2 Parallelisierung mit OpenMP I

Moderne Rechner sind mit Prozessoren ausgerüstet, die mit mehreren *cores* zugleich rechnen, gelegentlich befinden sich sogar zwei solcher Prozessoren auf einem Board, bei Compute–Servern noch mehr. Um diese Rechenleistung auf ein Problem zu bündeln, wird man nicht mehrere Prozesse starten, denn dann müsste man den Datenaustausch organisieren (*message passing*, MPI). Einfacher ist es, innerhalb *eines* Prozesses mehrere Rechenpfade (*threads*) parallel zu starten, die auf den gemeinsamen Speicher zugreifen (*shared memory*), aber unabhängig ablaufen und so die Prozessoren und/oder Cores auslasten. Man nutzt also die SMP–Fähigkeit der Systeme (*symmetric multiprocessing*) aus, und das dazu geschaffene Programmiermodell nennt sich **OpenMP**.
Ein wichtiger Gesichtspunkt für effiziente Parallelisierung: die Abläufe in den *threads* müssen weitgehend **unabhängig** sein, sonst entstehen entweder Wartezeiten, oder die Ergebnisse werden falsch. Es ist eine große Hilfe bei der Programmierung, dass die *threads* auf gemeinsame (**"shared"**) Daten zugreifen *können*, sie haben aber auch die Möglichkeit, eigene (**"private"**) Variablen anzulegen – es liegt in der Verantwortung des Programmierers, das richtig zu organisieren.
Die Parallelisierung wird in OpenMP durch Direktiven gesteuert, die in C wie Präprozessor–Anweisungen aussehen:

```
#pragma omp ...
   {..block..}
```

Ihr Einflußbereich umfasst, wie angedeutet, den unmittelbar darauf folgenden Block. So beginnt mit

```
#pragma omp parallel
   {..block..}
```

ein Bereich, der in mehreren *threads* **parallel** bearbeitet werden soll. Darin kann nun wiederum mit

```
#pragma omp critical
   {..block..}
```

ein Abschnitt stehen, auf den die *threads* **nicht** gleichzeitig zugreifen dürfen, z.B. der Aufruf eines Zufallszahlen–Generators, der allen *threads* gemeinsam ist.
In dieser allgemeinen Form brauchen wir die Parallelisierung aber eher selten. In unseren Programmen sind die rechenintensiven Teile als große Schleifen organisiert, und es genügt, die Schleifendurchläufe (nach einem geeigneten Schema) auf die verschiedenen *threads* zu verteilen. Dafür gibt es eine spezielle Direktive `#pragma omp parallel for`, die an folgendem Beispiel erläutert wird:

```
#include <omp.h>     /* hier nicht noetig */

int i, n;
double r[n], p[n];
double beta;

#pragma omp parallel for
for (i=0; i<n; i++) p[i] = r[i] + beta*p[i];
```

In diesem Programmfragment aus dem CG wird offenbar der neue Verschiebungsvektor $p' = r + \beta p$ berechnet. Die Direktive bewirkt eine Aufteilung der Schleifen–Iterationen in (ungefähr) gleich große Abschnitte, zur Verteilung an die *threads*. Da die Rechnung für alle Werte von `i` unabhängig abläuft, ist es völlig korrekt, sie auf den *shared* Feldern `p` und `r` ablaufen zu lassen, obwohl `p` verändert wird: jeder *thread* überschreibt ja nur "seine" Komponenten `p[i]`. Auch `n` und `beta` sind *shared*, nur die Schleifenvariable `i` muss jeder *thread* individuell führen (*private*), das liegt in der Natur der Sache.

Die Direktive würde also eigentlich so aussehen:

```
#pragma omp parallel for \
   shared(p, r, n, beta) private(i)
for (i=0; i<n; i++) p[i] = r[i] + beta*p[i];
```

Nun sagen die Voreinstellungen bei OpenMP: Variablen, die außerhalb des parallelen Bereichs definiert sind, werden als *shared* betrachtet, die Schleifenvariable (bei `omp parallel for`) ist *private*. Das ist genau das, was wir brauchen, deshalb konnte man die Direktive in diesem einfachen Fall ohne explizite Klauseln schreiben.
Als weiteres Beispiel hier die Funktion, die ein Skalarprodukt berechnet:

```
double dotprod(int n, double *x, double *y){
   int    i;
   double dp;

   dp = 0;
   #pragma omp parallel for \
      reduction(+: dp)
   for (i=0; i<n; i++) dp += x[i]*y[i];
   return dp;
}
```

Eigentlich ist die Summationsschleife über die Komponenten rekursiv, denn jede Iteration nimmt den Wert von `dp` aus der vorigen Iteration und addiert ihren Beitrag. Weil aber die Addition assoziativ ist, kann man den Ablauf umformulieren: jeder *thread* legt sich eine private Summationsvariable an und berechnet damit seine Teilsumme, am Ende werden alle Teilsummen zur Gesamtsumme `dp` addiert und als Resultat zurück gegeben. Genau das

verbirgt sich hinter der Spezifikation `reduction(+: dp)`, man braucht das also nicht selber zu programmieren.
Mit diesen einfachen Mitteln lassen sich innerhalb der Iterationsschleife des CG alle Schleifen parallelisieren, die sich über alle Gitterpunkte erstrecken.
Die gängigen **Compiler** beherrschen OpenMP, man aktiviert es durch folgende Optionen:

```
Intel:      -openmp -openmp-report1
Portland:   -mp
            -mp=allcores
GNU:        -fopenmp
```

Die **Anzahl** der *threads*, die das Programm zur Parallelisierung öffnet, kann man zu Laufzeit festlegen, am einfachsten mit einer Umgebungsvariablen, die man *vor* dem Programmaufruf setzt:

```
export OMP_NUM_THREADS=4
a.out
```

2.1 OpenMP in Fortran

Bislang wurde die Syntax der OpenMP–Direktiven an Beispielen in C erläutert, hier eine kurze Übersicht über die Notation in Fortran.
Da man in Fortran nicht willkürlich Blöcke definieren kann, gibt es zusätzliche `end` – Direktiven.
Unser obiges Beispiel zu $p' = r + \beta p$ sieht jetzt so aus:

```
!$omp parallel do &
!$omp    shared(p, r, n, beta) private(i)
do i=1,n
   p(i) = r(i) + beta*p(i)
enddo
!$omp end parallel do
```

Auch hier wären die Klauseln `shared(..)` und `private(..)` nicht nötig gewesen, und auch das `end parallel do` hätte man sich sparen können, weil es direkt hinter dem zugehörigen `enddo` steht:

```
!$omp parallel do
do i=1,n
   p(i) = r(i) + beta*p(i)
enddo
```

Wie man sieht, wurde die Vektornotation `p = r + beta*p` von Fortran wieder in eine explizite Schleife aufgelöst. Das ist leider nötig, denn *eigentlich* gibt es für diesen Zweck eine spezielle Direktive (`workshare`) – sie ist aber bisher in den Compilern nicht ordentlich implementiert und deshalb unbrauchbar. Aus demselben Grund müssen auch Summationen über Felder, die man in Fortran einfach als `sum(..)` aufrufen kann, wieder in Schleifen verwandelt werden, wenn die Summation auf parallele *threads* verteilt werden soll.