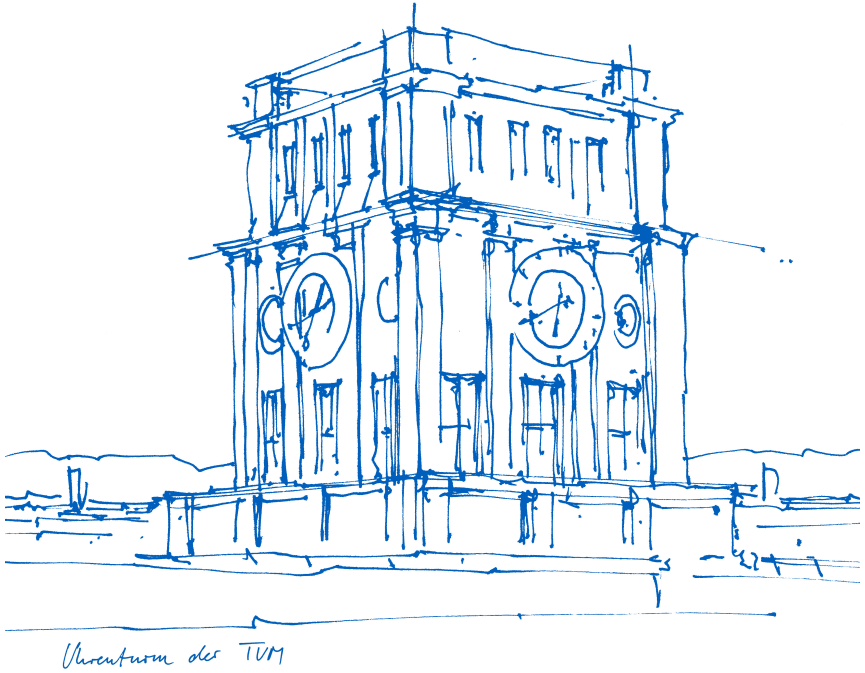
Lecture IN-2147
Parallel Programming

SoSe 2018

Martin Schulz
Exercises: Amir Raoofy
Technische Universität München
Fakultät für Informatik

Lecture 3:
OpenMP Basics



Summary from Last Time / Threading

A thread is a stream of execution

Hardware threads
- Implementation of the “von Neumann” control unit (CU)
- Complicated by multi-/many core and Hyperthreading/SMT

Software threads are abstracting execution streams for the programmer
- Distinguish user- and system-level threads

Widely available and standardized API: POSIX threads
- Routines for thread creation/destruction
- Several synchronization constructs, incl. mutexes and condition variables

Performance challenges
- Reduce thread management overhead
- Lock contention
- Bottlenecks in the memory system
- Missing thread pinning can lead to bad HW/SW mappings

Drawbacks of Using Pthreads

Pthreads represent direct abstraction of system-level parallelism

- Direct map to underlying hardware threads (in most cases)
- Even parallel loops are difficult to think about

Association of data and threads is up to the programmer

- No program construct to do this
- Data locality has to be done explicitly

Need to manage data visibility by hand

- Which variables are per thread
- Which variables are shared across all threads

Simple synchronization primitives

- Locks and condition variables
- Not sufficient to do easy parallel coordination

Finding a Higher Level of Abstraction

In the 1990's several vendors supplied their own language extensions

- Driven by rising popularity of shared memory machines
- Mainly as extensions to Fortran
- Users could specify parallel loops
- A compiler would then translate the code to a threaded program

Very helpful for users

- Small, often incremental additions to code
  - In many cases, one line or one pragma
  - Codes stayed readable
- Compiler/runtime would do the engineering work

But: each vendor had their extensions, which made codes non-portable

- First attempt at standardizing failed (ANSI X3H5) in 1994
- A few years later, another attempt led to OpenMP

OpenMP = Open Multi-Processing

Standard for writing parallel programs, mainly “on-node”

- Standardization across many vendors, systems, architectures, …
- “Lean and Mean” – simple API to achieve complex goals
- Ease of Use
- Portability

Managed by the OpenMP ARB

- Architecture Review Board
- Independent non-profit organization
- Members are companies, universities, research labs

First version published in 1997

- Many advances since then
  - Tasking, GPU support
- Currently at Version 4.5, Version 5.0 expected in November
- All documents (and more) at www.openmp.org
- More next week directly from the OpenMP CEO Michael Klemm!

OpenMP

OpenMP is …

... an Application Program Interface (API) that may be used to program multi-threaded, shared memory parallelism (plus accelerators)

... comprised of three primary API components:
- Compiler Directives (for C/C++ and Fortran)
- Runtime Library Routines
- Environment Variables

OpenMP is not ...

