Introduction to OpenMP

Rolf Rabenseifner
rabenseifner@hlrs.de
www.hlrs.de/people/rabenseifner/

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

Version 13, Aug 21, 2020 (for OpenMP 3.1 and older)

 ● REC → online

Outline

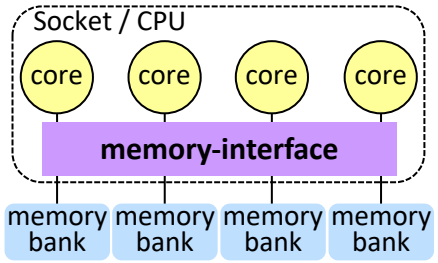
OpenMP Overview: What is OpenMP?

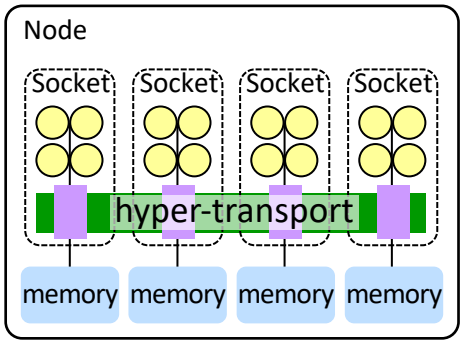
- OpenMP is a standard programming model for shared memory parallel programming
- Portable across all shared-memory architectures
- It allows incremental parallelization
- Compiler based extensions to existing programming languages
  - mainly by directives
  - a few library routines
- Fortran and C/C++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

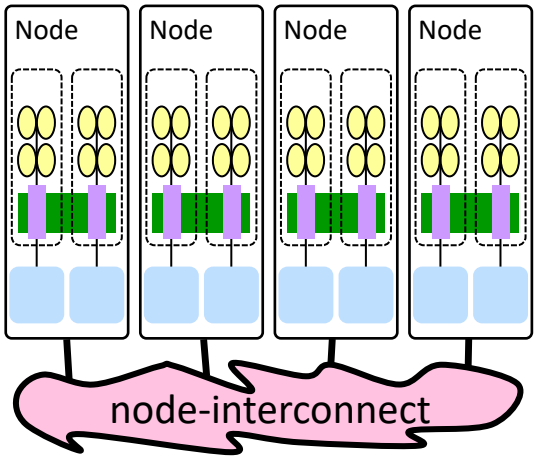
Parallel hardware architectures

shared memory

distributed memory







Socket/CPU

→ **memory interface**

UMA (uniform memory access)
SMP (symmetric multi-processing)
All cores connected to all memory banks with same speed

Parallel execution streams on each core, e.g.,
x[0 … 999] = … on 1st core
x[1000 … 1999] = … on 2nd core
x[2000 … 2999] = … on 3rd core
…

Node

→ **hyper-transport**

ccNUMA (cache-coherent non-uniform memory access)
#CPUs x memory bandwidth
Shared memory programming is possible
Performance problems:

- Each **parallel execution stream** should mainly access the memory of **its** CPU
  → **First-touch** strategy is needed to minimize remote memory access
- Threads should be **pinned** to the physical sockets

Cluster

→ **node-interconnect**

NUMA (non-uniform memory access)
!! fast access only on its own memory **!!**
Many programming options:

- Shared memory / symmetric multi-processing inside of each node
- distributed memory parallelization on the node interconnect
- **Or simply one MPI process on each core**

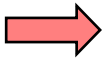
Shared memory programming with OpenMP

MPI works everywhere

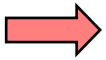
Slide, courtesy to Claudia Blaas-Schenner

Shared Memory Directives – OpenMP, I.

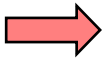
```
Real :: A(n,m), B(n,m)

!$OMP PARALLEL DO
do j = 2, m-1
  do i = 2, n-1
    B(i,j) =  ... A(i,j)
              ... A(i-1,j) ... A(i+1,j)
              ... A(i,j-1) ... A(i,j+1)
  end do
end do
!$OMP END PARALLEL DO
```



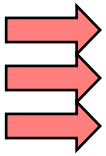
Data definition



Loop over y-dimension

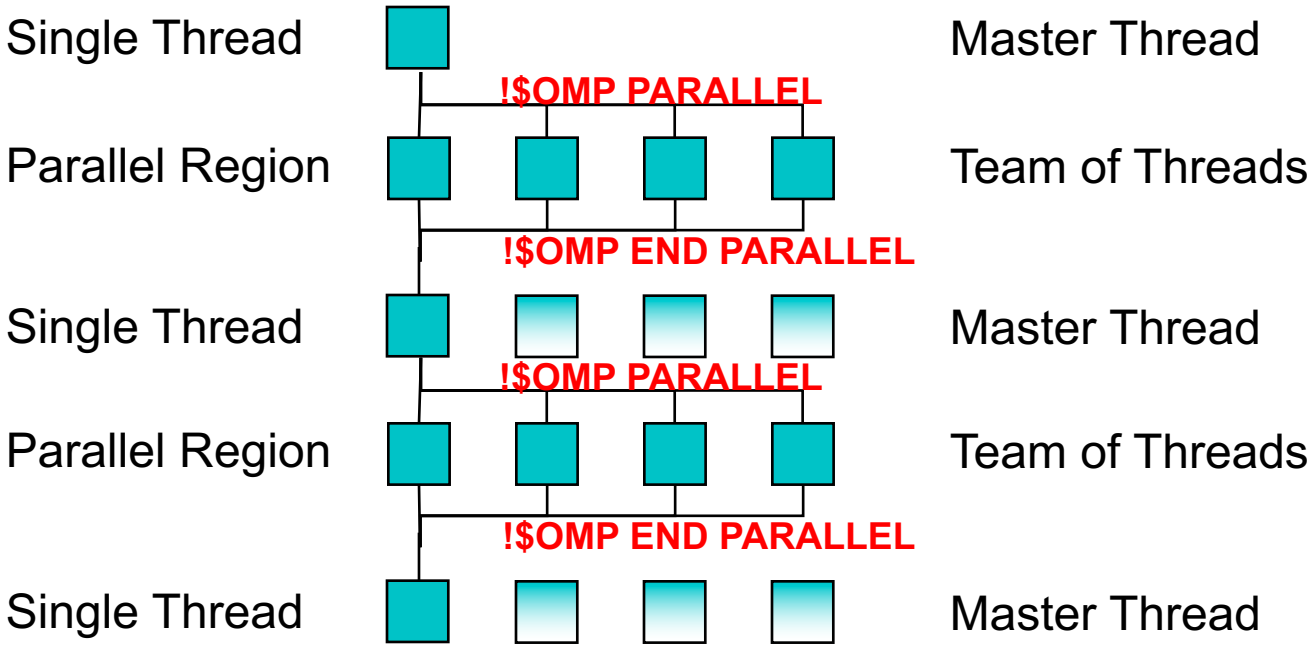


Vectorizable loop over x-dimension

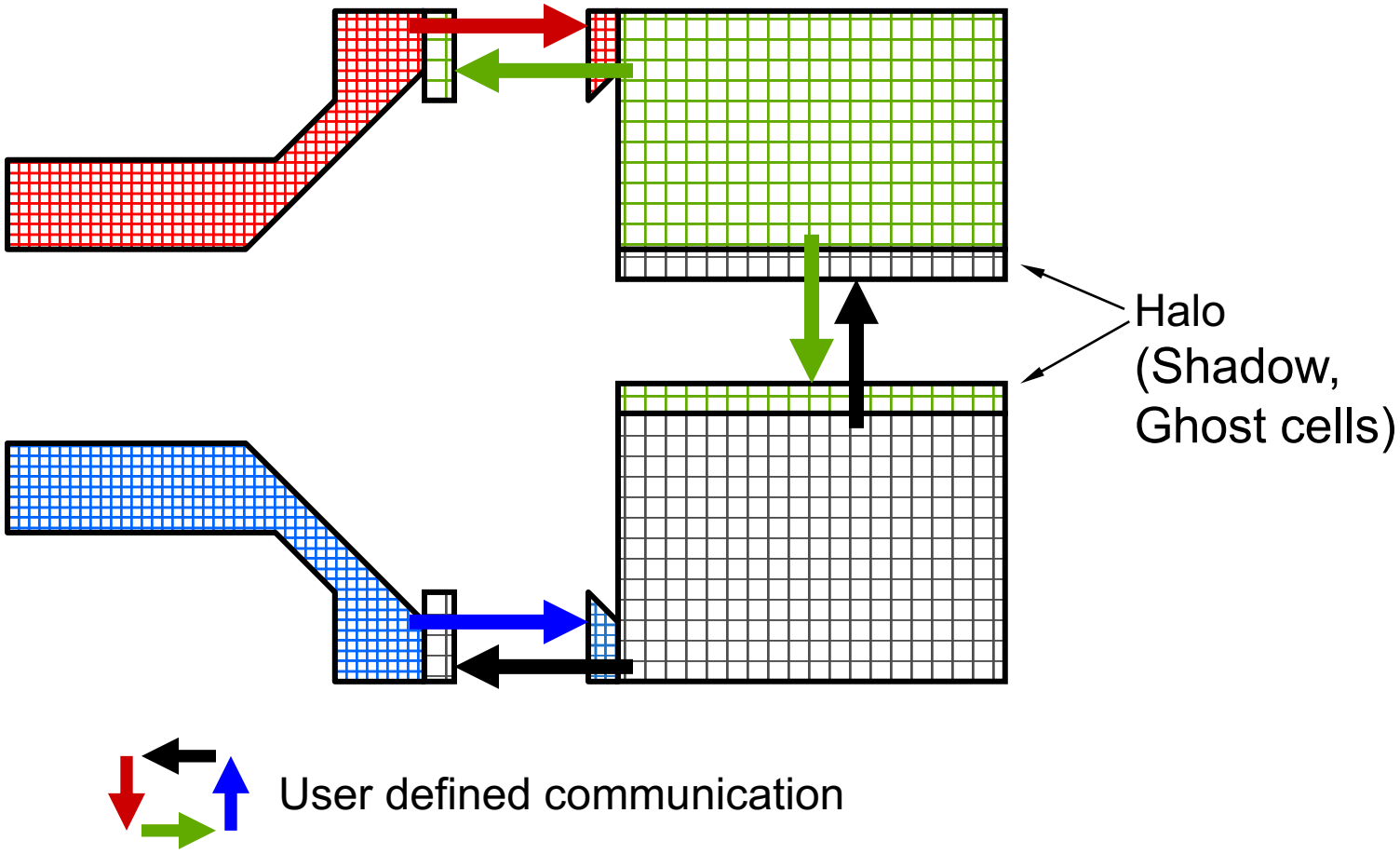


Calculate B,
using upper and lower,
left and right value of A

Shared Memory Directives – OpenMP, II.

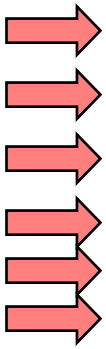


Additional Halo Cells – MPI, I.



Message Passing – MPI, II.

```
Call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
Call MPI_Comm_rank(MPI_COMM_WORLD, myrank, ierror)
m1 = (m+size-1)/size;   ja=1+m1*myrank;   je=min(m1*(myrank+1), m)
jax=ja-1;  jex=je+1   // extended boundary with halo

Call MPI_Send(.....)  ! - sending the boundary data to the neighbors
Call MPI_Recv(.....)  ! - receiving from the neighbors,
                      !   storing into the halo cells

Real :: A(n, jax:jex), B(n, jax:jex)
do j = max(2,ja), min(m-1,je)
  do i = 2, n-1
    B(i,j) =  ... A(i,j)
              ... A(i-1,j) ... A(i+1,j)
              ... A(i,j-1) ... A(i,j+1)
  end do
end do
```



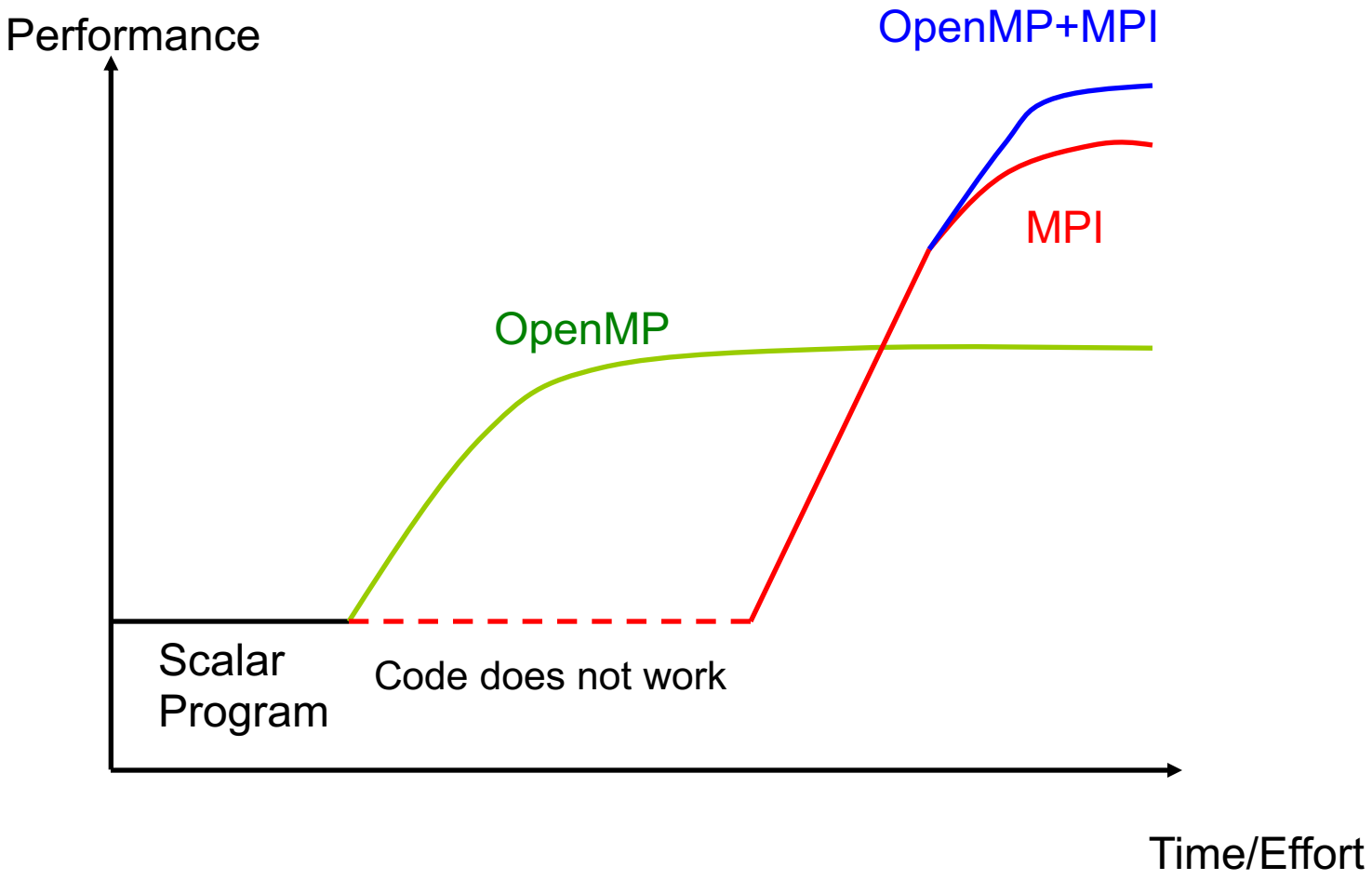
Data definition

Loop over y-dimension

Vectorizable loop over x-dimension

Calculate B,
using upper and lower,
left and right value of A

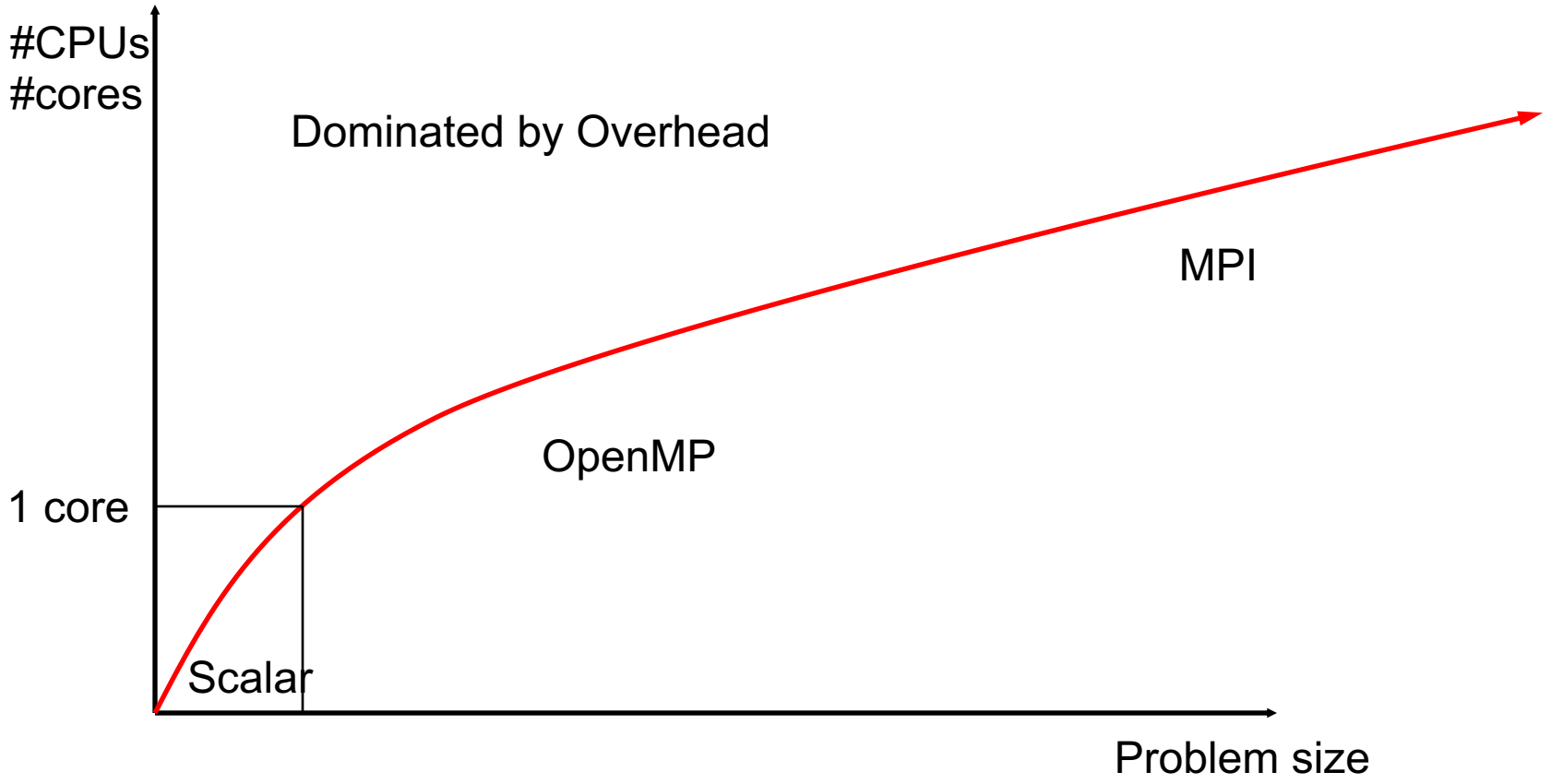
Motivation: Why should I use OpenMP?



Further Motivation to use OpenMP

- OpenMP is the easiest approach to multi-threaded programming
- Multi-threading is needed to exploit modern hardware platforms:
  - Several CPUs together within one ccNUMA node
  - Several cores per CPU
  - Hyperthreading

Where should I use OpenMP?



