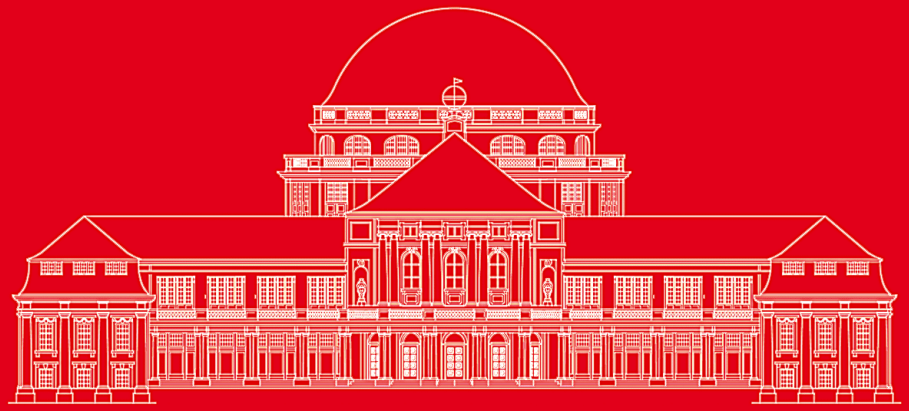
Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Praktikum: Paralleles Programmieren für Geowissenschaftler

Prof. Thomas Ludwig, Hermann Lenhart & Enno Zickler



Dr. Hermann-J. Lenhart
hermann.lenhart@informatik.uni-hamburg.de

OpenMP Einführung II:

- Parallele Konstrukte
- Clauses
- Synchronisation
- Reduction

OpenMP – Parallele Konstrukte I

Parallel Konstrukt:

!$omp parallel [clausel 1, …,clausel n] -> siehe Quick reference

Parallele Region

!$ omp end parallel

Innerhalb der vom "!$omp parallel" Konstrukt aufgespannten parallelen Region werden Konstrukte für die Verteilung auf die Threads benutzt, z.B.

!$omp do	Iteration in Schleifen
!$omp sections	unabhängige Arbeitseinheiten
!$omp workshare	Parallelisiert Array Syntax

OpenMP – Parallele Konstrukte II

Parallel Konstrukt kombiniert mit Section:

```
!$omp parallel                          =>  Beginn Parallele Region
    !$omp sections
        !$omp section
                call subroutine A        - strukturierter Block A

        !$omp section
                call subroutine B        - strukturierter Block B
    !$omp end sections                   - Ende Sections Blöcke A+B
!$ omp end parallel                     => Ende parallele Region
```

! Keine Annahme über Reihenfolge

Load-Balance Probleme können auftauchen!

OpenMP – Parallele Konstrukte III

Combined Konstrukt kombiniert mit Workshare: **(nur in FORTRAN!)**

```
!$omp parallel  workshare shared (n,a,b,c)
        a(1:n) = a(1:n) +1
        b(1:n) = b(1:n) + 2
        c(1:n) =  c(1:n) + 3
!$ omp end parallel  workshare
```

! Es wird nicht spezifiziert wie die Arbeitseinheiten auf die Threads zugeteilt werden

! User muss für Parallelität in den Daten sorgen
(es darf keine versteckten Abhängigkeiten geben)

OpenMP – Clauses I

Die OpeMP – Clauses siehe auch Quick Reference pro Direktive

SHARED erlaubt allen Threads den gleichzeitigen Zugriff auf die gelisteten Variablen

PRIVATE setzt die gelisteten Variablen private, so dass jeder Thread nur Zugang zu einer lokalen, einmaligen Kopie der Variablen hat.

OpenMP – Clause II

DEFAULT setzt z.B. mit DEFAULT(shared)

alle zugewiesenen Variablen auf shared.

Wird benutzt um schnell der Mehrzahl der Variablen

eine Attribut zuweisen zu können.

