Introduction to OpenMP

Daniel Langr, Ivan Šimeček, and Pavel Tvrdík

Department of Computer Systems
Faculty of Information Technology
Czech Technical University in Prague

Parallel and Distributed Programming (MIE-PDP)
Summer Semester 2020/21, Lecture 02
(Version Timestamp: 19. 2. 2021 22:54)

https://courses.fit.cvut.cz/MIE-PDP

Lecture outline

1. Parallel threads in general
2. The OpenMP Library
3. OpenMP directives in general
4. Creation of parallel regions using the parallel directive
5. Directives defining the type of parallel region processing
6. Loop data parallelism using the parallel for directive
7. Functional parallelism using the task directive
8. Basic OpenMP operations
9. Synchronization mechanisms in OpenMP
10. User controlled escape from a parallel region
11. Further internet OpenMP references

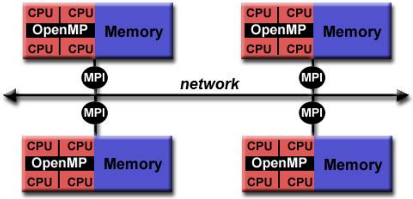
Programming at the thread level

- The standard today is the POSIX library (IEEE POSIX, see BIE-OSY).
- POSIX threads (pthreads) is a library of basic operations in the C language for writing arbitrary multithreaded programs.
- The POSIX API contains low-level functions, for example for:
  - thread management (creation, termination, ...),
  - synchronization and coordination among threads (e.g., locking of shared resources),
  - scheduling and running threads.

  A user has to take care of many implementation details.
- The API is suitable for MIMD applications with complicated schemes of thread coordination:
  - of client-server type (WWW servers, print servers),
  - for I/O services or GUI.
- For SPMD and data parallel applications unnecessary complicated.

The OpenMP Library

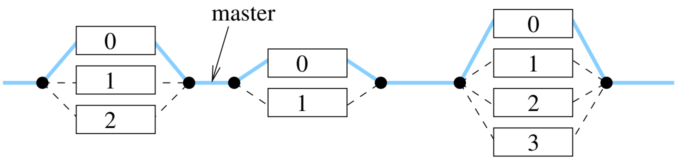
- The OpenMP library (www.openmp.org) is a high-level API for programming of multithreaded applications on a (virtually) shared memory.
- Role of OpenMP (Courtesy of https://computing.llnl.gov/tutorials/openMP/#ProgrammingModel):



- The OpenMP API consists of 3 main parts:
  - parameterizable **directives** for the compiler (over 50),
  - system (global) **variables** (over 20),
  - library of system **operations** of the runtime environment (over 60).
- A **thread** is the smallest piece of code that can be scheduled and executed at the OS level and that shares the context of a **process**.

Fork-join OpenMP Programming Model

- An OpenMP implementation code is an **explicit** model of a parallel computation – the programmer has full control and responsibility for parallel execution.
- In selected parts of originally sequential code, called **parallel regions**, parallel threads are created, executed and terminated using the `fork-join` mechanism.



- Outside of these parallel regions there exists only the **master** thread.

Notes:

- OpenMP can utilize a **thread pool** to reduce the overhead of creating and terminating of threads.
- OpenMP directives are not recommended to be mixed with other thread libraries.

Motivation behind the OpenMP

- Incremental method to parallelize a sequential code in C/C++/Fortran with insertions of directives.
- Portability of a parallel code.
- Standardization and support of industry.
- Simplicity of use.

What should not be expected from OpenMP?

- Not designed for programming on distributed memory.
- Does not guarantee the most efficient way of utilization of shared memory space.
- Cannot automatically check data dependencies, data races and data access collisions, deadlocks $\Rightarrow$ programmer responsibility.
- Cannot synchronize the computation with I/O operations.

A comprehensive courseware on OpenMP is, e.g., on https://computing.llnl.gov/tutorials/openMP (B.Barney, LLNL).

