Karlsruhe Institute of Technology

Paralleles Programmieren mit OpenMP und MPI

OpenMP

Vorlesung “Parallelrechner und Parallelprogrammierung”, SoSe 2016

Hartmut Häfner, Steinbuch Centre for Computing (SCC)

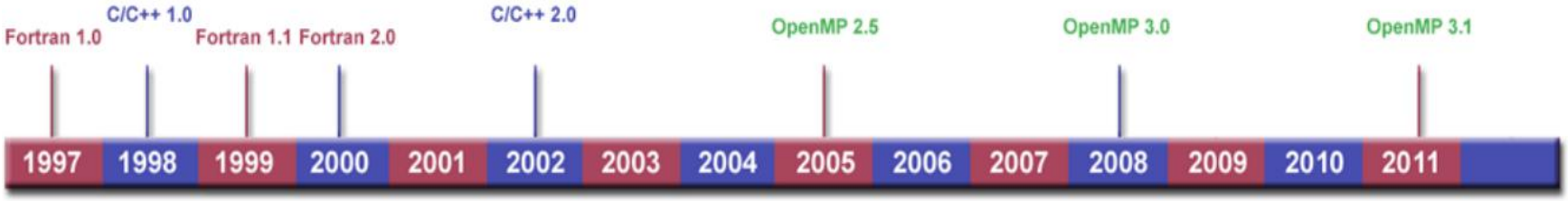
STEINBUCH CENTRE FOR COMPUTING - SCC

Informationen zu OpenMP

- OpenMP Homepage:
  http://www.openmp.org/
- OpenMP User Group:
  http://www.compunity.org
- OpenMP Tutorial:
  https://computing.llnl.gov/tutorials/openMP/

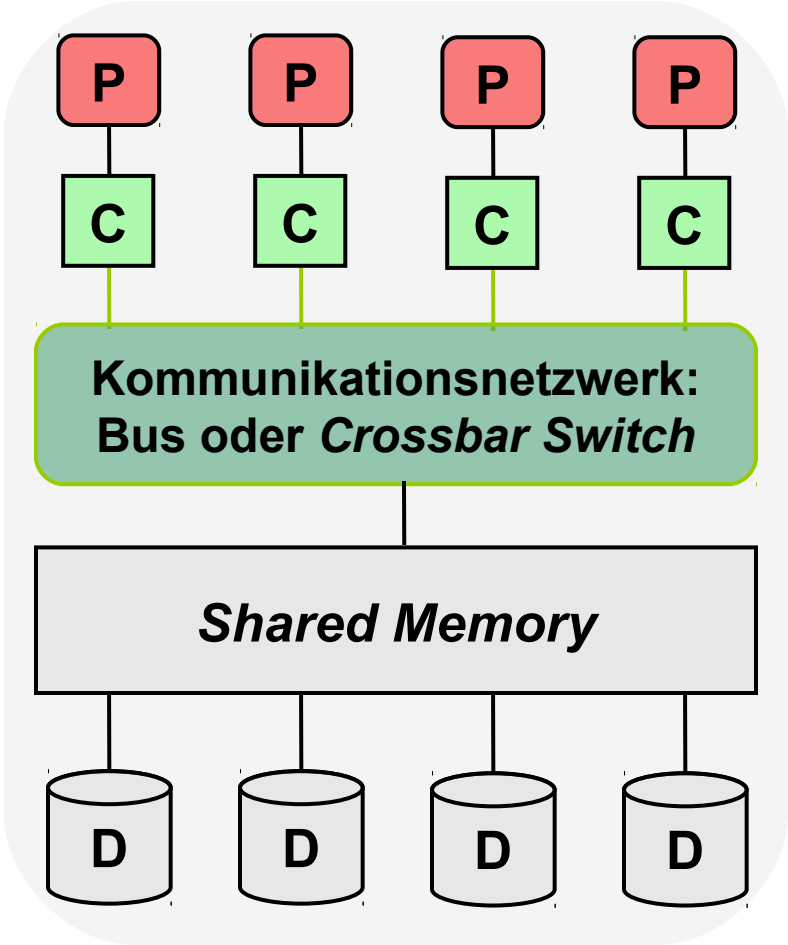
- R.Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon:
  Parallel programming in OpenMP.
  Academic Press, San Diego, USA, 2000, ISBN 1-55860-671-8
- R. Eigenmann, Michael J. Voss (Eds):
  OpenMP Shared Memory Parallel Programming.
  Springer LNCS 2104, Berlin, 2001, ISBN 3-540-42346-X
- Simon Hoffmann, Rainer Lienhart: OpenMP.
  www.eBook.de, 2009, ISBN 3-540-73122-9