... intended for distributed memory systems
... necessarily implemented identically by all vendors
... guaranteed to automatically make the most efficient use of shared memory
... required to check for data dependencies, race conditions, deadlocks, etc.
... designed to handle parallel I/O

A Simple Example

```
#include <omp.h>

main(){
    #pragma omp parallel
    {
     printf("Hello world");
    }
}
```

```
> export OMP_NUM_THREADS=2
> a.out
Hello world
Hello world
```

```
> export OMP_NUM_THREADS=3
> a.out
Hello world
Hello world
Hello world
```

Compilation

- `icc –O3 –openmp openmp.c`
- `gcc -O3 –fopenmp openmp.c`

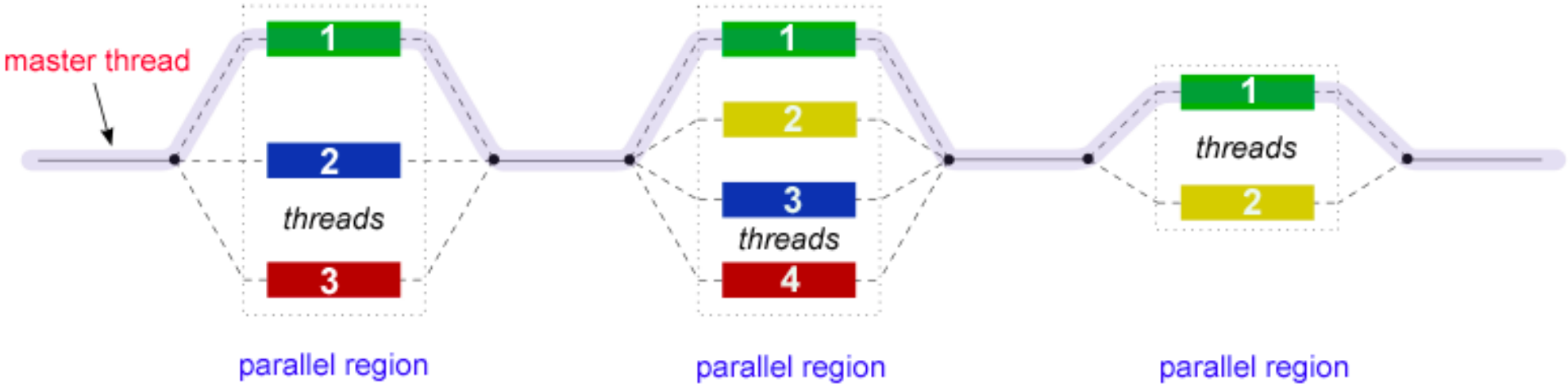
- This differs between compilers

Fork/Join Execution Model

Parallel Regions

1. An OpenMP-program starts as a single thread (*master thread*).
2. Additional threads are created when the master hits a parallel region.
3. After the parallel region, the new threads are given back to the runtime
4. The master continues after the parallel region.

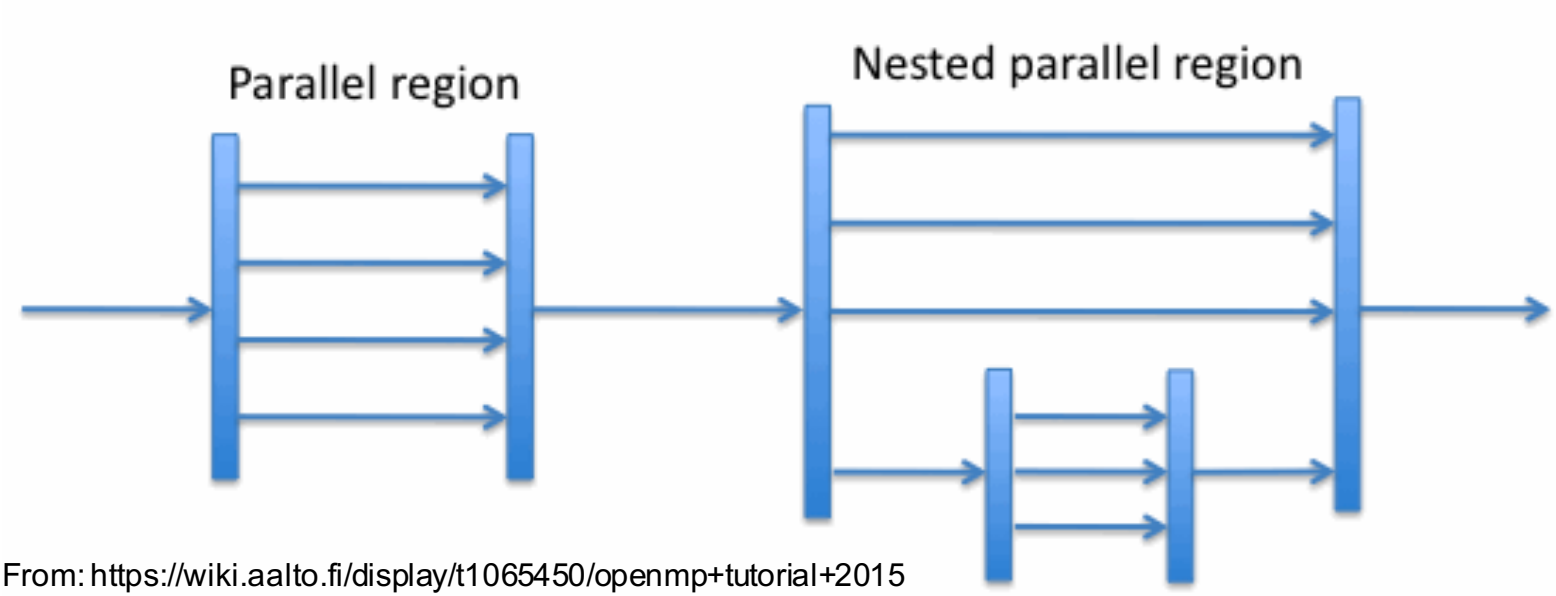
All threads in the team are synchronized at the end of a parallel region via a barrier.



