Master Course – High-Performance Computing

Parallel Programming with OpenMP
Part I

Olaf Schenk
Institute of Computational Science
USI Lugano, Switzerland
October 16, 2019

Outline

- 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?**
  - **Synchronization constructs, e.g., critical regions**
  - **Nesting and Binding**
  - **Exercise 2: Pi**
- Data environment and combined constructs
  - **Private and shared variables, Reduction clause**
  - **Combined parallel worksharing directives**
  - **Exercise 3: Pi with reduction clause and combined constructs**
  - **Exercise 4: Heat**
- Summary of OpenMP API
- OpenMP Pitfalls & Optimization Problems

Multiprocessor System with Shared Memory

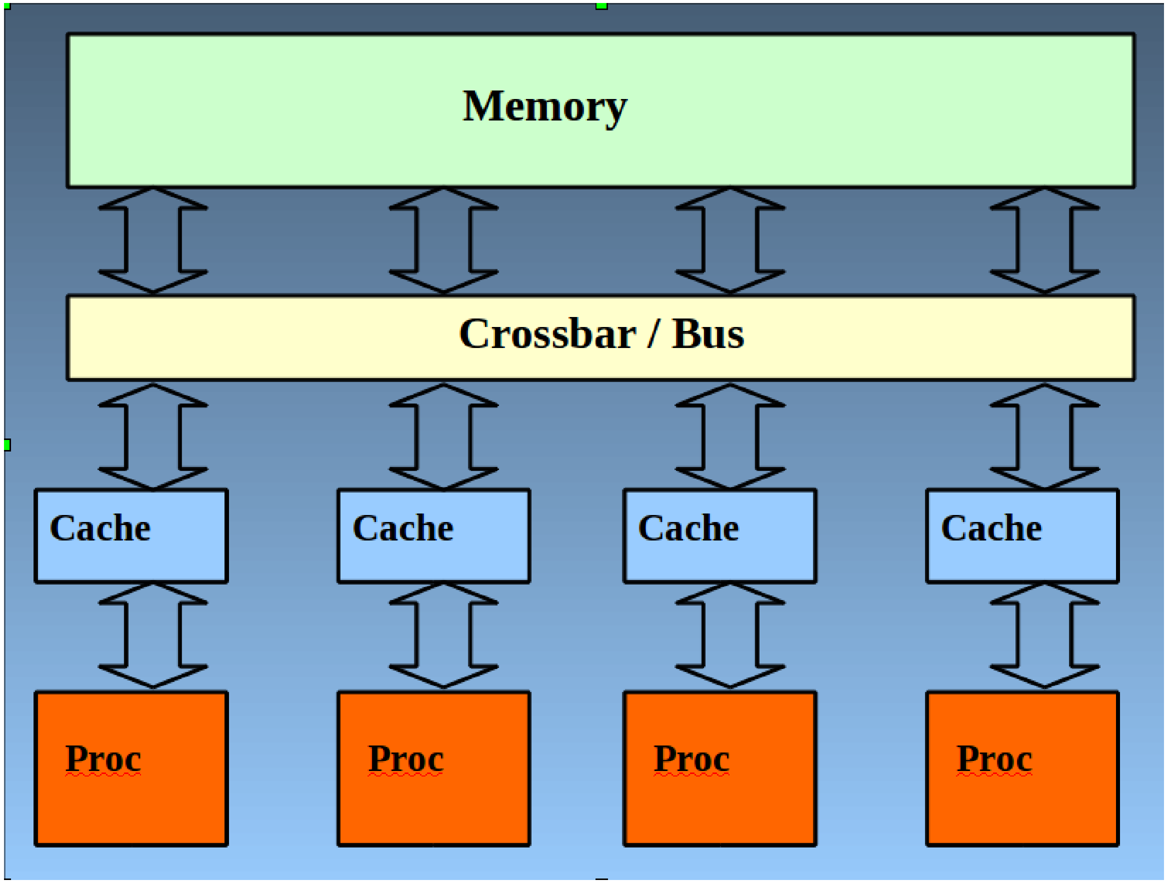
OpenMP

is a

parallel programming model

for

Shared-memory multiprocessors



OpenMP Overview: What is OpenMP?

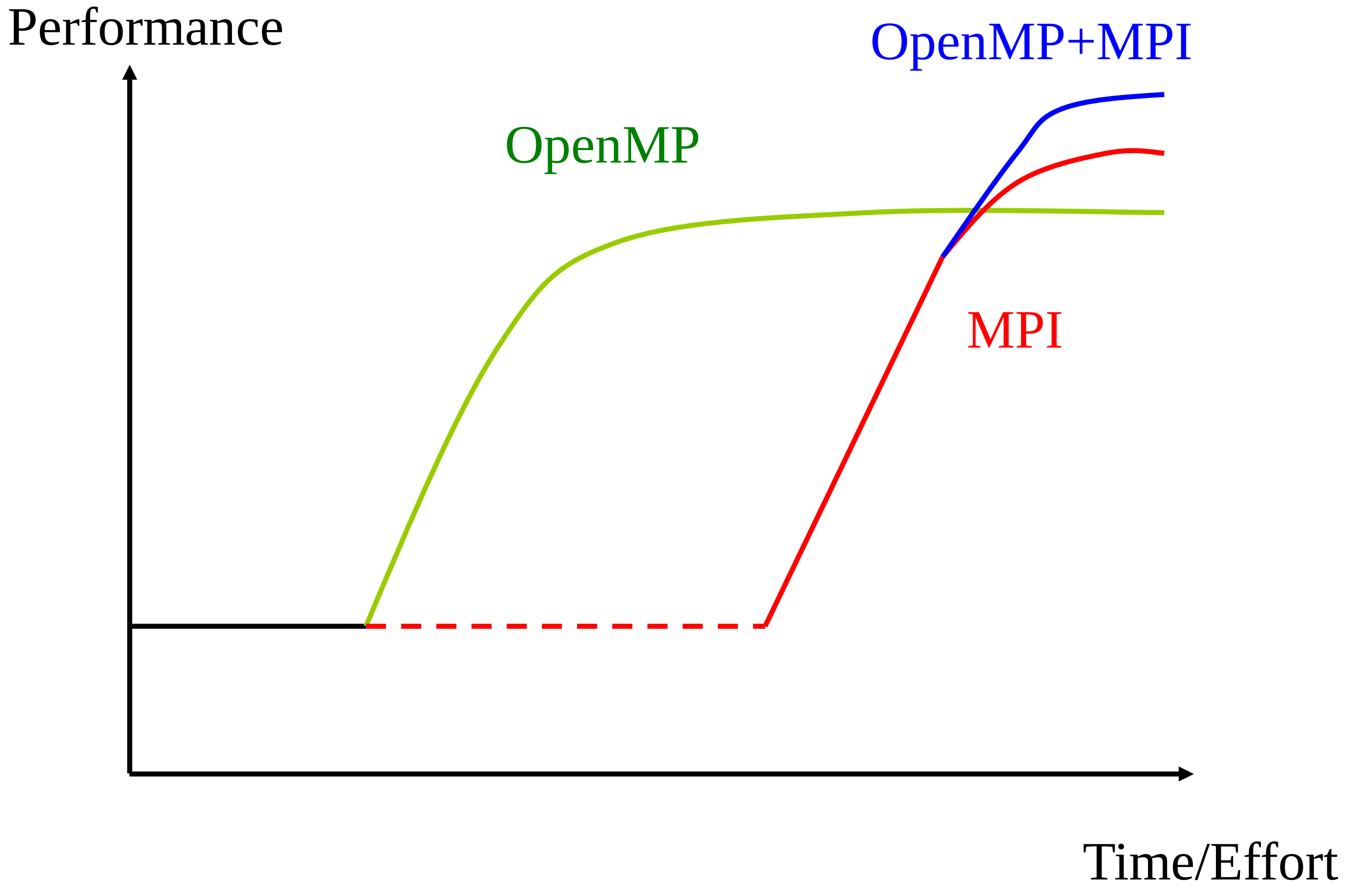
- OpenMP is a standard programming model for shared memory parallel programming
- Portable across all shared-memory architectures
- It allows incremental parallelization
- Compiler based extensions to existing programming language
  - mainly by directives
  - a few library routines
- Fortran and C/C++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

