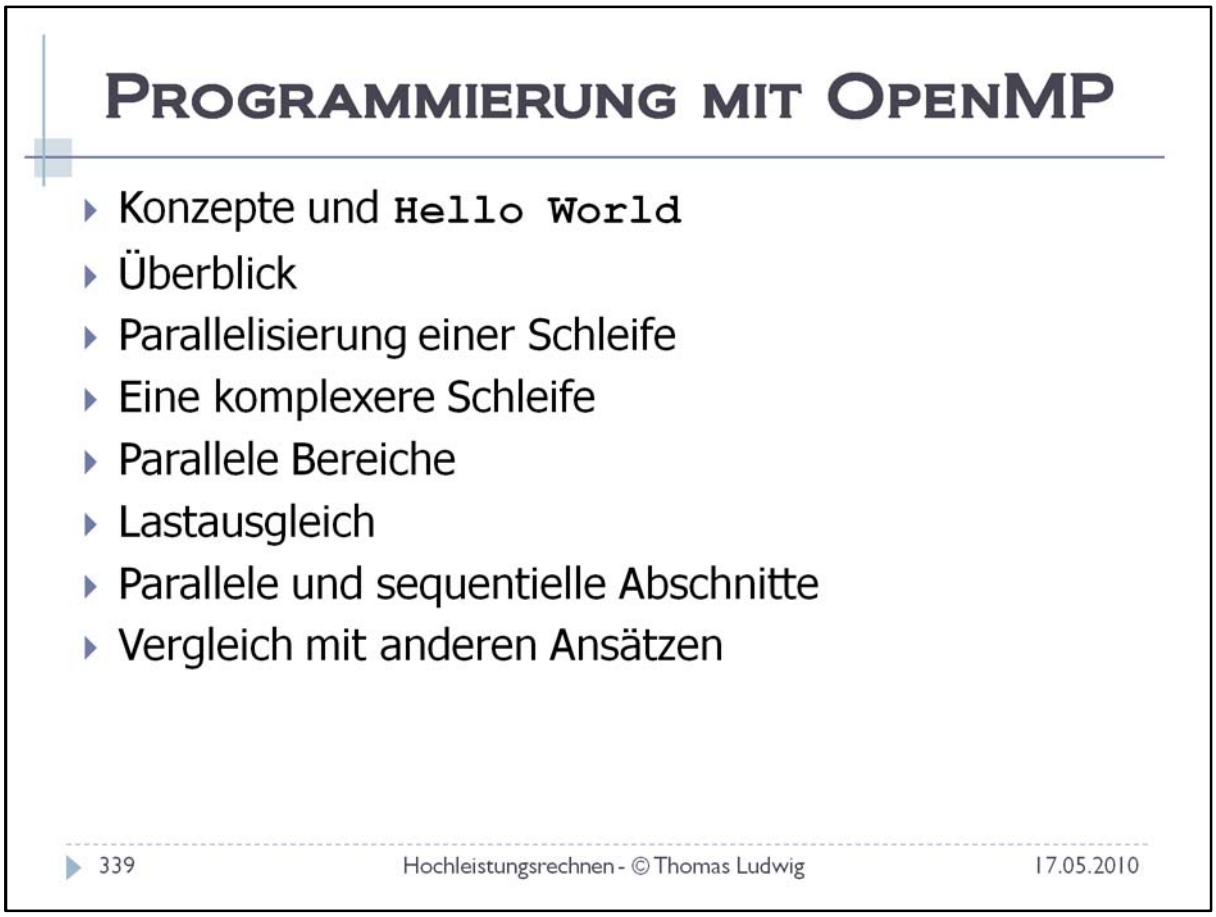


Siehe:
• http://www.openmp.org/
• http://en.wikipedia.org/wiki/OpenMP
• https://computing.llnl.gov/tutorials/openMP/ - sehr gutes Tutorial zum Thema

Bücher:
• R. Chandra et al.: Parallel Programming in OpenMP. Academic Press, London, UK. 2001. 230 Seiten.
• B. Chapman et al.: Using OpenMP: Portable Shared Memory Parallel Programming. Mit Pr, 2007, 353 Seiten.

Was bedeutet OpenMP?
• Kurze Version: Open Multi-Processing
• Lange Version: Open specifications for Multi-Processing via collaborative work between interested parties from the hardware and software industry, government and academia.

Programmierung mit OpenMP
Die zehn 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?