Zeitstrahl zu OpenMP



- OpenMP 3.0 beinhaltet Task-Konzept
- OpenMP 4.0 wurde am 23. Juli 2013 freigegeben und enthält Heterogenität, d.h. Unterstützung für Beschleunigerkarten
- OpenMP 4.5 wurde im November 2015 freigegeben, unterstützt Acceleratoren/Grafikkarten und bietet unterschiedliche Locking-Mechanismen

Shared Memory System



- Gesamtes System verhält sich wie ein einzelnes Computersystem.
- Alle Prozessoren haben Zugriff auf das *Shared Memory*.
- Eine Kopie des Betriebssystems.
- Parallelisierung über *Shared Memory* oder *Message Passing*
  - OpenMP
  - MPI
    Message Passing über *Shared Memory*
- Beispiele: SMP *workstations*, NUMA Knoten der HPC-Systeme in Karlsruhe

Wichtige Merkmale von OpenMP

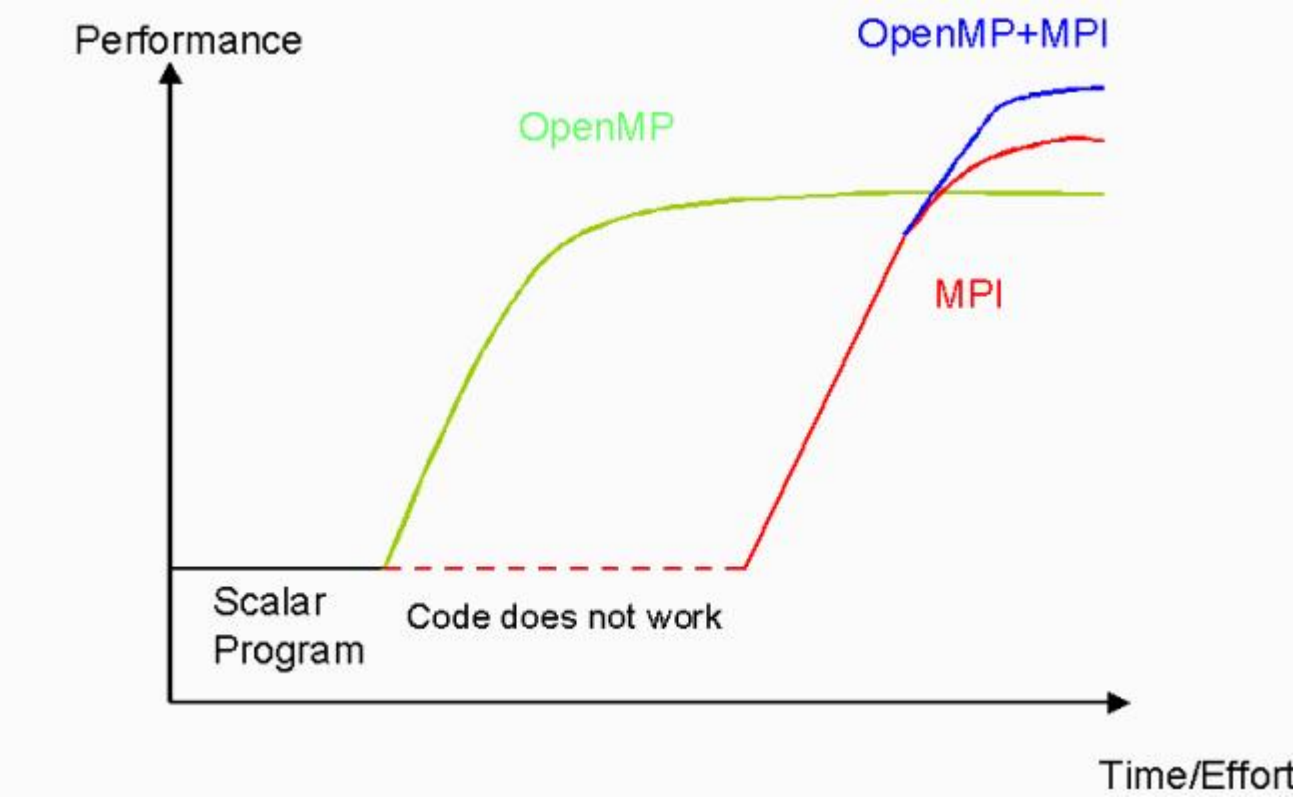
- Daten werden im *Shared Memory* gespeichert
- Ein Prozess beim Start der Applikation
- Parallele *threads* werden während der Programmausführung erzeugt und zerstört
- *Threads* können auf gemeinsame oder private Daten zugreifen
- Grobkörnige oder feinkörnige Parallelisierung

