Wissenschaftliches Programmieren

Bálint Aradi
http://www.bccms.uni-bremen.de/cms/people/b-aradi

11. Parallele Programmierung – Erste Schritte mit OpenMP

Projektarbeit

Aufgabe:

- Programm zur Lösung der eindimensionalen Schrödingergleichung
- Das zugrundeliegende numerische Verfahren (Diskretisierung der Gleichung) ist in der Beschreibung angegeben.
- Die Interpolationsalgorithmen (um das vom Benutzer angegebene Potential auf einen äquidistanten Gitter abzubilden) müssen selber „hergeleitet“ werden. (Polynominterpolation kann z.B. in

Press et al., Numerical Recipes: the art of scientific computing

nachgeschaut werden).

Bewertungsablauf

1. Projekt wird eingereicht

   Spätester Abgabetermin: 20.09.2017

Besperchungstermin wird vereinbart (die Woche danach)

2. Die eingereichte Arbeit wird von mir digital signiert und als Bestätigung zurückgeschickt.

3. Die eingereichte Arbeit wird von mir begutachtet.

4. Es findet eine Besprechung statt, in der eventuelle Fragen an den Entwickler gestellt werden und die Arbeit besprochen wird.

Wichtige Bemerkungen

- Das Projekt ist jeweils **zu zweit zu bearbeiten**. (In von mir genehmigten Ausnahmefällen zu dritt)
- Das **Entwicklungsgeschichte** (mindestens 4-5 Revisionen) muss **mit git** festgehalten werden, wobei **beide Entwickler ein eigenes Repository** haben müssen, die zweckmäßig oft synchronisiert werden sollten.
- Das Projekt ist **inklusive Repository abzugeben**.
- Es sollte im sauberen **Fortran 2003** geschrieben werden und sollte **mehrere Module** enthalten.
- Das Programm muss eine **LAPACK/LAPACK95-Routine** aufrufen
- Es sollte ein **Makefile** vorhanden sein, das die Modulabhängigkeiten korrekt abbildet.
- Die **Schnittstellen** sind **mit** Hilfe von **Doxygen** ausreichend zu **dokumentieren**.
- **README-Datei** mit entsprechenden Informationen muss vorhanden sein.
- Es sollen **automatische Tests** bereitgestellt werden, die sicherstellen, dass der compilierte Programm so funktioniert, wie die Entwickler das vorgesehen haben. (So wie in der entsprechenden Vorlesung/Übung).

Achtung!

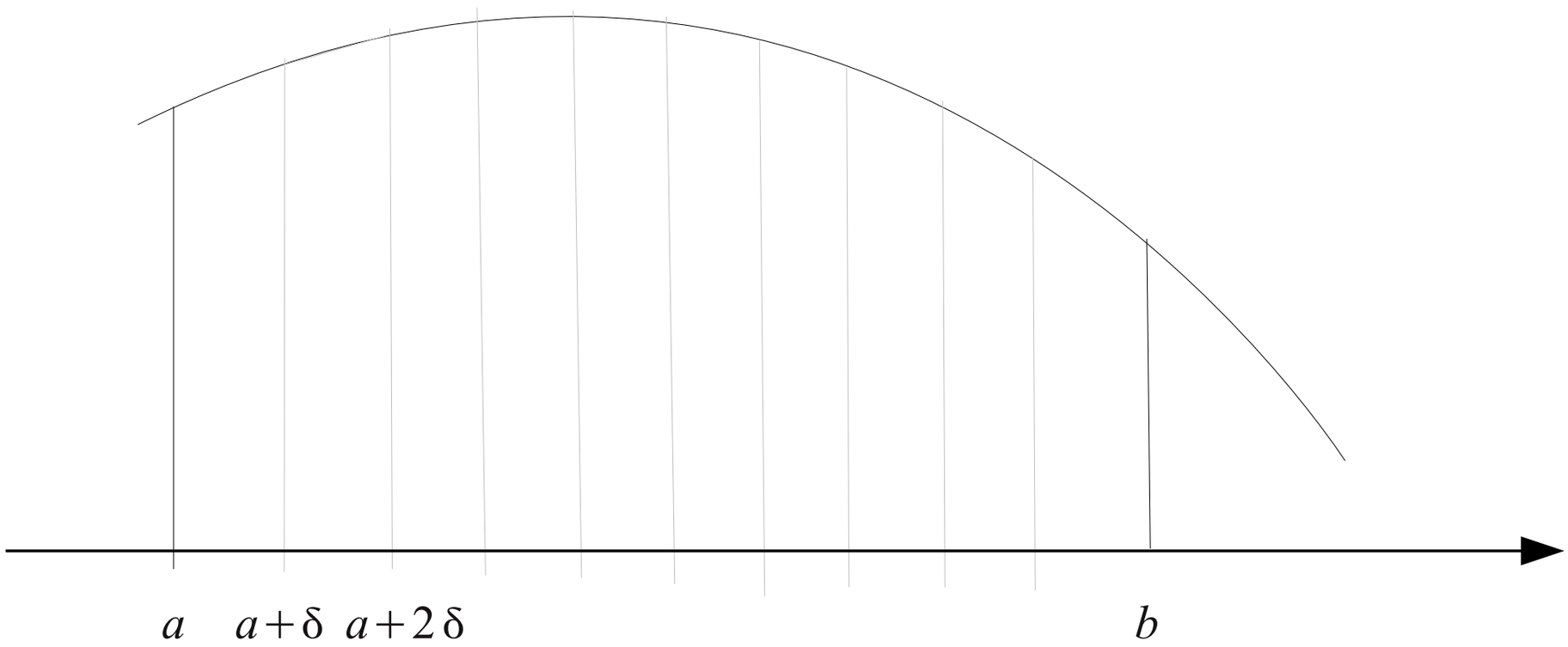
- **Nicht funktionierende Programme** (nicht compilierbar, es kommen offensichtlich falsche Zahlen für die vorgegebene Beispiele auf, das Programm kann die vorgegebenen Inputdateien nicht einlesen, etc.) gelten als nicht bewertbar!
- Das Programm muss in der Linuxumgebung **im Poolraum** ohne weiteres **compilierbar** sein!
- Es sollte auch die **Numerik verstanden** werden (warum weichen die Ergebnisse von den tatsächlich erwarteten ab, wovon hängt die Genauigkeit der Ergebnisse ab, wie kann sie verbessert werden, etc.)
- Es sollte sicher gestellt werden, dass die **Ausgabedateien die Spezifikationen** erfüllen! (also keine write(*,*)-Befehle für diese verwenden!)