From: Blaise Barney, https://computing.llnl.gov/tutorials/openMP/

Nested Parallelism

OpenMP threads can themselves create parallel regions



From: https://wiki.aalto.fi/display/t1065450/openmp+tutorial+2015

Noteworthy

- OpenMP is not required to spawn/use more threads in the nested region
  - It is compliant to just continue executing sequentially
- Mapping to hardware threads is up to the runtime

OpenMP Implementations

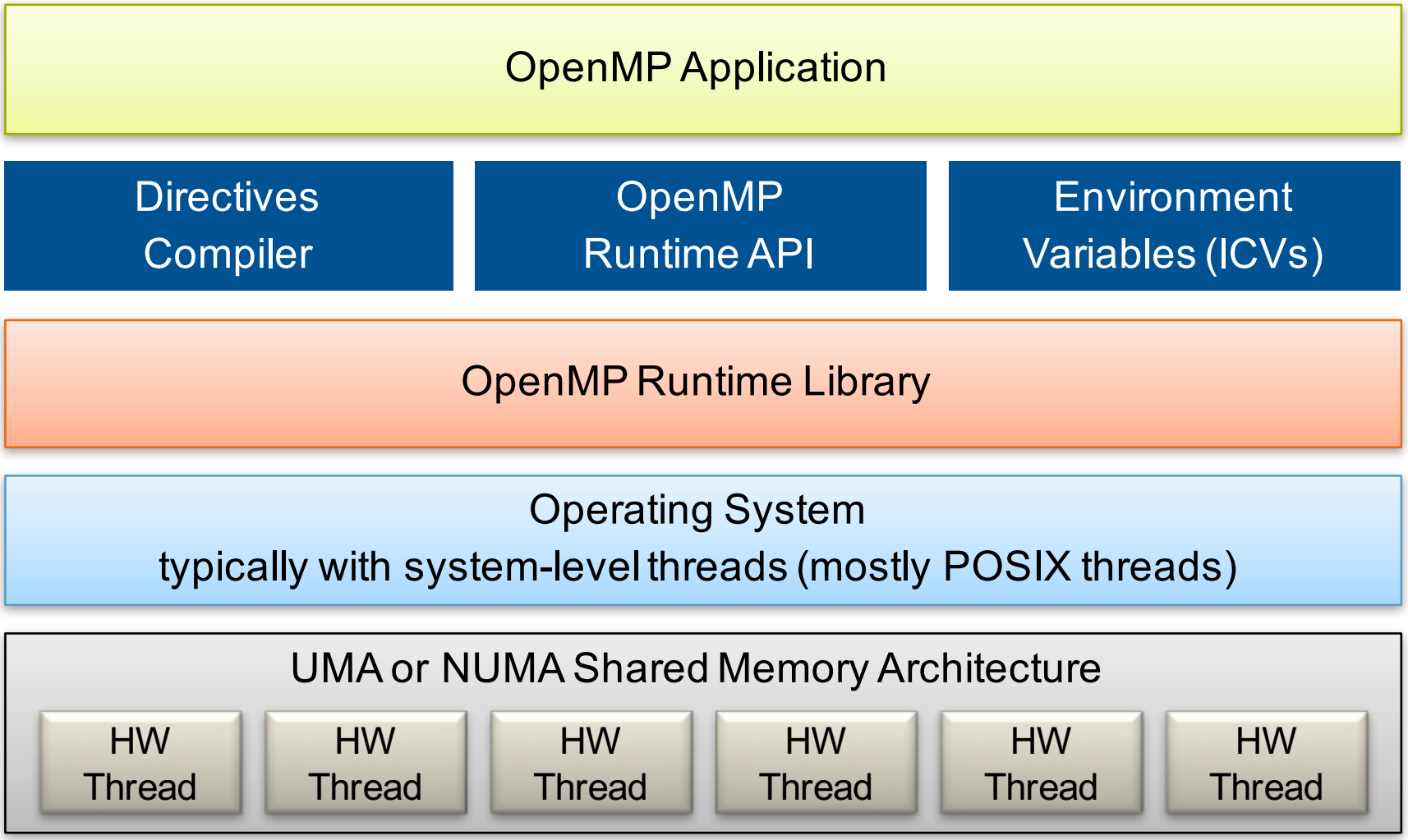
OpenMP is a language extension
- On top of C/C++ or Fortran
- Pragmas to the base language, which can be ignored

