7 Parallelisierung mit OpenMP II

Eine auf Schleifen basierende Parallelisierung mit OpenMP haben wir schon beim CG–Programm durchgeführt – mit gutem Erfolg, jedenfalls solange die Systeme insgesamt in den Cache passten.
Im MC–Programm sind aber die innersten Laufindizes für die Vektorisierung (über automatisch erzeugte SSE–Instruktionen) vorgesehen, da sollte die Parallelisierung nicht eingreifen – dafür bietet die Zellstruktur einen geeigneten Ansatzpunkt.

7.1 Parallelisierung in der Basiszelle

Bei einer "Mess"–Schleife, bei der über das vorgegebene Spinfeld summiert wird (z.B. zur Berechnung der Magnetisierung $M[\Phi]$), parallelisiert man einfach die Schleife über den Basisindex:

```
int ib, i;
double mag_r = 0., mag_i = 0.;

#pragma omp parallel for \
        private(i) \
        reduction(+: mag_r, mag_i)
for (ib=0; ib<nvcell; ib++){       /* parallel loop over base points */

  for (i=0; i<lvec; i++) {         /* vector loop */
    mag_r += phi_r[ib][i];
    mag_i += phi_i[ib][i];
  }
}
```

Dasselbe in Fortran:

```
real(8) mag_r, mag_i

mag_r = 0.; mag_i = 0.
!$omp parallel do &
!$omp reduction(+: mag_r, mag_i)
do ib=1,nvcell                     ! parallel loop over base points

  do i=1,lvec                      ! vector loop
    mag_r = mag_r + phi_r(i,ib)
    mag_i = mag_i + phi_i(i,ib)
  enddo
enddo
```

Die innere Vektorschleife wird nicht aufgebrochen, sondern nach wie vor auf einem Core mit SSE prozessiert. Zur Parallelisierung bekommt jeder *thread* einen Bruchteil der Punkte in der Basiszelle (mit den darauf stehenden Vektoren) zur Bearbeitung zugeteilt.

7.2 Parallelisierung der Metropolis–Updates

Bei der Parallelisierung des Metropolis–Updates gibt es ein neues Problem: nicht alle Vektoren von Spins können parallel bearbeitet werden, weil einige in einem Nachbarschaftsverhältnis zueinander stehen. Das ist immer dann der Fall, wenn ihre Fußpunkte in der Basiszelle nächste Nachbarn sind, mit Berücksichtigung der periodischen Randbedingungen. In einem solchen Fall behilft man sich mit einer **Kolorierung** der Gitterpunkte: nach einem (mehrdimensionalen) Schachbrettmuster werden die Punkte als schwarz/weiß (gerade/ungerde, *red/black*) gekennzeichnet. Das geht jedenfalls dann auf, wenn die Zellen in allen Richtungen **gerade** Abmessungen haben. Das Geometrie–Programm `geom_vec` hat das schon eingebaut und gibt die Farbe des Basispunkts `ib` als `nnflag[0][ib] = 0, 1` zurück. Wir müssen jetzt nur noch die Schleife über die Basispunkte so umbauen, dass zuerst alle “weißen”, dann alle “schwarzen” Punkte durchlaufen werden:

```
...
int ib, icolor, j, map[2][nvcell/2];

for (ib=0; ib<nvcell; ib++) map[nnflag[0][ib]][ib/2] = ib;

for (icolor=0; icolor<2; icolor++) {    /* Schleife ueber Farben */

  for (j=0; j<nvcell/2; j++) {   /* Schleife ueber Basispunkte einer Farbe */
    ib = map[icolor][j];

    .. Vektorielles Update ..

  }                             /* Ende der Schleife ueber Basispunkte */
}                               /* Ende der Schleife ueber Farben */
```