Parallel Computing: Effective Standards for Portable programming

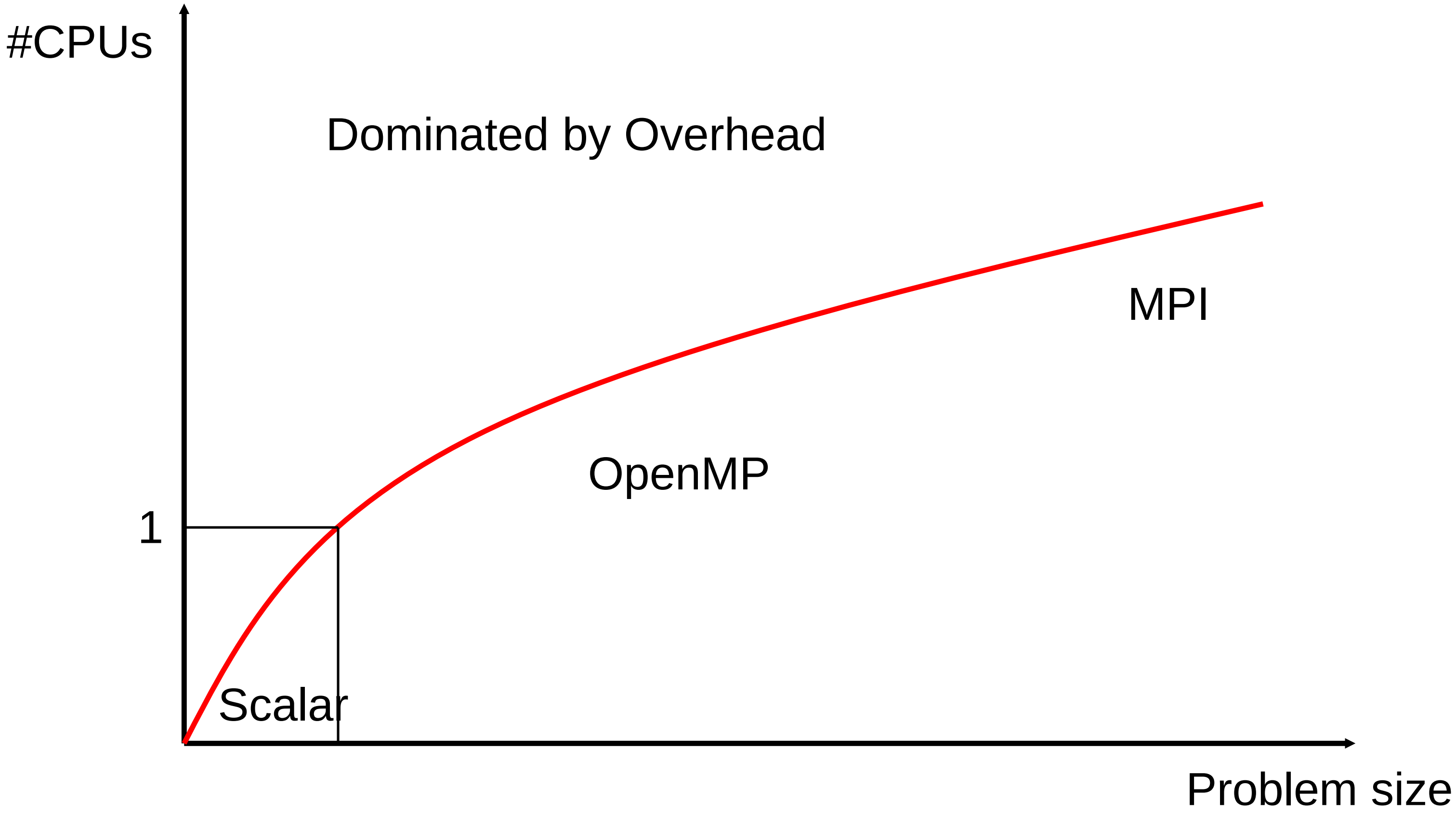
- Thread Libraries
  - Win32 API
  - POSIX threads
- Compiler Directives
  - **OpenMP - portable shared memory parallelism**
  - OpenACC, OpenCL
- Global Address Space Languages
  - Unified Parallel C
  - Titanium
  - X10
  - Chapel
- Message Passing Libraries
  - MPI

Our focus

Motivation: Why should I use OpenMP?



Where should I use OpenMP?



Multithreading versus Multi-Processing

- **Multiple Processes** (Heavyweight Process model)
  - traditional UNIX process model
  - interprocess communication techniques supported by OS: shared memory, sockets, file IO, memory map
  - Higher overhead associated with process creation and destruction
- **Multiple Threads** (Lightweight Process model, LWP)
  - thread concept: independent flow of control within one process with its own context: stack, register set
  - process data and opened files are shared
  - lower overhead of thread creation and destruction
  - shared address space
  - Auto-Parallelization, OpenMP, Explicit Multithreading using P-Threads
- **Hybrid Models** (e.g. MPI + OpenMP)

OpenMP - History

- 1997: OpenMP Version 1.0 for Fortran
  - de facto standard for shared memory programming
  - now available for all SMP systems
  - replaces proprietary parallelization directives and in many cases the explicit programming of [p]threads
  - 1998: OpenMP V1.0 for C and C++
  - 1999: OpenMP V1.1 for Fortran (error corrections, explanations)
  - 2000: OpenMP V2.0 for Fortran (support of Fortran90 modules)
  - 2001: OpenMP V2.0 for C and C++
  - 2008: OpenMP V3.0 for Fortran/ C and C++
  - 2014: OpenMP V4.0 for Fortran/C and C++

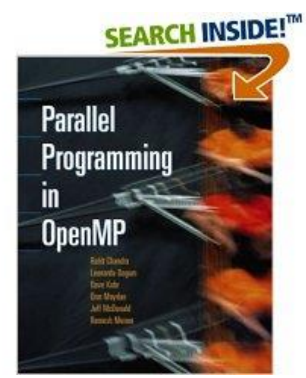
OpenMP - Information

- The OpenMP Architecture Review Board (ARB) Fortran and C Application Program Interfaces (APIs): www.openmp.org

- NCSA online course on OpenMP:
  http://www.citutor.org/login.php?course=24
  (you need to register for a free account, very useful)

- OpenMP tutorial from Lawrence Livermore National Laboratory:
  https://computing.llnl.gov/tutorials/openMP/

- Book: Rohit Chandra, et.al. ,,Parallel Programming in OpenMP"
  Morgan Kaufmann, ISBN 1-55860-671-8



OpenMP - Overview:

`C$OMP FLUSH`

`#pragma omp critical`

`C$OMP THREADPRIVATE(/ABC/)`

`CALL OMP_SET_NUM_THREADS(10)`

`C$OMP parallel do shared(a, b, c)`

`call omp_test_lock(jlok)`

- OpenMP: An API for Writing Multi-threaded Application
  - A set of compiler directives and library routines for parallel application programmers
  - Makes it easy to create multi-threaded (MT) programs in Fortran, C and C++
  - Standardizes last 15 years of SMP practice

`#pragma omp parallel for private(A, B)`

`!$OMP BARRIER`

`C$OMP PARALLEL COPYIN(/blk/)`

`C$OMP DO lastprivate(XX)`

`Nthrds = OMP_GET_NUM_PROCS()`

`omp_set_lock(lck)`

OpenMP Overview: How is OpenMP typically used?

- OpenMP is usually used to parallelize loops:
  - Find your most time consuming loops.
  - Split them up between threads.

Split-up this loop between multiple threads

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

Sequential Program

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

Parallel Program

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?
  - Synchronization constructs, e.g., critical regions
  - Exercise 2: Pi
- Data environment and combined constructs
  - Nesting and Binding
  - 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 & Optimization Problems

Components - Environment, Variables, Directives, Runtime

environment variables

```
#!/bin/tcsh
# Shell-Script
gcc -fopenmp test.c
setenv OMP_NUM_THREADS 4
a.out
```

directives

(special comment lines)

runtime library

```
/* Source file test.c */
#include <stdio.h>
#include <omp.h>
int main(void)
{
   #pragma omp parallel
   printf("me: %d\n",omp_get_thread_num());
}
```

```
me: 0
me: 3
me: 2
me: 1
```

OpenMP Overview: How do threads interact?

- OpenMP is a shared memory model.
  - Threads communicate by sharing variables.
- Unintended sharing of data causes 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.
- Synchronization is expensive so:
  - Change how data is accessed to minimize the need for synchronization.

OpenMP: Structured blocks

- Most OpenMP constructs apply to structured blocks
- Structured block: a block with one point of entry at the top and one point of exit at the bottom.
- The only "branches" allowed are exit() in C/C++.

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  more:
    res(id) = do_big_job(id);
  if( conv(res(id) ) goto more;
}
printf(" All done \n");
```

A structured block

```
if(go_now()) goto more;
#pragma omp parallel
{
  int id = omp_get_thread_num();
  more:
    res(id) = do_big_job(id);
  if( conv(res(id) ) goto done;
  go to more;
}
done:
  if(!really_done()) goto more;
```

Not a structured block

OpenMP: Structured blocks / Conditional Compilation

- In C/C++: a block is a single statement or a group of statements between brackets {}

```
#pragma omp parallel
{
   id = omp_thread_num();
   res(id) =
   lots_of_work(id);
}
```

