Programmierung mit OpenMP

1. Konzepte und **`hello world`**
2. Überblick
3. Parallelisierung einer Schleife
4. Eine komplexere Schleife
5. Parallele Bereiche
6. Lastausgleich
7. Parallele Abschnitte
8. Sequentielle Abschnitte
9. Bibliotheken und Compiler
10. Bewertung

1. Konzepte und `hello world`

Automatische Parallelisierung durch Compiler immer noch nicht möglich

- Trotz langjähriger Forschung weder für Nachrichtenaustausch noch für gemeinsame Speicherbereiche

Aber: Compilergestützte Parallelisierung möglich

- Im Falle von OpenMP für gemeinsamen Speicher!

Alternativ: Bibliotheksbasierte Ansätze

- Message Passing Interface (MPI) für verteilten Speicher
- Pthreads für gemeinsamen Speicher

Neuer Ansatz: OpenMP (Open Multi-Processing)

- Keine neue Programmiersprache
- Arbeitet mit Fortran und C/C++ zusammen

- Compiler-Direktiven steuern Übersetzung
- Zusätzliche (kleine) Bibliothek
- Direktiven+Bibliothek sind das API von OpenMP

- OpenMP-Compiler übersetzt in Programme mit Threads (nicht weiter spezifiziert)
  - Verwendung ausschließlich für gemeinsamen Speicher!

OpenMPs Hello World

rot: OpenMP-Konstrukte

```
   programm hello
   print *, "Hello world from thread:"
!$omp parallel
   print *, omp_get_thread_num()
!$omp end parallel
   print *, "Back to the sequential world."
   end
```

- Umgebungsvariable: **OMP_NUM_THREADS**
- Bibliotheksaufruf: **omp_get_thread_num()**
- Compiler-Direktive: **!$omp parallel**

- Thread-Nummern: 0...**OMP_NUM_THREADS**-1

Warum ein Compiler-Ansatz?

Zunächst reiner Compiler-Ansatz geplant

Vorteil gegenüber Bibliotheken

- Nicht-OpenMP-Compiler ignorieren parallele Konstrukte automatisch
- Compiler können zusätzlich optimieren
- Inkrementelle Parallelisierung möglich

Reiner Compiler-Ansatz zu schwierig

- Erweiterung durch einige einfache Bibliotheksaufrufe

Geschichte von OpenMP

- OpenMP sehr neu: seit 1997
  - Version 3.0, Mai 2008: Neu ist das Konzept der *tasks*
  - Version 3.1, Juli 2011
  - Version 4.0, Juli 2013: Unterstützung für Beschleunigerkarten, für Fortran 2003 u.a.
  - Aktuell Version 4.5 (Nov. 2015)

- Hat aber lange Vorgeschichte
  - Ehemaliger ANSI X3H5-Standard zur Programmierung von gemeinsamem Speicher

    Früher auf parallelen Maschinen verbreitet

- OpenMP jetzt von allen Herstellern akzeptiert
- Von unabhängiger Organisation gefördert

2. Überblick

Portabilität: Neuübersetzung reicht aus

Kategorien der Spracherweiterungen

- Kontrollstrukturen, um Parallelismus auszudrücken
- Datenumgebungskonstrukte zur Kommunikation zwischen Threads
- Synchronisationskonstrukte zur Ablaufsteuerung von Threads

Überblick (2)

Compiler-Direktiven

- in Fortran

  ```
  !$OMP <directive> <clauses>
  ```

- In C/C++

  ```
  #pragma omp <directive> <clauses>
  ```

Zusätzlich bedingte Übersetzung der OpenMP-Bibliotheksaufrufe

Überblick (3)

Parallele Kontrollstrukturen

- Ausführungsmodell genannt fork/join-Modell
- Parallele Kontrollstrukturen starten neue Threads und übergeben ihnen die Kontrolle

Zwei Varianten

- `parallel`-Direktive: umschließt Block und erzeugt Menge von Threads, die den Block nebenläufig abarbeiten
- `do`-Direktive: verteilt Instanzen von Schleifendurchläufen auf Threads

Überblick (4)