Paralleles Programmieren: Aufgabenstellung

- Wir haben ein rechnerisches Problem und viele Computer/Prozessoren/Rechenkerne

  Wie kann das Problem so schnell wie möglich gelöst werden?

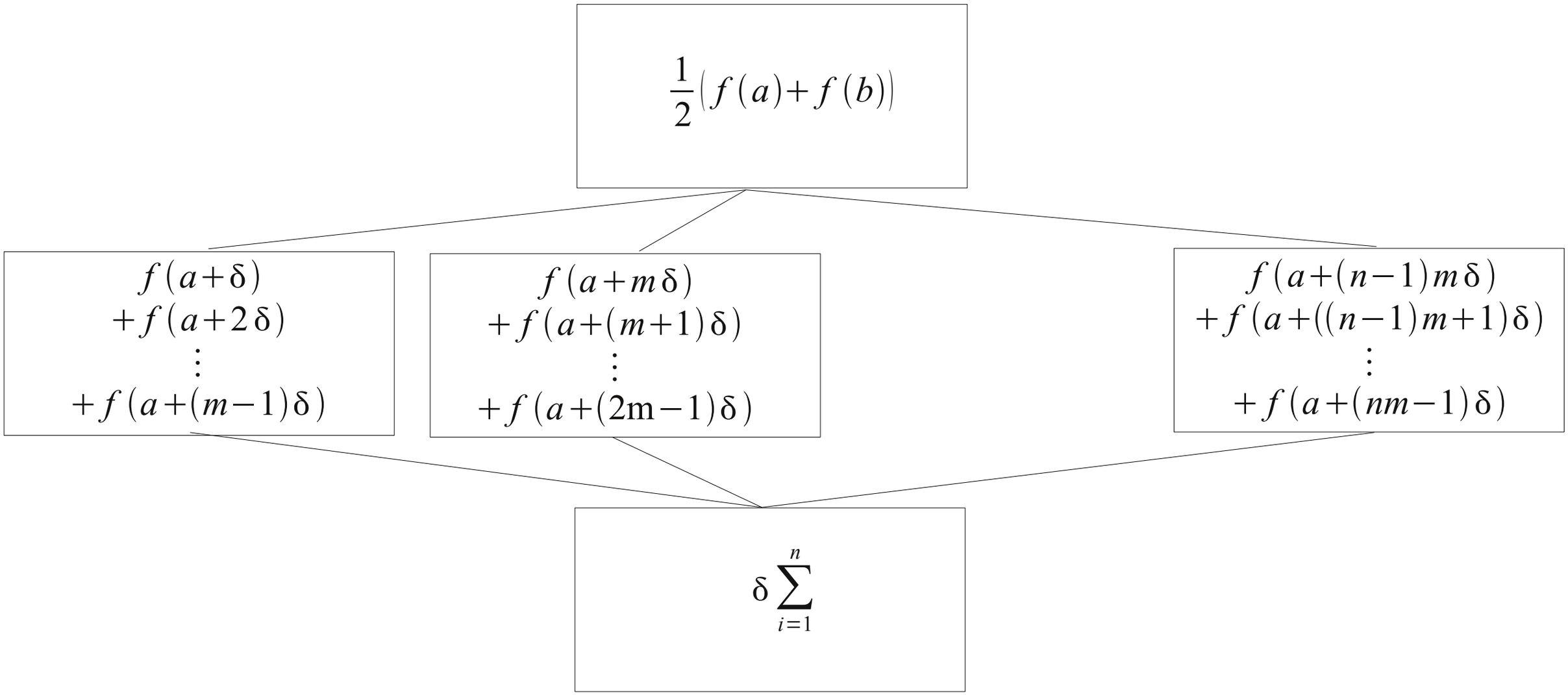
- Beispiel: 1D-Integration einer Funktion:



$$\int_a^b f(x)\,dx \approx \delta\left(\frac{1}{2}f(a)+f(a+\delta)+f(a+2\delta)+\cdots+\frac{1}{2}f(b)\right)$$

Wie kann das auf mehreren Rechenkernen verteilt werden?

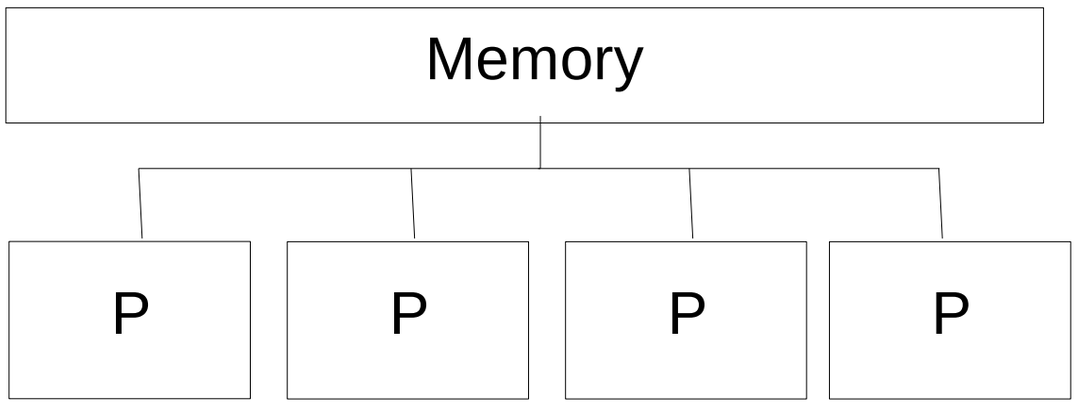
Grundlegende Überlegungen



- Welche Funktion wird integriert?
- Welche Punkte muss der aktuelle Prozess integrieren?
- Teilergebnisse müssen aufsummiert werden (Kommunikation zwischen den Prozessen)

Speichermodelle

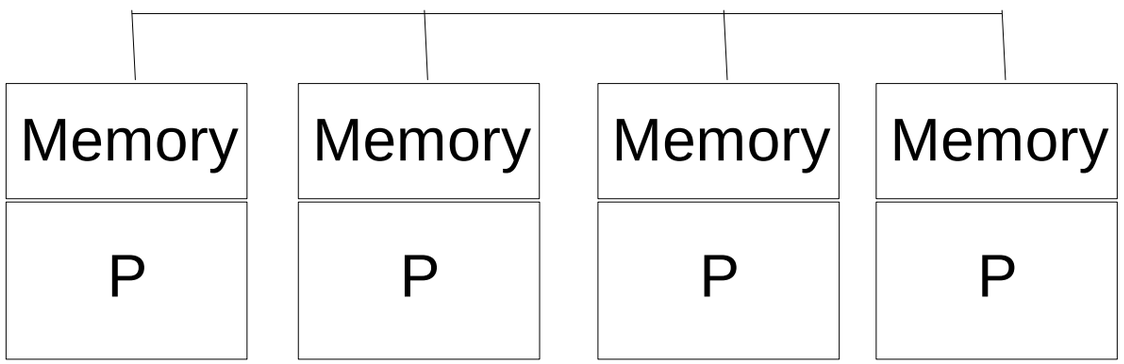
Gemeinsam benutzer Speicher (*shared memory*):



Bsp.: SMP- oder Multicorerechner

- Gemeinsamer Speicher für alle Prozessoren
- Zugriffgeschwindigkeit auf diverse Speicherbereiche entweder gleich (*uniform-memory access*) oder leicht unterschiedlich (*non-uniform memory access*)
- Datenaustusch über den gemeinsamen Speicher möglich

