Introduction to OpenMP

part I of III

Christian Terboven <terboven@rz.rwth-aachen.de>

30.07.2013 / Aachen, Germany

Stand: 22.07.2013

Version 2.3

History

- De-facto standard for Shared-Memory Parallelization.

- 1997: OpenMP 1.0 for FORTRAN
- 1998: OpenMP 1.0 for C and C++
- 1999: OpenMP 1.1 for FORTRAN (errata)

- 2000: OpenMP 2.0 for FORTRAN
- 2002: OpenMP 2.0 for C and C++
- 2005: OpenMP 2.5 now includes both programming languages.

- 05/2008: OpenMP 3.0 release
- 07/2011: OpenMP 3.1 release

- 07/2013: OpenMP 4.0 released

http://www.OpenMP.org

RWTH Aachen University is a member of the OpenMP Architecture Review Board (ARB) since 2006.

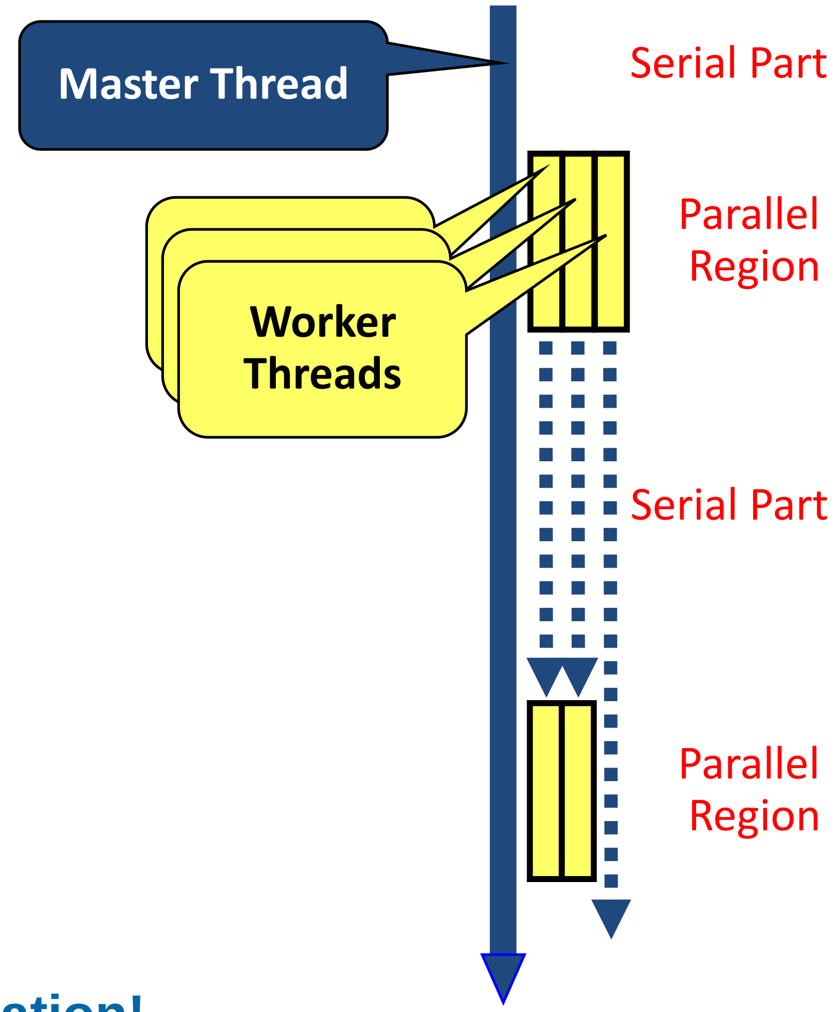
Agenda

- **Basic Concept: *Parallel Region***
- **The *For* Construct**
- **The *Single* Construct**
- **The *Task* Construct**
- ***Scoping*: Managing the Data Environment**
- **Synchronization: the *Critical* and *Reduction* Constructs**
- **More Components of OpenMP**

