens betrachten. Dabei beobachten wir, dass die Berechnung der Φ^{neu} an allen Gitterpunkten unabhangig von den Werten von Φ^{neu} an anderen Gitterpunkten erfolgt.

Allerdings benötigen wir für die Berechnung der Φ^{neu} am Rand der Teilgitter, die Werte von Φ^{alt} vom angrenzenden Rand der benachbarten Teilgitter. In unserem C-programm ist es daher sinnvoll die arrays für Φ^{neu} und Φ^{alt} wie folgt zu wählen:

```
double phialt[nx+2][Ny+2],phineu[nx+2][Ny+2];
```

Dabei ist beim Knoten 0 $\tilde{i} = 0$ für die Randbedingungen vorgesehen, und $\tilde{i} = n_x + 1$ für die Kopie von $\tilde{i} = 1$ auf dem Knoten 1. Bei Knoten $0 < k < p - 1$ ist $\tilde{i} = 0$ für die Kopie von $\tilde{i} = n_x$ auf Knoten $k - 1$ und $\tilde{i} = n_x + 1$ für die Kopie von $\tilde{i} = 1$ auf Knoten $k + 1$ vorgesehen. Beim Knoten $p - 1$ liegt auf $\tilde{i} = 0$ die Kopie von $\tilde{i} = n_x$ von Knoten $p - 2$; auf $\tilde{i} = n_x + 1$ liegen die Randbedingungen.

Abbildung 1: Beispiel mit 6×6 inneren Punkten. Die inneren Punkte sind mit Kreisen dargestellt. Die Randpunkte mit Quadraten. Im Beispiel wird mit 3 Knoten gerechnet. $i = 1, 2$ ist für den Knoten 0, $i = 3, 4$ für den Knoten 1 und $i = 5, 6$ für den Knoten 2 bestimmt.



Nachdem wir diskutiert haben wie die Daten verteilt sind, wollen wir nun betrachen, wie Rechenschritte und Kommunikation aufeinander folgen.

1)
-Initialisierung von Φ^{alt}
-Austausch der Daten auf den Rändern mit Message Passing

2) Iteration von:
- Berechnung von $\Phi^{neu}_{\tilde{i}j} = \frac{1}{4}(\Phi^{alt}_{\tilde{i}-1,j} + \Phi^{alt}_{\tilde{i}+1,j} + \Phi^{alt}_{\tilde{i},j-1} + \Phi^{alt}_{\tilde{i},j+1})$
für $1 \leq \tilde{i} \leq n_x$, $1 \leq \tilde{j} \leq N_y$
- Wir kopieren $\Phi^{neu}_{\tilde{i}j}$ auf $\Phi^{alt}_{\tilde{i}j}$ für $1 \leq \tilde{i} \leq n_x$, $1 \leq \tilde{j} \leq N_y$
(Für gute Performance ist es günstiger einfach die Rolle von Φ^{neu} und Φ^{alt} zu vertauchen und damit nichts explizit zu Kopieren. Hier wollen wir aber zunächt der Einfachheit des Programms den Vorrang geben)
-Austausch der Daten auf den Rändern mit Message Passing

3.1 Erste Schritte mit MPI

Mit

```
>mpicc beispiel.c -lm -o beispiel.exe
```

können wir ein Programm das Funktionen aus der MPI-bibliothek enthält compilieren.
z.B mit

```
>mpirun -np 4 beispiel.exe
```

wird das ausführbare Programm beispiel.exe auf 4 Knoten gestartet.

Funktionen die zur Initialisierung der Kommunikation nötig sind:

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

int main(int argc,char *argv[])
{
    int done = 0, n, myid, numprocs, i;
    int  namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);

...

    MPI_Finalize();
    return 0;
}
```

Alle MPI-funktionen fangen mit

```
MPI_
```

an. Konstanten wie

```
MPI_COMM_WORLD
```

sind in mpi.h definiert.

Mit

```
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
```

erfährt jeder Prozess wie viele Prozesse an der Parallelrechnung teilnehmen. Mit

```
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

bekommt jeder Prozess eine ihm eigene "Hausnummer" mitgeteilt. Schliesslich erfährt der Prozess mit

```
MPI_Get_processor_name(processor_name,&namelen);
```

auf welchem Knoten erläuft. Das ist besonders dann interessant, wenn es Probleme mit der Hardware oder der Systemsoftware gibt.