Usual method of multithreaded parallelization

- By inserting OpenMP **directives** in the sequential code we construct **parallel regions** in which several concurrent threads will perform the computation or we create **concurrently running tasks**, each task executed by one thread.
- It is not allowed to jump out of a parallel region (with the exception of the directive `cancel`, see Slide 53) or jump into the region from outside.
- Individual variables are assigned their OpenMP **properties**.
- Setting up the runtime parameters is performed and inquired using library operations.
- Within parallel regions **thread-unsafe** operations should be avoided.
- We compile the code using compiler switches that ensure compilation of the OpenMP directives. In case of the GCC compiler, it is the switch `-fopenmp`.
- A correctly written OpenMP source code compiled **without** this switch becomes a standard sequential C/C++/Fortran code.

Generic syntax of OpenMP programs

```
#include <omp.h>
int main()
{ int x1, x2, s3;
//Now only the master thread is running (sequential code).
//A parallel region follows where 4 new threads are created.
#pragma omp parallel private(x1,x2) shared(s3) num_threads(5)
  {
  //Inside the parallel region 5 threads are running.
  //Variables x1, x2 are local = each thread
  //has its own instances of them.
  //Variable s3 is shared by all threads.
  //Here, other directives can appear
  //or calls of runtime system procedures.
  }
//At the end of the par.region, the 4 new threads are killed.
//Only the master thread goes on sequentially.
}
```

OpenMP memory model

- OpenMP supports a **relaxed consistence memory model**.
- Threads can keep local copies of shared variables (in a cache memory) and are not forced to commit immediately (= *write-through*) every local update of the cached value into the shared memory.
- If we need to make sure that all threads see identical shared memory contents, it is the responsibility of the programmer to force individual threads, e.g., using the `flush()` operation, to explicitly flush the local values of shared variables into the shared memory.
- See Lecture 4 for more details.

Disclaimer 1

All codes presented in this lecture are pseudocodes, they are not compiler-ready, and do not comply with required syntax rules of the C language and its derivatives and libraries.

OpenMP directives in general

- In this course we consider only C/C++.
- The general directive syntax:

```
#include <omp.h>
int main()
{ ...............
 #pragma omp directive clause1 clause2
 {...............}
}
```

- Case-sensitive syntax, conventions of C/C++ compiler directives.
- The ordering of directive's clauses on a codeline is irrelevant.
- Each directive applies to the succeeding statement (structured block).
- At most 1 directive name can be placed on one codeline (except for `parallel for`).
- Before and after `#pragma ...` on a codeline, only whitespace can appear.
- Long directive lines can be continued on succeeding codelines by

Creation of parallel regions using the **parallel** directive

- The most important directive: it creates **parallel regions**.
- Possible clauses:
  - `if(condition)`: a parallel region is created if `condition` holds.
  - `num_threads(expr)`: the number of threads in the parallel region.
  - `properties(variable_list)`: OpenMP properties in a parallel region - see later.
- The value of `expr` determines the number of parallel threads in a parallel region **including** the original master thread 0.
- The code of the parallel region is cloned for all threads that start to execute it.
- Every parallel region is ended with an implicit **barrier** call.
- After the final barrier, all newly created threads are terminated (or placed in a *thread pool*, see Slide 5) and only the master thread 0 goes on.
- If during the parallel region execution, **any** of threads is **prematurely terminated**, then **all threads are prematurely terminated** and so is the whole program.

Properties of variables of a parallel region

- Property `shared`: given scalar variable (it cannot be an array or structure) is **shared** by all threads. The programmer is responsible for ensuring the proper concurrent access by parallel threads.
- Property `private`: given variable is **thread-local**, i.e., each thread has an independent uninitialized instance of this variable.
- Property `firstprivate`: given variable is **private** and it is initialized in all threads to its value in the master thread just before entry into the parallel region.
- Property `lastprivate` (only in parallel loops): given variable is **private** and after the parallel region is terminated, its value from the **sequentially last** iteration is copied to the master thread.
- Property `default`: defines which of the preceding properties becomes **implicit** for all variables in the parallel region (if not stated otherwise).
- **Important!!!** Properties of pointers apply only to them, not to the pointed data.

