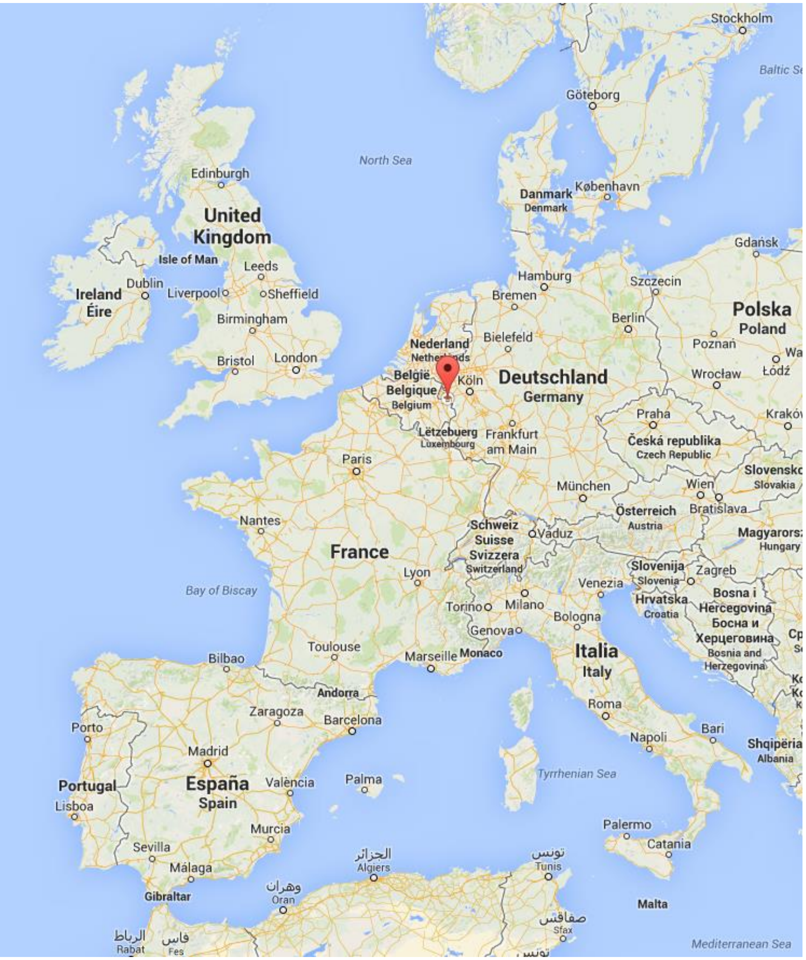
Basics of OpenMP (Recap)

Dirk Schmidl
IT Center, RWTH Aachen University
Member of the HPC Group
schmidl@itc.rwth-aachen.de

Christian Terboven
IT Center, RWTH Aachen University
Deputy lead of the HPC Group
terboven@itc.rwth-aachen.de

RWTH Aachen University

- → One of the largest technical universities in Germany.
- → ~ 36.000 students
- → ~ 500 professors
- → ~ 4500 academic staff



High Performance Computing Group at the IT Center

- → application support /optimization
- → focus on shared memory programming
- → member of OpenMP ARB