Parallelisierung - Grundidee

- *Parallel Region*
  - Code innerhalb einer *Parallel Region* wird von allen erzeugten Threads ausgeführt
- *Work sharing* Konstrukte
  - für DO-Schleifen (`!$OMP DO` bzw. `#pragma omp for`)
    Einzelne Schleifenindizes werden von unterschiedlichen *threads* abgearbeitet
  - für Codesegmente (`!$OMP SECTIONS` bzw. `#pragma omp sections`)
    Per `!OMP SECTION` bzw. `#pragma omp section` gegeneinander abgegrenzte Codesegmente werden von unterschiedlichen *threads* abgearbeitet

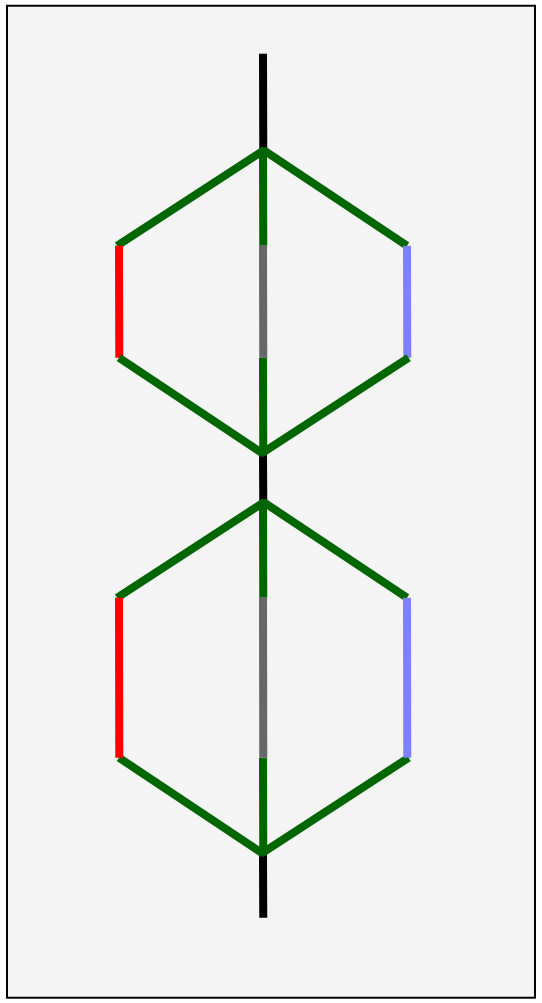
- *Work Sharing* Konstrukte müssen innerhalb von *Parallel Regions* liegen!

Wann wird OpenMP eingesetzt?



Feinkörnige Parallelisierung

```
program main
...
!$OMP PARALLEL DO
do i=1,n
      !work
end do
!$OMP END PARALLEL DO
...
!$OMP PARALLEL SECTIONS
!$OMP SECTION
...
!$OMP SECTION
...
!$OMP END PARALLEL SECTION
....
end
```