```
#pragma omp for
for(I=0; I<N; I++)
{
   res[I] = big_calc(I);
   A[I] = B[I] + res[I];
}
```

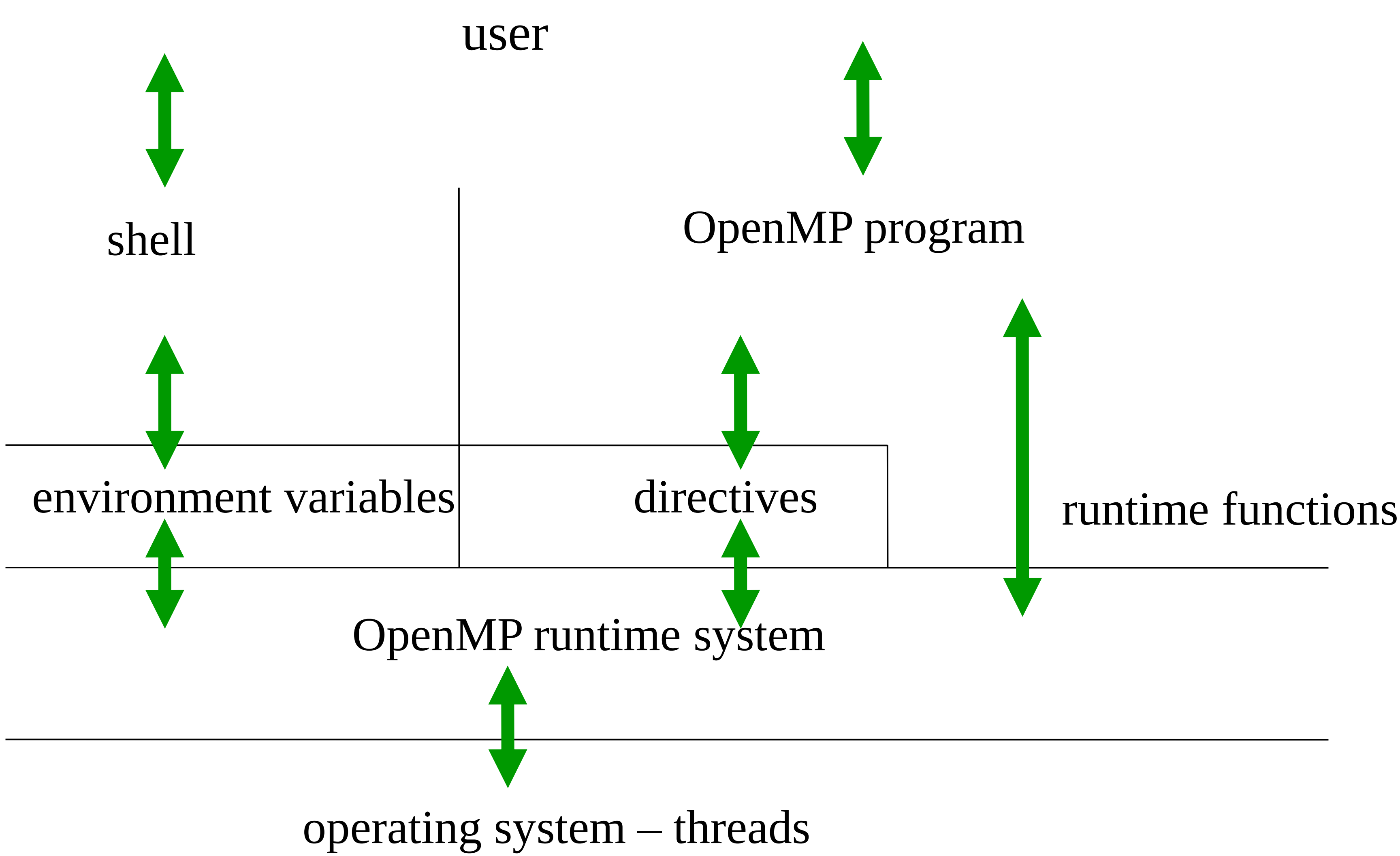
- Conditional Compilation

```
#ifdef _OPENMP
   iam=omp_get_thread_num();
#endif
```

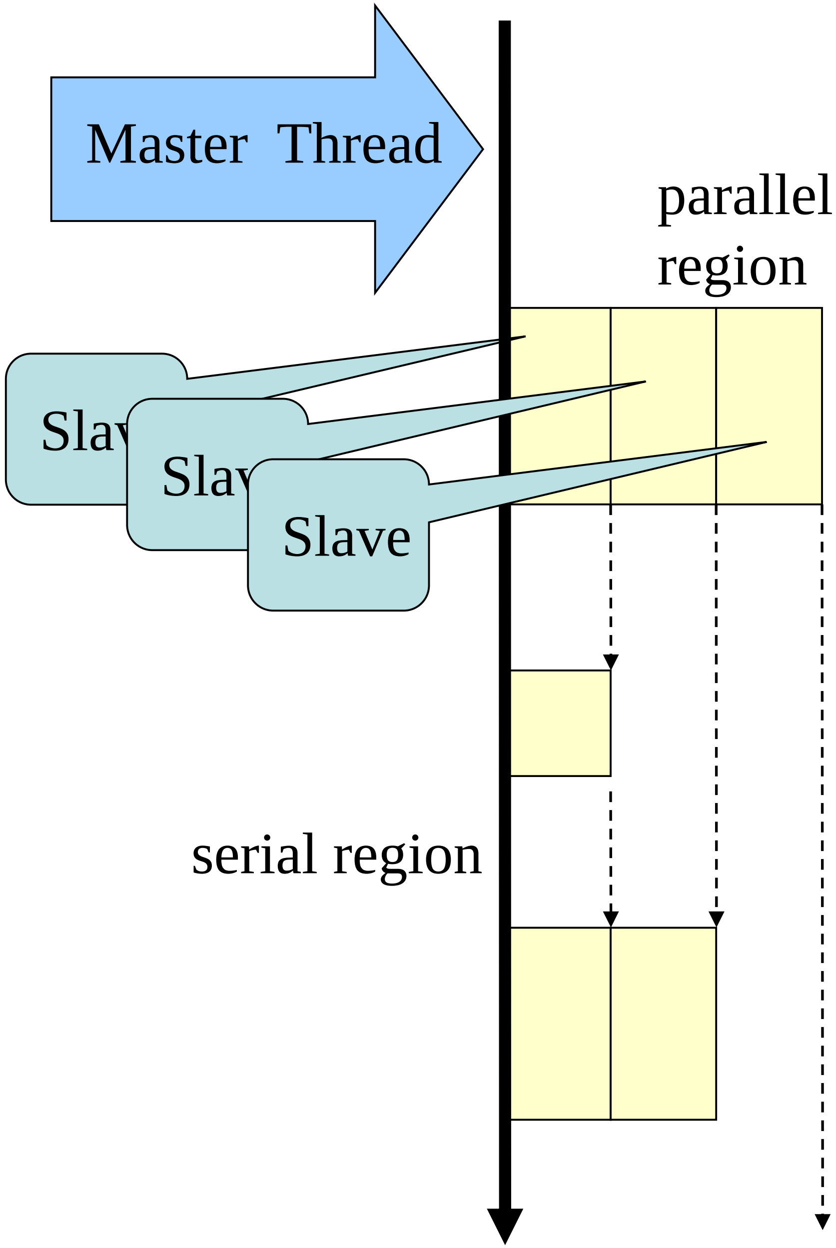
```
/* OpenMP directive */
#pragma omp directive [clause ..]