Kommunikation und Datenumgebung

- Regelt, wer wann wo welche Daten sehen kann

Programm beginnt immer mit einem Thread (master-Thread)

Bei **`parallel`** werden neue Threads gestartet – jeweils mit eigenem Keller

Variable können von folgenden Typen sein

- **`shared`** – allen Threads gemeinsam zugängliche Variable
- **`private`** – thread-lokale Variable
- **`reduction`** – Mischform zur Ergebniszusammenführung
- **...**

Überblick (5)

Synchronisation

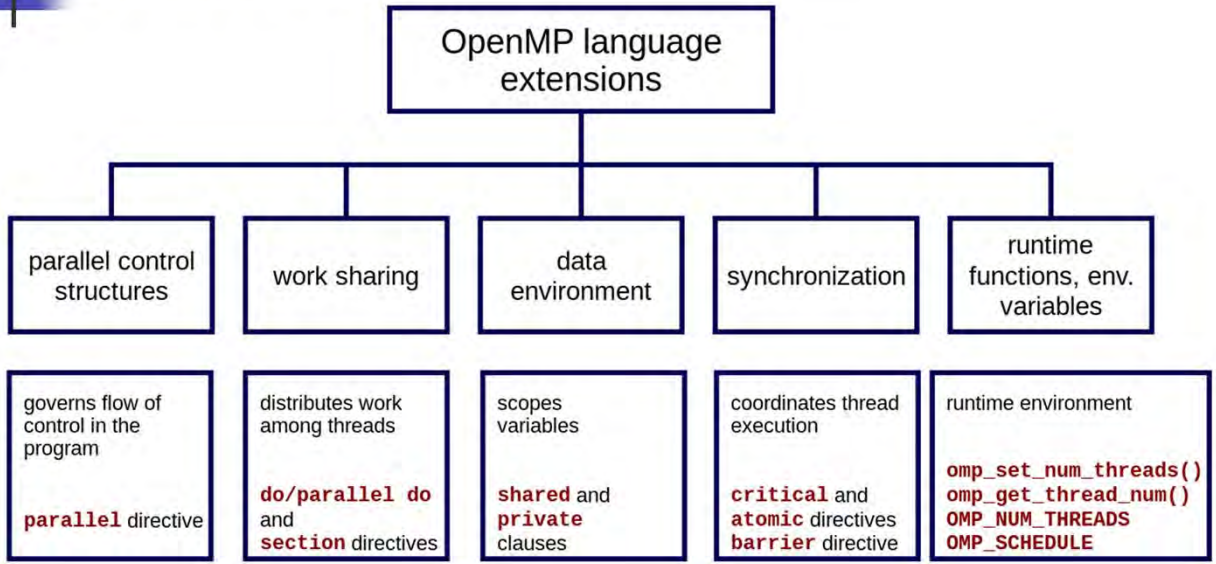
- Regelt den Ablauf der Threads

Zwei Hauptformen

- Wechselseitiger Ausschluß mittels **`critical`**-Direktive
- Ereignis-Synchronisation mittels **`barrier`**-Direktive

Weitere Konstrukte zur Bequemlichkeit oder Leistungsoptimierung

Überblick (6)


OpenMP language extensions
parallel control structures
work sharing
data environment
synchronization
runtime functions, env. variables
governs flow of control in the program
parallel directive
distributes work among threads
do/parallel do and section directives
scopes variables
shared and private clauses
coordinates thread execution
critical and atomic directives
barrier directive
runtime environment
omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

3. Parallelisierung einer Schleife

```
subroutine saxpy(z,a,x,y,n)
integer i,n
real z(n),a,x(n),y

do i=1,n
   z(i)=a*x(i)+y
enddo

return
end
```

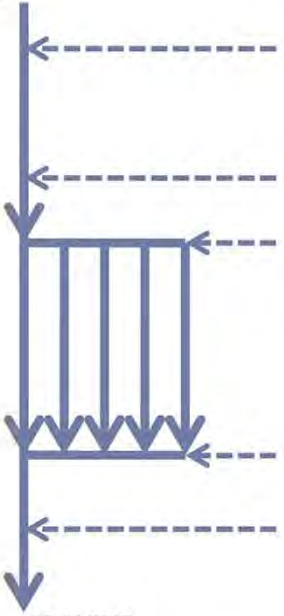
Keine Datenabhängigkeiten in der Schleife

