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 12, June 13, 2006 (for OpenMP 2.5 and older)
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?

- Performance
- OpenMP
- MPI
- Code does not work
- OpenMP+MPI
- Time/Effort
- Scalar Program
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?

<table>
<thead>
<tr>
<th>#CPUs</th>
<th>Problem size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OpenMP</td>
</tr>
<tr>
<td></td>
<td>Dominated by Overhead</td>
</tr>
<tr>
<td>1</td>
<td>MPI</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalar</td>
<td></td>
</tr>
</tbody>
</table>

---

7. Introduction to OpenMP 7-3
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:
```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]);
    }
}
```

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 + Kuck & Associates, Inc. (KAI)
- International Business Machines (IBM)
- Silicon Graphics, Inc.
- Sun Microsystems, Inc
- cOMPUnity
- NEC
OpenMP Release History

- 1997: OpenMP Fortran 1.0
- 1998: OpenMP C/C++ 1.0
- 1999: OpenMP Fortran 1.1
- 2000: OpenMP Fortran 2.0
- 2002: OpenMP C/C++ 2.0
- May 2005: OpenMP 2.5

OpenMP Availability

- OpenMP 1.0 (C/C++) and OpenMP 1.1 (Fortran 90) is available on all platforms in the commercial compilers.
- Most features from OpenMP 2.0 are already implemented.
- OpenMP 2.5 — no substantial new features/changes compared to 2.0.
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.  
- R. Eigenmann, Michael J. Voss (Eds):  
  OpenMP Shared Memory Parallel Programming.  

Outline — Programming and Execution Model

- Introduction into OpenMP
- Programming and Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Data environment (part 1)
    - Environment variables
    - Runtime library routines
    - Exercise 1: Parallel region / library calls / privat & shared variables
  - Work-sharing directives
    - Which thread executes which statement or operation?
    - Synchronization constructs, e.g., critical regions
    - Nesting and Binding
    - Exercise 2: Pi
  - Data environment and combined constructs
    - Private and shared variables, Reduction clause
    - Combined parallel work-sharing directives
    - Exercise 3: Pi with reduction clause and combined constructs
    - Exercise 4: 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

Sequential Part

Parallel Region

Sequential Part

Parallel Region

Sequential Part

Team of Threads

Master Thread

Team of Threads

Master Thread
**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:**

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

**C / 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)
    - ...

[ xxx ] = xxx is optional

OpenMP Directive Format: C/C++

- `#pragma directives` — case sensitive
- 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
  ```
- Include file for library routines:
  
  ```
  #ifdef _OPENMP
  #include <omp.h>
  #endif
  ```
- In the old OpenMP 1.0 syntax, the comma (,) between clauses was not allowed (some compilers in use still may have this restriction)
**OpenMP Directive Format: Fortran**

- Treated as Fortran comments – not case sensitive
- Format: 
  
  ```fortran
  sentinel directive_name [ clause [ , clause ] ... ]
  ```

- Directive sentinels:
  - Fixed source form: `!$OMP | C$OMP | !$OMP` [starting at column 1]
  - Free source form: `!$OMP` [may be preceded by white space]
- Conditional compilation
  - Fixed source form: `!$ | C$ | *
  - Free source form: `!$`
  - `#ifdef _OPENMP` [in `my_fixed_form.F` or `.f90`]
    - `block`
    - `#endif`
  - Example:
    - `!$ write(*,*) OMP_GET_NUM_PROCS(),' avail. processors'`
- Include file for library routines:
  - `include omp_lib.h` or `use omp_lib [implementation dependent]`

**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
  or use include file
  !$ INCLUDE 'omp_lib.h'
  or module
  !$ USE omp_lib
  ```
  Existence of include file or module or both is implementation dependent.
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:
    ```fortran
    integer function omp_get_num_threads()
    end function
    ```
  - C/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:
    ```fortran
    integer function omp_get_thread_num()
    end function
    ```
  - C/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
    ```c
    double omp_get_wtime()
    ```
    ```c
    ! Work to be measured
    END = OMP_GET_WTIME()
    ```
    ```c
    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
OpenMP Exercise 1: Parallel region (1)

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

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

- Serial programs:
  - Fortran 77: `pi.f`
  - Fortran 90: `pi.f90`
  - C: `pi.c`
OpenMP Exercise 1: Parallel region (2)

- Compile the serial program `pi.[f|f90|c]` and run.
- Compile as OpenMP program and run on 4 CPUs:
  - Add OpenMP compile option, see login slides, e.g.,
    - `-mp` on SGI
    - `--openmp` on Intel compiler ecc and efc
  - Export `OMP_NUM_THREADS=4`
  - `./pi`
- Expected result: program is not parallelized, therefore same pi-value and timing, additional output from `omp_get_wtime()`

OpenMP Exercise 1: Parallel region (3)

- Directly after the declaration part, add a parallel region that prints on each thread
  - Its rank (with `omp_get_thread_num()`)
  - The number of threads (with `omp_get_num_threads()`)
- Compile and run on 4 CPUs
- Expected results: numerical calculation is still not parallelized, therefore still same pi-value and timing, additionally output:

```
bash$ ecc --openmp -o pi0-parallel pi0.c
bash$ export OMP_NUM_THREADS=4; ./pi0-parallel
I am thread 0 of 4 threads
I am thread 2 of 4 threads
I am thread 3 of 4 threads
I am thread 1 of 4 threads
computed pi = 3.141592653589731
CPU time (clock) = 0.06734 sec
wall clock time (omp_get_wtime) = 0.06682 sec
wall clock time (gettimeofday) = 0.06683 sec
```

undefined sequence!
OpenMP Advanced Exercise 1a: Parallel region (4)

- Use a private variable for the rank of the threads
- Check, whether you can get a race-condition if you forget the `private` clause on the `omp parallel` directive, e.g.:

```
I am thread 2 of 4 threads
I am thread 2 of 4 threads
I am thread 2 of 4 threads
I am thread 2 of 4 threads
```

- Don’t wonder if you get always correct output because the compiler may use on each thread a private register instead of writing into the shared memory

OpenMP Advanced Exercise 1b: Parallel region (5)

- Guarantee with conditional compilation, that source code still works with non-OpenMP compilers (i.e., without OpenMP compile-option).
- Add an "else clause", printing a text if OpenMP is not used.
- Expected output:
  - If compiled with OpenMP, see previous slide.
  - If compiled without OpenMP:

```
bash$ ecc -o pi0-serial pi0.c
bash$ export OMP_NUM_THREADS=4; ./pi0-serial
This program is not compiled with OpenMP
computed pi = 3.1415926535897931
CPU time (clock) = 0.06734 sec
wall clock time (gettimeofday) = 0.06706 sec
```
OpenMP Exercise 1: Parallel region – Solution

Location: ~/OpenMP/solution/pi
- pi.[f|f90|c]  original program
- pi0.[f|f90|c]  solution (includes all 3 exercises)

OpenMP Exercise 1: 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 (Advanced Exercise 1a)
Outline — Work-sharing directives

- Introduction into OpenMP
- Programming and Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Data environment (part 1)
  - Environment variables
  - Runtime library routines
  - Exercise 1: Parallel region / library calls / privat & shared variables
- Work-sharing directives
  - Which thread executes which statement or operation?
  - Synchronization constructs, e.g., critical regions
  - Nesting and Binding
  - Exercise 2: Pi
- Data environment and combined constructs
  - Private and shared variables, Reduction clause
  - Combined parallel work-sharing directives
  - Exercise 3: Pi with reduction clause and combined constructs
  - Exercise 4: 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 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
- `for` directive (C/C++)
- `do` directive (Fortran)
- `workshare` directive (Fortran)
- `single` directive

OpenMP sections Directives – 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:

```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:
  ```fortran
  !$OMP sections [ clause [, clause] ... ]
  !$OMP section
  block1
  !$OMP section
  block2
  ...
  !$OMP end sections [ nowait ]
  ```
- C/C++:
  ```c
  #pragma omp sections [ clause [, clause] ... ] new-line
  {
  [ #pragma omp section new-line
  structured-block1
  [ #pragma omp section new-line
  structured-block2
  ...
  ]
  }
  ```
### C/C++ OpenMP do/for Directives – C/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 */
```

### Fortran OpenMP do/for Directives – Fortran

**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
```

---

7. — Introduction to OpenMP — 7. 7-21
OpenMP do/for Directives – Syntax

- Immediately following loop executed in parallel
- Fortran:
  
  ```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++:
  
  ```c
  #pragma omp for [ clause [ , clause ] ... ]
  ```

- The corresponding for loop must have canonical shape:
  ```c
  for( [integer type] var=ub; var<b; var++)
  $$\begin{align*}
  \text{var} &= \text{var} + \text{incr} \\
  \text{var} &= \text{var} + \text{incr} \\
  \text{var} &= \text{var} + \text{incr} \\
  \text{var} &= \text{var} + \text{incr} \\
  \end{align*}$$
  ```

  - `var, b, incr`: signed integer, must not be modified in the loop body

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.
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.