Parallel Region

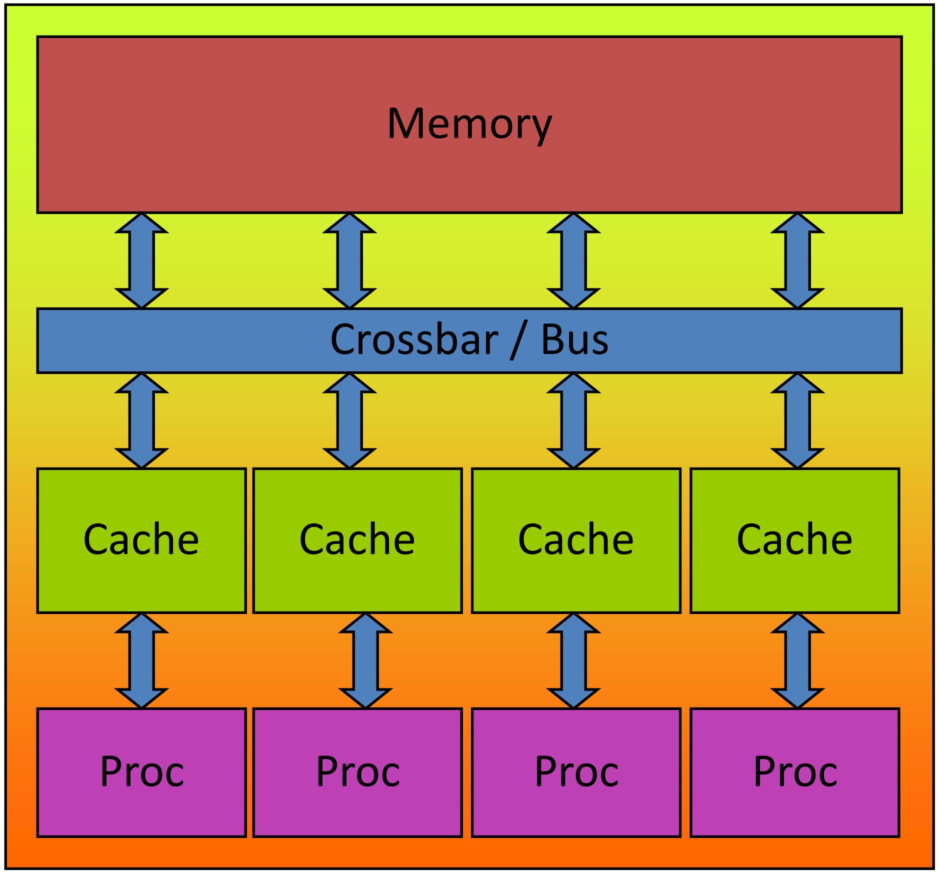
OpenMP Execution Model

- OpenMP programs start with just one thread: The *Master*.
- *Worker* threads are spawned at *Parallel Regions*, together with the Master they form the *Team* of threads.
- In between Parallel Regions the Worker threads are put to sleep. The OpenMP *Runtime* takes care of all thread management work.
- Concept: *Fork-Join*.
- Allows for an incremental parallelization!



OpenMP's machine model

- OpenMP: Shared-Memory Parallel Programming Model.



All processors/cores access a shared main memory.

Real architectures are more complex, as we will see later / as we have seen.

Parallelization in OpenMP employs multiple threads.

Parallel Region and Structured Blocks

- **The parallelism has to be expressed explicitly.**

C/C++

```
#pragma omp parallel
{
    ...
    structured block
    ...
}
```

Fortran

```
!$omp parallel
    ...
    structured block
    ...
$!omp end parallel
```

- ***Structured Block***
  - Exactly one entry point at the top
  - Exactly one exit point at the bottom
  - Branching in or out is not allowed
  - Terminating the program is allowed (abort / exit)

- **Specification of number of threads:**
  - Environment variable:

    `OMP_NUM_THREADS=…`
  - Or: Via `num_threads` clause:

    add `num_threads(num)` to the parallel construct

Hello OpenMP World

Hello orphaned World

Starting OpenMP Programs on Linux

- **From within a shell, global adjustment of the number of threads:**

```
export OMP_NUM_THREADS=4
./program
```

- **From within a shell, one-time adjustment of the number of threads:**

```
OMP_NUM_THREADS=4  ./program
```

- **Intel Compiler on Linux: asking for more information:**

```
export KMP_AFFINITY=verbose
export OMP_NUM_THREADS=4
./program
```

For Construct

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;
double a[N], b[N], c[N];
#pragma omp parallel for
for (i = 0; i < N; i++)
{
   a[i] = b[i] + c[i];
}
```

Fortran

