Shared Memory Programming Models I

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)
University of Heidelberg
INF 368, Room 532
D-69120 Heidelberg
phone: 06221/54-8264
email: Stefan.Lang@iwr.uni-heidelberg.de

WS 14/15

Shared Memory Programming Models I

Communication by shared memory

- Critical section
- Mutual exclusion: Petersons algorithm
- OpenMP
- Barriers – Synchronisation of all processes
- Semaphores

Critical Section

What is a critical section?

We consider the following situation:

- Application consists of P concurrent processes, these are thus executed simultaneously
- instructions executed by one process are subdivided into interconnected groups
  - critical sections
  - uncritical sections
- Critical section: Sequence of instructions, that perform a read or write access on a *shared variable*.
- Instructions of a critical section that may not be performed simultaneously by two or more processes.

$\rightarrow$ it is said only a single process may reside within the critical section.

Mutual Exclusion I

2 Types of synchronization can be distinguished

- Conditional synchronisation
- Mutual exclusion

Mutual exclusion consists of an *entry protocol* and an *exit protocol*:

Programm (Introduction of mutual exclusion)

```
parallel critical-section
{
    process Π [int p ∈ {0, ..., P − 1}]
    {
        while (1)
        {
            entry protocol;
            critical section;
            exit protocol;
            uncritical section;
        }
    }
}
```

Mutual Exclusion II

The following criteria have to be matched:

1. *Mutual exclusion*. At most one process executed the critical section at a time.
2. *Deadlock-freeness*. If two or more processes try to enter the critical section exactly one has to succeed within limited time.
3. *No unneccessary delay*. If a process wants to enter the critical section while all others process their uncritical sections this may not be prevented.
4. *Final entry*. Tries a process to enter the critical section then this must be allowed after limited waiting time (therefore is assumed, that each process in the critical section also leaves it again).

Petersons Algorithm: A Software Solution

We consider at first only two processes and develop the solution step by step ...

First approach: Wait until the other is *not* inside

int *in1*=0, *in2*=0; // 1=drin

Π_1:	Π_2:
while (*in2*) ;	**while** (*in1*) ;
in1=1;	*in2*=1;
crit. section;	crit. section;

- No machine instructions are necessary
- Problem: Reading and writing is not atomic

Petersons Algorithm: Second Variant

First set, then test

int *in1*=0, *in2*=0;

Π_1:	Π_2:
in1=1;	*in2*=1;
while (*in2*) ;	**while** (*in1*) ;
crit. section;	crit. section;

Problem: deadlock possible

Petersons Algorithm: Third Variant

Solve deadlock by choosing one process

Programm (Petersons Algorithm for two processes)

```
parallel Peterson-2
{
    int in1=0, in2=0, last=1;

    process Π1
    {
        while (1) {
            in1=1;
            last=1;
            while (in2 ∧ last==1) ;
            crit. section;
            in1=0;
            uncrit. section;
        }
    }
}
```

```
process Π2
{
    while (1) {
        in2=1;
        last=2;
        while (in1 ∧ last==2) ;
        crit. section;
        in2=0;
        uncrit. section;
    }
}
```

Consistency Models

Previous examples are based on the principle of *Sequential Consistency*:

1. *Read- and write operations are finished in the order of the program*
2. *This sequence is for all processors consistently visible*

Here one expects that $a = 1$ is printed:

```
int a = 0,flag=0;                    // important!
process Π1                           process Π2
    ...                                  ...
    a = 1;
    flag=1;                              while (flag==0) ;
                                         print a ;
```

Here one expects that only for one the **if** condition is true:

```
int a = 0, b = 0;                    // important!
process Π1                           process Π2
    ...                                  ...
    a = 1;                               b = 1;
    if (b == 0) ...                      if (a == 0) ...
```

Consistency Models

Why is there no sequential consistency?

- *Reordering of instructions*: Optimizing compilers can reorder operations for efficiency reasons. Then the first example does not work any more!
- *Out-of-order execution*: e. g. read accesses shall pass slow write accesses (invalidate) (as long as it is not the same memory location). The second example does not work any more!

Total store ordering: Read access may only pass write access

Weak consistency: All accesses may pass each other

In-order sequence can be enforced by special machine instructions,
e. g. *fence*: finish all memory accesses before a new one is started

This operations are inserted,

- through annotation of variables („synchronisation variable“)
- in parallel instructions (e. g. FORALL in HPF)
- by the programmer of synchronisation primitives (e. g. Semaphore)

Peterson for P Processes

Idea: Each passes $P-1$ stages, respectively the last arriving in a particular stage has to wait

Variables:

- $in[i]$: Stage $\in \{1,\ldots,P-1\}$ (!), that Π_i has reached
- $last[j]$: Number of process that arrived as the latest at stage j

Programm (Petersons Algorithm for P processes)

```
parallel Peterson-P
{
    const int P=8;
    int in[P] = {0[P]};
    int last[P] = {0[P]};
    ...
}
```

Peterson for P Processes

Programm (Petersons Algorithm for P Processes cont.)

```
parallel Peterson-P cont.
{
    process Π [int i ∈ {0, ..., P − 1}]
    {
        int j,k;
        while (1)
        {
            for (j=1; j ≤ P − 1; j++)                    // Traverse stages
            {
                in[i] = j;                               // I am in stage j
                last[j] = i;                             // I am the last of stage j
                for (k = 0; k < P; k++)                  // test all others
                    if (k ≠ i)
                        while (in[k]≥in[i] ∧ last[j]==i) ;
            }
            critical section;
            in[i] = 0;                                   // exit protocol
            uncritical section;
        }
    }
}
```

- $O(P^2)$ tests are necessary for entry
- Strategy is fair, who arrives first enters as first

Hardware Locks

Hardware operations to realize of mutual exclusion:

- *test-and-set*: Check whether a memory location has value 0, if yes write the contents of a register into the memory location (as indivisible operation).
- *fetch-and-increment*: Get the content of a memory location in a register and increment the content of the memor locaton by 1 (as indivisible operation).
- *atomic-swap*: Interchange the content of a register with the content of a memory location in an indivisible operation.

