Introduction to OpenMP

University of Stuttgart
High Performance Computing Center Stuttgart (HLRS)
www.hlrs.de

Matthias Müller, Rainer Keller
Isabel Loebich, Rolf Rabenseifner

[mueller | keller | loebich | rabenseifner] @hlrs.de

Version 10, 24.2.2004

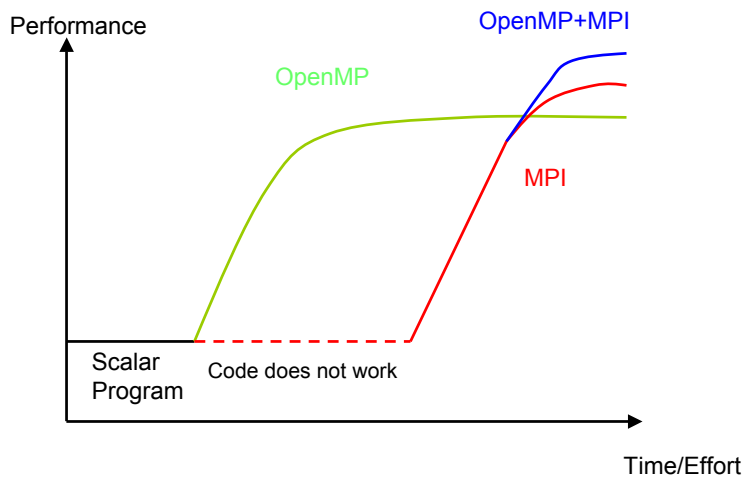
Outline

- Introduction into OpenMP
- Execution Model
  - **Parallel regions: team of threads**
  - **Syntax**
  - **Data environment (part 1)**
  - **Environment variables**
  - **Runtime library routines**
- Work-sharing directives
  - **Which thread executes which statement or operation?**
  - **Synchronization constructs, e.g., critical sections**
  - **Nesting and Binding**
  - **Exercise: Pi**
- Data environment and combined constructs
  - **Private and shared variables**
  - **Combined parallel work-sharing directives**
  - **Exercise: heat**
- Summary of OpenMP API
- OpenMP Pitfalls

OpenMP Overview: What is OpenMP?

- OpenMP is a standard programming model for shared memory parallel programming
- Portable across all shared-memory architectures
- It allows incremental parallelization
- Compiler based extensions to existing programming languages
  - mainly by directives
  - a few library routines
- Fortran and C/C++ binding
- OpenMP is a standard

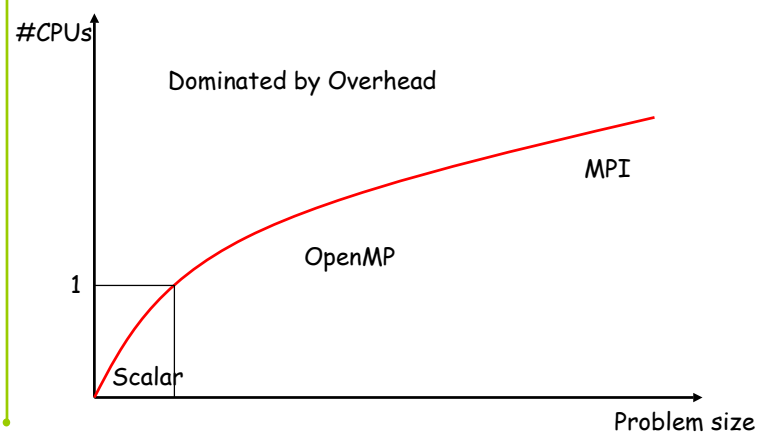
Motivation: Why should I use OpenMP?



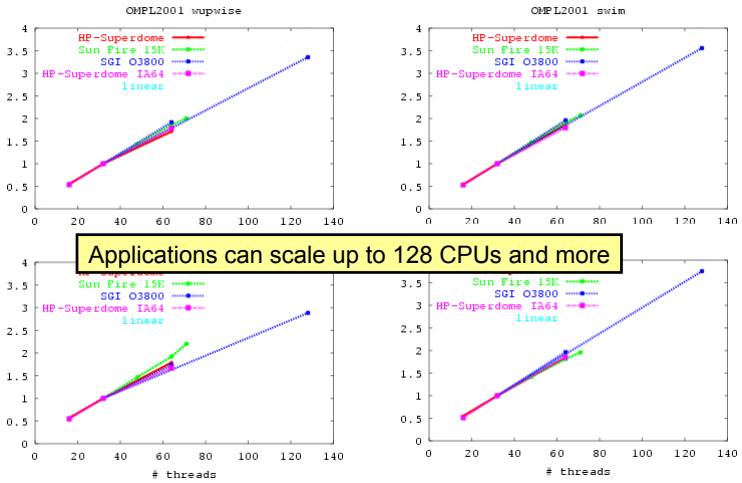
Further Motivation to use OpenMP

- OpenMP is the easiest approach to multi-threaded programming
- Multi-threading is needed to exploit modern hardware platforms:
  - Intel CPUs support Hyperthreading
  - AMD Opterons are building blocks for cheap SMP machines
  - A growing number of CPUs are multi-core CPUs
    - **IBM Power CPU**
    - **SUN UltraSPARC IV**
    - **HP PA8800**

Where should I use OpenMP?

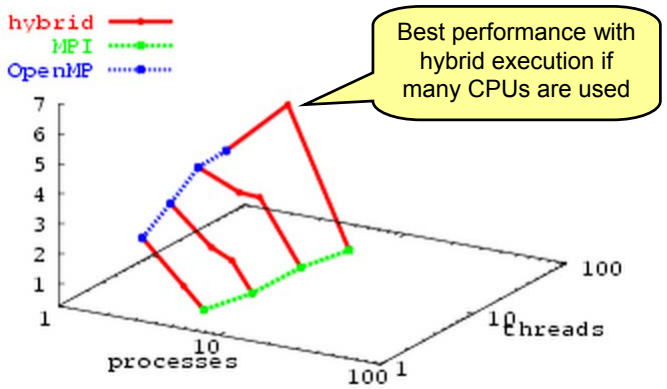


On how many CPUs can I use OpenMP?