```
INTEGER :: i
INTEGER, DIMENSION(N) :: a,b,c
!$omp parallel do
DO i = 1, N
   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

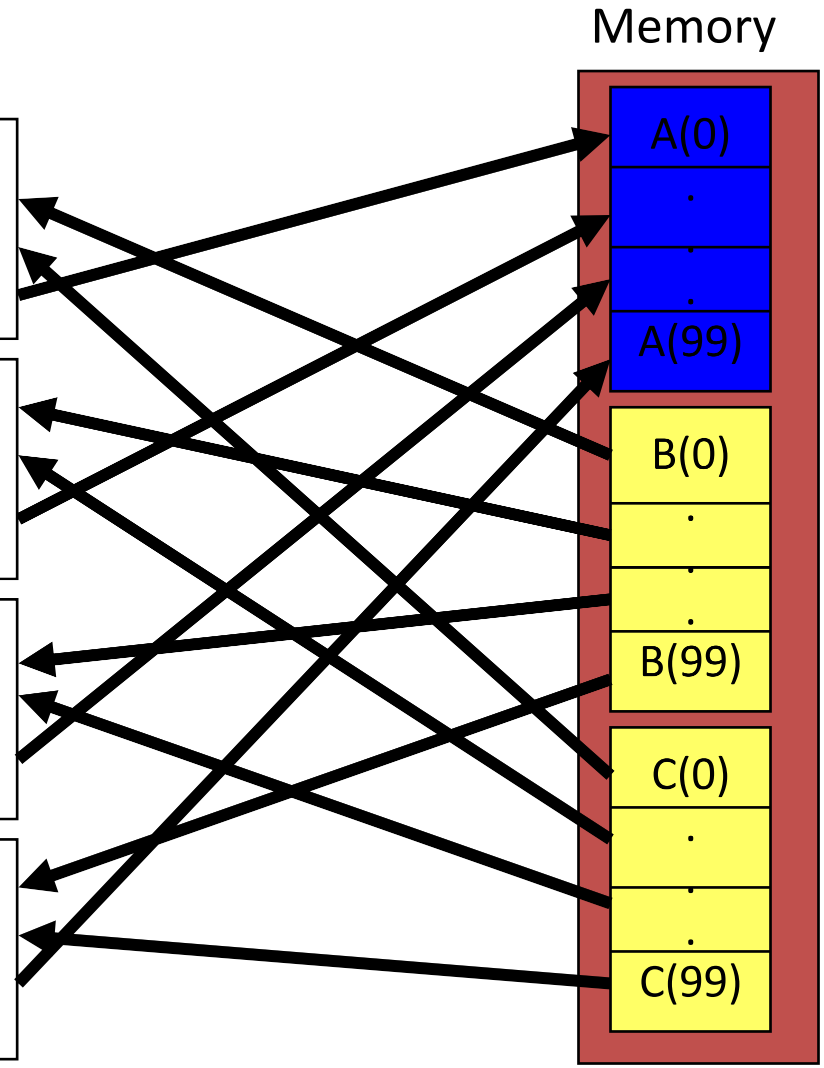
```
do i = 25, 49
    a(i) = b(i) + c(i)
end do
```