Verteilter Speicher (*distributed memory*):



Bsp.: Rechencluster

- Jeder Prozessor hat eigenen Speicher mit schnellem Zugriff
- Datenaustausch zwischen den Prozessoren mit speziellem Protokoll (z.B. MPI).
- Zugriffgeschwindigkeit auf Speicher von anderen Prozessoren im Allg. langsam (z.B. über Netzwerk)

OpenMP vs. MPI

- Unterschiedlichen Speicherarchitekturen erfordern unterschiedliche Parallelisierungen
- Beispiele:

  Gemeinsam benutzer Speicher:

    - Pthread (Bibliothek)
    - OpenMP (Compilererweiterung!)

  Verteilter Speicher:

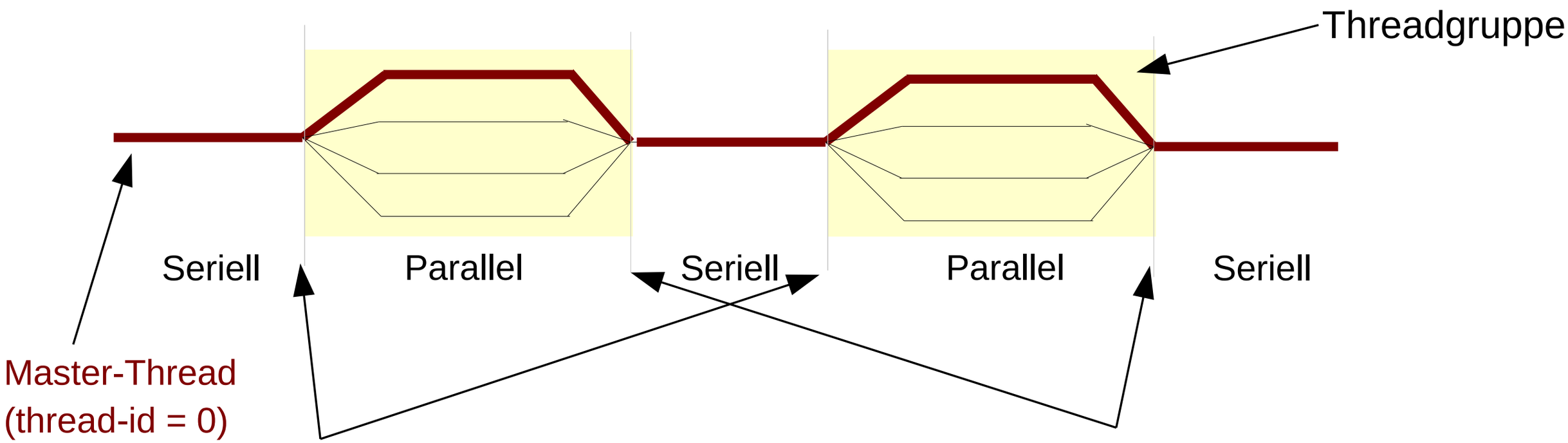
    - PVM (Parallel virtual machine, Bibliothek)
    - MPI 1, MPI 2, MPI 3 (message passing interface, Bibliothek)

Wir werden OpenMP behandeln.

- OpenMP ist ein Standard, der eine **Compilererweiterung** definiert.
  Es ist keine Bibliothek!
- OpenMP ist für Fortran und C/C++ definiert
- OpenMP-Direktiven werden nur von Compilern unterstützt, die diesen Standard umsetzten.
- Die gängigsten Fortran- und C-Compiler implementieren OpenMP 3 oder 4.

OpenMP

- Funktioniert nach dem ***fork-and-join***-Prinzip:



  - Serielles **Hauptprogramm** (Master) **spannt** an geigneten Stellen parallele **Threads auf** (*fork*)
  - **Operation** wird parallel ausgeführt
  - **Ergebnis** wird **eingesammelt**, **threads** wieder **eliminiert** (*join*)
  - **Hauptprogramm läuft seriell weiter**

- Einfache Parallelisierung, mit möglich wenig Veränderung im Quellcode
- Programm enthält spezielle Instruktionen fürs Managen von parallelen Threads

Hello world! – OpenMP

```
program hello
  use omp_lib
  implicit none

  integer :: tid, nthread

  !$OMP PARALLEL PRIVATE(tid)
  tid = omp_get_thread_num()
  write(*,*) "Hello world, from thread = ", tid

  if (tid == 0) then
    nthread = omp_get_num_threads()
    write(*,*) "Number of threads: ", nthread
  end if
  !$OMP END PARALLEL

end program hello
```