Bisherige Programmiermodelle

Ansätze mit Bibliotheken

- MPI für verteilten Speicher
- Pthreads für gemeinsamen Speicher

Automatische Parallelisierung durch Compiler immer noch nicht möglich

- Trotz langjähriger Forschung weder für Nachrichtenaustausch noch für gemeinsame Speicherbereiche

Aber: Compilergestützte Parallelisierung möglich

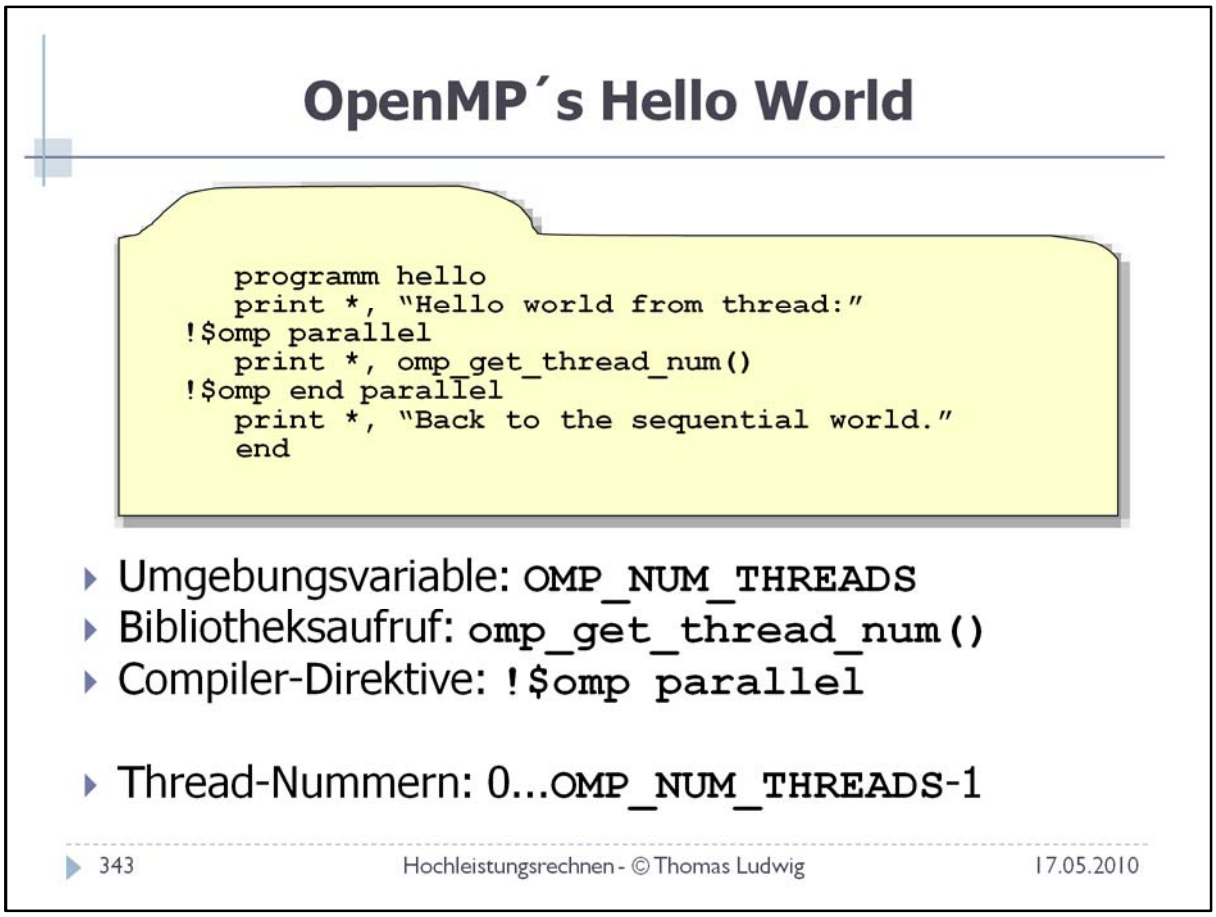
- Im Falle von OpenMP für gemeinsamen Speicher!

Neuer Ansatz: OpenMP

- 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!



Hier sieht man auch sofort alle drei Konstrukte, die OpenMP beinhaltet: Als wichtigstes die Compiler-Direktive, dann ein OpenMP-Bibliotheksaufruf und eine Verwendung von OpenMP-Umgebungsvariablen.