Nun kann die Schleife über `j` parallelisiert werden, weil dort nur Gitterpunkte je *einer* Farbe vorkommen, die sich unabhängig bearbeiten lassen. Das Metropolis–Programm sieht dann schematisch wie folgt aus:

```
  ...
  int i, k, ib, jb, ihit, nacpt, icolor, map[2][nvcell/2];
  double b_r[lvec], b_i[lvec], phi_new_r[lvec], phi_new_i[lvec];
  double sq[lvec], rho[lvec], rho_new[lvec];
  double r1[lvec], r2[lvec], r3[lvec];

  nacpt = 0;

  for (ib=0; ib<nvcell; ib++) map[nnflag[0][ib]][ib/2] = ib;

  for (icolor=0; icolor<2; icolor++) {    /* Schleife ueber Farben */

#pragma omp parallel for \
        private(ib, i, k, jb, ihit, b_r, b_i, phi_new_r, phi_new_i) \
        private(sq, rho, rho_new, r1, r2, r3) \
        reduction(+: nacpt)
    for (j=0; j<nvcell/2; j++) {   /* Schleife ueber Basispunkte einer Farbe */
```

```
    ib = map[icolor][j];

      /* Hintergrundfeld */

      ...berechnen...

      /* Metropolis hits */

      ...Ausgangswahrsch. berechnen...

      for (ihit = 0; ihit < nhits; ihit++) {

        dfrngv(lvec, r1)
        dfrngv(lvec, r2)
        dfrngv(lvec, r3)

        ...Vorschlag berechnen...

        ...Akzeptanzschritt...

      }
    }
  }                           /* Ende der Schleife ueber Basispunkte */
}                             /* Ende der Schleife ueber Farben */
...
```

Dasselbe für Fortran–Programmierer:

```
real(8), dimension(lvec) :: b_r, b_i, phi_new_r, phi_new_i, sq, rho, rho_new
real(8), dimension(lvec) :: r1, r2, r3
integer map(nvcell/2,0:1)

...

do ib=1,nvcell
   map((ib+1)/2, nnflag(ib,0)) = ib
enddo

do icolor=0,1         ! Schleife ueber Farben

   !$omp parallel do &
   !$omp private(ib, jb, b_r, b_i, phi_new_r, phi_new_i) &
   !$omp private(sq, rho, rho_new, r1, r2, r3) &
   !$omp reduction(+: nacpt)
   do j=1,nvcell/2    ! Schleife ueber Basispunkte einer Farbe

      ib = map(j,icolor)

      ! Hintergrundfeld berechnen
      ...
```

```
        ! Metropolis vorbereiten
        ...

        do ihit=1,nhits

           call dfrngv(lvec, r1)
           call dfrngv(lvec, r2)
           call dfrngv(lvec, r3)

           ! Vorschlag berechnen
           ...

           ! Akzeptanzschritt
           ...

        enddo
     enddo              ! Ende der Schleife ueber Basispunkte einer Farbe
  enddo                 ! Ende der Schleife ueber Farben
```

7.3 Parallelisierung des Zufallszahlengenerators

Der Aufruf von Zufallszahlen verlangt noch besondere Vorsicht: es ist möglich, mit einer Kopie des Generators auszukommen und den Zugriff so zu organisieren, dass die *threads* nacheinander bedient werden. Bei einer größeren Anzahl von *threads* führt das aber zu Wartezeiten und verhindert die reibungslose Parallelisierung. Besser ist es, jedem *thread* eine eigene Kopie des Generators zuzuordnen. Das muss man nicht extra programmieren: der Aufruf einer Funktion in einem parallelen Bereich (z.B. `dfrngv(lvec, r1)` in der obigen Schleife über `j`) wird in jedem *thread* unabhängig ausgeführt, die dabei intern benutzen *lokalen* Variablen sind automatisch `private`, und der Resultatvektor `r1` ist ebenfalls als `private` deklariert. Der interne Zustand des Generators wird allerdings in statischen Variablen gespeichert, und die wären allen *threads* gemeinsam (`shared`). Dem muss man entgegenwirken, indem man im Code von `frng.c` eine Direktive einfügt, die die "Zustandsgrößen" `ireg[..], i0, i1` privatisiert:

```
static unsigned INT64 ireg[LEN] =
   ...
static int i0 = 0, i1 = LEN - LAG;
#pragma omp threadprivate(ireg, i0, i1)
```