---

**OpenMP single Directive – Syntax**

- The block is executed by only one thread in the team (not necessarily the master thread)

**Fortran:**

```
 !$OMP single [ clause [,, clause] ... ]
 block
 !$OMP end single [nowait]
```

**C/C++:**

```
 #pragma omp single [ clause [,, clause] ... ] new-line structured-block
```

- Implicit barrier at the end of `single` construct (unless a `nowait` clause is specified)
- To reduce the fork-join overhead, one can combine
  - several parallel parts (for, do, workshare, sections)
  - and sequential parts (single)
  in one parallel region (parallel ... end parallel)
Outline — Synchronization constructs

- Introduction into OpenMP
- Programming and Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Data environment (part 1)
  - Environment variables
  - Runtime library routines
  - Exercise 1: Parallel region / library calls / privat & shared variables
- Work-sharing directives
  - Which thread executes which statement or operation?
  - Synchronization constructs, e.g., critical regions
  - Nesting and Binding
  - Exercise 2: Pi
- Data environment and combined constructs
  - Private and shared variables, Reduction clause
  - Combined parallel work-sharing directives
  - Exercise 3: Pi with reduction clause and combined constructs
  - Exercise 4: 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:
  \begin{verbatim}
  !$OMP CRITICAL[(name)] block
  !$OMP END CRITICAL [(name)]
  \end{verbatim}
- C/C++:
  \begin{verbatim}
  #pragma omp critical[(name)] new-line structured-block
  \end{verbatim}
- 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++:
\begin{verbatim}
cnt = 0;
f=7;
#pragma omp parallel
{
#pragma omp for
for (i=0; i<20; i++) {
  if (b[i] == 0) {
    #pragma omp critical
    cnt ++;
  }
  a[i] = b[i] + f * (i+1);
}
} /*omp end parallel */
\end{verbatim}
OpenMP critical — an example (Fortran)

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

OpenMP critical — another example (Fortran)

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

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

Outline — Nesting and Binding

- Introduction into OpenMP
- Programming and Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Data environment (part 1)
  - Environment variables
  - Runtime library routines
  - Exercise 1: Parallel region / library calls / privat & shared variables
- Work-sharing directives
  - Which thread executes which statement or operation?
  - Synchronization constructs, e.g., critical regions
  - Nesting and Binding
    - Exercise 2: Pi
- Data environment and combined constructs
  - Private and shared variables, Reduction clause
  - Combined parallel work-sharing directives
  - Exercise 3: Pi with reduction clause and combined constructs
  - Exercise 4: Heat
- Summary of OpenMP API
- OpenMP Pitfalls

---

7. — Introduction to OpenMP — 7-28
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

[The terms *lexical extent* and *dynamic extent* are no longer used in OpenMP 2.5, but still helpful to explain the complex impact of OpenMP directives.]
OpenMP Control Structures — Summary

- Parallel region construct
  - `parallel`

- Work-sharing constructs
  - `sections`
  - `for` (C/C++)
  - `do` (Fortran)
  - `workshare` (Fortran)
  - `single`

- Combined parallel work-sharing constructs [see later]
  - `parallel for` (C/C++)
  - `parallel do` (Fortran)
  - `parallel workshare` (Fortran)

- Synchronization constructs
  - `critical`

Outline — Exercise 2: pi

- Introduction into OpenMP
- Programming and Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Data environment
  - Environment variables
  - Runtime library routines
  - Exercise 1: Parallel region / library calls / privat & shared variables

- Work-sharing directives
  - Which thread executes which statement or operation?
  - Synchronization constructs, e.g., critical regions
  - Nesting and Binding
  - Exercise 2: PI

- Data environment and combined constructs
  - Private and shared variables, Reduction clause
  - Combined parallel work-sharing directives
  - Exercise 3: PI with reduction clause and combined constructs
  - Exercise 4: 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
  - Fortran 90: pi.f90
  - C: pi.c

- Use your result pi.[f|f90|c] from the exercise 1
- or copy solution of exercise 1 to your directory:
  - cp ~/OpenMP/solution/pi/pi0.* .

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 is 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.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 Advanced Exercise 2: pi Program (5)

- Modify the printing of the thread rank and the number of threads from Exercise 1:
  - Only one thread should print the real number of threads used in parallel regions.
  - For this, use a single construct
- Expected result:

```plaintext
OpenMP-parallel with 4 threads
computed pi = 3.14159265358967
CPU time (clock) = 0.01659 sec
wall clock time (omp_get_wtime) = 0.01678 sec
wall clock time (gettimeofday) = 0.01679 sec
```
OpenMP Exercise 2: pi Program - Solution

Location: ~/OpenMP/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

- Introduction into OpenMP
- Programming and Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Data environment (part 1)
  - Environment variables
  - Runtime library routines
  - Exercise 1: Parallel region / library calls / privat & shared variables
- Work-sharing directives
  - Which thread executes which statement or operation?
  - Synchronization constructs, e.g., critical regions
  - Nesting and Binding
  - Exercise 2: Pi
- Data environment and combined constructs
  - Private and shared variables, Reduction clause
  - Combined parallel work-sharing directives
  - Exercise 3: Pi with reduction clause and combined constructs
  - Exercise 4: Heat
- Summary of OpenMP API
- OpenMP Pitfalls

OpenMP Exercise 2: pi Program - Solution

Location: ~/OpenMP/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

- Introduction into OpenMP
- Programming and Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Data environment (part 1)
  - Environment variables
  - Runtime library routines
  - Exercise 1: Parallel region / library calls / privat & shared variables
- Work-sharing directives
  - Which thread executes which statement or operation?
  - Synchronization constructs, e.g., critical regions
  - Nesting and Binding
  - Exercise 2: Pi
- Data environment and combined constructs
  - Private and shared variables, Reduction clause
  - Combined parallel work-sharing directives
  - Exercise 3: Pi with reduction clause and combined constructs
  - Exercise 4: Heat
- 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 uninitialized
  - 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 )`
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:

```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
```

Diagram:

```
sm = 0
```

```
1.5 5.10
```

```
16.20
```

```
sm =
```

```
sm +=
```

```
sm +=
```

```
sm +=
```

```
sm +=
```

```
sm +=
```

```
sm +=
```

```
sm +=
```

```
sm +=
```
```
OpenMP reduction — an example (C/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;
}
/*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 — an example**

**Fortran**