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

Die Geschichte von OpenMP

- OpenMP sehr neu: seit 1997
- Erste Ansätze von SGI vorangetrieben

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

    Früher auf parallelen Maschinen verbreitet

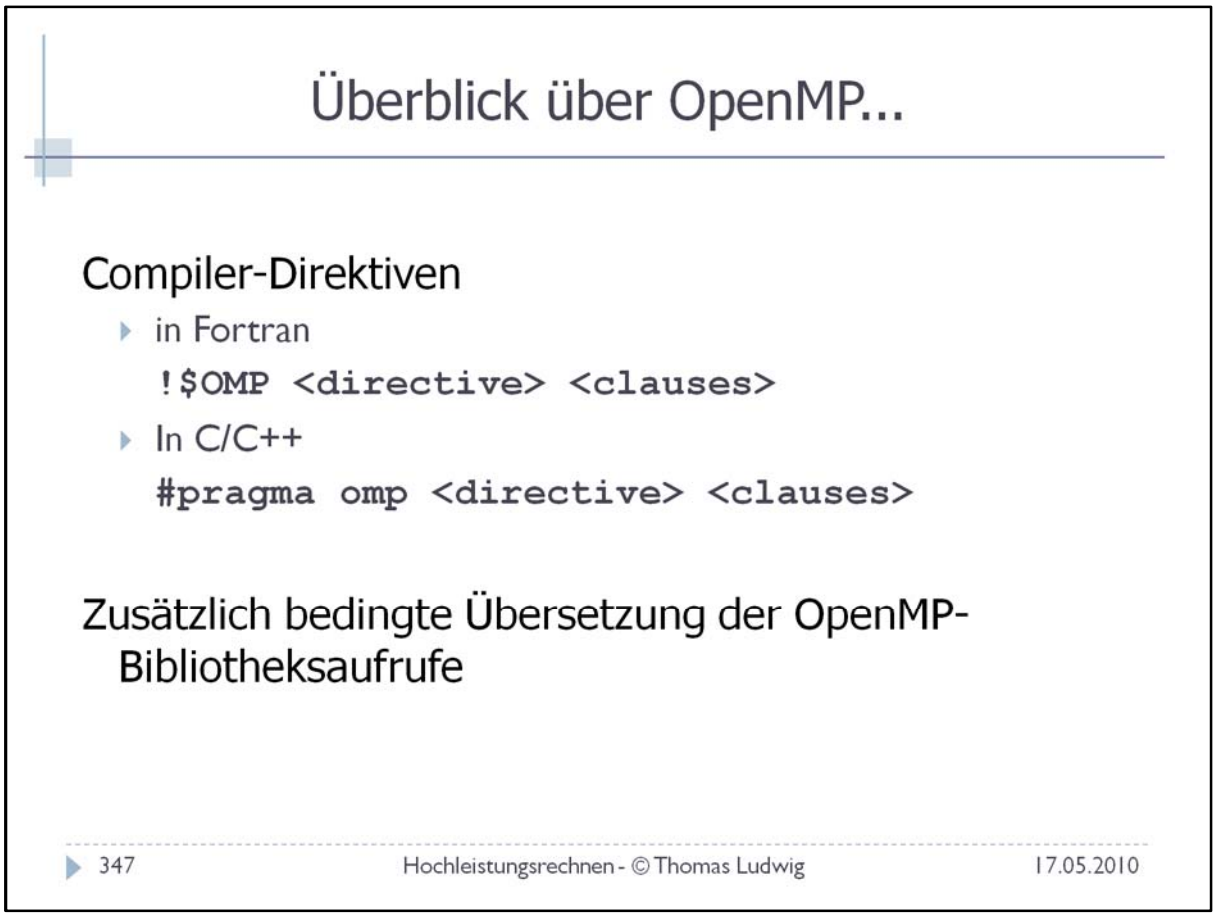
- OpenMP jetzt von allen Herstellern akzeptiert
- Von unabhängiger Organisation gefördert

Überblick über OpenMP

Portabilität: Neuübersetzung reicht aus

Kategorien der Spracherweiterungen

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



Die Compiler-Direktiven sind für einen nicht OpenMP-Compiler nicht wirksam.

Überblick über OpenMP...

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 über OpenMP...

Kommunikation und Datenumgebung

Programm beginnt immer mit einem Thread (master-Thread)

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

Variablen können von folgenden Typen sein

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

Überblick über OpenMP...