In each of the machine instructions a read access followed by a write access has to be executed without break in between!

Hardware Locks

Goal: Machine instruction and cache coherency model ensure exclusive entry into the critical section and generate low traffic on the interconnection network

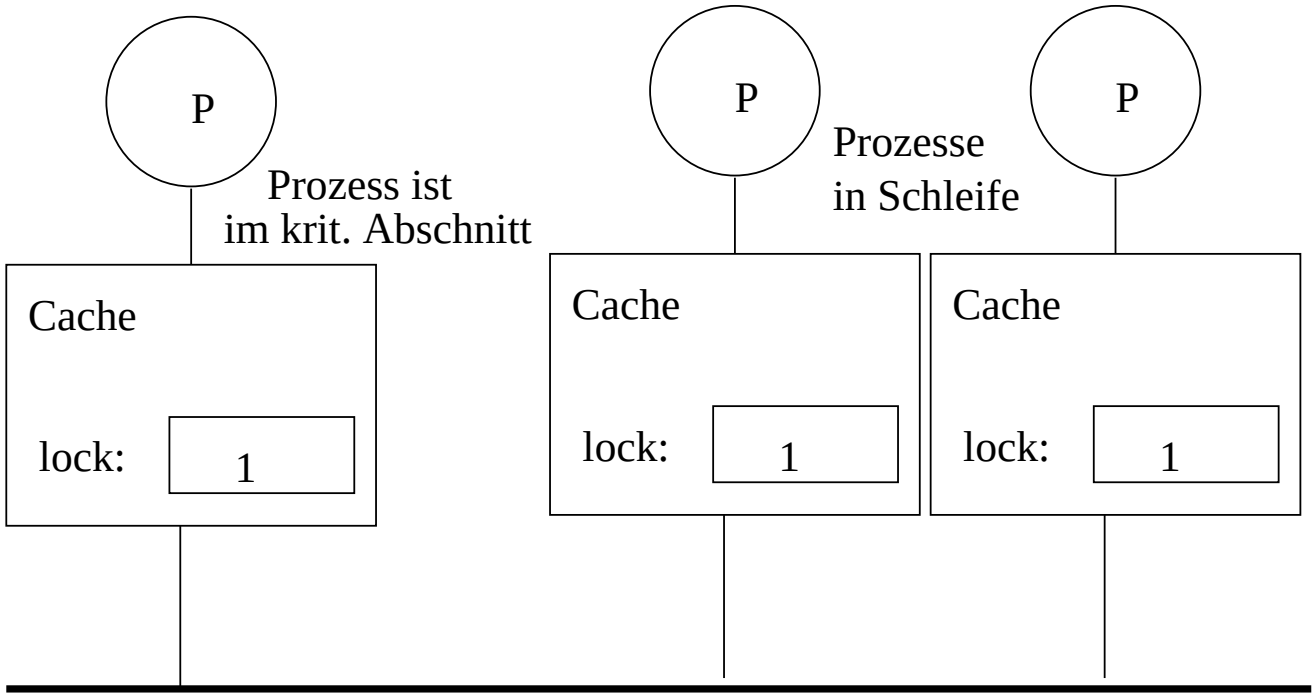
Programm (Spin Lock)

```
parallel spin–lock
{
    const int P = 8;                    // process count
    int lock=0;                         // variable for protection

    process Π [int p ∈ {0, ..., P − 1}]
    {
        ...
        while ( atomic − −swap(& lock)) ;
        ...                             // critical section
        lock = 0;
        ...
    }
}
```

Hardware Locks

What occurs inside the system?



Both waiting processes generate high bus traffic!

Hardware Locks

Activity with MESI protocol: Variable *lock* is 0 and is in none of the caches

- Process Π_0 executes the $atomic - swap$ operation

$\rightarrow$ Read access induces a read miss, block is fetched from memory and obtains the state E (we take MESI as a basis).

$\rightarrow$ Subsequent writing without further bus access, state change from E to M.

- other process Π_1 executes $atomic - swap$ operation

$\rightarrow$ Read miss induces Write-back of the block by Π_0, the state of both copies is now S, after the read access.

- Write access of Π_1 invalidates copy of Π_0 and state in Π_1 is M.
- $atomic - swap$ has result 1 in Π_1 and critical section is not entered by Π_1.
- If both processes execute the $atomic - swap$ operation simultaneously the bus decides finally who wins.
- Assume cache C_0 of processor P_0 and also C_1 have each a copy of the cache block in state S before execution of the $atomic - swap$ operation
- Both read initially the value 0 from the variable *lock*.
- In the following write access both compete for the bus to place their own Invalidate message.
- The winner sets its copy into state M, the loser sets its into state I. The cache logic of the loser finds the state I when it writes and has to arrange that the $atomic - swap$ instruction returns after all the value 1 (the atomic-swap instruction is yet not finished at this time).