Applications can scale up to 128 CPUs and more

Hybrid Execution (OpenMP+MPI) can improve the performance



Best performance with hybrid execution if many CPUs are used

Simple OpenMP Program

- Most OpenMP constructs are compiler directives or pragmas
- The focus of OpenMP is to parallelize loops
- OpenMP offers an incremental approach to parallelism

Serial Program:

```
void main()
{
   double Res[1000];

   for(int i=0;i<1000;i++) {
         do_huge_comp(Res[i]);
   }
}
```

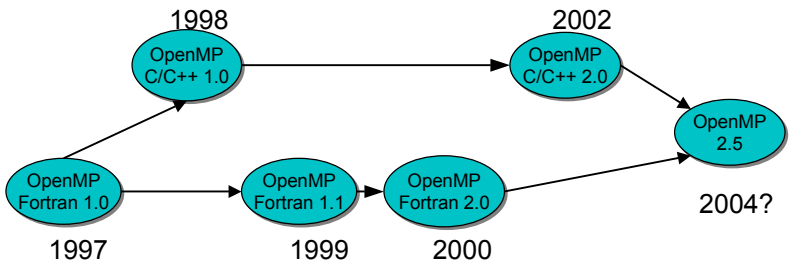
Parallel Program:

```
void main()
{
   double Res[1000];
#pragma omp parallel for
   for(int i=0;i<1000;i++) {
         do_huge_comp(Res[i]);
   }
}
```


Who owns OpenMP? - OpenMP Architecture Review Board

- ASCI Program of the US DOE
- Compaq Computer Corporation
- EPCC (Edinburgh Parallel Computing Center)
- Fujitsu
- Hewlett-Packard Company
- Intel Corporation
- International Business Machines (IBM)
- Silicon Graphics, Inc.
- Sun Microsystems, Inc
- cOMPunity
- NEC

OpenMP Release History



OpenMP Availability

	Fortran	C	C++
HP	yes	yes	yes
IBM	yes	yes	yes
SGI	yes	yes	yes
SUN	yes	yes	yes
Cray	yes	yes	yes
Hitachi SR8000	yes	yes	In prep
NEC SX	yes	yes	yes
Intel IA32	yes	yes	yes
Intel IA64	yes	yes	yes
AMD X86-64	yes	yes	yes

- Fortran indicates Fortran 90 and OpenMP 1.1
- C/C++ indicates OpenMP 1.0
- OpenMP is available on all platforms for all language bindings

OpenMP Information

- OpenMP Homepage:
  http://www.openmp.org/
- OpenMP user group
  http://www.compunity.org
- OpenMP at HLRS:
  http://www.hlrs.de/organization/tsc/services/models/openmp/

- R.Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon:
  Parallel programming in OpenMP.
  Academic Press, San Diego, USA, 2000, ISBN 1-55860-671-8
- R. Eigenmann, Michael J. Voss (Eds):
  OpenMP Shared Memory Parallel Programming.
  Springer LNCS 2104, Berlin, 2001, ISBN 3-540-42346-X

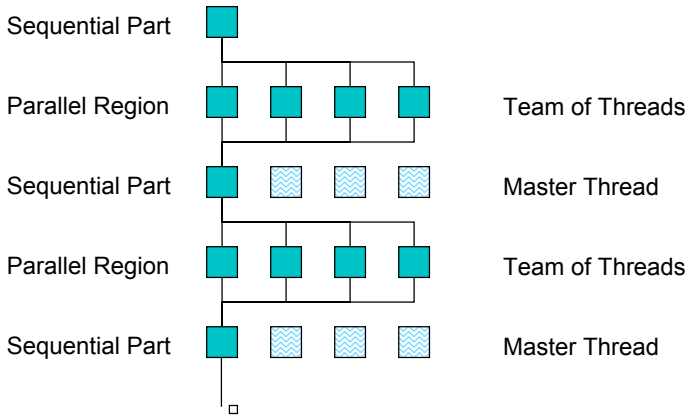
Outline — Programming and Execution Model

- Standardization Body
- OpenMP Application Program Interface (API)
- **Programming and Execution Model**
  - **Parallel regions: team of threads**
  - **Syntax**
  - **Data environment (part 1)**
  - **Environment variables**
  - **Runtime library routines**
- Work-sharing directives
  - Which thread executes which statement or operation?
  - Synchronization constructs, e.g., critical sections
  - Nesting and Binding
  - Exercise: Pi
- Data environment and combined constructs
  - Private and shared variables
  - Combined parallel work-sharing directives
  - Exercise: Heat
- Summary of OpenMP API
- OpenMP Pitfalls

OpenMP Programming Model

- OpenMP is a shared memory model.
- Workload is distributed between threads
  - Variables can be
    - **shared among all threads**
    - **duplicated for each thread**
  - Threads communicate by sharing variables.
- Unintended sharing of data can lead to race conditions:
  - race condition: when the program's outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization to protect data conflicts.

OpenMP Execution Model



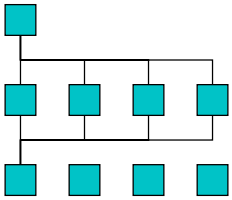
OpenMP Execution Model Description

- Fork-join model of parallel execution
- Begin execution as a single process (master thread)
- Start of a parallel construct:
  Master thread creates team of threads
- Completion of a parallel construct:
  Threads in the team synchronize:
  implicit barrier
- Only master thread continues execution

OpenMP Parallel Region Construct

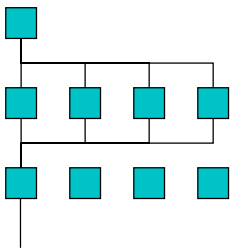
Fortran:

```
!$OMP PARALLEL
    block
!$OMP END PARALLEL
```



C / C++:

```
#pragma omp parallel
    structured block
/* omp end parallel */
```



OpenMP Parallel Region Construct Syntax

- Block of code to be executed by multiple threads in parallel. Each thread executes the **same code redundantly**!
- Fortran:
  `!$OMP PARALLEL` [*clause* [[,] *clause*] ...]
  block
  `!$OMP END PARALLEL`
  - `parallel/end parallel` directive pair must appear in the same routine
- C/C++:
  `#pragma omp parallel` [*clause* [*clause*] ...] *new-line*
  structured-block
- *clause* can be one of the following:
  - `private(`*list*`)`
  - `shared(`*list*`)`
  - ...

OpenMP Directive Format: Fortran

- Treated as Fortran comments
- Format:
  sentinel directive_name [*clause* [[,] *clause*] ...]
- Directive sentinels (starting at **column 1**):
  - Fixed source form: **`!$OMP`** | `C$OMP` | `*$OMP`
  - Free source form: **`!$OMP`**
- not case sensitive
- Conditional compilation
  - Fixed source form: **`!$␣`** | `C$␣` | `*$␣`
  - Free source form: **`!$␣`**
  - `#ifdef _OPENMP`
    block
    `#endif`
    [in `my_fixed_form.F` or `my_free_form.F90`]
  - Example:
    `!$ write(*,*) OMP_GET_NUM_PROCS(),' avail. processors'`

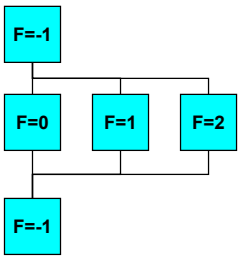
OpenMP Directive Format: C/C++

- `#pragma` directives
- Format:
  `#pragma omp` *directive_name* [*clause* [*clause*] ...] *new-line*
- Conditional compilation
  `#ifdef _OPENMP`
  block,
  e.g., `printf("%d avail.processors\n",omp_get_num_procs());`
  `#endif`
- case sensitive
- Include file for library routines:

```
#ifdef _OPENMP
#include <omp.h>
#endif
```


OpenMP Data Scope Clauses

- `private` (*list*)
  Declares the variables in *list* to be private to each thread in a team
- `shared` (*list*)
  Makes variables that appear in *list* shared among all the threads in a team