The `reduction` property of a private variable

- The `reduction(operator:variable)` property: defines that the given shared variable is instantiated into all threads after the parallel region terminates, all its local instances are **reduced** using a given **reduction operator** and the result will be written into the shared variable of the master thread. This value will be valid after the parallel region ends.
- It must be a variable of a scalar type.
- Possible reduction operators are: `+, *, -, &, ^, |, &&, ||`.
- The operator must not be overloaded for the given data type.
- The operator may not be **associative** due to the rounding errors for float numbers.
- A parallel region can contain several variables with the property `reduction`, but each of them can participated in at most one reduction.
- A reduction variable cannot be combined with the `task` directive.

Parallel reduction using a variable with the `reduction` variable

Code 1: An example of a parallel reduction

```
int main() {
int n = some_big_number;
float a[n], b[n], result = 0.0;
#pragma omp parallel
 {
   #pragma omp for
   for (int i=0; i < n; i++) {
     a[i] = ...; b[i] = ...; } //initialization
   #pragma omp for reduction(+:result)
   for (int i=0; i < n; i++) result += a[i]*b[i];
 } //end parallel
printf("Result = %f", result);
}
```

Implementation and complexity of the OpenMP reduction

There are 2 possible implementations of the parallel reduction.

- **Linear (sequential)**: all threads write its local variables into a shared array and after the barrier one thread computes the reduction result:

$$T_{\mathrm{PR}}(p,p) \doteq (1+p)\cdot(T_{comp}+T_{mem}) + T_{barr}(p).$$

$$T_{\mathrm{PR}}(n,p) \doteq \left(\frac{n}{p}+p\right)\cdot(T_{comp}+T_{mem}) + T_{barr}(p).$$

- **Logarithmic** (see Slide 34 in Lecture 1): the barrier is required in every parallel step!

$$T_{\mathrm{PR}}(p,p) \doteq (1+\log p)\cdot(T_{comp}+T_{mem}) + \log p\cdot T_{barr}(p)$$

$$T_{\mathrm{PR}}(n,p) \doteq \left(\frac{n}{p}+\log p\right)\cdot(T_{comp}+T_{mem}) + \log p\cdot T_{barr}(p).$$

OpenMP usually uses simple operators ($T_{comp}, T_{mem} \ll T_{barr}$) and small number of threads $p \ll n$, therefore the linear version is faster and it is used!!!

The `threadprivate` property

- The basic `private` property defines the variable only within the parallel region. Every thread creates its local instance whose value before and after the region is undefined (this is addressed with the `firstprivate` and `lastprivate` properties).
- The `threadprivate()` property defines the a global scope for these private variables. Their instances are defined and preserve their values throughout all parallel regions of a given program. The number of threads in the program must not change.
- This property excludes any other properties (`firstprivate`,. . .).
- Thus, if its value is to be defined prior to the first parallel region, the `copyin` clause must be used – initialization from the variable of the master thread.
- Property `threadprivate` has an exception: it cannot be locally redefined within a parallel region.

Parallel persistent counter using threadprivate variable

Code 2: Parallel counter

```
int count=0;
int tcount=0;
#pragma omp threadprivate(tcount)
#pragma omp parallel copyin(tcount)
{ if (event_happened) tcount++;
  ...   }
  .......
#pragma omp parallel
{ if (another_event_happened) tcount++;
  ...   }
#pragma omp parallel shared(count) {
#pragma omp atomic update //will be explained on Slide 47
    count += tcount;
}
```

Question:

How can this algorithm be implemented using the reduction property?

The number of threads in a parallel region in general

- Threads are numbered from 0 (master) to $p-1$.
- The number of threads in a parallel region p is defined using these statements **in this order if they are used**. The **first** valid statement applies!!!!
  1. The clause `if(condition)` is evaluated. If not true, the region will be executed sequentially using the master thread 0.
  2. The value of expression in clause `num_threads(expr)`.
  3. The value of the last call of procedure `omp_set_num_threads(expr)`.
  4. The latest value of the system variable `OMP_NUM_THREADS`.
- If the user does not specify either of these, the value is an implementation-dependent default one or **dynamic** (e.g., OS knows the number of CPU cores).
- The `if()` and `num_threads()` clause can appear at most once in front of a parallel region.
- Other mechanisms to control the number of threads are **synchronization directives** (see Slide 41).