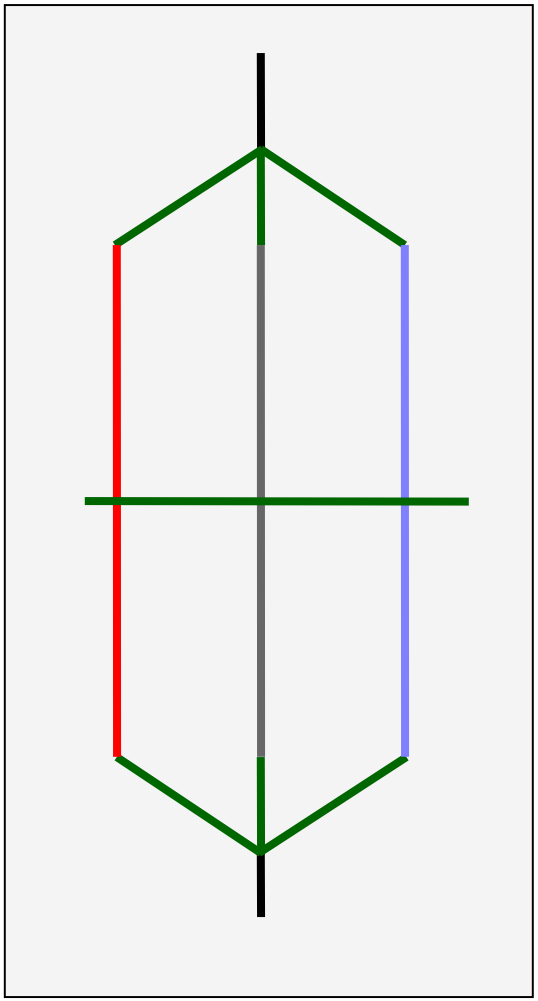
Grobkörnige Parallelisierung

```
program main

!$OMP PARALLEL
size = OMP_GET_NUM_THREADS()
iam  = OMP_GET_THREAD_NUM()

call work(a, b, n, size, iam)
!$OMP BARRIER
do i= 1, n/size
   ...
end do
...

!$OMP END PARALLEL
end
```



OpenMP Überblick

- Direktiven
  - Fortran: `!$OMP …` C: `#pragma omp …`
  - *Parallel regions* Konstrukt
    - `Fortran: !$OMP PARALLEL … !$OMP END PARALLEL`
    - `C: #pragma omp parallel{ … }`
  - *Work sharing* Konstrukte
    - `Fortran: !$OMP DO … !$OMP END DO`
    - `C: #pragma for …`
  - Synchronisierungskonstrukte
    - `!$OMP BARRIER`
  - *Variable scoping* Klauseln und Direktiven
    - `Fortran: !$OMP THREADPRIVATE (/CBLCK/[,/CBLCK/]…)`
    - `C: #pragma omp threadprivate (list of global vars)`
- Laufzeitbibliothek (ein paar Routinen)
- Umgebungsvariablen

OpenMP - typischer Gebrauch

- OpenMP wird gebräuchlicherweise benutzt um Schleifen zu parallelisieren:
  - Finde die Schleifen, die am meisten Zeit verbrauchen (`prof`, `gprof`, `Xprofiler`, `Intel Advisor XE 2013`).
  - Teile sie auf mehrere *threads* auf.

Teile diese Schleife auf mehrere *threads* auf.

```
void main()
{
    double Res[1000];

    for(int i=0;i<1000;i++)
{
       do_huge_comp(Res[i]);
    }
}
```

Sequentielles Programm

```
void main()
{
    double Res[1000];
#pragma omp parallel for
    for(int i=0;i<1000;i++)
{
       do_huge_comp(Res[i]);
    }
}
```

Paralleles Programm

OpenMP Direktiven

Karlsruhe Institute of Technology

Direktiven: `!$OMP directive clauses`

`#pragma omp directive clauses`

- `parallel`
  - `private:` Liste mit privaten Variablen
  - `shared:` Liste mit globalen Variablen
  - `firstprivate:` Wert der privaten Variable wird vom *master thread* in alle *threads* kopiert
  - `lastprivate:` Wert der "sequentiell letzten" Variablen wird in *master thread* kopiert
  - `reduction(op:variable)`: Operator: `+,-,*,/,&&,||,==`
  - `schedule`
- `do bzw. for:` Worksharing-Konstrukt Schleife
- `sections:` Worksharing-Konstrukt (Code-)Sektionen
- `critical:` Zu jedem beliebigen Zeitpunkt führt nur ein *thread* diese Anw. aus
- `master:` Nur der *master thread* führt das eingeschlossene Codesegment aus
- `single`: Nur der zeitlich erste *thread* führt das eingeschlossene Codesegment aus; die anderen *threads* warten bis zur Beendigung
- `barrier:` Alle *threads* warten an der Barriere aufeinander
- `atomic:` Atomare Operation wird als kritischer Abschnitt behandelt

Wichtige Direktive: `OMP PARALLEL`

- **`OMP PARALLEL` - Direktive startet *Parallel Region***
- **Innerhalb `[clauses]` wird u.a. spezifiziert, welche Variablen privat und welche gemeinsam fuer die *threads* sind.**

