Karlsruhe Institute of Technology

Parallel Programming with MPI and OpenMP

OpenMP

Hartmut Häfner, Steinbuch Centre for Computing (SCC)

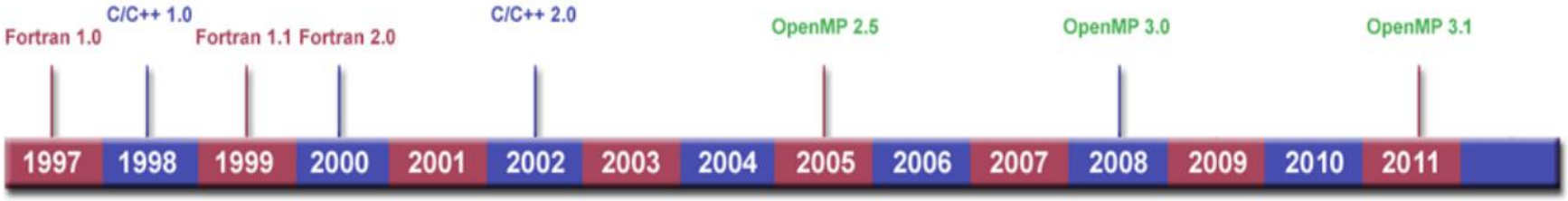
STEINBUCH CENTRE FOR COMPUTING - SCC

Informations on OpenMP

- OpenMP Homepage:
  http://www.openmp.org/
- OpenMP User Group:
  http://www.compunity.org
- OpenMP Tutorial:
  https://computing.llnl.gov/tutorials/openMP/

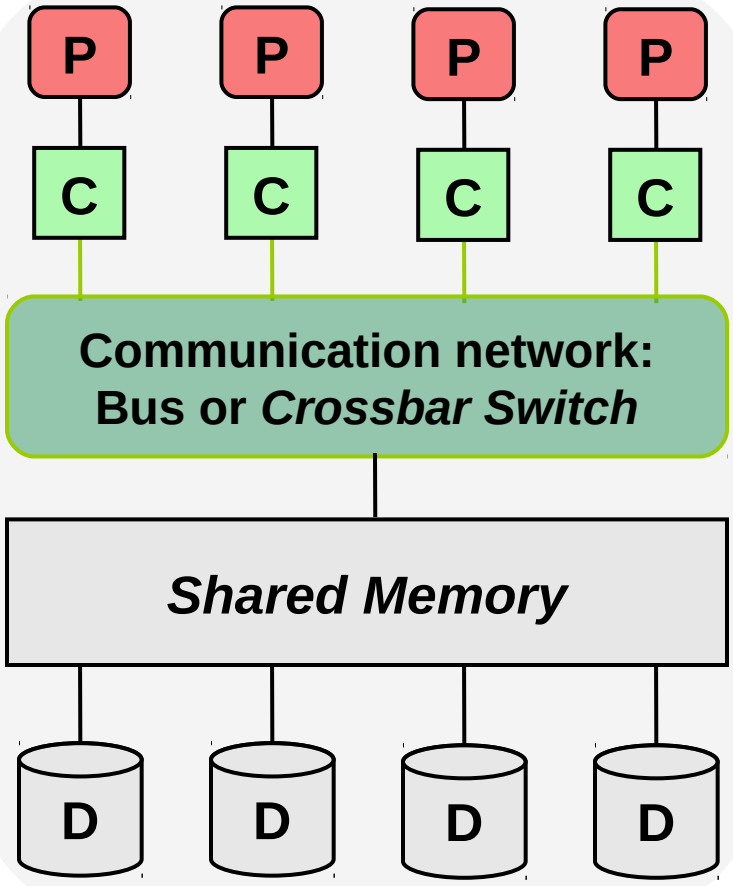
- R.Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon:
  Parallel programming in OpenMP.
  Academic Press, San Diego, USA, 2000, ISBN 1-55860-671-8
- R. Eigenmann, Michael J. Voss (Eds):
  OpenMP Shared Memory Parallel Programming.
  Springer LNCS 2104, Berlin, 2001, ISBN 3-540-42346-X
- Simon Hoffmann, Rainer Lienhart: OpenMP.
  www.eBook.de, 2009, ISBN 3-540-73122-9