Parallelisierung einer Schleife (2)

```
      subroutine saxpy(z,a,x,y,n)
      integer i,n
      real z(n),a,x(n),y
!$omp parallel do
      do i=1,n
         z(i)=a*x(i)+y
      enddo
!$omp end parallel do
      return
      end
```

Hier Parallelisierung alleine auf Schleifenindexeben

Parallelisierung einer Schleife (3)



Ausführungsmodell

master-Thread (mT) bearbeitet seriellen Anteil des Codes

mT betritt saxpy-Routine

mT trifft auf **`parallel-do`**-Direktive und kreiert Threads

mT und Kind-Threads teilen die Iterationen auf und bearbeiten sie nebenläufig

Implizite Barriere: warte auf alle Threads

mT macht nach der **`do`**-Schleife alleine weiter

Kindthreads verschwinden

Parallelisierung einer Schleife (6)

Kommunikation und Datengültigkeit

- Außerhalb des **`parallel-do`**-Blocks
  - Die Variablen z, a, x, y, n, i sind nur einmal vorhanden
- Innerhalb des **`parallel-do`**-Blocks
  - Die Variablen z, a, x, y, n sind nur einmal vorhanden
    Vorsicht mit der Semantik beim Zugriff!
  - Die Schleifenvariable i wird als thread-lokale Variable angelegt
    Aktualisierungen in einem Thread sind in anderen Threads nicht sichtbar

Parallelisierung einer Schleife (7)

Synchronisation

- Anforderungen
  - Variable z muß aktualisiert worden sein, wenn mit Anweisungen nach der Schleife fortgesetzt wird
- Realisierung
  - **`parallel do`**-Direktive hat implizite Barriere am Schleifenende

4. Eine komplexere Schleife

```
real*8 x,y
integer i,j,m,n,maxiter
integer depth(*,*)
integer mandel_val
...
maxiter=200

do i=1,m
   do j=1,n
      x=i/real(m)
      y=j/real(n)
      depth(j,i)=mandel_val(x,y,maxiter)
   enddo
enddo
```

Eine komplexere Schleife (2)

Zur Funktion **`mandel_val`**

- Darf nur von Eingabeparametern abhängen
- D.h. muss durch parallele Threads nutzbar sein (*thread-safe*)

Zu den Variablen

- Variable i per default **`private`**
  (weil diese Schleife parallelisiert wird)
- Variablen j,x,y explizit auf **`private`** gesetzt
  (default wäre **`shared`**)

Eine komplexere Schleife (3)

```
      real*8 x,y
      integer i,j,m,n,maxiter
      integer depth(*,*)
      integer mandel_val
      ...
      maxiter=200
!$omp parallel do private(j,x,y)
      do i=1,m
         do j=1,n
            x=i/real(m)
            y=j/real(n)
            depth(j,i)=mandel_val(x,y,maxiter)
         enddo
      enddo
!$omp end parallel do
```

Eine Verkomplizierung

```
maxiter=200
do i=1,m
   do j=1,n
      x=i/real(m)
      y=j/real(n)
      depth(j,i)=mandel_val(x,y,maxiter)
      total_iters=total_iters+depth(j,i)
   enddo
enddo
```

Mitzählen der gesamten Iterationen

Variable **`total_iters`** per default **`shared`**

Eine Verkomplizierung (2)

Zugriff auf **`total_iters`** in kritischem Bereich

```
!$omp critical
   total_iters=total_iters+depth(j,i)
!$omp end critical
```

Verfahren wird beim Zugriff auf **`total_iters`** serialisiert

- Zugriffszeit sollte prozentual kleiner Anteil sein!