Synchronisation

- Regelt den Ablauf der Threads

Zwei Hauptformen

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

Weitere Konstrukte zur Bequemlichkeit oder Leistungsoptimierung

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

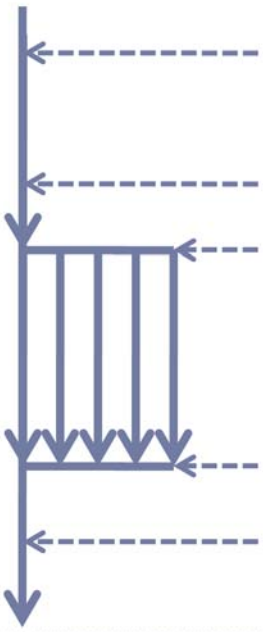
Parallelisierung einer Schleife...

```
      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 Schleifenindexebene

- Andere Ebenen auch möglich

Parallelisierung einer Schleife...



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...

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 wird als thread-lokale Variable angelegt
    Aktualisierungen in einem Thread sind in anderen Threads nicht sichtbar

Parallelisierung einer Schleife...

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

Eine kompliziertere 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 kompliziertere Schleife...

Zur Funktion **`mandel_val`**

- Darf nur von Eingabeparametern abhängen
- D.h. muß wiedereintrittsfähig sein (*reentrant*)

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 kompliziertere Schleife...

```
      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`**

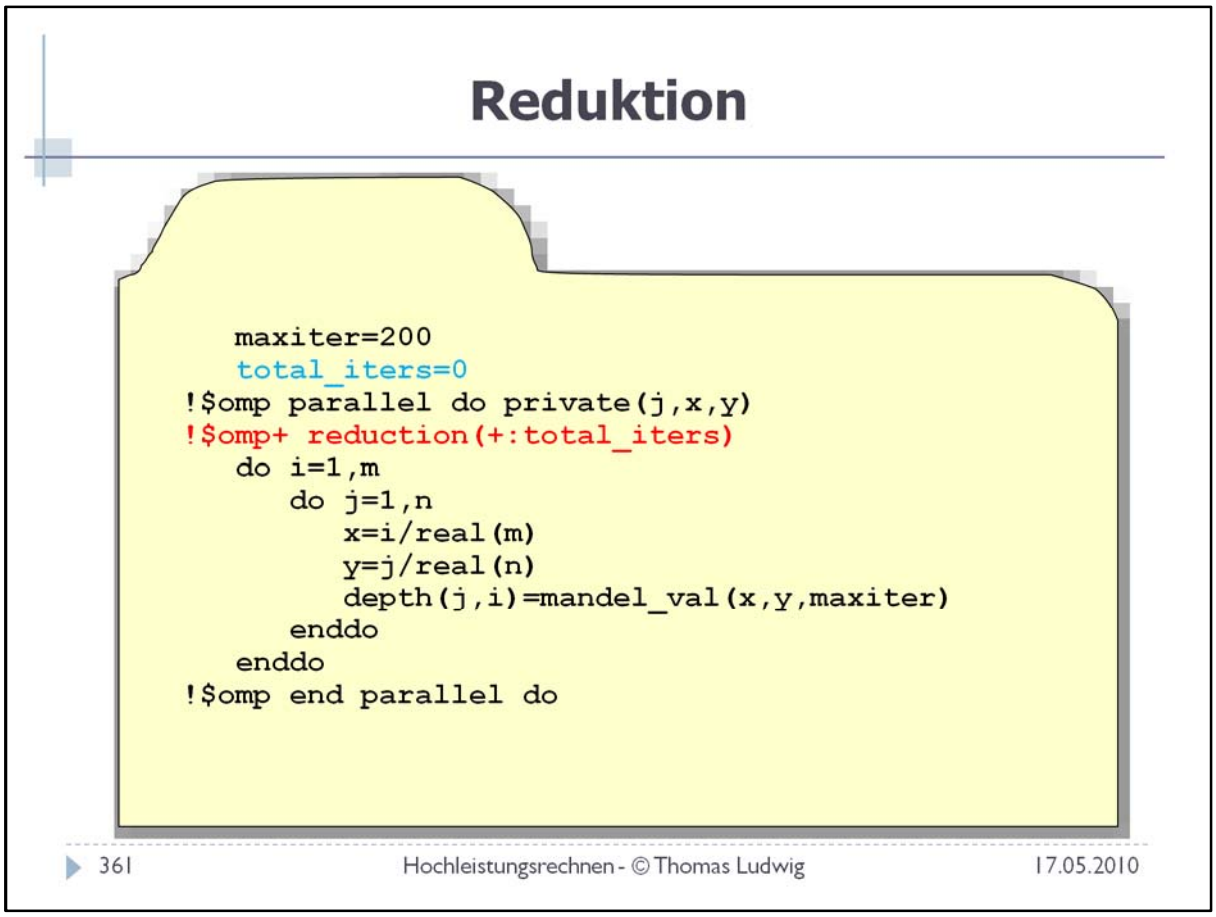
Eine Verkomplizierung...

