Introduction to OpenMP

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

20.11.2012 / Aachen, Germany

Stand: 19.11.2012

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.

- 08/2007: OpenMP 3.0 draft
- 05/2008: OpenMP 3.0 release
- 07/2011: OpenMP 3.1 release

http://www.OpenMP.org

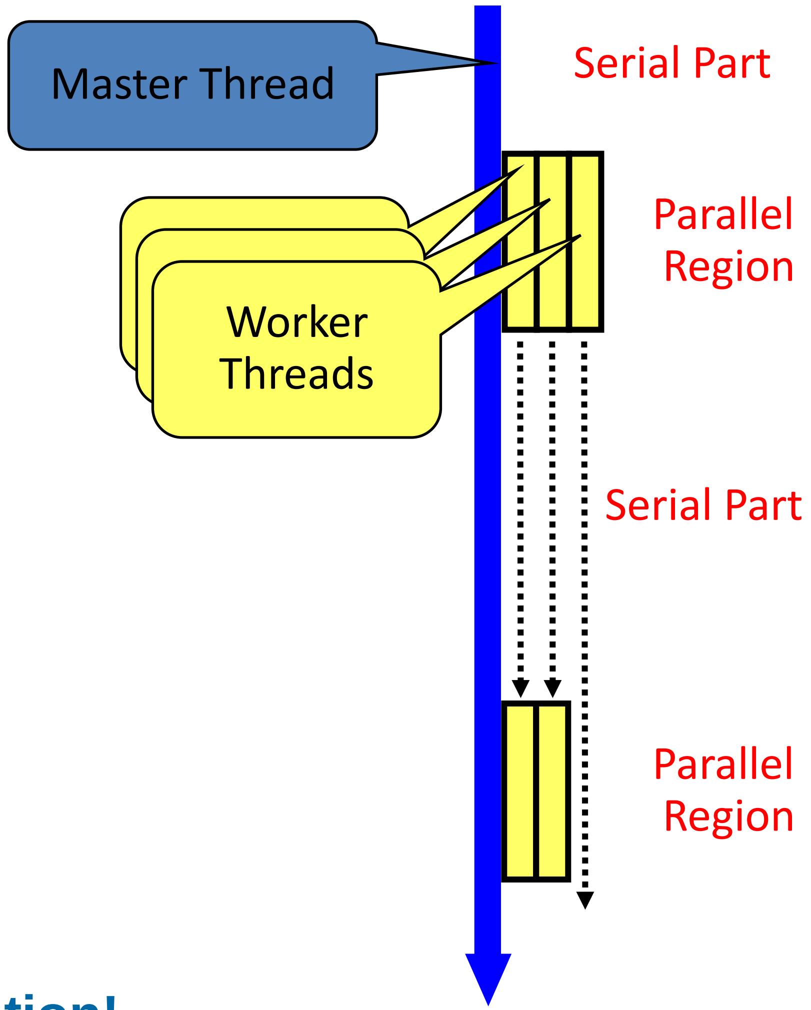
RWTH Aachen University is a member of the OpenMP Architecture Review Board (ARB) since 2006.

Agenda

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!



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)