- If not specified: default `shared`, but
  - stack (local) variables in called sub-programs are PRIVATE
  - Automatic variables within a block are PRIVATE
  - Loop control variable of parallel OMP
    - DO (Fortran)
    - for (C)

    is PRIVATE

[see later: Data Model]

OpenMP Environment Variables

- `OMP_NUM_THREADS`
  - sets the number of threads to use during execution
  - when dynamic adjustment of the number of threads is enabled, the value of this environment variable is the maximum number of threads to use
  - `setenv OMP_NUM_THREADS 16` **[csh, tcsh]**
  - `export OMP_NUM_THREADS=16` **[sh, ksh, bash]**
- `OMP_SCHEDULE`
  - applies only to `do/for` and `parallel do/for` directives that have the schedule type `RUNTIME`
  - sets schedule type and chunk size for all such loops
  - `setenv OMP_SCHEDULE "GUIDED,4"` **[csh, tcsh]**
  - `export OMP_SCHEDULE="GUIDED,4"` **[sh, ksh, bash]** □

OpenMP Runtime Library (1)

- Query functions
- Runtime functions
  - Run mode
  - Nested parallelism
- Lock functions
- C/C++: add `#include <omp.h>`
- Fortran: add all necessary OMP routine declarations, e.g.,

```
!$   INTEGER omp_get_thread_num
```

OpenMP Runtime Library (2)

- `omp_get_num_threads` Function
  Returns the number of threads currently in the team executing the parallel region from which it is called
  - Fortran:
    ```
    integer function omp_get_num_threads()
    ```
  - C/C++:
    ```
    int omp_get_num_threads(void);
    ```
- `omp_get_thread_num` Function
  Returns the thread number, within the team, that lies between `0` and `omp_get_num_threads()-1`, inclusive. The master thread of the team is thread `0`
  - Fortran:
    ```
    integer function omp_get_thread_num()
    ```
  - C/C++:
    ```
    int omp_get_thread_num(void);
    ```


OpenMP Runtime Library (3): Wall clock timers OpenMP 2.0

- Portable wall clock timers similar to MPI_WTIME
- `DOUBLE PRECISION FUNCTION OMP_GET_WTIME()`
  - provides elapsed time

    ```
    START=OMP_GET_WTIME()
    ! Work to be measured
    END = OMP_GET_WTIME()
    PRINT *, ´Work took ´, END-START, ´ seconds´
    ```

  - provides “per-thread time”, i.e. needs not be globally consistent
- `DOUBLE PRECISION FUNCTION OMP_GET_WTICK()`
  - returns the number of seconds between two successive clock ticks

Outline — Work-sharing directives

- Standardization Body
- OpenMP Application Program Interface (API)
- Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Data environment (part 1)
  - Environment variables
  - Runtime library routines
- **Work-sharing directives**
  - **Which thread executes which statement or operation?**
  - **Synchronization constructs, e.g., critical sections**
  - **Nesting and Binding**
  - **Exercise: pi**
- Data environment and combined constructs
  - Private and shared variables
  - Combined parallel work-sharing directives
  - Exercise: heat
- Summary of OpenMP API
- OpenMP Pitfalls

Work-sharing and Synchronization

- Which thread executes which statement or operation?
- and when?
  - Work-sharing constructs
  - Master and synchronization constructs

- **i.e., organization of the parallel <u>work</u>!!!**

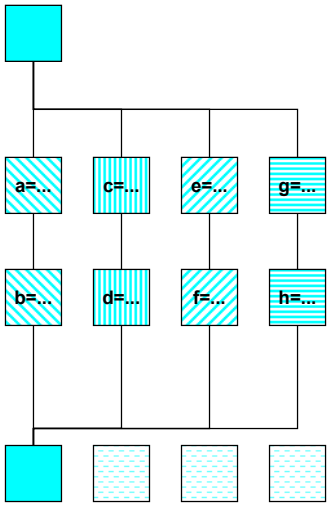
OpenMP Work-sharing Constructs

- Divide the execution of the enclosed code region among the members of the team
- Must be enclosed dynamically within a parallel region
- They do not launch new threads
- No implied barrier on entry
- `sections` directive
- `do` directive (Fortran)
- `for` directive (C/C++)

OpenMP `sections` Directives – C/C++

C / C++:

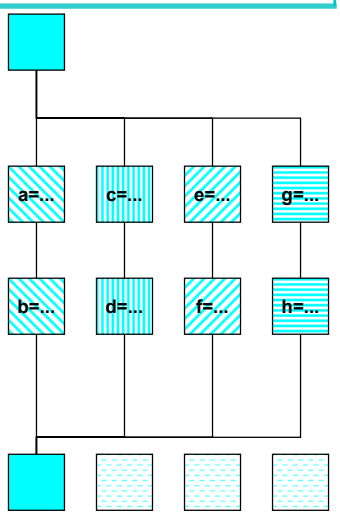
```
#pragma omp parallel
{
#pragma omp sections
  {{  a=...;
      b=...; }
#pragma omp section
    {  c=...;
       d=...; }
#pragma omp section
    {  e=...;
       f=...; }
#pragma omp section
    {  g=...;
       h=...; }
  } /*omp end sections*/
} /*omp end parallel*/
```



OpenMP sections Directives - Fortran

Fortran:

```
!$OMP PARALLEL
!$OMP SECTIONS
      a=...
      b=...
!$OMP SECTION
      c=...
      d=...
!$OMP SECTION
      e=...
      f=...
!$OMP SECTION
      g=...
      h=...
!$OMP END SECTIONS
!$OMP END PARALLEL
```



OpenMP sections Directives - Syntax

- Several *blocks* are executed in parallel
- Fortran:

```
 !$OMP SECTIONS  [ clause [ [ , ] clause ] ... ]
[!$OMP SECTION ]
   block1
[!$OMP SECTION
   block2 ]
  ...
 !$OMP END SECTIONS [nowait]
```

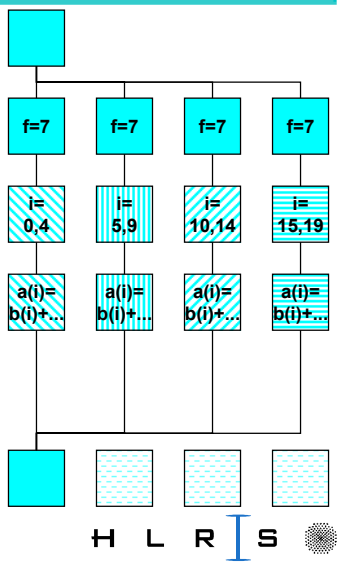
- C/C++:

```
#pragma omp sections  [ clause [ clause ] ... ] new-line
  {
    [#pragma omp section new-line ]
       structured-block1
    [#pragma omp section new-line
       structured-block2 ]
     ...
  }
```

OpenMP do/for Directives – C/C++

C / C++:

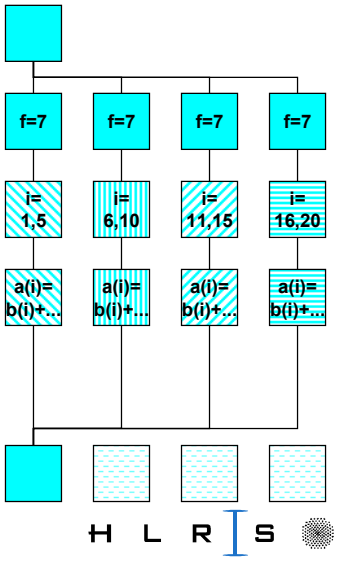
```
#pragma omp parallel private(f)
{
    f=7;

#pragma omp for
    for (i=0; i<20; i++)
        a[i] = b[i] + f * (i+1);

} /* omp end parallel */
```



OpenMP do/for Directives - Fortran

Fortran:

```
!$OMP PARALLEL private(f)
    f=7
!$OMP DO
    do i=1,20
        a(i) = b(i) + f * i
    end do
!$OMP END DO
!$OMP END PARALLEL
```



OpenMP do/for Directives - Syntax

- Immediately following loop executed in parallel
- Fortran:
  `!$OMP do` [*clause* [[,] *clause*] ...]
  do_loop
  [`!$OMP end do` [`nowait`]]
- If used, the `end do` directive must appear immediately after the end of the loop
- C/C++:
  `#pragma omp for` [*clause* [*clause*] ...] *new-line*
  for-loop
- The corresponding `for` loop must have *canonical shape*

OpenMP do/for Directives - Details

- *clause* can be one of the following:
  - `private(`*list*`)` **[see later: Data Model]**
  - `reduction(`*operator*`:`*list*`)` **[see later: Data Model]**
  - `schedule(` *type* [, *chunk*])
  - `nowait` (C/C++: on `#pragma omp for`)
    (Fortran: on `$!OMP END DO`)
  - ...
- Implicit barrier at the end of `do`/`for` unless `nowait` is specified
- If `nowait` is specified, threads do not synchronize at the end of the parallel loop
- `schedule` clause specifies how iterations of the loop are divided among the threads of the team.
  - Default is implementation dependent

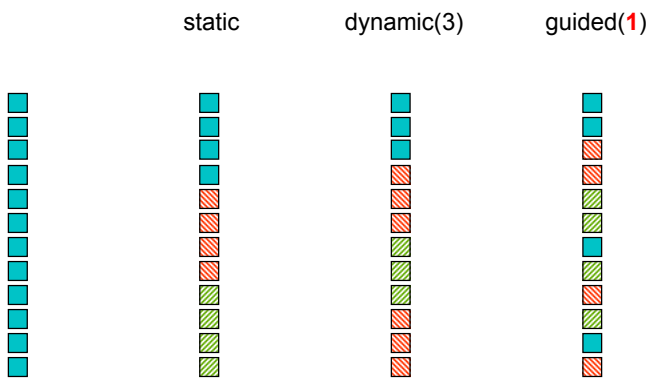
OpenMP `schedule` Clause

Within `schedule(` *type* [, *chunk*]) *type* can be one of the following:

- `static`: Iterations are divided into pieces of a size specified by *chunk*. The pieces are statically assigned to threads in the team in a round-robin fashion in the order of the thread number.
  Default chunk size: one contiguous piece for each thread.
- `dynamic`: Iterations are broken into pieces of a size specified by *chunk*. As each thread finishes a piece of the iteration space, it dynamically obtains the next set of iterations. Default chunk size: 1.
- `guided`: The chunk size is reduced in an exponentially decreasing manner with each dispatched piece of the iteration space.
  chunk specifies the smallest piece (except possibly the last).
  Default chunk size: 1. Initial chunk size is implementation dependent.
- `runtime`: The decision regarding scheduling is deferred until run time. The schedule type and chunk size can be chosen at run time by setting the `OMP_SCHEDULE` environment variable.

Default schedule: implementation dependent.

Loop scheduling



New Feature: WORKSHARE directive — OpenMP 2.0 Fortran

- WORKSHARE directive allows parallelization of array expressions and FORALL statements
- Usage:

```
!$OMP WORKSHARE
A=B
! Rest of block
!$OMP END WORKSHARE
```

- Semantics:
  - Work inside block is divided into separate units of work.
  - Each unit of work is executed only once.
  - The units of work are assigned to threads in any manner.
  - The compiler must ensure sequential semantics.
  - Similar to PARALLEL DO without explicit loops.

Outline — Synchronization constructs

- Standardization Body
- OpenMP Application Program Interface (API)
- Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Data environment (part 1)
  - Environment variables
  - Runtime library routines
  - Exercise and Compilation
- **Work-sharing directives**
  - **Which thread executes which statement or operation?**
  - **Synchronization constructs, e.g., critical sections**
  - **Nesting and Binding**
  - **Exercise: pi**
- Data environment and combined constructs
  - Private and shared variables
  - Combined parallel work-sharing directives
  - Exercise: heat
- Summary of OpenMP API
- OpenMP Pitfalls

OpenMP Synchronization

- Implicit Barrier
  - beginning and end of `parallel` constructs
  - end of all other control constructs
  - implicit synchronization can be removed with **`nowait`** clause
- Explicit
  - `critical`
  - ...

OpenMP `critical` Directive

- Enclosed code
  - executed by all threads, but
  - **restricted to only one thread at a time**
- Fortran:

```
!$OMP CRITICAL [ ( name ) ]
    block
!$OMP END CRITICAL [ ( name ) ]
```

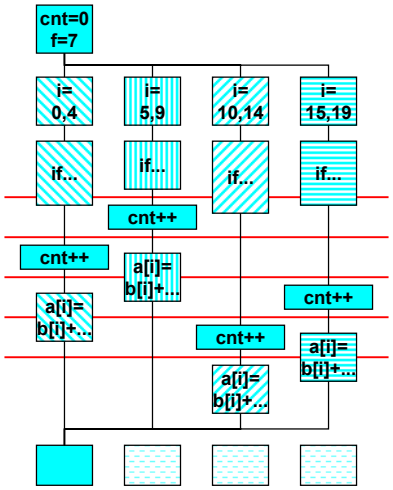
- C/C++:

```
#pragma omp critical [ ( name ) ] new-line
    structured-block
```

- A thread waits at the beginning of a critical region until no other thread in the team is executing a critical region with the same name. All unnamed `critical` directives map to the same unspecified name.