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!



Die Variable total_iters wird hierbei erst als private-Variable behandelt. Ihre Aktualisierung ist somit unkritisch. Nach Abschluß der Schleife erfolgt die Reduktion, bei der allen privaten Einzelvariablen zur gemeinsam genutzten außerhalb der Schleife aufaddiert werden.

Das ´+´ bei ´!$omp+´ kennzeichnet die Fortsetzung der vorhergehenden Zeile. Fortran-Notation.

Schleifenparallelisierung

Hauptproblem der Praxis:

Datenabhängigkeiten zwischen Schleifenindizes

Bespiel:

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

Lösung des Problems

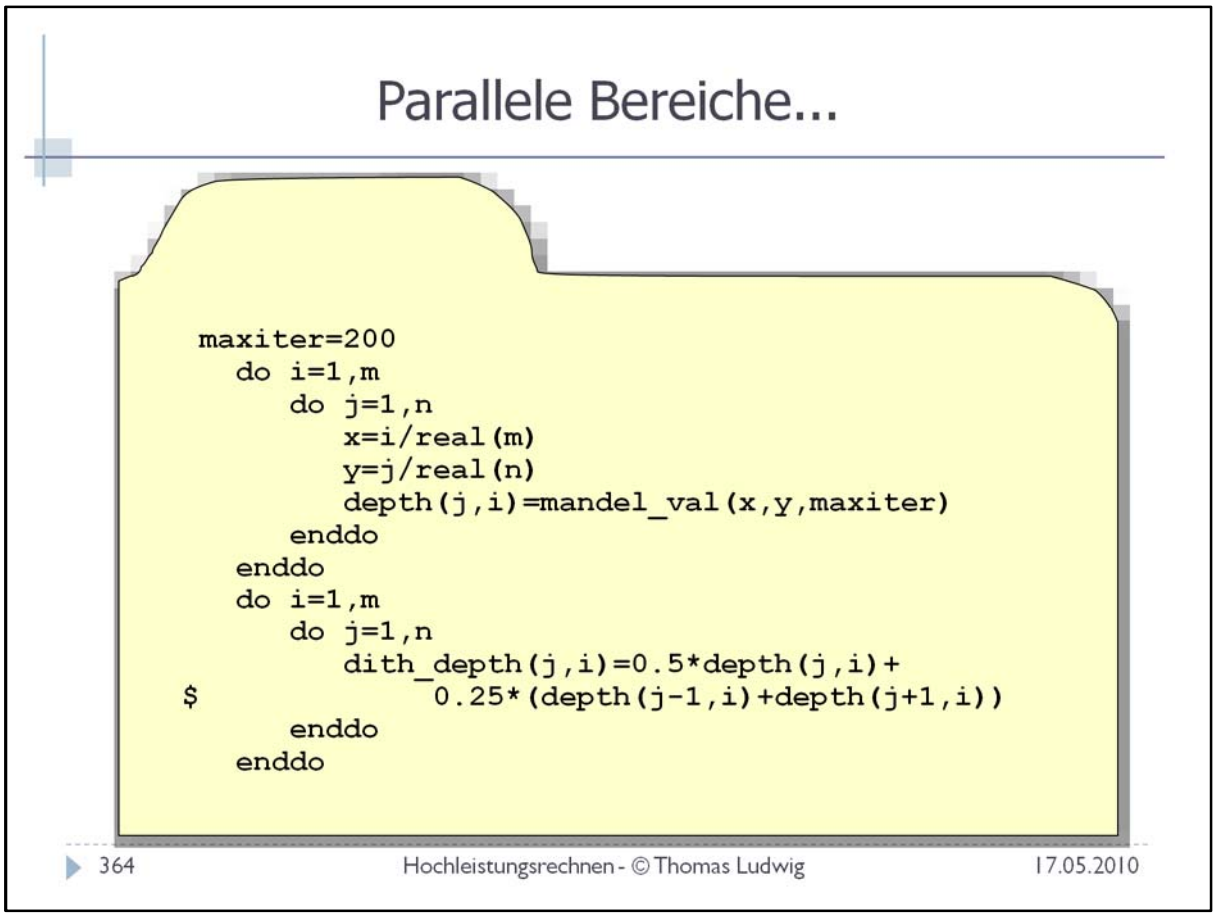
- Komplizierte Methoden zum Finden der Abhängigkeiten
  - Großes Forschungsgebiet seit vielen Jahren
- Verschiedene Methoden zu ihrer Beseitigung
  - Zusätzliche Variable
  - Zugriffskoordination mit kritischem Bereich