Improved Lock

Idea: Do not perform any write access as long as the critical section is occupied

Programm (Improved Spin Lock)

```
parallel improved–spin–lock
{
    const int P = 8;                    // process count
    int lock=0;                         // variable for protection

    process Π [int p ∈ {0, ..., P − 1}] {
        ...
        while (1)
            if (lock==0)
                if (read − and − set(& lock)==0 )
                    break;
        ...                             // critical section
        lock = 0;
        ...
    }
}
```

Improved Lock

1. Problem: Strategy guarantees no fairness
2. Situation with three processes: Two always alternate, while the third can enter
3. Effort if P processes want to enter at a time: $O(P^2)$, instruction $lock = 0$ causes P bus transactions for cache block copies
4. Solution is a queuing lock: During exit from the critical section the process chooses a successor

Ticketing algorithmus:

- Fairness with hardware lock
- Idea: Before lining up in the queue one draws a number. The one with the smallest number is the next to be choosen.

Ticketing Algorithm

Programm (Ticketing Algorithm for P processes)

```
parallel Ticket
{
    const int P=8;
    int number=0;
    int next=0;

    process Π [int i ∈ {0, ..., P − 1}]
    {
        int mynumber;
        while (1)
        {
            [mynumber=number; number=number+1;]
            while (mynumber≠next) ;
            critical section;
            next = next+1;
            uncritical section;
        }
    }
}
```

Ticketing Algorithm

1. Fairness is based on a small duration for drawing a number. Opportunity of a collision is small.
2. Works also for counter overflow, (MAXINT>P)
3. Incrementing of *next* is possible without synchronisation, since this always can only be done by one

Conditional Critical Section I

- Producer-Consumer problem:
  - m processes P_i (producers) generate requests, that shall be finished by n other processes C_j (consumers).
  - The processes communicate by a central waiting queue (WQ) with k positions.
  - Is the WQ full the producers have to wait, is the WQ empty the consumers have to wait.
- Problem: Waiting may not block the (exclusive) access onto the WQ!
- Critical section (manipulation of the WQ) may only be entered *if* WQ is not full (for producer), resp. not empty (for consumer).
- Idea: Entry on a trial basis and busy-wait

Conditional Critical Section II

Programm (Producer–Consumer Problem with Active Waiting)

```
parallel producer-consumer-busy-wait
{
    const int m = 8; n = 6; k = 10;
    int orders=0;
    process P [int 0 ≤ i < m]
    {
        while (1) {
            produce request;
            CSenter;
            while (orders==k){
                CSexit;
                CSenter;
            }
            store request;
            orders=orders+1;
            CSexit;

        }
    }
}
```

```
process C [int 0 ≤ j < n]
{
    while (1) {

        CSenter;
        while (orders==0){
            CSexit;
            CSenter;
        }
        read request;
        orders=orders-1;
        CSexit;
        process request;
    }
}
```

Conditional Critical Section III

- Permanent Entry and Exit of the critical section is inefficient if several are waiting (trick of the improved lock doesn't help)
- (Practical) solution: Random delay between *CSenter*/*CSexit*, *exponential back-off*.

OpenMP (Open Multi Processing) I: Approach

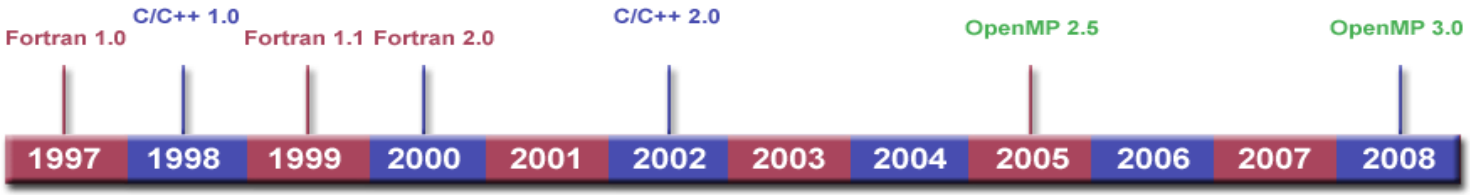
OpenMP is a parallel programming model on
the basis of the following assumptions:

- A process uses multiple threads (lightweight processes)
- All threads share the same status variables of the program
- Each thread can own additional private variables
- Threads can run on different processors/cores
- Mechanisms for synchronization and for locking are provided

OpenMP II

What is OpenMP?

- API (application programming interface) to write multi-threaded applications
  - Compiler directives and library functions
  - Standardized for C/C++ and Fortran
- New standard under steady enhancement
- Important model, since many important companies are participating
- Parallelisation process using OpenMP
  - Program is parallelized in steps
  - Starting point is the serial version, which remains in many cases unchanged
  - Parallelism is not coded directly, but is influenced by directives



OpenMP Example I

Hello World:

- The original program is preserved
- Execution when an environment variable is set:

```
void main(void)
{
#pragma omp parallel
  {
     printf(„Hallo Welt\n“);
  }
}
```