Einbinden von OpenMP Funktionen und Subroutinen

OpenMP-Direktive
Aufspannen von Threads
Paralleler Teil beginnt

ID des aktuellen Threads

Nur vom Master-thread ausgeführt

OpenMP-Direktive
Ende des parallelen Teils

Anzahl der Threads in der Threadgruppe

HelloWorld Compilieren, Ausführen

- Es muss dem Compiler (und beim getrennten Linken auch dem Linker!) explizit mitgeteilt werden, dass die speziellen OpenMP-Kommentare berücksichtigt werden sollen, und ein OpenMP-fähiges Binary erzeugt werden soll.

```
gfortran -fopenmp -c hello.f90      gfortran -fopenmp -o hello hello.o
```

- Vor dem Ausführen muss über eine entsprechende Umgebungsvariable dem Programm mitgeteilt werden, wieviele Threads er in den parallelen Teilen benutzen soll.

```
export OMP_NUM_THREADS=2
echo $OMP_NUM_THREADS
2
```

- Ausführen, wie ein gewöhnliches Programm:

```
./hello
```

```
Hello world, from thread =           0
Number of threads:           2
Hello world, from thread =           1
```

Reihenfolge der Ausgabe bei parallel ausgeführten I/O-Befehlen indeterministisch!

Variablen in OpenMP-Programme

- Variablen sind in OpenMP-Programme per Default geteilt (*shared*), sodass alle Threads mit der selben Instanz der Variable arbeiten.
- Muss eine Variable in unterschiedlichen Threads unterschiedliche Werte annehmen, muss diese explizit als *private* gekennzeichnet werden, sodass jeder Thread eine eigene Kopie (Instanz) bekommt.

```
program hello
  use omp_lib
  implicit none

  integer :: tid, nthread

  !$OMP PARALLEL PRIVATE(tid)
  tid = omp_get_thread_num()
  write(*,*) "Hello world, from thread = ", tid

  if (tid == 0) then
    nthread = omp_get_num_threads()
    write(*,*) "Number of threads: ", nthread
  end if
  !$OMP END PARALLEL

end program hello
```

- Variable tid **privat**
- jeder Thread bekommt eine eigene Variable.
- Setzen/Verändern dieser Variable hat keine Auswirkung auf den Wert in den anderen Threads

Globale Variable, jeder Thread sieht die Änderungen (*nach genügend Zeit*)

Parallelisierung von Schleifen

- OpenMP enthält Konstrukte, um Schleifen einfach zu parallelisieren

```
real(dp) :: aa(10), bb(10), cc(10)
:
!$OMP PARALLEL PRIVATE(ii, tid)
tid = omp_get_thread_num()
!$OMP MASTER
nthread = omp_get_num_threads()
write(*,*) "Number of threads =", nthread
!$OMP END MASTER
write(*,*) "Thread ", tid, " starting..."

!$OMP DO
do ii = 1, 10
  cc(ii) = aa(ii) + bb(ii)
  write(*,"(' Thread',I2,': C(',I3,')=',F8.2)") &
      &tid, ii, cc(ii)
end do
!$OMP END DO

write(*,*) "Thread ", tid, "done."
!$OMP END parallel
```

Fork:
Variablen *ii*, *tid* private
für jeden Thread

Nur vom Master ausgeführt
(statt if (tid == 0) then ...)

Schleife wird auf Threads aufgeteilt
Thread0 macht ii = 1, 5
Thread1 macht ii = 6, 10

Schleife zu Ende (optional)

Join

Parallelisierung von Schleifen

```
export OMP_NUM_THREADS=2
./a.out | sort
```

Ausgabe von a.out alphabetisch sortieren

```
 Number of threads =           2
  Thread           0  starting...
  Thread           0 done.
  Thread           1  starting...
  Thread           1 done.
  Thread 0: C(  1)=    2.00
  Thread 0: C(  2)=    4.00
  Thread 0: C(  3)=    6.00
  Thread 0: C(  4)=    8.00
  Thread 0: C(  5)=   10.00
  Thread 1: C(  6)=   12.00
  Thread 1: C(  7)=   14.00
  Thread 1: C(  8)=   16.00
  Thread 1: C(  9)=   18.00
  Thread 1: C( 10)=   20.00
```

Parallelisierung von Schleifen

- Parallelisierung von Schleifen nur dann möglich, wenn die Operationen unabhängig von vorherigen oder späteren Iterationen sind:

Falsch

```
integer :: ii, jj
integer :: aa(100)

jj = 5
do ii = 1, 100
  jj = jj + 2
  aa(ii) = func(jj)
end do
```

Ergebnis eines Durchlaufs hängt vom Ergebnis der vorherigen Durchläufe ab! Nicht parallelisierbar, setzt eine sequentielle Ausführung voraus!