Timeline on OpenMP



- OpenMP 3.0 incorporates Task-concept
- OpenMP 4.0 has been released on 7/23/2013 and contains heterogenity, i.e. support for accelerator cards
- OpenMP 4.5 has been released in Novembre 2015, supports Accelerators/graphics boards and has implemented different Locking-Mechanisms

Shared Memory System



- The whole system behaves like a single computer.
- All processors (cores) access the *Shared Memory*.
- ONE copy of the operating system.
- Parallelization via *Shared Memory* or *Message Passing*
  - OpenMP
  - MPI
    Message Passing via *Shared Memory*
- Examples: SMP *workstations*, NUMA nodes of HPC-systems in Karlsruhe

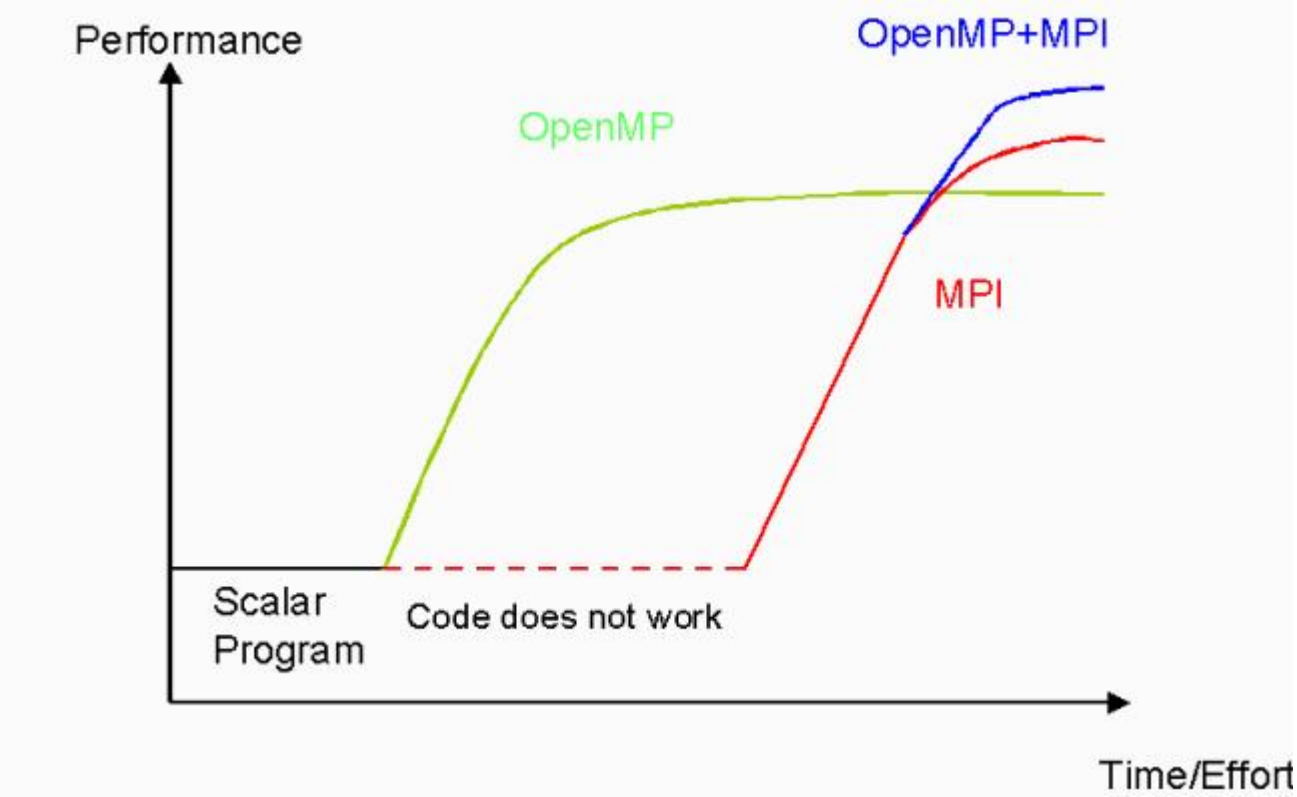
Important Features of OpenMP

- Data are stored in *Shared Memory*
- ONE process when starting the application
- Parallel *threads* will be created and destroyed during the execution of the program