Directives defining the type of parallel region processing

- A predefined group of threads executes in parallel the code of a region.
- OpenMP is primarily focused on **data parallelism**.
- It is defined with the `for` directive that realizes data parallelism by a parallel execution of data independent loop iterations (see the next Slide).
- The **function parallelism** is supported with 2 mechanisms: the `sections` directive and the `task` directive.
  - Parallelization by decomposition into parallel sections using the `sections` directive will not be be considered in this course.
  - Function parallelism by decomposition into disjoint parallel subcomputations (Divide and Conquer) using the `task` directive will be explained from Slide 35.
- Serialization: a structured code block of a parallel region denoted by a `single` or `master` directive will be executed by only one thread (sequentially) (see Slide 42).

Loop data parallelism using the `parallel for` directive

- A directive for assigning individual iterations of a `for` loop **inside a parallel region** to individual threads.
- An implicit barrier at the end of the loop.
- Possible clauses:
  - `schedule()`: defines the way the loop iterations are scheduled among threads (see the next Slide).
  - `collapse()`: parallelization of nested loops. Implicitly, the `for` directive applies only to the outermost loop (see Code 0 on Slide 31).
  - `ordered`: the order of iteration execution is the same as in sequential processing (will not be discussed).
  - `nowait`: threads after completion of all iterations will not wait on the final barrier.

Note:
Since the combination of `parallel` and `for` directives is so common, they are allowed to be written as a single `parallel for` directive on the same codeline (see Slide 10).

Clause `schedule` of the `for` directive

`schedule(type)` or `schedule(type,chunk-size)`.

- `type`::=`static`: If the `chunk-size` expression is stated, the threads are assigned **statically cyclically chunks** of `chunk-size` successive loop iterations. Otherwise, the threads are assigned uniformly chunks of size $\frac{n}{p}$.
- `type`::=`dynamic`: If the expression `chunk-size` is stated, the threads are assigned **chunks** of `chunk-size` successive loop iterations. Otherwise, the `chunk-size` is 1.
- `type`::=`guided`: The threads are **dynamically** assigned chunks of x iteration where $x = \max(\lceil \#$ **so far unassigned** iterations$/p\rceil$, `chunk-size`) (except for the last remaining iteration chunk). If the `chunk-size` is not stated, it is 1 by default.
- `type`::=`runtime`: Scheduling is decided at the moment of loop execution according to the system variable `OMP_SCHEDULE`.
- `type`::=`auto`: Scheduling is left to the compiler and OS.

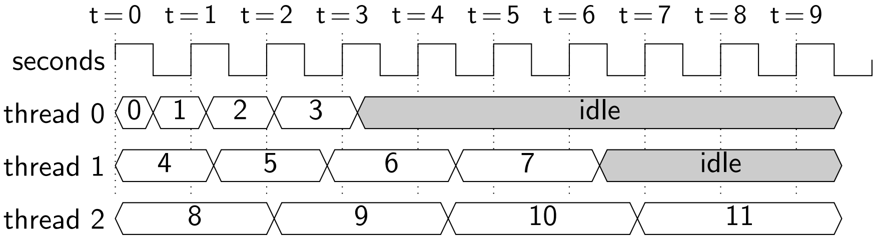
An example of loop iterations scheduling

```
int i, myID; iterationsCount = 12;
omp_set_num_threads(3); //3 parallel threads
#pragma omp parallel for private(i,myID) schedule(type[,k])
for (i = 0; i < iterationsCount; i++) {
  myID = omp_get_thread_num(); //get my local thread id
  printf("myID = %i, iteration = %i", myID, i);
  sleep(0.5+0.2*i);
  }
}
```

- Assume that
  - the iteration scheduling overhead is zero,
  - function `sleep`(x) makes a thread to sleep for x seconds,
  - all functions terminate without errors.
- Let us show how the clause `schedule` parameters have implications for the resulting execution time.