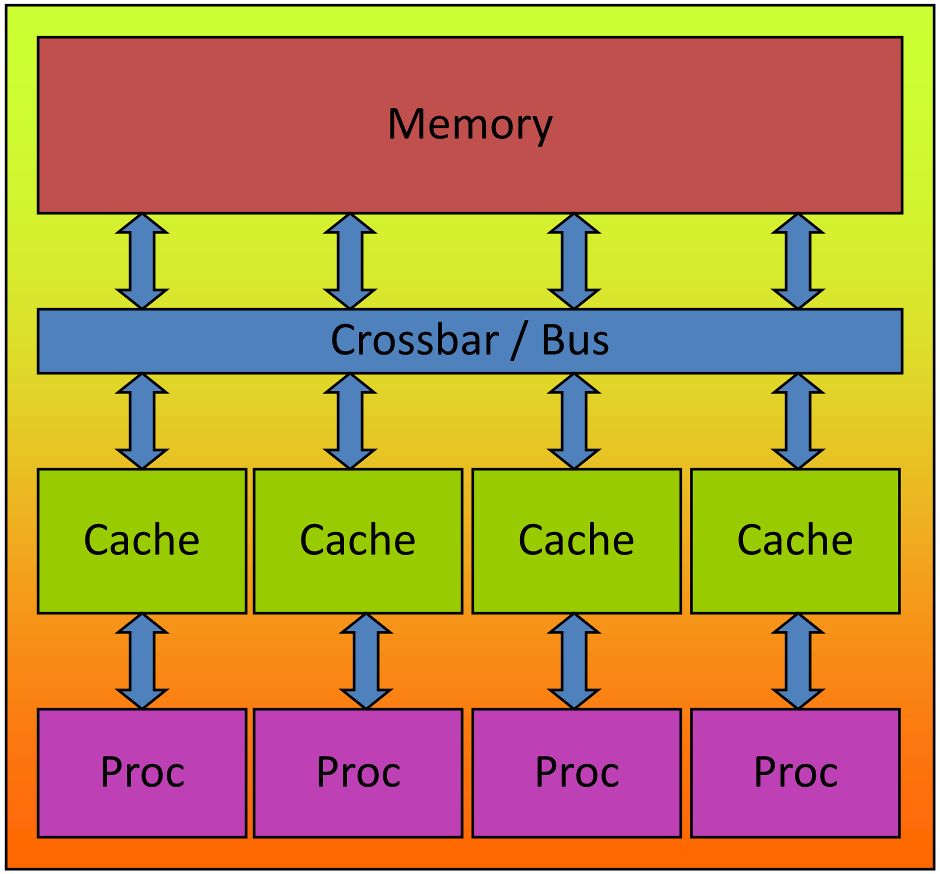
- **Specification of number of threads:**
  - Environment variable:

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

    add `num_threads(num)` to the parallel construct

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.