Als nächstes betrachten wir Funktionen, die Datenpakete zwischen den einzelnen Prozessen austauschen können:

Mit

```
MPI_Send(&phialt[n_x][1], Ny, MPI_DOUBLE,myid+1, 83, MPI_COMM_WORLD);
```

kann ein Datenpaket zu dem Prozess `myid+1` geschickt werden. Dabei gibt `MPI_DOUBLE` den Typ der Daten welche verschickt werden sollen (hier double precision floating point Zahl, `Ny` die Anzahl der Daten.
&`phialt[n_x][1]` gibt die Adresse der ersten Zahl die verschickt werden soll. 83 ist hier das Label für den Sendevorgang. Das entsprechende recieve bekommt das gleiche Label.

Neben `MPI_DOUBLE` gibt es unter anderem noch die Datentypen `MPI_CHAR`, `MPI_BYTE`, `MPI_SHORT`, `MPI_INT`, `MPI_FLOAT` mit den naheligenden Bedeutungen.

Das entsprechene recieve:

```
MPI_Recv(&phialt[0][1], Ny, MPI_DOUBLE, myid-1, 83, MPI_COMM_WORLD, &status);
```

Hier gibt &`phialt[0][1]` die Zieladresse des Datenpakets.

3.2 Mehr zu Send und Recieve

Allgemein ist bei MPI garantiert (bei korrekter Implementierung), dass sich Nachrichten nicht überholen können. I.e. schickt Prozess i eine Reihe von Datenpaketen an Prozess j, so ist garantiert, dass sie auch in der Gleichen Reihenfolge empfangen werden.

Bei der "point-to-point" Kommunikation muss man zunächt zwischen "blocking" und "non-blocking" und darüber hinaus zwischen verschieden Modi der Kommunikation unterscheiden. Die wesentlichen Unterschiede zwischen diesen Formen der Kommunikation liegen darin, zu welchem Zeitpunkt die MPI-Funktion die Kontrolle an das rufende Programm zurückgibt.

3.2.1 blocking send and recieve

Die bisher besprochenen `MPI_Send` und `MPI_Recv` sind "blocking" und haben den Modus "standard". Die Modi sind

- standard
- synchronous
- buffered
- ready

Die "synchronous" und "blocking" Funktionen von MPI heissen `MPI_SSend` und `MPI_SRecv`. "synchronous" bedeuted, dass `MPI_SSend` die Kontrolle an das rufende Programm zurückgibt, wenn es von dem zugehörigen `MPI_SRecv` den Empfang des Datenpakets bestätigt bekommt.

Die "buffered" und "blocking" Funktionen von MPI heissen `MPI_Bsend` und `MPI_Brecv`. Hier wird das Datenpaket zunächst in einen "Buffer" gelegt. Dieser Buffer muss vom Benutzerprogramm selbst angelegt werden. Dazu dient `int MPI_Buffer_attach( void *buffer, int size )` bzw. `int MPI_Buffer_detach(void *bufferptr, int *size )`. `MPI_Bsend` gibt die Kontrolle zurück, sobald das Datenpaket in den Buffer geschrieben ist.

Der Modus "standard" liegt irgendwo zwischen "synchronous" und "buffered". Was genau passiert, hängt von der Implementierung von MPI ab. Man kann erwarten (hoffen?) dass hier die beste Wahl für die gegebene Maschine getroffen wurde. Falls hier ein Zwischenspeicher benutzt wird, muss sich, im Gegensatz zu dem "buffered" Mode, der Anwender nicht um dessen Bereitstellung kümmern.

Der Modus "ready" wirft das Datenpaket ins Netz und macht dann weiter. Die Funktionen heissen `MPI_Rsend` , `MPI_Rrecv`. Von "ready" Modus wird abgeraten, da Fehleranfällig.

Allen "blocking" sends ist gemeinsam, dass die Daten die von send verschickt werden, nach Rückgabe der Kontrolle an das Programm überschrieben werden dürfen.

