Introduction to OpenMP

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

Matthias Müller,
Isabel Loebich, Rolf Rabenseifner
[mueller | loebich | rabenseifner] @hlrs.de

OpenMP Overview: What is OpenMP?

- Model for parallel programming
- Shared-memory parallelism
- Portable across shared-memory architectures
- Scalable
- Incremental parallelization
- Compiler based
- Extensions to existing programming languages
  - mainly by directives
  - a few library routines
- Fortran and C/C++ binding
- Supports data parallelism
Motivation: Why should I use OpenMP?

Where should I use OpenMP?
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:
```c
void main()
{
    double Res[1000];
    for(int i=0;i<1000;i++)
    {
        do_huge_comp(Res[i]);
    }
}
```

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

Outline

- Standardization Body: Who owns OpenMP?
- OpenMP Application Program Interface (API)
- Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Data environment (part 1)
  - Environment variables
  - Runtime library routines
  - Exercises 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
  - Combined parallel work-sharing directives
- Summary of OpenMP API
- OpenMP Pitfalls
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)
- Kuck & Associates, Inc. (KAI)
- Silicon Graphics, Inc.
- Sun Microsystems, Inc

OpenMP timeline

- OpenMP 1.0 for Fortran released October 1997
- OpenMP 1.0 for C/C++ released at October 1998
- OpenMP 1.1 for Fortran released at October 1999
- OpenMP 2.0 for Fortran released at November 2000
- OpenMP 2.0 for C/C++ to be released at November 2001

- OpenMP 3.0 for C/C++ next
OpenMP Partners: Software Vendors

- Absoft Corporation
- Edinburgh Portable Compilers
- GENIAS Software GmbH
- KAI Software a Division of Intel America
- Myrias Computer Technologies Inc.
- The Portland Group Inc. (PGI)
- Veridian Pacific-Sierra Research

OpenMP Partners: Application Developers

- ADINA R&D Inc.
- ANSYS Inc.
- Dash Associates
- Fluent Inc.
- ILOG CPLEX Division
- Livermore Software Technology Corporation (LSTC)
- MECALOG SARL
- Oxford Molecular Group PLC
- The Numerical Algorithms Group Ltd. (NAG)
OpenMP Availability

- SGI MIPSpro F90 & F77 compilers version 7.2.1
- SGI MIPSpro C++ & C compilers version 7.3
- CRAY CF90 Programming Environment version 3.1
- CRAY C++ Programming Environment version 3.2
- KAI KAP/Pro Toolset: Guide (Fortran & C/C++)
- PGI PGF77/PGF90 Compilers (Linux, Solaris, Windows NT)
- PGI C and C++ (Linux, Solaris, Windows NT)
- DIGITAL Fortran V5.1 for Alpha/Unix
- DIGITAL C/C++ for Alpha/Unix
- NEC Fortran Compiler
- Hitachi Fortran and C Compiler
- IBM
- SUN

OpenMP Availability

<table>
<thead>
<tr>
<th></th>
<th>Fortran</th>
<th>C</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compaq</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>HP</td>
<td>✔</td>
<td>beta</td>
<td>beta</td>
</tr>
<tr>
<td>IBM</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>SGI</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>SUN</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Cray</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Hitachi</td>
<td>✔</td>
<td>✔</td>
<td>In prep</td>
</tr>
<tr>
<td>NEC</td>
<td>✔</td>
<td>In prep</td>
<td>In prep</td>
</tr>
<tr>
<td>Intel IA32</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Intel IA64</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

- Fortran means Fortran 90 and OpenMP 1.1
- OpenMP is available on all platforms for all language bindings
OpenMP Information

- OpenMP Homepage: http://www.openmp.org/
- OpenMP at HLRS: http://www.hlrs.de/organization/par/services/models/openmp/

Outline — 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
  - 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
  - Combined parallel work-sharing directives
- 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.
- Careless use of synchronization can lead to dead-locks

Outline — Execution Model

- Standardization Body
- OpenMP Application Program Interface (API)
- Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Data environment (part 1)
  - Environment variables
  - Runtime library routines
  - Compilation + Exercise 1: hello
- 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
  - Combined parallel work-sharing directives
- Summary of OpenMP API
- OpenMP Pitfalls
OpenMP Execution Model (1)

- 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 Execution Model (2)

Sequential Part

Parallel Region

Sequential Part

Parallel Region

Sequential Part

Team of Threads

Master Thread

Team of Threads

Master Thread
OpenMP Parallel Region Construct (1)

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

OpenMP Parallel Region Construct (2)

Fortran:
```
!$OMP PARALLEL
block
!$OMP END PARALLEL
```

C/C++:
```
#pragma omp parallel
structured block
*/ omp end parallel */
```
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 [in my_fixed_form.F or my_free_form.F90] #endif`
- 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);
    ```
Outline — Exercise 1: hello

- Standardization Body
- OpenMP Application Program Interface (API)
- Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Data environment (part 1)
  - Environment variables
  - Runtime library routines
  - Compilation + Exercise 1: hello
- 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
  - Combined parallel work-sharing directives
- Summary of OpenMP API
- OpenMP Pitfalls