- *Threads* can access shared or privat data
- Coarse-grain or fine-grain parallelization

Parallelization – Basic Idea

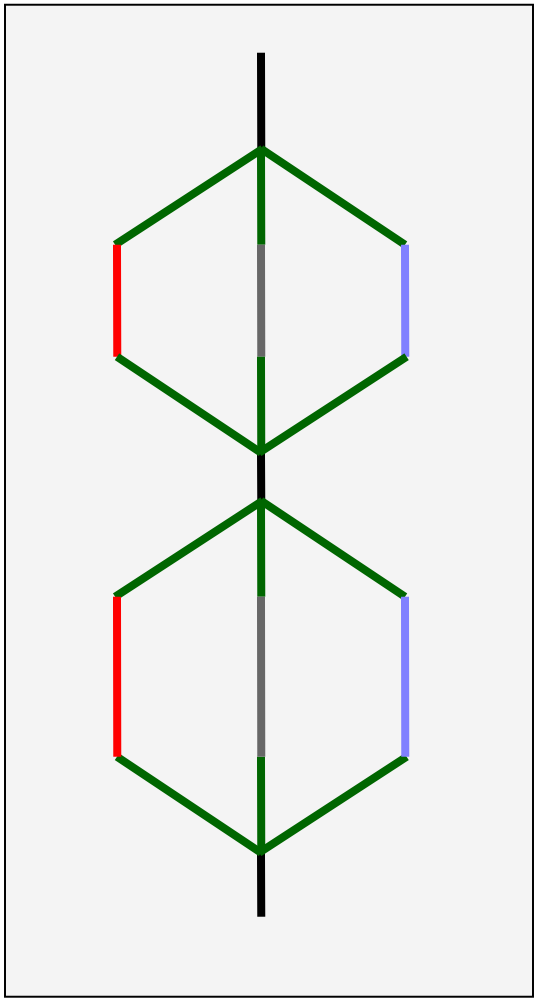
- *Parallel Region*
  - Code within a *Parallel Region* is executed by all created threads
- *Work sharing* Constructs
  - for DO-loops (`!$OMP DO` or `#pragma omp for`)
    Single loop-indices are executed by different *threads*
  - for code segments (`!$OMP SECTIONS` or `#pragma omp sections`)
    By `!OMP SECTION` or `#pragma omp section` seperated code segments are executed by different *threads*
- *Work Sharing* constructs must be implemented within *Parallel Regions!*

When to use OpenMP?



Fine-grain Parallelization

```
program main
...
!$OMP PARALLEL DO
do i=1,n
      !work
end do
!$OMP END PARALLEL DO
...
!$OMP PARALLEL SECTIONS
!$OMP SECTION
...
!$OMP SECTION
...
!$OMP END PARALLEL SECTION
....
end
```