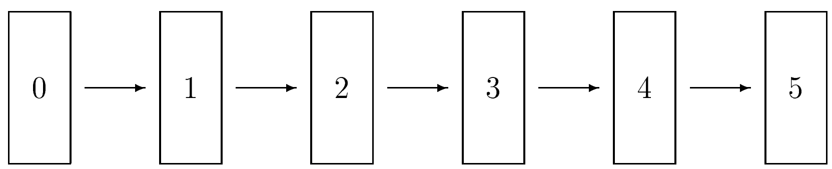
Bevor wir "non-blocking send" besprechen wollen wir uns anschauen, was bei "synchronous" und "buffered" send in unserem Jacobi-programm passiert.

Hier nochmal das Programmstück mit "standard" send und recieve:

```
void exchange()
{
/* send the boundary values to the neighbour myid+1 */
if(myid<(numprocs-1))
  MPI_Send(&phiold[NX][1], NY, MPI_DOUBLE,myid+1, 83, MPI_COMM_WORLD);
  /* recieve the corresponding data from  myid-1 */
  if(myid>0)
```

```
  MPI_Recv(&phiold[0][1], NY, MPI_DOUBLE,myid-1, 83, MPI_COMM_WORLD,&status);
  /* send the boundary values to the neighbour myid-1 */
  if(myid>0)
  MPI_Send(&phiold[1][1], NY, MPI_DOUBLE,myid-1, 84, MPI_COMM_WORLD);
  /* recieve the corresponding data from  myid+1 */
  if(myid<(numprocs-1))
  MPI_Recv(&phiold[NX+1][1], NY, MPI_DOUBLE,myid+1, 84, MPI_COMM_WORLD,&status);
}
```

Grafisch können wir uns den ersten Teil wie folgt verdeutlichen:



Dabei steht ein Kasten für einen Prozess und ein Pfeilanfang für Senden und ein Pfeilende für Empfangen.

Im Modus “synchronous” können die Prozesse 0 bis $numprocs-2$ erst mit dem lesen anfangen, nachdem `MPI_Ssend` fertig ist. Dies geschieht aber erst dann, wenn vom entsprechenden `MPI_Srecv` die Bestätigung geschickt wird, dass empfangen wurde. Diese Blockade wird nur dadurch aufgehoben, dass Prozess $numprocs-1$ nichts verschickt, und gleich empfangen kann. So kann $numprocs-2$ sein send beenden und empfangen usw. Hat man viele Prozesse, kann dies zu einer deutlichen Verlangsamung führen.

Mit periodischen Randbedingungen ginge es gar nicht. Die Blockade könnte nicht aufgehoben werden.

Im “buffered” Modus tritt dieses Problem nicht auf. Nachdem die zu verschickenden Daten im buffer sind, kann von allen Prozesses empfangen werden.

Bei “standard” hängt es von der Implementierung ab, was genau passiert. So kann z.B. der oben abgedruckte Code für periodische Randbedingungen auf einer Maschine laufen, aber einer anderen nicht, da es zu einer Blockade kommt.

In unseren Fall kommt auf jedes verschickte Datenpaket ein Paket, das empfangen wird. Hier ist es nützlich das kombinierte

```
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 )
```

zu verwenden. Hier werden send und recieve unabhängig voneinander gestartet. Die Funktion gibt die Kontrolle zurück sobald der send-buffer überschrieben werden kann, und im recieve-buffer die empfangenen Daten stehen. Send und recieve-buffer müssen voneinander verschieden sein.

Aus unserer Funktion "exchange" wird

```
void exchange2()
{
/* send the boundary values to the neighbour myid+1 */
if(myid%2==0)
  {
  if(myid<(numprocs-1))
  MPI_Sendrecv(&phiold[NX][1], NY, MPI_DOUBLE,myid+1, 83,
  &phiold[NX+1][1], NY, MPI_DOUBLE,myid+1, 84, MPI_COMM_WORLD,&status);
  if(myid>0)
  MPI_Sendrecv(&phiold[1][1], NY, MPI_DOUBLE,myid-1, 84,
  &phiold[0][1], NY, MPI_DOUBLE,myid-1, 83, MPI_COMM_WORLD,&status);
  }
else
  {
  if(myid>0)
  MPI_Sendrecv(&phiold[1][1], NY, MPI_DOUBLE,myid-1, 84,
  &phiold[0][1], NY, MPI_DOUBLE,myid-1, 83, MPI_COMM_WORLD,&status);
  if(myid<(numprocs-1))
  MPI_Sendrecv(&phiold[NX][1], NY, MPI_DOUBLE,myid+1, 83,
  &phiold[NX+1][1], NY, MPI_DOUBLE,myid+1, 84, MPI_COMM_WORLD,&status);
  }
```

3.2.2 non-blocking send and recieve

Die Idee bei der "non-blocking" Kommunikation ist es, während der Kommunikation mit Daten weiterzurechnen die nicht ausgetauscht werden. In unserem Problem heisst das, dass man "phinew" im inneren der Teilgitter rechnet, während die Kommunikation von statten geht.