Consequence: need a new compiler
- Implemented within existing compilers
- Most compilers support this now: gcc, icc, LLVM, PGI, ...

Additionally: need a runtime system
- Maps program to underlying thread package
- Often built on top of Pthreads
- Standard defines some user functions, but NOT the entire interface
- Two widely known open source systems: gomp and Intel‘s OpenMP/LLVM runtime

Well defined environment variables, also called „internal control variables“ or „ICVs“

OpenMP System Stack



OpenMP Syntax

Most of the constructs in OpenMP are compiler directives

```
#pragma omp construct [clause [clause]…]
```

Example: `#pragma omp parallel`

Most OpenMP constructs apply to a “structured block”.

- A block of one or more statements
- With one point of entry at the top and one point of exit at the bottom
- Think: C/C++

```
{ … }
```

Additionally: runtime API

- Function prototypes and types in the `omp.h` header file
- Namespace: `omp_`

Fortran

OpenMP also defines pragmas for Fortran

- Same concepts and similar pragmas
- Different syntax
- Need to deal with scoping differently

```
!$OMP directive name  [parameters]
```

Example:

```
!$OMP PARALLEL DEFAULT(SHARED)
        write(*,*) ´Hello world´
!$OMP END PARALLEL
```

More on Directives Syntax

Directives can have continuation lines

- C

```
#pragma omp parallel private(i) \
        private(j)
```

- Fortran

```
!$OMP directive name first_part &
!$OMP continuation_part
```

Parallel Regions

Parallel regions are marked explicitly

```
#pragma omp parallel [parameters]
{
        block
}
```

Block executed in parallel

Degree of parallelism controlled by ICV

OMP_NUM_THREADS

A Simple Example (repeated)

```
#include <omp.h>

main(){
    #pragma omp parallel
    {
     printf("Hello world");
    }
}
```

Compilation

- `icc –O3 –openmp openmp.c`
- `gcc -O3 –fopenmp openmp.c`

- This differs between compilers

```
> export OMP_NUM_THREADS=2
> a.out
Hello world
Hello world

> export OMP_NUM_THREADS=3
> a.out
Hello world
Hello world
Hello world
```

Sections in a Parallel Region

Parallel regions can have separate sections

```
#pragma omp parallel [parameters]
{
 #pragma omp section
        { … block … }
 #pragma omp section
        { … block … }
  ...
}
```

Most “thread like” option

- Parallel regions can have sections
- Each section of a parallel region is executed once by one thread
- Threads that finished their section wait at an implicit barrier at the end

Example: Sections

```
main(){
int i, a[1000], b[1000]

#pragma omp parallel private(i)
 {
 #pragma omp sections
  {
  #pragma omp section
  for (int i=0; i<1000; i++)
    a[i] = 100;
  #pragma omp section
  for (int i=0; i<1000; i++)
    b[i] = 200;
  }
 }
}
```

Work Sharing in a Parallel Region

Need easy way to split tasks among threads
Most important construct: loops

```
main (){
int a[100];
#pragma omp parallel
 {
   #pragma omp for
    for (int i= 1; i<n;i++)
       a(i) = i;
 }
}
```

```
main (){
int a[100];

#pragma omp parallel for
    for (int i= 1; i<n;i++)
       a(i) = i;

}
```

Creates a parallel region
Splits iterations of **`for`** loop among threads

Parallel Loop

```
#pragma omp for [parameters]
     for ...
```

- The iterations of the do-loop are distributed to the threads
- There is no synchronization at the beginning
- All threads of the team synchronize at an implicit barrier
  - Unless the parameter **`nowait`** is specified
- Note: the expressions in the for-statement are very restricted

Iterations must be independent

- No data dependencies
- Can be executed in any order
- Programmer responsibility

How do Loops get Split up?