OpenMP on vision.rus....:

- Log into vision (vision.rus.uni-stuttgart.de)
- Change into directory ~/OpenMP/#NR/
  Compile in this directory.
- Compiler calls:
  - Fortran 77/90: f90 -mp -o executable source.f / .f90 / .F / .F90
  - C: cc -mp -o executable source.c
  - C++: CC -mp -o executable source.cc
- Execution:
  - environment: setenv OMP_NUM_THREADS number_of_threads
  - program start: ./executable
hwwsr8k: Cross Compiler Environment

- Log into hwwhpn.hww.de
- `bash` or `tcsh` [please ignore some stupid warnings]
- Change into directory `~/OpenMP/#NR/
  Compile in this directory.
- Compiler calls: [Crosscompiler on hwwhpv]
  - Fortran 77/90: `xf90 -OSS -parallel -omp`
  - C: `gcc -O4 -pvec -Op -parallel -omp`
  - C++: `not yet available`
- Execution:
  - Log into hwwsr8k.hww.de
  - Change to `~/../hp-v/OpenMP/#NR/
  - This is the same directory.
  - If you create an executable at hwwhpv you can execute here.
  - Execute with `prun -p single ./a.out`
**hwxsx*: Cross Compiler Environment**

- Log into hwwhpn.hww.de
- Change into directory `~/OpenMP/#NR/`
  Compile in this directory.
- Compiler calls: [Crosscompiler on hwwhpn]
  - Fortran 77/90: `sxf90 [-sx4] -P openmp`
  - C: available with C++
  - C++: available soon
- Log into hwxsx2.hww.de
  Change to `~/hpv/OpenMP/#NR/`
  This is the same directory.
  If you create an executable at hwwhpn you can execute here.

- Don’t forget -sx4 compiler flag to create executables for sx4!

**OpenMP Exercise 1: Hello World Program (1)**

- Goal: usage of
  - runtime library calls
  - conditional compilation
  - environment variables
  - parallel regions, private and shared clauses

- Working directory: `~/OpenMP/#NR/hello/`
  `#NR = number of your PC, e.g., 07`

- Serial programs:
  - Fortran 77: `hello.f`
  - Fortran 90: `hello.f90`
  - C: `hello.c`
OpenMP Exercise 1: Hello World Program (2)

- compile serial program hello.[f|f90|c] and run
- add call to omp_get_thread_num() in hello.[f|f90|c] and assign function result to variable i, and do this with conditional compilation (!$ in Fortran, #ifdef _OPENMP in C)
- compile serially and run
  - expected result: one line with i=-1
- compile as OpenMP program and run
  - add OpenMP compile option, e.g., -mp on SGI
  - expected result: one line with i=0
- set environment variable OMP_NUM_THREADS to 2 and run
  - expected result: one line with i=0
  - Why?
- value of i? number of printed lines?

OpenMP Exercise 1: Hello World Program (3)

- add parallel region in hello.[f|f90|c], compile and run
  - value of i? number of printed lines?
  - expected results: two lines, but values may be 0&0, 0&1, 1&0 or 1&1
- change OMP_NUM_THREADS to 4 and run
  - value of i? number of printed lines?
  - expected results: four lines, but also unpredictable
- add PRIVATE(i) clause in hello.[f|f90|c], compile and run
  - value of i? number of printed lines?
  - expected results: four lines with i=0, 1, 2, and 3 in some order
- change OMP_NUM_THREADS and run
  - value of i? number of printed lines?
- compile again serially and run
  - expected result: one line with i=-1
OpenMP Exercise 1: Hello World Program - Solution

Location: ~/OpenMP/Aufgabe/solution/hello
- hello1.[f|f90|c]: original program
- hello2.[f|f90|c]: incorrect (without parallel region) !!!
- hello3.[f|f90|c]: incorrect (i is not private) !!!
- hello4.[f|f90|c]: solution

OpenMP Exercise 1: Hello World Program - Summary

- Conditional compilation allows to keep the serial version of the program in the same source files
- Compilers need to be used with special option for OpenMP directives to take any effect
- Parallel regions are executed by each thread in the same way unless worksharing directives are used
- Decision about private or shared status of variables is important
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
  - 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

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 work!!!
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 (1)

- Several blocks are executed in parallel
- Fortran:
  ```
  !$OMP SECTION [ clause [, clause ] ... ]
  [ !$OMP SECTION ]
  block1
  [ !$OMP SECTION
  block2 ]
  ...
  !$OMP END SECTION [ 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 sections Directives (2)

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
```

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 (3)
OpenMP do/for Directives (1)

- 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 (2)

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 (3)