Simple OpenMP Program

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

Serial Program:

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

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

Parallel Program:

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

Speedup, Efficiency, Scaleup, and Weak Scaling

- Definition: T(p,N) = **time** to solve **problem of total size N** on **p processors**

- Parallel speedup: S(p,N) = T(1,N) / T(p,N)
  compute **same problem** with more processors in **shorter time**

- Parallel Efficiency: E(p,N) = S(p,N) / p

- Scaleup: Sc(p,N) = N / n with T(1,n) = T(p,N)
  compute **larger problem** with more processors in **same time**

- Weak scaling: T(p, p•n) / T(1,n) is reported,
  i.e., problem size per process (n) is fixed

Three different ways of reporting the success

- Problems:
  - Absolute MFLOPS rate **/** hardware peak performance?
  - Super-scalar speedup: S(p,N)>p, e.g., due to cache*) usage for large p:
    - **T(1,N) may be based on a huge number of N data elements in the memory in the one process, whereas**
    - **T(p,N) may be based on *cache based execution* due to only N/p data elements per process**
  - S(p,N) close to **p** or far less? → see Amdahl's Law on next slide

*) Faster memory for temporary copies of recently used data

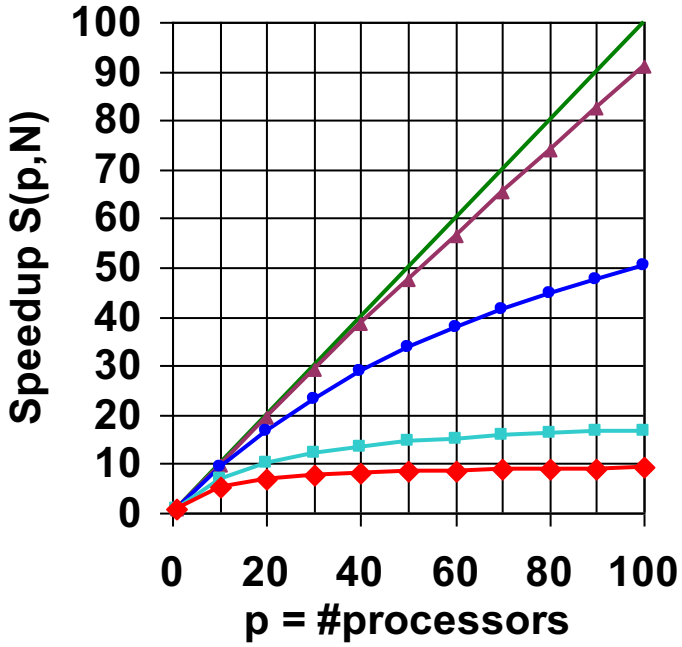
Amdahl's Law

$T(p,N) = f \cdot T(1,N) + (1-f) \cdot T(1,N) / p$

f ... sequential part of code that can not be done in parallel

$S(p,N) = T(1,N) / T(p,N) = 1 / (f + (1-f) / p)$

For p —> infinity, speedup is limited by $S(p,N) < 1 / f$



S(p,N) = p (ideal speedup)

f=0.1% => S(p,N) < 1000

f= 1% => S(p,N) < 100

f= 5% => S(p,N) < 20

f= 10% => S(p,N) < 10

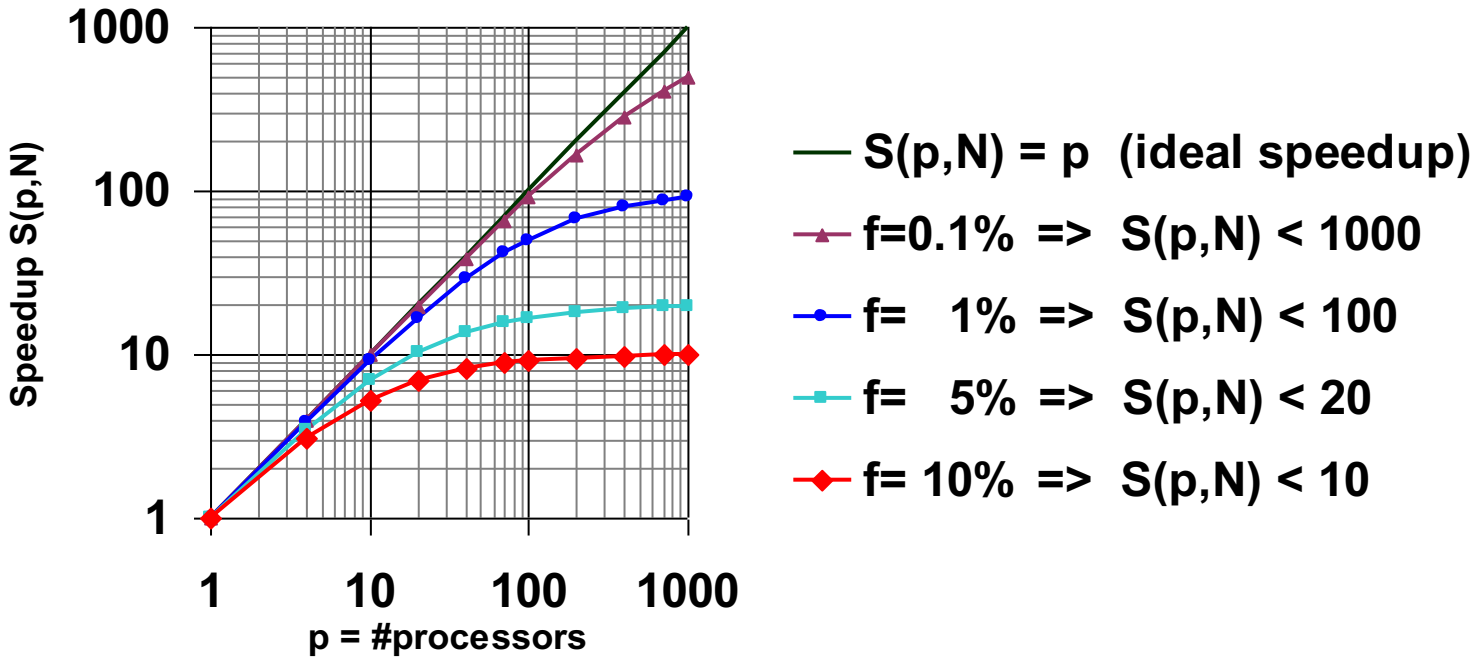
Amdahl's Law (double-logarithmic)

$T(p,N) = f \cdot T(1,N) + (1-f) \cdot T(1,N) / p$

f ... sequential part of code that can not be done in parallel

$S(p,N) = T(1,N) / T(p,N) = 1 / (f + (1-f) / p)$

For p —> infinity, speedup is limited by $S(p,N) < 1 / f$



Who owns OpenMP? - OpenMP Architecture Review Board

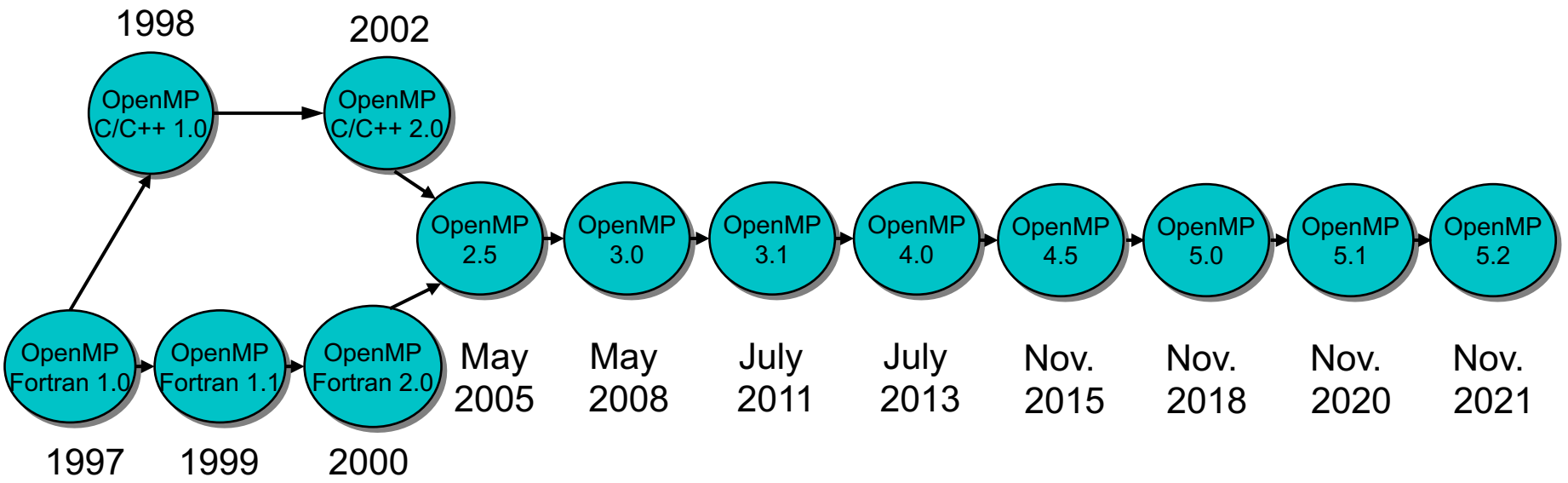
AMD[1)] (Greg Rodgers)
Argonne National Laboratory (Kalyan Kumaran)
ARM (Graham Hunter)
ASC/Lawrence Livermore NL[1)] (Bronis R. de Supinski)
Barcelona Supercomputing Center (Xavier Martorell)
Brookhaven National Laboratory (Vivek Kale)
cOMPunity[1)] (Yonghong Yan)
CRAY[1)], a HPE company (Deepak Eachempati)
Edinburgh Parallel Computing Centre[1)] (Mark Bull)
Fujitsu[1)] (Naoki Sueyasu)
IBM[1)] (Kelvin Li)
INRIA (Olivier Aumage)
Intel[1)] (Xinmin Tian)
Lawrence Berkeley National Laboratory (Helen He)
Leibniz Supercomputing Centre (Volker Weinberg)
Los Alamos National Laboratory (Jamal Mohd-Yusof)
Maui HPC Center (Alice Koniges)
Mentor Graphics, a Siemens Business (Catherine Moore)
Micron (Randy Meyer)
NASA[1)] (Henry Jin)
NEC[1)] (Shin-ichi Okano)
NVIDIA (Jeff Larkin)
Oak Ridge National Laboratory (Oscar Hernandez)
RWTH Aachen University[1)] (Dieter an Mey)
Sandia National Laboratory (Stephen Olivier)
Stony Brook University (Dr. Barbara Chapman)
SUSE (Michael Matz)
Texas Advanced Computing Center (Kent Milfeld)
The University of Manchester (Antoniu Pop)
University of Basel (Florina Ciorba)
University of Bristol (Simon McIntosh-Smith)
University of Delaware (Sunita Chandrasekaran)
University of Tennessee (Piotr Luszczek)

[1)] Already 2008 member of the ARB

From: https://www.openmp.org/about/members/
Last update: Aug 21, 2020

2020: 11 (=**30%**) companies + 22 (**70%**) HPC centers
2008: 11 (=**70%**) companies + 5 (**30%**) HPC centers

OpenMP Release History



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 3.0
  - task concept added
  - new features for loop worksharing
- OpenMP 3.1
  - `final` and `mergeable` clauses for `task` construct
  - `taskyield` construct to allow user-defined task switching points
  - min and max reduction in C/C++
  - OMP_NUM_THREADS can handle a list for nested regions
  - OMP_PROC_BIND to bind threads to processors
- OpenMP 4.0
  - Thread affinity and OMP_PLACES
  - SIMD support
  - Support for accelerators
  - Tasking extensions
- OpenMP 4.5
  - `taskloop`
- OpenMP 5.0
  - `reduction` clause for `taskloop` and `taskloop simd`
  - default loop schedule **changed** from **`monotonic`** to **`nonmonotonic`** → enables work stealing for dynamic & guided

OpenMP Information

- OpenMP Homepage: http://www.openmp.org/
- OpenMP user group: http://www.compunity.org/

- Barbara Chapman, Gabriele Jost, and Ruud van der Pas:
  Using OpenMP.
  MIT Press, 2008, ISBN-13: 978-0-262-53302-7.
- Georg Hager, Gerhard Wellein:
  Introduction to High Performance Computing for Scientists and Engineers.
  CRC Press, 2010, 256p, ISBN13: 978-1-439-81192-4
- R.Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon:
  Parallel programming in OpenMP.
  Academic Press, San Diego, USA, 2000, ISBN 1-55860-671-8

Outline — Programming and Execution Model

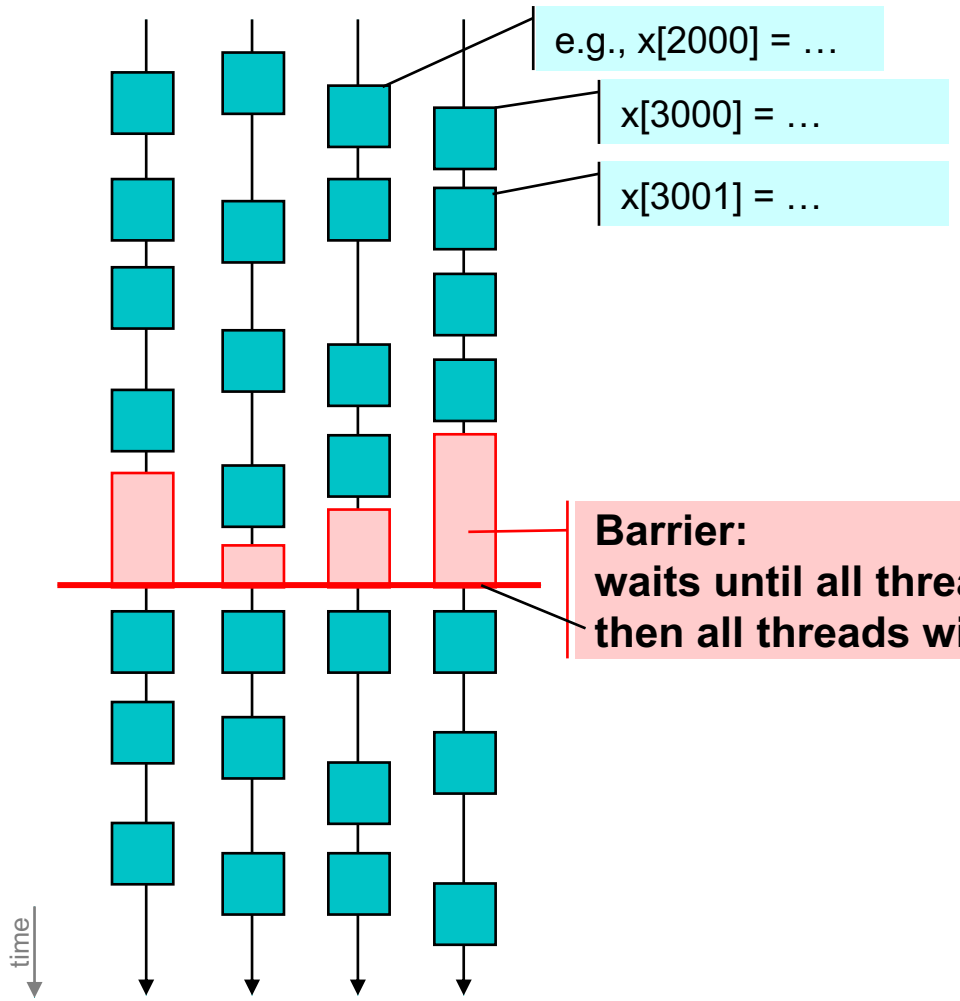
- Introduction into OpenMP
- **Programming and Execution Model**
  - **Parallel regions: team of threads**
  - **Syntax**
  - **Data environment (part 1)**
  - **Environment variables**
  - **Runtime library routines**
  - **Exercise 1:** Parallel region / library calls / privat & shared variables
- Worksharing directives
  - Which thread executes which statement or operation?
  - Exercise 2a: Pi
  - Tasks
  - Synchronization constructs, e.g., critical regions
  - Nesting and Binding
  - Exercise 2b: Pi
- Data environment and combined constructs
  - Private and shared variables, Reduction clause
  - Combined parallel worksharing 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 through barriers (→ next slide).
- 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.

Barrier

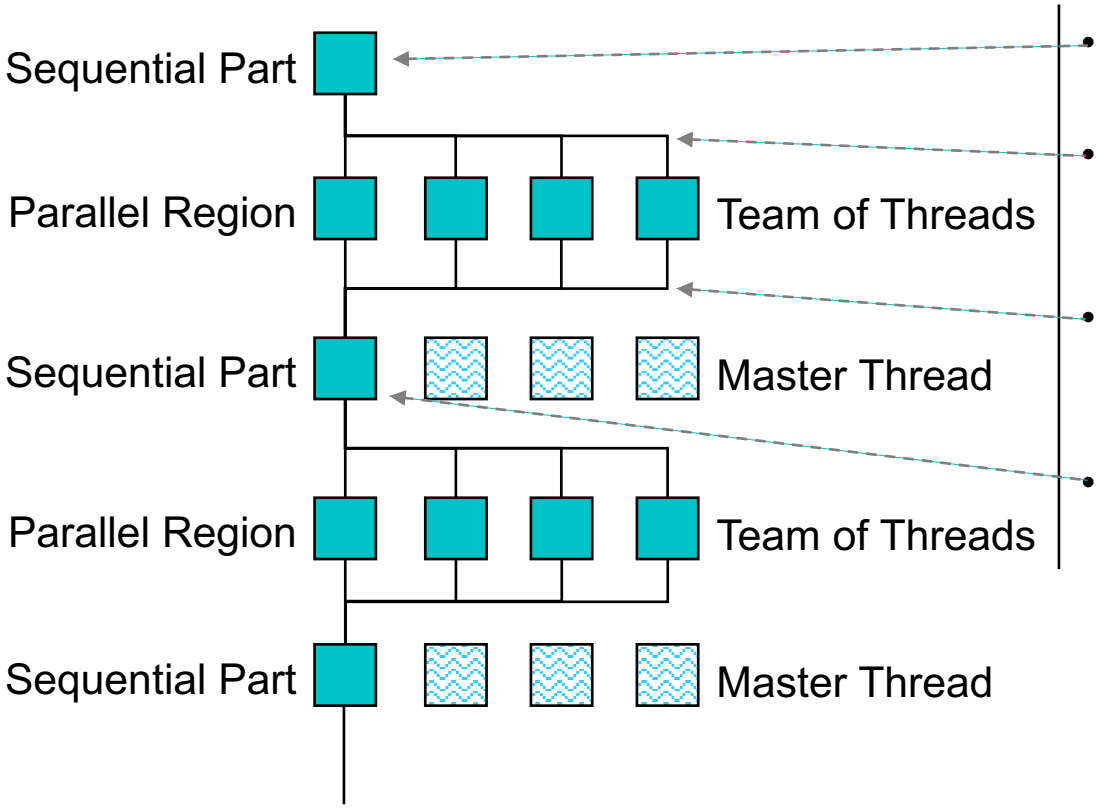
threads = parallel execution streams



Barrier:
waits until all threads have called the barrier, and then all threads will continue with the next operation

OpenMP Execution Model

Fork-join model of parallel execution



- Begin execution as a single thread/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

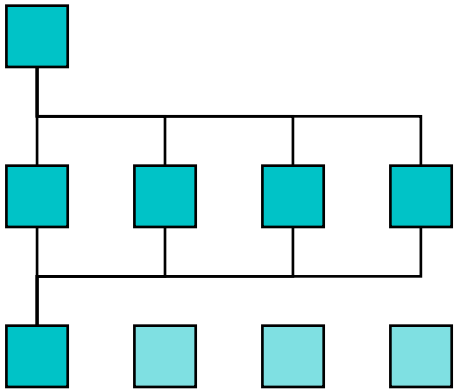
OpenMP Parallel Region Construct

Fortran

Fortran:

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

A block of executable statements with a single entry at the top and a single exit at the bottom or an OpenMP construct.

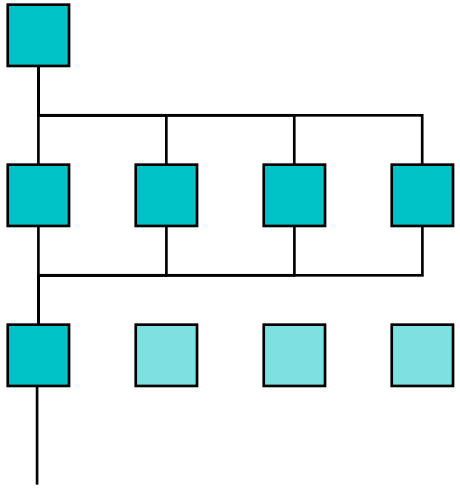


C/C++

C / C++:

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

One executable statement, **possibly compound** { …; …; … }, with a single entry at the top and a single exit at the bottom or an OpenMP construct.



OpenMP Parallel Region Construct Syntax

- Block of code to be executed by multiple threads in parallel.
  Each thread executes the **same code in a replicated way**!

Fortran

- Fortran:

  `!$OMP PARALLEL` [*clause* [[,] *clause*] ...]
  block
  `!$OMP END PARALLEL`

  - `parallel/end parallel` directive pair must appear in the same routine

C/C++

- 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

C/C++

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)**

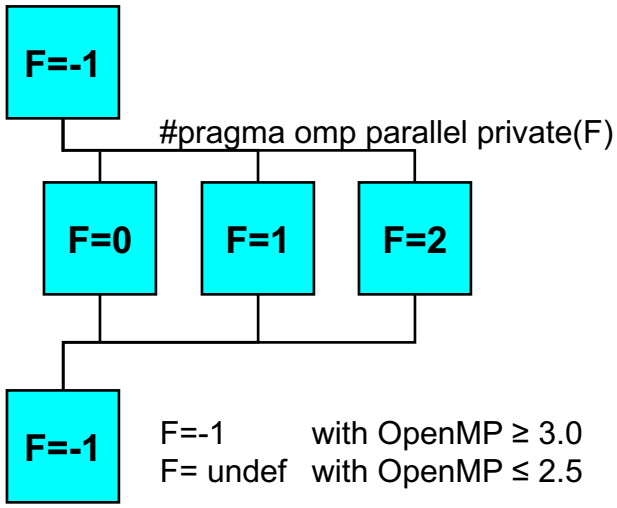
Fortran

OpenMP Directive Format: Fortran

- Treated as Fortran comments – not case sensitive
- Format:
  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.
  They are **uninitialized**.
- `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()
```

C/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()
```

C/C++

  – C/C++:

```
int omp_get_thread_num(void);
```

OpenMP Runtime Library (3): Wall clock timers

Fortran
C/C++

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

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

  - provides “per-thread time”, i.e. needs not be globally consistent

Fortran
C/C++

- ```
  DOUBLE PRECISION FUNCTION OMP_GET_WTICK()
  double omp_get_wtick(void);
  ```
  - returns the number of seconds between two successive clock ticks

Exercise 1: Library-calls

Outline — Exercise 1: Parallel region

- 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**
- Worksharing directives
  - Which thread executes which statement or operation?
  - Exercise 2a: Pi
  - Tasks
  - Synchronization constructs, e.g., critical regions
  - Nesting and Binding
  - Exercise 2b: Pi
- Data environment and combined constructs
  - Private and shared variables, Reduction clause
  - Combined parallel worksharing directives
  - Exercise 3: Pi with reduction clause and combined constructs
  - Exercise 4: Heat
- Summary of OpenMP API
- OpenMP Pitfalls

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/hello/`

  Always “01” in online courses

  `#NR` = number of your PC, e.g., 01

- Serial programs:
  - Fortran 90: `hello.f90`
  - C: `hello.c`

Fortran

```
program hello

integer i

   i=-1
   print *,'hello world',i

end program hello
```

i = “my rank”

into a parallel region

C/C++

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char** argv)
{
  int i;
    i = -1;
    printf( "hello world %d\n",i);
  return 0;
}
```

i = “my rank”

into a parallel region

OpenMP Exercise 1: Parallel region (2)

- Compile **serial** program `hello.[c|f90]` and run
- Compile **as OpenMP** program and run on 4 cores
  - add OpenMP compile option, e.g.,
    - **`-fopenmp` on gnu compilers gcc, and gfortran**
    - **`-qopenmp` on Intel compiler icc and ifort**
  - `export OMP_NUM_THREADS=4 ; ./a.out` (with bash or similar)
  - `setenv OMP_NUM_THREADS 4 ; ./a.out` (with tcsh or similar)
  - expected result: program is not parallelized, same output

```
bash$ gcc -fopenmp  hello.c
or    gfortran -fopenmp  hello.f90
bash$ export OMP_NUM_THREADS=4;  ./a.out
hello world -1
bash$
```

OpenMP Exercise 1: Parallel region (3)

- Please add the following additional code (i.e., don't change existing code!)
  - **If compiled with OpenMP then**
  - `i` := **the rank of the current thread**
- Compile **as OpenMP** and **as non-OpenMP** program and run on 4 cores
  - Expected result:
    - Without OpenMP: no difference

```
bash$ gcc  hello.c
or    gfortran  hello.f90
bash$ export OMP_NUM_THREADS=4;  ./a.out
hello world -1
bash$
```

    - With OpenMP: Only the master thread is running, and therefore `i` = 0

```
bash$ gcc -fopenmp  hello.c
or    gfortran -fopenmp  hello.f90
bash$ export OMP_NUM_THREADS=4;  ./a.out
hello world 0
bash$
```

  - After successful execution, you may compare with the provided solution:
    - `../../solution/hello/hello2.c` or `hello2.f90` (hello1 was your starting point)

OpenMP Exercise 1: Parallel region (4)

- Please add the following additional code (i.e., don't change existing code!)
  - Execute the `i=omp_get_...()` together with the print statement **within a parallel region**
- Compile **as OpenMP** and **as non-OpenMP** program and run on 4 cores
  - Expected result:
    - Without OpenMP: no difference to previous slide

```
…
hello world -1
```

    - With OpenMP: Each thread executes its print statement after each thread stored a different value into `i` = 0, 1, 2, 3, but the variable `i` is shared! Therefore the output can be:

```
bash$ gcc -fopenmp  hello.c
or    gfortran -fopenmp  hello.f90
bash$ export OMP_NUM_THREADS=4;  ./a.out
hello world 3
hello world 1
hello world 1
hello world 3
```

Such a race condition may be invisible, i.e., you may see a correct result, e.g., 0 / 2 / 3 / 1, although the software is WRONG!

  - After successful execution, you may compare with the provided solution:
    - `../../solution/hello/hello3.c` or `hello3.f90`

OpenMP Exercise 1: Parallel region (5)

- If you could not see the race condition, then you can apply a small trick:
  - **Make the race-condition obvious:**
  - Add a `sleep(1);` in C, or a `CALL sleep(1)` in Fortran
  - between the `i=` … and the print statement
- Compile **as OpenMP** and **as non-OpenMP** program and run on 4 cores
  - Expected result:
    - Without OpenMP: no difference to previous slide

```
…
hello world -1
```

    - With OpenMP: Each thread stores a different value into `i` = 0, 1, 2, 3, but the variable `i` is still shared. And printing starts one second later ☺ Therefore the output can be:

```
bash$ gcc -fopenmp  hello.c
or    gfortran -fopenmp  hello.f90
bash$ export OMP_NUM_THREADS=4;  ./a.out
hello world 2
hello world 2
hello world 2
hello world 2
```

If thread 2 was the last one storing its rank into the shared variable i

  - After successful execution, you may compare with the provided solution:
    - `../../solution/hello/hello4.c` or `hello4.f90`

 ● REC → online

OpenMP Exercise 1: Parallel region (6)

- As last step, please add the private() clause to your parallel region directive:
  - Make the variable `i` **private**
- Compile **as OpenMP** and **as non-OpenMP** program and run on 4 cores
  - Expected result:
    - Without OpenMP: no difference to previous slide

```
…
hello world -1
```

    - With OpenMP: Each thread stores a different value 0, 1, 2, 3 into its private `i`.
      And then it prints its private value.
      Therefore the output can be:

```
bash$ gcc -fopenmp  hello.c
or    gfortran -fopenmp  hello.f90
bash$ export OMP_NUM_THREADS=4;  ./a.out
hello world 3
hello world 1
hello world 0
hello world 2
```

In any sequence

  - After successful execution, you may compare with the provided solution:
    - `../../solution/hello/hello5.c` or `hello5.f90`

OpenMP Exercise 1: Summary

- **Conditional compilation** allows to keep the serial version of the program in the same source files
- Testing an OpenMP library routine
  (always within conditional compilation!)
- 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
  (here, in this exercise, we used no worksharing)
- Decision about `private` or `shared` status of variables is important

You may compare your solution with the file in the solution-directory

During the Exercise (25-30 min.)

Please **stay here in the main room** while you do this exercise

And have fun with this short exercise 

Please do not look at the solution before you finished this exercise,
otherwise,
90% of your learning outcome may be lost 

As soon as you finished the exercise,
please **go to your breakout room**
and continue your discussions with your fellow learners:

Exercise + Discussion + Break – planned: 25-30 Minutes

Discussion: MPI, we compared it with 
"each process ~ a human being,
each message or collective communication ~ human communication" 


Which analogy would fit to OpenMP –
or is this a stupid  *question?*

 ● REC → online

Outline — Worksharing 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
- **Worksharing directives**
  - **Which thread executes which statement or operation?**
  - **Exercise 2a: Pi**
  - **Tasks**
  - **Synchronization constructs, e.g., critical regions**
  - **Nesting and Binding**
  - **Exercise 2b: Pi**
- Data environment and combined constructs
  - Private and shared variables, Reduction clause
  - Combined parallel worksharing directives
  - Exercise 3: Pi with reduction clause and combined constructs
  - Exercise 4: Heat
- Summary of OpenMP API
- OpenMP Pitfalls

Worksharing and Synchronization

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

- **i.e., organization of the parallel work!!!**

OpenMP Worksharing 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

C/C++

Fortran

Task generating construct – *another option for organizing the work in parallel*

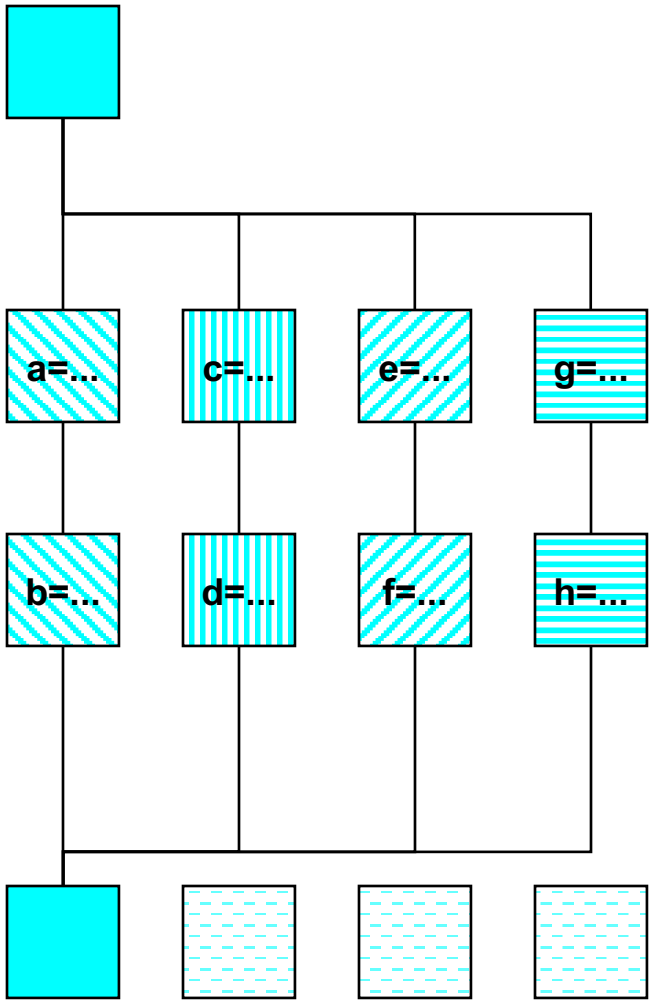
- `task` directive

C/C++

OpenMP `sections` Directives – C/C++

C / C++:

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

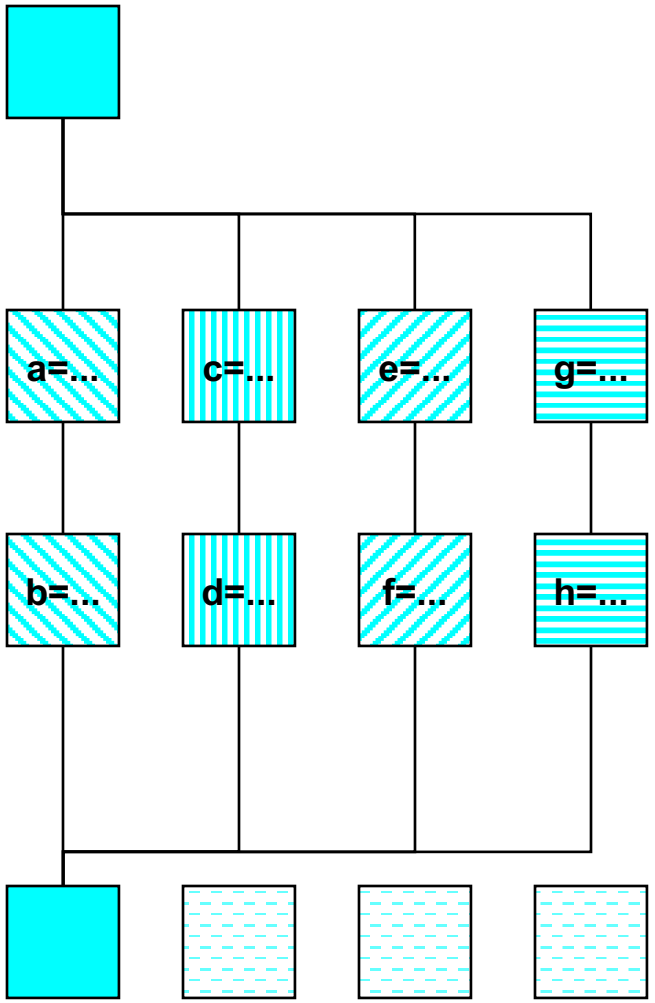


Fortran

OpenMP `sections` Directives – Fortran

Fortran:

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



OpenMP `sections` Directives – Syntax

- Several *blocks* are executed in parallel

Fortran

- Fortran:

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

Blocks must be independent, i.e., they can be executed in parallel

C/C++

- C/C++:

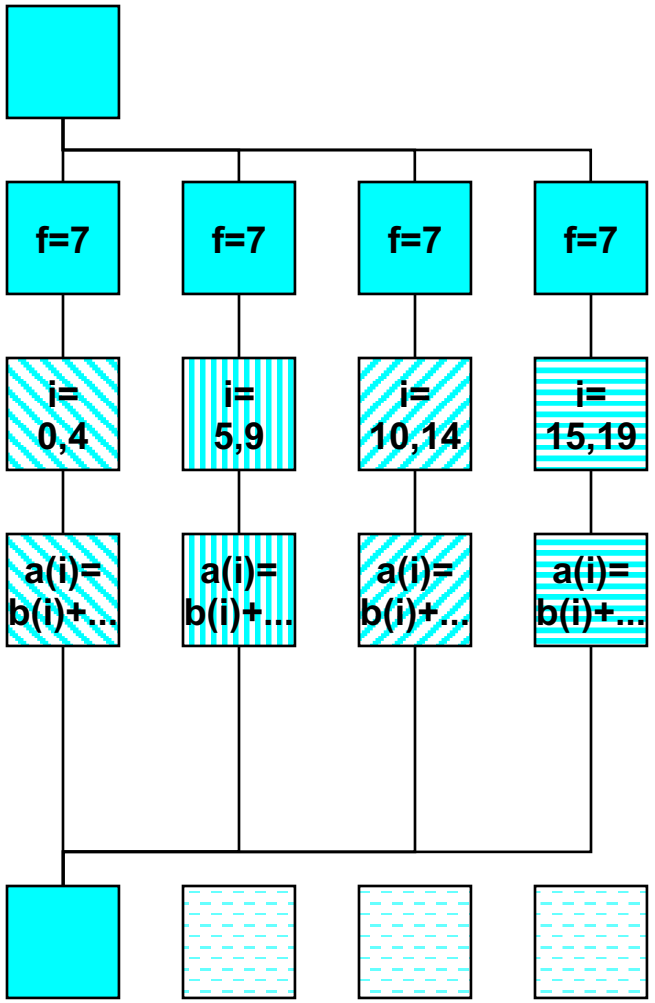
```
#pragma omp sections  [ clause [ [ , ] clause ] ... ] new-line
  {
    [#pragma omp section new-line ]
       structured-block1
    [#pragma omp section new-line
       structured-block2 ]
     ...
  }
```

C/C++

OpenMP `do/for` Directives – C/C++

C / C++:

```
#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 */
```

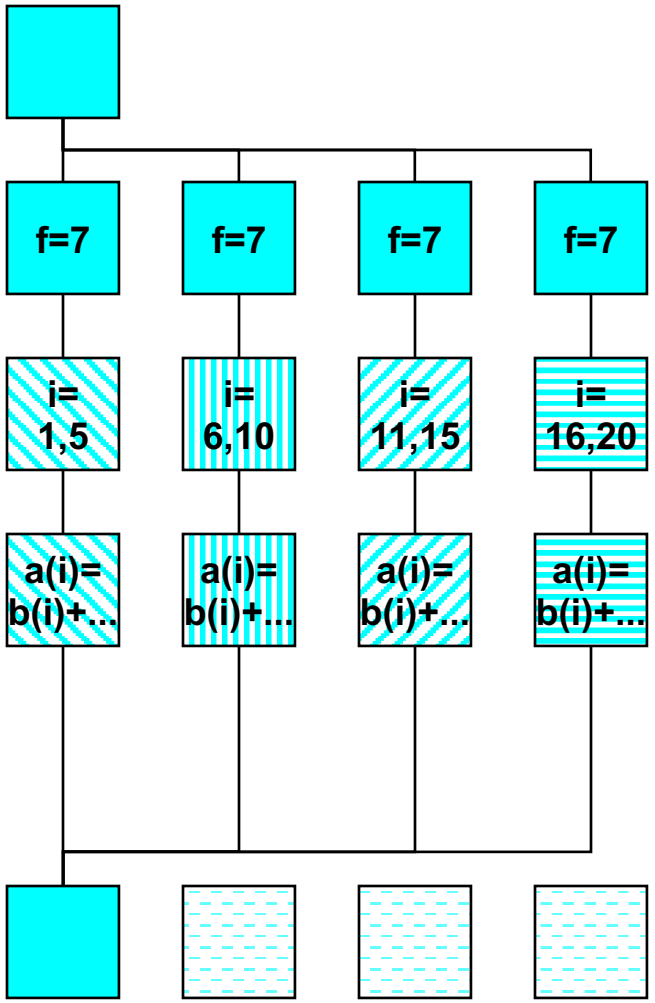


Fortran

OpenMP `do`/`for` Directives – Fortran

Fortran:

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



OpenMP `do`/`for` Directives – Syntax

- Immediately following loop executed in parallel

Fortran

- 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/C++:

```
#pragma omp for [ clause [ [ , ] clause ] ... ] new-line
    for-loop
```

- The corresponding `for` loop must have *canonical shape*
  → *next slide*

Loop iterations must be independent, i.e.,
they can be executed in parallel

OpenMP `for` Directives – Canonical Shape

C/C++

- C/C++:
  `#pragma omp for` [*clause* [[,] *clause*] ...] *new-line*
  for-loop
- The corresponding `for` loop must have *canonical shape:*

```
for( [integer type] var=lb; var < b;  var++       )
                            <=    ++var
                            >     var+=incr
                            >=    var=var+incr
                                  var-- ...
```

`var`: must not be modified in the loop body;
integer (signed or unsigned),
or pointer type (C only), ⟸ (OpenMP ≥ 3.0)
or random access iterator type (C++ only)

`lb,b,incr`: loop invariant expression
→ the number of iterations must be computable at loop begin

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]**
  - `collapse(` *n* `)` (OpenMP ≥ 3.0)
  - `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, i.e.,
  if `nowait` is specified, threads do not synchronize at the end of the parallel loop
- `collapse(` *n* `)` with constant integer expression *n*:
  The iterations of the following *n* nested loops are collapsed into one larger iteration space.
- `schedule` clause specifies how the iterations (iteration space)
  of the (nested) loop(s) are divided among the threads of the team.

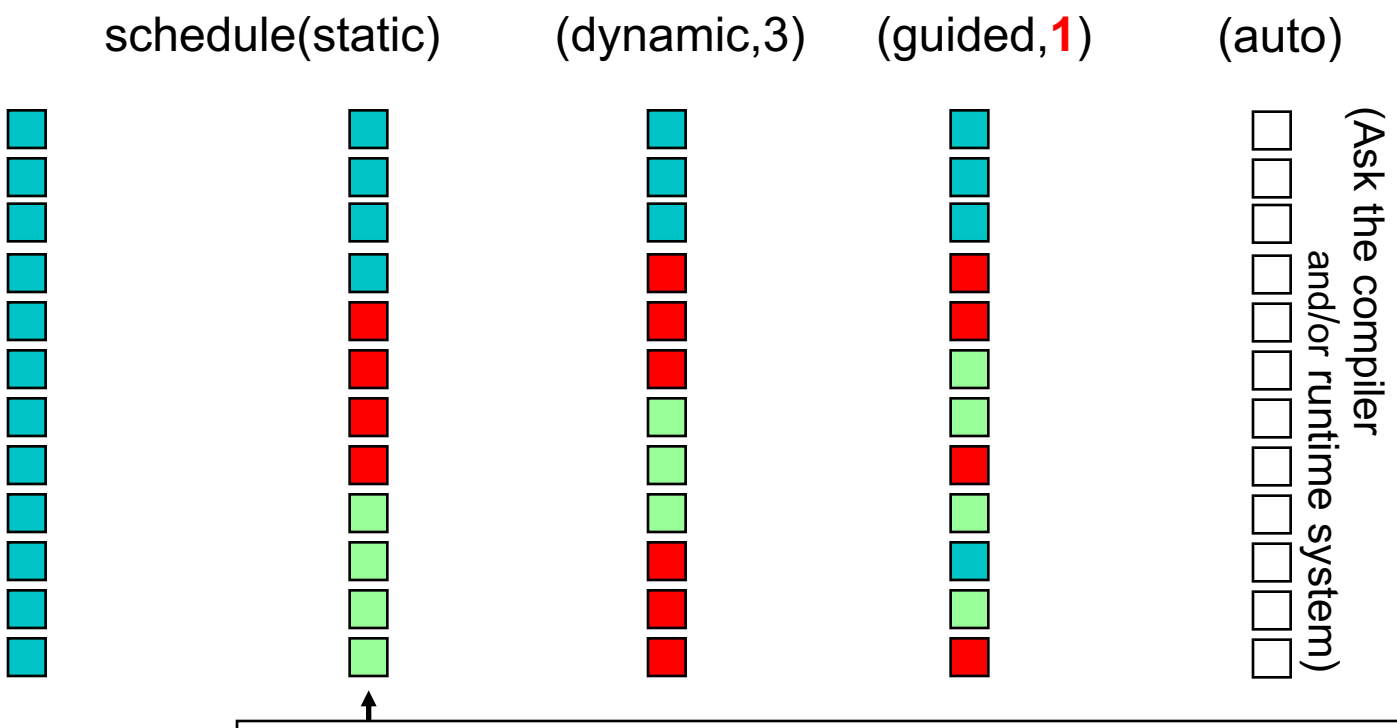
OpenMP `schedule` Clause

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.
- `auto`: Scheduling is delegated to the compiler and/or runtime system **(OpenMP ≥ 3.0)**
- `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



- Method is implementation dependent, e.g.,
  13 iterations on 3 threads = 5+5+3 or = 5+4+4
- Two loops in same parallel region and with same count are divided in same way, i.e., static schedule is **deterministic**

OpenMP ≥ 3.0

(in OpenMP-5.1/5.2, text moved to Sect. 2.11.2/4.4.5 *Consistent Loop Schedules*)

Further information: OpenMP 3.0, Sect. 2.5.1, Description, pp 41-44

Fortran

The WORKSHARE directive

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

```
!$OMP WORKSHARE
A=B+C
! Rest of block
!$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 DO worksharing without explicit loops.

OpenMP `single` Directive – Syntax

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

Fortran

- Fortran:

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

C/C++

- 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 — Exercise 2a: Pi

- 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
- **Worksharing directives**
  - **Which thread executes which statement or operation?**
  - **Exercise 2a: Pi**
  - **Tasks**
  - **Synchronization constructs, e.g., critical regions**
  - **Nesting and Binding**
  - **Exercise 2b: Pi**
- Data environment and combined constructs
  - Private and shared variables, Reduction clause
  - Combined parallel worksharing directives
  - Exercise 3: Pi with reduction clause and combined constructs
  - Exercise 4: Heat
- Summary of OpenMP API
- OpenMP Pitfalls

OpenMP Exercise 2a: pi Program (1)

- Goal: usage of
  - worksharing constructs: `do`/`for`
  - `critical` directive → in Exercise 2b

- Working directory: `~/OpenMP/#NR/pi/`

  `#NR` = number of your PC, e.g., 01

- Serial programs:
  - Fortran 90: `pi.f90`
  - C: `pi.c`

Fortran

C/C++

- Calculating $4 \int_0^1 \frac{1}{1+x^2}\, dx = 4 \arctan(x)\big|_0^1 = 4\left(\frac{\pi}{4} - 0\right) = \pi$

OpenMP Exercise 2a: pi Program (2)

- compile serial program `pi.[c|f90]` and run
- Compile and run with OpenMP
  - Expected result: same pi-value and timing,
  - but the program reports the number of threads, and timing with omp_get_wtime
- **add parallel region** and **`do/for` directive** in `pi.[c|f90]` 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?)

- After *this test with race-conditions*, you may compare your result with the provided solution:
  - `../../solution/pi/pi1.c` or `pi1.f90`

 ● REC → online

OpenMP Exercise 2a: pi Program (3)

- add `private(x)` clause in `pi.[c|f90]` 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?)

- After *this test with race-conditions*, you may compare your result with the provided solution:
  - `../../solution/pi/pi2.c` or `pi2.f90`

- And where is the second race-condition?

  → We'll resolve this in Exercise 2b – after the talk "Synchronization constructs, e.g., critical regions"

You may compare your solution with the file in the solution-directory

During the Exercise (15 min.)

Please **stay here in the main room** while you do this exercise

And have fun with this short exercise 

Please do not look at the solution before you finished this exercise,
otherwise,
90% of your learning outcome may be lost

As soon as you finished the exercise,
please **go to your breakout room**
and continue your discussions with your fellow learners:

Exercise + Discussion + Break – planned: 15 Minutes

Discussion: Do you have ideas how one can solve the problem based on the knowledge you learnt so far?

Please do  *not program – only discuss!* 

Tasks

Outline — Tasks

- 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
- **Worksharing directives**
  - **Which thread executes which statement or operation?**
  - **Exercise 2a: Pi**
  - **Tasks** *(a task generating construct)*
  - **Synchronization constructs, e.g., critical regions**
  - **Nesting and Binding**
  - **Exercise 2b: Pi**
- Data environment and combined constructs
  - Private and shared variables, Reduction clause
  - Combined parallel worksharing directives
  - Exercise 3: Pi with reduction clause and combined constructs
  - Exercise 4: Heat
- Summary of OpenMP API
- OpenMP Pitfalls

OpenMP `task` Directive – Example:

Parallelized traversing of a tree

OpenMP 3.0

C/C++

```
struct node {
   struct node *left;
   struct node *right;
};
extern void process(struct node *);
void traverse( struct node *p ) {
    if (p->left)
#pragma omp task // p is firstprivate by default
                 traverse(p->left);
    if (p->right)
#pragma omp task // p is firstprivate by default
                 traverse(p->right);
    process(p); // significant work with p
}
int main(int argc, char **argv)
{ struct node tree;
  ... // producing the tree
#pragma omp parallel
  {
#pragma omp single
    {
      traverse(&tree);//traversing the existing tree
    } // end of omp single
  } // end of omp parallel
}
```

- Starting the parallel team of threads
- Using only one thread for starting the traversal
- First execution with single thread (= 1st task)
- A new task is started (on a new thread)
- A recursive call to traverse() in this 2nd task
- 3rd task is started
- Work is done in 1st task
- Recursive calls starting 4th, 5th, … tasks

Same example in Fortran, see OpenMP 3.0, Exa. A.13.1f, page 178

Trick: OpenMP can choose whether new tasks are immediately started or deferred until free thread is available.

OpenMP `task` Directive – Syntax

OpenMP 3.0

- The **`task`** construct defines an explicit task.

Fortran

- Fortran:

```
!$OMP task [ clause [ [ , ] clause ] ... ]
    block
!$OMP end task
```

C/C++

- C/C++:

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

- Clauses:
  - `untied`
  - `default(shared | none | private | firstprivate)`
  - `private(`*list*`)`
  - `firstprivate(`*list*`)`
  - `shared(`*list*`)`
  - `if(`*scalar expression*`)`

OpenMP `task` Directive – Principles OpenMP 3.0

- When a thread encounters a **`task`** construct,
  a task is generated from the code for the associated structured block.
- The encountering thread
  - may immediately execute the task,
  - or may defer its execution.