```
f=7

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

!$OMP END PARALLEL DO
```

**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

- Introduction into OpenMP
- Programming and Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Data environment (part 1)
  - Environment variables
  - Runtime library routines
  - Exercise 1: Parallel region / library calls / privat & shared variables
- Work-sharing directives
  - Which thread executes which statement or operation?
  - Synchronization constructs, e.g., critical regions
  - Nesting and Binding
  - Exercise 2: Pi
- Data environment and combined constructs
  - Private and shared variables, Reduction clause
  - Combined parallel work-sharing directives
  - Exercise 3: Pi with reduction clause and combined constructs
  - Exercise 4: Heat
- Summary of OpenMP API
- OpenMP Pitfalls

OpenMP Exercise 3: pi Program (6)

- Goal: usage of
  - work-sharing constructs: \texttt{do/for}
  - critical directive
  - reduction clause
  - combined parallel work-sharing constructs:
    \texttt{parallel do/parallel for}

- Working directory: \texttt{~/OpenMP/#NR/pi/}
  \#NR = number of your PC, e.g., 07

- Use your result \texttt{pi.[f|f90|c]} from the exercise 2
- or copy solution of exercise 2 to your directory:
  \texttt{cp ~/OpenMP/solution/pi/pic2.*}
OpenMP Exercise 3: pi Program (7)

- remove critical directive in pi.f90, 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?

- At the end, compile again without OpenMP
  - Does your code still compute a correct value of pi?
OpenMP Exercise 3: pi Program - Solution

Location: ~/OpenMP/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

<table>
<thead>
<tr>
<th>no. of processes</th>
<th>pi</th>
<th>pic</th>
<th>pic2</th>
<th>pir</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
OpenMP Exercise 3: pi Program - Summary

- Decision about private or shared status of variables is important
- 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 \(\Rightarrow\) no parallelism left
- More sophisticated use of critical directive leads to much better performance
- Convenient reduction clause
- Convenient shortcut form

Outline — Exercise 4: Heat

- Introduction into OpenMP
- Programming and Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Data environment (part 1)
  - Environment variables
  - Runtime library routines
  - Exercise 1: Parallel region / library calls / privat & shared variables
- Work-sharing directives
  - Which thread executes which statement or operation?
  - Synchronization constructs, e.g., critical regions
  - Nesting and Binding
  - Exercise 2: Pi
- Data environment and combined constructs
  - Private and shared variables, Reduction clause
  - Combined parallel work-sharing directives
  - Exercise 3: Pi with reduction clause and combined constructs
- Exercise 4: Heat Conduction Exercise
- Summary of OpenMP API
- OpenMP Pitfalls
OpenMP Exercise: Heat Conduction(1)

- solves the PDE for unsteady heat conduction \( \frac{df}{dt} = \Delta f \)
- 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: 20x20

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: `heat.F`
  - C: `heat.c`
- Compiler calls:
  - See login slides
- Options:
  - `-O4 -Dimax=80 -Dkmax=80` (default is 20x20)
  - `-O4 -Dimax=250 -Dkmax=250`
  - `-O4 -Dimax=1000 -Dkmax=1000 -Ditmax=500`
OpenMP Exercise: Heat Conduction (3)

Tasks:

TO DO
Parallelize heat.c or heat.F
- Use critical sections for global maximum
- Use trick with partial maximum inside of the parallelized loop, and critical section outside of the loop to compute global maximum
- Hints:
  - Parallelize outer loop (index \( k \) in Fortran, \( i \) in C)
  - make inner loop index private!

TO DO
Compile an run with 80x80 serial, and parallel with 1, 2, 3, 4 threads
- Result may look like
  Serial: 0.4 sec, 1 thread: 0.5 sec, 2 threads: **2.8 sec**, ...
- Why is the parallel version significantly slower than the serial one?

TO DO

OpenMP Exercise: Heat Conduction (4)

- Reason already in the serial program:
  - Bad sequence of the nested loops

```
for (k=1;k<kmax;k++)
for (i=1;i<imax;i++)
dphi=(phi[i+1][k]+phi[i-1][k]-2.*phi[i][k])*dy2i + (phi[i][k+1]+phi[i][k-1]-2.*phi[i][k])*dx2i;

for (i=1;i<imax;i++)
for (k=1;k<kmax;k++)
dphi=(phi[i+1][k]+phi[i-1][k]-2.*phi[i][k])*dy2i + (phi[i][k+1]+phi[i][k-1]-2.*phi[i][k])*dx2i;
```

- Inner loop should use contiguous index in the array, i.e.,
  - First index in Fortran ➔ “do i=...” must be inner loop
  - Second index in C/C++ ➔ “for (k=...)” must be inner loop

Automatically fixed by serial compiler!

Not fixed by OpenMP compiler!
**OpenMP Exercise: Heat Conduction (5)**

*TO DO*

**OpenMP Exercise: Heat Conduction (6)**

*TO DO*

---

**OpenMP Exercise: Heat Conduction (5)**

**Interchange sequence of nested loops for i and k**

**Compile an run parallel with 80x80 and with 1, 2, 3, 4 threads**

- Result may look like
  - 1 thread: 0.5 sec, 2 threads: **0.45 sec**, 3 threads: **0.40 sec**
  - Reasons:
    - Problem is too small — parallelization overhead too large

**Compile an run parallel with 250x250 and with 1, 2, 3, 4 threads**

- 1 thread: 4.24 sec, 2 threads: **2.72 sec**, 3 threads: **2.27 sec**
- Don’t worry that computation is prematurely finished by itmax=15000

**With 1000x1000 and –Ditmax=500 and with 1, 2, 3, 4 threads**

- 1 thread: 5.96 sec, 2 threads: **2.79 sec**, 3 threads: **1.35 sec**
- Super-linear speed-up due to better cache reuse on smaller problem

---

**OpenMP Exercise: Heat Conduction (6)**

**Advanced exercise**

- Substitute
  - the current parallel region that is forked and joined in each $it=\ldots$ iteration
  - by a parallel region around $it=\ldots$ loop forked and joined only once
  - Caution:
    - dphimax=0 must be surrounded by
      - #pragma omp barrier
      - #pragma omp single
        - dphimax=0;
    - Why?