/* OpenMP directive with
   continuation line */
#pragma omp directive clause \
        clause ...
```

OpenMP Components Diagram



Parallel Regions (1) - The fork-join concept

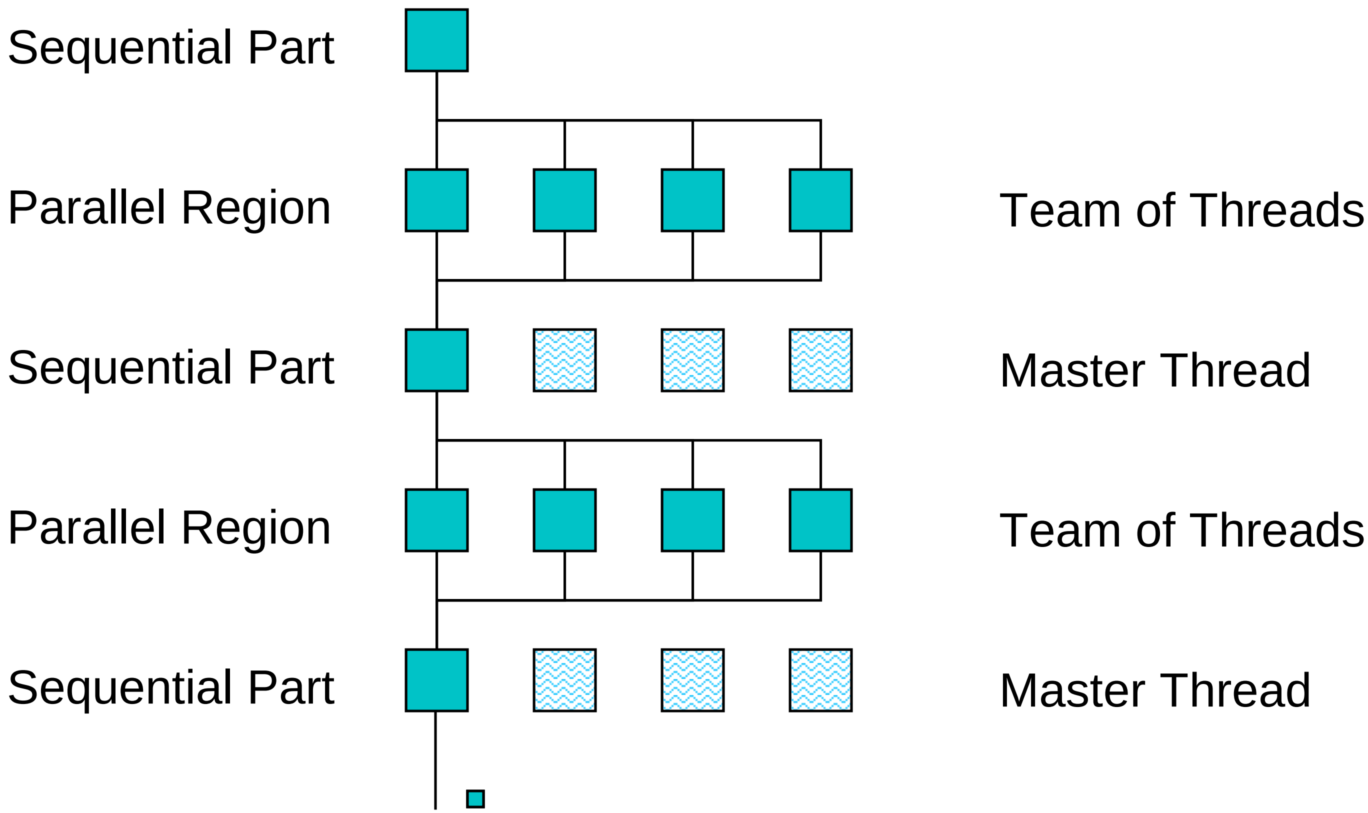


The OpenMP program starts like
a serial program: single threaded

In the beginning of the first parallel
region the slave threads are started.
Together with the master, they form
a team.

Between the parallel regions
the slave threads are put to sleep.

OpenMP Execution Model



OpenMP: Some syntax details to get us started

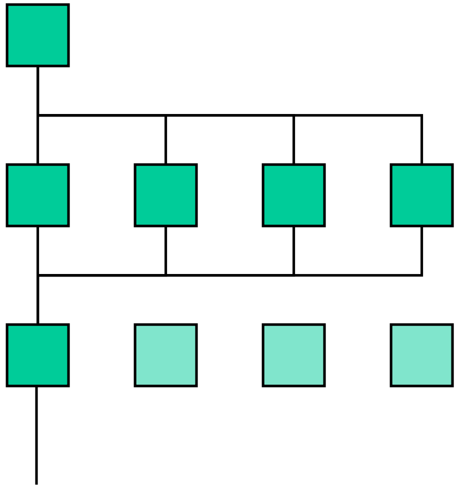
- Most of the constructs in OpenMP are compiler directives or pragmas.
  - For C and C++, the pragmas take the form:

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

- Include OpenMP file
  - **`#include "omp.h"`**

C/C++

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



OpenMP Parallel Region Construct Syntax

- C/C++:

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

- *clause* can be one of the following:
  - private(*list*)
  - shared(*list*)
  - ...
- Conditional compilation

  ```
  #ifdef _OPENMP
     printf("%d avail.processors\n",omp_get_num_procs());
  #endif
  ```

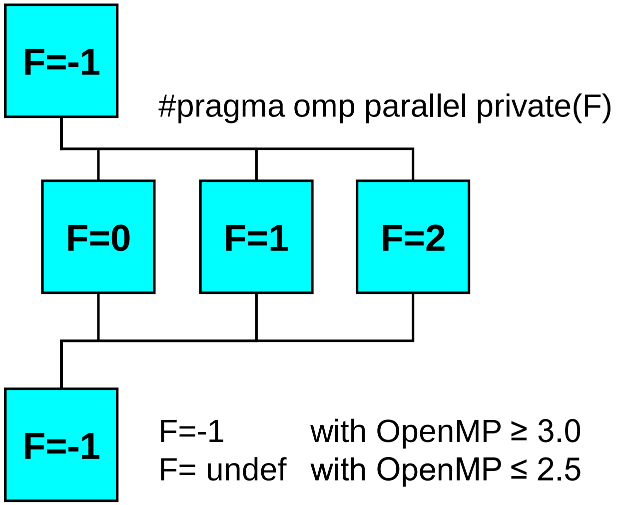
- Include file for library routines:

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

OpenMP Data Scope Clauses

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

    is PRIVATE



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]**

Parallel Regions (2) - Runtime Library

```
#include "omp.h"
void main()
{
 printf("inside parallel region? %d\n", omp_in_parallel());
 printf("number of available processors? %d\n",omp_get_num_procs());
 printf("maximum number of threads? %d\n", omp_get_max_threads());
 omp_set_num_threads (omp_get_max_threads() );
 #pragma omp parallel
 {
   printf("inside parallel region? %d\n", omp_in_parallel());
   printf("number of threads in the team %d\n", omp_get_num_threads());
   printf("my thread id %d\n", omp_get_thread_num() );
 }
}
```

Sequent. region

Parallel region

```
mint [oschenk] export OMP_NUM_THREADS=3
mint [oschenk] ./a.out
inside parallel region? 0
number of available processors? 2
maximum number of threads? 3
```

```
inside parallel region? 1
number of threads in the team 3
my thread id 0
inside parallel region? 1
number of threads in the team 3
my thread id 2
inside parallel region? 1
number of threads in the team 3
my thread id 1
```

Parallel Regions (3) - Runtime Library

	Serial region	Parallel region
void omp_set_num_threads (int)	Set # threads to use in a team	don´t
int omp_get_num_threads (void)	1	Return # threads
int omp_get_max_threads (void)	Return max # threads (OMP_NUM_THREADS)	
int omp_get_thread_num (void)	0	Return thread id 0 … #threads-1
int omp_get_num_procs (void)	Return # CPUs	
void omp_set_dynamic (int)	Control dynamic adjustment of # threads	don´t
int omp_get_dynamic(void)	.TRUE: if dynamic thread adjustment enabled .FALSE. Otherwise	
int omp_in_parallel (void)	.FALSE.	.TRUE.

Parallel Regions (4)

- You create threads in OpenMP with the “omp parallel” pragma.
- For example, to create a 4 thread parallel region:

Each thread executes a copy of the code within the structured block

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
int ID = omp_get_thread_num();
        pooh(ID,A);
}
```

Runtime function to request a certain number of threads

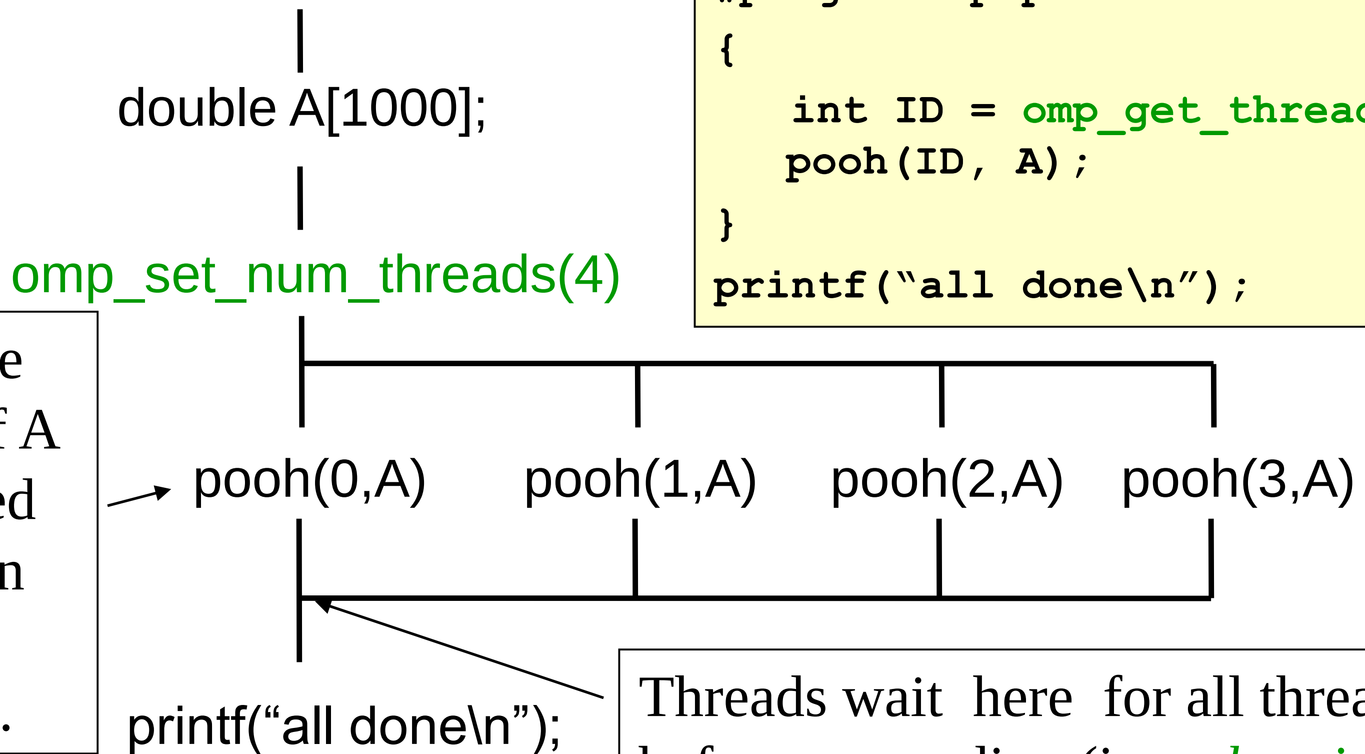
Runtime function returning a thread ID

- Each thread calls pooh(ID,A) for ID = 0 to 3

Parallel Regions (5)

- Each thread executes the same code redundantly.