Hello OpenMP World

Hello orphaned World

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

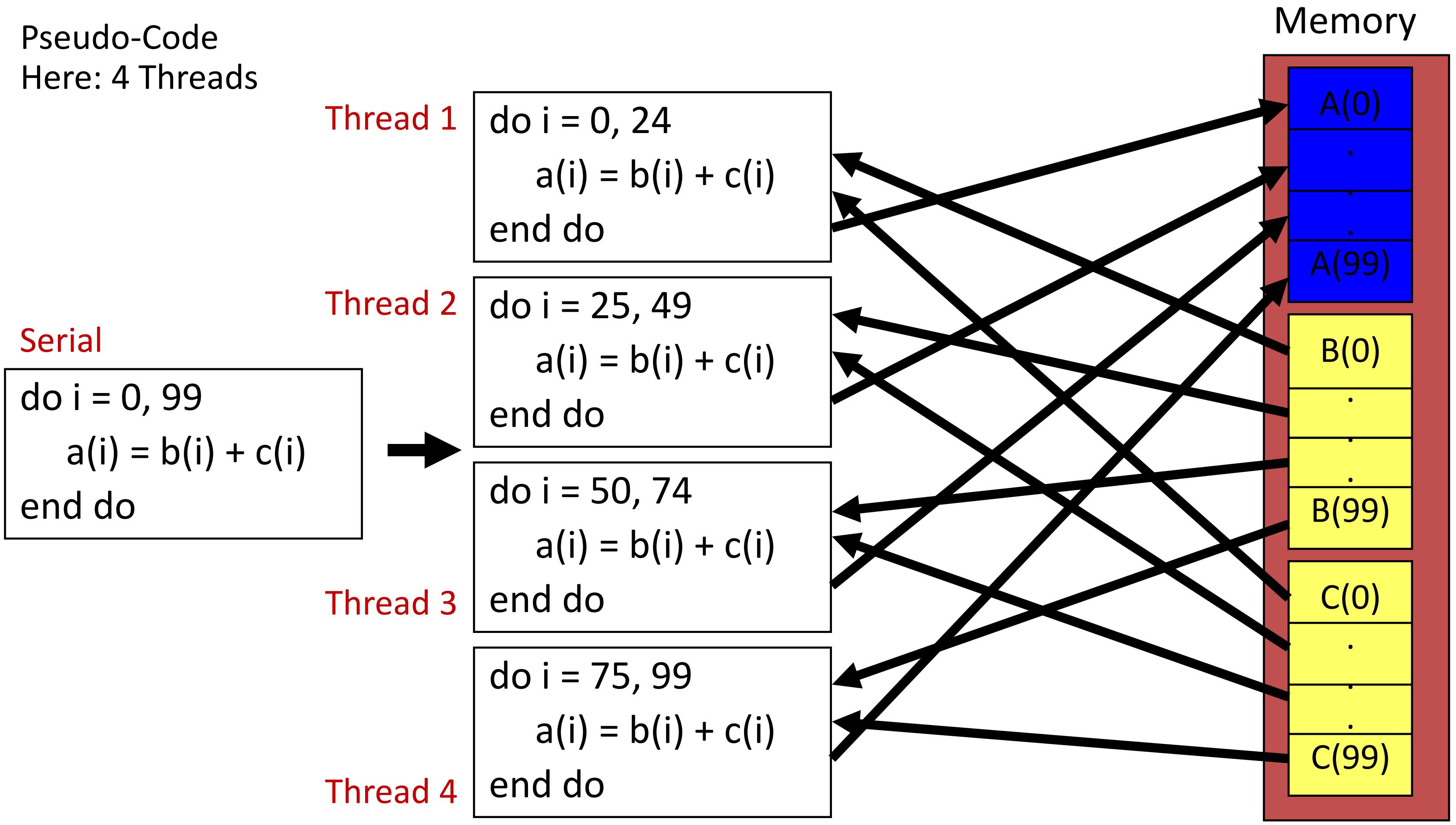
Fortran

```
INTEGER :: i
!$omp parallel 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



Vector Addition

Schedule Clause

Load Imbalance

Influencing the For Loop Scheduling

- ***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 on most implementations is `schedule(static)`.**

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…

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*
  - 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*

Scoping Rules

- Managing the Data Environment is the challenge of OpenMP.

- *Scoping* in OpenMP: Dividing variables in *shared* and *private*:

Recommendation:
use the `default(none)` clause on a Parallel Region to force yourself to think about the scope of every single variable!

  - *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)
```

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;
#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 ...
}
```

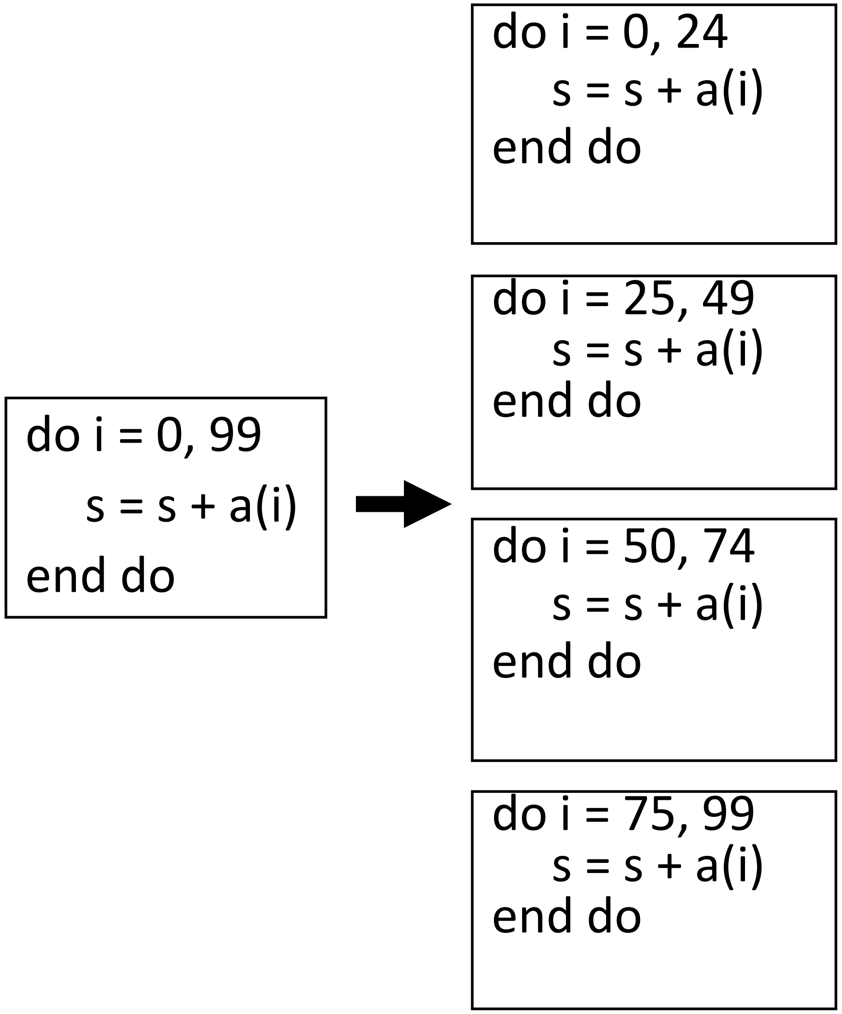
- **Do you think this solution scales well?**

C/C++