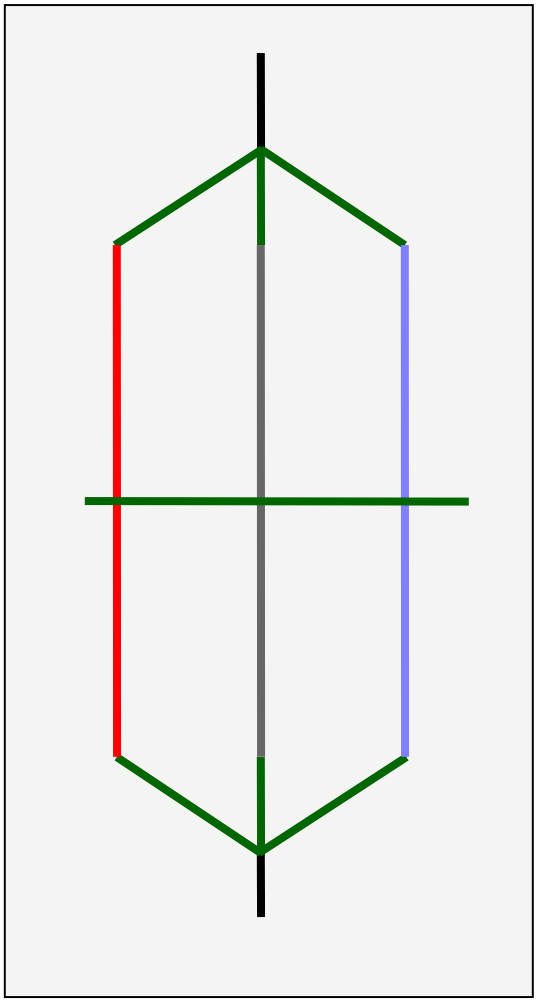
Coarse-grain Parallelization

```
program main

!$OMP PARALLEL
size = OMP_GET_NUM_THREADS()
iam  = OMP_GET_THREAD_NUM()

call work(a, b, n, size, iam)
!$OMP BARRIER
do i= 1, n/size
   ...
end do
...

!$OMP END PARALLEL
end
```



Overview on OpenMP

- Directives
  - Fortran: `!$OMP …` C: `#pragma omp …`
  - *Parallel regions* construct
    - `Fortran: !$OMP PARALLEL … !$OMP END PARALLEL`
    - `C: #pragma omp parallel{ … }`
  - *Work sharing* construct
    - `Fortran: !$OMP DO … !$OMP END DO`
    - `C: #pragma for …`
  - Synchronization construct
    - `!$OMP BARRIER`
  - *Variable scoping* clauses and directives
    - `Fortran: !$OMP THREADPRIVATE (/CBLCK/[,/CBLCK/]…)`
    - `C: #pragma omp threadprivate (list of global vars)`
- Runtime library (a few routines)
- Environment variables

OpenMP – typical usage

- **Normal case - OpenMP is used to parallelize loops:**
  - **Find the most time consuming loops with e.g. `prof`, `gprof`, `xprofiler`, `Intel Advisor XE 2017`.**
  - **Separate them onto many *threads*.**

Separate this loop onto many *threads*.

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

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

Sequential program

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

Parallel program

OpenMP Directives

Directives: `!$OMP` ***`directive clauses`***

`#pragma omp` ***`directive clauses`***

- `parallel`
  - `private:` List with private variables
  - `shared:` List with global variables
  - `firstprivate:` `Value of` private Variable w is copied from *master thread* to all *threads*
  - `lastprivate:` `Value of` sequential last variable is copied to *master thread*
  - `reduction(op:variable)`: Operator: `+,-,*,/,&&,||,==`
  - `schedule`
- `do or for:` Worksharing-construct loop
- `sections:` Worksharing-construct (Code-)sections
- `critical:` At each arbitrary time only ONE *thread* executes the assignment
- `master:` `Only the` *master thread* executes the enclosed codesegment
- `single`: Only the chronologically first *thread* executes the enclosed code segment; all other *threads* wait till the first thread has ended its execution
- `barrier:` All *threads* wait at the barrier
- `atomic:` Atomic operations are handled as critical section

Important Directive: OMP PARALLEL

- **OMP PARALLEL - Directive starts *Parallel Region***
- **Within [*clauses*] can be specified, which variables are privat and which are shared for the *threads*.**