```
/*time step iteration*/
for (it=1;it<=itmax;it++)
{
  dphimax=0.; //line A/
  for (k=1;k<kmax;k++)
  {
    dphi=(phi[i+1][k]+phi[i-1][k]-2.*phi[i][k])*dy2i
     +(phi[i][k+1]+phi[i][k-1]-2.*phi[i][k])*dx2i;
    dphi=dphi*dt;
    dphimax=max(dphimax,dphi);
    phin[i][k]=phi[i][k]+dphi;
  }
}
for (k=1;k<kmax;k++)
{
  for (i=1;i<imax;i++)
  {
    phi[i][k]=phin[i][k];
  }
}
if(dphimax<eps) break; //line B/
```

---

**Shared(dphimax) is necessary for B.**

*Write-write-conflict on A-A without single.*

*Write-read-conflict on A-B without barrier.*
OpenMP Exercise: Heat Conduction (7)
Advanced exercise

TO DO
Execute abort-statement (if (dphimax<eps) ...)
only each 20th it=... iteration
Move omp barrier directly after if (dphimax<eps) ...
that this barrier is also executed only each 20th it=... iteration

TO DO
Add schedule(runtime) and compare execution time

Torran only:
Substitute critical-section-trick
by reduction(max:dphimax) clause

TO DO

OpenMP Exercise: Heat - Solution (1)

Location: ~/OpenMP/solution/heat
- heat.[F|c]
  Original program
- heat_x.[F|c]
  Better serial program with interchanged nested loops
- heatc.[F|c]
  Extremely slow solution with critical section inside iteration loop
- heatc2.[F|c]
  Slow solution with critical section outside inner loop, one parallel region inside time step iteration loop (it=...)
- heatc2_x.[F|c]
  Fast solution with critical section outside inner loop, one parallel region inside iteration loop, interchanged nested loops
- heatc3_x.[F|c]
  ... and parallel region outside of it=... loop
- heatc4_x.[F|c]
  ... and abort criterion only each 20th iteration
- heats2_x.F
  Solution with schedule(runtime) clause
- heatr2_x.F
  Solution with reduction clause, one parallel region inside iteration loop [reduction(max:... not available in C]
OpenMP Exercise: Heat - Solution (2)

- **heatc2 → heatc2_x**
  - Loss of optimization with OpenMP directives (and compilers)
- For controlling the parallelization:
  - Version 20x20: 1116 iterations
  - Version 80x80: 14320 iterations
  - Version 250x250: 15001 iterations [if itmax = 15000 (default)]
    - 110996 iterations [if itmax is extended to 150000]
- **heatc2_x ↔ heatc3_x**
  - Additional overhead for barriers and single sections
    - (including implied barrier)
    - must be compared with fork-join-overhead

OpenMP Exercise: Heat - Solution (3) – 20x20 Time

**Heat example, 20x20, 1116 iterations**

- heatc is extremely slow

Measurements on NEC TX-7 (asama.hww.de), 16 CPUs, May 4-5, 2006
OpenMP Exercise: Heat - Solution (4) – 20x20 Efficiency

Heat example, 20x20, 1116 steps

- 20x20: 20% overhead already with 1 thread
- 20x20: >90% overhead with ≥2 threads

OpenMP Exercise: Heat - Solution (5) – 80x80 Time

Heat example, 80x80, 14320 iterations

- 80x80: no chance without efficient loop nesting
- 80x80: still no speedup due to small (80x80) data set
OpenMP Exercise: Heat - Solution (6) – 80x80 Efficiency

Heat example, 80x80, 14320 steps

OpenMP Exercise: Heat - Solution (7) – 250x250 Time

Heat example, 250x250, 15000 iterations

250x250: speedup 1.6 with 2 threads
7. Introduction to OpenMP

OpenMP Exercise: Heat - Solution (8) – 250x250 Efficiency

Heat example, 250x250, 15000 steps

- Efficiency < 70% with ≥3 threads
- Number of threads: serial 1 2 3 4 6 8 10
- Heat example files: heat.c, heatc.c, heatc2.c, heat_x.c, heatc2_x.c, heatc3_x.c, heatc4_x.c

OpenMP Exercise: Heat - Solution (9) – 1000x1000 Time

Heat example, 1000x1000, 500 iterations

- Wall-clock execution time [sec]: serial 1 2 3 4 6 8 10
- Heat example files: heat.c, heatc.c, heatc2.c, heat_x.c, heatc2_x.c, heatc3_x.c, heatc4_x.c

Matthew Müller et al.
OpenMP Exercise: Heat - Solution (10) – 1000x1000 Efficiency

Heat example, 1000x1000, 500 steps

- Efficiency 100-200% with 3-10 threads due to cache effects

OpenMP Exercise: Heat - Execution Times F90 with 150x150

- Maximum overhead with dynamic schedule

Measurements with previous version of the software
OpenMP Exercise: Heat Conduction - Summary

• Overhead for parallel versions using 1 thread.
• Be careful with compiler based optimizations.
• Datasets must be large enough to achieve good speed-up.
• Thread Checker should be used to guarantee zero race conditions.
• Be careful when using other than default scheduling strategies:
  - dynamic: generally expensive
  - static: overhead for small chunk sizes is clearly visible

Outline — Summary of the OpenMP API

• Introduction into OpenMP
• Programming and Execution Model
  – Parallel regions: team of threads
  – Syntax
  – Data environment (part 1)
  – Environment variables
  – Runtime library routines
  – Exercise 1: Parallel region / library calls / privat & shared variables
• Work-sharing directives
  – Which thread executes which statement or operation?
  – Synchronization constructs, e.g., critical regions
  – Nesting and Binding
  – Exercise 2: Pi
• Data environment and combined constructs
  – Private and shared variables, Reduction clause
  – Combined parallel work-sharing directives
  – Exercise 3: Pi with reduction clause and combined constructs
  – Exercise 4: 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

Application

Directive Compiler

User

Environment Variables

Runtime Library

Threads in Operating System
OpenMP Constructs

Control Constructs
- Parallel Region
- Work Sharing
- Do/For
- Schedule
- Ordered
- Sections
- Single
- Workshare

Data Constructs
- Thread Private
- Data Scope
- Shared
- Private
- First Private
- Last Private
- Reduction
- CopyIn
- Default

Synchronization Constructs
- Master
- Critical
- Barrier
- Atomic
- Ordered

Binding
- Nesting
- Conditional Compilation

Directives
- OMP_SCHEDULE
- Static
- Dynamic, chunk
- Guided, chunk

Environment Functions
- OMP_NUM_THREADS
- OMP_DYNAMIC
- OMP_NESTED

Environment Variables
- Environment Functions
- Lock Functions

Outline — OpenMP Pitfalls

- Introduction into OpenMP
- Programming and Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Data environment (part 1)
  - Environment variables
  - Runtime library routines
  - Exercise 1: Parallel region / library calls / privat & shared variables
- Work-sharing directives
  - Which thread executes which statement or operation?
  - Synchronization constructs, e.g., critical regions
  - Nesting and Binding
  - Exercise 2: Pi
- Data environment and combined constructs
  - Private and shared variables, Reduction clause
  - Combined parallel work-sharing directives
  - Exercise 3: Pi with reduction clause and combined constructs
  - Exercise 4: Heat
- Summary of OpenMP API
- OpenMP Pitfalls
Implementation-defined behavior

- 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 levels of nested parallelism supported
- Atomic directives might be replaced by critical regions
- Behavior in case of thread exhaustion
- Allocation status of allocatable arrays that are not affected by COPYIN clause are undefined if dynamic thread mechanism is enabled
- Fortran: Is `include 'omp_lib.h'` or use `omp_lib` or both available?

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, FOR, SECTIONS)
  - END (SINGLE, WORKSHARE)
  - ORDERED AND END ORDERED
  - PARALLEL and END PARALLEL with their combined variants
- It is NOT implied at the following constructs:
  - Begin of DO, FOR
  - Begin of MASTER and END MASTER
  - Begin of SECTIONS
  - Begin of SINGLE
  - Begin of 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)

!$OMP MEAN SECTION
  A = B + C
!$OMP MEAN SECTION
  B = A + C
!$OMP MEAN SECTION
  C = B + A
!$OMP END PARALLEL MEAN

• 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.
  - Use tools like “Intel® Thread Checker” (formerly 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 (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 / .f / .f90

```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

Reproducible & efficient reduction results if OMP_NUM_THREADS is fixed

- On any platform with same rounding algorithm (e.g., IEEE)
- But with different OpenMP implementations and defaults
- Tricks:
  - Loop schedule(static, with fixed chunk size)
  - Reduction operation explicitly sequential
  - Disable dynamic adjustment of number of threads
Reproducible & efficient reduction results if OMP_NUM_THREADS is fixed

```c
n=100000000; w=1.0/n; sum=0.0;
#pragma omp parallel private (x,sum0,num_threads)
    shared(w,sum)
    { ifdef _OPENMP
        num_threads=omp_get_num_threads();
    else
        num_threads=1
    endif
    sum0=sum;
#pragma omp for schedule(static,(n-1)/num_threads+1)
    for (i=1;i<=n;i++)
        { x=w*{(double)i-0.5};
          sum0=sum0+4.0/(1.0+x*x);
        }
#pragma omp for ordered schedule(static,1)
    for (i=0;i<num_threads;i++)
        { #pragma omp ordered
            sum=sum+sum0;
        }
    pi=w*sum;
}
```

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.
Reentrant library functions

- Library functions (if called inside of parallel regions) must be reentrant
- **Automatically switched** if OpenMP option is used:
  - e.g., Intel compiler:
    - `efc -openmp -o my_prog my_prog.f` or `my_prog.f90` (Fortran)
    - `ecc -openmp -o my_prog my_prog.c` (C, C++)
- **Manually by compiler option**:
  - e.g., IBM compiler:
    - `xlf_r -O -qsmp=omp -o my_prog my_prog.f` (Fortran, fixed form)
    - `xlf90_r -O -qsmp=omp -o my_prog my_prog90.f` (Fortran, free form)
    - `xlc_r -O -qsmp=omp -o my_prog my_prog.c` (C)
- **The "_r" forces usage of reentrant library functions**
- **Manually by programmer**: Some library function are using an internal buffer to store its state – one must use its reentrant counterpart:
  - e.g., reentrant `erand48()` instead of `drand48()`
  - `gmtime_r()` `gmtime()`

Optimization Problems – Overview

- Prevent unnecessary fork and join of parallel regions
  - if you can execute several loop / workshare / sections / single inside of one parallel region
- Prevent unnecessary synchronizations
  - e.g. with critical or ordered regions inside of loops
- Prevent false-sharing (of cache-lines)
- Prevent unnecessary cache-coherence or memory communication
  - E.g., same schedules for same memory access patterns
  - First touch on (cc)NUMA architectures
    - To locate arrays/objects already in a parallelized initialization to the threads where they are mainly used
    - Pin the threads to CPUs [not useful in time sharing on over-committed systems]
  - Otherwise, after each time slice, threads may run on other CPUs
  - `dplace -x2 (SGI), O(1)` scheduler (HP), SUNW_OMP_PROBIND envir. (Sun)
  - `fplace -r -o 1.2, command (Intel), ...
Do not pin management threads
Get a feeling for the involved overheads

<table>
<thead>
<tr>
<th>Operation</th>
<th>Minimum overhead (cycles)</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hit L1 cache</td>
<td>1-10</td>
<td>Constant</td>
</tr>
<tr>
<td>Function call</td>
<td>10-20</td>
<td>Constant</td>
</tr>
<tr>
<td>Thread ID</td>
<td>10-50</td>
<td>Constant, log, linear</td>
</tr>
<tr>
<td>Integer divide</td>
<td>50-100</td>
<td>Constant</td>
</tr>
<tr>
<td>Static do/for, no barrier</td>
<td>100-200</td>
<td>Constant</td>
</tr>
<tr>
<td>Miss all caches</td>
<td>100-300</td>
<td>Constant</td>
</tr>
<tr>
<td>Lock acquisition</td>
<td>100-300</td>
<td>Depends on contention</td>
</tr>
<tr>
<td>Dynamic do/for, no barrier</td>
<td>1000-2000</td>
<td>Depends on contention</td>
</tr>
<tr>
<td>Barrier</td>
<td>200-500</td>
<td>Log, linear</td>
</tr>
<tr>
<td>Parallel</td>
<td>500-1000</td>
<td>Linear</td>
</tr>
<tr>
<td>Ordered</td>
<td>5000-10000</td>
<td>Depends on contention</td>
</tr>
</tbody>
</table>

All numbers are approximate!! They are very platform dependant!!