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
```



Vector Addition

The Single Construct

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 on 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 now…

Task Construct

Sudoko for Lazy Computer Scientists

- **Lets solve Sudoku puzzles with brute multi-core force**

	6						8	11			15	14			16
15	11				16	14				12			6		
13		9	12					3	16	14		15	11	10	
2		16		11		15	10	1							
	15	11	10			16	2	13	8	9	12				
12	13			4	1	5	6	2	3					11	10
5		6	1	12		9		15	11	10	7	16			3
	2				10		11	6		5			13		9
10	7	15	11	16				12	13						6
9						1			2		16	10			11
1		4	6	9	13			7		11		3	16		
16	14			7		10	15	4	6	1				13	8
11	10		15				16	9	12	13			1	5	4
		12		1	4	6		16				11	10		
		5		8	12	13		10			11	2			14
3	16			10			7			6				12	

(1) Find an empty field

(2) Insert a number

(3) Check Sudoku

(4 a) If invalid:
Delete number,
Insert next number

(4 b) If valid:
Go to next field

The Task Construct

C/C++

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

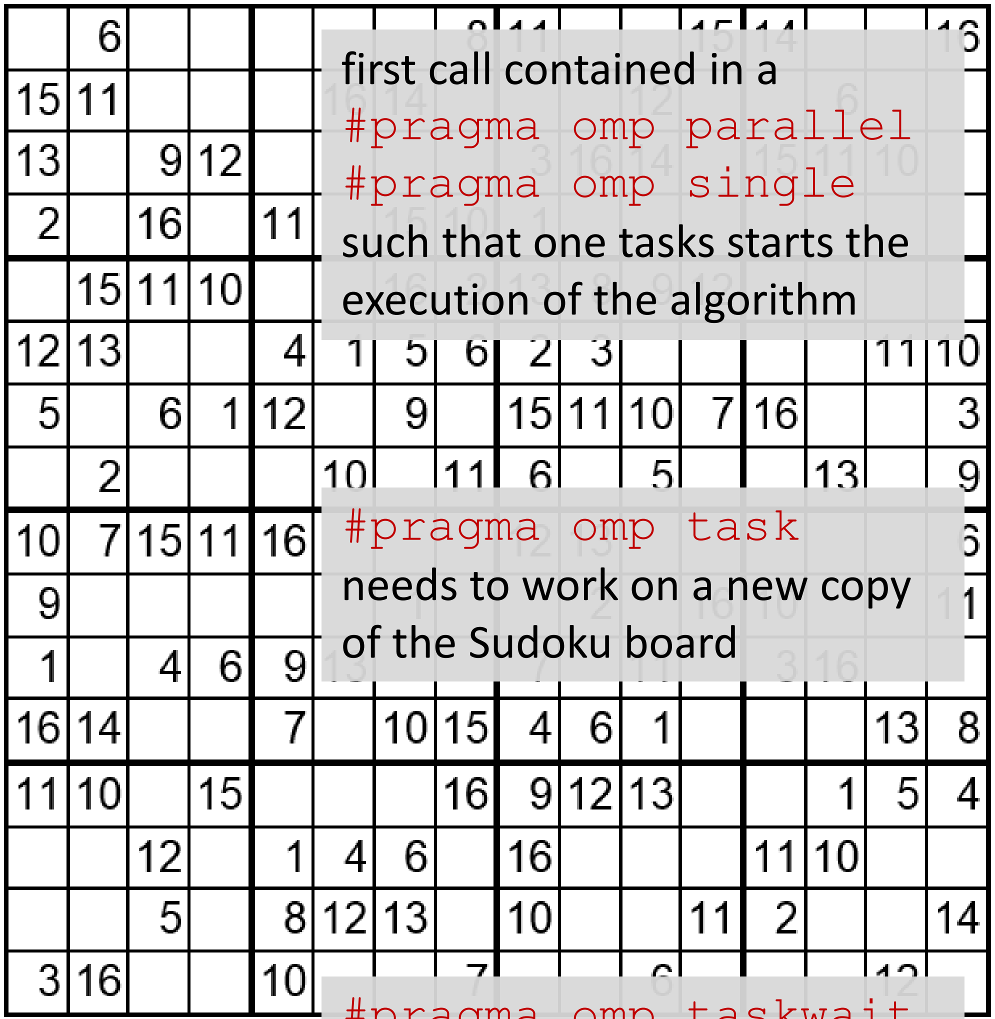
Fortran

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

- **Each encountering thread/task creates a new Task**
  - Code and data is being packaged up
  - Tasks can be nested
    - Into another Task directive
    - Into a Worksharing construct
- **Data scoping clauses:**
    - `shared`(*list*)
    - `private`(*list*) `firstprivate`(*list*)
    - `default`(*shared* | *none*)

Parallel Brute-force Sudoku (1/3)

- This parallel algorithm finds all valid solutions



(1) Search an empty field

(2) Insert a number

(3) Check Sudoku

(4 a) If invalid: Delete number, Insert next number

(4 b) If valid: Go to next field

Parallel Brute-force Sudoku (2/3)

- **OpenMP parallel region creates a team of threads**

```
#pragma omp parallel
{
#pragma omp single
   solve_parallel(0, 0, sudoku2,false);
} // end omp parallel
```

  - Single construct: One thread enters the execution of `solve_parallel`
  - the other threads wait at the end of the `single` …
    - … and are ready to pick up tasks „from the work queue“

- **Syntactic sugar (either you like it or you do not)**