Iterations must be distributed to threads

Loop Schedule

- Defines how iterations are split up
  - Leads to the creation of “Chunks”
- Defines how chunks are mapped to threads

OpenMP offers several options

- Specified using the `schedule` parameter

Example:

- `#pragma omp for schedule(static)`

Available Loop Schedules

`static`
- Fix sized chunks (default size is about n/t)
- Distributed in a round-robin fashion

`dynamic`
- Fix sized chunks (default size is 1)
- Distributed one by one at runtime as chunks finish

`guided`
- Start with large chunks, then exponentially decreasing size
- Distributed one by one at runtime as chunks finish

`runtime`
- Controlled at runtime using control variable

`auto`
- Compiler/Runtime can choose

Examples: Scheduling

```
#define S 25
int main(int argc, char** argv)
{ int a[S],b[S],c[S];

#pragma omp parallel
{
 #pragma omp for schedule(static)
  for (int i=0; i<S;i++)
    a[i] = omp_get_thread_num();

 #pragma omp for schedule(dynamic, 4)
  for (int i=0; i<S;i++)
    b[i] = omp_get_thread_num();

 #pragma omp for schedule(guided)
  for (int i=0; i<S;i++)
    c[i] = omp_get_thread_num();
 }

 for (int i=0; i<S;i++)
    printf("Iter %4d: %4d %4d %4d\n",i,a[i],b[i],c[i]);
}
```

```
Iter    0:  0    3    2
Iter    1:  0    3    2
Iter    2:  0    3    2
Iter    3:  0    3    1
Iter    4:  0    1    1
Iter    5:  0    1    0
Iter    6:  0    1    0
Iter    7:  1    1    1
Iter    8:  1    0    1
Iter    9:  1    0    0
Iter   10:  1    0    0
Iter   11:  1    0    2
Iter   12:  1    1    1
Iter   13:  2    1    0
Iter   14:  2    1    2
Iter   15:  2    1    0
Iter   16:  2    3    2
Iter   17:  2    3    1
Iter   18:  2    3    0
Iter   19:  3    3    2
Iter   20:  3    0    0
Iter   21:  3    0    2
Iter   22:  3    0    1
Iter   23:  3    0    0
Iter   24:  3    1    2
```

Shared vs. Private Data

Important decision: visibility of data / variables

Shared data
- Accessible by all threads
- One copy for all threads
- Example: a reference a[5] to a shared array accesses

Private data
- Accessible only by a single thread
- Each thread has its own copy
- Example: iterator variables

The default for global variables: shared
Variables declared within the “dynamic extent” of parallel regions: local
- Includes routines called from parallel regions

Private clause for parallel loop

```
main (){
int a[100], t;

#pragma omp parallel
 {
   #pragma omp for private(t)
    for (int i= 1; i<n;i++){
       t=f(i);
       a[i]=t;
     }
 }
}
```

Global variable: a, t

Local variable: i

Variable t by default shared, but now “privatized”

Example / Thread Identities

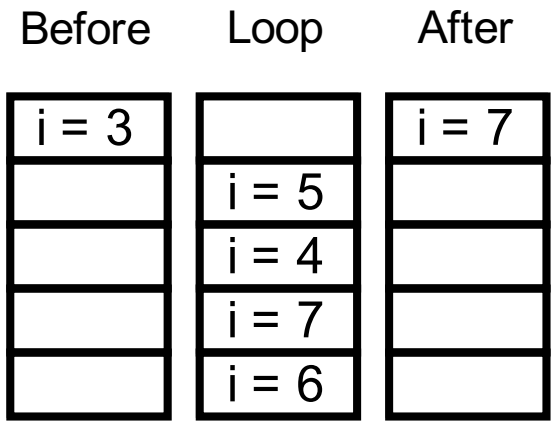
```
main (){
int iam, nthreads;

#pragma omp parallel private(iam,nthreads)
 {
  iam = omp_get_thread_num();
  nthreads = omp_get_num_threads();
  printf("ThradID %d, out of %d threads\n", iam, nthreads);

  if (iam == 0)
    printf("Here is the Master Thread.\n");
  else
    printf("Here is another thread.\n");
 }
}
```

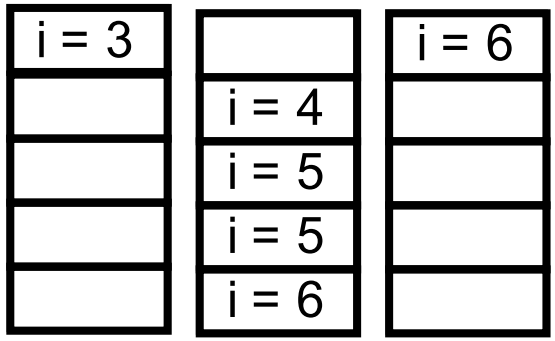
Example / Thread Identities – Better Option

```
main (){


#pragma omp parallel
 {
  int iam = omp_get_thread_num();
  int nthreads = omp_get_num_threads();
  printf("ThradID %d, out of %d threads\n", iam, nthreads);

  if (iam == 0)
    printf("Here is the Master Thread.\n");
  else
    printf("Here is another thread.\n");
 }
}
```