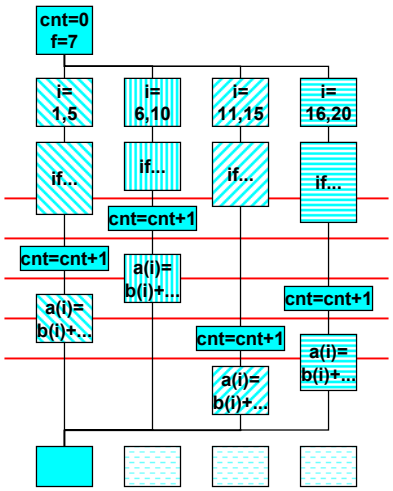
OpenMP `critical` — an example (C/C++)

```
C / C++:   cnt = 0;
           f=7;
      #pragma omp parallel
      {
      #pragma omp for
            for (i=0; i<20; i++) {
              if (b[i] == 0) {
                  #pragma omp critical
                   cnt ++;
              } /* endif */
              a[i] = b[i] + f * (i+1);
            } /* end for */
      } /*omp end parallel */
```



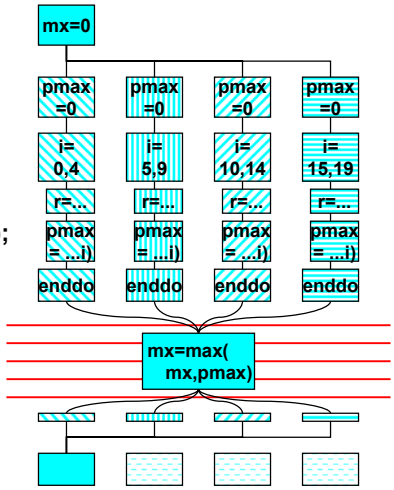
OpenMP `critical` — an example (Fortran)

```
Fortran:     cnt = 0
             f=7
      !$OMP PARALLEL
      !$OMP DO
             do i=1,20
                if (b(i).eq.0) then
      !$OMP CRITICAL
                   cnt = cnt+1
      !$OMP END CRITICAL
                endif
                a(i) = b(i) + f * i
             end do
      !$OMP END DO
      !$OMP END PARALLEL
```



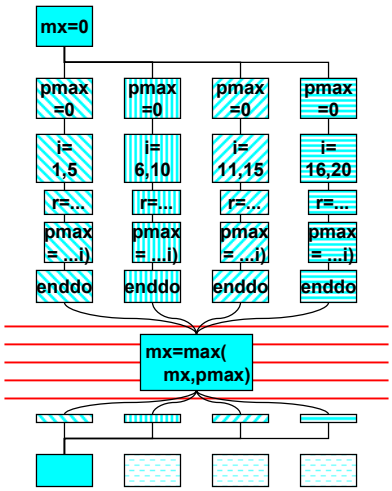
OpenMP `critical` — another example (C/C++)

```
      mx = 0;
#pragma omp parallel private(pmax)
{     pmax = 0;
      #pragma omp for private(r)
      for (i=0; i<20; i++)
      {   r = work(i);
          pmax = (r>pmax ? r : pmax);
      } /*end for*/
      /*omp end for*/
      #pragma omp critical
      mx= (pmax>mx ? pmax : mx);
      /*omp end critical*/
} /*omp end parallel*/
```



OpenMP `critical` — another example (Fortran)

```
      mx = 0
!$OMP PARALLEL private(pmax)
      pmax = 0
!$OMP DO private(r)
      do i=1,20
         r = work(i)
         pmax = max(pmax,r)
      end do
!$OMP END DO
!$OMP CRITICAL
      mx = max(mx,pmax)
!$OMP END CRITICAL
!$OMP END PARALLEL
```



Outline — Nesting and Binding

- Standardization Body
- OpenMP Application Program Interface (API)
- Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Data environment (part 1)
  - Environment variables
  - Runtime library routines
  - Exercise and Compilation
- **Work-sharing directives**
  - **Which thread executes which statement or operation?**
  - **Synchronization constructs, e.g., critical sections**
  - **Nesting and Binding**
  - **Exercise 2: pi**
- Data environment and combined constructs
  - Private and shared variables
  - Combined parallel work-sharing directives
- Summary of OpenMP API
- OpenMP Pitfalls

OpenMP Vocabulary

- **Static extent** of the parallel construct:
  statements enclosed lexically within the construct
- **Dynamic extent** of the parallel construct:
  further includes the routines called from within the construct
- **Orphaned Directives**:
  Do not appear in the lexical extent of the parallel construct but lie in the dynamic extent
  - Parallel constructs at the top level of the program call tree
  - Directives in any of the called routines

OpenMP Vocabulary

```
      program a                    subroutine b
!$OMP PARALLEL                 !$OMP DO
      call b                       do i=1,n
      call c                       ...
!$OMP END PARALLEL                 enddo
      call d                   !$OMP END DO
      stop                         return
      end                          end
                                   subroutine c
                                   return
                                   end
```

Static Extent | Dynamic Extent | Orphaned Directives

OpenMP Control Structures — Summary

- Parallel region construct
  - `parallel`
- Work-sharing constructs
  - `sections`
  - `do` (Fortran)
  - `for` (C/C++)
- Combined parallel work-sharing constructs **[see later]**
  - `parallel do` (Fortran)
  - `parallel for` (C/C++)
- Synchronization constructs
  - `critical`

Outline — Exercise: pi

- Standardization Body
- OpenMP Application Program Interface (API)
- Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Data environment (part 1)
  - Environment variables
  - Runtime library routines
  - Exercise and Compilation
- **Work-sharing directives**
  - **Which thread executes which statement or operation?**
  - **Synchronization constructs, e.g., critical sections**
  - **Nesting and Binding**
  - **Exercise: pi**
- Data environment and combined constructs
  - Private and shared variables
  - Combined parallel work-sharing directives
  - Exercise: heat
- Summary of OpenMP API
- OpenMP Pitfalls

OpenMP Exercise 2: pi Program (1)

- Goal: usage of
  - work-sharing constructs: `do`/`for`
  - `critical` directive
- Working directory: `~/OpenMP/#NR/pi/`
  `#NR` = number of your PC, e.g., 07
- Serial programs:
  - Fortran 77: `pi.f` and `scdiff.f90`
  - Fortran 90: `pi.f90` and `scdiff.f90`
  - C: `pi.c`

OpenMP Exercise 2: pi Program (2)

- compile serial program `pi.[f|f90|c]` and run
- add parallel region and `do/for` directive in `pi.[f|f90|c]`and compile
- set environment variable `OMP_NUM_THREADS` to 2 and run
  - value of pi? (should be wrong!)
- run again
  - value of pi? (...wrong and unpredictable)
- set environment variable `OMP_NUM_THREADS` to 4 and run
  - value of pi? (...and stays wrong)
- run again
  - value of pi? (...but where s the race-condition?)

OpenMP Exercise 2: pi Program (3)

- add `private(x)` clause in `pi.[f|f90|c]`and compile
- set environment variable `OMP_NUM_THREADS` to 2 and run
  - value of pi? (should be still incorrect ...)
- run again
  - value of pi?
- set environment variable `OMP_NUM_THREADS` to 4 and run
  - value of pi?
- run again
  - value of pi? (... and where is the second race-condition?)

OpenMP Exercise 2: pi Program (4)

- add `critical` directive in `pi.[f|f90|c]` around the sum-statement and **don't** compile
- reduce the number of iterations to 1,000,000 and compile
- set environment variable `OMP_NUM_THREADS` to 2 and run
  - value of pi? (should be now correct!, but huge CPU time!)
- run again
  - value of pi? (but not reproducble iin the last bit!)
- set environment variable `OMP_NUM_THREADS` to 4 and run
  - value of pi? execution time? (Oh, takes it longer?)
- run again
  - value of pi? execution time?
  - How can you optimize your code?