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
   pooh(ID, A);
}
printf("all done\n");
```



A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e. a *barrier*)

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

- Portable wall clock timers
- **DOUBLE PRECISION FUNCTION OMP_GET_WTIME()**
  provides elapsed time

```
START= OMP_GET_WTIME()
! Work to be measured
END = OMP_GET_WTIME()
printf("Work took %e seconds\n", END-START);
```

- **DOUBLE PRECISION FUNCTION OMP_GET_WTICK()**

  returns the number of seconds between two successive clock ticks

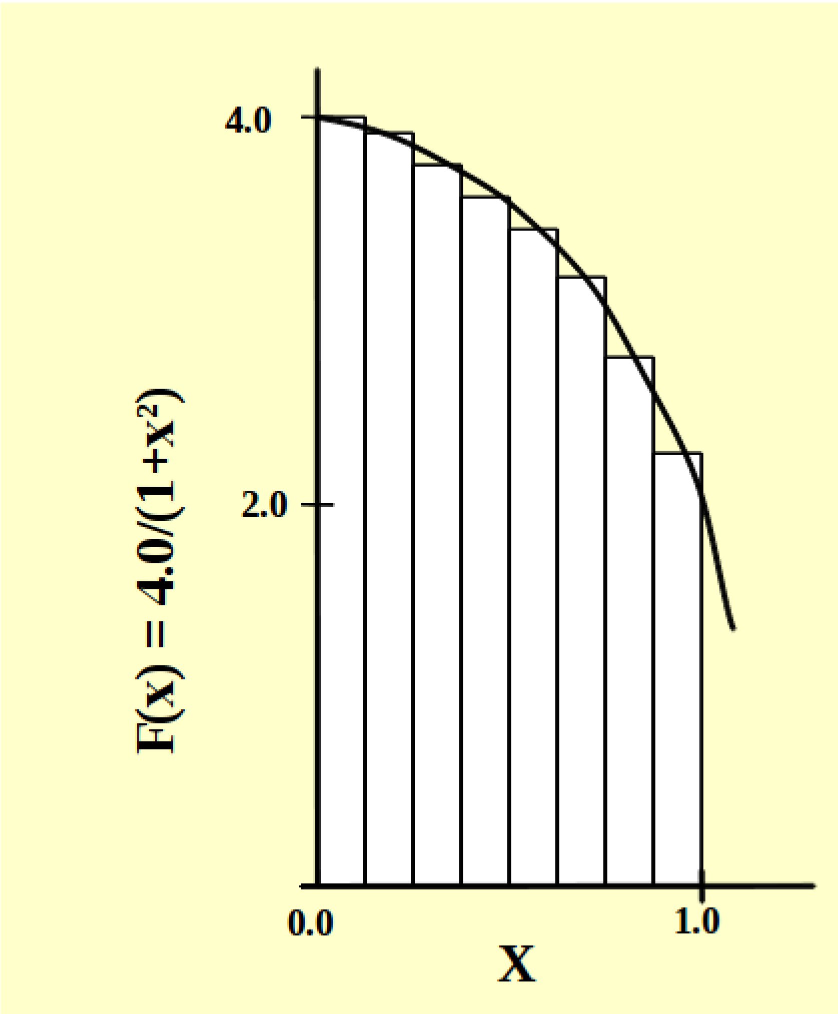
Outline

- 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?
  - Synchronization constructs, e.g., critical regions
  - Nesting and Binding
  - Exercise 2: Pi
- Data environment and combined constructs
  - Private and shared variables, Reduction clause
  - Combined parallel worksharing directives
  - Exercise 3: Pi with reduction clause and combined constructs
  - Exercise 4: Heat
- Summary of OpenMP API
- OpenMP Pitfalls & Optimization Problems

In-class exercise: A multi-threaded "pi" program

- On the following slide, you'll see a sequential program that uses numerical integration to compute an estimate of PI.
- Parallelize this program using OpenMP.
- Remember, you'll need to make sure multiple threads don't overwrite each other's private variables.

In-class exercise: A multi-threaded "pi" program



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\, dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

In-class exercise: Pi program: The sequential program

```
static long num_steps = 100000;
double step;
void main ()
{
int i;
double x, pi, sum = 0.0;

step = 1.0/(double) num_steps;

for (i=0; i<num_steps; i++)
{
       x = (i+0.5)*step;
       sum = sum + 4.0/(1.0+x*x);
}
pi = step * sum;
}
```

In-class exercise: pi0.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

```
  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

timing block C

```
  t2=clock();
# ifdef _OPENMP
    wt2=omp_get_wtime();
# endif
  gettimeofday(&tv2, &tz);
  printf( "computed pi = %24.16g\n", pi);
  printf( "CPU time (clock)
   = %12.4g sec\n", (t2-t1)/1000000.0 );
# ifdef _OPENMP
    printf( "wall clock time
     (omp_get_wtime) = %12.4g sec\n",
      wt2-wt1 );
# endif
  printf( "wall clock time (gettimeofday)
   = %12.4g sec\n",
   (tv2.tv_sec-tv1.tv_sec) +
   (tv2.tv_usec-tv1.tv_usec)*1e-6 );
  return 0;
}
```

In-class exercise: Parallel region (1)

- Goal: usage of
  - runtime library calls
  - conditional compilation, environment variables
  - parallel regions, private and shared clauses
- Serial programs: **pi0.c** on https://www2.icorsi.ch/
- Build a team of 2 students, compile **serial** program **pi0.c** on CUB cluster and run e.g. with
  - **scp pi0.c student@cub.inf.usi.ch:.**
  - **ssh student@cub.inf.usi.ch**
  - **salloc -pdebug -t 00:30:00** (to request one node in interactive mode for 30 min)
- Compile **as OpenMP** program and run on 4 core
  - **gcc –O3 –fopenmp pi0.c –o pi**
  - **export OMP_NUM_THREADS=4**
  - **./pi**

expected result: program is not parallelized,
therefore same pi-value and timing,
additional output from omp_get_wtime()

In-class exercise: pi1.c

- Modify pi0.c -> pi1.c
- Directly after the declaration part, add in a **parallel region that prints on each thread**
  - **its rank** (with **omp_get_thread_num()**) and
  - **the number of threads** (with **omp_get_num_threads()**)
- compile **gcc –O3 –fopenmp pi1.c –o pi1** and run on 4 cores
- Expected results: numerical calculation is still not parallelized, therefore still same pi-value and timing, additionally output:

```
bash$ gcc -O3 -fopenmp -o pi1 pi1.c
bash$ export OMP_NUM_THREADS=4;  ./pi1
I am thread  0 of  4 threads
I am thread  2 of  4 threads
I am thread  3 of  4 threads
I am thread  1 of  4 threads
computed pi =        3.141592653589731
CPU time (clock)                =        0.16 sec
wall clock time (omp_get_wtime) =      0.1681 sec
wall clock time (gettimeofday)  =      0.1681 sec
```

undefined sequence!

OpenMP Advanced Exercise: pi1.c

- Modify pi1.c
- Use a private variable for the rank of the threads
- Check, whether you can get a race-condition if you forget the private clause on the **omp parallel** directive, e.g.

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

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

OpenMP Advanced Exercise: pi1.c

- Modify pi1.c
- Guarantee with conditional compilation, that source code still works **with non-OpenMP compilers** (i.e., without OpenMP compile-option).
- Add an "**else clause**", printing a text if OpenMP is not used.
- Expected output:

  If compiled with OpenMP, see previous slide.