```
int i;
#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
```

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++

```
#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 (least number), max (largest number)
```

The Barrier Construct

- 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
```

The nowait Clause

- **A worksharing construct (do/for, sections, single) has no barrier on entry – however, an implied barrier exists at the end of the worksharing region, unless the *nowait* clause is specified.**

- **Static schedule guarantees since OpenMP 3.0:**

```
#pragma omp for schedule(static) nowait
   for(i = 1; i < N; i++)
      a[i] = …
#pragma omp for schedule(static)
   for (i = 1; i < N; i++)
      c[i] = a[i] + …
```

Allowed in OpenMP 3.0 if and only if:
 - Number of iterations is the same
 - Chunk is the same (or not specified)

Example

PI

Example: Pi (1/2)

- **Simple example: calculate Pi by integration**

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


void computePi() {
   double h = (double)1.0 / (double)iNumIntervals;
   double sum = 0, x;


...
   for (int i = 1; i <= iNumIntervals; i++) {
      x = h * ((double)i - (double)0.5);
      sum += f(x);
   }


   myPi = h * sum;
}
```

$$\Pi = \int_0^1 \frac{4}{(1+x^2)}dx$$

Example: Pi (1/2)

- **Simple example: calculate Pi by integration**

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


void computePi() {
   double h = (double)1.0 / (double)iNumIntervals;
   double sum = 0, x;

#pragma omp parallel for private(x) reduction(+:sum)
   for (int i = 1; i <= iNumIntervals; i++) {
      x = h * ((double)i - (double)0.5);
      sum += f(x);
   }

   myPi = h * sum;
}
```

$$\Pi = \int_0^1 \frac{4}{(1+x^2)}dx$$

Example: Pi (2/2)

- **Results (with C++ version):**

# Threads	Runtime [sec.]	Speedup
1	1.11	1.00
2		
4		
8	0.14	7.93

- **Scalability is pretty good:**
  - About 100% of the runtime has been parallelized.
  - As there is just one parallel region, there is virtually no overhead introduced by the parallelization.
  - Problem is parallelizable in a trival fashion ...

Runtime Library

Runtime Library

- **C and C++:**
  - If OpenMP is enabled during compilation, the preprocessor symbol `_OPENMP` is defined. To use the OpenMP runtime library, the header `omp.h` has to be included.
  - `omp_set_num_threads(int)`: The specified number of threads will be used for the parallel region encountered next.
  - `int omp_get_num_threads`: Returns the number of threads in the current team.
  - `int omp_get_thread_num()`: Returns the number of the calling thread in the team, the Master has always the id 0.
- **Additional functions are available, e.g. to provide locking functionality.**