  The number of tasks can be limited, e.g., to the number of threads.
- Completion of a task can be guaranteed using task synchronization constructs → **`taskwait`** construct.
- When `if(false)` clause exists, then execution is *"serial"*
- **Task scheduling points**:
  - In the generating task: Immediately following the generation of an explicit task.
  - In the generated task: After the last instruction of the task region.
  - If task is "**`untied`**": Everywhere inside of the task.
  - In implicit and explicit barriers.
  - In **`taskwait`**.

  At task scheduling points, tasks can be resumed or suspended.
  (Further constraints → OpenMP 3.0, Sect. 2.7.1, page 62)

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
- **Worksharing directives**
  - **Which thread executes which statement or operation?**
  - **Exercise 2a: Pi**
  - **Tasks**
  - **Synchronization constructs, e.g., critical regions**
  - **Nesting and Binding**
  - **Exercise 2b: Pi**
- Data environment and combined constructs
  - Private and shared variables, Reduction clause
  - Combined parallel worksharing directives
  - Exercise 3: Pi with reduction clause and combined constructs
  - Exercise 4: Heat
- Summary of OpenMP API
- OpenMP Pitfalls

OpenMP Synchronization

- Implicit Barrier
  - end of `parallel` constructs
  - end of all other control constructs
  - implicit synchronization on control constructs 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/C++:

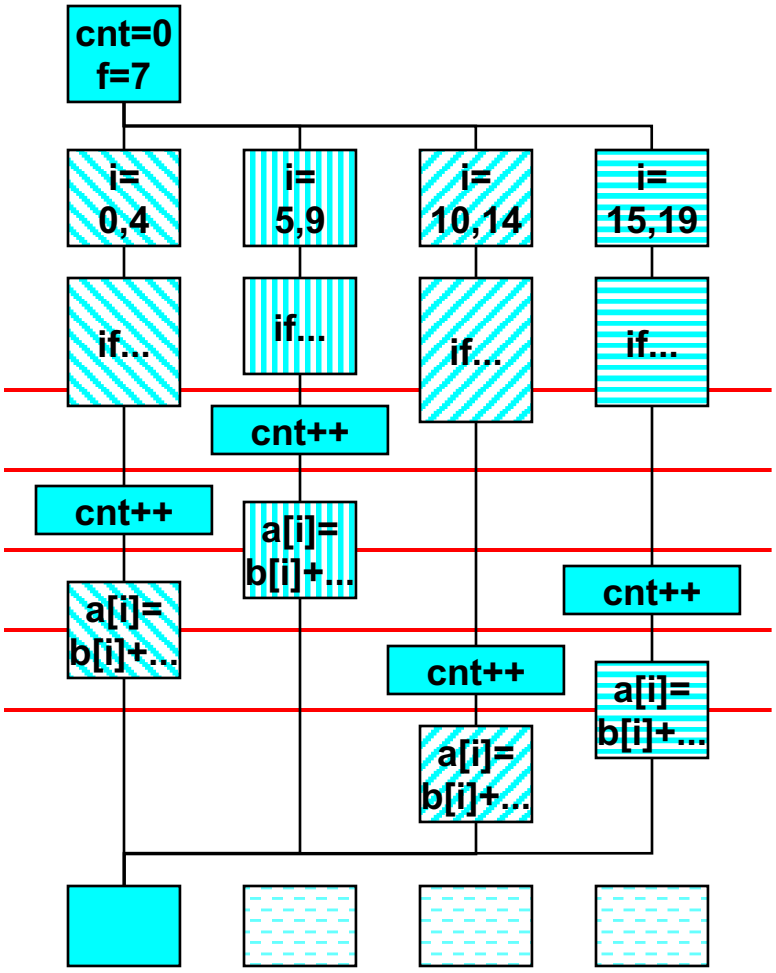
```
#pragma omp critical [ ( name ) ] new-line
    structured-block
```

- 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.

C/C++

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

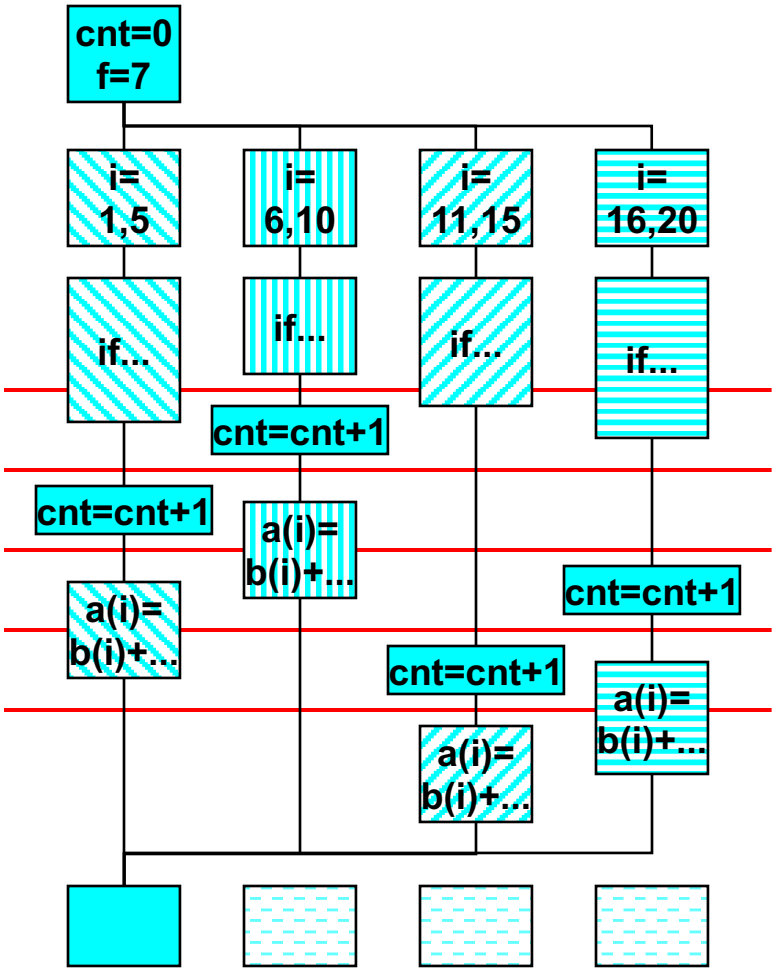
```
C / C++:   cnt = 0;
           f=7;
      #pragma omp parallel
      {
      #pragma omp for
            for (i=0; i<20; i++) {
               if (b[i] == 0) {

                  #pragma omp critical
                   cnt ++;

               } /* endif */
               a[i] = b[i] + f * (i+1);
            } /* end for */
      } /*omp end parallel */
```



Fortran

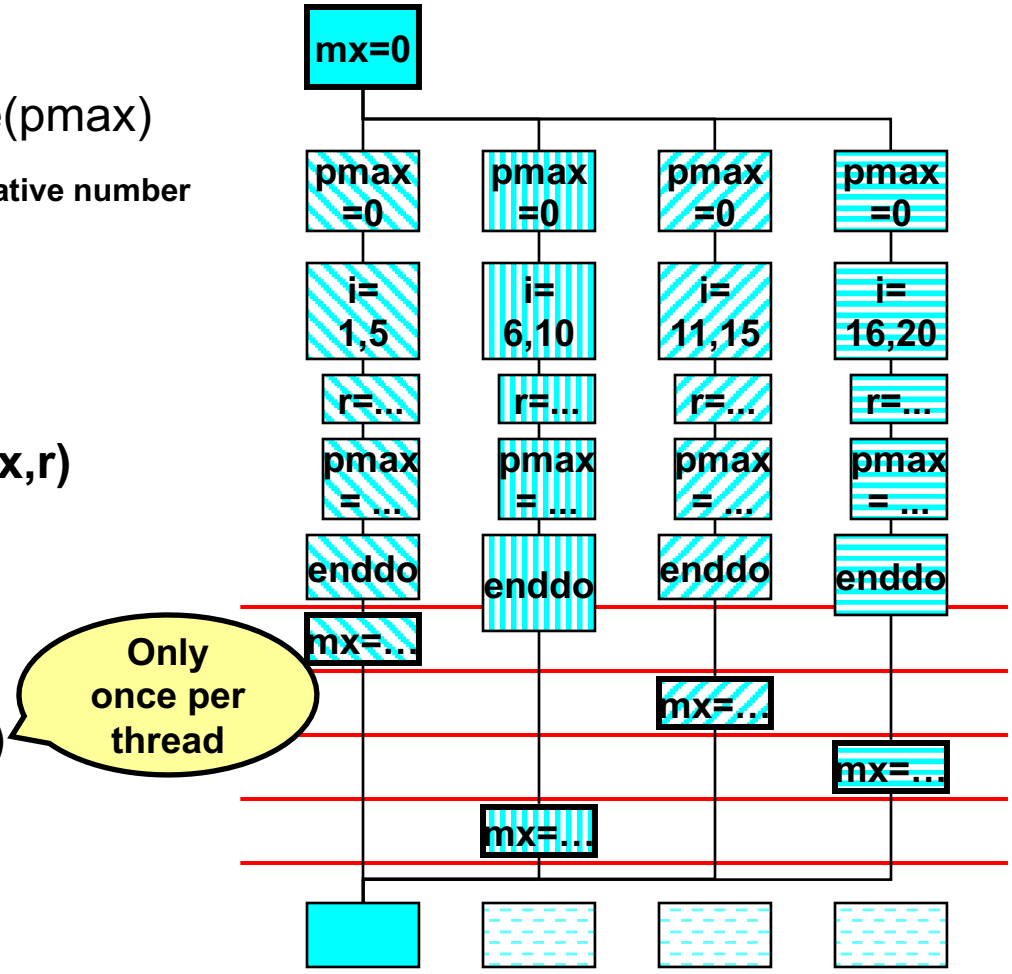
OpenMP `critical` — an example (Fortran)

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



Fortran OpenMP `critical` — another example (Fortran)

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



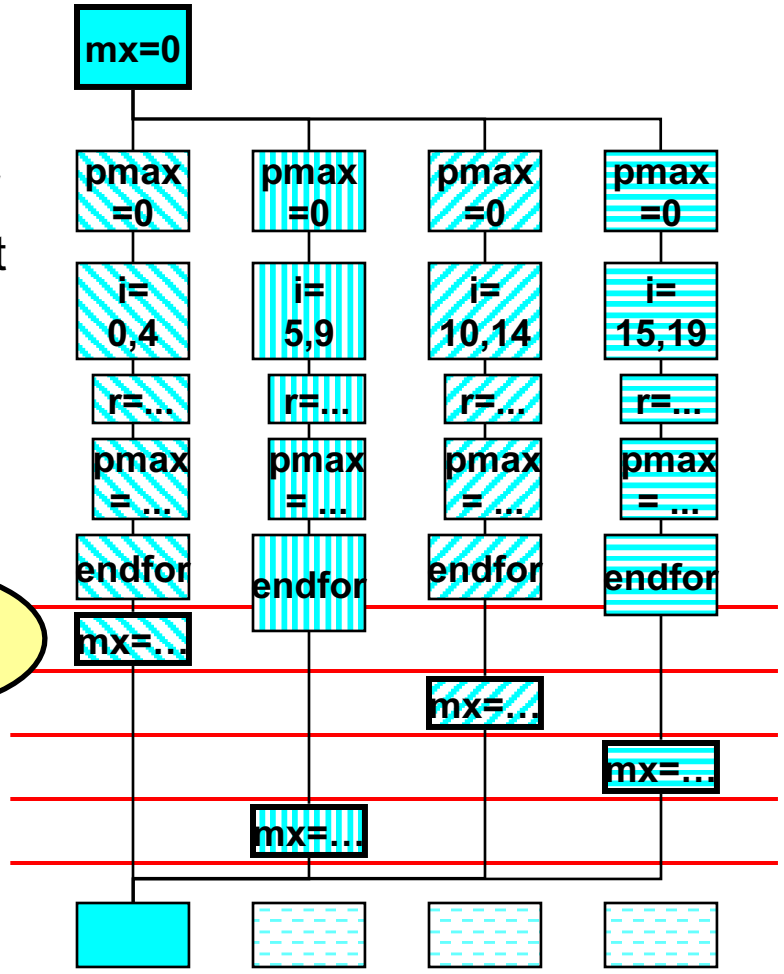
 ● REC → online

C/C++

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

```
      mx = 0;
#pragma omp parallel private(pmax)
{     pmax = 0; /* or most negative number */
      #pragma omp for private(r) nowait
      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*/
```

Only once per thread



Outline — Nesting and Binding

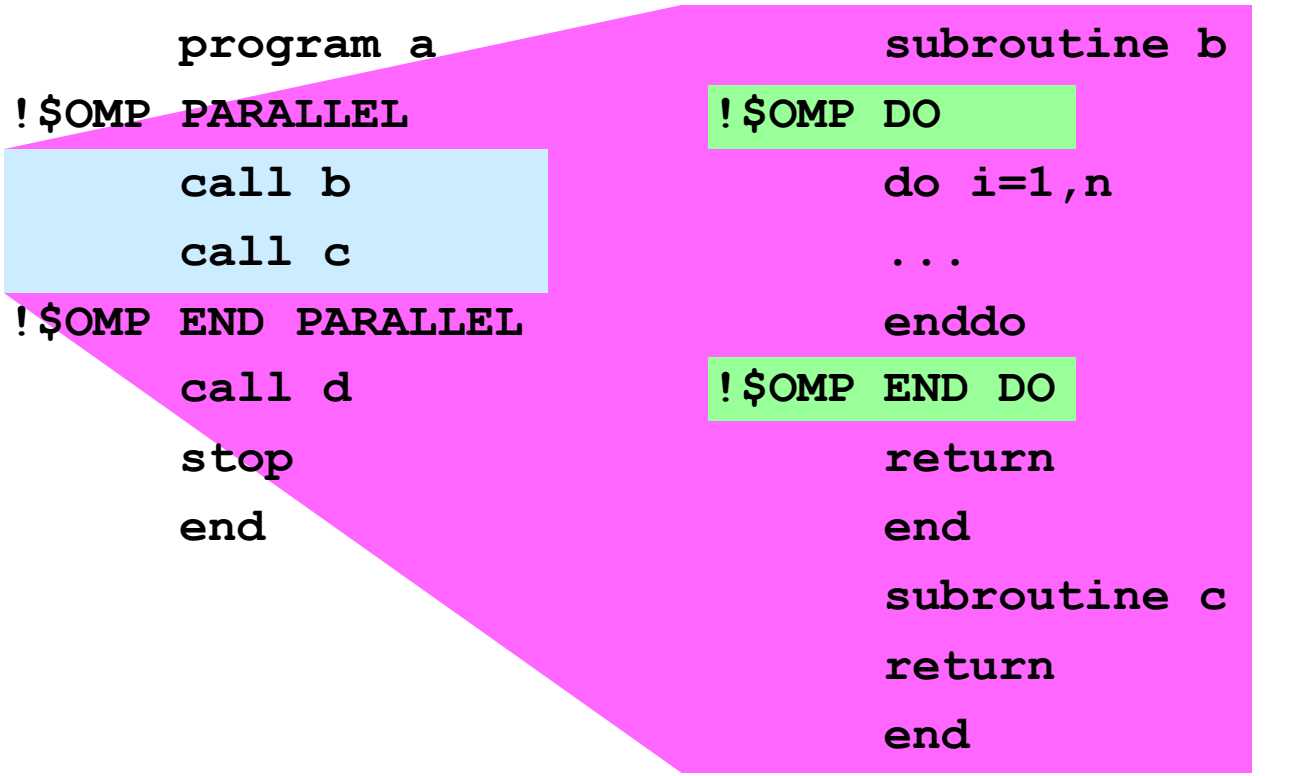
- Introduction into OpenMP
- Programming and Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Data environment (part 1)
  - Environment variables
  - Runtime library routines
  - Exercise 1: Parallel region / library calls / privat & shared variables
- **Worksharing directives**
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

[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 Vocabulary



OpenMP Control Structures — Summary

- Parallel region construct
  - `parallel`
- Worksharing constructs
  - `sections`
  - `for` (C/C++)
  - `do` (Fortran)
  - `workshare` (Fortran)
  - `Single`
- Task generating construct
  - `task`
- Combined parallel worksharing constructs **[see later]**
  - `parallel for` (C/C++)
  - `parallel do` (Fortran)
  - `parallel workshare` (Fortran)
- Synchronization constructs
  - `critical`

Outline — Exercise 2b: pi

- 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
- **Worksharing directives**
  - **Which thread executes which statement or operation?**
  - **Exercise 2a: Pi**
  - **Tasks**
  - **Synchronization constructs, e.g., critical regions**
  - **Nesting and Binding**
  - **Exercise 2b: Pi**
- Data environment and combined constructs
  - Private and shared variables, Reduction clause
  - Combined parallel worksharing directives
  - Exercise 3: Pi with reduction clause and combined constructs
  - Exercise 4: Heat
- Summary of OpenMP API
- OpenMP Pitfalls

OpenMP Exercise 2b: pi Program (4)

- Use your result from Exercise 2a (or solution `pi2.[c|f90]`)
- Add `critical` directive in your `pi.[c|f90]` around the sum-statement and compile
- set environment variable `OMP_NUM_THREADS` to `2` and run
  - value of pi? (should be now correct!, but **huge wtime!**)
- 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, does it take longer?)
- run again
  - value of pi? execution time?
  - How can you optimize your code?

Look at wall-clock time, not at CPU time!

- After *successful execution*, you may compare your result with the provided solution:
  - `../../solution/pi/pic.c` or `picf90`

OpenMP Exercise 2b: pi Program (5)

- move `critical` directive in `pi.[c|f90]` outside loop, and compile
- set environment variable `OMP_NUM_THREADS` to `2` and run
  - 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?

- After *successful execution*, you may compare your result with the provided solution:
  - `../../solution/pi/pi`**`c2`**`.c` or `pi`**`c2`**`.f90`

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

You may compare your solution with the file in the solution-directory

During the Exercise (25 min.)

Please **stay here in the main room** while you do this exercise

And have fun with this short exercise 

Please do not look at the solution before you finished this exercise,
otherwise,
90% of your learning outcome may be lost

As soon as you finished the exercise,
please **go to your breakout room**
and continue your discussions with your fellow learners:

Exercise + Discussion + Break – planned: 25 Minutes

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
- Worksharing directives
  - Which thread executes which statement or operation?
  - Exercise 2a: Pi
  - Tasks
  - Synchronization constructs, e.g., critical regions
  - Nesting and Binding
  - Exercise 2b: Pi
- **Data environment and combined constructs**
  - **Private and shared variables, Reduction clause**
  - **Combined parallel worksharing 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`
- Exceptions: `private`
  - stack (local) variables in called subroutines
  - Automatic variables within a block
  - Loop control variable of parallel DO (Fortran) and FOR (C) loops
  - Fortran only:
    - Loop control variable of a sequential loop enclosed in a parallel or task construct
    - Implied-do and forall indices
- Recommendation for C/C++:
  - Avoid private variables, use variables local to a block instead

Further information: OpenMP 3.0, Sect. 2.9.1, page 77

Private Clause

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

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

→ writes -777 or undefined value

(OpenMP ≥ 3.0)
(OpenMP ≤ 2.5)

- 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
- Nested `private(var)` with same `var` → allocates again new private storage
- Sometimes `shared/private` is undefined → see OpenMP 3.0, Example A.30.2c/f pp. 234-235

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.,`
    `max, min, iand, ior,` or `ieor`
  - C/C++:
    `+, *, -, &, ^, |, &&,` or `||`
    With OpenMP 3.1 and later: `max, min`
- Variables must **not** be `private` in the enclosing context
- With OpenMP 2.0 and later, 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

Fortran

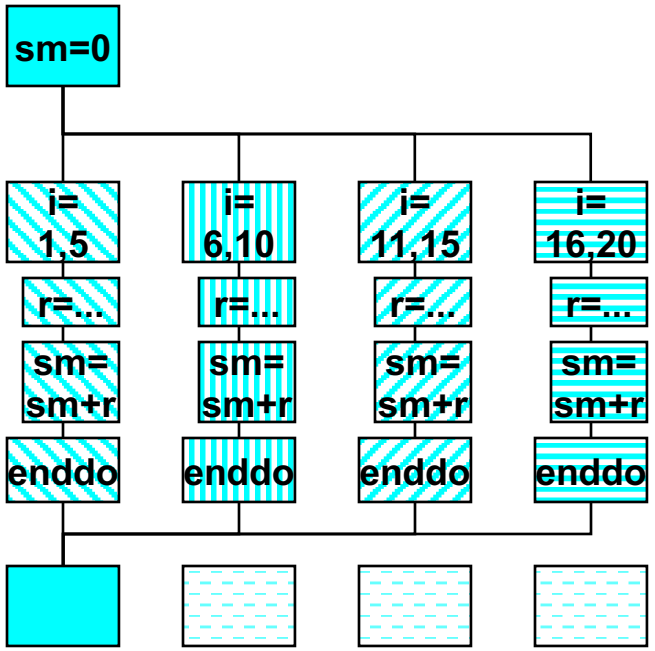
C/C++

Fortran

OpenMP `reduction` — an example (Fortran)

Fortran:

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

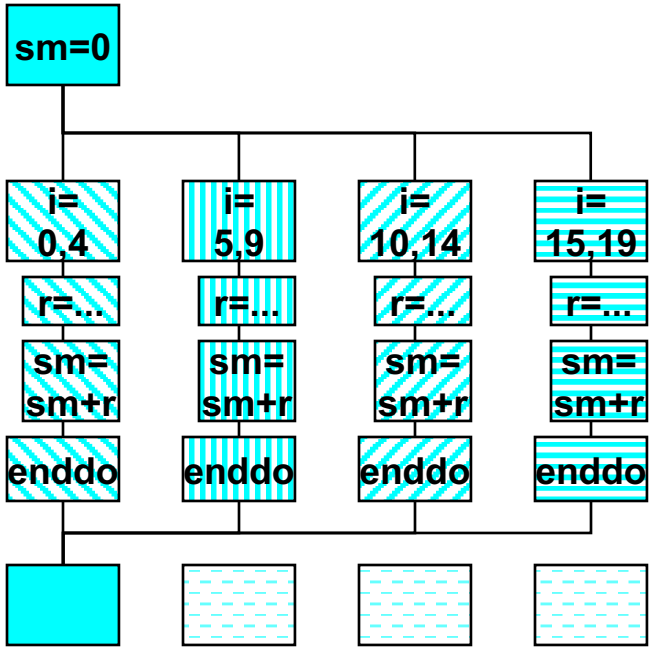


C/C++

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

C / C++:

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



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/C++:

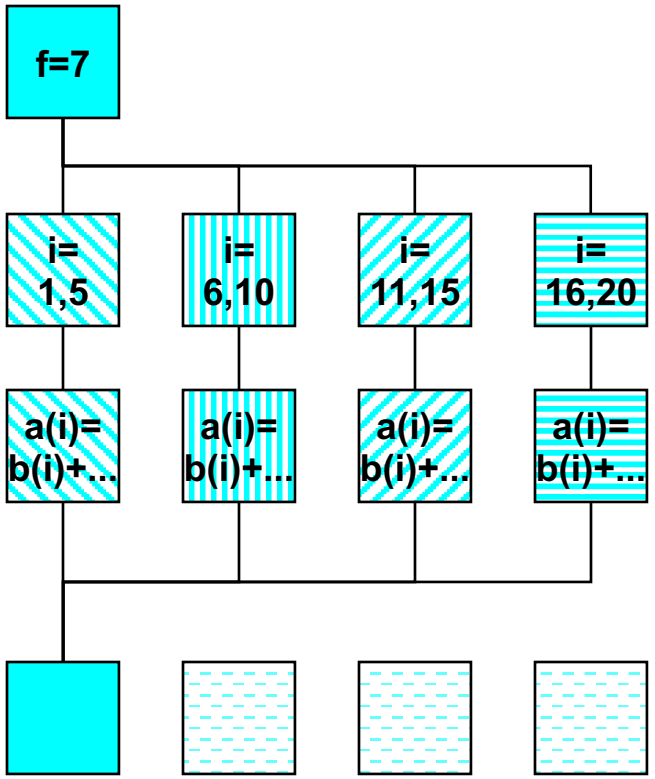
`#pragma omp parallel for` [*clause* [*clause*] ...] *new-line*
for-loop

- This directive admits all the clauses of the `parallel` directive and the `do/for` directive except the `nowait` clause, with identical meanings and restrictions

Fortran OpenMP Combined `parallel do/for` — an example

Fortran:

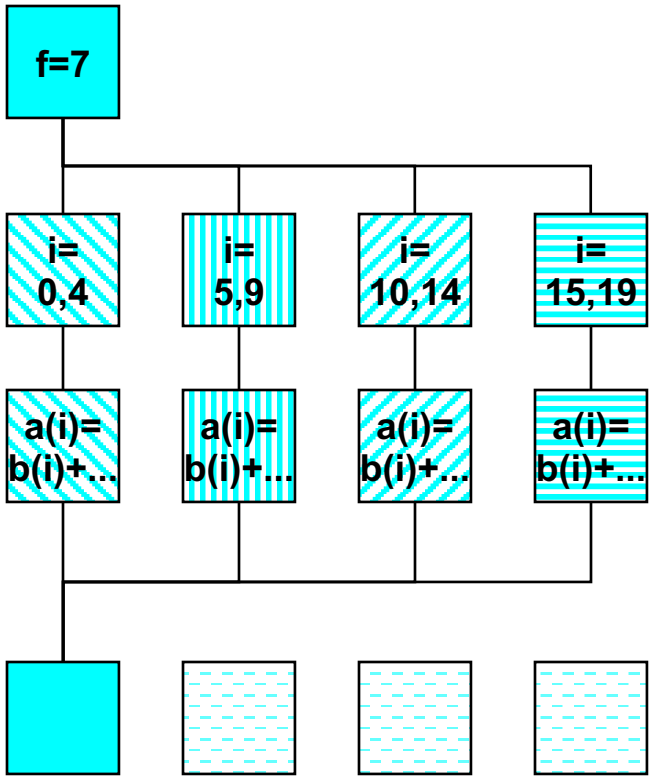
```
      f=7
!$OMP PARALLEL DO
      do i=1,20
         a(i) = b(i) + f * i
      end do
!$OMP END PARALLEL DO
```



C/C++ OpenMP Combined `parallel do/for` — an example

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
- Worksharing directives
  - Which thread executes which statement or operation?
  - Exercise 2a: Pi
  - Tasks
  - Synchronization constructs, e.g., critical regions
  - Nesting and Binding
  - Exercise 2b: Pi
- **Data environment and combined constructs**
  - **Private and shared variables, Reduction clause**
  - **Combined parallel worksharing directives**
  - **Exercise 3: Pi with reduction clause and combined constructs**
  - **Exercise 4: Heat**
- Summary of OpenMP API
- OpenMP Pitfalls

OpenMP Exercise 3: pi Program (1)

- Goal: usage of
  - worksharing constructs: `do`/`for`
  - `critical` directive
  - `reduction` clause
  - combined parallel worksharing constructs: `parallel do`/`parallel for`

- Working directory: `~/OpenMP/#NR/pi/`

  `#NR` = number of your PC, e.g., 01

- Use your result `pi.[c|f90]` from the exercise 2
- or copy solution of exercise 2 to your directory:
  - `cp ../../solution/pi/pic2.* .`
- or copy slow "*critical inside loop*" solution of exercise 2 to your directory:
  - `cp ../../solution/pi/pic.* .`

OpenMP Exercise 3: pi Program (2)

- remove `critical` directive in `pi.[c|f90]`,
  remove your additional partial sum variable, i.e., restore original numeric,
  and 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?

- After *successful execution*, you may compare your result with the provided solution:
  - `../../solution/pi/pir.c` or `pir.f90`

OpenMP Exercise 3: pi Program (3)

- change parallel region + `do`/`for` to the combined parallel worksharing 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?

- After *successful execution*, you may compare your result with the provided solution:
  - `../../solution/pi/pir2.c` or `pir2.f90`

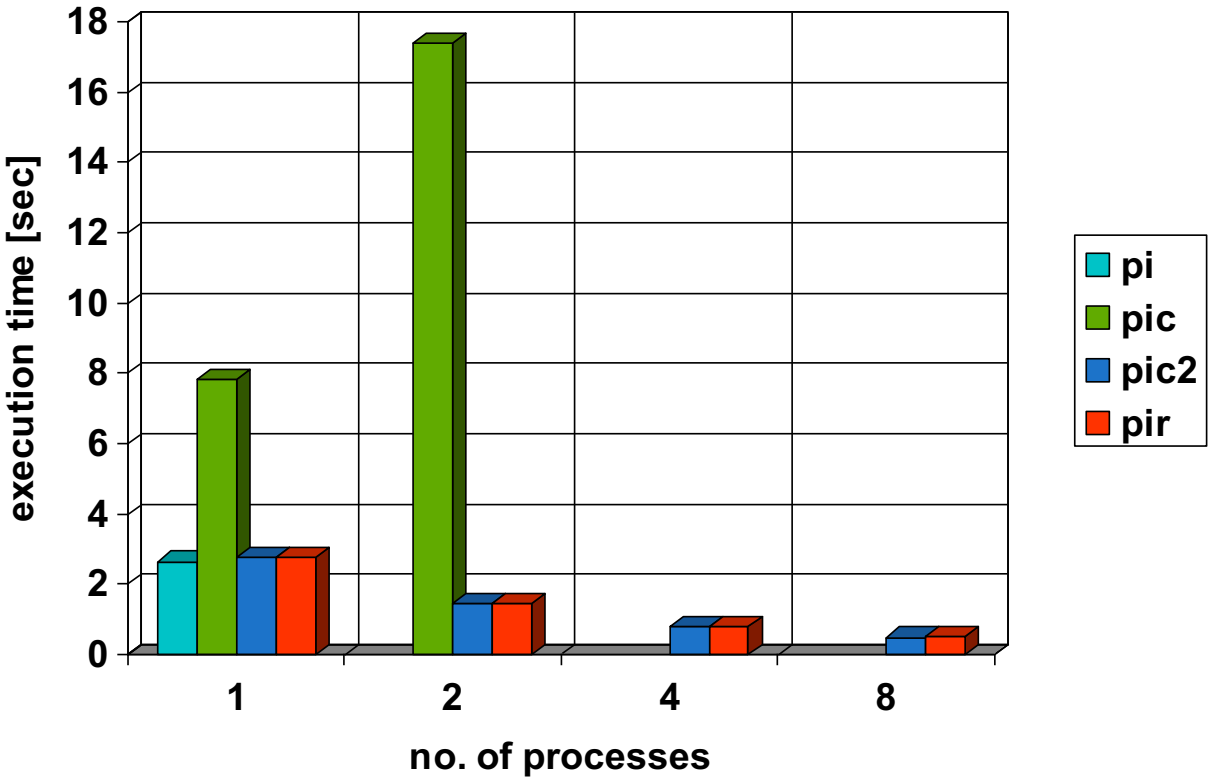
- At the end, compile again **without** OpenMP
  - Does your code still compute a **correct** value of **pi?**

OpenMP Exercise 3: pi Program - Solution

Location: ~/OpenMP/solution/pi

- `pi.[c|f90]` original program
- `pi1.[c|f90]` incorrect (no private, no synchronous global access) !!!
- `pi2.[c|f90]` incorrect (still no synchronous global access to `sum`) !!!
- `pic.[c|f90]` solution with `critical` directive, but extremely slow!
- `pic2.[c|f90]` solution with `critical` directive outside loop
- `pir.[c|f90]` solution with `reduction` clause
- `pir2.[c|f90]` solution with combined `parallel do/for` and `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
- 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
- Convenient `reduction` clause
- Convenient shortcut form

You may compare your solution with the file in the solution-directory

During the Exercise (20 min.)

Please **stay here in the main room** while you do this exercise

And have fun with this short exercise 

Please do not look at the solution before you finished this exercise,
otherwise,
90% of your learning outcome may be lost 

As soon as you finished the exercise,
please **go to your breakout room**
and continue your discussions with your fellow learners:

Exercise + Discussion + Break – planned: 20 Minutes

Already in your breakout room, you may test different loop schedules (static, dynamic, guided, auto) and share your results with your colleagues.

Did you oversubscribe your hardware?  

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
- Worksharing directives
  - Which thread executes which statement or operation?
  - Exercise 2a: Pi
  - Tasks
  - Synchronization constructs, e.g., critical regions
  - Nesting and Binding
  - Exercise 2b: Pi
- **Data environment and combined constructs**
  - **Private and shared variables, Reduction clause**
  - **Combined parallel worksharing 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 $df/dt=\Delta f$
- 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., 01
- Serial programs:
  - Fortran: `heat.F90`
  - 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
```

unlimit or
ulimit -s 200000
once before calling **mpirun**

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
- Or just with: reduction(max:dphimax)
- Hints:
  - **Parallelize outer loop (index `i` in Fortran, `k` in C)**
  - **make inner loop index private!**

TO DO Compile and 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?

OpenMP Exercise: Heat Conduction (4)

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

Fortran

```
do i=1,imax-1                                                    Fortran
   do k=1,kmax-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
     phin(i,k) = phi(i,k) + dphi
   enddo
enddo
```

C/C++

```
for (k=1;k<kmax;k++)                                             C/C++
{ 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;
    phin[i][k] = phi[i][k] + dphi;
  }
}
```

Automati-cally fixed by serial compiler !

Not fixed by OpenMP compiler !

  - 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

Fortran

C/C++

OpenMP Exercise: Heat Conduction (5)

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

Don‘t forget to modify name of private inner loop index!!!

TO DO Compile and 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

TO DO Compile and 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

TO DO 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

TO DO – the current parallel region
that is forked and joined in each `it=...` iteration

– by a parallel region around `it=...` loop
forked and joined only once

- Caution:

TO DO – dphimax=0
must be surrounded by

```
#pragma omp barrier
#pragma omp single
{ dphimax=0;
}
```

– Why?

Shared(dphimax) is necessary for **B**.
Write-write-conflict on **A-A** without **single**.
Write-read-conflict on **A-B** without **barrier**.