  If compiled without OpenMP

```
bash$ gcc -o pi1 pi1.c
bash$ export OMP_NUM_THREADS=4;  ./pi1
This program is not compiled with OpenMP
computed pi =        3.141592653589731
CPU time (clock)                =        0.16 sec
wall clock time (gettimeofday)  =      0.1681 sec
```

Outline

- 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?**
  - **Synchronization constructs, e.g., critical regions**
  - **Nesting and Binding**
  - **Exercise 2: Pi**
- Data environment and combined constructs
  - Private and shared variables, Reduction clause
  - Combined parallel worksharing directives
  - Exercise 3: Pi with reduction clause and combined constructs
  - Exercise 4: Heat
- Summary of OpenMP API
- OpenMP Pitfalls & Optimization Problems

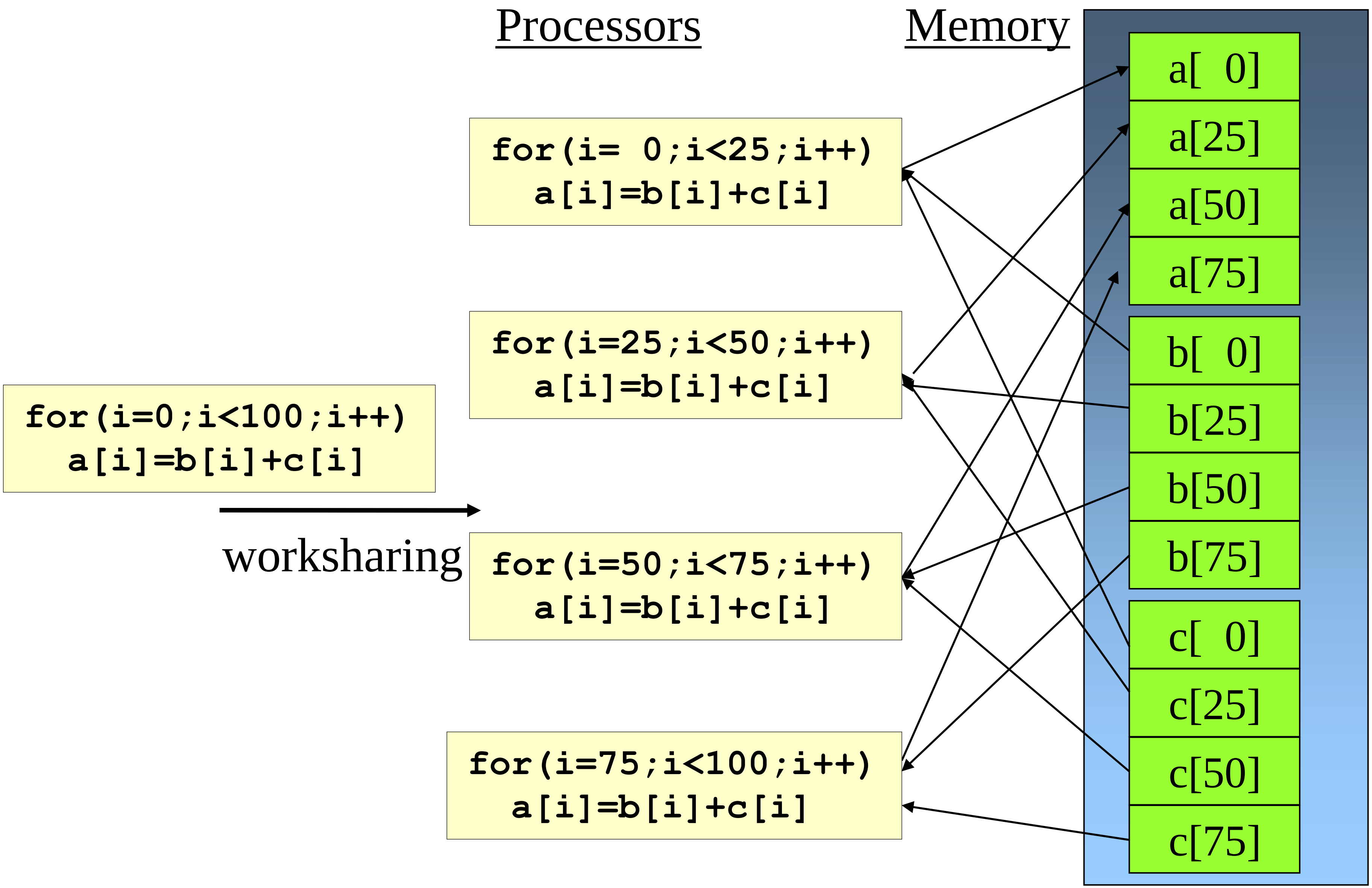
Worksharing and Synchronization

- Which thread executes which statement or operation?
- and when?
  - Worksharing constructs
  - Master and synchronization constructs
- **i.e., organization of the parallel work!!!**

Worksharing and Synchronization

- 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
    - `for` directive
    - `sections` directive
    - `task` directive
    - `single` directive

Work Sharing (1) - Principle



Work Sharing (1) - A motivating example

Sequential code

```
for(i=0;I<N;i++)   { a[i] = a[i] + b[i];}
```

OpenMP
parallel region

```
#pragma omp parallel
{
   int id, i, Nthrds, istart, iend;
   id = omp_get_thread_num();
   Nthrds = omp_get_num_threads();
   istart = id * N / Nthrds;
   iend = (id+1) * N / Nthrds;
   for(i=istart;I<iend;i++)
   { a[i] = a[i] + b[i];}
}
```

OpenMP
parallel region and
a work-sharing
for-construct

```
#pragma omp parallel
#pragma omp for schedule(static)
for(i=0;I<N;i++)   { a[i] = a[i] + b[i];}
```

Work Sharing (2) - Sharing Constructs

- The "for" work-sharing construct splits up loop iterations among the threads in a team

```
#pragma omp parallel

for (I=0;I<N;I++)
{
   NEAD_STUFF(I);
}
```

All threads select all loop indices "I "

```
#pragma omp parallel
#pragma omp for
for (I=0;I<N;I++)
{
   NEAD_STUFF(I);
}
```

The loop indices "I " are distributed among threads

By default, there is a **barrier** at the end of the "omp for". Use the **nowait** clause to turn off the barrier.

Work Sharing (7) - Combined parallel/work-share

- OpenMP shortcut:
  - Put the "parallel" and the work-share on the same line

```
double  res[MAX];
int i;
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++)
    {
        res[i] = huge();
    }
}
```

```
double  res[MAX];
int i;
#pragma omp parallel for
for (i=0;i< MAX; i++)
{
    res[i] = huge();
}
```

These are equivalent

OpenMP **`for`** Directives – Syntax

- Immediately following loop executed in parallel

```
#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-- ...
```

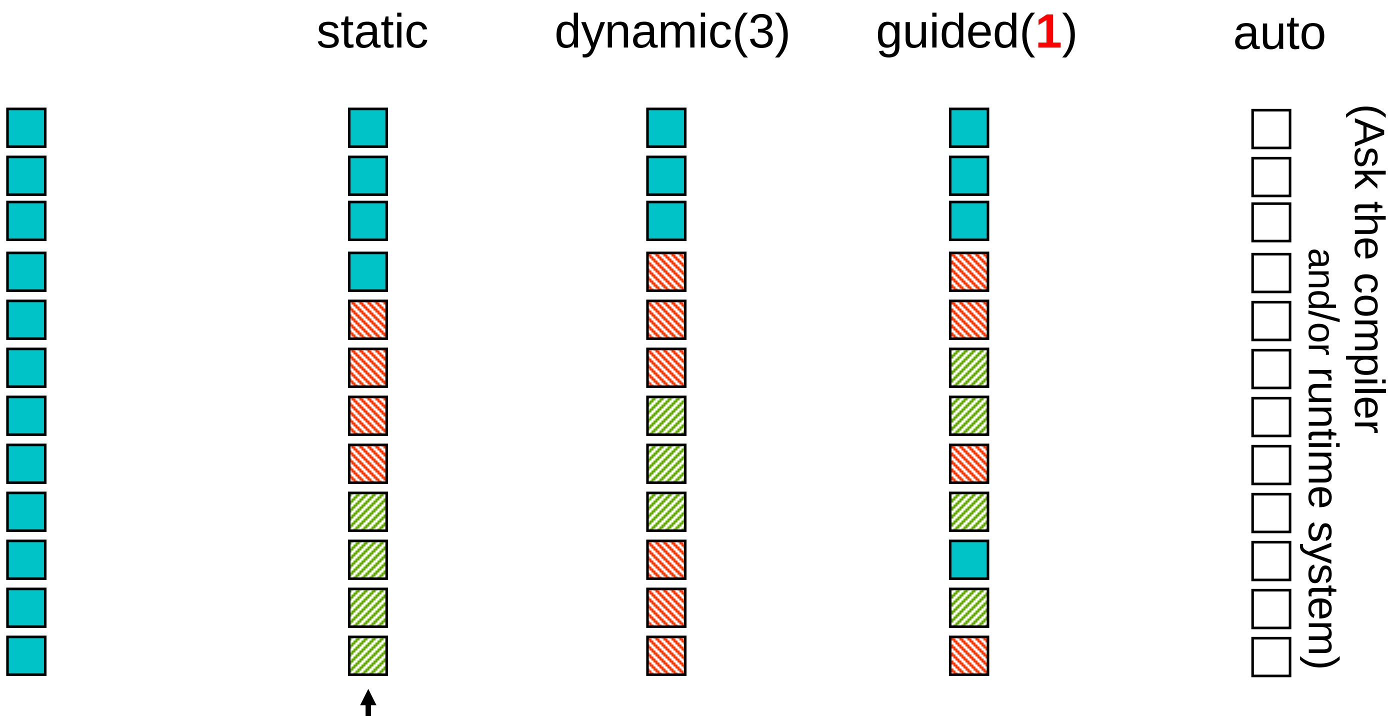
`var`: must not be modified in the loop body;
integer (signed or unsigned),
or pointer type (C only), (OpenMP ≥ 3.0)
or random access iterator type (C++ only)

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

Work Sharing (3) - The schedule clause

- The schedule clause effects how loop iterations are mapped onto threads
  - **schedule(static [,chunk])**
    - Deal-out blocks of iterations of size “chunk” to each thread.
    - In our example: thread #0: i=0 to24; thread #1: i=25 to 49; …
  - **schedule(dynamic[,chunk])**
    - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.
    - In our example: thread #0: i=0, 3, 8, ..; thread #1: i=1, 2, 5, ..;
  - **schedule(guided[,chunk])**
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.
  - **schedule(auto)**
    - Scheduling is delegated to the compiler and/or runtime system **(OpenMP ≥ 3.0)**
  - **schedule(runtime)**
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable.