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 release

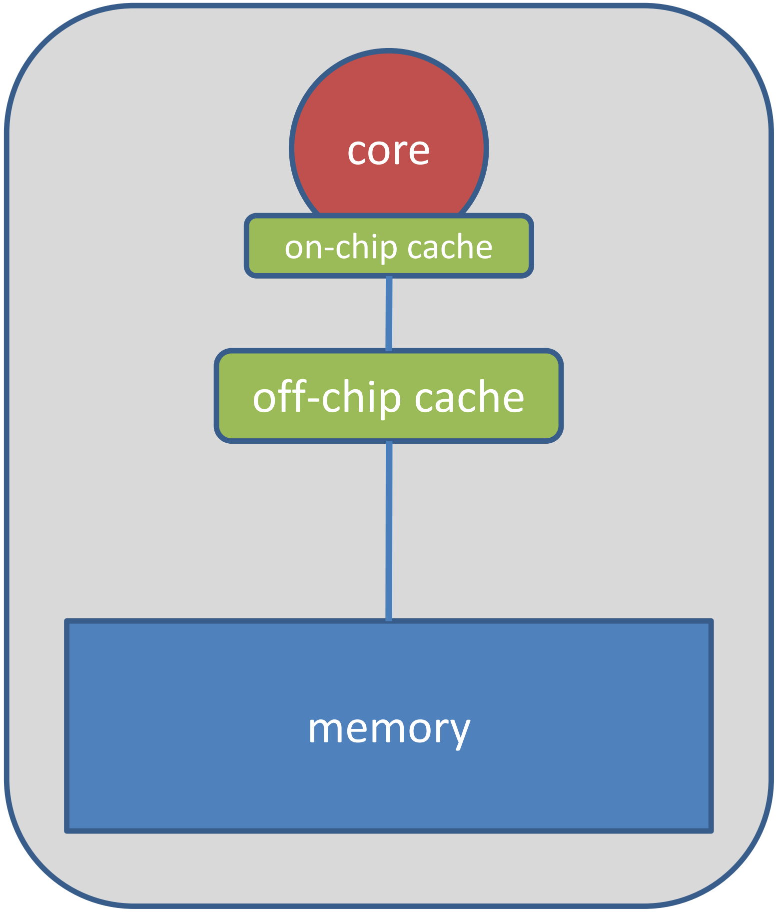
http://www.OpenMP.org

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

Shared Memory Architectures

Single Processor System (dying out)

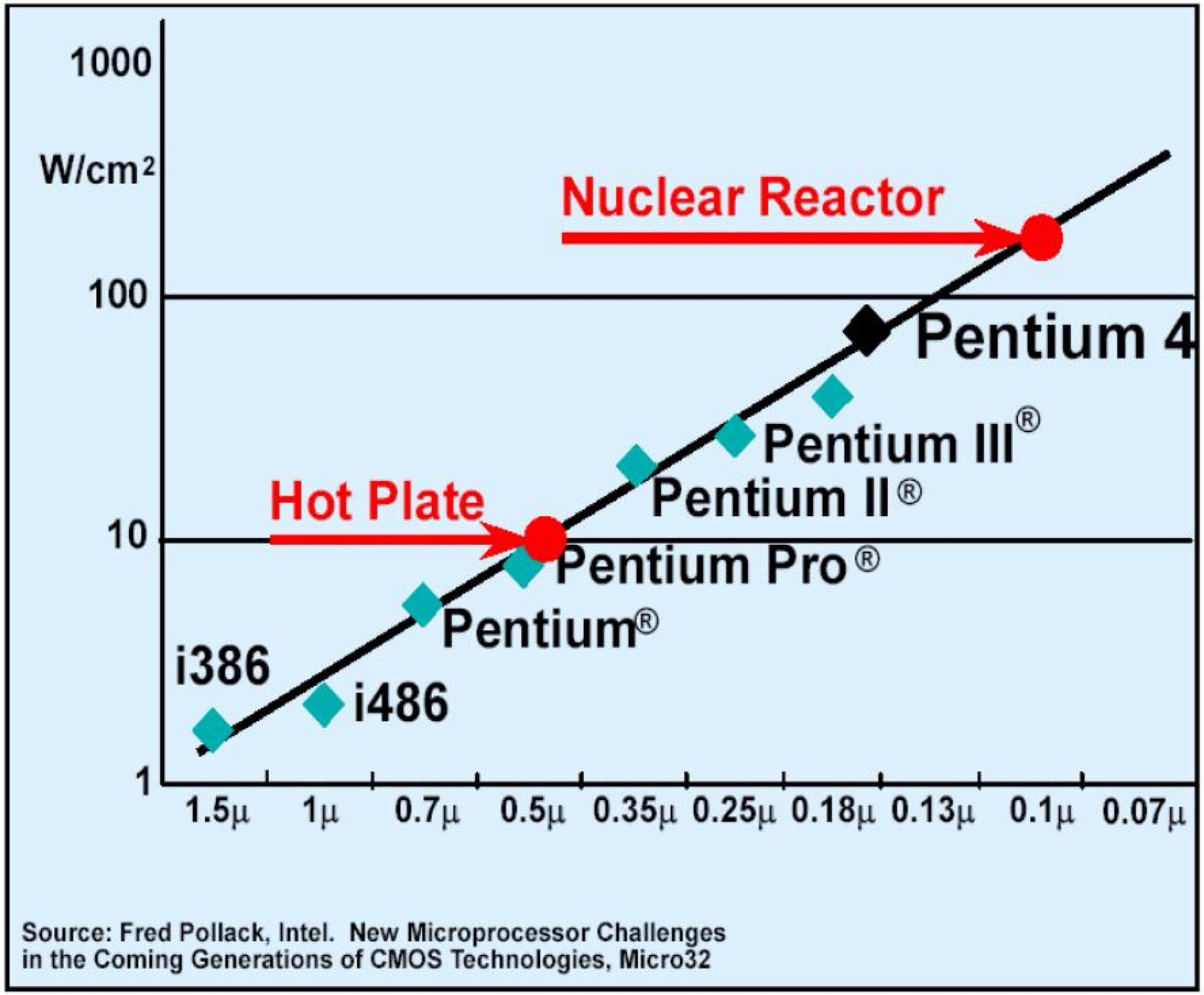
- **CPU is fast**
  - → Order of 3.0 GHz
- **Caches:**
  - → Fast, but expensive
  - → Thus small, order of MB
- **Memory is slow**
  - → Order of 0.3 GHz
  - → Large, order of GB