Optimization Problems

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

```c
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 region: `max` is updated only up to `#threads` times
False-sharing

Several threads are accessing data through the same cache-line.
This cache-line has to be moved between these threads.
This is very time-consuming.

False-sharing – an experiment (solution/pi/piarr.c)

```
n = 100000000;   w=1.0/n;   sum=0.0;
stride=1;  if (argc>1) stride=atoi(argv[1]);  /* unit is *double* */
#pragma omp parallel private (x,index) shared(w,sum,p_sum)
{  
   #ifdef _OPENMP
        index=stride*omp_get_thread_num();  
   #else
        index=0;  
   #endif
   p_sum[index]=0;
   #pragma omp for
   for (i=1;i<=n;i++)
   {  
      x=w*((double)i-0.5);
      p_sum[index]+(x>1?1:0)=p_sum[index]+4.0/(1.0+x*x);
      /* The term (x>1?1:0) is always zero. It is used to prohibit
         register caching of of p_sum[index], i.e., to guarantee that
         each access to this variable is done via cache in the memory. */
   }
   #pragma omp critical
   {  
      sum=sum+p_sum[index];  
   }
   pi=w*sum;
}  
```
Measurements on NEC TX-7 with 128 bytes cache lines, timings with false-sharing (stride 1-8 with more than 1 thread) were varying from run to run.

Although each thread accesses independent variables, the performance will be terrible, if these variables are located in the same cache-line.

False-sharing – results from the experiment

False-sharing – same with speed-up

Measurements on NEC TX-7 with 128 bytes cache lines, timings with false-sharing (stride 1-8 with more than 1 thread) were varying from run to run.
Communication overhead, e.g., due to different schedules

Storing data into …

- a[0..7]
- a[8..15]
- a[16..23]
- a[24..31]

Reusing data from different threads...

- a[0,4,8]
- a[1,5,9]
- a[2,6,10]
- a[3,7,11,12,16,17,18,19,20,24,25,29,30]

Only partially fast cache access
Often slow cache coherence protocol

Communication overhead – an experiment

```
n=10000000; w=1.0/n; sum=0.0;
for (j=1;j<n;j=j+m)
{
  j_e = (j+m-1>n ? n : j+m-1);
  #pragma omp parallel private(x,sum0) 
      shared(w,sum)
  {
    sum0=0;
    #pragma omp for
    for (i=j;i<=j_e;i++)
    {
      x=w*((double)i-0.5);
      vec[i-j] = 4.0/(1.0+x*x);
    }
  }
  #pragma omp critical
  {
    sum=sum+sum0;
  }
} /*end loop j*/
pi=w*sum;
```

**Normal:**
```
#pragma omp for
for (i=0;i<16;i++)
{ /*dummy loop */
  sum0=1.0*sum0;
}
```

**Split:** with cache coherence communication !!!
```
#pragma omp for
for (i=0;i<n/2;i++)
{ /*dummy loop */
  sum0=1.0*sum0;
}
```

2 variants
Communication overhead – results from the experiment

Calculation of pi with vector chunks
Wall-clock time - the smaller the better!
Timing on NEC TX-7, with n=10,000,000

Significant performance penalties because several threads are accessing the same data in different loops

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 Intel® Thread Checker help to write correct parallel programs
Appendix – Content

- Example pi, written in C
- Example pi, written in Fortran (fixed form)
- Example pi, written in Fortran (free form)
- Example heat
  - heatc2.c – parallel version in C, with critical region (4 pages)
  - heatr.f – parallel version in Fortran, with reduction clause (4 pages)
Example pi, written in C

- pi.c – sequential code
- pi0.c – sequential code with a parallel region, verifying a team of threads
- pic2.c – parallel version with a critical region outside of the loop
- pir.c – parallel version with a reduction clause
- pir2.c – parallel version with combined parallel for
- pio.c – parallel version with ordered region
- pio2.c – parallel version with ordered execution if the number of threads is fixed

pi.c – sequential code

```c
#include <stdio.h>
#include <time.h>
#include <sys/time.h>
#ifdef _OPENMP
#  include <omp.h>
#endif
#define f(A) (4.0/(1.0+A*A))
const int n = 1000000;
int main(int argc, char** argv)
{
    int i;
    double w, x, sum, pi;
    clock_t t1, t2;
    struct timeval tv1, tv2;
    struct timezone tz;
    #ifdef _OPENMP
    double wt1, wt2;
    #endif
    gettimeofday(&tv1, &tz);
    #ifdef _OPENMP
    wt1 = omp_get_wtime();
    # endif
    t1 = clock();
    /* pi = integral [0..1] 4/(1+x**2) dx */
    w = 1.0/n;
    sum = 0.0;
    for (i=1; i<=n; i++)
    {
        x = w*((double)i-0.5);
        sum = sum + f(x);
    }
    pi = w * sum;
    t2 = clock();
    #ifdef _OPENMP
    wt2 = omp_get_wtime();
    #endif
    gettimeofday(&tv2, &tz);
    printf( "computed pi = %24.16g\n", pi);
    printf( "CPU time (clock) = %12.4g sec\n", (t2-t1)/1000000.0);
    #ifdef _OPENMP
    printf( "wall clock time (omp_get_wtime) = %12.4g sec\n",
            wt2-wt1);
    # endif
    printf( "wall clock time (gettimeofday) = %12.4g sec\n",
            (tv2.tv_sec-tv1.tv_sec) +
                (tv2.tv_usec-tv1.tv_usec)*1e-6);
    return 0;
}
```

The include and timing blocks are removed on the next slides.
### 7. Introduction to OpenMP

#### pi0.c – only verification of team of threads

- **without parallelization**

```c
--- INCLUDE BLOCK ---
#define f(A) (4.0/(1.0+A*A))
const int n = 10000000;
int main(int argc, char** argv)
{
    int i;
    double w,x,sum,pi;
    --- TIMING BLOCK A ---
    # ifdef _OPENMP
    int myrank, num_threads;
    # pragma omp parallel private(myrank,num_threads)
    {
        myrank = omp_get_thread_num();
        num_threads = omp_get_num_threads();
        printf("I am thread %2d of %2d threads\n", myrank, num_threads);
    } /* end omp parallel */
    # else
    printf("This program is not compiled with OpenMP\n");
    # endif
    --- TIMING BLOCK B ---
    /* calculate pi = integral [0..1] 4/(1+x**2) dx */
    w=1.0/n;
    sum=0.0;
    for (i=1;i<=n;i++)
    {
        x=w*((double)i-0.5);
        sum=sum+f(x);
    }
    pi=w*sum;
    --- TIMING BLOCK C ---
    return 0;
}
```

#### pic2.c – parallelization with critical region outside of the loop

```c
--- INCLUDE BLOCK ---
#define f(A) (4.0/(1.0+A*A))
const int n = 10000000;
int main(int argc, char** argv)
{
    int i;
    double w,x,sum,sum0,pi;
    --- TIMING BLOCK A ---
    --- PRINT NUM_THREADS ---
    --- TIMING BLOCK B ---
    /* pi = integral [0..1] 4/(1+x**2) dx */
    w=1.0/n;
    sum=0.0;
    #pragma omp parallel private(x,sum0), shared(w,sum)
    {
        sum0=0.0;
        # pragma omp for
        for (i=1;i<=n;i++)
        {
            x=w*((double)i-0.5);
            sum0=sum0+f(x);
        }
        # pragma omp critical
        {
            sum=sum+sum0;
        }
    } /*end omp parallel*/
    pi=w*sum;
    --- TIMING BLOCK C ---
    return 0;
    # ifdef _OPENMP
        # pragma omp parallel
        # pragma omp single
        printf("OpenMP-parallel with %1d threads\n", omp_get_num_threads());
    } /* end omp parallel */
    # endif
```

--- shortened according to advanced practical 2 ---
pir.c – parallelization with reduction clause