Loop scheduling

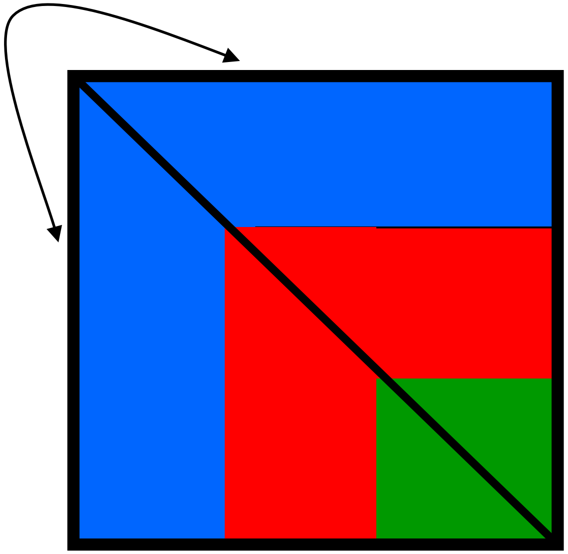


- Method is implementation dependent, e.g.,
  13 iterations on 3 threads = 5+5+3 or = 5+4+4
- Two loops in same parallel region and with same count are divided in same way, i.e., static schedule is **deterministic**

OpenMP ≥ 3.0

Work Sharing (4) – Matrix Transposition

```
#pragma omp parallel for private(h,i,j)
schedule static numthreads(3)
for (i=0; i<n; i++) {
  for (j=i+1; j<n; j++) {
    h = A[i][j];
    A[i][j] = A[j][i];
    A[j][i] = h;
  }
} // end parallel for
```



- Thread 0 would have much more work than thread 2!

Work Sharing (5) - The schedule clause

Schedule Clause	When To Use
STATIC	Predictable and similar work per iteration thread #0: i=0 to24; thread #1: i=25 to 49 thread #2: i=50 to74; thread #0: i=75 to 99
DYNAMIC	Unpredictable, highly variable work per iteration thread #0: i=0,2, 8,.., thread #1: 4,6,7,… thread #2: i=2,3,9,..; thread #0: i=15,18,91,..
GUIDED	Special case of dynamic to reduce scheduling overhead

Work Sharing (6) - The section clause

- The **section work-sharing** construct gives a different structured block to each thread.

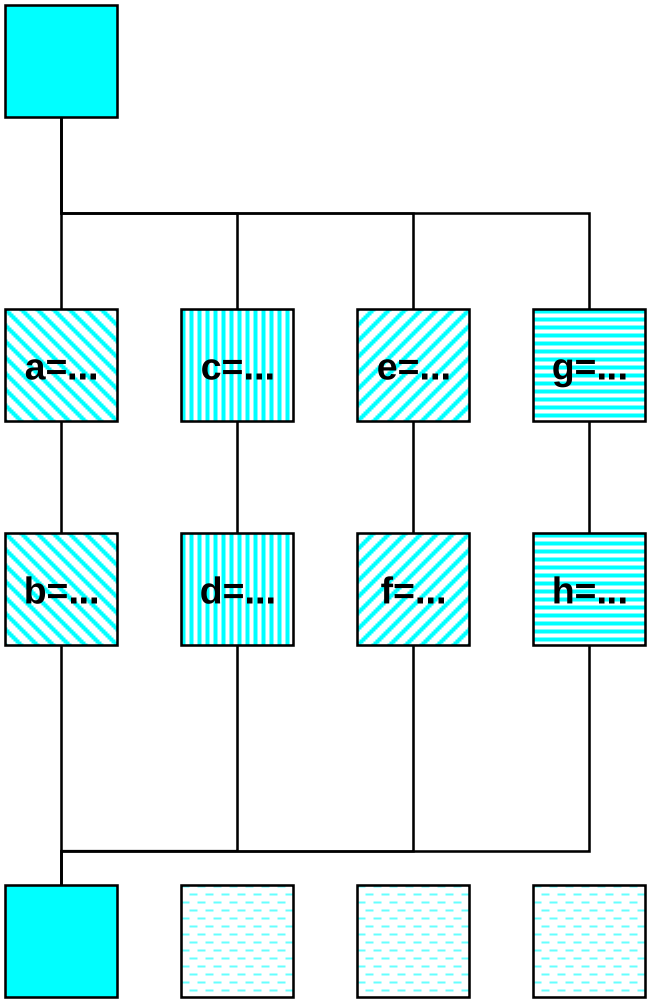
```
#pragma omp parallel
#pragma omp sections
{
  #pragma omp section
  X_calculation(); // only one thread
  #pragma omp section
  y_calculation(); // only one thread
  #pragma omp section
  z_calculation(); // only one thread
}
```

By default, there is a **barrier** at the end of the "omp sections". Use the "**nowait**" clause to turn off the barrier.

Work Sharing (6) - The `section` clause

C / C++:

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



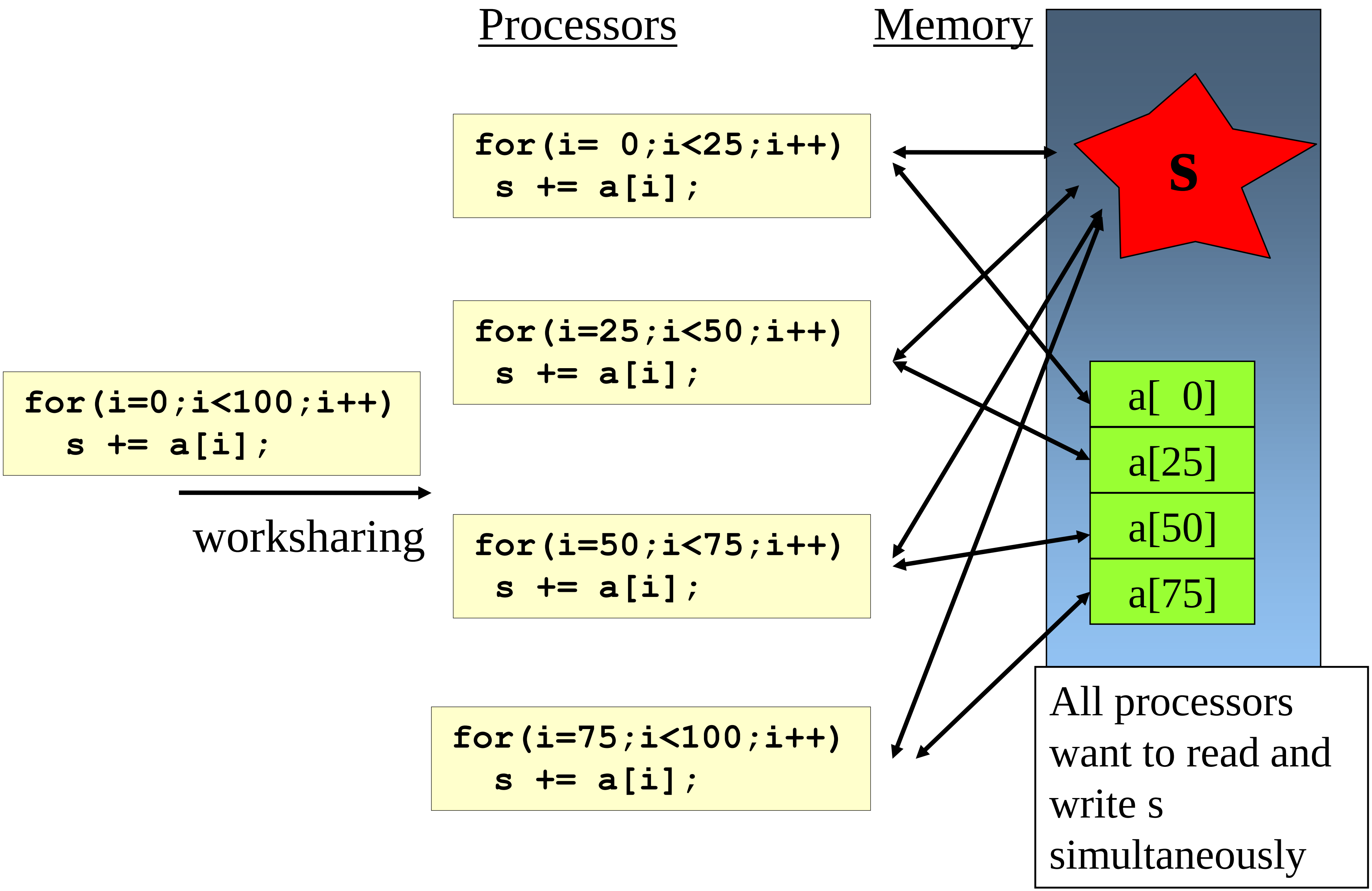
OpenMP **task** Directive – Parallelized traversing of a tree

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

Critical sections (1)