Parallele Bereiche

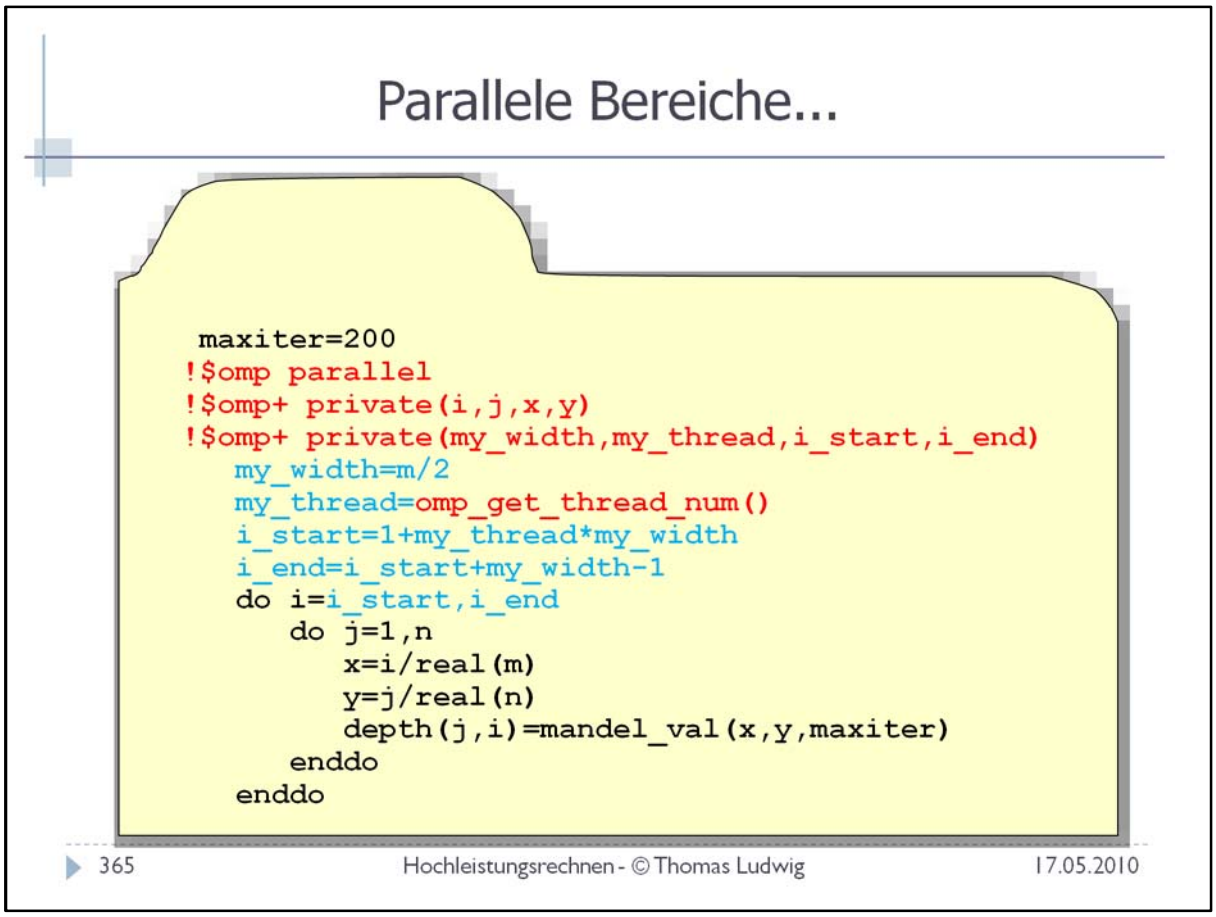
Bisher nur parallele Schleifen (feingranular)
Jetzt auch grobgranularer Parallelismus

Konstrukt: `parallel / end parallel`

- eingeschlossener Code wird mit mehreren Threads nebenläufig bearbeitet



Zwei Einzelberechnungen in Schleifen nacheinander ausgeführt.



Das Ganze wird hier auf 2 Threads aufgeteilt. Jeder Thread ermittelt seine Nummer und bestimmt daraus den Block von Indizes, die er durchlaufen muß.