```c
--- INCLUDE BLOCK ---
#define f(A) (4.0/(1.0+A*A))
const int n = 10000000;
int main(int argc, char** argv)
{
    int i;
    double w,x,sum,pi;
    --- TIMING BLOCK A ---
    --- PRINT NUM_THREADS ---
    --- TIMING BLOCK B ---
    /* calculate pi = integral [0..1] 4/(1+x**2) dx */
    w=1.0/n;
    sum=0.0;
    #pragma omp parallel private(x), shared(w,sum)
    {
        #pragma omp for reduction(+:sum)
        for (i=1;i<=n;i++)
        {
            x=w*((double)i-0.5);
            sum=sum+f(x);
        }
        pi=w*sum;
    } /*end omp parallel*/
    --- TIMING BLOCK C ---
    return 0;
}
```

pir2.c – combined parallel for with reduction clause

```c
--- INCLUDE BLOCK ---
#define f(A) (4.0/(1.0+A*A))
const int n = 10000000;
int main(int argc, char** argv)
{
    int i;
    double w,x,sum,pi;
    --- TIMING BLOCK A ---
    --- PRINT NUM_THREADS ---
    --- TIMING BLOCK B ---
    /* calculate pi = integral [0..1] 4/(1+x**2) dx */
    w=1.0/n;
    sum=0.0;
    #pragma omp parallel for private(x), shared(w), reduction(+:sum)
    for (i=1;i<=n;i++)
    {
        x=w*((double)i-0.5);
        sum=sum+f(x);
    }
    /*end omp parallel for*/
    pi=w*sum;
    --- TIMING BLOCK C ---
    return 0;
}
```
C6. Introduction to OpenMP

OpenMP
Höchstleistungsrechenzentrum Stuttgart
Matthias Müller et al.

[7] Slide 143 / 132

pio.c – parallelization with ordered clause

--- INCLUDE BLOCK ---
#define f(A) (4.0/(1.0+A*A))
const int n = 10000000;
int main(int argc, char** argv)
{
    int i;
    double w,x,sum,myf,pi;
    --- TIMING BLOCK A ---
    --- PRINT NUM_THREADS ---
    --- TIMING BLOCK B ---
    /* calculate pi = integral [0..1] 4/(1+x**2) dx */
    w=1.0/n;
    sum=0.0;
    #pragma omp parallel private(x,myf), shared(w,sum)
    |
    # pragma omp for ordered
    for (i=1;i<=n;i++)
        |
        x=w*((double)i-0.5);
        myf = f(x);
        # pragma omp ordered
        |
        sum=sum+myf;
        |
    #pragma omp for ordered
    |
    sum=sum0;
    |
    #pragma omp ordered
    |
    sum=sum+sum0;
    |
    } /*end omp parallel*/
    |
    pi=w*sum;
    |
    --- TIMING BLOCK C ---
    |
    return 0;
    |

CAUTION
The sequentialization of the ordered region may cause heavy synchronization overhead

pio2.c – ordered, but only if number of threads is fixed

... double w,x,sum,sum0,pi;
/* calculate pi = integral [0..1] 4/(1+x**2) dx */
w=1.0/n;
sum=0.0;
#pragma omp parallel private(x,sum0,num_threads), shared(w,sum)
|
    sum0=0.0;
    |
    #ifdef _OPENMP
    num_threads=omp_get_num_threads();
    #else
    num_threads=1
    #endif
    |
    # pragma omp for schedule(static,(n-1)/num_threads+1)
    for (i=1;i<n;i++)
        |
        x=w*((double)i-0.5);
        sum0=sum0+f(x);
    |
    #pragma omp for ordered schedule(static)
    for (i=0;i<num_threads;i++)
        |
        #pragma omp ordered
        |
        sum=sum+sum0;
    |
    #pragma omp ordered
    |
    pi=w*sum;
    |
    ...
Example pi, written in Fortran (fixed form)

- pi.f — sequential code
- pi0.f — sequential code with a parallel region, verifying a team of threads
- pic2.f — parallel version with a critical region outside of the loop
- pir.f — parallel version with a reduction clause
- pir2.f — parallel version with combined parallel for
- pio.f — parallel version with ordered region
- pio2.f — parallel version with ordered execution if the number of threads is fixed

---

pi.f — sequential code

```fortran
program compute_pi
  implicit none
  integer n, i
  double precision w, x, sum, pi, f, a
  parameter (n=10_000_000)

  ! times using cpu_time
  real t0
  real t1
  !--unused-- include 'omp_lib.h'
  !$ double precision omp_get_wtime
  !$ double precision wa0,wt

  ! function to integrate
  f(a)=4.d0/(1.d0+a*a)
  !$ wa0=omp_get_wtime()

  call cpu_time(t0)

  ! calculate pi = integral [0..1] 4/(1+x**2) dx
  w=1.0d0/n
  sum=0.0d0
  do i=1,n
    x=w*(i-0.5d0)
    sum=sum+f(x)
  enddo
  pi=w*sum

  call cpu_time(t1)
  !$ wt1=omp_get_wtime()

  !$ write (*,'(/,a,1pg24.16)') 'computed pi = ', pi
  !$ write (*,'(/,a,1pg12.4)') 'cpu_time : ', t1-t0
  !$ write (*,'(/,a,1pg12.4)') 'omp_get_wtime:', wt1-wa0

end
```

The timing blocks are removed on the next slides
pi0.f – only verification of team of threads

— without parallelization

program compute_pi
implicit none
integer n,i
double precision w,x,sum,pi,f,a
parameter (n=10 000 000)
--- TIMING BLOCK A ---
!$ integer omp_get_thread_num, omp_get_num_threads
!$ integer myrank, num_threads
logical openmp_is_used
! function to integrate
f(a)=4.d0/(1.d0+a*a)
!$omp parallel private(myrank, num_threads)
!$ num_threads = omp_get_num_threads()
!$ write (*,*) 'I am thread',myrank, 'of',num_threads,'threads'
!$omp end parallel
openmp_is_used = .false.
!$ openmp_is_used = .true.
if (.not. openmp_is_used) then
write (*,*) 'This program is not compiled with OpenMP'
endif
--- TIMING BLOCK B ---
! calculate pi = integral [0..1] 4/(1+x**2) dx
w=1.0d0/n
sum=0.0d0
do i=1,n
x=w*(i-0.5d0)
sum=sum+f(x)
enddo
pi=w*sum
--- TIMING BLOCK C ---
end

pic2.f

parallelization with critical region outside of the loop

program compute_pi
implicit none
integer n,i
double precision w,x,sum,pi,f,a
parameter (n=10 000 000)
--- TIMING BLOCK A ---
! function to integrate
f(a)=4.d0/(1.d0+a*a)
--- PRINT NUM_THREADS
--- TIMING BLOCK B ---
! calculate pi = integral [0..1] 4/(1+x**2) dx
w=1.0d0/n
sum=0.0d0
!$OMP PARALLEL PRIVATE(x,sum0), SHARED(w,sum)
sum0=0.0d0
!$OMP DO
do i=1,n
x=w*(i-0.5d0)
sum=sum+f(x)
enddo
!$OMP END DO
!$OMP CRITICAL
sum=sum+sum0
!$OMP END CRITICAL
!$OMP END PARALLEL
pi=w*sum
--- TIMING BLOCK C ---
end

shortened according to advanced practical 2
pir.f – parallelization with reduction clause

```fortran
program compute_pi
  implicit none
  integer n, i
  double precision w, x, sum, pi, f, a
parameter (n=10000000)
  ! function to integrate
  f(a)=4.0d0/(1.0d0+a*a)
  ! calculate pi = integral [0..1] 4/(1+x**2) dx
  w=1.0d0/n
  sum=0.0d0
  !$OMP PARALLEL PRIVATE(x), SHARED(w,sum)
  !$OMP DO REDUCTION(+:sum)
  do i=1,n
    x=w*(i-0.5d0)
    sum=sum+f(x)
  enddo
  !$OMP END DO
  !$OMP END PARALLEL
  pi=w*sum
end```