Clause `schedule(static)`

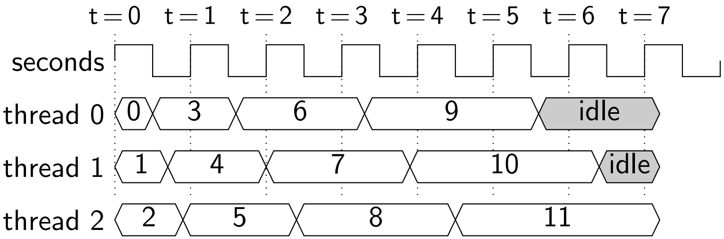
- Each thread is assigned a chunk of four successive iterations.



The total time is 9.6 seconds.

Clause `schedule(static,1)`

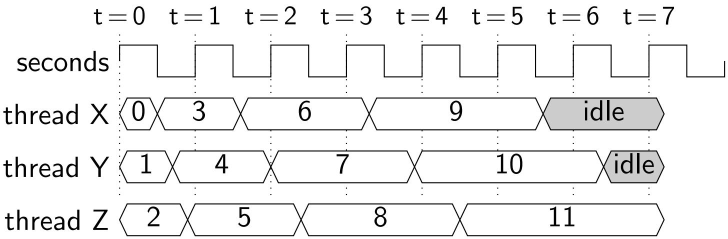
- Each thread is assigned statically every 3rd iteration based on its number.



The total time is 7.2 seconds.

Clause `schedule(dynamic)`

- Every thread, once executes its assigned iteration, is assigned the currently next free one.
- It is similar to the previous case, except that the primary assignment is in some unpredictable order.



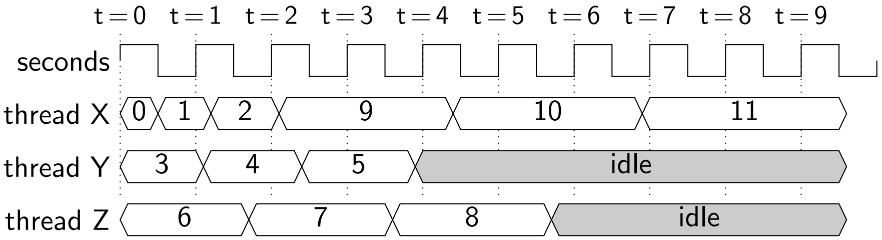
The total time is 7.2 seconds.

Note:

- X,Y,Z is a permutation of 0,1,2.

Clause `schedule(dynamic,3)`

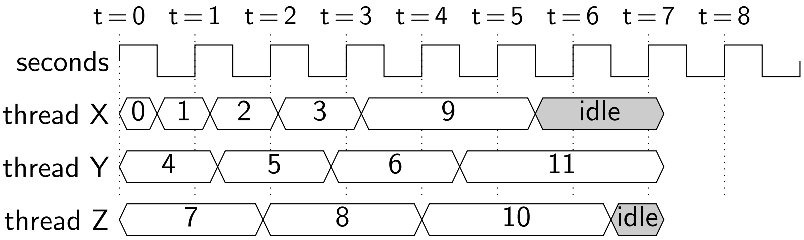
- As soon as a thread executes the assigned chunk of 3 iterations, it is dynamically assigned the next free chunk of size 3.



The total time is 9.6 seconds.

Clause `schedule(guided)`

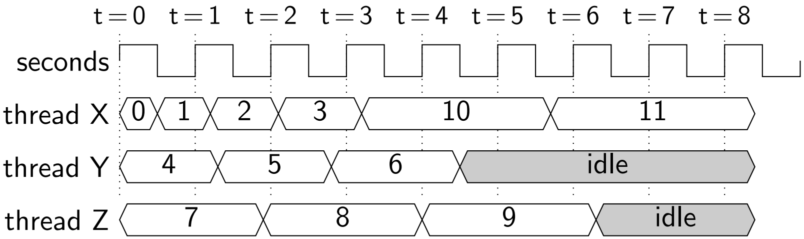
- Dynamic scheduling where a thread is assigned the next free chunk of size computed uniformly from the remaining number of iterations, up to granularity of 1.