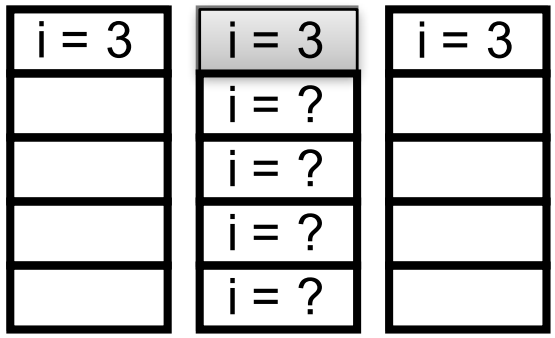
Private Data Options

```
int i=3;
#pragma omp parallel for
  for (int j=0; j<4; j++)
    { i=i+1;
      printf("-> i=%d\n", i); }
printf("Final Value of I=%d\n", i);
```



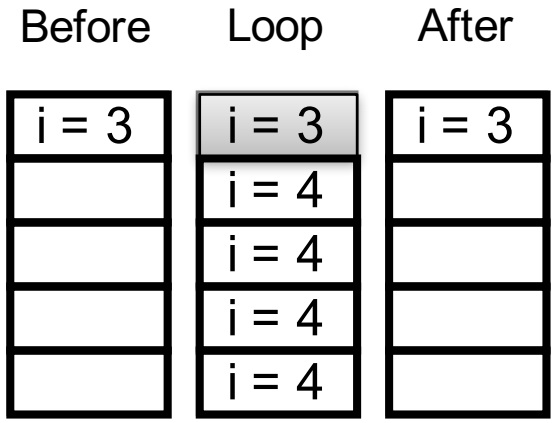



```
int i=3;
#pragma omp parallel for private(i)
  for (int j=0; j<4; j++)
    { i=i+1;
      printf("-> i=%d\n", i); }
printf("Final Value of I=%d\n", i);
```

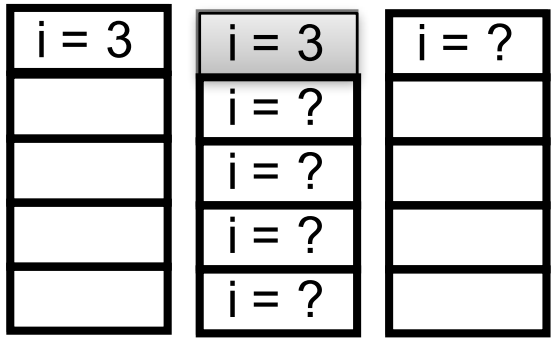


First/Last Private Data Options

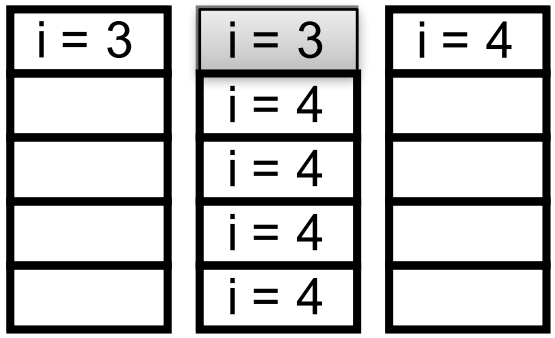
```
int i=3;
#pragma omp parallel for firstprivate(i)
  for (int j=0; j<4; j++)
    { i=i+1;
      printf("-> i=%d\n", i); }
printf("Final Value of I=%d\n", i);
```




```
int i=3;
#pragma omp parallel for lastprivate(i)
  for (int j=0; j<4; j++)
    { i=i+1;
      printf("-> i=%d\n", i); }
printf("Final Value of I=%d\n", i);
```




```
int i=3;
#pragma omp parallel for firstprivate(i) \
                         lastprivate(i)
  for (int j=0; j<4; j++)
    { i=i+1;
      printf("-> i=%d\n", i); }
printf("Final Value of I=%d\n", i);
```



Summary: Sharing Attributes of Variables

`private(var-list)`
- Variables in var-list are private

`shared(var-list)`
- Variables in var-list are shared

`default(private | shared | none)`
- Sets the default for all variables in this region

`firstprivate(var-list)`
- Variables are private
- Initialized with the value of the shared copy before the region.

`lastprivate(var-list)`
- Variables are private
- Value of the thread executing the last iteration of a parallel loop in sequential order is copied to the variable outside of the region.