- **Fortran:**

```
!$OMP PARALLEL [clauses]
 block
!$OMP END PARALLEL
```

- **C:**

```
#pragma omp parallel [clauses]
 {block}
```

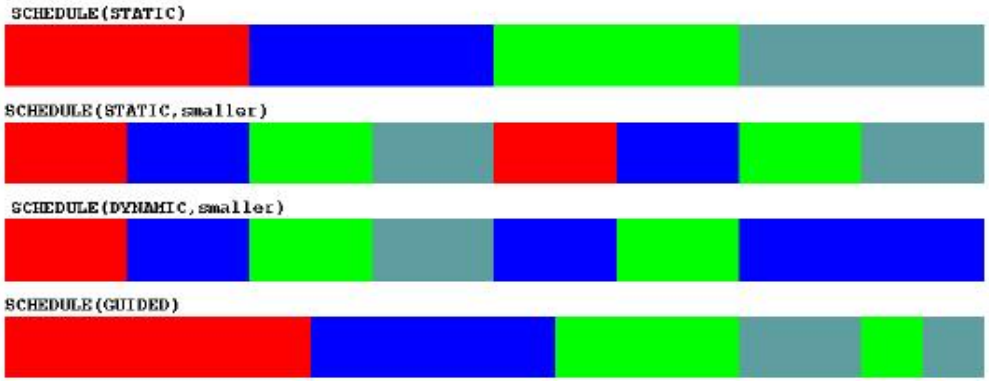
- **`[clauses]` können u.a. sein**
  - `private(list)`
  - `shared(list)`
  - `schedule(type[,chunk])`

Die Klausel `schedule`

Karlsruhe Institute of Technology

Der Typ bei der Klausel `schedule` kann sein:

- **`static`: Iterationen werden in Stücke der Größe `chunk` unterteilt. Die Stücke werden statisch den *threads* in einem *team* zugewiesen**
- **`dynamic`: Iterationen werden in Stücke der Größe `chunk` unterteilt. Sobald ein *thread* einen Brocken des Iterationsraums abgearbeitet hat, erhält er dynamisch den nächsten Brocken**
- **`guided`: Jeder *thread* bekommt ein Stück der Größe `chunk`. Ist ein *thread* fertig, bekommt er ein neues Stück, dessen Größe abfällt**
- **`runtime:` Schedule durch Umgebungsvariable "OMP_SCHEDULE" bestimmt**



Wichtige Direktive: `OMP DO`

- **Fortran:**

```
!$OMP DO [clauses]
Fortran DO Konstrukt
[!$OMP END DO [NOWAIT] ]
```

- **C:**

```
#pragma omp for [clauses]
for Schleife
```

- **Die Schleife, die sofort auf diese Direktive folgt, wird auf alle *threads* des parallelen *team* aufgeteilt**
- **Schleife muss die Syntax haben**

  `do i = i1, i2 [ ,i3 ]` **bzw.**

  `for`**-Schleife muß kanonische Gestalt haben**
- **Anzahl der Iterationen muss bekannt sein, wenn die Schleife gestartet wird**

Direktive: OMP SECTIONS

- **Fortran:**

```
!$OMP PARALLEL [clauses]
!$OMP SECTIONS
  a= …
!$OMP SECTION
  b= …
!$OMP SECTION
  c= …
!$OMP END SECTIONS [NOWAIT]
!$OMP END PARALLEL
```

C:

```
#pragma omp parallel
{
#pragma omp sections
 {{ a= … ;}
#pragma omp section
   { b= … ;}
#pragma omp section
   { c= … ;}
 } /* end sections */
} /* end parallel */
```

- **Die vorhandenen *sections* werden zwischen den parallelen *teams* aufgeteilt. Jede *section* wird von einem *thread* einmal durchlaufen**

OpenMP 3.0 Direktive: `OMP TASK`

- Direktive `TASK` erzeugt eine Task (Arbeitspaket: Code + Daten)
  - Auf `TASK`-Konstrukt treffender Thread verpackt Arbeitspaket
  - Ausführung der Task kann verzögert werden
  - Task kann von beliebigem *thread* des *team* bearbeitet werden
- Ähnlich zu dem Konstrukt `OMP SECTIONS`
- Vermeidet zu viele verschachtelte *Parallel Regions*
- Erlaubt irreguläre Probleme zu parallelisieren (z.B. rekursive Algorithmen)

OpenMP 4.5 Direktive: `OMP TASKLOOP`

- **Direktive `TASKLOOP` nutzt OpenMP-Tasks zur Ausführung und ermöglicht so z.B. die Verschränkung von Tasks und einer normalen Schleife**

```
#pragma omp taskgroup
{
#pragma omp task
   long_running_task()    // kann nebenher ablaufen

#pragma omp taskloop collapse(2) grainsize(500) nogroup
   for (int i = 0; i < N; i++)
      for (int j = 0; j < M; j++)
         loop_body();
}
```