```
/*time step iteration */
 for (it=1;it<=itmax;it++)
 {
   dphimax=0.;  /*line A*/
   for (i=1;i<imax;i++)                    worksharing
   {
     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 (i=1;i<imax;i++)                    worksharing
   {
     for (k=1;k<kmax;k++)
     {
       phi[i][k]=phin[i][k];
     }
   }
   if(dphimax<eps) break;  /*line B*/
 }
```

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

Fortran

TO DO Fortran only:
Substitute critical-section-trick
by `reduction(max:dphimax)` clause

OpenMP Exercise: Heat - Solution (1)

Location: `~/OpenMP/solution/heat`

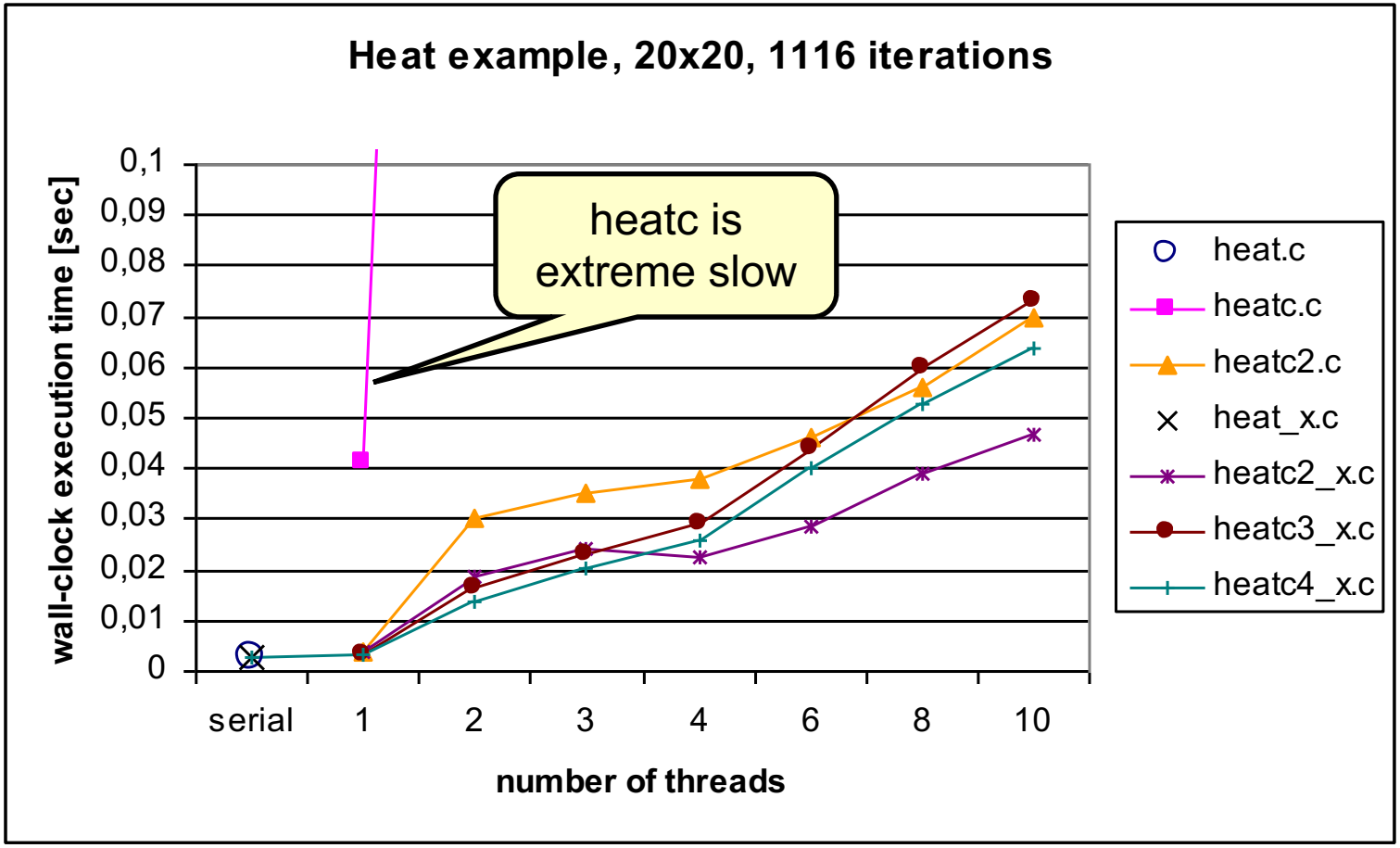
- `heat.[F|c]` Original program
- `heat_x.[F|c]` Better serial program with interchanged nested loops
- `heatc.[F|c]` Extremely slow solution with `critical` section inside iteration loop
- `heatc2.[F|c]` Slow solution with `critical` section outside inner loop, one parallel region inside time step iteration 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|c]` Solution with `schedule(runtime)` clause
- `heatr2_x.[F|c]` Solution with `reduction` clause, one parallel region inside iteration loop

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]**

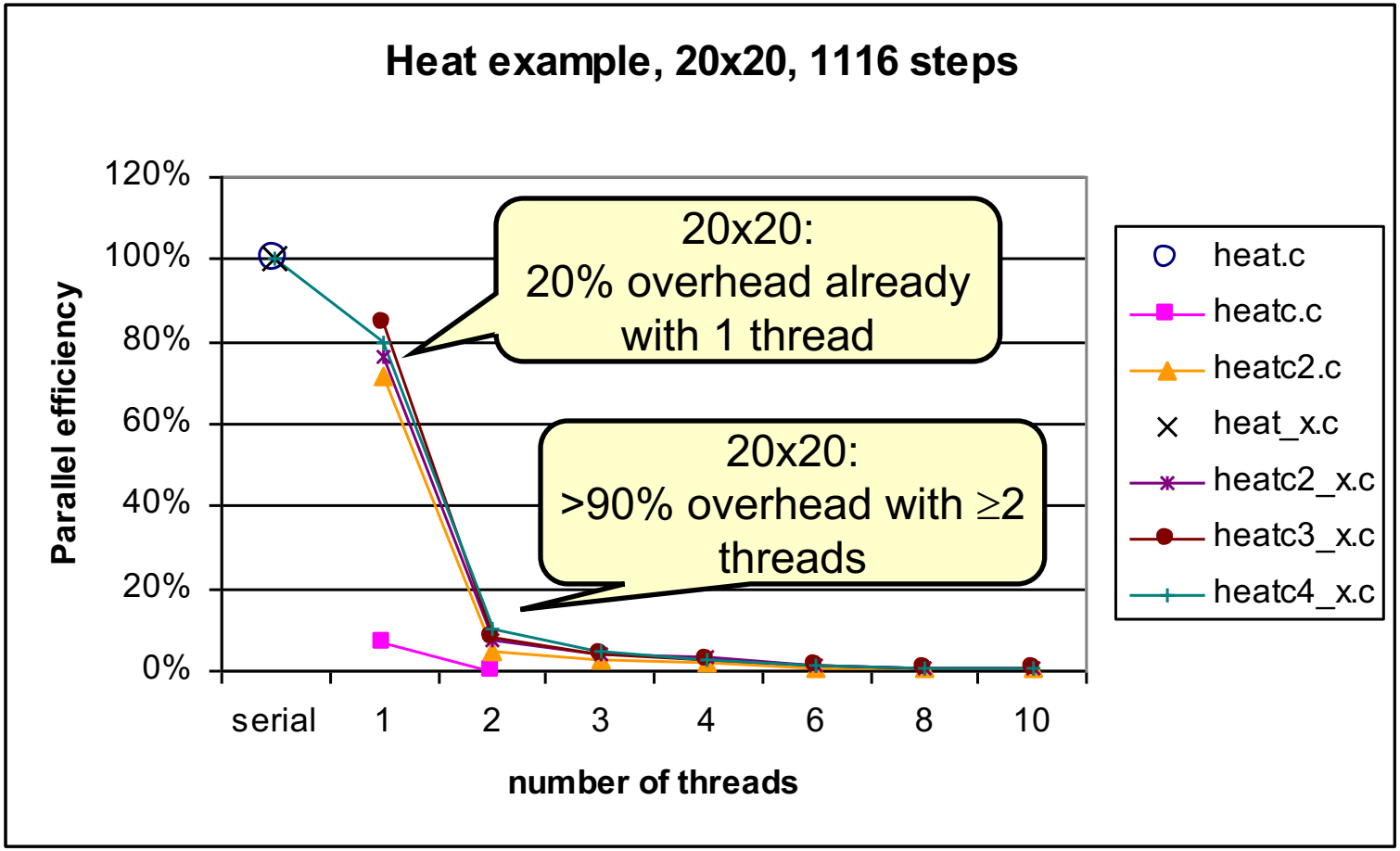
- heatc2_x ←→ heatc3_x
  Additional overhead for barriers and single sections (including implied barrier)
  must be compared with fork-join-overhead

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



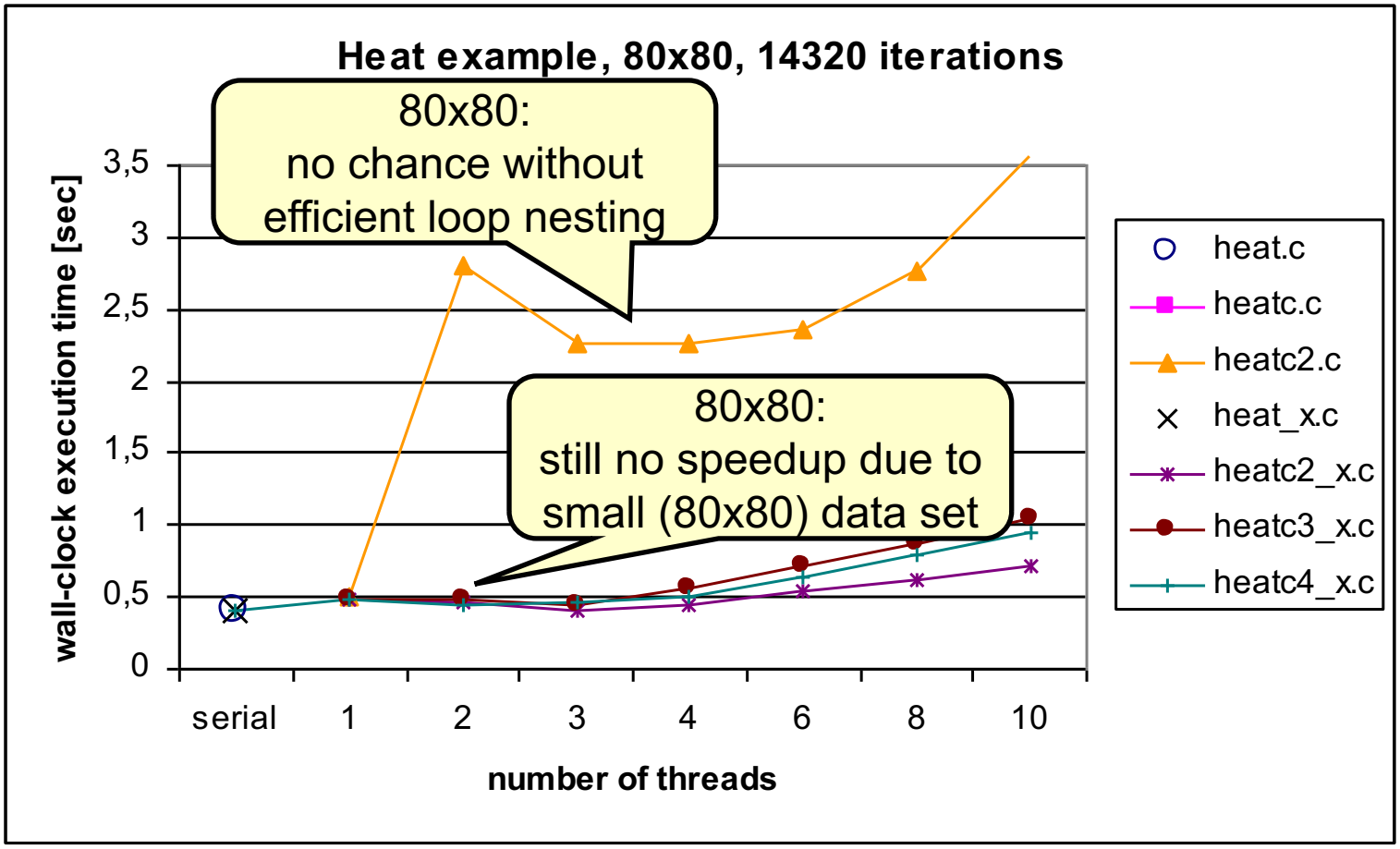
Measurements on NEC TX-7 (asama.hww.de), 16 cores, May 4-5, 2006

OpenMP Exercise: Heat - Solution (4) – 20x20 Efficiency

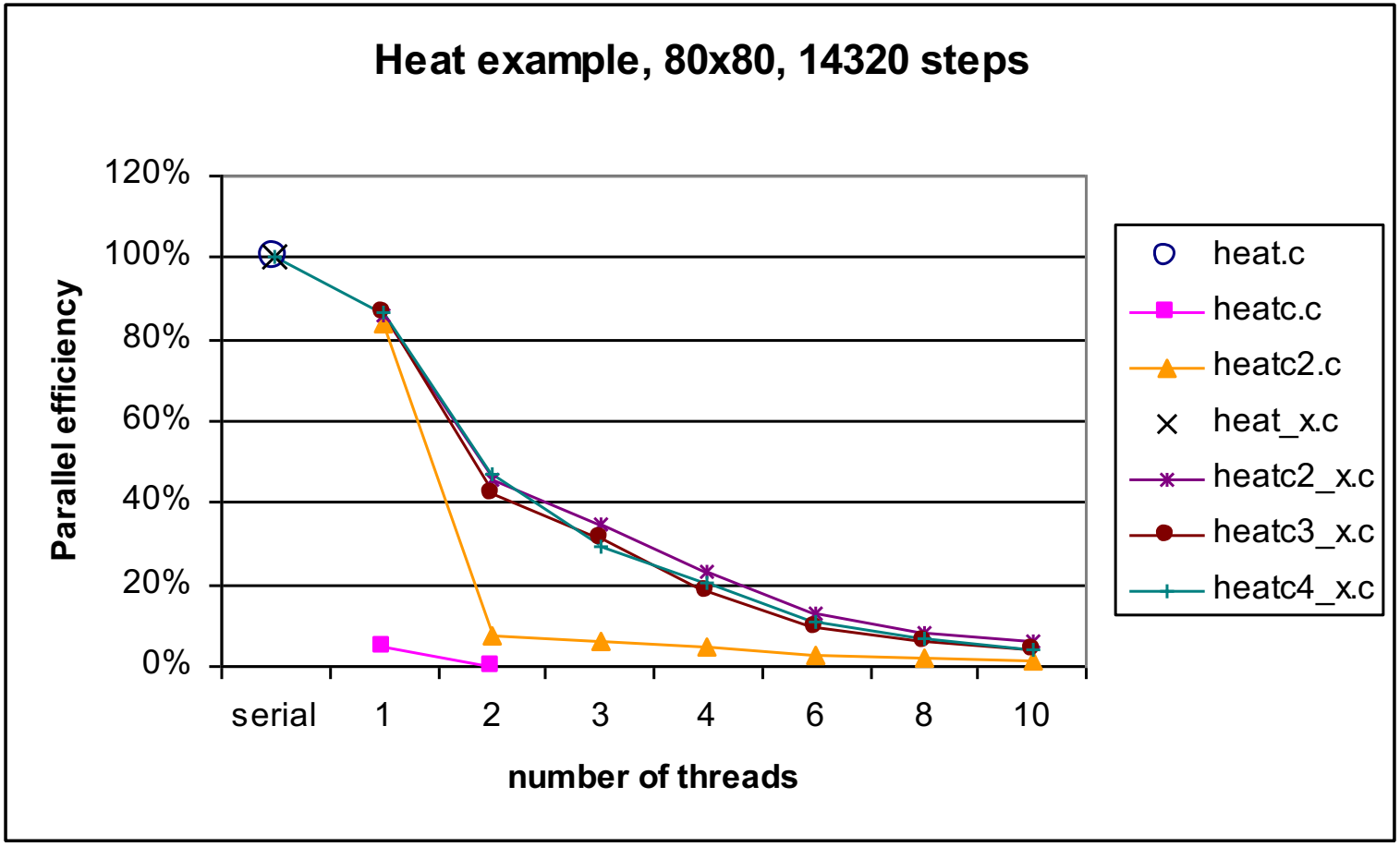


 ● REC → online

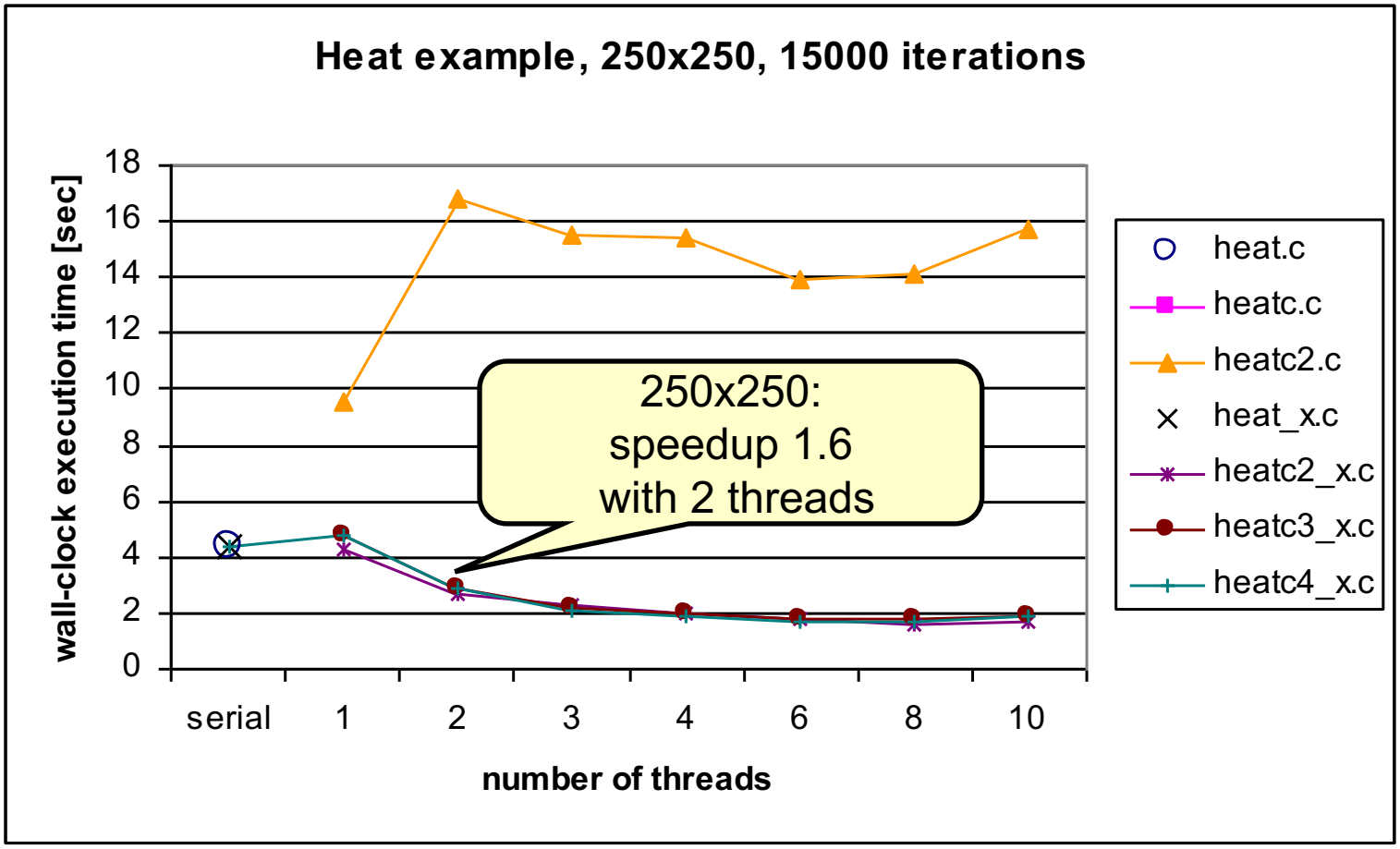
OpenMP Exercise: Heat - Solution (5) – 80x80 Time



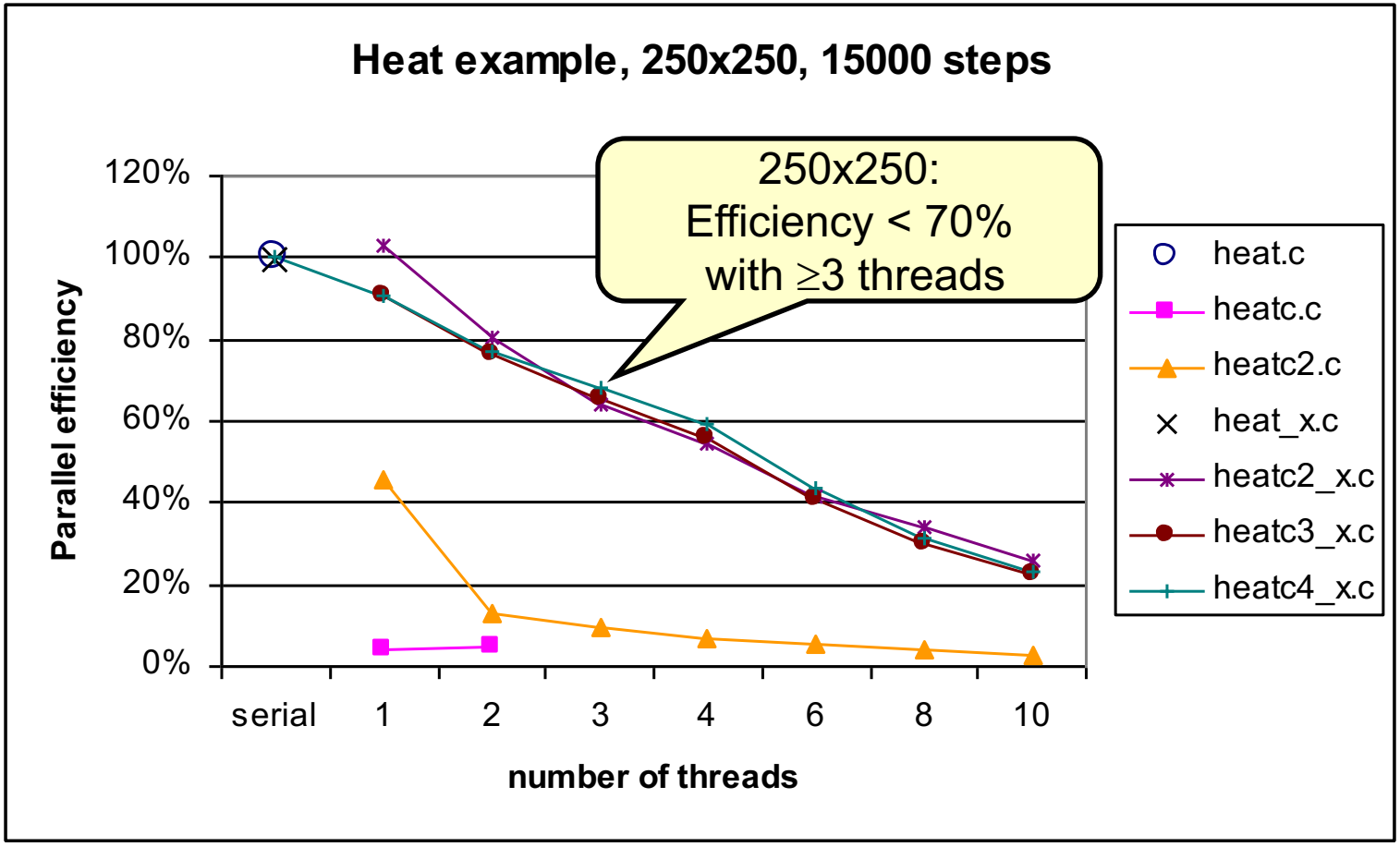
OpenMP Exercise: Heat - Solution (6) – 80x80 Efficiency



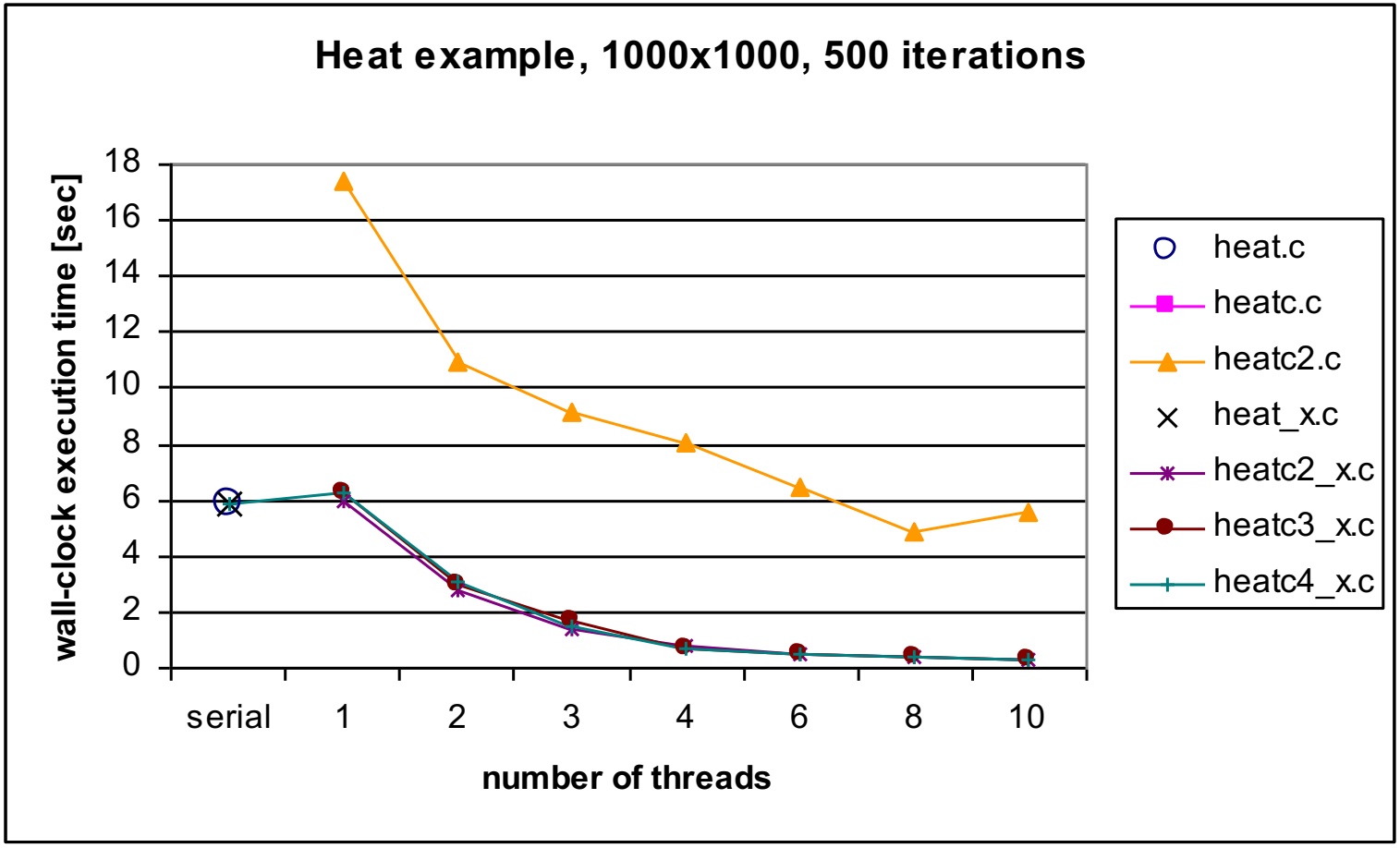
OpenMP Exercise: Heat - Solution (7) – 250x250 Time



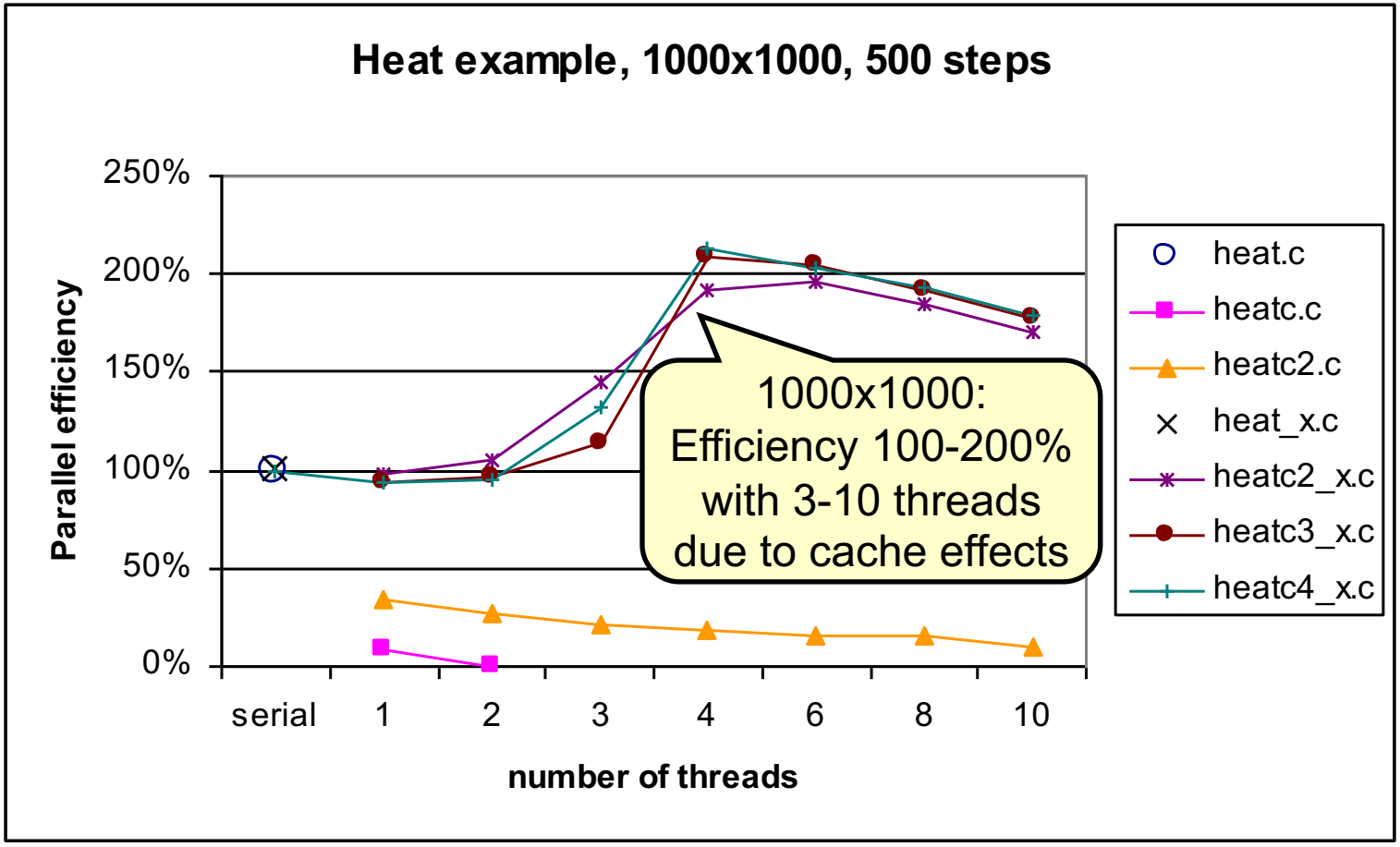
OpenMP Exercise: Heat - Solution (8) – 250x250 Efficiency



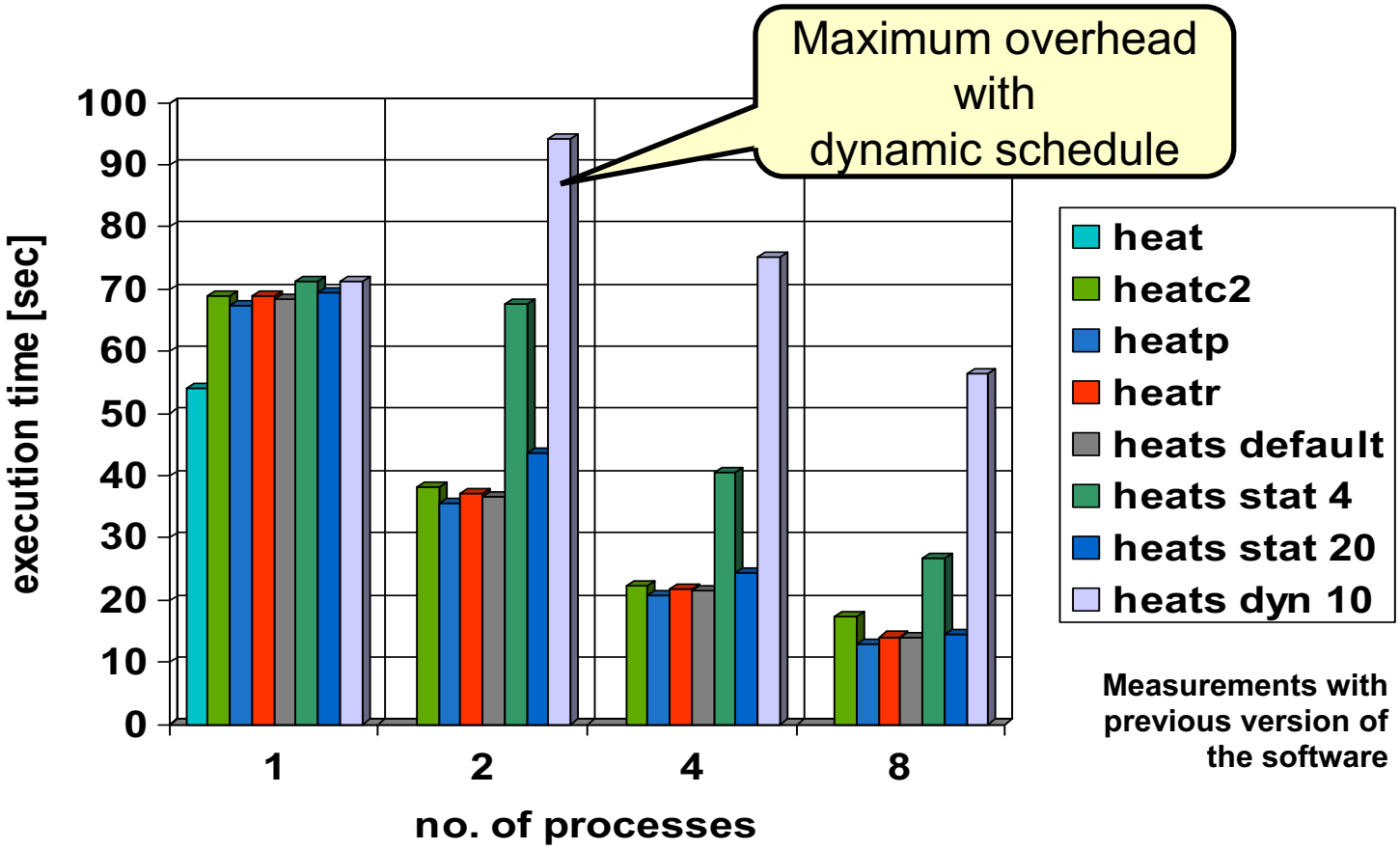
OpenMP Exercise: Heat - Solution (9) – 1000x1000 Time



OpenMP Exercise: Heat - Solution (10) – 1000x1000 Efficiency



OpenMP Exercise: Heat - Execution Times F90 with 150x150



 ● REC → online

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.
- Intel Inspector or another *race condition detection tool* should be used to guarantee zero race conditions.
- Be careful when using other than default scheduling strategies:
  - `dynamic` is generally expensive