pir2.f – combined parallel do with reduction clause

```fortran
program compute_pi
  implicit none
  integer n, i
  double precision w, x, sum, pi, f, a
parameter (n=10000000)
  ! function to integrate
  f(a)=4.0d0/(1.0d0+a*a)
  ! calculate pi = integral [0..1] 4/(1+x**2) dx
  w=1.0d0/n
  sum=0.0d0
  !$OMP PARALLEL DO PRIVATE(x), SHARED(w), REDUCTION(+:sum)
  do i=1,n
    x=w*(i-0.5d0)
    sum=sum+f(x)
  enddo
  !$OMP END PARALLEL DO
  pi=w*sum
end```
program compute_pi
implicit none
integer n, i
double precision w, x, sum, myf, pi, f, a
parameter (n=10 000 000)
--- TIMING BLOCK A ---
! function to integrate
f(a)=4.d0/(1.d0+a**2)
--- PRINT NUM_THREADS ---
--- TIMING BLOCK B ---
! calculate pi = integral [0..1] 4/(1+x**2) dx
w=1.d0/n
sum=0.0d0
!$OMP PARALLEL PRIVATE(x, myf), SHARED(w, sum)
!$OMP DO ORDERED
do i=1,n
x=w*(i-0.5d0)
myf=f(x)
!$OMP ORDERED
  sum=sum+myf
!$OMP END ORDERED
endo
!$OMP END DO
!$OMP END PARALLEL
pi=w*sum
--- TIMING BLOCK C ---
end

CAUTION
The sequentialization of the ordered region may cause heavy synchronization overhead

pio2.f – ordered, but only if number of threads is fixed

!$omp include 'omp_lib.h'
!$omp integer omp_get_num_threads
!$omp double precision w, x, sum, sum0, pi, f, a
...!
!$omp do schedule(static,(n-1)/omp_get_num_threads+1)
do i=1,n
  x=w*(i-0.5d0)
  myf=f(x)
  sum0=sum0+myf
enddo
!$omp end do
!$omp end parallel
pi=w*sum

CAUTION
1. Only if the number of threads is fixed AND if the floating point rounding is the same.
then the result is the same on two different platforms and any repetition of this program.
2. This program cannot be verified with Assure because it has to call omp_get_num_threads().
3. To use Assure, the second loop must be substituted by the critical region as shown in pic2.f
Example pi, written in Fortran (free form)

- pi.f90 – sequential code
- pi0.f90 – sequential code with a parallel region, verifying a team of threads
- pic2.f90 – parallel version with a critical region outside of the loop
- pir.f90 – parallel version with a reduction clause
- pir2.f90 – parallel version with combined parallel for
- pio.f90 – parallel version with ordered region
- pio2.f90 – parallel version with ordered execution if the number of threads is fixed
pi0.f90 – only verification of team of threads
– without parallelization

program compute_pi
  implicit none
  integer i
  integer, parameter :: n=10000000
  real(kind=8) w,x,sum,pi,f,a
  !$ integer omp_get_thread_num, omp_get_num_threads
  !$ integer myrank, num_threads
  logical openmp_is_used
  ! function to integrate
  f(a)=4.0_8/(1.0_8+a*a)
  !$omp parallel private(myrank, num_threads)
  !$  myrank = omp_get_thread_num()
  !$  num_threads = omp_get_num_threads()
  !$  if (.not. openmp_is_used) then
  !$omp parallel private(x,sum0), shared(w,sum)
  !$omp do
  do i=1,n
    x=w*(i-0.5_8)
    sum0=sum0+f(x)
  enddo
  !$omp end do
  !$omp critical
  pi=sum+sum0
  !$omp end critical
  !$omp end parallel
  pi=w*sum
end program compute_pi

pic2.f90
parallelization with critical region outside of the loop

program compute_pi
  implicit none
  integer i
  integer, parameter :: n=10000000
  real(kind=8) w,x,sum,pi,f,a
  ! function to integrate
  f(a)=4.0_8/(1.0_8+a*a)
  !$omp parallel private(x,sum0), shared(w,sum)
  !$omp do
  do i=1,n
    x=w*(i-0.5_8)
    sum0=sum0+f(x)
  enddo
  !$omp end do
  !$omp critical
  pi=sum+sum0
  !$omp end critical
  !$omp end parallel
  pi=w*sum
end program compute_pi
pir.f90 – parallelization with reduction clause

```fortran
program compute_pi
  implicit none
  integer i
  integer, parameter :: n=10000000
  real(kind=8) w,x,sum,pi,f,a
  ! function to integrate
  f(a)=4.0_8/(1.0_8+a*a)
  --- PRINT NUM_THREADS ---
  ! calculate pi = integral [0..1] 4/(1+x**2) dx
  w=1.0_8/n
  sum=0.0_8
  !$OMP PARALLEL PRIVATE(x), SHARED(w,sum)
  !$OMP DO REDUCTION(+:sum)
  do i=1,n
    x=w*(i-0.5_8)
    sum=sum+f(x)
  enddo
  !$OMP END DO
  !$OMP END PARALLEL
  pi=w*sum
  --- TIMING BLOCK C ---
end program compute_pi
```

pir2.f90 – combined parallel do with reduction clause

```fortran
program compute_pi
  implicit none
  integer i
  integer, parameter :: n=10000000
  real(kind=8) w,x,sum,pi,f,a
  ! function to integrate
  f(a)=4.0_8/(1.0_8+a*a)
  --- PRINT NUM_THREADS ---
  ! calculate pi = integral [0..1] 4/(1+x**2) dx
  w=1.0_8/n
  sum=0.0_8
  !$OMP PARALLEL DO PRIVATE(x), SHARED(w,sum), REDUCTION(+:sum)
  do i=1,n
    x=w*(i-0.5_8)
    sum=sum+f(x)
  enddo
  !$OMP END PARALLEL DO
  pi=w*sum
  --- TIMING BLOCK C ---
end program compute_pi
```
### pio.f90 – parallelization with ordered clause

```fortran
program compute_pi
  implicit none
  -- TIMING BLOCK A --
  integer i
  integer, parameter :: n=1000000
  real(kind=8) w,x,sum,myf,pi,f,a
  ! function to integrate
  f(a)=4.0_8/(1.0_8+a*a)
  -- PRINT NUM_THREADS --
  -- TIMING BLOCK B --
  ! calculate pi = integral [0..1] 4/(1+x**2) dx
  w=1.0_8/n
  sum=0.0_8
  !$OMP PARALLEL PRIVATE(x,myf), SHARED(w,sum)
  !$OMP DO ORDERED
  do i=1,n
    x=w*(i-0.5_8)
    myf=f(x)
    !$OMP ORDERED
    sum=sum+myf
  !$OMP END ORDERED
  enddo
  !$OMP END PARALLEL
  pi=w*sum
  -- TIMING BLOCK C --
end program compute_pi
```

**CAUTION**

The sequentialization of the ordered region may cause heavy synchronization overhead.

### pio2.f90 – ordered, but only if number of threads is fixed

```fortran
!-- unused -- use omp_lib
!$ integer omp_get_num_threads
real(kind=8) w,x,sum,sum0,pi,f,a
...
! calculate pi = integral [0..1] 4/(1+x**2) dx
w=1.0_8/n
sum=0.0_8
!$OMP PARALLEL PRIVATE(x,sum0,num_threads), SHARED(w,sum)
sum0=0.0_8
num_threads=1
!$ num_threads=omp_get_num_threads()
!$OMP DO SCHEDULE(STATIC,(n-1)/num_threads+1)
  do i=1,n
    x=w*(i-0.5_8)
    myf=f(x)
  enddo
!$OMP END DO
!$OMP END PARALLEL
```

**CAUTION**

1. Only if the number of threads is fixed AND if the floating point rounding is the same, then the result is the same on any two different platforms and any repetition of this program.
2. This program cannot be verified with Assure because it has to call `omp_get_num_threads()`.
3. To use Assure, the second loop must be substituted by the critical region as shown in pic2.f90.
heatc2.c – Parallelization of main loop and critical region
(page 1 of 4) – declarations

#include <stdio.h>
#include <sys/time.h>
#ifdef _OPENMP
    # include <omp.h>
#endif
#define min(A,B) ((A) < (B) ? (A) : (B))
#define max(A,B) ((A) > (B) ? (A) : (B))
#define imax 20
#define kmax 11
#define itmax 20000
void heatpr(double phi[imax+1][kmax+1]);

int main()
{
    double eps = 1.0e-08;
    double phi[imax+1][kmax+1], phin[imax][kmax];
    double dx,dy,dx2,dy2,dx2i,dy2i,dt,dphi,dphimax,dphimax0;
    int i,k,it;
    struct timeval tv1,tv2; struct timezone tz;
    #ifdef _OPENMP
        double wt1,wt2;
    # endif
    #ifdef _OPENMP
        #pragma omp parallel
        {
            #pragma omp single
            printf("OpenMP-parallel with %1d threads\n", omp_get_num_threads());
        } /* end omp parallel */
    # endif
    #ifdef _OPENMP
        #pragma omp parallel
        {
            #pragma omp for
            for (k=0;k<kmax;k++)
            { for (i=1;i<imax;i++)
                { phi[i][k]=0.0;
                }
            }
        }/*end omp parallel*/
    # pragma omp parallel private(i,k) shared(phi)
    {
        #pragma omp for
        for (i=0;i<=imax;i++)
        { phi[i][kmax]=1.0;
        }
    }/*end omp parallel*/
    #pragma omp for
    for (k=1;k<kmax;k++)
    { phi[0][k]=phi[0][k-1]+dx;
        phi[imax][k]=phi[imax][k-1]+dx;
    }
    printf("\nHeat Conduction 2d\n");
    printf("\ndx = %12.4g, dy = %12.4g, dt = %12.4g, eps = %12.4g\n", dx,dy,dt,eps);
    heatpr(phi);
}

heatc2.c (page 2 of 4) – initialization
dx=1.0/kmax; dy=1.0/imax;
dx2=dx*dx; dy2=dy*dy;
dx2i=1.0/dx2; dy2i=1.0/dy2;
dt=min(dx2,dy2)/4.0;
/* start values 0.d0 */
# pragma omp parallel private(i,k) shared(phi)
{
    #pragma omp for
    for (k=0;k<kmax;k++)
    { for (i=1;i<imax;i++)
        { phi[i][k]=0.0;
        }
    }
    /* start values 1.d0 */
    #pragma omp for
    for (i=0;i<=imax;i++)
    { phi[i][kmax]=1.0;
    }
}/*end omp parallel*/
/* start values dx */
phi[0][0]=0.0;
phi[imax][0]=0.0;
for (k=1;k<kmax;k++)
{ phi[0][k]=phi[0][k-1]+dx;
    phi[imax][k]=phi[imax][k-1]+dx;
}
printf("\nHeat Conduction 2d\n");
printf("\ndx = %12.4g, dy = %12.4g, dt = %12.4g, eps = %12.4g\n", dx,dy,dt,eps);
heatpr(phi);
heatc2.c (page 3 of 4) – time step integration