Richtig

```
integer :: ii, jj
integer :: aa(100)

do ii = 1, 100
  jj = 5 + 2 * ii
  aa(ii) = func(jj)
end do
```

Durchläufe für verschiedene Werte von ii völlig unabhängig voneinander. Ausführungsreihenfolge beliebig, parallelisierbar.

Parallelisierung von Schleifen

- Private Variablen haben weder beim *fork*, noch beim *join* einen definierten Wert:

Falsch

```
integer :: ii
ii = 12
!$OMP PARALLEL PRIVATE(ii)
  ii = func(ii)
  print *, "In thread:", ii
!$OMP END PARALLEL
print *, "After join:", ii
```

ii hat keinen definierten Wert

ii hat keinen definierten Wert
(vom welchem Thread?)

- Mit dem firstprivate-Konstrukt lässt sich die Variable initialisieren

FIRSTPRIVATE = PRIVATE + Initialisierung

```
integer :: ii
ii = 12
!$OMP PARALLEL FIRSTPRIVATE(ii)
  ii = func(ii)
  print *, "In thread:", ii
!$OMP END PARALLEL
print *, "After join:", ii
```

OK, ii hat den Wert 12 in jedem Thread
Jeder Thread hat eine eigene Kopie von ii.

Das ist noch immer undefiniert!

Kurzform

- Wenn die parallele Threads nur für die Dauer einer Schleife gestartet werden sollen, kann die PARALLEL und die DO Direktive in kompakter Form geschrieben werden:

```
!$OMP PARALLEL DO PRIVATE(ii, res)
 do ii = 1, nn
   res = res + (aa(ii) * bb(ii))
 end do
!$OMP END PARALLEL DO
```

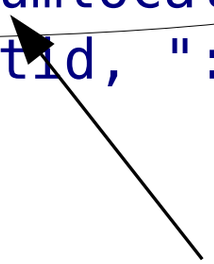
Reduktion

```
program reduce
  use omp_lib
  implicit none

  integer :: aa(10), totsum, sumlocal, tid, ii

  aa = 1
  totsum = 0
  !$OMP PARALLEL PRIVATE(tid, sumlocal, ii)
  tid = omp_get_thread_num()
  sumlocal = 0
  !$OMP DO
  do ii = 1, 10
    sumlocal = sumlocal + aa(ii)
  end do
  !$OMP END DO
  totsum = totsum + sumlocal
  write(*,*) "TID=", tid, ": sumlocal=", sumlocal, ", total=", totsum
  !$OMP END PARALLEL

end program reduce
```



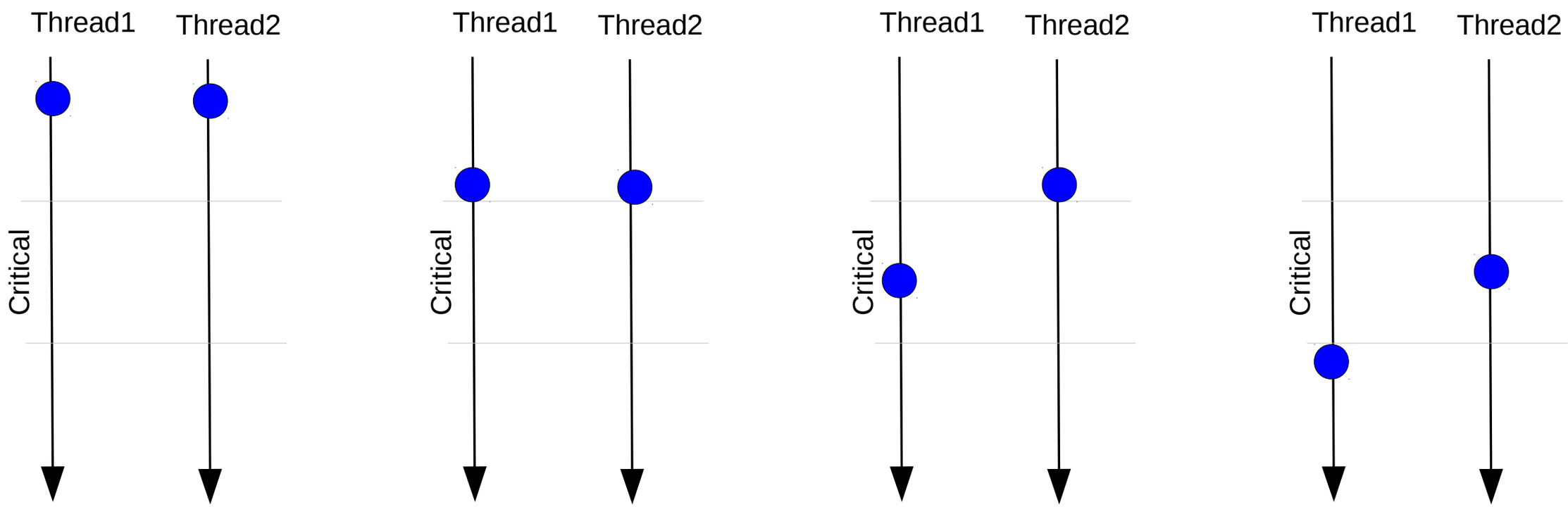
„race condition“: zwischen Auslesen von totsum und Zurückschreiben des erhöhten Wertes kann sich totsum geändert haben!