  - `static`: overhead for small chunk sizes is clearly visible

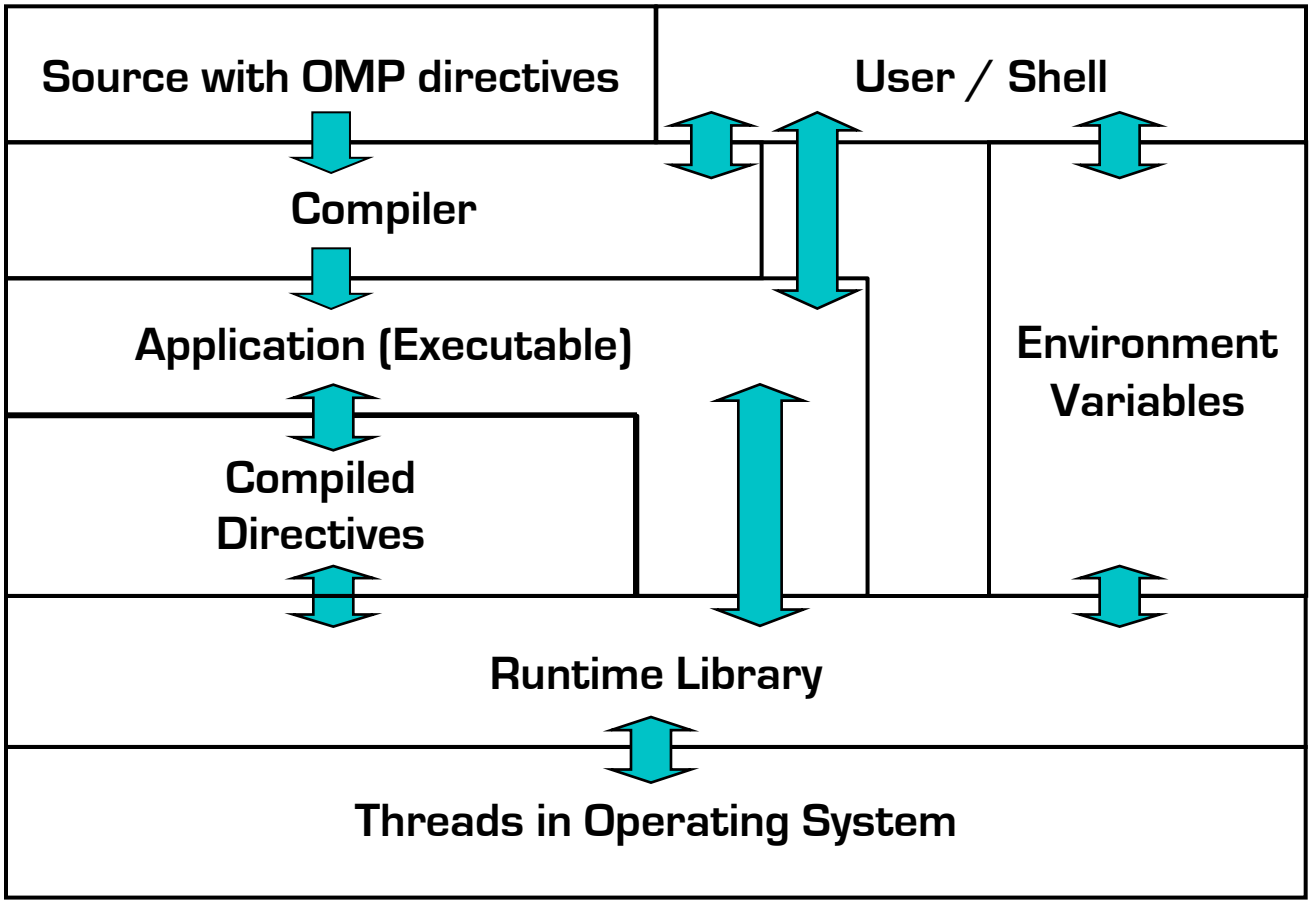
Outline — Summary of the OpenMP API

- 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
- Worksharing directives
  - Which thread executes which statement or operation?
  - Exercise 2a: Pi
  - Tasks
  - Synchronization constructs, e.g., critical regions
  - Nesting and Binding
  - Exercise 2b: Pi
- Data environment and combined constructs
  - Private and shared variables, Reduction clause
  - Combined parallel worksharing directives
  - Exercise 3: Pi with reduction clause and combined constructs
  - Exercise 4: Heat
- **Summary of OpenMP API**
- OpenMP Pitfalls

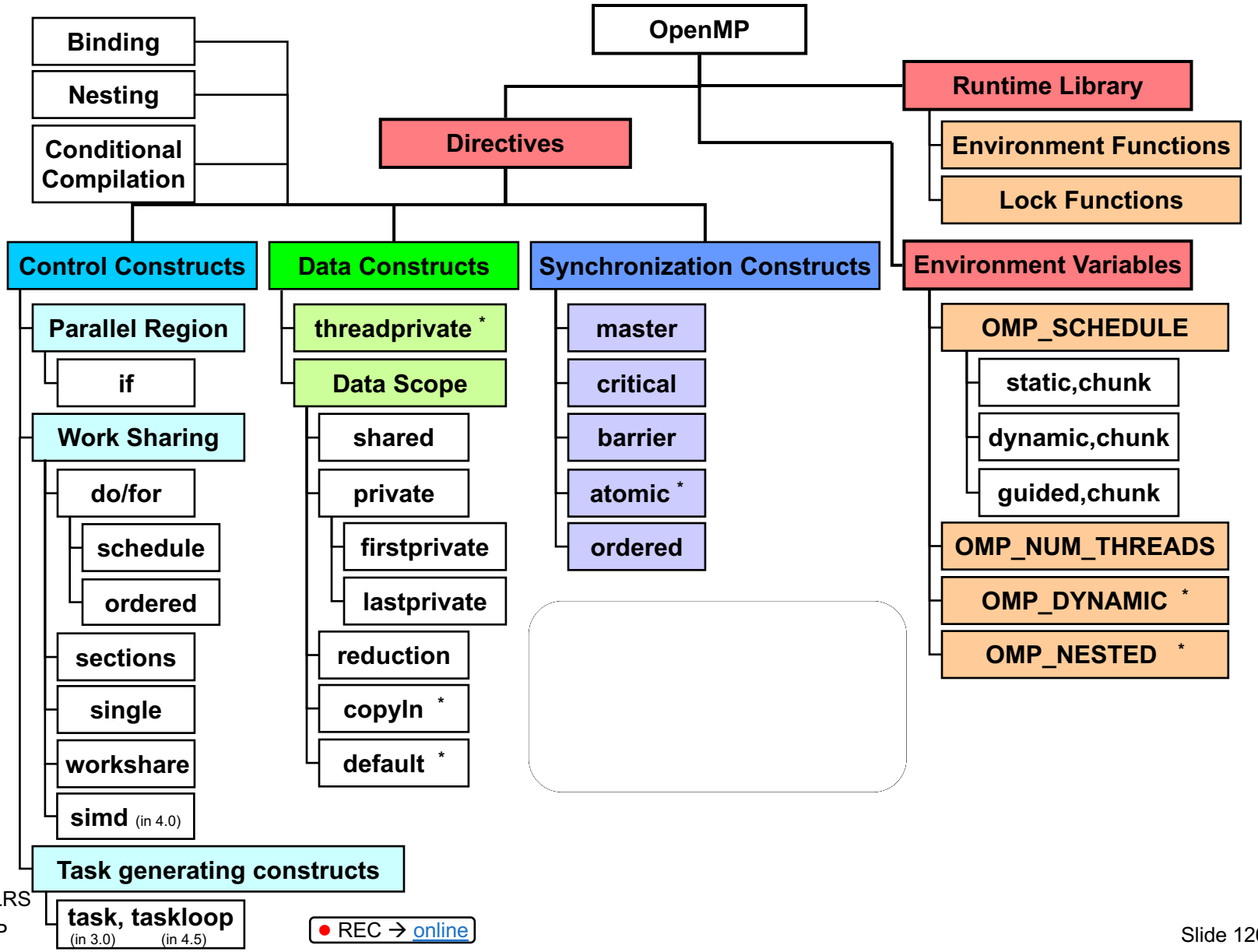
OpenMP Components

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

OpenMP Architecture



OpenMP Constructs



Outline — OpenMP Pitfalls

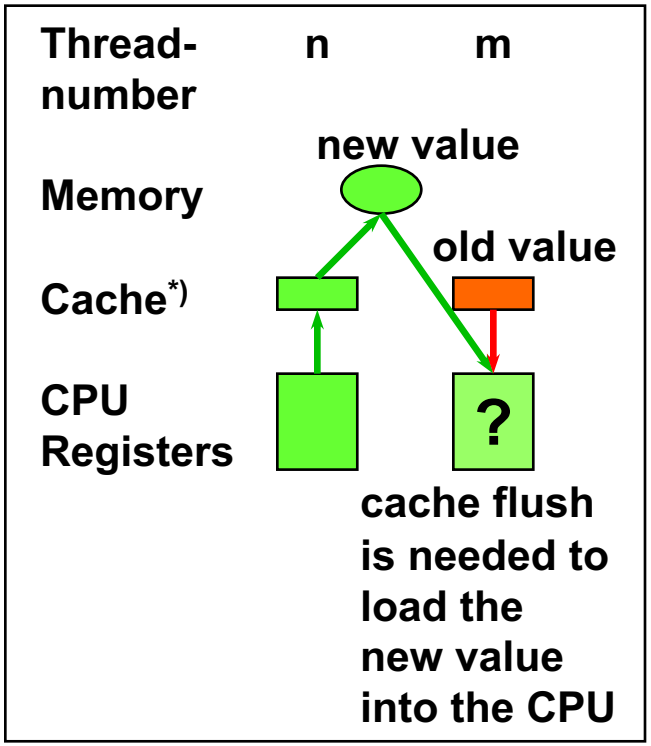
Implementation-defined behavior

See Appendix D of the OpenMP 4.0 standard, e.g.:

- 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
  - Immediately before and after every task scheduling point (OpenMP ≥ 3.0)
- It is NOT implied at the following constructs:
  - Begin of DO, FOR, WORKSHARE, SECTIONS
  - **Begin of MASTER and END MASTER**
  - Begin of SINGLE



*) Faster memory for temporary copies of recently used data

- Recommendation: Do **not** loop with testing on data modifications caused by other threads (without correct and sufficient flushing)
  More secure: Use OpenMP synchronization methods

Two types of SMP errors

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

Example for race condition (1)

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

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

Example for race condition (2)

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

- Write SMP code that is
  - portable and
  - equivalent to the sequential form.

- Use tools like “Intel® Inspector”
  (formerly “Intel® Thread Checker” and “Assure”).

Sequential Equivalence

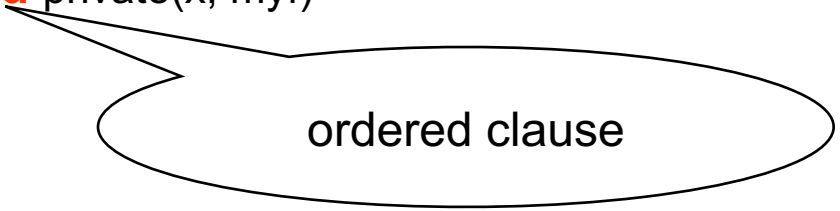
- Two forms of sequential equivalence
  - Strong SE: bitwise identical results.
  - Weak SE: equivalent mathematically but due to quirks of floating point arithmetic, not bitwise identical.
- Using a limited subset of OpenMP
  e.g., no locks
- Advantages:
  - program can be tested,debugged and used in sequential mode
  - this style of programming is also less error prone

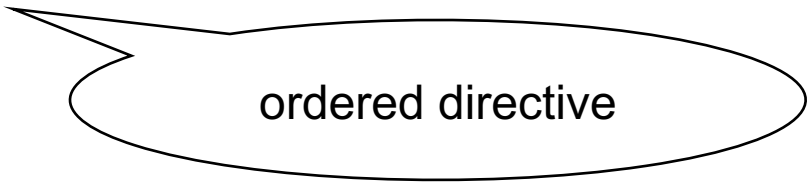
Rules for Strong Sequential Equivalence

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

Example for Ordered Clause: pio.c / .f90