Die "standard non-blocking send" Funktionen ist

```
int MPI_Isend( void *buf, int count, MPI_Datatype datatype, int dest, int tag,
               MPI_Comm comm, MPI_Request *request )
```

Im Vergleich mit `MPI_Send` haben wir das zusätzliche Argument `MPI_Request *request`. Hier wird die Kontrolle (im Prinzip) sofort wieder an das rufende Programm zurückgegeben. Um zu erfahren, ob die zu verschickenden Daten überschrieben werden dürfen, bedarf es hier einer weiteren Funktion:

```
int MPI_Wait (MPI_Request  *request, MPI_Status   *status)
```

Diese gibt die Kontrolle zurück, sobald die verschickten Daten überschrieben werden dürfen.

Die Funktion exchange mit "non-blocking" Kommunikation

```
void exchange_nb()
{
/* send the boundary values to the neighbour myid+1 */
if(myid<(numprocs-1))
MPI_Isend(&phiold[NX][1], NY, MPI_DOUBLE,myid+1, 83, MPI_COMM_WORLD,&req1);
/* recieve the corresponding data from  myid-1 */
if(myid>0)
MPI_Irecv(&phiold[0][1], NY, MPI_DOUBLE,myid-1, 83, MPI_COMM_WORLD,&req2);
/* send the boundary values to the neighbour myid-1 */
if(myid>0)
MPI_Isend(&phiold[1][1], NY, MPI_DOUBLE,myid-1, 84, MPI_COMM_WORLD,&req3);
/* recieve the corresponding data from  myid+1 */
if(myid<(numprocs-1))
MPI_Irecv(&phiold[NX+1][1], NY, MPI_DOUBLE,myid+1, 84, MPI_COMM_WORLD,&req4);
}
```

Die dazugehöhrige wait-Funktion

```
void exchange_wait()
{
if(myid<(numprocs-1)) MPI_Wait (&req1,&status);
if(myid>0)            MPI_Wait (&req2,&status);
if(myid>0)            MPI_Wait (&req3,&status);
if(myid<(numprocs-1)) MPI_Wait (&req4,&status);
}
```

Unsere Funktion `jacobi_step` müssen wir noch in zwei Teile zerlegen:

```
double jacobi_inner()
{
int i,j;
double diff,diff2,maxdif2;
maxdif2=0.;
/* update the phinew*/
/* for(i=1;i<=NX;i++) for(j=1;j<=NY;j++)  in jacobi_step */
for(i=2;i<NX;i++) for(j=1;j<=NY;j++)
  {
  phinew[i][j]=
  0.25*(phiold[i-1][j]+phiold[i+1][j]+phiold[i][j-1]+phiold[i][j+1]);
  diff=(phinew[i][j]-phiold[i][j]);
```

```
  diff2=diff*diff;
  if(diff2>maxdif2) maxdif2=diff2;
  }
return maxdif2;
}
```

und

```
double jacobi_boundary()
{
int i,j;
double diff,diff2,maxdif2;
maxdif2=0.;
/* update the phinew*/
/* for(i=1;i<=NX;i++) for(j=1;j<=NY;j++)  in jacobi_step */
for(i=1;i<=NX;i+=NX-1) for(j=1;j<=NY;j++)
  {
  phinew[i][j]=
  0.25*(phiold[i-1][j]+phiold[i+1][j]+phiold[i][j-1]+phiold[i][j+1]);
  diff=(phinew[i][j]-phiold[i][j]);
  diff2=diff*diff;
  if(diff2>maxdif2) maxdif2=diff2;
  }
/* copy the  phinew to phiold */
for(i=1;i<=NX;i++) for(j=1;j<=NY;j++) phiold[i][j]=phinew[i][j];
return maxdif2;
}
```

