

# Programming OpenMP

## *Scoping*

Christian Terboven



## Scoping Rules

- Managing the Data Environment is the challenge of OpenMP.
- *Scoping* in OpenMP: Dividing variables in *shared* and *private*:
  - *private*-list and *shared*-list on Parallel Region
  - *private*-list and *shared*-list on Worksharing constructs
  - General default is *shared* for Parallel Region, *firstprivate* for Tasks.
  - Loop control variables on *for*-constructs are *private*
  - Non-static variables local to Parallel Regions are *private*
  - *private*: A new uninitialized instance is created for the task or each thread executing the construct
    - *firstprivate*: Initialization with the value before encountering the construct
    - *lastprivate*: Value of last loop iteration is written back to Master
  - Static variables are *shared*

Tasks are  
introduced later

## Privatization of Global/Static Variables

- Global / static variables can be privatized with the *threadprivate* directive
  - One instance is created for each thread
    - Before the first parallel region is encountered
    - Instance exists until the program ends
    - Does not work (well) with nested Parallel Region
  - Based on thread-local storage (TLS)
    - TlsAlloc (Win32-Threads), pthread\_key\_create (Posix-Threads), keyword `__thread` (GNU extension)

C/C++

```
static int i;  
#pragma omp threadprivate(i)
```

Fortran

```
SAVE INTEGER :: i  
!$omp threadprivate(i)
```

## Privatization of Global/Static Variables

- Global / static variables can be privatized with the *threadprivate* directive
  - One instance is created for each thread
    - Before the first parallel region is encountered
    - Instance exists until the program ends
    - Does not work (well) with nested Parallel Region
  - Based on thread-local storage (TLS)
    - TlsAlloc (Win32-Threads), pthread\_key\_create (Posix-Threads), keyword `__thread` (GNU extension)

<p>C/C++</p> <pre>static int i; #pragma omp threadprivate(i)</pre>	<p>Fortran</p> <pre>SAVE INTEGER :: i !\$omp threadprivate(i)</pre>
--	---

**Really: try to avoid the use of threadprivate and static variables!**

# Back to our example

C/C++

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

# It's your turn: Make It Scale!

```
#pragma omp parallel
{

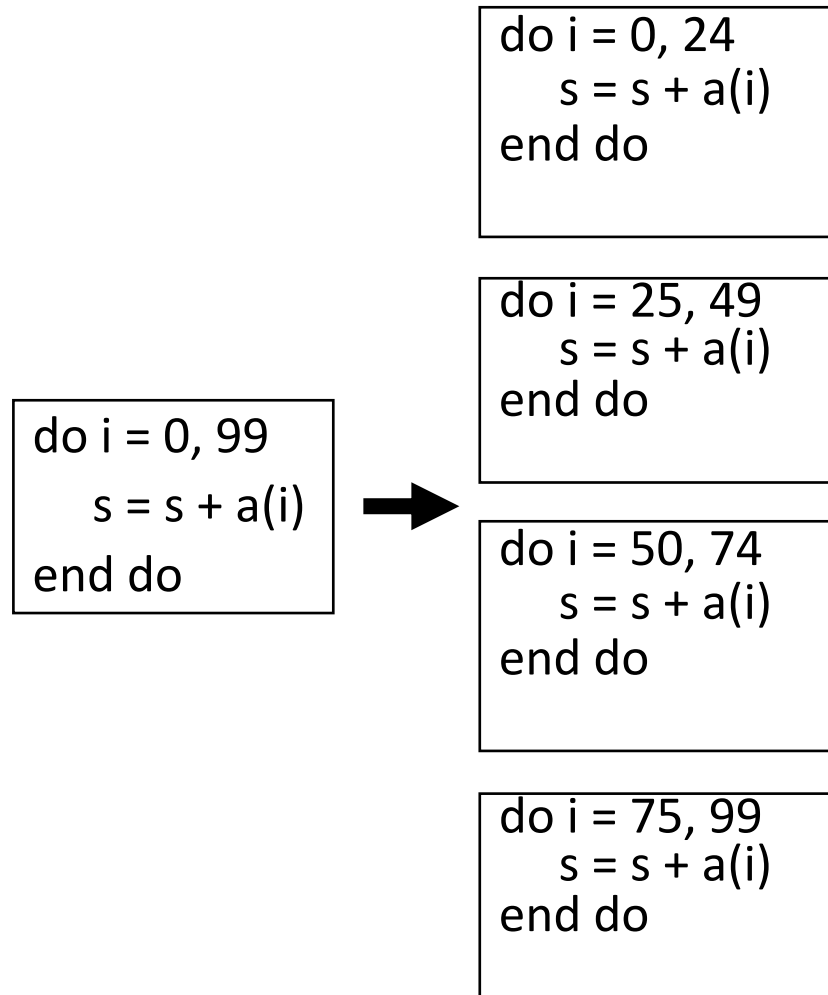
#pragma omp for
  for (i = 0; i < 99; i++)
  {

      s = s + a[i];

  }

} // end parallel
```