OpenMP Exercise 2: pi Program (5)

- move `critical` directive in `pi.[f|f90|c]` outside loop, restore old iteration length (10,000,000) and compile
- set environment variable `OMP_NUM_THREADS` to 2 and run
  - value of pi?
- run again
  - value of pi?
- set environment variable `OMP_NUM_THREADS` to 4 and run
  - value of pi? execution time? (correct pi, half execution time)
- run again
  - value of pi? execution time?

OpenMP Exercise 2: pi Program - Solution

`Location: ~/OpenMP/Aufgabe/solution/pi`

- `pi.[f|f90|c]`: original program
- `pi1.[f|f90|c]`: incorrect (no private, no synchronous global access) !!!
- `pi2.[f|f90|c]`: incorrect (still no synchronous global access to `sum`) !!!
- `pic.[f|f90|c]`: solution with `critical` directive, but extremely slow!
- `pic2.[f|f90|c]`: solution with `critical` directive outside loop ▫

Outline — Data environment and combined constructs

- Standardization Body
- OpenMP Application Program Interface (API)
- Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Data environment (part 1)
  - Environment variables
  - Runtime library routines
- Work-sharing directives
  - Which thread executes which statement or operation?
  - Synchronization constructs, e.g., critical sections
  - Nesting and Binding
- **Data environment and combined constructs**
  - **Private and shared variables**
  - **Reduction clause**
  - **Combined parallel work-sharing directives**
  - **Exercise 3: pi**
- Summary of OpenMP API
- OpenMP Pitfalls

OpenMP Data Scope Clauses

- `private ( list )`
  Declares the variables in *list* to be private to each thread in a team
- `shared ( list )`
  Makes variables that appear in *list* shared among all the threads in a team
- If not specified: default `shared`, but
  - stack (local) variables in called subroutines are PRIVATE
  - Automatic variables within a block are PRIVATE
  - Loop control variable of parallel OMP
    - DO (Fortran)
    - FOR (C)

    is PRIVATE
- Recommendation: Avoid private variables, use variables local to a block instead (only possible for C/C++)

Private Clause

- `Private (variable)` creates a local copy of variable for each thread
  - value is unitialized
  - private copy is not storage associated with the original

```
      program wrong
      JLAST = -777
!$OMP PARALLEL DO PRIVATE(JLAST)
      DO J=1,1000
           ...
           JLAST = J
      END DO
!$OMP END PARALLEL DO
      print *, JLAST
```

—> writes -777 !!!
or undefined value

- If initialization is necessary use `FIRSTPRIVATE( var )`
- If value is needed after loop use `LASTPRIVATE( var )`

  —> `var` is updated by the thread that computes
  - the sequentially last iteration (on `do` or `for` loops)
  - the last section

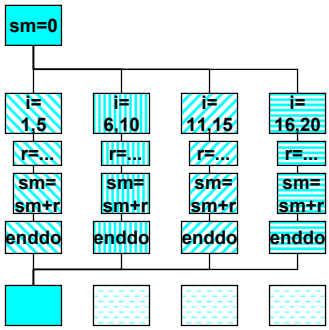
OpenMP `reduction` Clause

- `reduction` (*operator*:*list*)
- Performs a reduction on the variables that appear in *list*, with the operator *operator*
- *operator*: one of
  - Fortran:
    `+, *, -, .and., .or., .eqv., .neqv.` or
    `max, min, iand, ior,` or `ieor`
  - C/C++:
    `+, *, -, &, ^, |, &&,` or `||`
- Variables must be `shared` in the enclosing context
- With OpenMP 2.0 variables can be arrays (Fortran)
- At the end of the `reduction`, the shared variable is updated to reflect the result of combining the original value of the shared reduction variable with the final value of each of the private copies using the operator specified

OpenMP `reduction` — an example (Fortran)

Fortran:

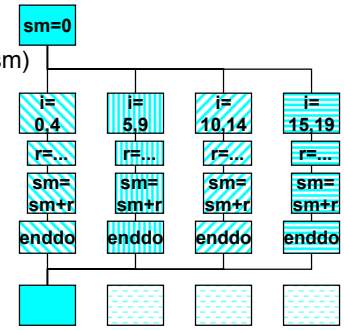
```
      sm = 0
!$OMP PARALLEL DO private(r),
                  reduction(+:sm)
      do i=1,20
         r = work(i)
         sm = sm + r
      end do
!$OMP END PARALLEL DO
```



OpenMP `reduction` — an example (C/C++)

C / C++:

```
      sm = 0;
#pragma omp parallel for reduction(+:sm)
      for( i=0; i<20; i++)
      {   double r;
          r = work(i);
          sm = sm + r ;
      } /*end for*/
/*omp end parallel for*/
```



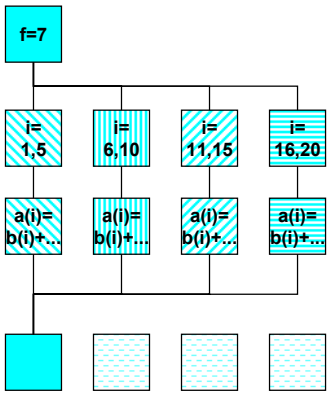
OpenMP Combined `parallel` do/`for` Directive

- Shortcut form for specifying a parallel region that contains a single `do/for` directive
- Fortran:
  `!$OMP PARALLEL DO` [*clause* [[,] *clause*] ...]
  do_loop
  [`!$OMP END PARALLEL DO`]
- C/C++:
  `#pragma omp parallel for` [*clause* [*clause*] ...] *new-line*
  for-loop
- This directive admits all the clauses of the `parallel` directive and the `do/for` directive except the `nowait` clause, with identical meanings and restrictions

OpenMP Combined `parallel` do/for -- example (Fortran)

Fortran:

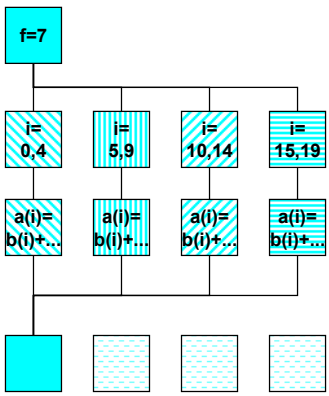
```
f=7
!$OMP PARALLEL DO
do i=1,20
   a(i) = b(i) + f * i
end do
!$OMP END PARALLEL DO
```



OpenMP Combined `parallel` do/for -- example (C/C++)

C / C++:

```
f=7;
#pragma omp parallel for
for (i=0; i<20; i++)
   a[i] = b[i] + f * (i+1);
```



OpenMP
Exercise Heat Conduction

Isabel Loebich
loebich@hlrs.de

OpenMP Exercise: Heat Conduction(1)

- solves the PDE for unsteady heat conduction $df/dt=\Delta f$
- uses an explicit scheme: forward-time, centered-space
- solves the equation over a unit square domain
- initial conditions: $f=0$ everywhere inside the square
- boundary conditions: $f=x$ on all edges
- number of grid points in each direction: 80

