

# Programming OpenMP

## *Worksharing*

**Christian Terboven**



## For Worksharing

- If only the *parallel* construct is used, each thread executes the Structured Block.
- Program Speedup: *Worksharing*
- OpenMP's most common Worksharing construct: *for*

### C/C++

```
int i;  
#pragma omp for  
for (i = 0; i < 100; i++)  
{  
    a[i] = b[i] + c[i];  
}
```

### Fortran

```
INTEGER :: i  
!$omp do  
DO i = 0, 99  
    a[i] = b[i] + c[i]  
END DO
```

- Distribution of loop iterations over all threads in a Team.
  - Scheduling of the distribution can be influenced.
- Loops often account for most of a program's runtime!

# Worksharing illustrated

Pseudo-Code  
Here: 4 Threads

Serial

```
do i = 0, 99
  a(i) = b(i) + c(i)
end do
```

Thread 1

```
do i = 0, 24
  a(i) = b(i) + c(i)
end do
```

Thread 2

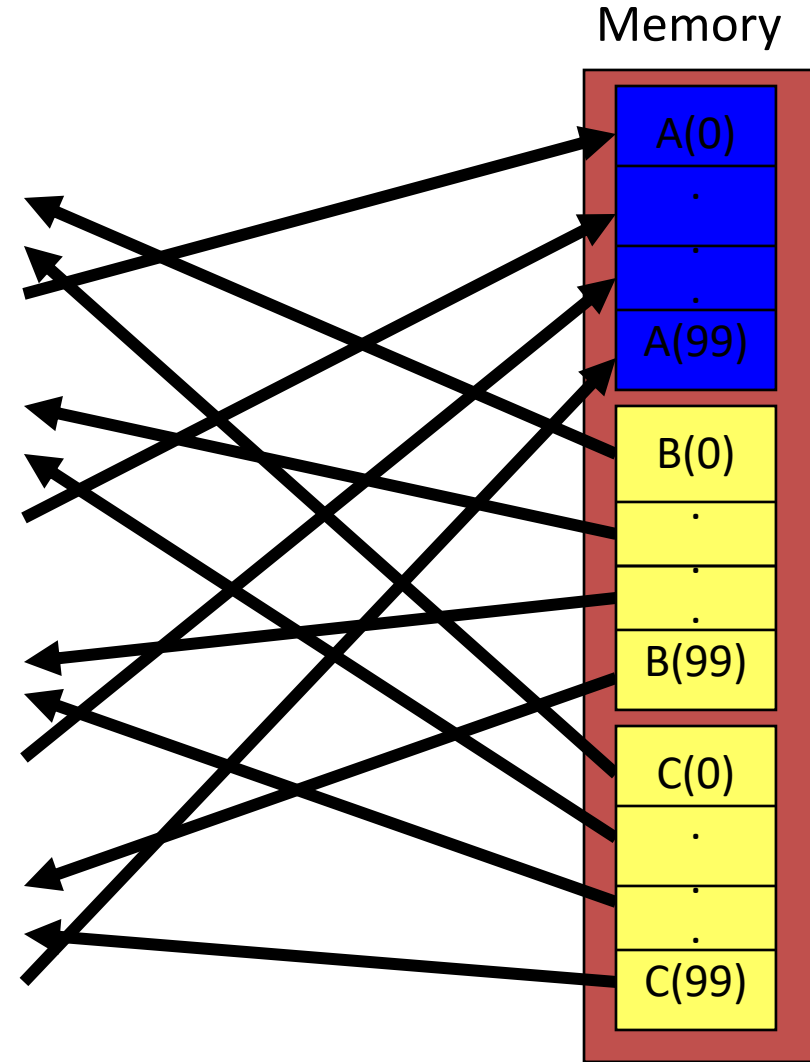
```
do i = 25, 49
  a(i) = b(i) + c(i)
end do
```

Thread 3

```
do i = 50, 74
  a(i) = b(i) + c(i)
end do
```

Thread 4

```
do i = 75, 99
  a(i) = b(i) + c(i)
end do
```



## The Barrier Construct

- OpenMP `barrier` (implicit or explicit)
  - Threads wait until all threads of the current *Team* have reached the barrier

```
C/C++  
#pragma omp barrier
```

- All worksharing constructs contain an implicit barrier at the end

## The Single Construct

C/C++

```
#pragma omp single [clause]  
... structured block ...
```

Fortran

```
!$omp single [clause]  
... structured block ...  
!$omp end single
```

- The `single` construct specifies that the enclosed structured block is executed by only one thread of the team.
  - It is up to the runtime which thread that is.
- Useful for:
  - I/O
  - Memory allocation and deallocation, etc. (in general: setup work)
  - Implementation of the single-creator parallel-executor pattern as we will see later...

## The Master Construct

C/C++

```
#pragma omp master[clause]  
... structured block ...
```

Fortran

```
!$omp master[clause]  
... structured block ...  
!$omp end master
```

- The `master` construct specifies that the enclosed structured block is executed only by the master thread of a team.
- Note: The master construct is no worksharing construct and does not contain an implicit barrier at the end.