Critical

- Mit Hilfe des critial-Konstrukt läßt sich sicherstellen, dass ein gewisser Programmteil zur gleichen Zeit nur von einem Thread ausgeführt wird.
- Wird der Codeteil schon von einem Thread ausgeführt, warten andere Threads, bis dieser fertig ist.
- Verlangsamt den Code! (Nur Verwenden, wenn wirklich nötig!)

```
!$OMP CRITICAL (name)
:
!$OMP END CRITICAL (name)
```

(name) optional



Reduktion

```
program reduce
  use omp_lib
  implicit none

  integer :: aa(10), totsum, sumlocal, tid, ii

  aa = 1
  totsum = 0
  !$OMP PARALLEL PRIVATE(tid, sumlocal, ii)
  tid = omp_get_thread_num()
  sumlocal = 0
  !$OMP DO
  do ii = 1, 10
    sumlocal = sumlocal + aa(ii)
  end do
  !$OMP END DO
  !$OMP CRITICAL (update_sum)
  totsum = totsum + sumlocal
  write(*,*) "TID=", tid, ": sumlocal=", sumlocal, ", total=", totsum
  !$OMP END CRITICAL (update_sum)
  !$OMP END PARALLEL

end program reduce
```

Reduktion

- Mit Hilfe von reduction lassen sich einfache Reduktionsoperation automatisch ausführen
- Format:

!$ OMP DO … REDUCTION(Operator:Variable)

!$ OMP END DO

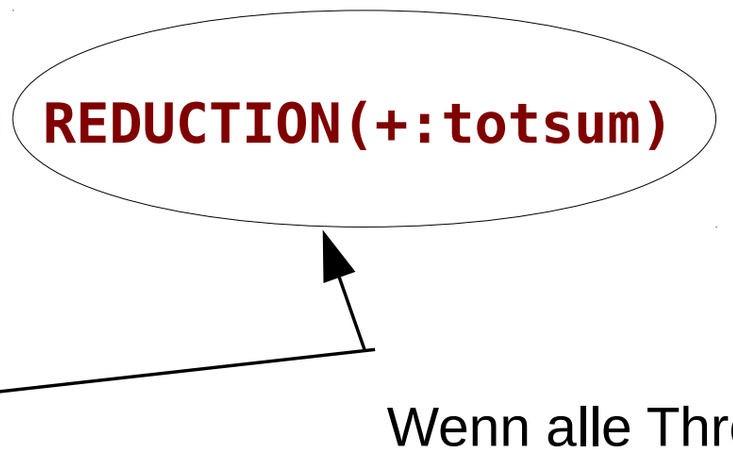
```
program reduction
  use omp_lib
  implicit none

  integer :: aa(10), totsum
  integer :: ii

  aa = 1
  totsum = 0
  !$OMP PARALLEL DO PRIVATE(ii) REDUCTION(+:totsum)
  do ii = 1, 10
    totsum = totsum + aa(ii)
  end do
  !$OMP END PARALLEL DO

  write(*,*) "total=", totsum

end program reduction
```



Wenn alle Threads ihre Arbeit beendet haben, werden die Werte in den *totsum* Variablen aufsummiert und in der *totsum* Variable des Masterthreads gespeichert.