OpenMP Exercise: Heat Conduction (2)

- Goals:
  - parallelization of a real application
  - usage of different parallelization methods with respect to their effect on execution times
- Working directory: `~/OpenMP/#NR/heat/`
  `#NR` = number of your PC, e.g., 07
- Serial programs:
  - Fortran 77: `heat.f` and `scdiff.f90`
  - Fortran 90: `heat.f90` and `scdiff.f90`
  - C: `heat.c`
- Compiler calls:
  - Fortran 77/90: sxf90 -sx4/-sx5
  - C: `not yet available for OpenMP`

OpenMP Exercise: Heat Conduction (3)

3 versions provided:

- small version, for verifying purposes: `heat.[f|f90|c]`
  - 20 x 11 grid points, max 20000 iterations
  - prints array values before and after iteration loop
- big version: `heat-big.[f|f90|c]`
  - 80 x 80 grid points, max 20000 iterations
  - doesn't print array values
- version for use with compiler switch -O3: `heat-opt.[f|f90|c]`
  - 150 x 150 grid points, max 50000 iterations
  - doesn't print array values

OpenMP Exercise: Heat Conduction (4)

- parallelize small version using different methods and check results
  - `critical` directive
  - `reduction` clause
  - parallel region + work-sharing constructs
  - combined parallel work-sharing construct
- select one method and parallelize big version
- watch execution times
- use `SCHEDULE` clause with different values for `type` and `chunk` and watch effects on execution times
- optional: also parallelize version for use with compiler option -O3

OpenMP Exercise: Heat - Solution C/F77/F90 (1)

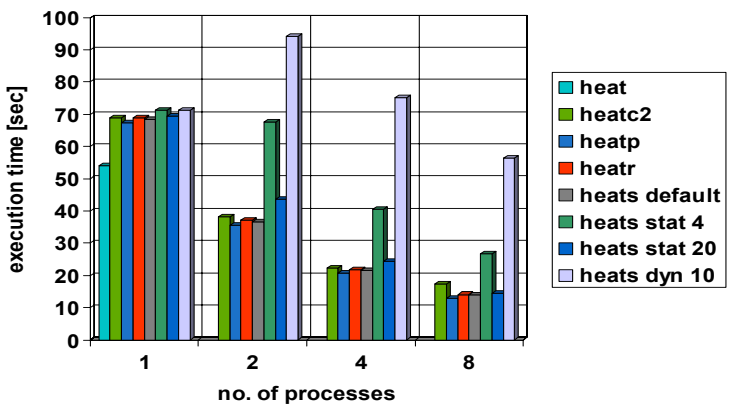
`Location: ~/OpenMP/Aufgabe/solution/hello`

- `heat.[f|f90|c]`: original program
- `heatc.[f|f90|c]`: solution with `critical` directive, one parallel region inside iteration loop
- `heatc2.[f|f90|c]`: solution with `critical` directive outside inner loop, one parallel region inside iteration loop
- `heatr.[f|f90|c]`: solution with `reduction` clause, one parallel region inside iteration loop
- `heatp.[f|f90|c]`: solution with `reduction` clause, two combined `parallel do`s inside iteration loop
- `heats.[f|f90|c]`: same as `heatr.[f|f90|c]`, `schedule(runtime)` clause added
- `heat?-big.[f|f90|c]` and `heat?-opt.[f|f90|c]`: corresponding versions with 80 x 80 grid and 150 x 150 grid, for use with -O3 compiler switch □

OpenMP Exercise: Heat - Solution C/F77/F90 (2)

- As we already learned `heatc` does not use parallelism very well
- Better: `heatc2`
- Overhead for creating two parallel regions in version `heatp` expected
- There should be no execution time differences between versions `heatr` and `heats` with default schedule
- Different execution times for different `schedule` schemes expected
- Version `big`: 14320 iterations
- Version `opt`: 44616 iterations

OpenMP Exercise: Heat - Execution Times F90/opt



OpenMP Exercise: Heat Conduction - Summary

- Overhead for parallel versions using 1 thread
- Be careful when using other than default scheduling strategies:
  - `dynamic` is generally expensive
  - `static`: overhead for small chunk sizes is clearly visible

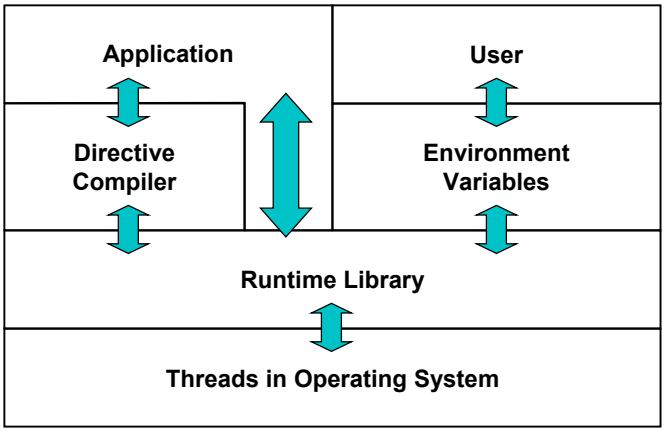
Outline — Summary of the OpenMP API

- Standardization Body
- OpenMP Application Program Interface (API)
- Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Data environment (part 1)
  - Environment variables
  - Runtime library routines
- Work-sharing directives
  - Which thread executes which statement or operation?
  - Synchronization constructs, e.g., critical sections
  - Nesting and Binding
- Data environment and combined constructs
  - Private and shared variables
  - Reduction clause
  - Combined parallel work-sharing directives
  - Exercise : heat
- **Summary of OpenMP API**
- OpenMP Pitfalls

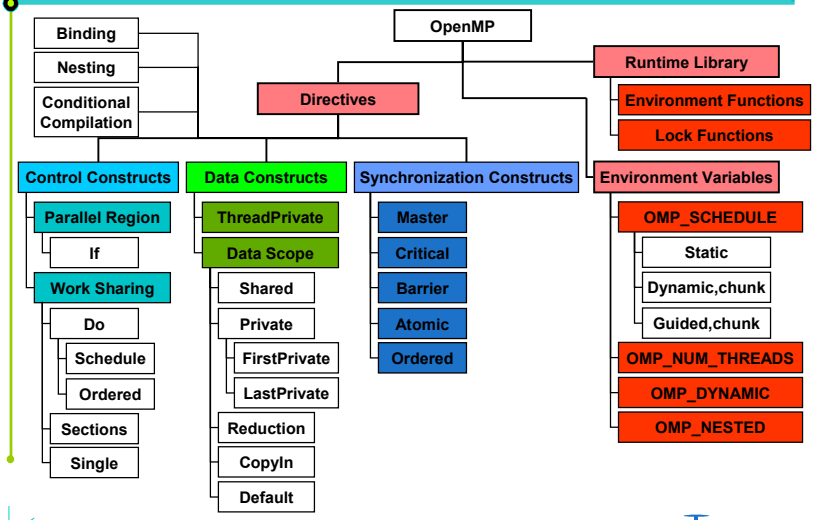
OpenMP Components

- Set of compiler directives
  - Control Constructs
    - **Parallel Regions**
    - **Work-sharing constructs**
  - Data environment
  - Synchronization
- Runtime library functions
- Environment variables