The total time is 7.2 seconds.

Clause `schedule(guided,3)`

- Granularity of assigned chunks cannot drop below 3, except possibly for the last one.



The total time is 8.4 seconds.

Discussion of the `schedule` clause

- `schedule(static[,k])`:
  - The least overhead.
  - Iterations are distributed among threads uniformly.
  - Therefore, ideal if all iterations have a similar execution time, see the next Slide.
  - The `k` parameter affects the iteration shuffling.
- `schedule(dynamic[,k])`:
  - Higher overhead due to synchronization.
  - Higher `k` decreases the overhead.
  - Favorable if the execution times of individual iterations vary.
- `schedule(guided[,k])`:
  - Higher overhead due to synchronization.
  - Higher `k` decreases the overhead.
  - Favorable if the execution times of individual iterations monotonically grow.

Uniform static scheduling of iteration chunks

The `schedule(static)` clause is equivalent to the Code 1 on Slide 14:

Code 3: An equiv.implementation of the parallel reduction

```
int main() {
nt n = some_big_number;
int p = omp_get_num_threads();
int k = n/p;
float a[n], b[n], result = 0.0;
#pragma omp parallel
 {
  #pragma omp for schedule(static,k)
  for (int i=0; i < n; i++) {
    a[i] = ...; b[i] = ...; } //initialization
  #pragma omp for schedule(static,k) reduction(+:result)
  for (int i=0; i < n; i++) result += a[i]*b[i];
 } //end parallel
printf("Result = %f",result);
}
```

Parallelization of nested loops I

- Consider a 2-level loop with data independent iterations of constant size.
- Consider for simplicity implicit static scheduling.

Code 4: Only the outermost loop is parallelized.

```
#pragma omp parallel for
for (int y = 0; y < YM; y++)
 for (int x = 0; x < XM; x++)
  {computeElement(x,y);}
```

Code 5: 2-level loop transforms internally to a simple one and it is parallelized.

```
#pragma omp parallel for collapse(2)
for (int y = 0; y < YM; y++)
 for (int x = 0; x < XM; x++)
  {computeElement(x,y);}
```

- Code 4 is the most common one. Threads get $\lceil \mathrm{YM}/p \rceil * \mathrm{XM}$ or $\lfloor \mathrm{YM}/p \rfloor * \mathrm{XM}$ iterations. The thread pool is handled only once.
- In Code 5 threads get $\lceil \mathrm{YM} * \mathrm{XM}/p \rceil$ or $\lfloor \mathrm{YM} * \mathrm{XM}/p \rfloor$ iterations. So if p does not divide YM and if XM is big, Code 5 leads to more uniform distribution of iterations among threads and therefore faster execution. On the other hand, Code 5 is slowed down due to recomputing the linearized indexes to 2D.

Parallelization of nested loops II

- **Only** the inner x-loop is parallelized.
- One thread gets $\lceil \mathrm{XM}/p \rceil * \mathrm{YM}$ iterations.

Code 6: Inner loop iterations are scheduled YM times, a barrier is at the end of the loop body.

```
for (int y = 0; y < YM; y++)
#pragma omp parallel for
  for (int x = 0; x < XM; x++)
    {computeElement(x,y);}
```

Code 7: Inner loop iterations are scheduled only once, a barrier is at the end of the loop body.

```
#pragma omp parallel
for (int y = 0; y < YM; y++)
  #pragma omp for
  for (int x = 0; x < XM; x++)
    {computeElement(x,y);}
```

- In Code 6 the outer y-loop is executed by the main thread sequentially. In Code 7 all threads execute it concurrently, each with its private variable y.
- Code 7 can have **lesser overhead**, since the parallel region is created (using the `fork-join`) only once, whereas in Code 6 the parallel region is created and terminated YM-times repeatedly, but optimizing compilers of Code 6 may recycle threads in *thread pool*.

Parallelization of nested loops III

- Code 7 has a bit lower overhead with the thread pool than Code 6 and so, it should run faster.
- Under the same conditions, it should be slower than Code 4 or 5.
- But! Code 6/7 **must be used** if the outer loop has **data dependent iterations**.
- A trivial and significant example is **Gauss elimination**.