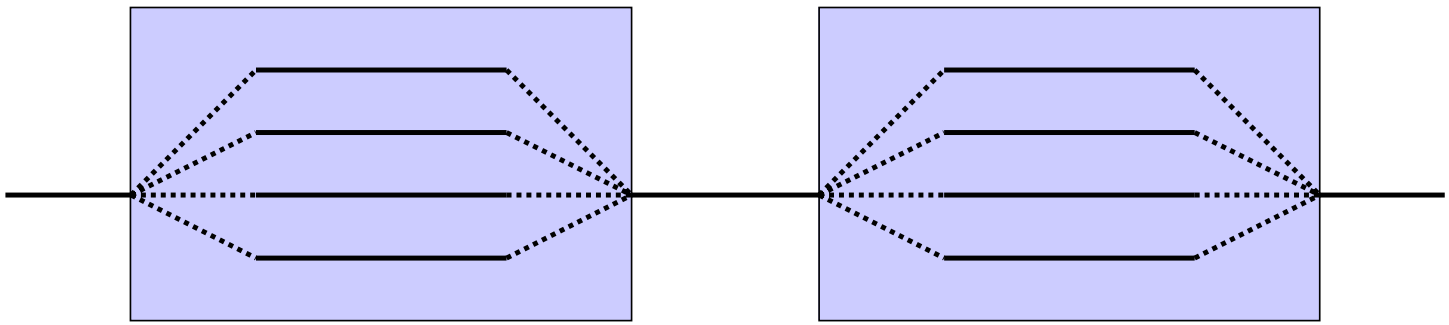
```
(~): export OMP_NUM_THREADS=4
(~): ./hello-openmp
Hallo Welt
Hallo Welt
Hallo Welt
Hallo Welt
```

OpenMP Example II

Parallel Regions (blocks)

- By usage of the compiler directive `#pragma omp parallel` the following block is executed in parallel
- Therefore a set of threads is started
  - the thread count depends on the environment variable `OMP_NUM_THREADS`, that can be changed by the program
  - we speack of fork-join parallelism
- After all threads are finished, these are either terminated or remain waiting

Control flow in block 1 control flwo in block 2



OpenMP Example III

- Primary application area of OpenMP is the parallelisation of loops
- **#pragma omp parallel for**
- i-loop will be executed simultaneously by `OMP_NUM_THREADS` threads

Matrix-Matrix multiplication

```
#pragma omp parallel for
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < K; k++)
      C[i,j] = A[i,k] * B[k,j];
```

OpenMP Example IV

Runtime conditions

- if multiple threads read and write the same variable, inconsistencies can occur
- this is comparable to the already known inconsistencies in shared-memory architectures

Example: Scalar product

```
sum = 0.0;
#pragma omp parallel for
for (i = 0; i < N; i++)
      sum = sum + x[i] * y[i];
```

OpenMP Example V

Solution 1: Locking

- To secure the addition of the summing-up one can declare these as atomic or critical
- Disadvantage: This is inefficient, because the threads can not work in parallel anymore

```
sum = 0.0;
#pragma omp parallel for
for (i = 0; i < N; i++)
#pragma omp critical
{
      sum = sum + x[i] * y[i];
}
```

OpenMP Example VI

Solution 2: Private variables

- In parallel regions specific variables can be declared as private
- This can be written in a more compact form

```
sum = 0.0;
#pragma omp parallel private(local_sum)
{
  local_sum = 0.0;
#pragma omp parallel for
  for (i = 0; i < N; i++)
    local_sum = local_sum + x[i] * y[i];

#pragma omp critical
  { sum = sum + local_sum; }
}
```

OpenMP Example VII

Solution 3: Reduction variables

- Such cases are typical, one can declare critical variables as reduction variables
- Within threads these are generated as private and then connected with an appropriate operation at the loop end (synchronized)

```
sum = 0.0;
#pragma omp parallel for reduction (+ : sum)
for (i = 0; i < N; i++)
     sum = sum + x[i] * y[i];
```

OpenMP Pragmas I

- Directives (in C):
  - **#pragma omp clauses** . . .
  - are ignored by non-OpenMP compilers
- Parallel regions (blocks):
  - **#pragma omp parallel**
  - following block ({. . .}) is executed in parallel
- Variable scoping
  - **#pragma omp private(. . .) shared(. . .) reduction(. . .) firstprivate(. . .) lastprivate(. . .)**
  - defines which variables are used together and which are used as copies in each thread
  - **shared is default value**

OpenMP Pragmas II

- Synchronization
  - **#pragma omp atomic, critical, ordered, barrier, flush**
  - essential for program correctness
- Parallel loops (work-sharing)
  - **#pragma omp parallel for**
  - following **for** is parallelized
  - type of distribution can be determined with **schedule** clause
  - e.g. **schedule(dynamic,4)**: each thread is assigned four loop iterations (1. . .4, 5. . .8) and new ones, as soon as a thread is ready
  - other variant are **static, guided** and **runtime**

OpenMP Run Time Environment I

Run time environment

- Processor count
  - **omp_get_num_procs()**
- Thread count
  - **omp_set_num_thread(int)**
  - **omp_get_num_thread()**
    same as envrionment variable **OMP_NUM_THREADS**
  - **omp_get_thread_num()**

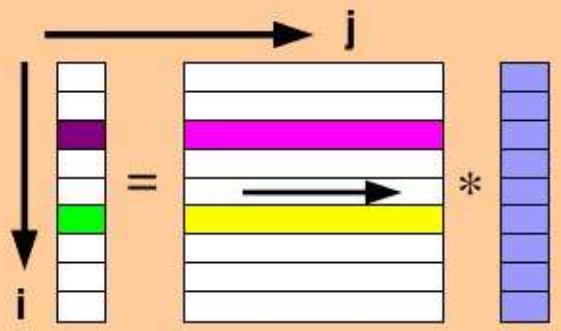
OpenMP Run Time Environment II

Run time environment

- Dynamic mode: Is in different blocks a different number of threads allowed?
  - **omp_set_dynamic(), omp_get_dynamic()**
  - equal to **OMP_DYNAMIC (TRUE / FALSE)**
- Nesting: Are in parallel regions new thread teams allowed? (nested threads)
  - **omp_set_nested(), omp_get_nested()**
  - equal to **OMP_NESTED (TRUE / FALSE)**