- **A good utilization of caches is crucial for good performance of HPC applications!**

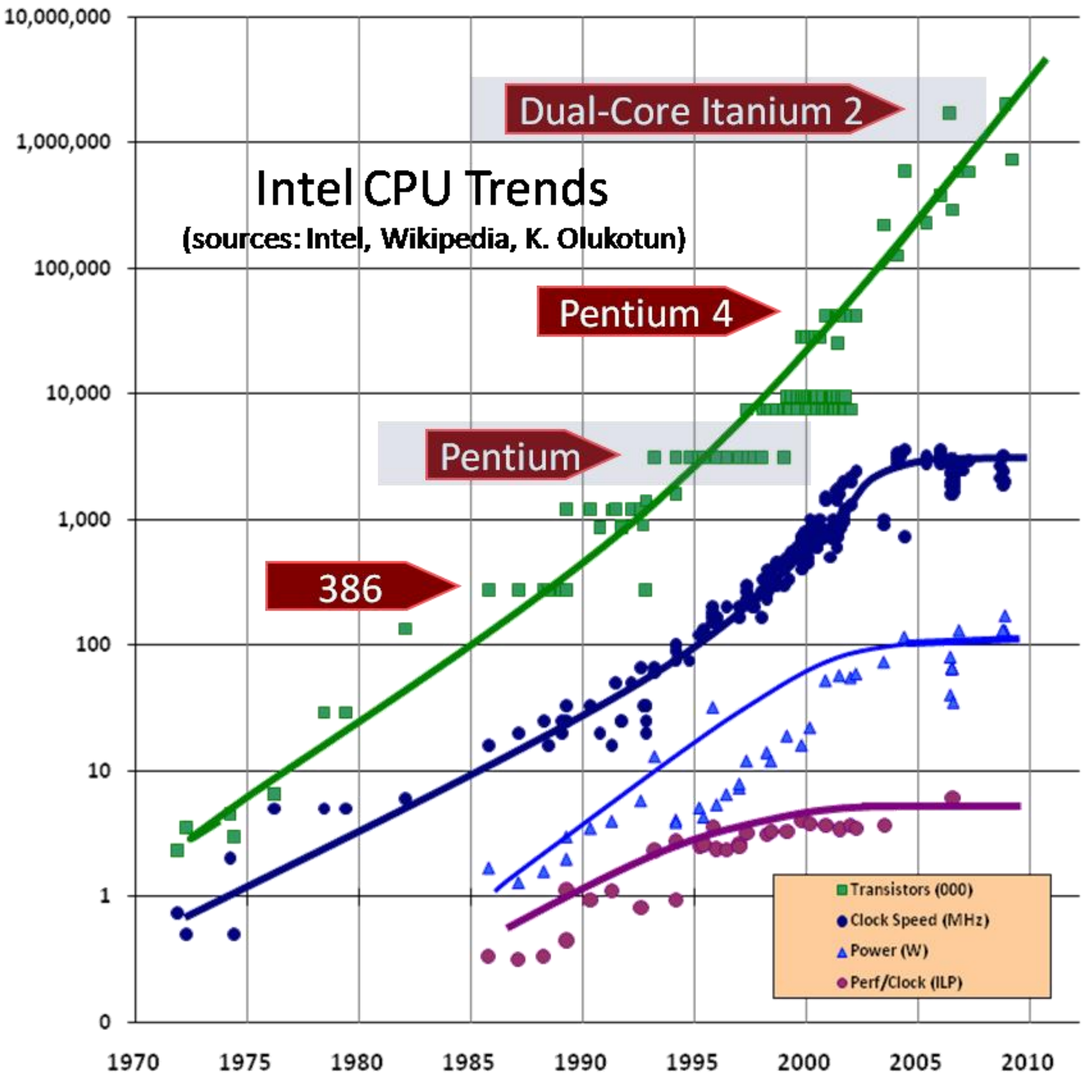
Why is there no 4.0 GHz x86 CPU?