Code 8: Inner loop parallelization.

```
int ge1(float **A, *y, int n)
{for (int i = 0; i < n; i++)
  {float d = A[i][j];
   #pragma omp parallel for
   for (int j = i+1; j < n; j++)
     {float p = A[j][i]/d;
      for (int k = 0; k < n; k++)
        A[j][k] -= p*A[i][k];
      y[j] -= p*y[i];
}}}
```

Code 9: Inner loop parallelization.

```
int ge2(float **A, *y, int n)
{#pragma omp parallel
 for (int i = 0; i < n; i++)
   {float d = A[i][j];
     #pragma omp for
     for (int j = i+1; j < n; j++)
       {float p = A[j][i]/d;
        for (int k = 0; k < n; k++)
          A[j][k] -= p*A[i][k];
        y[j] -= p*y[i];        }}}
```

Parallelization of nested loops IV

- In case of the Gauss elimination, the outer loop cannot be parallelized since OpenMP does not check the data dependencies of iterations and such a code could be translated and executed, but thanks to race conditions with reading and writing of elements of arrays a and y, it would not compute the correct result.
- Therefore, Codes 8 or 9 are correct.
- Similarly to the case of Codes 6 and 7, here also thanks to a lower overhead with the thread pool, Code 9 would run faster.
- OpenMP implementation creates in both cases a corresponding number of threads at the beginning of the inner loop parallel region and even in case of Code 8 these threads are not terminated at completing the inner iteration but they are just stored temporarily into the thread pool and withdrawn later on.
- In case of Code 9 threads remain continuously active and only at the end of the loop body they are synchronized on a hidden barrier (which exists in Code 8 as well).

Functional parallelism using the `task` directive

- The `task` directive supports a more complicated functional parallelism with a greater overhead.
- Thanks to the data and code encapsulation, it is suitable namely for parallel execution of **recursive algorithms** (Divide-and-Conquer).
- The mechanism to assign computation to parallel threads is the **producer-consumer**.
- A **task** is a unit of a parallel computation consisting of:
  - code to be executed,
  - input data,
  - data structure for storing the identity of the consumer thread once it will start to execute the task.
- The `#pragma omp task` directive makes the **producer** thread executing the **parent** task **to generate** a new **child** task and to store it into the **task pool**.
- The new child task will wait here until some free thread will not withdraw it and become a **consumer** thread that starts to execute it.

Conditional execution of parallel tasks

- The `if(condition)` clause serves for effective control of the `task` parallelization of recursive codes where recursive nesting depends on fulfill a given condition.
  - If this `condition` is fulfilled, then the producer thread puts the new child task into the task pool from where it is later withdrawn by another free consumer thread to be executed.
  - If this `condition` is NOT fulfilled, then the producer thread postpones the execution of the parent task and temporarily stores it onto the task pool and immediately starts to execute sequentially the rest of the code as a new child task. After its completion, it withdraws from the task pool its postponed parent task and completes it.
- The `#pragma omp taskwait` directive causes that the parent task must wait for completion of all its child tasks.

An example: parallel binary tree recursive computation

Code 10: Naive parallelization of binary tree recursive computation

```
int subtreesize(node *p) {
 int i,j;
 if (p == NULL) return 0;
 #pragma omp task
   i = subtreesize(p->left); //left child task created
 #pragma omp task
   j = subtreesize(p->right); //right child task created
 #pragma omp taskwait //waits for completion of both children
 return 1+i+j;
}
```

- This code is very inefficient since it generates for every tree node (including leaves) one task and it produces a big overhead whose time complexity exceeds the useful work time.
- If the `taskwait` directive were missing, the code would return nonsense.

Better parallel binary tree recursive computation

- If there is not enough computational work in the recursion step, the efficiency can be improved with the `if` clause that makes the recursion fork conditioned by, e.g., the depth of the recursion h and that allows to complete small parts of the recursive call tree sequentially.