Barrier

Key synchronization construct

- Synchronizes all the threads in a team
- Each thread waits until all other threads in the team have reached the barrier

Each parallel region has an implicit barrier at the end

- Synchronizes the end of the region
- Can be switched off by adding **`nowait`**

Additional barriers can be added when needed

- **`#pragma omp barrier`**

Warning

- Can cause load imbalance
- Use only when really needed

Reductions

Synchronizing at the end of parallel regions is often not enough

- Need to communicate results of the region
- Aggregate data at master thread
- Inform next parallel region

Reductions

- Perform operations on results from all threads
- Aggregate results
- Make it available to master thread

Problem

- Potentially costly and time sensitive operation
- Hard to implement by oneself
- Should be part of the base language

OpenMP Reductions

OpenMP offers a special clause to specify reductions

```
reduction(operator: list)
```

This clause performs a reduction …

- … on the variables that appear in *list*,
- … with the operator *operator*.

Variables must be shared scalars
operator is one of the following:

```
+, *, -, &, ^, |, &&, ||
```

Reduction variable can only appear in statements with the following form:

- `x = x operator expr`
- `x binop= expr`
- `x++, ++x, x--, --x`

User defined reductions are an option in newer versions of OpenMP

Example: Reduction

```
#pragma omp parallel for reduction(+: a)
for (int j=0; j<4; j++)
{
  a = omp_get_thread_num();
}
printf("Final Value of a=%d\n", a);
```

Final value of a:

0+1+2+3 = 6

Data Synchronization

As with Pthreads, we need options to synchronize data accesses
- Communcation is through shared variables
- Synchronization has to be separate

OpenMP offers a wide range of pragma based options
- Barriers
- Master regions
- Single regions
- Critical sections
- Atomic statements
- Ordering construct

Additional runtime routines for locking

Master Region

```
#pragma omp master
        block
```

A master region enforces that only the master executes the code block
- Other threads skip the region
- No synchronization at the beginning of region

Possible uses
- Printing to screen
- Keyboard entries
- File I/O

Single Region

```
#pragma omp single [parameter]
        block
```

A single region enforces that only a (arbitrary) single thread executes the code block

- Other threads skip the region
- Implicit barrier synchronization at the end of region (unless **`nowait`** is specified)

Possible uses

- Initialization of data structures

Critical Section

```
#pragma omp critical [(Name)]
        block
```

Mutual exclusion
- A critical section is a block of code
- Can only be executed by only one thread at a time.
- Compare to Pthreads locks

Critical section name identifies the specific critical section
- A thread waits at the beginning of a critical section until its available
- All unnamed critical directives map to the same name

Keep in mind
- Critical section names are global entities of the program
- If a name conflicts with any other entity, program behavior is unspecified
- Avoid long critical sections for performance reasons

Atomic Statements

```
#pragma ATOMIC
      expression-stmt
```

The ATOMIC directive ensures that a specific memory location is updated atomically

Has to have the following form:

- **x binop= expr**
- **x++ or ++x**
- **x-- or --x**

- where x is an lvalue expression with scalar type
- and expr does not reference the object designated by x

Equivalent to using critical section to protect the update

Useful for simple/fast updates to shared data structures

- Avoids locking
- Often implemented directly by native instructions

Simple Runtime Locks

In addition to pragma based options, OpenMP also offers runtime locks

- Same concept as Pthread mutex
- Locks can be held by only one thread at a time.
- A lock is represented by a lock variable of type omp_lock_t.
- The thread that obtained a simple lock cannot set it again.

Operations

- omp_init_lock(&lockvar) initialize a lock
- omp_destroy_lock(&lockvar) destroy a lock
- omp_set_lock(&lockvar) set lock
- omp_unset_lock(&lockvar) free lock
- logicalvar = omp_test_lock(&lockvar)
  check lock and possibly set lock
  returns true if lock was set by the executing thread.

Example: Simple Lock

```
#include <omp.h>
int id;
omp_lock_t lock;

omp_init_lock(lock);
#pragma omp parallel shared(lock) private(id)
{
  id = omp_get_thread_num();
  omp_set_lock(&lock); //Only a single thread writes
    printf("My Thread num is: %d", id);
  omp_unset_lock(&lock);

  while (!omp_test_lock(&lock))
    other_work(id);          //Lock not obtained
  real_work(id);             //Lock obtained
  omp_unset_lock(&lock);     //Lock freed
}
omp_destroy_lock(&lock);
```

locked

locked

Nestable Locks

Similar to simple locks

But, nestable locks can be set multiple times by a single thread.