# Vector Addition

## Influencing the For Loop Scheduling / 1

- *for*-construct: OpenMP allows to influence how the iterations are scheduled among the threads of the team, via the *schedule* clause:
  - `schedule(static [, chunk])`: Iteration space divided into blocks of chunk size, blocks are assigned to threads in a round-robin fashion. If chunk is not specified: #threads blocks.
  - `schedule(dynamic [, chunk])`: Iteration space divided into blocks of chunk (not specified: 1) size, blocks are scheduled to threads in the order in which threads finish previous blocks.
  - `schedule(guided [, chunk])`: Similar to dynamic, but block size starts with implementation-defined value, then is decreased exponentially down to chunk.
- Default is `schedule(static)`.



# Influencing the For Loop Scheduling / 2

## ■ Static Schedule

→ `schedule(static [, chunk])`

→ Decomposition  
depending on chunksize

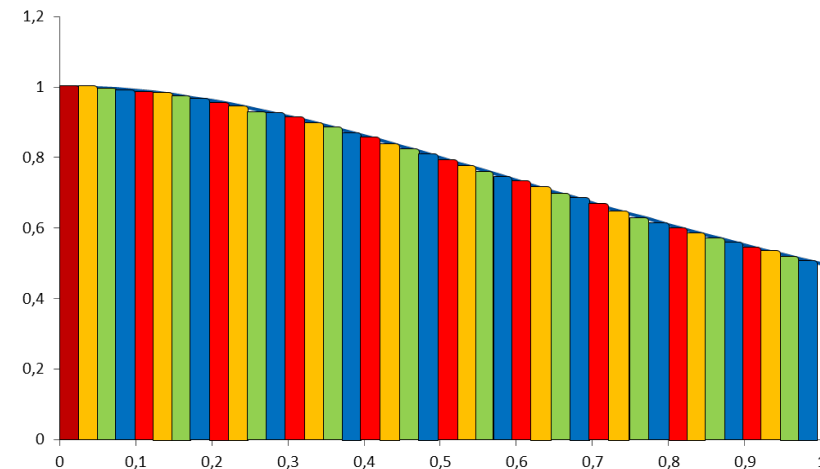
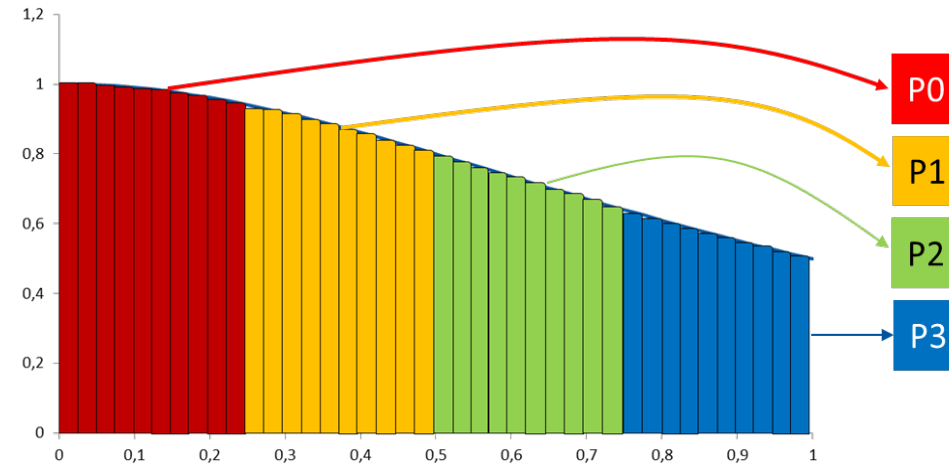
→ Equal parts of size 'chunksize'  
distributed in round-robin  
fashion

## ■ Pros?

→ No/low runtime overhead

## ■ Cons?

→



## Influencing the For Loop Scheduling / 3

- Dynamic schedule
  - `schedule(dynamic [, chunk])`
  - Iteration space divided into blocks of chunk size
  - Threads request a new block after finishing the previous one
  - Default chunk size is 1
- Pros ?
  - Workload distribution
- Cons?
  - Runtime Overhead
  - Chunk size essential for performance
  - No NUMA optimizations possible

## Synchronization Overview

- Can all loops be parallelized with `for`-constructs? No!
  - Simple test: If the results differ when the code is executed backwards, the loop iterations are not independent. BUT: This test alone is not sufficient:

```
C/C++
int i, int s = 0;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
    s = s + a[i];
}
```

- *Data Race*: If between two synchronization points at least one thread writes to a memory location from which at least one other thread reads, the result is not deterministic (race condition).

## Synchronization: Critical Region

- A *Critical Region* is executed by all threads, but by only one thread simultaneously (*Mutual Exclusion*).

C/C++

```
#pragma omp critical (name)
{
    ... structured block ...
}
```

- Do you think this solution scales well?

C/C++

```
int i, s = 0;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
    #pragma omp critical
    { s = s + a[i]; }
}
```