Parallele Bereiche...

```
      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

Parallelisierung nicht auf Schleifenebene

Lastausgleich

Standard: jeder Thread erledigt gleich viele Iterationen einer Schleife

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

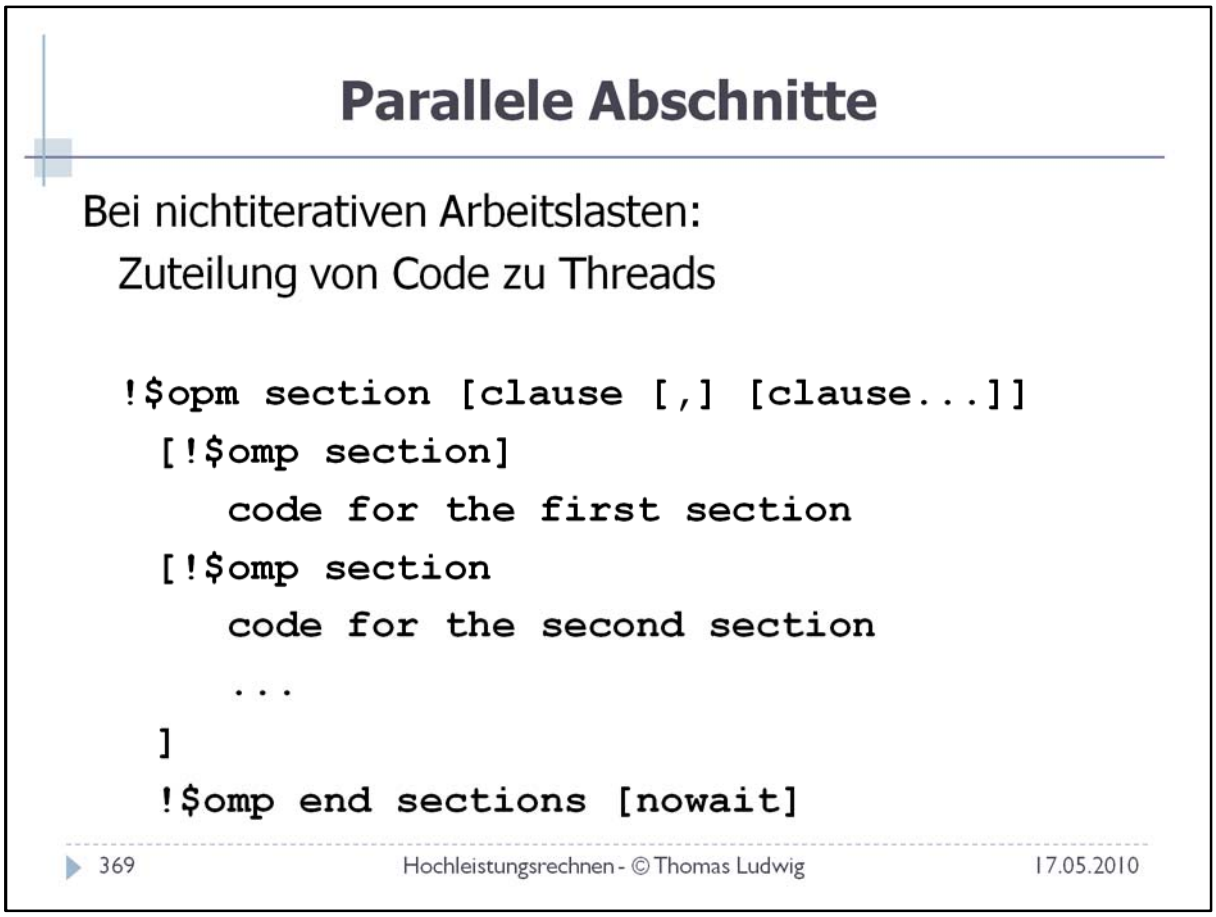
Mechanismus: `schedule`-Klausel

- `static`: Zuteilung der Indizes zu Schleifen-beginn
- `dynamic`: Indizes werden zur Laufzeit zugeteitl

Lastausgleich...

`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



Mit Klauseln kann man wieder z.B. die Verwaltung der Variablen oder das Scheduling steuern.

Parallele Abschnitte...

- 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

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...

Beispiel: Ausgabe

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

Dynamische Threads