- Each set operation increments a lock counter
- Each unset operation decrements the lock counter

If the lock counter is 0 after an unset operation, lock can be set by another thread

Separate routines for nestable locks

Look them up ☺

Ordered Construct

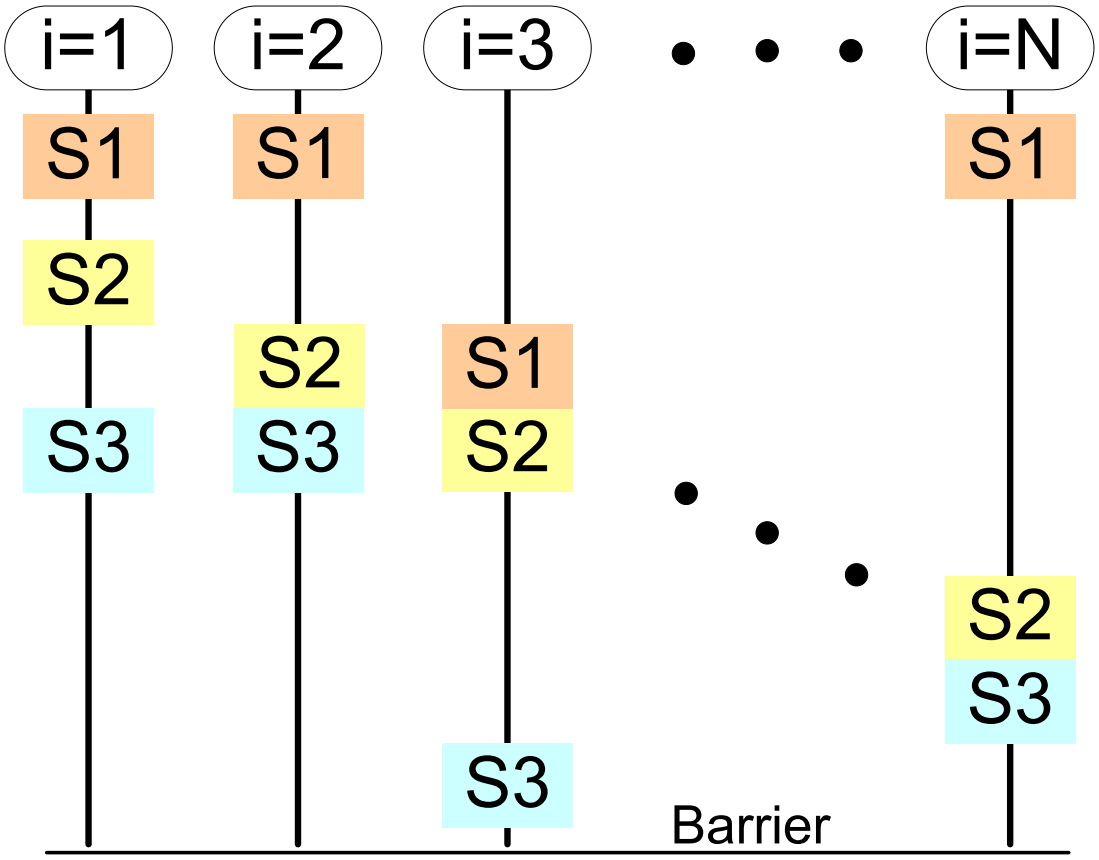
```
#pragma omp for ordered
    for (...)
     { ...
       #pragma omp ordered
       { ... }
       ...
     }
```

Construct must be within the dynamic extent of an **`omp for`** construct with an ordered clause.

Ordered constructs are executed strictly in the order in which they would be executed in a sequential execution of the loop.

Example: ordered clause

```
#pragma omp for ordered
for (...)
 { S1
  #pragma omp ordered
  { S2 }
  S3
 }
```



Important Runtime Routines

Runtime offers additional routines to control behavior

Determine the number of threads for parallel regions
- `omp_set_num_threads(count)`

Query the maximum number of threads for team creation
- `numthreads = omp_get_max_threads()`

Query number of threads in the current team
- `numthreads = omp_get_num_threads()`

Query own thread number (0..n-1)
- `iam = omp_get_thread_num()`

Query the number of processors
- `numprocs = omp_get_num_procs()`

Relevant Environment Variables (ICVs)

`OMP_NUM_THREADS=4`

- Number of threads in a team of a parallel region

`OMP_SCHEDULE="dynamic"  OMP_SCHEDULE="GUIDED,4"`

- Selects scheduling strategy to be applied at runtime
- Schedule clause in the code takes precedence

`OMP_DYNAMIC=TRUE`

- Allow runtime system to determine the number of threads

`OMP_NESTED=TRUE`

- Allow nesting of parallel regions
- If supported by the runtime

Runtime routines to query and set ICVs

Summary: OpenMP Basics

OpenMP was created to standardize the programming of shared memory systems

- First standard in 1997, currently at OpenMP 4.5
- Goals were easy of use, simplicity and portability

Key concepts

- Parallel regions
- Worksharing through parallel for loops
- Additional clauses to control distribution, synchronization, …
- Reduction operations
- Options to control data visibility/sharing

Programmer responsibility

- Ensure no loop dependencies exist
- Ensure the right variables or private or shared
- Ensure the necessary synchronization is added