OpenMP – Clause III

SCHEDULE nur für Loops anzuwenden

Syntax: !$omp do schedule(kind[,chunk_size])

kind:

static die direkteste Zuordnung mit dem wenigsten Overhead

Iterationen werden in Portionen der Größe *chunk_size* aufgeteilt

ohne Angabe von *chunk_size* wird der Iterationsraum gleichmäßig auf die Threads aufteilt

OpenMP – Clause IV

SCHEDULE nur für Loops anzuwenden

Syntax: !$omp do schedule(kind[,chunk_size])

kind:

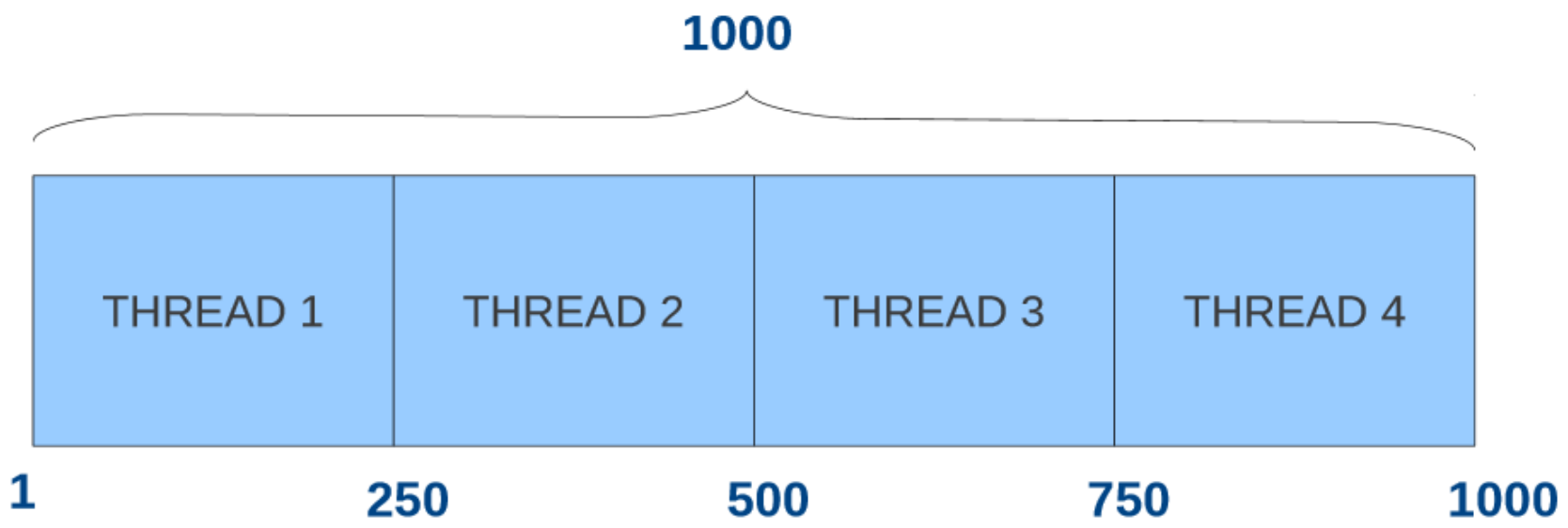
dynamic Iterationen werden nach der Verfügbarkeit der Threads zugewiesen. Jeder freie Thread bekommt einen Chunk zugewiesen bis alle Iterationen abgearbeitet sind.

guided wie dynamic, nur dass die Chunks der noch zu bearbeitenden Iterationen immer kleiner werden.

How OMP schedules iterations?

Although the OpenMP standard does not specify how a loop should be partitioned most compilers split the loop in N/p (N #iterations, p #threads) chunks by default. This is called a static schedule (with chunk size N/p)

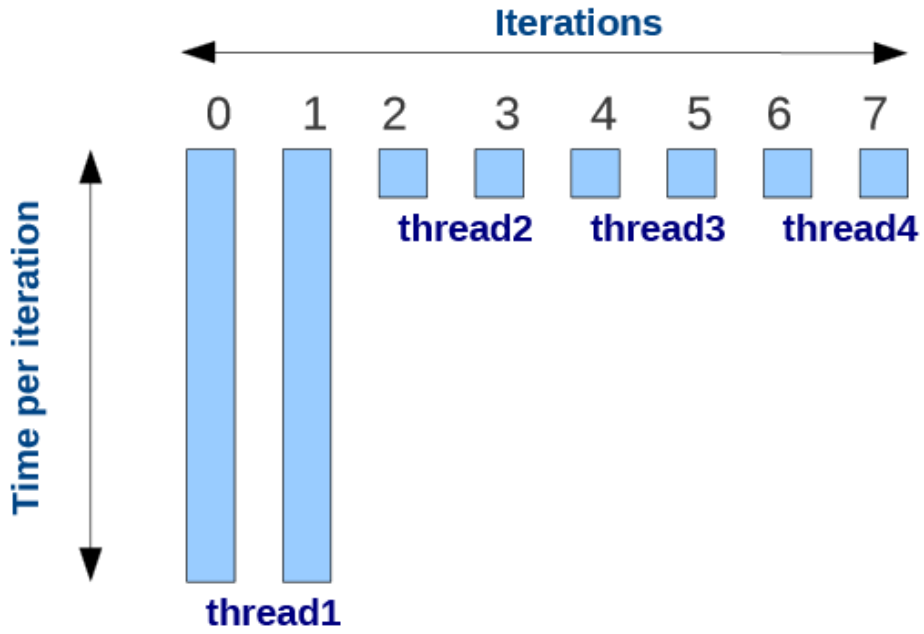
For example, suppose we have a loop with 1000 iterations and 4 omp threads. The loop is partitioned as follows:



Quelle: http://people.math.umass.edu/~johnston/PHI_WG_2014/OpenMPSlides_tamu_sc.pdf

Issues with Static schedule

With static scheduling the number of iterations is evenly distributed among all openmp threads (i.e. Every thread will be assigned similar number of iterations). This is not always the best way to partition. Why is This?



This is called load imbalance. In this case threads 2,3, and 4 will be waiting very long for thread 1 to finish

How can this happen?

Quelle: http://people.math.umass.edu/~johnston/PHI_WG_2014/OpenMPSlides_tamu_sc.pdf

Dynamic Schedule

With a dynamic schedule new chunks are assigned to threads when they come available. OpenMP provides two dynamic schedules:

- $!OMP DO SCHEDULE(DYNAMIC,n) // n is chunk size
  Loop iterations are divided into pieces of size chunk. When a thread finishes one chunk, it is dynamically assigned another.
- $!OMP DO SCHEDULE(GUIDED,n) // n is chunk size
  Similar to DYNAMIC but chunk size is relative to number of iterations left.

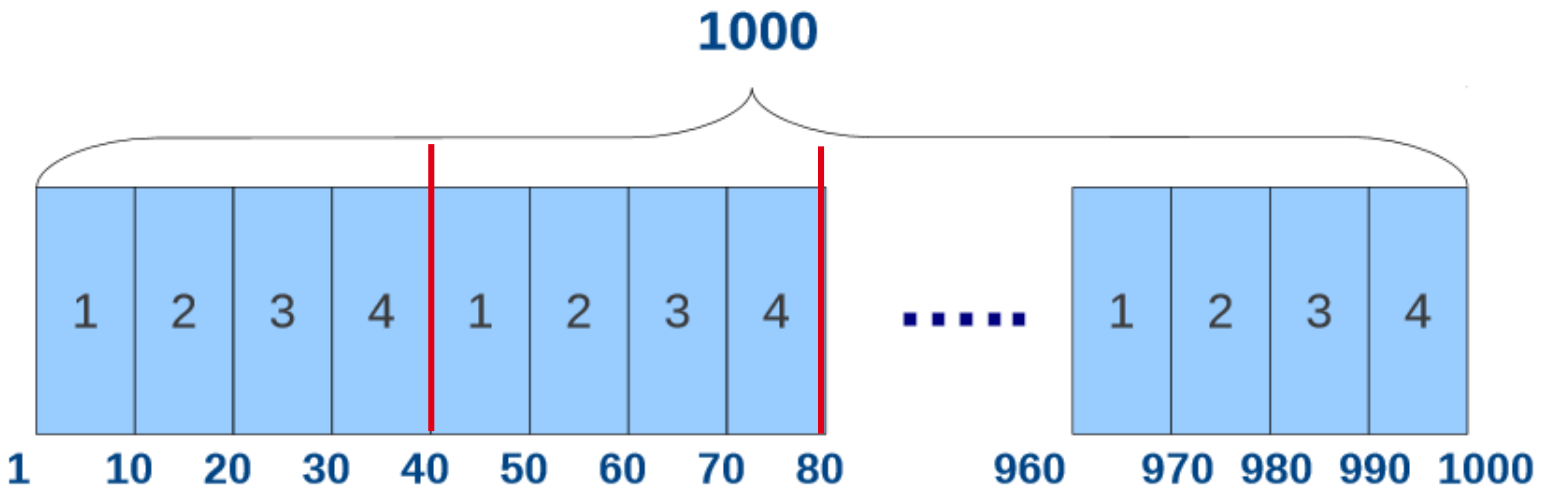
Quelle: http://people.math.umass.edu/~johnston/PHI_WG_2014/OpenMPSlides_tamu_sc.pdf