Reduktion

```
   maxiter=200
   total_iters=0
!$omp parallel do private(j,x,y)
!$omp+ reduction(+:total_iters)
   do i=1,m
      do j=1,n
         x=i/real(m)
         y=j/real(n)
         depth(j,i)=mandel_val(x,y,maxiter)
         total_iters=total_iters+depth(j,i)
      enddo
   enddo
!$omp end parallel do
```

Schleifenparallelisierung

Hauptproblem der Praxis:

Datenabhängigkeiten zwischen Schleifenindizes

Bespiel:

```
do i=2,n
   a(i)=a(i)+a(i-1)
enddo
```

Lösung des Problems

- Komplizierte Methoden zum Finden der Abhängigkeiten
  - Großes Forschungsgebiet seit >20 Jahren
- Verschiedene Methoden zu ihrer Beseitigung
  - Zusätzliche Variable
  - Zugriffskoordination mit kritischem Bereich

5. Parallele Bereiche

Bisher nur parallele Schleifen (feingranular)

Jetzt auch grobgranularer Parallelismus

Konstrukt: **`parallel / end parallel`**

- eingeschlossener Code wird mit mehreren Threads nebenläufig bearbeitet

Parallele Bereiche (2)

```
maxiter=200
  do i=1,m
     do j=1,n
        x=i/real(m)
        y=j/real(n)
        depth(j,i)=mandel_val(x,y,maxiter)
     enddo
  enddo
  do i=1,m
     do j=1,n
        dith_depth(j,i)=0.5*depth(j,i)+
$            0.25*(depth(j-1,i)+depth(j+1,i))
     enddo
  enddo
```

Parallele Bereiche (3)

```
 maxiter=200
!$omp parallel
!$omp+ private(i,j,x,y)
!$omp+ private(my_width,my_thread,i_start,i_end)
   my_width=m/2
   my_thread=omp_get_thread_num()
   i_start=1+my_thread*my_width
   i_end=i_start+my_width-1
   do i=i_start,i_end
      do j=1,n
         x=i/real(m)
         y=j/real(n)
         depth(j,i)=mandel_val(x,y,maxiter)
      enddo
   enddo
```

Parallele Bereiche (4)

```
      do i=i_start,i_end
         do j=1,n
            dith_depth(j,i)=0.5*depth(j,i)+
$                 0.25*(depth(j-1,i)+depth(j+1,i))
         enddo
      enddo
!$omp end parallel
```

Diese Parallelisierung ist für genau 2 Threads programmiert

Keine Compiler-Parallelisierung auf Schleifenebene

6. Lastausgleich

Standard: jeder Thread erledigt gleich viele Iterationen einer Schleife

Aber: falls Schleifenrumpf in der Bearbeitungszeit variiert, führt das zu Lastungleichheit

Mechanismus: **`schedule`**-Klausel

- **`static`**: Zuteilung der Indizes zu Schleifenbeginn
- **`dynamic`**: Indizes werden zur Laufzeit zugeteilt

Lastausgleich (2)

`schedule(type[,chunk])`

- **`static`**: etwa Gleichverteilung
- **`static chunk`**: Rundumverteilung von Blöcken der Größe **`chunk`**
- **`dynamic`**: dynamische Rundumverteilung von Blöcken der Größe **`chunk`** (default=1)
- **`guided`**: Die Blockgröße sinkt exponentiell bis auf **`chunk`** ab; dynamische Rundumverteilung
- **`runtime`**: Verfahren wird durch die Umgebungsvariable **`OMP_SCHEDULE`** bestimmt

7. Parallele Abschnitte

Bei nichtiterativen Arbeitslasten:
Zuteilung von Code zu Threads

```
!$opm section [clause [,] [clause...]]
  [!$omp section]
     code for the first section
  [!$omp section
     code for the second section
     ...
  ]
  !$omp end sections [nowait]
```

Parallele Abschnitte (2)

- Die Anzahl der gewählten Threads bearbeitet unabhängig die Abschnitte
- Jeder Abschnitt genau einmal durchlaufen
- Nicht bestimmbar, welcher Thread welchen Abschnitt bearbeitet
- Nicht bestimmbar, wann welcher Abschnitt an die Reihe kommt
- Deshalb: Ausgabe eines Abschnittes sollte nicht Eingabe für einen anderen sein

8. Sequentielle Abschnitte

Parallele Abarbeitung manchmal zeitweilig nicht erwünscht

```
!$omp single [clause [,] [clause...]]
    Anweisungsblock der nur von einem
    Thread bearbeitet wird
!$omp end single [nowait]
```