Dann im Hauptprogramm:

```
while(maxdiff>0.000001)
  {
  exchange_nb();
  diff1=jacobi_inner();
  MPI_Allreduce (&diff1, &maxdiff, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
  exchange_wait();
  diff1=jacobi_boundary();
  MPI_Allreduce (&diff1, &maxdiff2, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
  if(maxdiff2>maxdiff) maxdiff=maxdiff2;
  }
```

3.3 Testläufe

Als Test des oben Besprochenen haben wir die Programme auf 2 bzw. 4 Doppel-Prozessor Pentium Pro PCs mit 200 MHz laufen lassen. Die PCs sind über

Tabelle 1: *CPU-Zeiten in Sekunden für verschiedene Kommunikations - Funktionen. Die Programme jacobi1, jacobi2 und jacobi3 liefen auf zwei Doppel-Prozessor Pentium Pro PCs mit 200 MHz. Die beiden PCs sind mit 100Mbit-Ethernet verbunden. Es wurde die MPICH-1.2.2.3 Implementierung von MPI verwendet. Gitter gibt die Zahl der inneren Gitterpunkte. jacobi1 ist die Implementierung mit* `MPI_Send` *und* `MPI_Recv`. *jacobi2 mit* `MPI_Sendrecv` *und schleisslich jacobi3 mit dem non-blocking* `MPI_Isend` *und* `MPI_Irecv`. *Zum Vergleich ist jacobi0 das Programm ohne message passing, das auf einer CPU gelauffen ist. Iterationen sind die Zahl der Iterationen, die nötig sind um das Abbruchkriterium 10^{-10} zu erreichen. Um zu prüfen, dass wirklich von allen Programmen das gleiche gerechnet wird, haben wir geprüft, dass für alle die gleiche Iterationszahl erreicht wird.*

Gitter	jacobi1	jacobi2	jacobi3	jacobi0	Iterationen
64x64	4.1	4.9	6.4	3.2	4488
128x128	19.9	22.7	30.6	60.6	13059
256x256	338	290	348	748	33381

100Mbit-Ethernet mit einem switch verbunden. Die Ergebnisse für zwei PCs sind in der Tabelle 1 zusammengefasst.

Als Abbruchkriterium haben wir

$$\max_{i,j}(\Phi_{i,j}^{alt} - \Phi_{i,j}^{neu})^2 < 10^{-10} \qquad (3.1)$$

gewählt.

Die Programme wurden mit

```
mpicc -O jacobi1.c -o jacobi1
```

compiliert. Das `-O` steht für Optimierung. Dadurch braucht das compilieren länger, jedoch arbeiten die ausführbaren Programme schneller. So braucht etwa das sequenzielle Programm für das 64x64 Gitter 12.7 sec ohne `-O` aber nur 3.2 sec mit `-O`. Was genau bei `-O` passiert, kann man mit `man gcc` erfahren. Leider steigt mit zunehmender Optimierung auch die Gefahr, dass der Compiler Fehler macht. Daher ist es insbesodere bei grösseren Projekten sinnvoll, sich an Beispielen zu überzeugen, dass das executable mit Optimierung das gleiche tut, wie dasjenige ohne Optimierung.

Bei mpirun können wir steueren auf welchen Maschinen der Lauf ausgeführt werden soll:

```
mpirun -machinefile mfile -np 4 jacobi1
```

in dem file “mfile” sind die rechner eingetragen, die benutzt werden können. Etwa z.B.

```
pool11
pool12
pool13
pool14
```

Als default wird

```
/usr/local/mpich-1.2.4/share/machines.LINUX
```

verwendet.

Nun zu den Ergebnissen in Tabelle 1. Für das 64x64 Gitter braucht das sequenzielle Programm auf einer CPU weniger Zeit als das auf 4 CPUs verteilte Programm. Die Urache ist, dass der Kommunikation-overhead die Rechenzeit übersteigt.

Beim 128x128 gewinnen wir nun wirklich, verglichen mit dem sequenziellen Programm. Interessant ist zu beobachten, dass "jacobi1" und "jacobi2" etwa die gleiche Zeit benötigen. Hingegen "jacobi3" mit dem "non-blocking" send/recieve länger braucht. Offensichtlich ist "non-blocking" nicht effizient implementiert, und der höhere Programmieraufwand hat sich leider nicht ausgezahlt.