OpenMP in Practise: Matrix-Vector Product

```
#pragma omp parallel for default(none) \
            private(i,j,sum) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
   sum = 0.0;
   for (j=0; j<n; j++)
     sum += b[i][j]*c[j];
   a[i] = sum;
}
```



TID = 0

```
for (i=0,1,2,3,4)
 i = 0
sum =    b[i=0][j]*c[j]
  a[0] = sum
 i = 1
sum =    b[i=1][j]*c[j]
  a[1] = sum
```

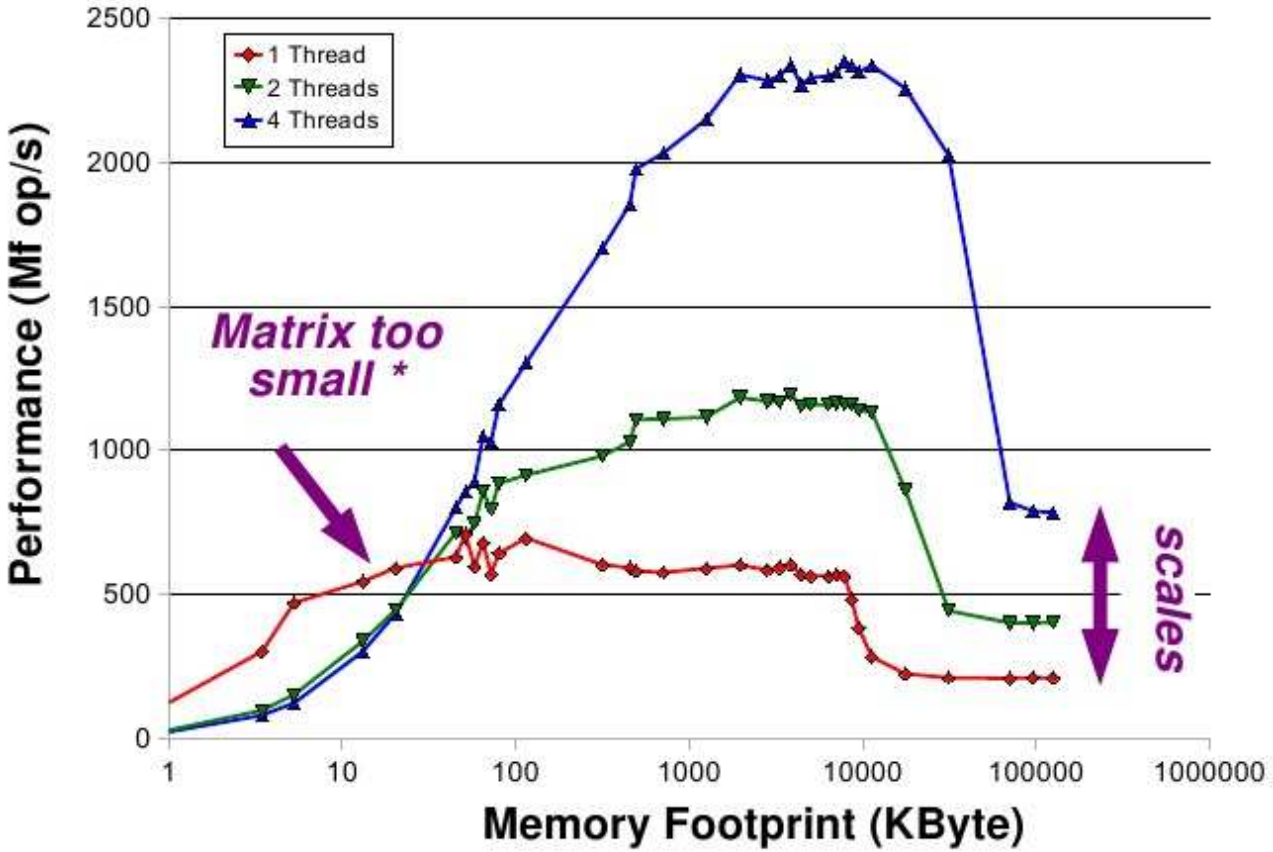
TID = 1

```
for (i=5,6,7,8,9)
 i = 5
sum =    b[i=5][j]*c[j]
  a[5] = sum
 i = 6
sum =    b[i=6][j]*c[j]
  a[6] = sum
```

... etc ...