- **Fortran:**

```
!$OMP PARALLEL [clauses]
 block
!$OMP END PARALLEL
```

- **C:**

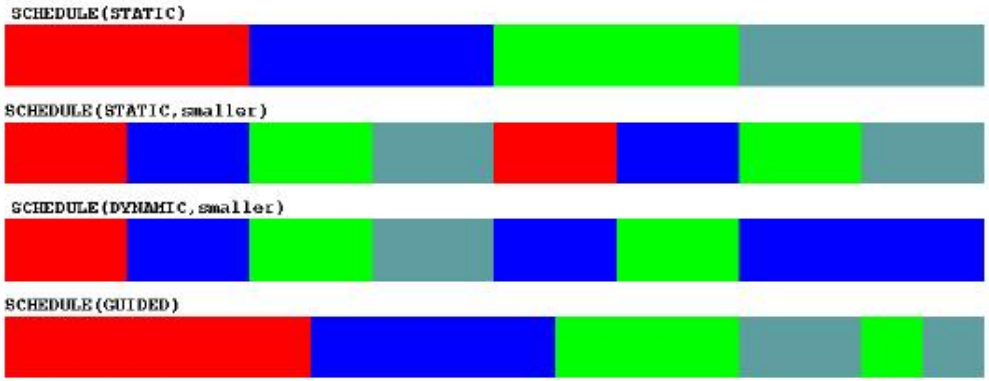
```
#pragma omp parallel [clauses]
 {block}
```

- **[*clauses*] can be**
  - **private(list)**
  - **shared(list)**
  - **schedule(type[,chunk])**

The Clause `schedule`

The clause `schedule` can be:

- `static`: Iterations are divided into pieces of size `chunk`. The pieces are assigned statically to the *threads of a team*
- `dynamic`: Iterations are divided into pieces of size `chunk`. When a *thread* has ended its work, it gets dynamically the next piece
- `guided`: Each *thread* works at a piece of size `chunk`. When a *thread* has ended its work, it gets a new piece of work of decreasing size
- `runtime:` Schedule is determined by environment variable "OMP_SCHEDULE"



Important Directive: OMP DO

- Fortran:
  ```
  !$OMP DO [clauses]
  Fortran DO Konstrukt
  [!$OMP END DO [NOWAIT] ]
  ```
- C:
  ```
  #pragma omp for [clauses]
  for Schleife
  ```

- The loop following on the directive is divided on all *threads of the* parallel *team*
- Loops must have the syntax
  `do i = i1, i2 [ ,i3 ]` or
  `for`-loop must be canonically
- Number of loops must be known at start time of the loop

Directive: OMP SECTIONS

- **Fortran:**

```
!$OMP PARALLEL [clauses]
!$OMP SECTIONS
  a= …
!$OMP SECTION
  b= …
!$OMP SECTION
  c= …
!$OMP END SECTIONS [NOWAIT]
!$OMP END PARALLEL
```

C:

```
#pragma omp parallel
{
#pragma omp sections
 {{ a= … ;}
#pragma omp section
  { b= … ;}
#pragma omp section
  { c= … ;}
 } /* end sections */
} /* end parallel */
```

- **The existing *sections* are separated between the parallel *teams*. Each *section* will be executed once by a *thread***

OpenMP 3.0 Directive: `OMP TASK`

- Directive `TASK` creates a Task (working package: code + data)
  - Thread executing `TASK`-construct creates working package
  - Execution of the working package can be delayed
  - Working package can be executed by an arbitrary t*hread of the team*
- Similar to construct `OMP SECTIONS`
- Avoids to many nested *Parallel Regions*
- Allows to parallelize irregular problems (e.g. recursive algorithms)

OpenMP 4.5 Directive: OMP TASKLOOP

- **Directive `TASKLOOP` uses OpenMP-tasks for execution and allows e.g. the concurrent usage of tasks and an ordinary loop**

```
#pragma omp taskgroup
{
#pragma omp task
   long_running_task()    // kann nebenher ablaufen

#pragma omp taskloop collapse(2) grainsize(500) nogroup
   for (int i = 0; i < N; i++)
      for (int j = 0; j < M; j++)
         loop_body();
}
```

Scope of Variables

- *Shared Memory* programming model:
  - The most variables are shared by default.
- Global variables are shared among *threads*.
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: *File scope* variables, static
- But not all ist shared...
  - Stack variables in subroutines called within *parallel regions* are private.
  - Automatic variables within a block of assignments are private.
- Some loop indices are private by default:
  - Fortran: Loop indices are private, even when they are specified as *shared*.