Damit nicht in allen *threads* dieselbe Sequenz von Zufallszahlen benutzt wird, muss man noch für eine individuelle Initialisierung sorgen. Dazu erzeugt man *vor Beginn* der Simulation kurz einen parallelen Bereich und läßt die Initialisierung von der *thread*–Nummer abhängen:

```
#include <omp.h>

#pragma omp parallel
frnini(omp_get_thread_num());
```

In einem parallelen Bereich stehen nämlich eine Reihe von Hilfsfunktionen zur Verfügung, z.B. gibt `num = omp_get_num_threads()` die Anzahl der *threads* an und `omp_get_thread_num() = 0,1,..(num-1)`.
In `frng.f90` sieht die Privatisierung der Zufallszahlen so aus:

```
integer(k8) :: ireg(0:30) = ...
integer :: i0 = 0, i1 = 18
!$omp threadprivate(ireg, i0, i1)
```

und so die Initialisierung:

```
integer omp_get_thread_num
...
!$omp parallel
call frnini(omp_get_thread_num())
!$omp end parallel
```

7.4 Kleine Übersicht über OpenMP

7.4.1 shared / private

Variablen, die **außerhalb** des parallelen Bereichs definiert werden, sind auf *shared* voreingestellt. Das gilt auch für **globale** Variablen, die per Kopfdeklaration (in C) bzw. `use` (in Fortran) zur Verfügung stehen.
Die **Laufvariable** eines *parallel for* (in C) ist *private*, ebenso *alle* Laufvariablen innerhalb eines *parallel do* (in Fortran).
In C sind Variablen, die in Blöcken **innerhalb** des parallelen Bereichs definiert werden, per Voreinstellung *private*.
Bei Aufruf von **Unterprogrammen** im parallelen Bereich sind lokal definierte Variablen i.a. *private*, nur die oben genannten globalen Variablen und die durch `static` bzw. `save` als **statisch** deklarierten sind *shared*.
Eine *private* Variable ist bei **Eintritt** in den parallelen Bereich undefiniert und wird am **Ende** wieder ungültig. Dem kann man mit den Spezifikationen *firstprivate* und *lastprivate* entgegenwirken.
Sind dagegen Variablen als *threadprivate* deklariert, dann werden sie per *thread* angelegt (wie bei *private*), bleiben aber zwischen verschiedenen parallelen Bereichen erhalten und können so statische Daten *thread*–spezifisch verwalten.

7.4.2 Felder

Ein mit `malloc()` im seriellen Bereich angelegtes Feld kann man *nicht* dadurch privatisieren, dass man nur den Pointer als *private* deklariert – man müsste die Aufrufe von `malloc() / free()` im parallelen Bereich vornehmen. Das ist wahrscheinlich nicht sehr effizient.
Dasselbe gilt für Felder mit dem Attribut `allocatable` in Fortran.
Dagegen kann man *automatische* Arrays (in C und Fortran) einfach als *private* deklarieren, indem man den Namen in die Direktive setzt und die Speicherverwaltung dem Stack überlässt.

7.4.3 Bedingte Kompilierung

Wenn man ein Programm *ohne* OpenMP–Unterstützung kompiliert, werden die Direktiven ignoriert. Die Funktionsaufrufe führen aber zu Fehlermeldungen, die man vermeiden kann, indem man sie geeignet "versteckt":

```
in C:
   #ifdef _OPENMP
      ...
      ...
   #endif
in Fortran:
   !$ ...
   !$ ...
```

Eine tabellarische Übersicht über Syntax, Direktiven und OpenMP–Aufrufe findet sich in Chandra et al., Anhang A.