```
#pragma omp parallel sections
{
   solve_parallel(0, 0, sudoku2,false);
} // end omp parallel
```

Parallel Brute-force Sudoku (3/3)

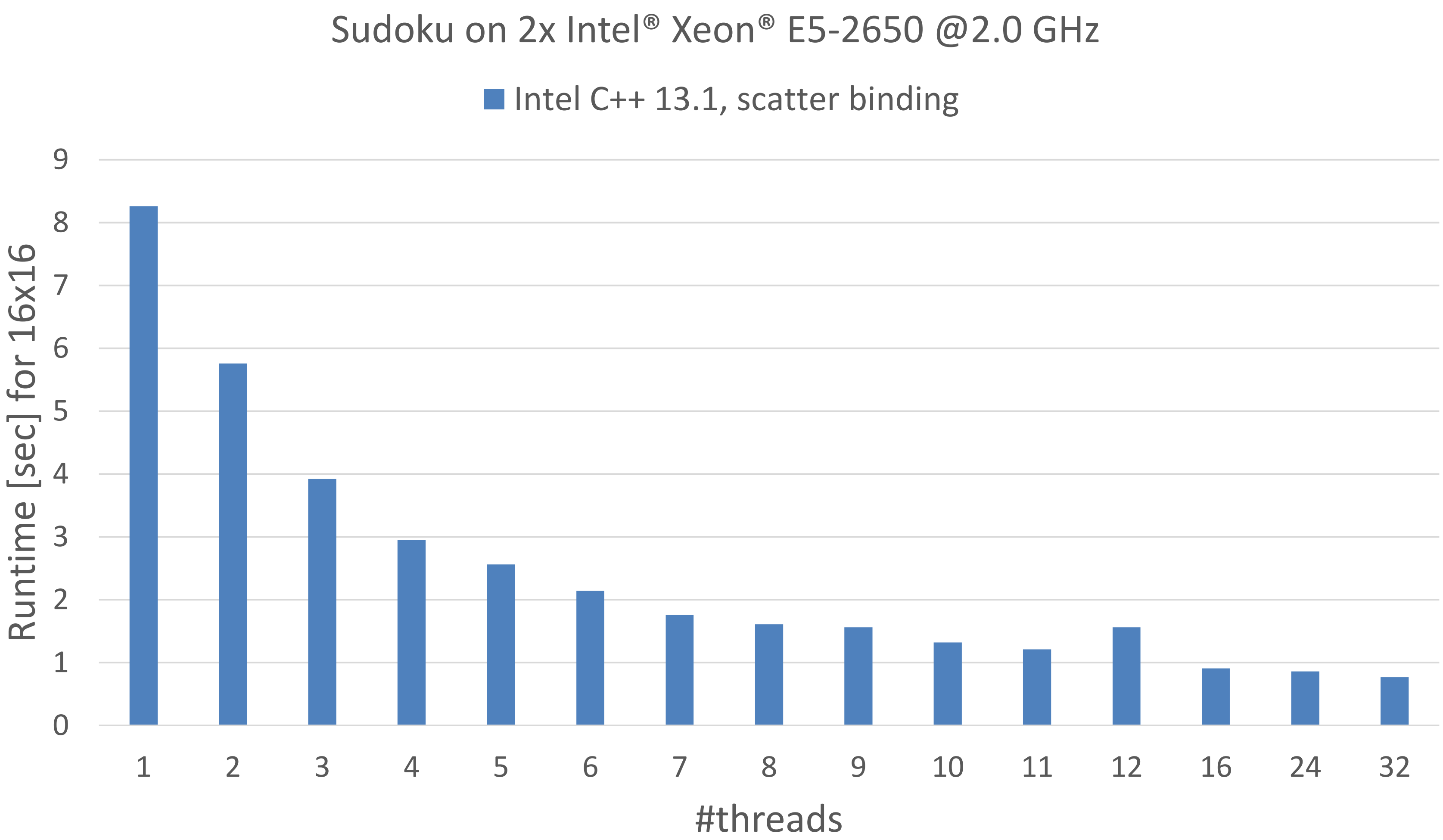
- **The actual implementation**