openmp.org

OpenMP in Practise: Scaling behaviour in MFLOPs



*) With the IF-clause in OpenMP this performance degradation can be avoided

openmp.org

OpenMP in Practise: IF-Clause

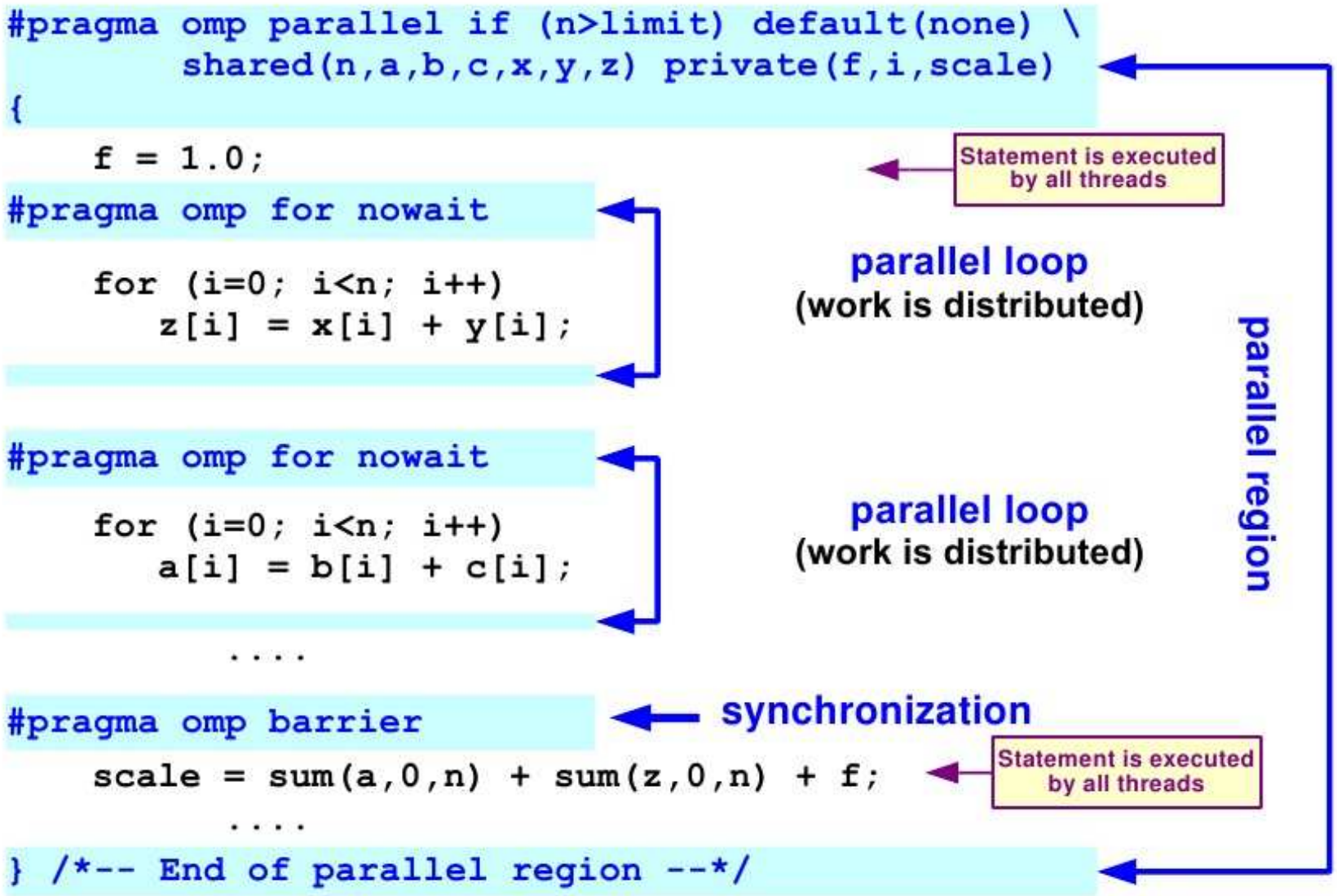
if (scalar expression)

- ✔ ***Only execute in parallel if expression evaluates to true***
- ✔ ***Otherwise, execute serially***

```
#pragma omp parallel if (n > some_threshold) \
        shared(n,x,y) private(i)
  {
    #pragma omp for
     for (i=0; i<n; i++)
        x[i] += y[i];
  } /*-- End of parallel region --*/
```