OpenMP Architecture



OpenMP Constructs



Outline — OpenMP Pitfalls

- Standardization Body
- OpenMP Application Program Interface (API)
- Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Data environment (part 1)
  - Environment variables
  - Runtime library routines
  - Exercise and Compilation
- Work-sharing directives
  - Which thread executes which statement or operation?
  - Synchronization constructs, e.g., critical sections
  - Nesting and Binding
- Data environment and combined constructs
  - Private and shared variables
  - Reduction clause
  - Combined parallel work-sharing directives
  - Exercise 3: pi
- Summary of OpenMP API
- **OpenMP Pitfalls**

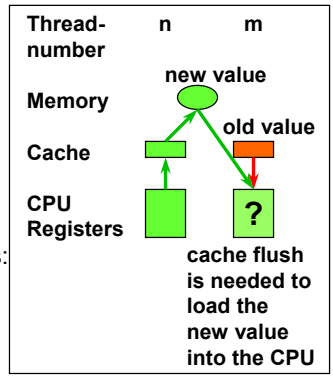
Implementation-defined behavior

See Appendix E of the OpenMP 2.0 standard

- The size of the first chunk in SCHEDULE(GUIDED)
- default schedule for SCHEDULE(RUNTIME)
- default schedule
- default number of threads
- default for dynamic thread adjustment
- number of threads used to execute nested parallel regions
- atomic directives might be replaced by critical sections
- behavior in case of thread exhaustion
- use of parameters other than OMP_*_KIND in generic interfaces
- allocation status of allocatable arrays that are not affected by COPYIN clause are undefined if dynamic thread mechanism is enabled

Implied flush directive

- A FLUSH directive identifies a sequence point at which a consistent view of the shared memory is guaranteed
- It is implied at the following constructs:
  - BARRIER
  - CRITICAL and END CRITICAL
  - END {DO, SECTIONS}
  - END {SINGLE, WORKSHARE}
  - ORDERED AND END ORDERED
  - PARALLEL and END PARALLEL with their combined variants
- It is NOT implied at the following constructs:
  - DO
  - MASTER and END MASTER
  - SECTIONS
  - SINGLE
  - WORKSHARE



Two types of SMP errors

- Race Conditions
  - Def.: *Two threads access the same shared variable* ***and*** *at least one thread modifies the variable* ***and*** *the sequence of the accesses is undefined, i.e. unsynchronized*
  - The outcome of a program depends on the detailed timing of the threads in the team.
  - This is often caused by unintended share of data
- Deadlock
  - Threads lock up waiting on a locked resource that will never become free.
    - **Avoid lock functions if possible**
    - **At least avoid nesting different locks**

Example for race condition (1)

```
!$OMP PARALLEL SECTIONS
       A = B + C
!$OMP SECTION
       B = A + C
!$OMP SECTION
       C = B + A
!$OMP END PARALLEL SECTIONS
```

- The result varies unpredictably based on specific order of execution for each section.
- Wrong answers produced without warning!

Example for race condition (2)

```
!$OMP PARALLEL SHARED (X), PRIVATE(TMP)
      ID = OMP_GET_THREAD_NUM()
!$OMP DO REDUCTION(+:X)
      DO 100 I=1,100
             TMP = WORK1(I)
             X = X + TMP
100  CONTINUE
!$OMP END DO NOWAIT
      Y(ID) = WORK2(X,ID)
!$OMP END PARALLEL
```

- The result varies unpredictably because the value of X isn't dependable until the barrier at the end of the do loop.
- Solution: Be careful when you use **NOWAIT**.

OpenMP programming recommendations

- Solution 1:
  Analyze your code to make sure every semantically permitted interleaving of the threads yields the correct results.
- Solution 2:
  Write SMP code that is portable and equivalent to the sequential form.
  - Use a safe subset of OpenMP.
  - Follow a set of "rules" for Sequential Equivalence.
  - Use tools like "assure".

Sequential Equivalence

- Two forms of sequential equivalence
  - Strong SE: bitwise identical results.
  - Weak SE: equivalent mathematically but due to quirks of floating point arithmetic, not bitwise identical.
- Using a limited subset of OpenMP and a set of rules allows to program this way
- Advantages:
  - program can be tested,debugged and used in sequential mode
  - this style of programming is also less error prone

Rules for Strong Sequential Equivalence

- Control data scope with the base language
  - Avoid the data scope clauses.
  - Only use private for scratch variables local to a block (eg. temporaries or loop control variables) whose global initialization don't matter.
- Locate all cases where a shared variable can be written by multiple threads.
  - The access to the variable must be protected.
  - If multiple threads combine results into a single value, enforce sequential order.
  - Do not use the reduction clause carelessly. (no floating point operations +,-,*)
  - **Use the ordered directive and the ordered clause.**
- Concentrate on loop parallelism/data parallelism

Example for Ordered Clause: pio.c

```
#pragma omp for ordered
  for (i=1;i<=n;i++)
  {
    x=w*((double)i-0.5);
    myf=f(x);      /* f(x) should be expensive! */
#pragma omp ordered
    {
      sum=sum+myf;
    }
  }
```

ordered clause

ordered directive

- "ordered" corresponds to "critical" + "order of execution"
- only efficient if workload outside ordered directive is large enough

Rules for weak sequential equivalence

- For weak sequential equivalence only mathematically valid constraints are enforced.
  - **Floating point arithmetic is not associative and not commutative.**
  - **In many cases, no particular grouping of floating point operations is mathematically preferred so why take a performance hit by forcing the sequential order?**
  - In most cases, if you need a particular grouping of floating point operations, you have a bad algorithm.
- How do you write a program that is portable and satisfies weak sequential equivalence?
  - Follow the same rules as the strong case, but relax sequential ordering constraints.

Optimization Problems

- Prevent frequent synchronizations, e.g., with critical sections

```
max = 0;
#pragma omp parallel private(partial_max)
{
   partial_max = 0;
#pragma omp for
   for (i=0; i<10000; i++)
   {
     x[i] = ...;
     if (x[i] > partial_max) partial_max = x[i];
   }
#pragma omp critical
   if (partial_max > max) max = partial_max;
}
```

- Loop: `partial_max` is updated locally up to `10000/#threads` times
- Critical section: `max` is updated only up to `#threads` times

OpenMP Summary

- Standardized compiler directives for shared memory programming
- Fork-join model based on threads
- Support from all relevant hardware vendors
- OpenMP offers an incremental approach to parallelism
- OpenMP allows to keep one source code version for scalar and parallel execution
- Equivalence to sequential program is possible if necessary
  - strong equivalence
  - weak equivalence
  - no equivalence
- OpenMP programming includes race conditions and deadlocks, but a subset of OpenMP can be considered safe
- Tools like assure help to write correct parallel programs