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);
}
```

- C / C++:
  - `#pragma omp parallel private(f)`
  - `#pragma omp for` for loop
  - `a[i] = b[i] + f * (i+1);`

OpenMP do/for Directives (4)

- `clause` can be one of the following:
  - `private(list)`
  - `reduction(operator; list)`
  - `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

---

7. Introduction to OpenMP 7-23
**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**

<table>
<thead>
<tr>
<th>static</th>
<th>dynamic(3)</th>
<th>guided(1)</th>
</tr>
</thead>
</table>

- [Diagram of loop scheduling]
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 2: π
- Data environment and combined constructs
  - Private and shared variables
  - Combined parallel work-sharing directives
- 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:
  ```fortran
  !$OMP CRITICAL [( name ) ]
  block
  !$OMP END CRITICAL [( name ) ]
  ```
- C/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 (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 PARALLEL
```
OpenMP critical — an example (C/C++)

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

OpenMP critical — another example (Fortran)
```fortran
integer :: mx = 0
!
!$OMP PARALLEL private(pmax)
!
!$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
```
OpenMP critical — another example (C/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*/
```

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

```fortran
program a
  !$OMP PARALLEL
  call b
  call c
  !$OMP END PARALLEL
  call d
  stop
  end

subroutine b
  !$OMP DO
  do i=1,n
  ...
  enddo
  return
end

subroutine c
  return
end
```
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 2: 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 2: pi
• Data environment and combined constructs
  – Private and shared variables
  – Combined parallel work-sharing directives
• 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 reproducible in 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 — Date 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.
**Private Clause**

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

  ```fortran
  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 !!!
  ```

- 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`, `.and`, `max`, `min`, `iand`, `ior`, `iand`, `ior`, `ieor`
  - C/C++:
    - `+`, `-`, `&`, `^`, `|`, `&&`, `||`
- Variables must be shared in the enclosing context
- 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 parallel for private(r)
   reduction(+:sm)
for( i=0; i<20; i++)
   { 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:
  ```fortran
  !$OMP PARALLEL DO [ clause [ , ] clause ] ... 
      do_loop 
  [ !$OMP END PARALLEL DO ]
  ```
- C/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 — an example (Fortran)

Fortran:

```fortran
f=7

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

!$OMP END PARALLEL DO
```

```plaintext
f=7

i=1,5
f=7

i=6,10

i=11,15

i=16,20
```

```plaintext
a(1) = b(1) + 7 * 1
a(2) = b(2) + 7 * 2
...
a(5) = b(5) + 7 * 5
```

```plaintext
a(6) = b(6) + 7 * 6
a(7) = b(7) + 7 * 7
...
a(10) = b(10) + 7 * 10
```

```plaintext
a(11) = b(11) + 7 * 11
a(12) = b(12) + 7 * 12
...
a(15) = b(15) + 7 * 15
```

```plaintext
a(16) = b(16) + 7 * 16
a(17) = b(17) + 7 * 17
...
a(20) = b(20) + 7 * 20
```
OpenMP Combined parallel do/for — an example (C/C++)

C / C++:

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

Outline — Exercise 3: $\pi$ with reduction

- 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: $\pi$ with reduction
- Summary of OpenMP API
- OpenMP Pitfalls
OpenMP Exercise 3: pi Program (6)

• Goal: usage of
  – work-sharing constructs: do/for
  – critical directive
  – reduction clause
  – combined parallel work-sharing constructs:
    parallel do/parallel for

• Working directory: ~/OpenMP/#NR/pi/
  #NR = number of your PC, e.g., 07

• Use your result pi.[f|f90|c] from the exercise 2
• or copy solution of exercise 2 to your directory:
  – cp ~/OpenMP/Aufgabe/solution/pi/pic2.* .

OpenMP Exercise 3: pi Program (7)

• remove critical directive in pi.[f|f90|c], add reduction clause 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?
• run again
  – value of pi? execution time?
OpenMP Exercise 3: pi Program (8)

- change parallel region `+ do/for` to the combined parallel work-sharing construct `parallel do/parallel for` 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?
- run again
  - value of pi?

OpenMP Exercise 3: 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
- `pir.[f|f90|c]`: solution with reduction clause
OpenMP Exercise 3: pi Program - Execution Times F90

OpenMP Exercise 3: pi Program - Summary

- Decision about private or shared status of variables is important
- In Fortran correct results with reduction clause and with critical directive
- Using the simple version of the critical directive is much more time consuming than using the reduction clause ⇒ no parallelism left
- More sophisticated use of critical directive leads to much better performance in Fortran;
- Convenient reduction clause
- Convenient shortcut form
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 3: pi

- 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
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

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)

```c
!$OMP PARALLEL SECTIONS
   A = B + C
!$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)

```c
!$OMP PARALLEL SHARED (X), PRIVATE(TMP)
   ID = OMP_GET_THREAD_NUM()
!$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.

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 (e.g., 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

```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” 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
- Support from all relevant hardware vendors
- OpenMP offers an incremental approach to parallelism
  - strong equivalence
  - weak equivalence
  - no equivalence
- OpenMP programming includes race conditions and deadlocks, but a subset of OpenMP can be considered safe
- Optimization: prevent frequent synchronizations

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: `sx90 -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 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

- 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