Static Schedule

To explicitly tell the compiler to use a static schedule (or a different schedule as we will see later) OpenMP provides the SCHEDULE clause

$!OMP DO SCHEDULE (STATIC,n) // (n is chunk size)

For example, suppose we set the chunk size n=10



Quelle: http://people.math.umass.edu/~johnston/PHI_WG_2014/OpenMPSlides_tamu_sc.pdf

Dynamic Schedule

With a dynamic schedule new chunks are assigned to threads when they come available. OpenMP provides two dynamic schedules:

- $!OMP DO SCHEDULE(DYNAMIC,n) // n is chunk size
  Loop iterations are divided into pieces of size chunk. When a thread finishes one chunk, it is dynamically assigned another.
- $!OMP DO SCHEDULE(GUIDED,n) // n is chunk size
  Similar to DYNAMIC but chunk size is relative to number of iterations left.

Keep in mind: although Dynamic scheduling might be the prefered choice to prevent load inbalance in some situations, there is a significant overhead involved compared to static scheduling.

Quelle: http://people.math.umass.edu/~johnston/PHI_WG_2014/OpenMPSlides_tamu_sc.pdf

OpenMP – Clause - Beispiel Programm

```
!$omp parallel do default(none) &
!$omp shared(dataOld,data) private(i,j)
      do i=2,ubound(data,1)-1
         do j=2,ubound(data,2)-1
            data(i,j) = Berechnung Star
         end do
       end do
!$omp end parallel do
```

OpenMP – Clause – Beispiel - Makefile

Im makefile: für 10 Threads

Static mit Chunk-Size 4:

export omp_schedule=static,4 OMP_NUM_THREADS=10

Dynamic mit Chunk-Size 4

export omp_schedule=dynamic,4 OMP_NUM_THREADS=10

Guided:

export omp_schedule=guided OMP_NUM_THREADS=10

Im Programm:

!$omp parallel do default(none) &

!$omp shared(dataOld,data) private(i,j)

……

!$omp end parallel do

OpenMP – Reduction I

Die **REDUCTION CLAUSE** wird von OpenMP bereitgestellt um wiederkehrende Berechnungen, z.B. Summationen, einfach durchzuführen.

Syntax: reduction ({operator I intrinsic_procedure_name} :list)

```
!$omp parallel do reduction(+:sum)
    do i = 1, n
    sum = sum + a(i)
    enddo
```

Sorgt aber intern auch für spezifische Zugriffsrechte.

Dazu folgendes Beispiel.

OpenMP – Reduction II

Problemstellung:

```
!$omp  parallel do
  do i = 1,1000
    a = a  + i
  end do
!$omp end parallel do
```

a ist „shared“ da alle Threads darauf zugreifen müssen

aber nur ein Thread soll zu einem gegebenen Zeitpunkt schreiben bzw. a updaten können

sonst würde ein undefinierbares Ergebnis erfolgen

Quelle: Miguel Hermanns

OpenMP – Reduction III

Problemstellung:

```
!$omp  parallel do        reduction(+:a)

  do j = 1,1000

                          nur ein Thread pro Zeitpunkt darf  a verändern
      a = a  + i

   end do

!$omp end parallel do
```

Quelle: Miguel Hermanns

OpenMP – Reduction II

Für die **REDUCTION CLAUSE** stehen folgende Operatoren und Initialwerte bereit:

Operator	Initialwert
+ / -	0
*	1
.and. / .eqv.	.true.
.or. / .neqv.	.false.

Intrinsische Funktion	Initialwert
max	kleinste Zahl in der reduction Elemente Liste
min	größte Zahl in der reduction Elemente Liste

OpenMP – Synchronisation

BARRIER sind Synchronisationspunkte bei denen die einzelnen Threads aufeinander warten. Keinem Thread wird erlaubt im Programm fortzufahren, bis alle anderen Threads ebenfalls diesen Programmpunkt erreicht haben.

Syntax: !$omp barrier

Danke das wars!