Environment Variables

- `OMP_NUM_THREADS`
  - Sets the number of *threads* used during execution
  - If the number of threads changes dynamically during execution, the value of the environment variable is the maximal value of running threads
  - sh, ksh, bash: `export OMP_NUM_THREADS=16`

- `OMP_SCHEDULE`
  - Is only interpreted at do/for or parallel do/parallel for directives when the clause `schedule(runtime)` is used
  - Sets `type` and chunk for all loops with above mentioned clause
  - sh, ksh, bash: `export OMP_SCHEDULE="STATIC,4"`

Shared memory Programming Faults

- ***Race conditions***
  - ***Data-race:*** **Minimally 2 *threads* access the same *shared variable* and at least 1 *thread* modifies the variable and the accesses take place concurrently and not synchronized (often when using *shared data* accidentally)**

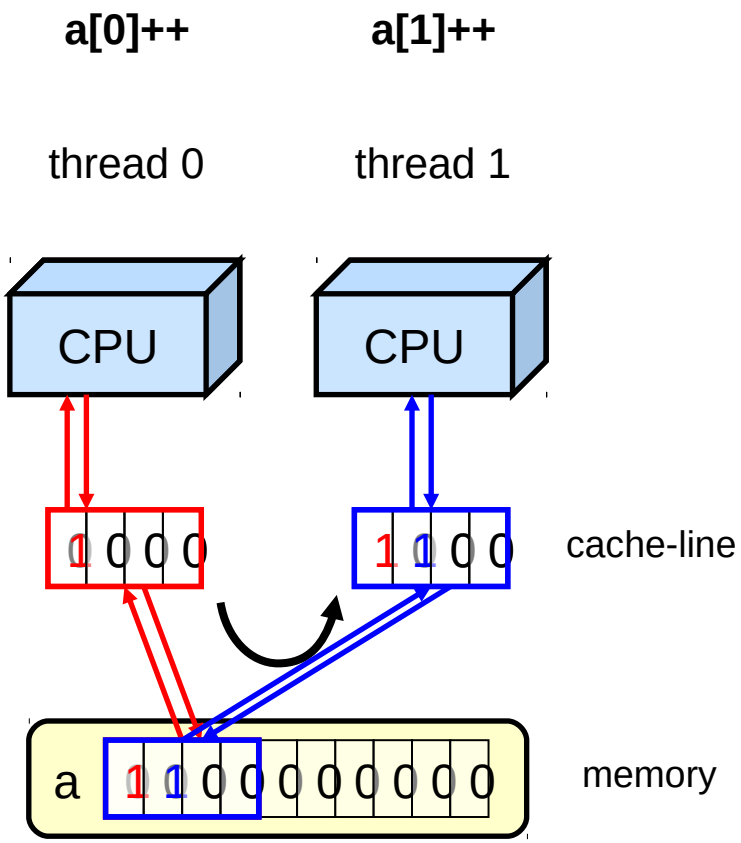
- ***Deadlock***
  - **Threads wait on a *locked* resource never reaching the state *unlocked* (avoid nesting of different *locks)***

Example for a *race condition*

```
!$OMP PARALLEL SECTIONS
       a = b + c
!$OMP SECTION
       b = a + c
!$OMP SECTION
       c = b + a
!$OMP END PARALLEL SECTIONS
```

- The results are varying unpredictable
- No warning from your program

False-sharing



- Many *threads* write! data into the same *cacheline*
- The *cacheline* must be swapped between the caches of the CPUs dedicated to the accessing *threads* → consumes much time