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



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

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


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

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

(done)

```
#pragma omp parallel
{
    double ps = 0.0;    // private variable
    #pragma omp for
    for (i = 0; i < 99; i++)
    {
        ps = ps + a[i];
    }
    #pragma omp critical
    {
        s += ps;
    }
} // end parallel
```

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



```
do i = 0, 24
    s1 = s1 + a(i)
end do
s = s + s1
```

```
do i = 25, 49
    s2 = s2 + a(i)
end do
s = s + s2
```

```
do i = 50, 74
    s3 = s3 + a(i)
end do
s = s + s3
```

```
do i = 75, 99
    s4 = s4 + a(i)
end do
s = s + s4
```

## The Reduction Clause

- In a *reduction*-operation the operator is applied to all variables in the list. The variables have to be *shared*.
  - `reduction(operator:list)`
  - The result is provided in the associated reduction variable

C/C++

```
int i, s = 0;
#pragma omp parallel for reduction(+:s)
for(i = 0; i < 99; i++)
{
    s = s + a[i];
}
```

- Possible reduction operators with initialization value:  
`+` (0), `*` (1), `-` (0), `&` (`~0`), `|` (0), `&&` (1), `||` (0), `^` (0), `min` (largest number), `max` (least number)
- Remark: OpenMP also supports user-defined reductions (not covered here)



# PI

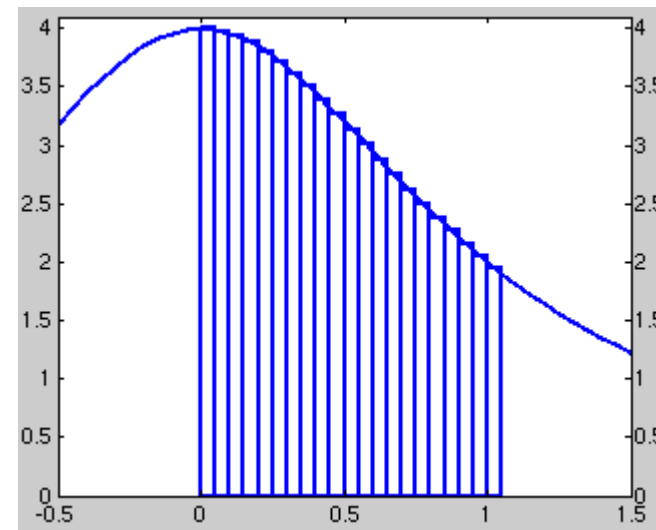
## Example: Pi (1/2)

```
double f(double x)
{
    return (4.0 / (1.0 + x*x));
}
```

```
double CalcPi (int n)
{
    const double fH = 1.0 / (double) n;
    double fSum = 0.0;
    double fX;
    int i;
```

```
#pragma omp parallel for
for (i = 0; i < n; i++)
{
    fX = fH * ((double)i + 0.5);
    fSum += f(fX);
}
return fH * fSum;
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2}$$



## Example: Pi (2/2)

```
double f(double x)
{
    return (4.0 / (1.0 + x*x));
}
```

```
double CalcPi (int n)
{
    const double fH = 1.0 / (double) n;
    double fSum = 0.0;
    double fX;
    int i;
```

```
#pragma omp parallel for private(fX,i) reduction(+:fSum)
    for (i = 0; i < n; i++)
    {
        fX = fH * ((double)i + 0.5);
        fSum += f(fX);
    }
    return fH * fSum;
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2}$$