```c
gettimeofday(&tv1, &tz);
#ifdef _OPENMP
  wt1=omp_get_wtime();
#endif
/* iteration */
for (it=1;it<=itmax;it++)
{
  dphimax=0.;
#pragma omp parallel private(i,k,dphi,dphimax0) \
    shared(phi,phin,dx2i,dy2i,dt,dphimax)
  {
    dphimax0=dphimax;
#pragma omp for
    for (k=1;k<=kmax;k++)
      for (i=1;i<=imax;i++)
      { dphi=(phi[i+1][k]+phi[i-1][k]-2.*phi[i][k])*dy2i
        +(phi[i][k+1]+phi[i][k-1]-2.*phi[i][k])*dx2i;
        dphi=dphi*dt;
        dphimax0=max(dphimax0,dphi);
      }
  }
#pragma omp critical
  {   dphimax=max(dphimax,dphimax0);
  }
// save values */
#pragma omp for
for (k=1;k<=kmax;k++)
  for (i=1;i<=imax;i++)
  { phi[i][k]=phin[i][k];
  }
/*end omp parallel*/
if(dphimax<eps) break;
}
```

Caution:
In C, phi and phin are contiguous in the last index [k]. Therefore the k-loop should be the inner loop!

heatc2.c (page 4 of 4) – wrap up

```c
#ifdef _OPENMP
  wt2=omp_get_wtime();
#endif
gmtimeofday(&tv2, &tz);
printf("phi after %d iterations\n",it);
heatpr(phi);
#ifdef _OPENMP
  printf( "wall clock time (omp_get_wtime) = %12.4g sec\n", wt2-wt1 );
#endif
printf( "wall clock time (gettimeofday)  = %12.4g sec\n", (tv2.tv_sec-tv1.tv_sec) + (tv2.tv_usec-tv1.tv_usec)*1e-6 );
}

void heatpr(double phi[imax+1][kmax+1])
{ int i,k,kl,kk,kkk;
  kl=6;
  kkk=kl-1;
  for (k=0;k<kmax;k++)
    printf("\ncolumns %5d to %5d\n",k,k+kkk);
  for (i=0;i<imax;i++)
    printf("%5d %s",i);
  printf("\n");
  for (k=0;k<kmax;k++)
    for (kk=0;kk<kkk;kk++)
      printf("%#12.4g",phi[i][k+kk]);
  printf("\n");
}
return;
```
heatr2_x.f — Parallelization of main loop with reduction clause
(page 1 of 4) — declarations

```fortran
program heat
  implicit none
  integer i, k, it, imax, kmax, itmax
  c using reduction
  parameter (imax=20, kmax=11)
  parameter (itmax=20000)
  double precision eps
  parameter (eps=1.d-08)
  double precision phi(0:imax,0:kmax),phin(1:imax-1,1:kmax-1)
  double precision dx, dy, dx2, dy2, dx2i, dy2i, dt, dphi, dphimax
  ! times using cpu_time
  real t0
  real t1

  !--unused-- include 'omp_lib.h'
  !$ integer omp_get_num_threads
  !$ double precision omp_get_wtime
  !$ double precision wt0, wt1
  $
  !$omp parallel
  !$omp single
  !$    write(*,*)'OpenMP-parallel with',omp_get_num_threads(),'threads'
  !$omp end single
  !$omp end parallel
  c
  dx=1.d0/kmax
  dy=1.d0/imax
  dx2=dx*dx
  dy2=dy*dy
  dx2i=1.d0/dx2
  dy2i=1.d0/dy2
  dt=min(dx2,dy2)/4.d0
```

heatr2_x.f (page 2 of 4) — initialization

```fortran
!$omp parallel private(i,k), shared(phi)
c start values 0.d0
!$omp do
do k=0,kmax-1
do i=1,imax-1
  phi(i,k)=0.d0
endo
dendo
!$omp end do
c start values 1.d0
!$omp do
do i=0,imax
  phi(i,kmax)=1.d0
endo
!$omp end do
!$omp end parallel
c
  dx=1.d0/kmax
  dy=1.d0/imax
  dx2=dx*dx
  dy2=dy*dy
  dx2i=1.d0/dx2
  dy2i=1.d0/dy2
  dt=min(dx2,dy2)/4.d0

c print array
write ('(/,a)')
'Heat Conduction 2d'
write ('(/,4(a,1pg12.4))')
dx =', dx,' dt=', dt,' eps=', eps
call heatpr(phi,imax,kmax)
```
heatr2_x.f (page 3 of 4) – time step integration

```fortran
!$ wt0=omp_get_wtime()
call cpu_time(t0)
c iteration
    do it=1,itmax
dphimax=0.
!$OMP PARALLEL PRIVATE(i,k,dphi), SHARED(phi,phin,dx2i,dy2i,dt,dphimax)
!$OMP DO REDUCTION(max:dphimax)
do k=1,kmax-1
    do i=1,imax-1
        dphi=(phi(i+1,k)+phi(i-1,k)-2.*phi(i,k))*dy2i
        + (phi(i,k+1)+phi(i,k-1)-2.*phi(i,k))*dx2i
dphi=dphi*dt
dphimax=max(dphimax,dphi)
phin(i,k)=phi(i,k)+dphi
enddo
enddo
!$OMP END DO
!$OMP END PARALLEL
if(dphimax.lt.eps) goto 10
enddo
10    continue
```

In Fortran, phi and phin are contiguous in the first index [i]. Therefore the i-loop should be the inner loop! This optimization is done in all heatr..._x.f versions.

heatr2_x.f (page 4 of 4) – wrap up

```fortran
!$ wt1=omp_get_wtime()
call cpu_time(t1)
!$ write (*,'(/,a,1pg12.4)') 'omp_get_wtime:', wt1-wt0
!
c print array
    write (*,'(/,a,i6,a)') 'phi after',it,' iterations'
    write (*,'(/,a,1pg12.4)') 'cpu_time  :   ', t1-t0
!$ write (*,'(/,a,1pg12.4)') 'omp_get_wtime:', wt1-wt0
!
c subroutine heatpr(phi,imax,kmax)
double precision phi(0:imax,0:kmax)
c
kl=6
kkk=kl-1
do k=0,kmax,kk
    if(k+kkk.gt.kmax) kkk=kmax-k
    write (*,'(/,a,1pg12.4)') 'columns',k, to',k+kkk
do i=0,imax
    write (*,'(/,a,6(1pg12.4))') i,(phi(i,k+k)),kk=0,kkk
enddo
enddo
return
end
```

In Fortran, phi and phin are contiguous in the first index [i]. Therefore the i-loop should be the inner loop! This optimization is done in all heatr..._x.f versions.