- **Because that beast would get too hot!**



Fast clock cycles make processor chips more expensive, hotter and more power consuming.

Moore's Law still holds!



The number of transistors on a chip is still doubling every 24 months …

… but the clock speed is no longer increasing that fast!

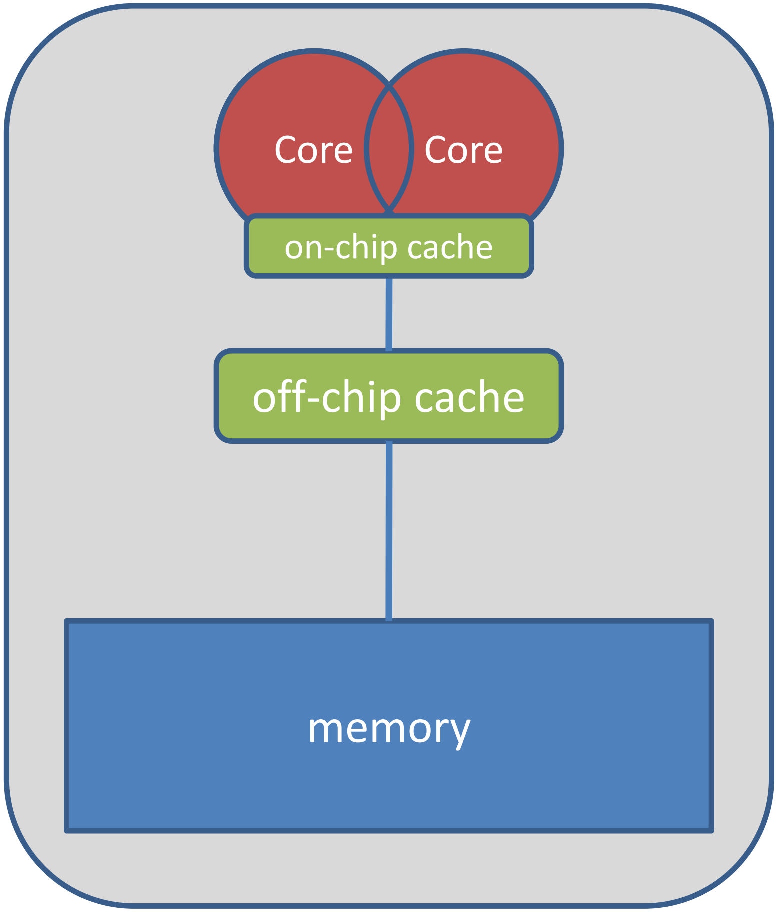
Instead, we will see many more cores per chip!