```
#pragma omp for ordered private(x, myf)
 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.

Thread-safe library functions

- Library functions (if called inside of parallel regions) must be thread-safe
- **Automatically switched** if OpenMP option is used:
  - e.g., Intel compiler:
    - ifort -qopenmp -o *my_prog* *my_prog.f* or *my_prog.f90* (Fortran)
    - icc -qopenmp -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_prog.f90* (Fortran, free form)
    - xlc**_r** -O -qsmp=omp -o *my_prog* *my_prog.c* (C)
    - **The "_r" forces usage of reentrant library functions** (not identical with thread-safe)
- **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) → 3rd next slide
- Prevent unnecessary cache-coherence or memory communication
  - E.g., same schedules for same memory access patterns → 6th next slide
  - First touch on (cc)NUMA architectures → 8th next slide
    - **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**
    - **numactl, or LIKWID (portable !!!)**
    - dplace –x2 (SGI), O(1) scheduler (HP), SUNW_OMP_PROBIND envir. (Sun)
    - fplace –r –o 1,2 *command* (Intel), …

      Do not pin the management threads

Get a feeling for the involved overheads

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

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

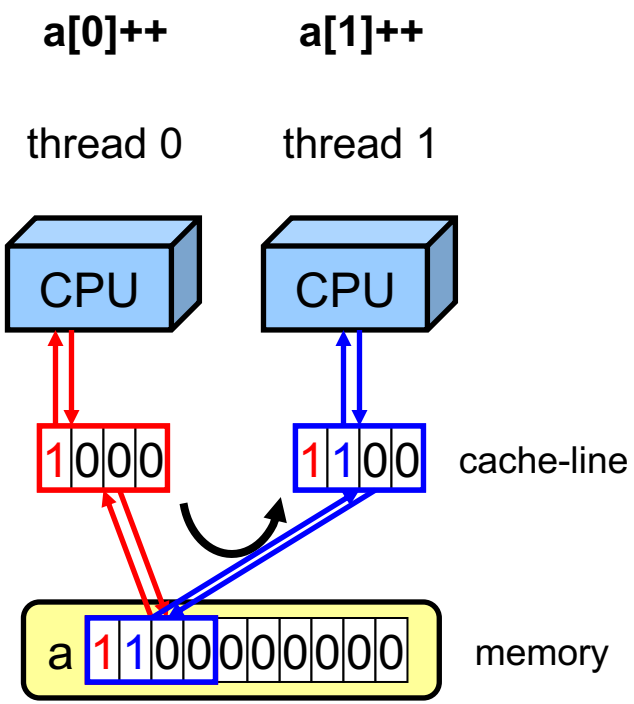
Optimization Problems

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

```
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

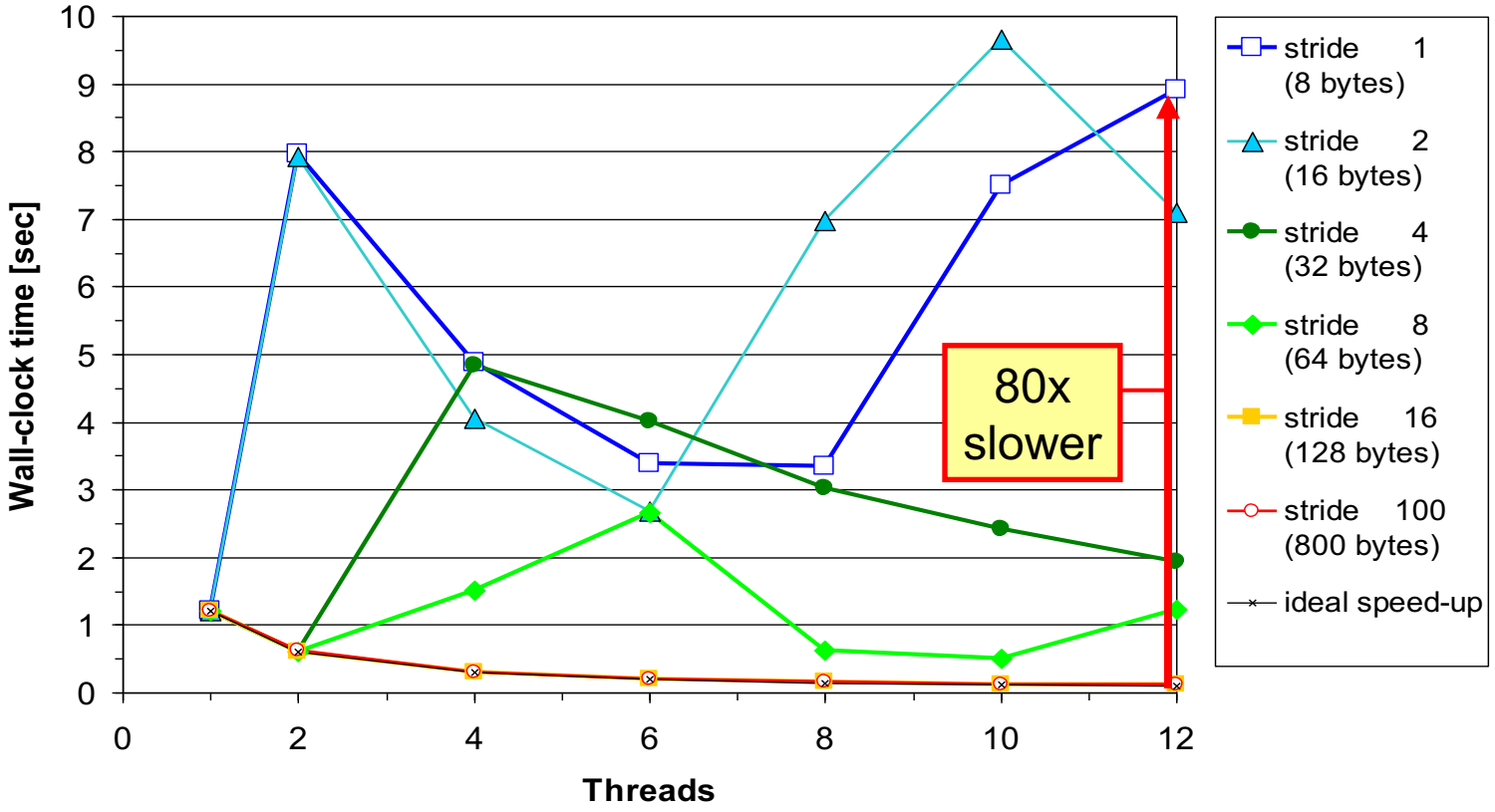
False-sharing



- 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.

*) The smallest accessible unit in memory and caches, usually 64 or 128 bytes, i.e., 8 or 16 doubles

False-sharing – results from the experiment

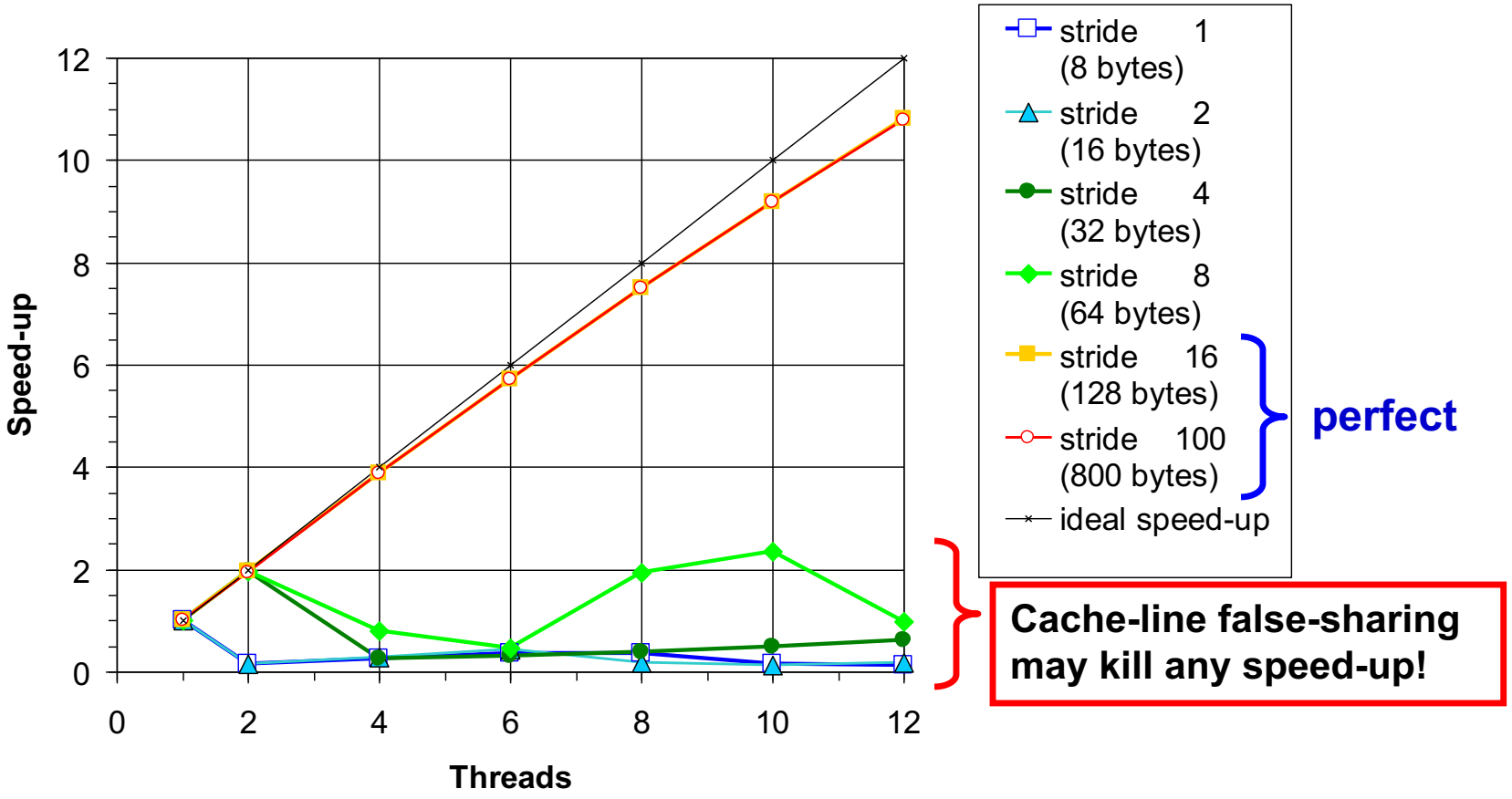


Although each thread accesses independent variables, the performance will be terrible, if these variables are located in the same cache-line

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.

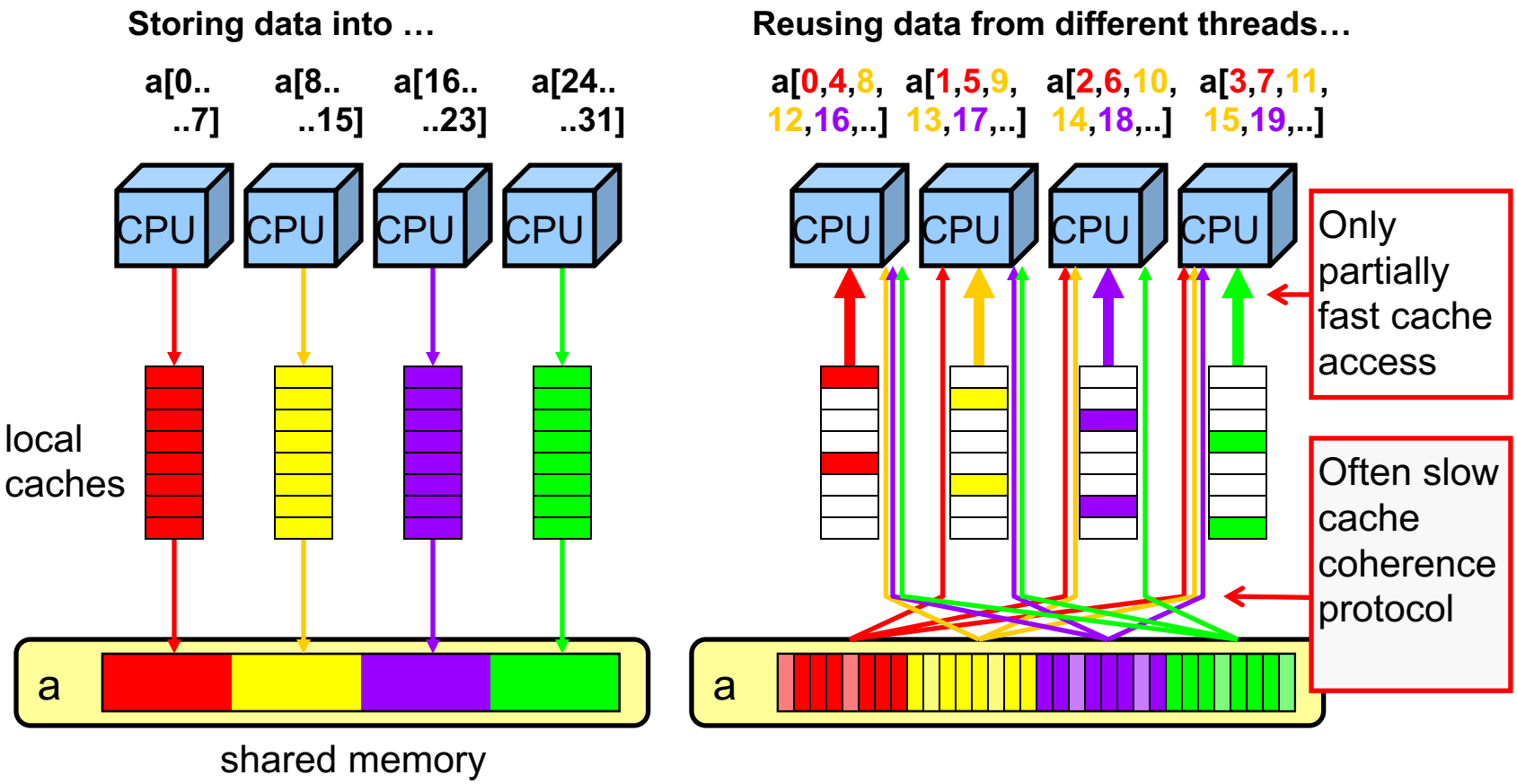
Test program, see Appendix: piarr.c

False-sharing – same with speed-up

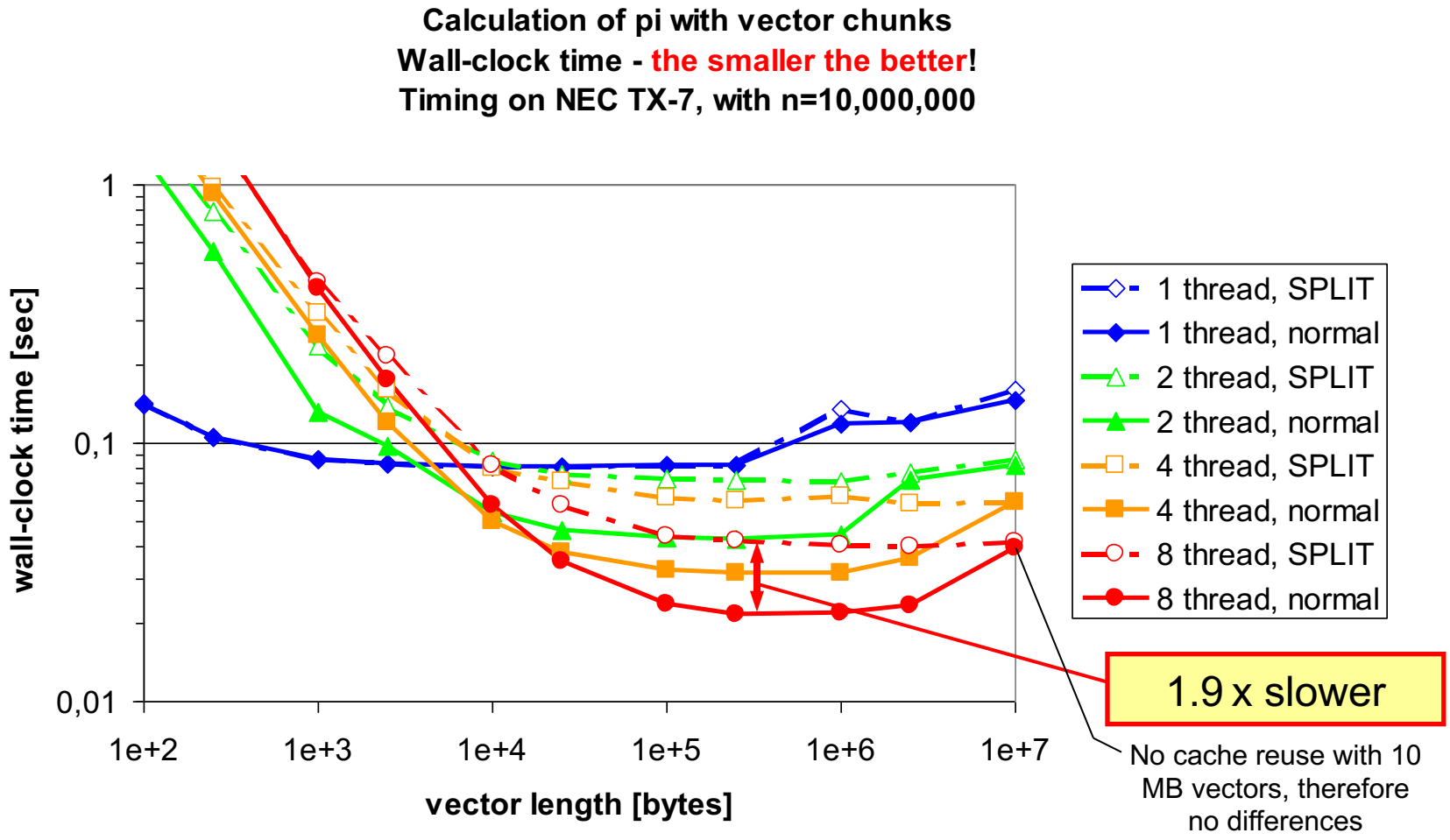


Measurements on NEC TX-7 with **128 bytes cache lines**,
timings with false-sharing (stride 1-8 with more than 1 thread) were varying from run to run.

Communication overhead, e.g., due to different schedules



Communication overhead – results from the experiment



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

First touch

First write of a byte in a memory page
→ memory page is located in the physical memory of the executing thread

Don't use **c**alloc() with OpenMP – no first touch
whole array is already mapped to physical memory

```
#define n 1000000
double *x; int i;
x = (double *) malloc(n*sizeof(double));
```

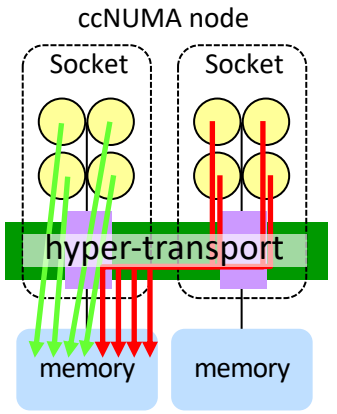
malloc() does not specify the physical location of the memory pages of the array!

```
// sequential initialization of the data
for (i=0 ; i<n; i++) x[i]=0;
```

First touch only by master thread
→ Whole array is in 1st CPU's memory

```
#pragma omp parallel
{  // parallelized numerical loop
  #pragma omp for schedule(static)
    for (i=0 ; i<n; i++) x[i]=huge_computation(i);
}
```

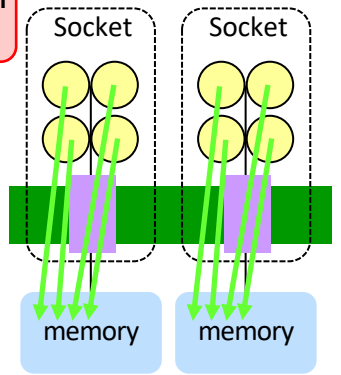
→ slower accesses by threads on 2nd CPU



impossible with any dynamic schedule or taskloops (i.e., only with static schedule)

```
#pragma omp parallel
{// parallel initialization of the data
  #pragma omp for schedule(static)
    for (i=0 ; i<n; i++) x[i]=0;
 // parallelized numerical loop
  #pragma omp for schedule(static)
    for (i=0 ; i<n; i++) x[i]=huge_computation(i);
}
```

Parallelized first touch **with same schedule** as in the numerical loop



→ Fast accesses by all threads because x[i] is in the own CPU's memory

Acknowledgements

Thanks to

- Rainer Keller (Hochschule für Technik, Stuttgart),
- Matthias Müller (RWTH Aachen),
- Isabel Loebich,

the original co-authors of this course.

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 the Intel® Inspector help to write correct parallel programs

Appendix

Appendix – Content

- Example hello – parallel region – written in C and Fortran
- Example pi, written in C
- 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)
- Cache-line false-sharing experiment: piarr.c
- Communication overhead / different schedules experiment: pivec.c

Example hello – Parallel region

- hello6.c – final solution in C
- hello6.f90 – final solution in Fortran

hello5.c – Parallel region

```
#include <stdio.h>
#include <unistd.h>

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

int main(int argc, char** argv)
{
  int i;

    i = -1;

#pragma omp parallel private(i)
  {
# ifdef _OPENMP
    i = omp_get_thread_num();
# endif
    sleep(1);
    printf( "hello world %d\n", i );
  }

  return 0;
}
```

Conditional compilation because library routines are available only within OpenMP

Header file for OpenMP library routines

Parallel region

Private-clause that each thread has its own private `i` variable

Conditional compilation because library routines are available only within OpenMP

Query the *my_rank* of the thread that calls this function

With this sleep(1)-trick between storing data into `i` and reading `i` in the print statement, one would see the race-condition in the output if one forgets to make `i` private. **(Do not do this in programs in production!)**

hello5.f90 – Parallel region

```
program hello

integer i
!$ integer OMP_GET_THREAD_NUM

   i=-1

!$OMP PARALLEL PRIVATE(i)

!$ i=OMP_GET_THREAD_NUM()
   call sleep(1)
   print *,'hello world',i

!$OMP END PARALLEL

end program hello
```

Conditional compilation because library routines are available only within OpenMP

Direct declaration of the result type of the library function.
You may prefer to use the general statement **USE omp** (although not 100% portable).

Parallel region

Private-clause that each thread has its own private `i` variable

Conditional compilation because library routines are available only within OpenMP

Query the *my_rank* of the thread that calls this function

With this sleep(1)-trick between storing data into `i` and reading `i` in the print statement, one would see the race-condition in the output if one forgets to make `i` private. **(Do not do this in programs in production!)**

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

The include and timing blocks are removed on the next slides

```
#include <stdio.h>
#include <time.h>
#include <sys/time.h>
#ifdef _OPENMP
#  include <omp.h>
#endif
```
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;
```

```
  clock_t t1,t2;
  struct timeval tv1,tv2;
  struct timezone tz;
# ifdef _OPENMP
    double wt1,wt2;
# endif
```
timing block A

```
# ifdef _OPENMP
#   pragma omp parallel
    {
#     pragma omp single
      printf("OpenMP-parallel with %1d
      threads\n", omp_get_num_threads());
    } /* end omp parallel */
# endif
# pragma omp barrier
```
OpenMP check

```
  gettimeofday(&tv1, &tz);
# ifdef _OPENMP
    wt1=omp_get_wtime();
# endif
  t1=clock();
```
timing block B

```
/* 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;
```
the calculation

```
  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 );
```
timing block C

```
  return 0;
}
```

pi0.c – only verification of team of threads – without parallelization

```
--- 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;
}
```

pi2.c
parallelization with race-condition, i.e., WRONG program

```
--- 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 ---
/* 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
  for (i=1;i<=n;i++)
  {
    x=w*((double)i-0.5);
    sum=sum+f(x);
  }
} /*end omp parallel*/
  pi=w*sum;
  --- TIMING BLOCK C ---
  return 0;
}
```

Read/write race-conditions on shared variable **sum.**
Or wrong results if **sum** would be declared as private.

pic2.c
parallelization with critical region outside of the loop

```
--- 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 nowait
  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;
}
```

pir.c – parallelization with reduction 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,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);
  }
} /*end omp parallel*/
  pi=w*sum;

  --- TIMING BLOCK C ---
  return 0;
}
```

pir2.c – combined `parallel for` with reduction 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,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;
}
```

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;
    }
  }
} /*end omp parallel*/
  pi=w*sum;

  --- TIMING BLOCK C ---
  return 0;
}
```

Most compute time should be outside of the ordered region

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,1)
  for (i=0;i<num_threads;i++)
  {
#pragma omp ordered
    sum=sum+sum0;
  }
} /*end omp 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.c

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

pi.f90 – sequential code

The timing blocks are removed on the next slides

```
program compute_pi
implicit none
! times using cpu_time
real t0,t1
!--unused-- use omp_lib
!$ double precision omp_get_wtime
!$ double precision wt0,wt1
```
timing block A.f

```
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)
```

```
…
!$  write (*,*) 'OpenMP-parallel with',omp_get_num_threads(),'threads'
…
```
OpenMP check

```
!$ wt0=omp_get_wtime()
call cpu_time(t0)
```
timing block B.f

```
! calculate pi = integral [0..1] 4/(1+x**2) dx
w=1.0_8/n
sum=0.0_8
do i=1,n
   x=w*(i-0.5_8)
   sum=sum+f(x)
enddo
pi=w*sum
```
the calculation

```
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-wt0
end program compute_pi
```
timing block C.f

pi0.f90 – only verification of team of threads – without parallelization

```
program compute_pi
implicit none
--- TIMING BLOCK A ---
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()
!$  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=1.0_8/n
sum=0.0_8
do i=1,n
   x=w*(i-0.5_8)
   sum=sum+f(x)
enddo
pi=w*sum
--- TIMING BLOCK C ---
end program compute_pi
```

pi2.f90
parallelization with race-condition, i.e., WRONG program

```
program compute_pi
implicit none
--- TIMING BLOCK A ---
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 ---
--- TIMING BLOCK B ---
! calculate pi = integral [0..1] 4/(1+x**2) dx
w=1.0_8/n
sum=0.0_8
!$OMP PARALLEL PRIVATE(x), SHARED(w,sum)
!$OMP DO
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
```

Read/write race-conditions on shared variable **sum.**
Or wrong results if **sum** would be declared as private.

pic2.f90
parallelization with critical region outside of the loop

```
program compute_pi
implicit none
--- TIMING BLOCK A ---
integer i
integer, parameter :: n=10000000
real(kind=8) w,x,sum,sum0,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
!$OMP PARALLEL PRIVATE(x,sum0), SHARED(w,sum)
sum0=0.0_8
!$OMP DO
do i=1,n
   x=w*(i-0.5_8)
   sum0=sum0+f(x)
enddo
!$OMP END DO NOWAIT
!$OMP CRITICAL
   sum=sum+sum0
!$OMP END CRITICAL
!$OMP END PARALLEL
pi=w*sum
--- TIMING BLOCK C ---
end program compute_pi
```

pir.f90 – parallelization with reduction clause

```
program compute_pi
implicit none
--- TIMING BLOCK A ---
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 ---
--- TIMING BLOCK B ---
! calculate pi = integral [0..1] 4/(1+x**2) dx
w=1.0_8/n
sum=0.0_8
!$OMP PARALLEL PRIVATE(x), SHARED(w,sum)
!$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

```
program compute_pi
implicit none
--- TIMING BLOCK A ---
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 ---
--- TIMING BLOCK B ---
! calculate pi = integral [0..1] 4/(1+x**2) dx
w=1.0_8/n
sum=0.0_8
!$OMP PARALLEL DO PRIVATE(x), SHARED(w), REDUCTION(+:sum)
do i=1,n
   x=w*(i-0.5_8)
   sum=sum+f(x)
enddo
!$OMP END PARALLEL DO
pi=w*sum
--- TIMING BLOCK C ---
end program compute_pi
```

pio.f90 – parallelization with ordered clause

```
program compute_pi
implicit none
--- TIMING BLOCK A ---
integer i
integer, parameter :: n=10000000
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
!$OMP END DO
!$OMP END PARALLEL
pi=w*sum
--- TIMING BLOCK C ---
end program compute_pi
```

Most compute time should be outside of the ordered region

CAUTION

The sequentialization of the ordered region may cause heavy synchronization overhead

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

```
...
!--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()
!$OMP DO SCHEDULE(STATIC,(n-1)/num_threads+1)
do i=1,n
   x=w*(i-0.5_8)
   sum0=sum0+f(x)
enddo
!$OMP END DO
!$OMP DO ORDERED SCHEDULE(STATIC,1)
do i=1,num_threads
!$OMP ORDERED
   sum=sum+sum0
!$OMP END ORDERED
enddo
!$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.f90

heatc2_x.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
```


heatc2_x.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 (i=1;i<imax;i++)
  { for (k=0;k<kmax;k++)
    { 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);
  printf("\nstart values\n");
  heatpr(phi);
```

heatc2_x.c (page 3 of 4) – time step integration

```
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 (i=1;i<imax;i++)
    { for (k=0;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;
        dphimax0=max(dphimax0,dphi);
        phin[i][k]=phi[i][k]+dphi;
      }
    }
#pragma omp critical
{   dphimax=max(dphimax,dphimax0);
}
/* save values */
#pragma omp for
    for (i=1;i<imax;i++)
    { for (k=0;k<kmax;k++)
      { phi[i][k]=phin[i][k];
      }
    }
}/*end omp parallel*/
    if(dphimax<eps) break;
  }
```

In C, phi and phin are contiguous in the last index [k]. Therefore the k-loop should be the inner loop! This optimization is done in all heat..._x.c versions

heatc2_x.c (page 4 of 4) – wrap up

```
# ifdef _OPENMP
    wt2=omp_get_wtime();
# endif
  gettimeofday(&tv2, &tz);
  printf("\nphi 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=k+kl)
  { if(k+kkk>kmax) kkk=kmax-k;
    printf("\ncolumns %5d to %5d\n",k,k+kkk);
    for (i=0;i<=imax;i++)
    { printf("%5d ",i);
      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

```
      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

```
!$OMP PARALLEL PRIVATE(i,k), SHARED(phi)
c start values 0.d0
!$OMP DO
      do k=0,kmax-1
        do i=1,imax-1
          phi(i,k)=0.d0
        enddo
      enddo
!$OMP END DO
c start values 1.d0
!$OMP DO
      do i=0,imax
        phi(i,kmax)=1.d0
      enddo
!$OMP END DO
!$OMP END PARALLEL
c start values dx
      phi(0,0)=0.d0
      phi(imax,0)=0.d0
      do k=1,kmax-1
      phi(0,k)=phi(0,k-1)+dx
      phi(imax,k)=phi(imax,k-1)+dx
      enddo
c print array
      write (*,'(/,a)')
     f      'Heat Conduction 2d'
      write (*,'(/,4(a,1pg12.4))')
     f      'dx =',dx,', dy =',dy,', dt =',dt,', eps =',eps
      write (*,'(/,a)')
     f      'start values'
      call heatpr(phi,imax,kmax)
```

heatr2_x.f (page 3 of 4) – time step integration

```
!$     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
     f          +(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
!$OMP END DO
c save values
!$OMP DO
       do k=1,kmax-1
         do i=1,imax-1
           phi(i,k)=phin(i,k)
         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 heat..._x.f versions

heatr2_x.f (page 4 of 4) – wrap up

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

c print array
      write (*,'(/,a,i6,a)')
     f      'phi after',it,' iterations'
      write (*,'(/,a,1pg12.4)')  'cpu_time  :   ', t1-t0
!$    write (*,'(/,a,1pg12.4)')  'omp_get_wtime:', wt1-wt0
c
      stop
      end
c
c
      subroutine heatpr(phi,imax,kmax)
      double precision phi(0:imax,0:kmax)
c
      kl=6
      kkk=kl-1
      do k=0,kmax,kl
      if(k+kkk.gt.kmax) kkk=kmax-k
      write (*,'(/,a,i5,a,i5)') 'columns',k,' to',k+kkk
      do i=0,imax
      write (*,'(i5,6(1pg12.4))') i,(phi(i,k+kk),kk=0,kkk)
      enddo
      enddo
c
      return
      end
```

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;
```

The thread-locality of psum[index] variables is forced manually, and **in the same cache-line** if **stride is small enough**

Communication overhead – an experiment (solution/pi/pivec.c)

```
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;
```

2 variants

Normal:

```
#pragma omp for
  for (i=j;i<=j_e;i++)
  {
    sum0=sum0+vec[i-j];
  }
#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=j;i<(j+j_e)/2;i++)
  {
    sum0=sum0+vec[i-j];
  }
#pragma omp for
  for (i=(j+j_e)/2;i<=j_e;i++)
  {
    sum0=sum0+vec[i-j];
  }
```