Noch eine Beobachtung zum sequenziellen Programm:
Für das 64x64 Gitter brauchen wir $3.2/(4488 * 64^2) = 0.174 * 10^{-6}$ sec Progitterpunkt und Iteration. Für das 128x128 Gitter hingegen $60.6/(13059 * 128^2) = 0.283 * 10^{-6}$ sec. (Für 256x256 $0.341 * 10^{-6}$ sec)

Dies hat mit der Grösse der L2-cache der CPU zu tun. Diese ist beim Pentium Pro 256 KB. Bei dem 64x64 Gitter brauchen wir $8 * 2 * 64^2 = 65536$ Byte für Φ^{alt} und Φ^{neu}. Beim 128x128 Gitter passen die Daten gerade nicht mehr in den cache.

Für eine feste Anzahl der Prozessoren gilt: Lassen wir solche cache-effekte ausser acht, so steigt die CPU-Zeit pro Iteration für die Rechnung linear mit dem Volumen, also mit N^d wobei d die Dimension ist (hier 2). Die Zeit für die Kommunikation linear mit der Oberfläche der Teilgitter. Also mit N^{d-1}. N sei hier die lineare Ausdehnung des Gitters. Es ergibt sich also

$$\frac{\text{Zeit für Kommunikation}}{\text{Zeit für Rechnung}} \propto N^{-1} \tag{3.2}$$

In Tabelle 2 ist das 256x256 Gitter nochmal auf 4 PCs mit 8 CPUs gerechnet.

Nun noch einmal zur Iterationszahl und der Fehlerabschätzung. Wir haben in der CPI gelernt, dass der Fehler sich wie

$$||\Phi^{(t)} - \Phi^{(\infty)}|| \propto \exp(-t/\tau) \tag{3.3}$$

verhält. Und für Gauss-Seidel und Jacobi

$$\tau \propto h^{-2} \propto N^2 \tag{3.4}$$

Tabelle 2: *Dasselbe wie vorherige tabelle auf 4 statt 2 PCs.*

Gitter	jacobi1	jacobi2	jacobi3	jacobi0	Iterationen
256x256	127	130	165		33381

gilt. Nun steigt die Iterationszahl in Tabelle 1 aber nicht um einen Faktor 4 wenn wir N verdoppeln!

Das liegt daran, dass wir $\Phi^{(t)}$ und $\Phi^{(t+1)}$ vergleichen. Leiten wir Gleichung 3.3 nach t ab bekommen wir

$$||\Phi^{(t)} - \Phi^{(t+1)}|| \propto \frac{1}{\tau} \exp(-t/\tau) \tag{3.5}$$

Insbesondere bei grossen τ ist also Vorsicht geboten!
In unserem Beispiel kennen wir die exakte Lösung. Φ muss überall 0 sein. Mit unserem Abbruchkriterium 10^10 für das Maximum des Quadrates der Differenz erhalten wir aber $1.314537e-01$ für das Feld in der Mitte des Gitters!

Ein andere, eventuell bessere Möglichkeit abzubrechen ist es eine Bedingung an das Residuum zu stellen. Sei

$$Ax = b \tag{3.6}$$

das lineare Gleichungssystem, welches wir lösen wollen. Sei $x^{(i)}$ die Näherungslösung nach der i-ten Iteration. Dann ist

$$r^{(i)} = Ax^{(i)} - b \tag{3.7}$$

das Residuum nach der i-ten Iteration.

3.4 Gauss-Seidel

Zur genauen Spezifikation des Gauss-Seidel-Algorithmus gehört die Festlegung der Reihenfolge, in der das Gitter durchlaufen wird. Ein natürliche Reihenfolge ist es nach "Schreibmaschinen" oder "type-writer" Art durch das Gitter zu laufen. Dies verhindert leider die Parallelisierung, da das Ergebnis an einem Ort (i, j) im Gitter von den Ergebissen der zuvor besuchten Punkte abhängt.

Eine Lösung dieses Problems ist die sogenannte Schachbrett (check board, red-black,even-odd) Zerlegung des Gitters. Wichtig dabei ist, dass alles weisen Gitterpunkte nur schwarze als nächste Nachbarn hat und umgekehrt.

Nun werden zuerst die Felder auf den weisen Gitterpunkten auf ihre neuen Werte gesetzt, und danach die Felder auf den schwarzen Gitterpunkten.