Source: Herb Sutter

www.gotw.ca/publications/concurrency-ddj.htm

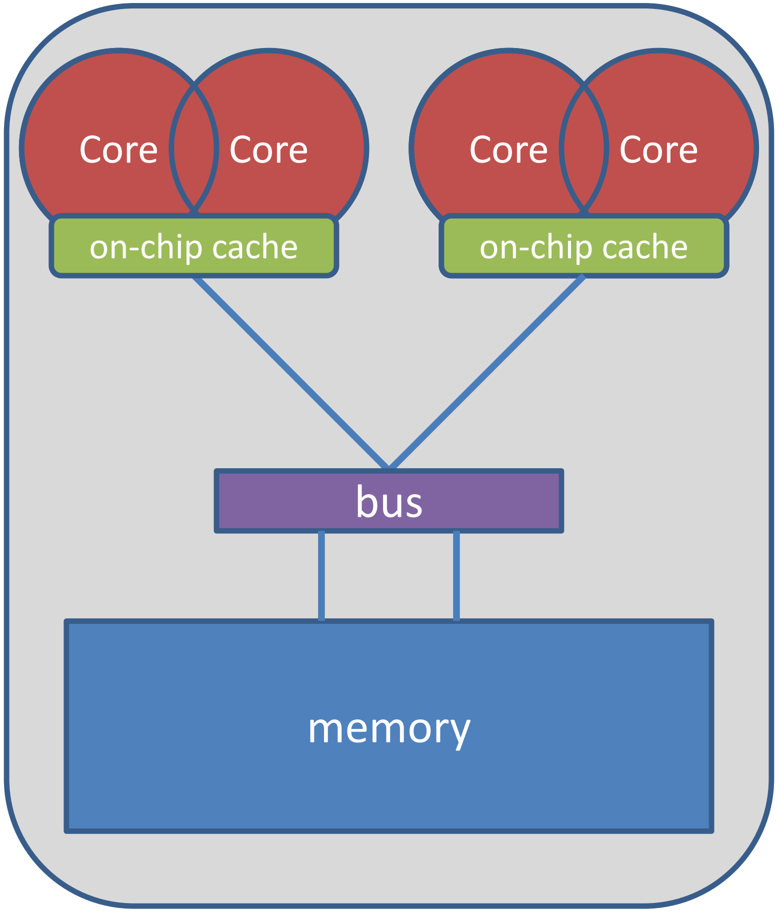
Dual-Core Processor System

- **Since 2005/2006 Intel and AMD are producing dual-core processors for the mass market!**
- **In 2006/2007 Intel and AMD introduced quad-core processors.**
- **→ Any recently bought PC or laptop is a multi-core system already!**



Example for a SMP system

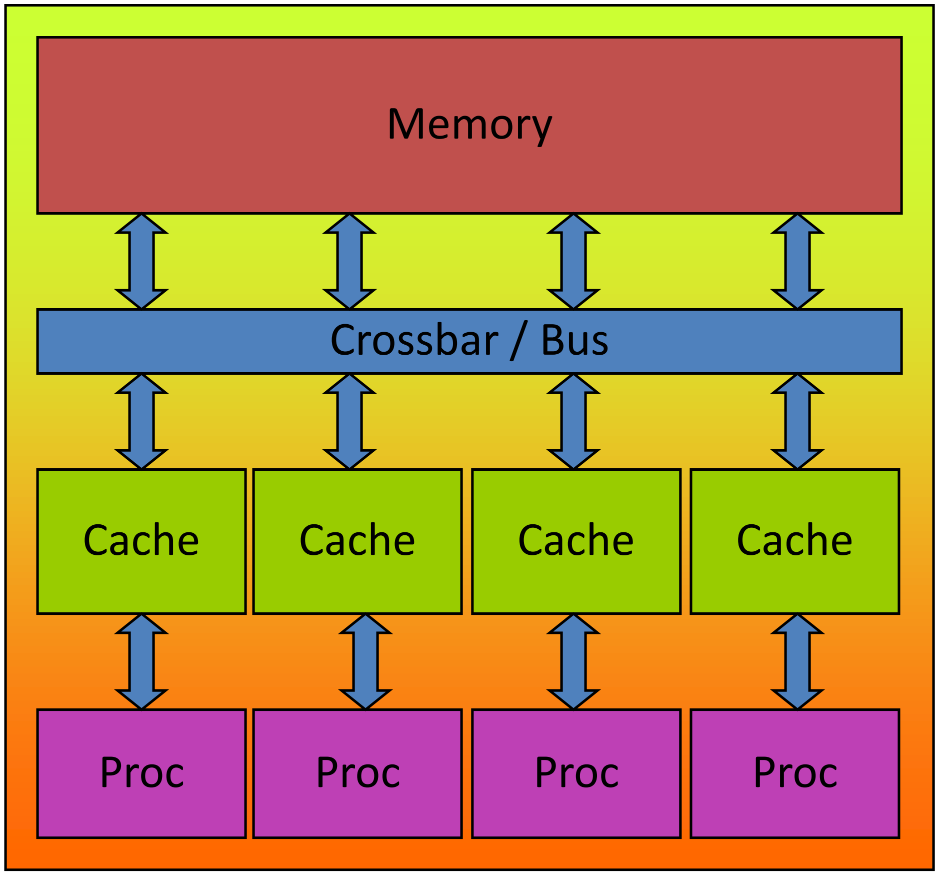
- **Dual-socket Intel Woodcrest (dual-core) system**
  - → Two cores per chip, 3.0 GHz
  - → Each chip has 4 MB of L2 cache on-chip, shared by both cores
  - → No off-chip cache
  - → Bus: Frontsidebus
- **SMP: Symmetric Multi Processor**
  - → Memory access time is uniform on all cores
  - → Limited scalabilty