Unterschieldliche Anweisungen für Threads

- Bis jetzt: Jeder Thread führt die gleichen Operationen (auf anderen Teil der Daten) aus
- Mit OpenMP können unterschiedliche Threads unterschiedliche Operationen ausführen.

```
!$OMP PARALLEL SHARED(aa, bb, cc, dd)

!$OMP SECTIONS

!$OMP SECTION
cc = aa + bb

!$OMP SECTION
dd = aa * bb

!$OMP END SECTIONS

!$OMP END PARALLEL
```

Operationen für Thread1

Operationen für Thread2

Bereich, wo unterschiedliche Threads unterschiedliche Befehle ausführen sollen

- Es können beliebig viele SECTION-Blöcke spezifiziert werden
- Sind mehr Blöcke als Threads spezifiziert, werden manche Threads mehr als einen Block ausführen

Workshare-Konstrukt

- OpenMP enthält ein spezielles Konstrukt (workshare) für Fortran90-Array-Operationen
- Es ermöglicht, die Fortra90-Array-Operation zu verwenden, ohne Schleifen schreiben zu müssen:

```
!$OMP PARALLEL SHARED(aa, bb, cc)
!$OMP WORKSHARE
  bb(:) = bb(:) + 1
  cc(:) = cc(:) + 2
  aa(:) = bb(:) + cc(:)
!$OMP END WORKSHARE
!$OMP END PARALLEL
```

- Funktioniert mit folgenden Fortran90-Operationen:
  - Arrayzuweisungen
  - FORALL-Konstrukte
  - WHERE-Konstrukte

- Compiler muss sicherstellen, dass die Reihenfolge der Operationen in den einzelnen Threads (und die Synchronisation zwischen den Threads) zum richtigen Ergebnis führt
- Allerdings keine Kontrolle über die Aufteilung der Arbeit auf die einzelne Threads möglich.

OpenMP und parallele Programmierung im Allg.

Das war nur eine Minieinführung in OpenMP und parallele Programmierung

- **Wichtige wesentliche Konzepte** fehlen noch:
  Prozessorarchitekturen (insb. Cache, Speicherzugriff)
  Synchronisierung von Threads, Barriere, Flushing, etc.
  Schleifenanorndungen mit Hinsicht auf Parallelisierung
  Programmoptimierung für Parallelisierung
  :
- Es würde noch ein Semester in Anspruch nehmen, eine richtige Einführung in parallele Programmierung zu geben.
- Es gibt allerdings gute Bücher für OpenMP (und auch für MPI)
  z.B.: P. S. Pacheco: Parallel Programming with MPI; Chapman et al., Using OpenMP

Paralleles Programmieren ist essentiell für wissenschaftliches Programmieren:
Multicore-Architekturen
Berechnungen auf Graphikkarten (z.B. CUDA)
Rechencluster, Supercomputer (Cray, etc.)

Unterprogramme als Parameter

- Wenn ein Unterprogramm mit Hilfe einer Benutzerfunktion bzw. einer Benutzersubsubroutine etwas erledigen soll, kann diese als Parameter übergeben werden.

- Entsprechender Parameter muss die Funktion/Subroutine mit Hilfe eines Interface-Blocks spezifizieren.

- Das übergebene Unteprogramm muss die Spezifikation erfüllen.

- Übergebenes Unterprogramm muss in einem Modul definiert sein.

- Es können keine interne Unterprogramme übergeben werden.

```
real(dp) function myPoly(x0)
  real(dp), intent(in) :: x0
  :
end function myPoly
```

```
 :
 call getroot(myPoly, from, to, xx)
 :
```

```
 real(dp) function getroot(func, from, to)
   interface
     function func(xx)
       import :: dp
       real(dp), intent(in) :: xx
       real(dp) :: func
     end function func
   end interface
   real(dp), intent(in) :: from, to
   :
   real(dp) :: ff, alpha
   :
   ff = func(alpha)
   :
 end function getroot
```

Importiert Namen 'dp'

Aufgabe (fortgeschrittene Programmierung)

- Schreiben Sie eine Integrationsroutine, die die Trapezintegration anwendet.
- Die Routine soll als Parameter eine 1D reelle Funktion, den Anfang und das Ende des Integrationsintervalles und die Anzahl der Integrationspunkte empfangen, und das numersiche Integral als Ergebnis zurückgeben.
- Schreiben Sie ein Modul, das die Testfunktionen sin(x)**2 und xx**2 implementiert, und ein Testprogramm, das mit der Integrationsroutine diese beide Funktionen auf dem Intervall [0, 1] bzw. [0, 2*Pi] auswertet.
- Checken Sie das Ergebnis in ein Repository ein! (Vergessen Sie die Dokumentation via Doxygen und das korrekte Makefile nicht!)
- Schreiben Sie die Integrationsroutine mit Hilfe von OpenMP so um, dass diese sinnvoll parallelisiert wird.
- Testen Sie die Laufzeiten mit verschiedenen Anzahl von OpenMP-Threads auf einer Multicore-Maschine. (Verwenden Sie dabei so viele Integrationspunkte, dass die Integration mit einem Thread mind. 5 Sekunden dauert.)
- Wie skalieren die Laufzeiten, wenn Sie von 1 auf 2 Threads gehen?
- Sind die Ergebnisse in den beiden Fällen exakt die selben? Wenn nein, wieso?