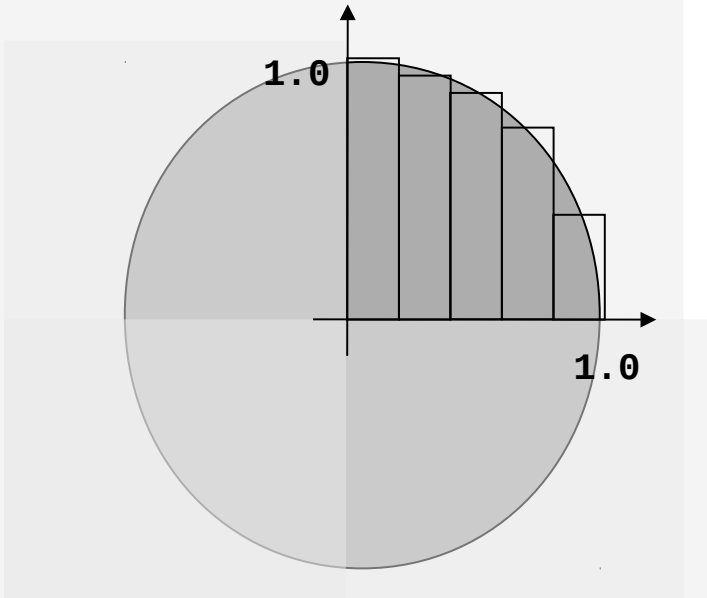
Computation of PI

```
program compute_pi
integer                         :: i
integer, parameter              :: n=50000000, dp = kind(1.d0)
real(kind=dp)                   :: w,x,sum,pi,d

w=1.0/n; sum=0.0

do i=1,n
   x   = (i-0.5) * w
   d   = w * SQRT(1.0 - x**2)
   sum = sum  + d
enddo
pi = 4. * sum
print *, 'computed pi = ', pi

end program compute_pi
```



Parallel Computation of PI

```
program compute_pi
integer                    :: i
integer, parameter         :: n=500000000, dp = kind(1.d0)
real(kind=dp)              :: w,x,sum,pi,d

w=1.0/n; sum=0.0
!$OMP PARALLEL PRIVATE(x,d), SHARED(w,sum)
!$OMP DO REDUCTION(+: sum)
do i=1,n
   x   = (i-0.5) * w
   d   = w * SQRT(1.0 - x**2)
   sum = sum  + d
enddo
!$OMP END DO
!$OMP END PARALLEL
pi = 4. * sum
print *, 'computed pi = ', pi
end program compute_pi
```

HP XC3000

1 core:

real	2.92s
user	2.93s

2 cores:

real	1.50s
user	2.99s

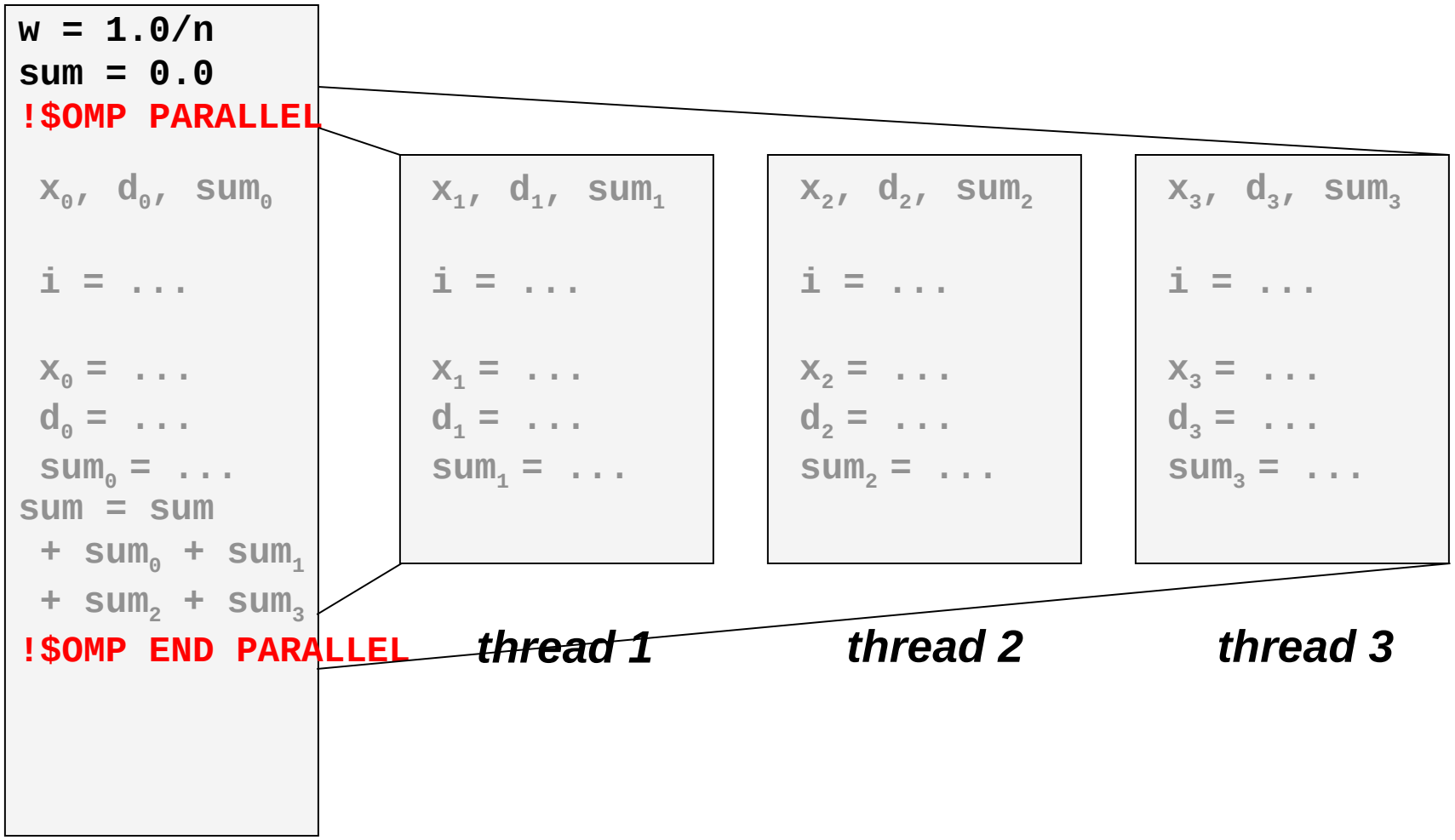
4 cores:

real	0.75s
user	2.97s

8 cores:

real	0.39s
user	3.10s

Parallel Program PI descriptively



Mutual exclusion Synchronization – `critical` (section)

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

```
 a_max = MINUS_INFINITY; a_min = PLUS_INFINITY

!$OMP PARALLEL DO
 do i=1,n

   if (a(i) > a_max) then
!$OMP CRITICAL(MAXLOCK)
     if (a(i) > a_max) then; a_max = a(i); endif
!$OMP END CRITICAL(MAXLOCK)
   endif

   if (a(i) < a_min) then
!$OMP CRITICAL(MINLOCK)
     if (a(i) < a_min) then; a_min = a(i); endif
!$OMP END CRITICAL(MINLOCK)
   endif

 enddo
```

Mutual exclusion Synchronization – `atomic` (*update*)

- `atomic` is a special case of a critical section, that can be used, to set scalar variables in simple assignments.
  - Fortran: `!$OMP ATOMIC`
  - C: `#pragma omp atomic`

- The assignmant must have the following syntax:

```
x = x operator expr
x = intrinsic(x, expr)
```

  Operators

  Fortran: `+, -, *, /, .AND., .OR., .EQV., .NEQV., MAX, MIN, IAND, IOR, IEOR`

  C: `+, -, *, /, &, |, ^, &&, ||`