OpenMP Overview
&
Parallel Region
&
Basic Worksharing

OpenMP‘s machine model

- **OpenMP: Shared-Memory Parallel Programming Model.**



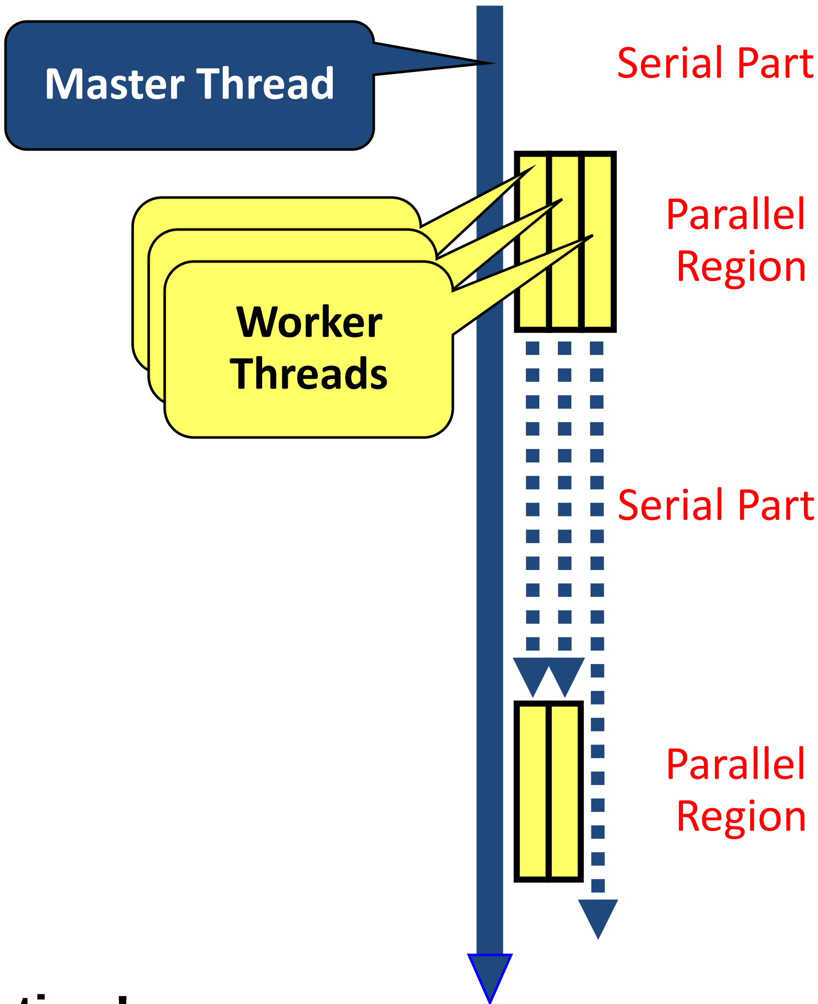
All processors/cores access a shared main memory.

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

Parallelization in OpenMP employs multiple threads.

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

Hello OpenMP World

Hello orphaned OpenMP World

Starting OpenMP Programs on Linux

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

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

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

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

- **Intel Compiler on Linux: ask the runtime for more information:**

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

If Clause: Parallel Region and Worksharing

- **If the expression of an `if` clause on a *Parallel Region* evaluates to false**
  - → The Parallel Region is executed with a Team of one Thread only
  - → Used for optimization, e.g. avoid going parallel
- **OpenMP data scoping rules still apply!**