`nogroup` schaltet implizit vorhandene `taskgroup` um das Konstrukt ab → Vermeiden von automatischer Synchronisation mit den erzeugten Tasks

Gültigkeitsbereich von Variablen

- *Shared Memory* Programmiermodell:
  - Die meisten Variablen sind standardgemaess gemeinsam (*shared*).
- Globale Variablen sind gemeinsam unter den *threads* (*shared among threads*).
  - Fortran: COMMON Blöcke, SAVE Variablen, MODULE Variablen
  - C: *File scope* Variablen, static
- Aber nicht alles ist shared...
  - Stapelvariablen in Unterprogrammen, die von *parallel regions* aufgerufen werden, sind privat.
  - Automatische Variablen innerhalb eines Anweisungsblocks sind privat.
- Einige Schleifenindizes sind standardmäßig privat:
  - Fortran: Schleifenindizes sind auch dann privat, wenn sie als *shared* spezifiziert sind.

Umgebungsvariablen

- `OMP_NUM_THREADS`
  - Setzt die Anzahl der *threads*, die während der Ausführung benutzt werden
  - Bei dynamischer Einstellung der Anzahl der *threads* ist der Wert der Umgebungsvariable der Maximalwert *laufender* threads
  - sh, ksh, bash: `export OMP_NUM_THREADS=16`

- `OMP_SCHEDULE`
  - Wir nur bei do/for oder parallel do/parallel for Direktiven angewandt mit der Klausel `schedule(runtime)`
  - Setzt `type` und `chunk` für alle Schleifen mit obiger Klausel
  - sh, ksh, bash: `export OMP_SCHEDULE="STATIC,4"`

Shared memory Programmierfehler

- *Race conditions*
  - *Data-race:* Mindestens 2 *threads* greifen auf die gleiche *shared variable* zu und zumindest 1 *thread* modifiziert die Variable und die Zugriffe finden “gleichzeitig” und nicht synchronisiert statt (oftmals bei nicht beabsichtigtem Gebrauch von *shared data*)

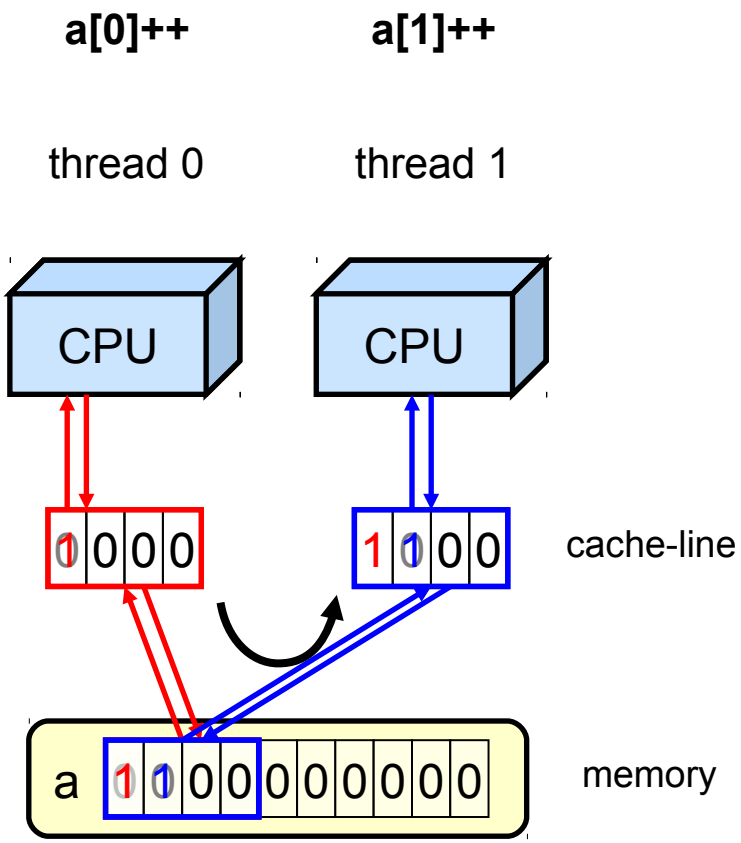
- *Deadlock*
  - Threads warten auf eine *locked* Ressource, die niemals den Zustand *unlocked* erreicht (vermeide das Schachteln von verschiedenen *locks)*