Critical sections (2)

```
for(i=0;i<100;i++)
  s += a[i];
```

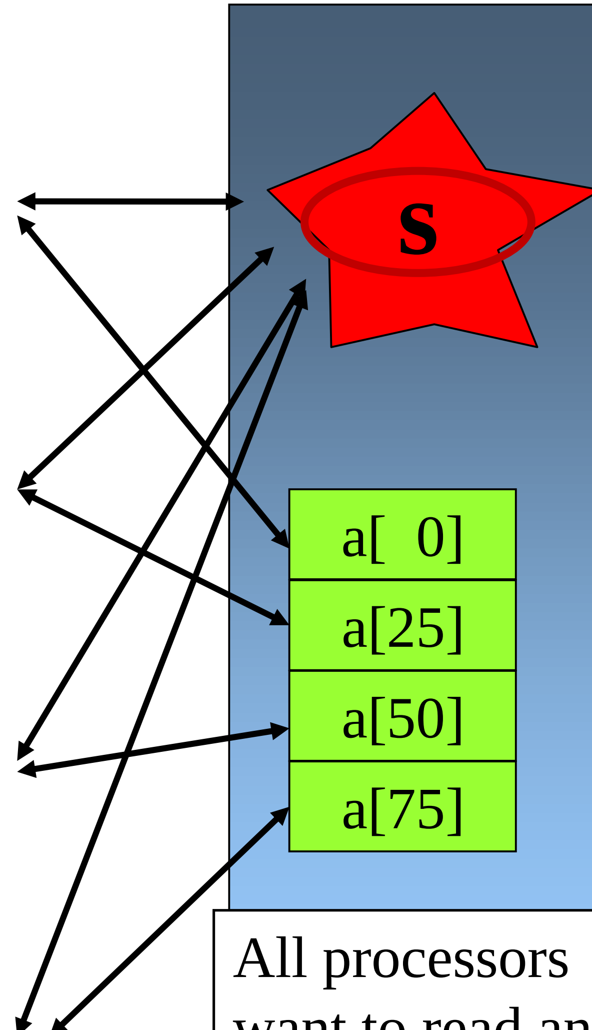
worksharing

```
for(i= 0;i<25;i++)
 s += a[i];
```

```
for(i=25;i<50;i++)
 s += a[i];
```

```
for(i=50;i<75;i++)
 s += a[i];
```

```
for(i=75;i<100;i++)
 s += a[i];
```



Only one processor is allowed to enter the critical region at a time → **BAD** performance

All processors want to read and write s simultaneously

Critical sections (3)

- Only one thread at a time can enter a critical section.

Threads wait their turn – only one at a time calls consum()

```
float res;
#pragma omp parallel
{
   float B;
   int i;
   #pragma omp for
   for(i=0;i<niters;i++)
   {
      B =  big_job(i);
      #pragma omp critical
      consum (B, RES);
   }
}
```

Critical sections (4) – Critical / end critical

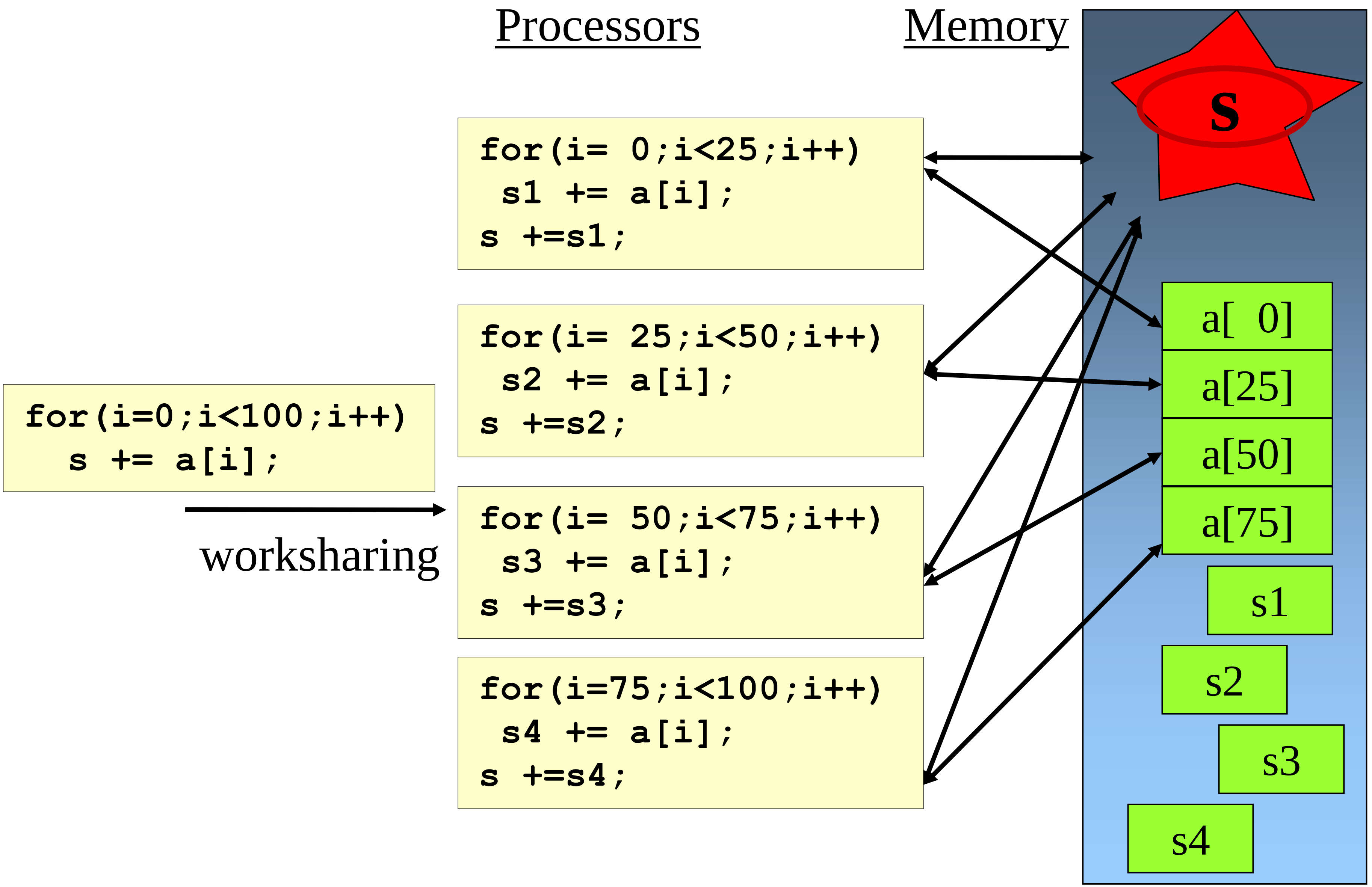
```
for(i=0;i<100;i++)
   s= s + a[i];
```

```
#pragma omp parallel for private (i)
for (i=0; i<100; i++) {
   #pragma omp critical
   { s += a[i]; }
}
```

Only one processor is allowed to enter the **critical section** at a time

As the loop body consists of a critical section only, the parallel program will run much slower

Critical section (5)



Critical section (6) – Critical / end critical

```
for(i=0;i<100;i++)
   s= s + a[i];
```

```
#pragma omp parallel for \
        private (i)
for (i=0; i<100; i++) {
   #pragma omp critical
   { s += a[i]; }
}
```

```
#pragma omp parallel \
       private (i,s_local)
{
   s_local = 0;
   #pragma omp for
     for (i=0; i<100; i++)
        { s_local += a[i]; }
   #pragma omp critical
       { s += s_local; }
}
```

Only one processor is allowed to enter the **critical section** at a time

As the loop body consists of a critical region only, the parallel program will run much slower

Now the partial sums are calculated in parallel. The critical region is entered only once per thread.

Outline

- 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?
  - Synchronization constructs, e.g., critical section
  - **Exercise 2: Pi**
- Data environment and combined constructs
  - Nesting and Binding
  - 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 & Optimization Problems

In-class exercise 2: pi Program (1)

- Goal: usage of
  - worksharing constructs: **`#pragma omp for`**
  - **`#pragma omp critical`** directive
  - Use your result **`pi1.c`** from the last in-class exercise
  - Modify **`pi1.c`** -> **`pi2.c`**
  - compile serial program **`pi2.c`** and run
- add **`parallel region`** and **`pragma omp for`** directive 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?)

In-class exercise 2: pi Program (2)

- add `private(x)` clause in `pi2.c` and compile
- set environment variable `OMP_NUM_THREADS` to `2` and run

  value of pi? (should be still incorrect ...)
- run again

  value of pi?
- set environment variable `OMP_NUM_THREADS` to `4` and run

  value of pi?
- run again

  value of pi? (... and where is the second race-condition?)

In-class exercise 2: pi Program (3)

- add **`critical`** directive in **`pi2.c`** 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 CPU time!)
- run again

  value of pi? (but not reproducible in the last bit!)
- set environment variable **`OMP_NUM_THREADS`** to **`4`** and run

  value of pi? execution time? (Oh, does it take longer?)
- run again

  value of pi? execution time?

  How can you optimize your code?

In-class advanced exercise 2: pi Program (4)

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

```
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
```