C/C++

```
#pragma omp parallel if(expr)
    ...
```

Fortran

```
!$omp parallel if(expr)
    ...
```

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

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

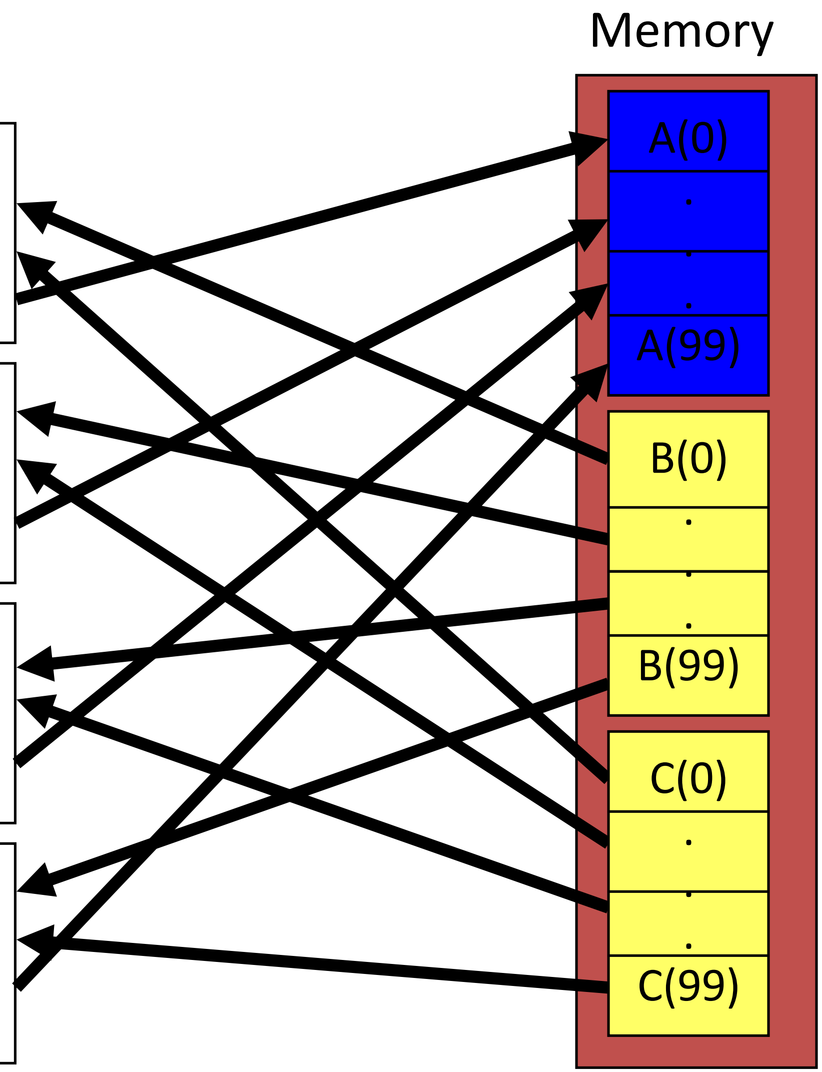
```
do i = 25, 49
    a(i) = b(i) + c(i)
end do
```

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

Example: Sparse Matrix Vector Mult.

- $A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 4 & 0 & 4 & 4 \end{pmatrix}$

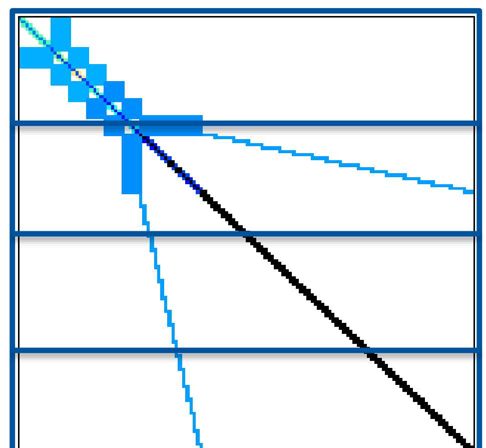
```
for (i = 0; i < num_rows; i++){
  sum = 0.0;
  for (nz=row[i]; nz<row[i+1]; ++nz){
    sum+= value[nz]*x[index[nz]];
  }
  y[i] = sum;
}
```

- **Format: compressed row storage**

- **store all values and columns in arrays (length nnz)**
- **store beginning of a new row in a third array (length n+1)**

value:	1	2	2	3	4	4	4
index:	0	0	1	2	0	2	3

row:	0	1	3	4	7



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)`.**

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 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 now…

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 (largest number), max (least 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