Dabei ist nun die Reihenfolge innerhalb der weisen bzw. innerhalb der schwarzen Gitterpunkte völlig egal. Insbesondere kann eine parallele Abarbeitung erfolgen.

Zur Definition von eigenen Datentypen nehmen wir etwas aus den Lecturenotes “MPI Course: Student Notes” des epcc in Edinburgh

Sub-block of a matrix
double results[IMAX][JMAX];

where we want to send results[0][5], results[1][5], ..., results[IMAX][5]. The data to be sent does not lie in one contiguous area of memory and so cannot be sent as a single message using a basic datatype. It is however made up of elements of a single type and is strided i.e. the blocks of data are regularly spaced in memory.

Consider

```
struct {
        int nResults;
double results[RMAX];
       } resultPacket;
```

where it is required to send resultPacket. In this case the data is guaranteed to be contiguous in memory, but it is of mixed type.

So, for example, one inelegant solution to SSub-block of a matrixön page 41, would be to send the elements in the column one at a time. In C this could be done as follows:

```
int count=1;
/*
 **********************************************************
* Step through column 5 row by row
**********************************************************
*/
for(i=0;i<IMAX;i++){
  MPI_Send (&(results[i][5]), count, MPI_DOUBLE,
            dest, tag, comm);
}
```

```
Derived datatypes are created at run-time.
Before a derived datatype can be used in a communication,
the program must create it. This is done in two
stages.

- Construct the datatype.
New datatype definitions are built up from existing datatypes (either derived or basic) using a call, or a recursive series
of calls, to the following routines:
MPI_TYPE_CONTIGUOUS, MPI_TYPE_VECTOR, MPI_TYPE_HVECTOR, MPI_TYPE_INDEXED
MPI_TYPE_HINDEXED,
```

```
MPI_TYPE_STRUCT.

-Commit the datatype.
The new datatype is "committed" with a call to MPI_TYPE_COMMIT.
It can then be used in any number of communications.
The form of MPI_TYPE_COMMIT is:

MPI_TYPE_COMMIT (datatype)

Finally, there is a complementary routine to MPI_TYPE_COMMIT,
namely MPI_TYPE_FREE, which marks a datatype for de-allocation.

MPI_TYPE_FREE (datatype)

Any datatypes derived from datatype are unaffected when it is freed,
as are any communications which are using the datatype at the time of freeing.
datatype is returned as MPI_DATATYPE_NULL.
```

Für unser Problem benötigen wir

```
MPI_TYPE_VECTOR (count, blocklength, stride, oldtype, newtype)
```

Das Beispiel mit der Benutzung der neu definierten Datentypen:

```
double results[IMAX][JMAX];
/* ******************************************************** *
* We want to send results[0][5], results[1][5],
* results[2][5], ...., results[IMAX-1][5]
* ******************************************************** */
MPI_Datatype newtype;
/* ******************************************************** *
* Construct a strided vector type and commit.
* IMAX blocks, each of length 1 element, separated by
* stride JMAX elements
* oldtype=MPI_DOUBLE
* ******************************************************** */
MPI_Type_vector (IMAX, 1, JMAX, MPI_DOUBLE, &newtype);
MPI_Type_Commit (&newtype);
/* ******************************************************** *
* Use new type to send data, count=1
* ******************************************************** */
MPI_Ssend(&(results[0][5]), 1, newtype, dest, tag, comm);
```

3.5 Mehrgitterverfahren

Bei dem Jacobi und Gauss-Seidel-Verfahren benötigt man mit zunehmender Verfeinerung der Gitter immer mehr Iterationsschritte, um eine bestimmte Genauigkeit zu erreichen. Mit Hilfe der Überrelaxation kann dieses Problem deutlich gemildert, aber nicht beseitigt werden. Für Gauss-Seidel und Jacobi gilt

$$\tau \approx h^{-2} \tag{3.8}$$

Bei optimaler Einstellung des Überrelaxationsparameters kann

$$\tau \approx h^{-1} \tag{3.9}$$

erreicht werden.

Mit Hilfe von Mehrgitter-Verfahren kann

$$\tau \approx h^{0} \tag{3.10}$$

erreicht werden.

Bei Mehrgitterverfahren wird eine Sequenz von Gittern verschiedener Feinheit verwendet.