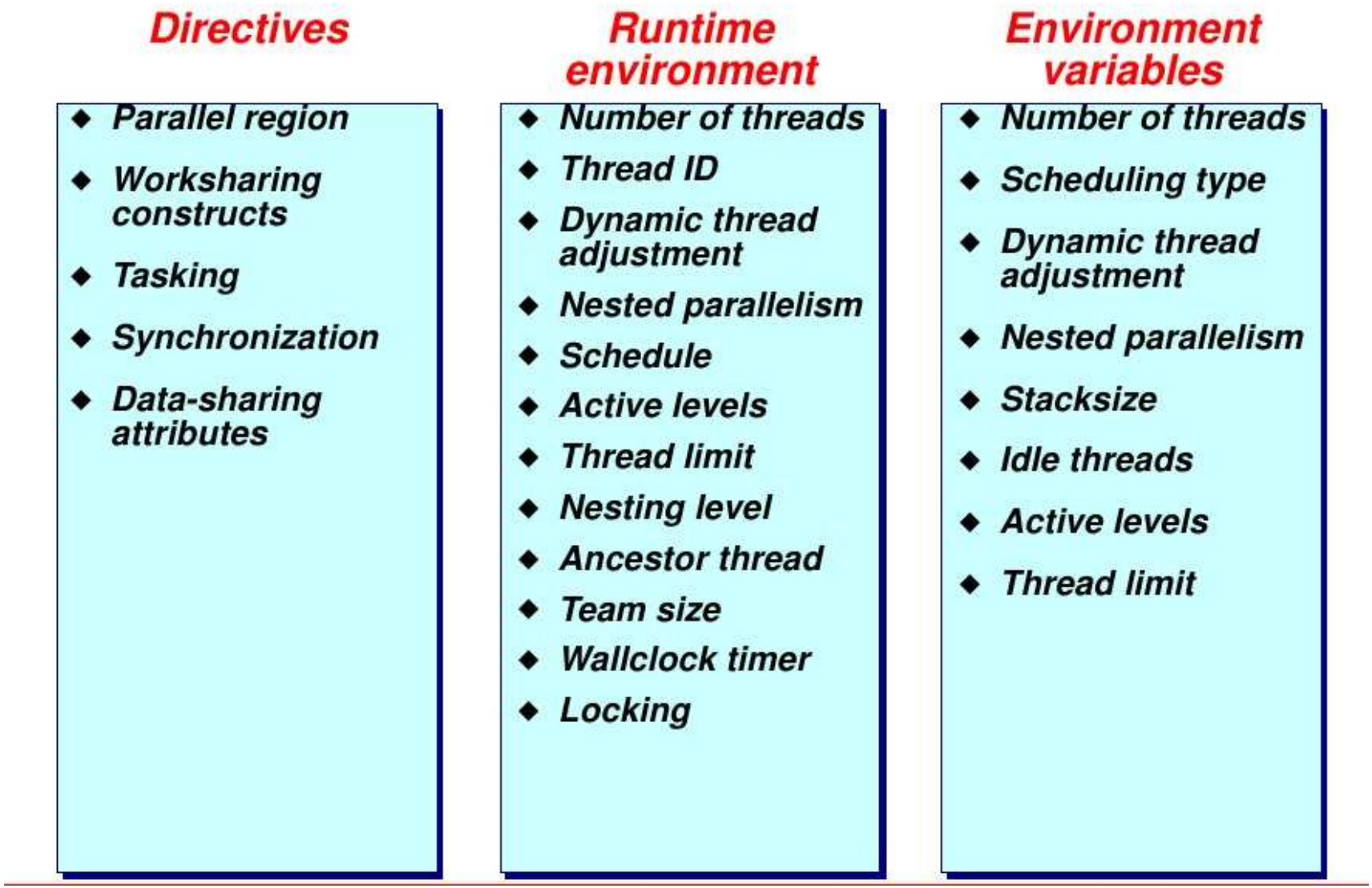
openmp.org

OpenMP in Practise: More Elaborate Example



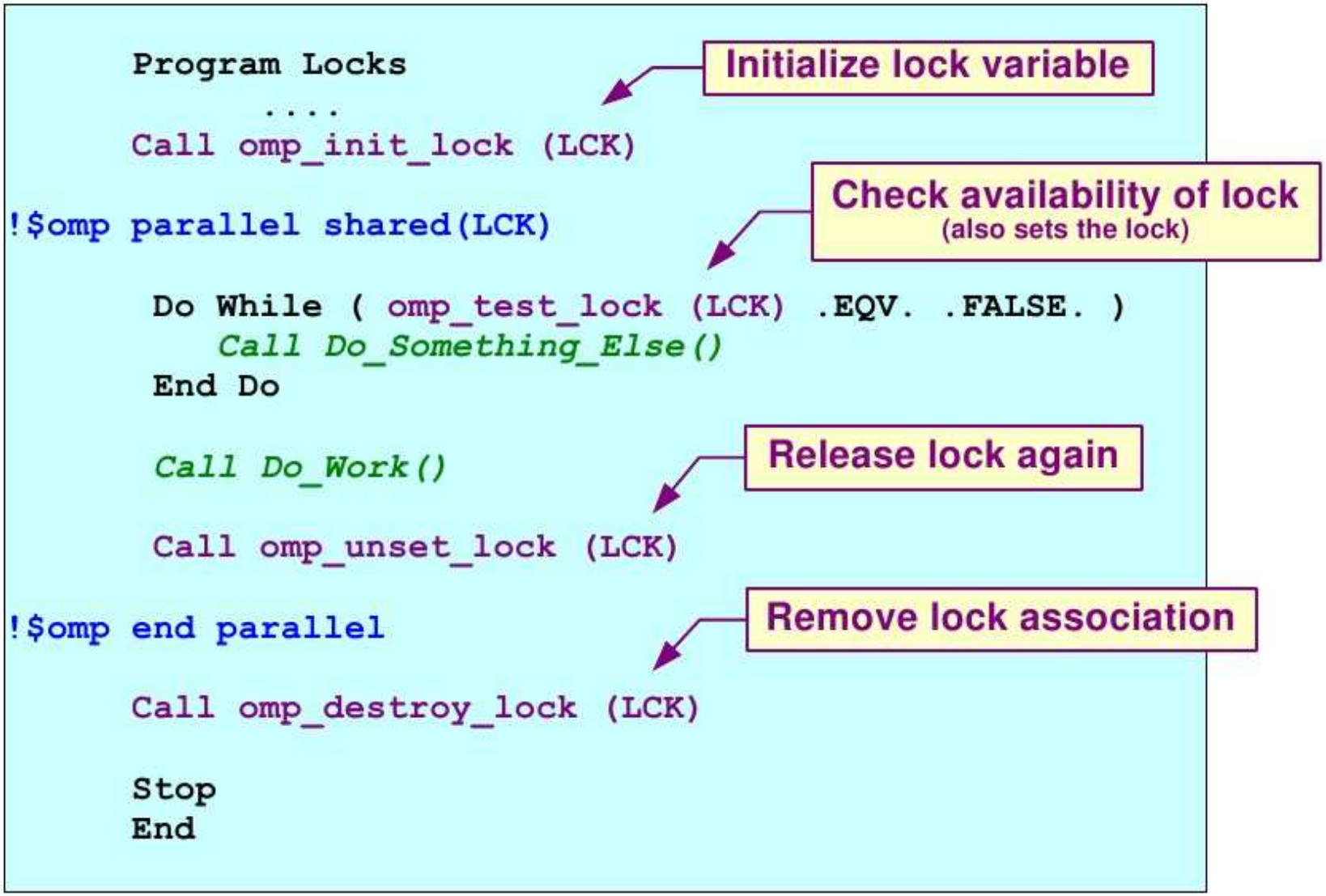
openmp.org

OpenMP in Practise: OpenMP Summary



openmp.org

OpenMP in Practise: Locking Mechanism



openmp.org

OpenMP in Practise: Scheduling I

schedule (static | dynamic | guided | auto [, chunk])
schedule (runtime)

static [, chunk]