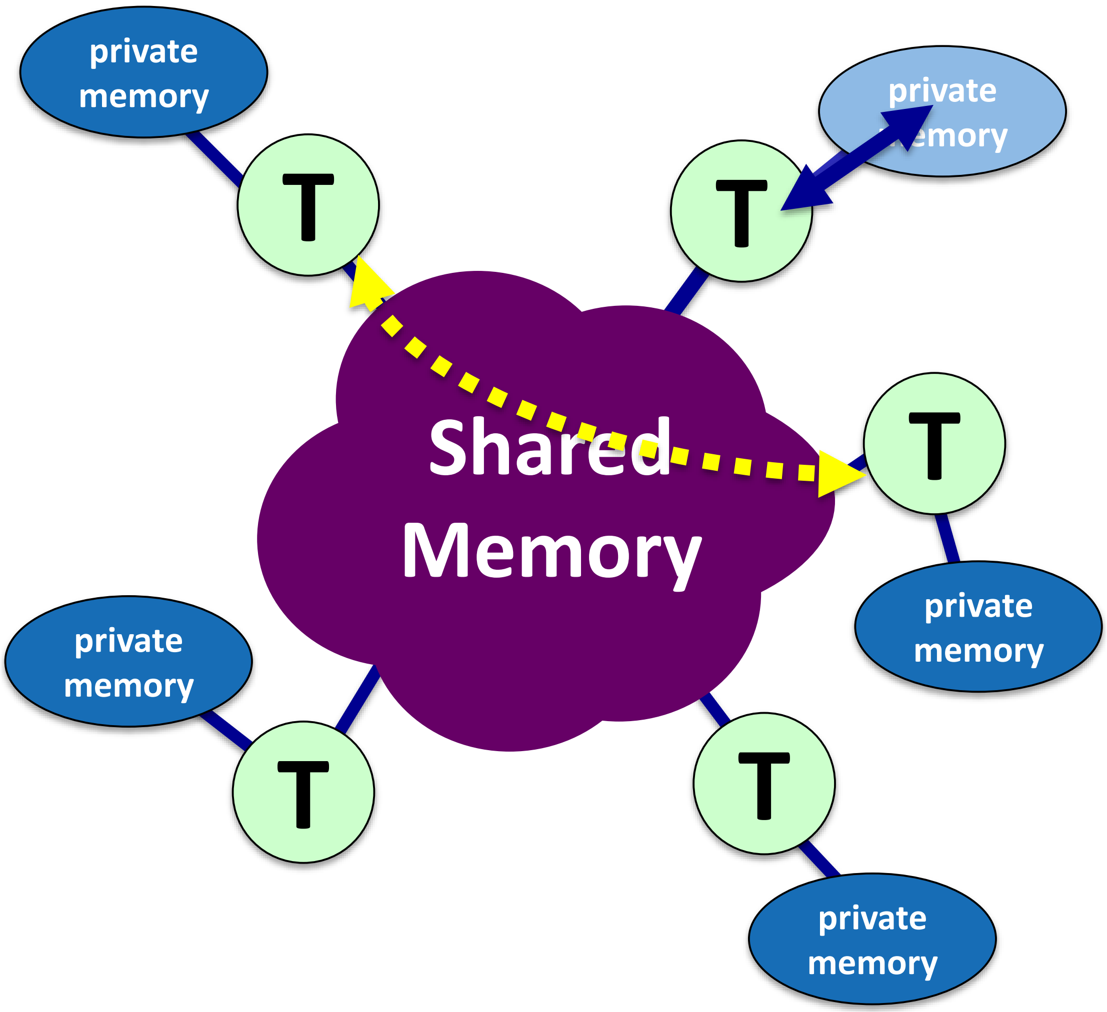
C/C++

```
#pragma omp barrier
```

- **All worksharing constructs have an implied barrier**
  - → This is a safety net
- **In some cases, the implied barrier can be left out through the “nowait” clause**

Data Scoping

The Memory Model



- All threads have access to the same, globally shared memory
- Data in private memory is only accessible by the thread owning this memory
- No other thread sees the change(s) in private memory
- Data transfer is through shared memory and is 100% transparent to the application

Gotcha's

- **Need to get this right**
  - → Part of the learning curve
- **Private data is undefined on entry and exit**
  - → Can use firstprivate and lastprivate to address this
- **Each thread has its own temporary view on the data**
  - → Applicable to shared data only
  - → Means different threads may temporarily not see the same value for the same variable ...
  - → All threads have a consistent view of the memory after synchronization constructs
    - → Technically: synchronization constructs contain a flush construct…

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++	Fortran
`static int i;` `#pragma omp threadprivate(i)`	`SAVE INTEGER :: i` `!$omp threadprivate(i)`

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

Questions?