Keine Barriere zu Beginn des sequentiellen Abschnitts

Mittels **`nowait`** warten Threads nicht auf das Ende des sequentiellen Abschnitts

Sequentielle Abschnitte (2)

Beispiel: Ausgabe

```
!$omp parallel shared (out,len)
   ...
!$omp single
   call write_array(out,len)
!$omp end single nowait
   ...
!$omp end parallel
```

Ereignissynchronisierung

Barrieren: alle Threads warten an der Barriere, bis alle parallelen Threads eingetroffen sind – dann erfolgt die Fortsetzung der Arbeit

```
!$omp barrier
```

Ordnung: erzwingt ein Durchlaufen von Anweisungen in der ursprünglichen Reihenfolge der Indexwerte (d.h. wie in einem sequentiellen Programm)

```
!$omp ordered
   block
!$omp end ordered
```

9. Bibliotheken und Compiler

Bibliotheken

Zusammenbinden von OpenMP-Programmen mit Bibliotheken von Dritten

- Es muss sichergestellt sein, dass die Bibliotheksaufrufe *thread-sicher* sind
- Andernfalls Bibliothek nicht in parallelen Bereichen verwenden
- Oder z.B. als kritischen Bereich kennzeichnen

- Heutzutage sind allerdings die meisten Bibliotheken schon *thread-sicher*

Bibliotheken und Compiler (2)

Compiler

Früher: spezielle Präprozessoren und Compiler

Heute: in alle gängigen Compiler integriert

- Unterstützung variiert aber!

Beispiele für OpenMP 3.1

- GCC 4.7
- Intel Fortran and C/C++ compilers 12.1
- LLVM/Clang 3.7

10. Bewertung

Vorteile

- Portabler Multithread-Code
- Einfach zu programmieren
- Inkrementelle Parallelisierung: beginne mit kleinen Änderungen
- Einheitlicher Code für sequentielle und parallele Programmversion
- Sequentieller Code muss nicht modifiziert werden, dadurch keine versehentlichen neuen Fehler
- Parallelismus auf verschiedenen Ebenen möglich
- Kann mit verschiedenen Beschleunigern verwendet werden
- Kann mit Nachrichtenaustausch gemischt werden

Bewertung (2)

Nachteile

- Risiko für schwer erkennbare Fehler bei Synchronisationen
- Risiko für unerkannte Datenabhängigkeiten in Schleifen
- Läuft nur auf Architekturen mit gemeinsamem Speicher
- Skalierbarkeit und Leistungsausbeute durch die Architektur begrenzt
- Zunächst vermeintlich einfach programmierbar – hohe Leistungsausbeute dann aber nicht einfach erzielbar

Programmierung mit OpenMP
Zusammenfassung

- OpenMP wird ausschließlich für Architekturen mit gemeinsamem Speicher verwendet
- OpenMP ist ein Ansatz, der auf Compiler-Direktiven aufbaut
- OpenMP-Programme mit regulärem Compiler problemlos übersetzbar
- Konstrukte zur Parallelisierung von Schleifen (feingranular) und anderen Code-Bereichen (grobgranular)
- Konstrukte zur Kontrolle der Variableninstanzen in den Threads
- Konstrukte zur Instanziierung von Threads und zu deren Beendigung
- Konstrukte zur Synchronisation der Threads untereinander
- OpenMP bildet mit MPI den Standard der parallelen Programmierung für alle modernen Maschinen

Programmierung mit OpenMP
Die wichtigsten Fragen

- Was charakterisiert den Ansatz von OpenMP?
- Welche Konzeptklassen beinhaltet OpenMP?
- Welche Konstrukte zur Parallelarbeit gibt es?
- Wie werden Variablen verwaltet?
- Welche Synchronisationskonzepte gibt es?
- Wie werden Schleifen parallelisiert?
- Was ist das Hauptproblem der parallelen Schleifen?
- Wofür verwendet man sequentielle Abschnitte?
- Wie programmiert man allgemeine Parallelarbeit?
- Welche Konzepte zum Lastausgleich gibt es?