```
for (int i = 1; i <= sudoku->getFieldSize(); i++) {
   if (!sudoku->check(x, y, i)) {
#pragma omp task firstprivate(i,x,y,sudoku)
{
      // create from copy constructor
      CSudokuBoard new_sudoku(*sudoku);
      new_sudoku.set(y, x, i);
      if (solve_parallel(x+1, y, &new_sudoku)) {
         new_sudoku.printBoard();
      }
} // end omp task
   }
}
#pragma omp taskwait
```

`#pragma omp task`
needs to work on a new copy of the Sudoku board

`#pragma omp taskwait`
wait for all child tasks

Performance Evaluation



Scoping

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 each thread
    - *firstprivate*: Initialization with Master‘s value
    - *lastprivate*: Value of last loop iteration is written back to Master
  - Static variables are *shared*

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)

C/C++

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

Fortran

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

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

Tasks in OpenMP: Data Scoping

- **Some rules from *Parallel Regions* apply:**
  - Static and Global variables are shared
  - Automatic Storage (local) variables are private

- **If `shared` scoping is not derived by default:**
  - Orphaned Task variables are `firstprivate` by default!
  - Non-Orphaned Task variables inherit the `shared` attribute!

  → Variables are `firstprivate` unless `shared` in the enclosing context

- **So far no verification tool is available to check Tasking programs for correctness!**

Synchronization

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;
double s, a[N];
#pragma omp parallel for
for (i = 0; i < N; 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;
double s, a[N];
#pragma omp parallel for
for (i = 0; i < N; 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 < N; 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
```

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

  - Possible reduction operators with initialization value:

```
+ (0), * (1), - (0),
& (~0), | (0), && (1), || (0),
^ (0), min (least number), max (largest number)
```

The Barrier and Taskwait Constructs

- **OpenMP `barrier` (implicit or explicit)**
  - All tasks created by any thread of the current *Team* are guaranteed to be completed at barrier exit

C/C++

```
#pragma omp barrier
```

- **Task barrier: `taskwait`**
  - Encountering Task suspends until child tasks are complete
    - Only direct childs, not descendants!

C/C++

```
#pragma omp taskwait
```

More Components of OpenMP

Components of OpenMP

Directives	Runtime Functions	Env. Variables
Array Section Expressions	Number of Threads	Number of Threads
Parallel Region	Thread ID	Scheduling Type
Worksharing Constructs	Dynamic Thread Adjustment	Dynamic Thread Adjustment
SIMD Constructs	Cancellation Status	Thread Affinity Places
Device Constructs	Nested Parallelism	Nested Parallelism
Tasking	Schedlue	Stacksize
Synchronization Constructs	Active Levels	Idle Thread
Cancellation Constructs	Device Selection	Active Levels
Declaration Constructs	Thread Limit	Thread Limit
Memory Flush	Nesting Level	Cancellation
Data-sharing attributes	Ancestor Thread	Default Printout
	Team Size	
	Wallclock Timer	
	Locking	

Red color indicates new addition to OpenMP 4.0

The Worksharing Constructs

```
#pragma omp for
{
    ....
}

!$OMP DO
    ....
!$OMP END DO
```

```
#pragma omp sections
{
        ....
}

!$OMP SECTIONS
        ....
!$OMP END SECTIONS
```

```
#pragma omp single
{
        ....
}

!$OMP SINGLE
        ....
!$OMP END SINGLE
```

- **The work is distributed over the threads**
- **Must be enclosed in a parallel region**
- **Must be encountered by all threads in theam, or non at all**
- **No implied barrier on entry; implied barrier on exit**
- **A worksharing construct does not launch any new threads**

Some Additional Directives

```
#pragma omp master
{<code-block>}
```

```
!$omp master
    <code-block>
!$omp end master
```

There is no implied barrier on entry or exit !

```
#pragma omp critical [(name)]
{<code-block>}
```

```
!$omp critical [(name)]
    <code-block>
!$omp end critical [(name)]
```

Very useful to avoid data races

```
#pragma omp atomic
```

```
!$omp atomic
```

Also supports fine tuning controls

Appendix A: make/gmake

make / gmake

- **make: "smart" utility to manage compilation of programs and much more**
  - automatically detects which parts need to be rebuild
  - general rules for compilation of many files
  - dependencies between files can be handled

- **Usage:**

```
make <target>  or  gmake <target>
```

- **Rules:**

```
target ... : prerequisites ...
	< tab > command
	< tab > ...
```

  - target: output file (or only a name)
  - prerequisites: input files (e.g. source code files)
  - command: action to be performed