- *Distribute iterations in blocks of size "chunk" over the threads in a round-robin fashion*
- *In absence of "chunk", each thread executes approx. N/P chunks for a loop of length N and P threads*
  - *Details are implementation def ned*
- *Under certain conditions, the assignment of iterations to threads is the same across multiple loops in the same parallel region*

openmp.org

OpenMP in Practise: Scheduling II

dynamic [, chunk]

- ✔ *Fixed portions of work; size is controlled by the value of chunk*
- ✔ *When a thread f nishes, it starts on the next portion of work*

guided [, chunk]

- ✔ *Same dynamic behavior as "dynamic", but size of the portion of work decreases exponentially*

auto

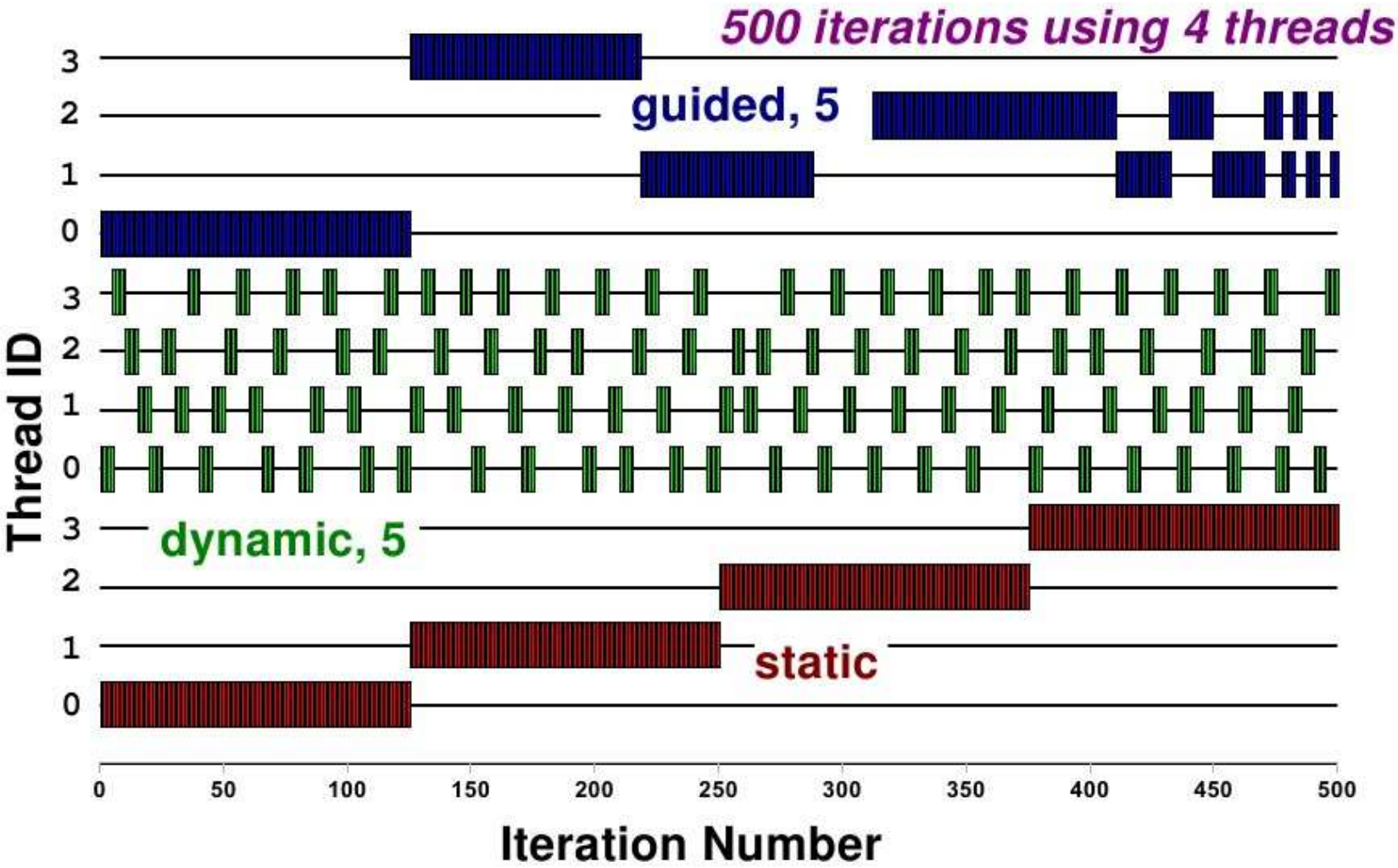
- ✔ *The compiler (or runtime system) decides what is best to use; choice could be implementation dependent*

runtime

- ✔ *Iteration scheduling scheme is set at runtime through environment variable OMP_SCHEDULE*

openmp.org

OpenMP in Practise: Scheduling III



openmp.org