Beispiel für eine *race condition*

```
!$OMP PARALLEL SECTIONS
       a = b + c
!$OMP SECTION
       b = a + c
!$OMP SECTION
       c = b + a
!$OMP END PARALLEL SECTIONS
```

- Die Ergebnisse variieren unvorhersehbar
- Keine Warnung von Seiten des Programms

False-sharing



- Mehrere *threads* greifen schreibend!!! auf Daten derselben *cacheline* zu
- Die *cacheline* muss zwischen den Caches der den *threads* zugeordneten CPUs hin- und herbewegt werden → sehr zeitintensiv

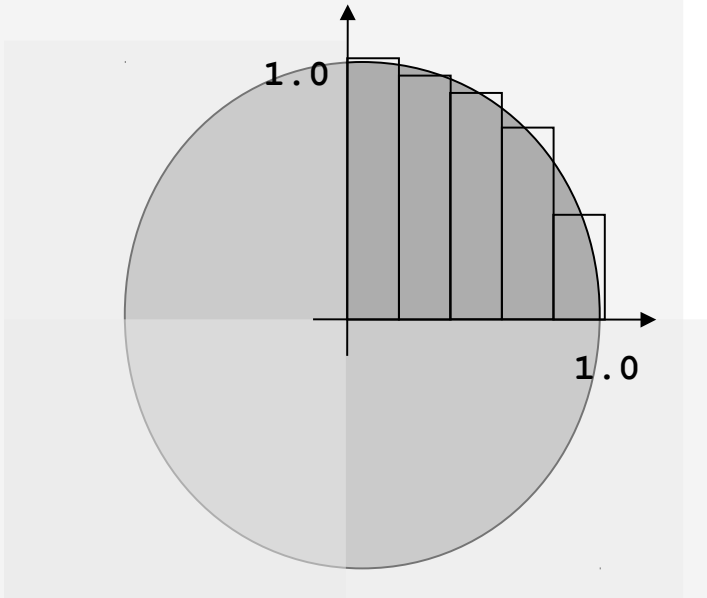
Berechnung von PI

```
program compute_pi
integer                         :: i
integer, parameter              :: n=50000000, dp = kind(1.d0)
real(kind=dp)                   :: w,x,sum,pi,d

w=1.0/n; sum=0.0

do i=1,n
   x   = (i-0.5) * w
   d   = w * SQRT(1.0 - x**2)
   sum = sum  + d
enddo
pi = 4. * sum
print *, 'computed pi = ', pi

end program compute_pi
```



Parallele Berechnung von PI

```
program compute_pi
integer                         :: i
integer, parameter              :: n=500000000, dp = kind(1.d0)
real(kind=dp)                   :: w,x,sum,pi,d

w=1.0/n; sum=0.0
!$OMP PARALLEL PRIVATE(x,d), SHARED(w,sum)
!$OMP DO REDUCTION(+: sum)
do i=1,n
   x   = (i-0.5) * w
   d   = w * SQRT(1.0 - x**2)
   sum = sum  + d
enddo
!$OMP END DO
!$OMP END PARALLEL
pi = 4. * sum
print *, 'computed pi = ', pi
end program compute_pi
```

HP XC3000

1 core:

real	2.92s
user	2.93s

2 cores:

real	1.50s
user	2.99s

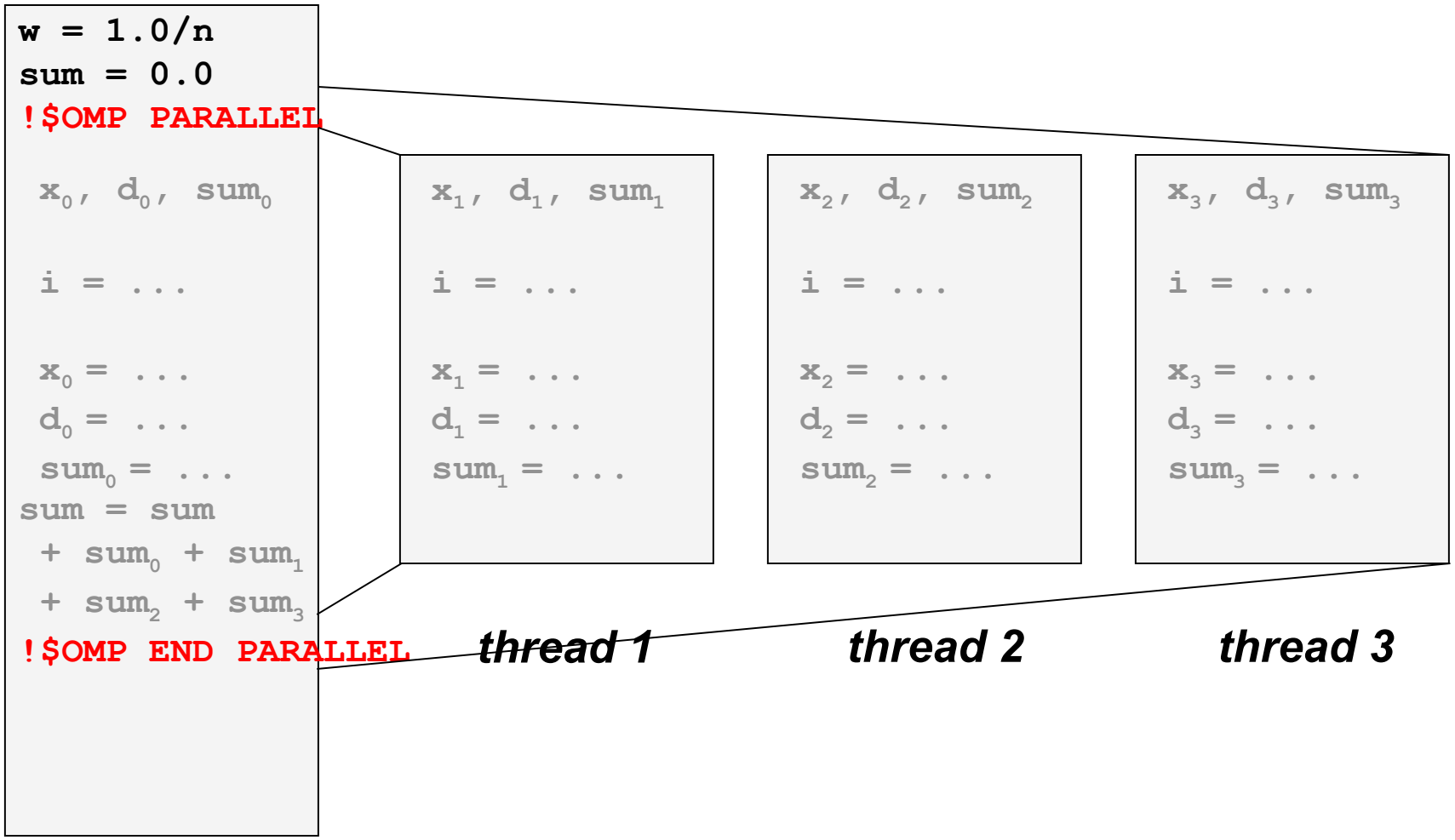
4 cores:

real	0.75s
user	2.97s

8 cores:

real	0.39s
user	3.10s

Paralleles Programm PI anschaulich



Mutual exclusion Synchronisierung – `critical` (section)

- Nur ein einziger *thread* kann zu einer bestimmten Zeit einen kritischen Abschnitt betreten.

```
 a_max = MINUS_INFINITY; a_min = PLUS_INFINITY

!$OMP PARALLEL DO
 do i=1,n

   if (a(i) > a_max) then
!$OMP CRITICAL(MAXLOCK)
     if (a(i) > a_max) then; a_max = a(i); endif
!$OMP END CRITICAL(MAXLOCK)
   endif

   if (a(i) < a_min) then
!$OMP CRITICAL(MINLOCK)
     if (a(i) < a_min) then; a_min = a(i); endif
!$OMP END CRITICAL(MINLOCK)
   endif

 enddo
```

Mutual exclusion Synchronisierung – `atomic` (*update*)

- `atomic` ist ein Spezialfall eines kritischen Abschnitts, der dazu benutzt werden kann, skalaren Variablen in einfachen Zuweisungen einen neuen Wert zuzuweisen.
  - Fortran: `!$OMP ATOMIC`
  - C: `#pragma omp atomic`

- Die Zuweisung muss folgende Syntax aufweisen:

  x = x operator expr

  x = intrinsic(x, expr)

  Operatoren

  Fortran: `+, -, *, /, .AND., .OR., .EQV., .NEQV., MAX, MIN, IAND, IOR, IEOR`

  C: `+, -, *, /, &, |, ^, &&, ||`