In Mehrbenutzerumgebungen Threadanzahl nicht optimal einstellbar

- Zu viele Threads: Verluste durch Umschalten
- Zu wenige Threads: Nicht genutzte Ressourcen

OpenMP bietet dynamische Anpassung der Threadanzahl durch Laufzeitumgebung

Umgebungsvariable `OMP_DYNAMIC`

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)

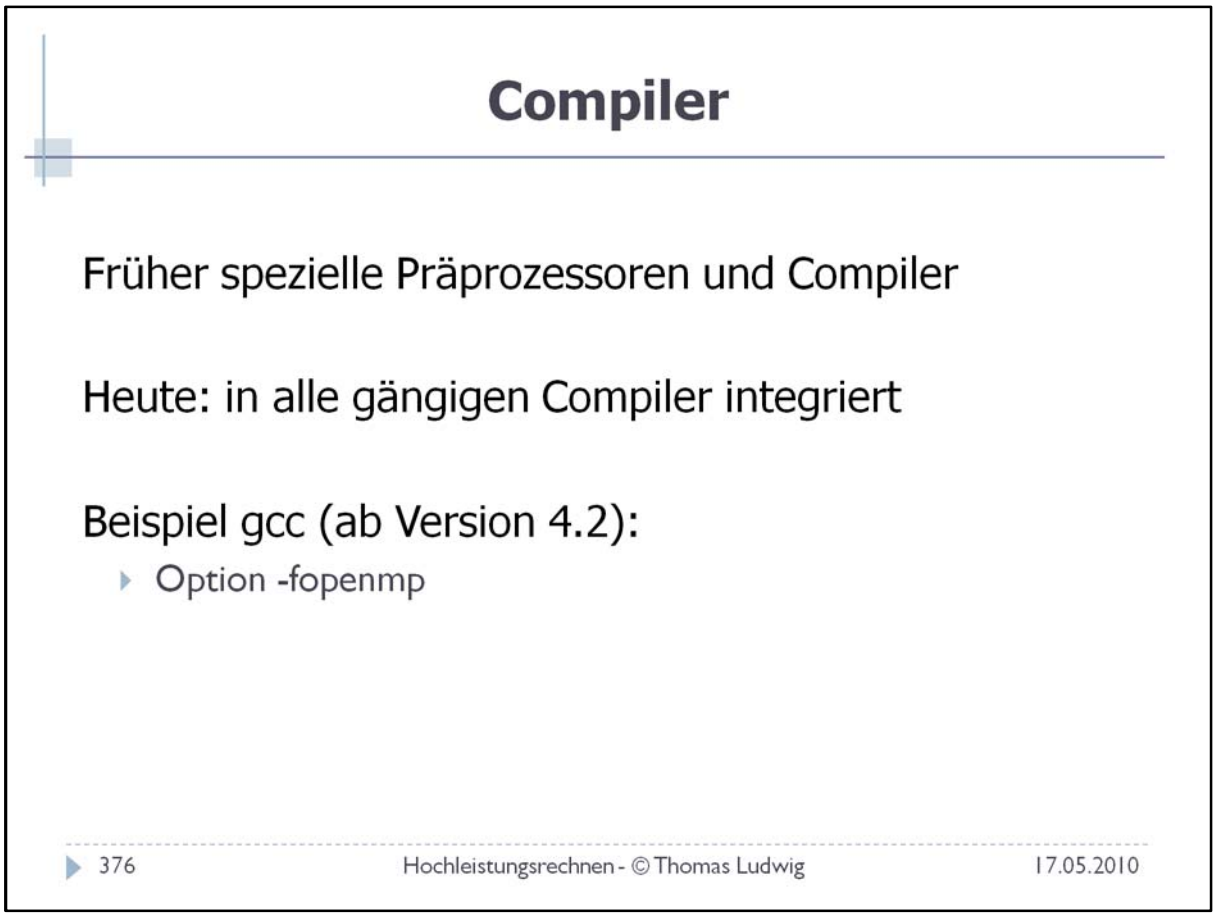
```
!$omp ordered
   block
!$omp end ordered
```

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*



Siehe: http://openmp.org/wp/openmp-compilers/

Vergleich der Ansätze

	Pthreads	OpenMP	MPI
Skalierbarkeit	Begrenzt	Begrenzt	Ja
Fortran / C und C++	Ja? / Ja	Ja / Ja	Ja / Ja
Hohe Abstraktion	Nein	Ja	Nein
Leistungsorientierung	Nein	Ja	Ja
Portierbarkeit	Ja	Ja	Ja
Herstellerunterstützung	Unix/SMP	Verbreitet	Verbreitet
Inkrement. Parallelisierung	Nein	Ja	Nein

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