Code 11: Better parallel binary tree recursive computation

```
int subtreesize2(node *p, int h) {
 int i,j;
 if (p == NULL) return 0;
 #pragma omp task if (h < THRESHOLD)
   i = subtreesize2(p->left, h+1); //left child task created
 #pragma omp task if (h < THRESHOLD)
   j = subtreesize2(p->right, h+1); //right child task created
 #pragma omp taskwait //waits for completion of both children
 return 1+i+j;
}
```

Parallel region and the `task` parallelism

- The `task` directive must appear **inside a parallel region**.
- Otherwise, the whole recursion tree would be executed sequentially by one thread!!!

Code 12: Task parallelism in a parallel region

```
void main() { int k;
#pragma omp parallel num_threads(4) {
  #pragma omp single
  k = subtreesize2(p);
} }
```

- The `single` directive ensures that the call `subtreesize2(p)` is executed only by a single thread out of 4 prepared ones.
- If there were no `single` directive, all 4 threads would call `subtreesize2(p)` and instead of 1 input recursion tree, they would execute 4 identical recursions and the computation would be slower(!!) than the sequential one due to the access conflicts to the shared memory and due to higher cache miss rates.

Basic OpenMP operations

- The `omp_get_num_procs()` operation: returns the number of CPU cores available for OpenMP.
- The `omp_get_thread_num()` operation: returns the index of this current thread. In a sequential part of the master thread, it returns 0.
- The `omp_get_num_threads()` operation: returns the number of threads in the current parallel region. In a sequential part of the master thread it returns 1.
- The `omp_set_num_threads(int i)` operation: it will **set up** the number of created threads in the following parallel regions on the value of parameter i. It must be called from the sequential part foregoing the parallel region. See more details on Slide 18. The set-up remains valid till the next call.
- The `omp_get_wtime()` operation: returns a number (of the `double` type) equal to the elapsed time from some (implementation dependent) moment in the past.

Synchronization mechanisms in OpenMP

- OpenMP defines all basic mechanisms for synchronization of thread accesses into the shared memory in parallel regions.
- The main synchronization directive:
  - `barrier`: location where parallel threads of the given parallel region must arrive and wait for all others.
  - `master`: Only the master tread is allowed to execute the given code.
  - `single`: Only a single arbitrary thread is allowed to execute the code.
  - `critical`: Critical region (typically to ensure mutually exclusive access to shared resources).
  - `atomic`: Given memory operation on given memory address (typically of Read-Modify-Write type) will be executed atomically, i.e., uninterruptedly by a single thread.
  - `flush()`: Write-through of the local instances of shared variables into the shared memory, but we will not discuss this.
  - `taskwait`: Synchronization of child tasks with the parent one, see Codes 10 a 11 on Slides 37 and 38.
  - In OpenMP, there are also **mutexes**, but we will not discuss them.

Barrier and serialization in a parallel region

- The `#pragma omp barrier` directive is a synchronization point in a parallel region where the arriving threads are turned to sleep until all threads arrive. Then all threads are awaken and proceed. Implicit at the end of every parallel region (see, e.g., Slide 39) and at the end of a `single` block.
- The `#pragma omp master` directive: The succeeding structured block of a parallel region may execute only the master thread 0. The remaining threads skip this block of code.
- The `#pragma omp single` directive: The succeeding structured block of a parallel region may be executed by only one single arbitrary (OS decides) thread. Also here, the remaining threads skip this block of code, but then they wait on an implicit barrier after that block.
- The key usage of the `single` directive is in Code 12 on Slide 39.

An example on the single, master, and barrier directives

Synchronization by serialization

```
#pragma omp parallel
{
 Phase1();
 #pragma omp single
  { printf("Thread No.%i.", omp_get_thread_num());
    printf(" opens Phase1");
    }
 Phase2();
 #pragma omp barrier
 #pragma omp master
  printf("Phase3 opens for all threads");
 Phase3();
}
```

Question: How would the semantics of the code change if the single, barrier, or master directive were removed?

Critical region

Definice 2

Critical region is a set of one or more pieces of code within a parallel region such that at any time, threads can execute it only in mutual exclusion, i.e., if a thread is executing any piece of the critical region code, no other thread can execute any piece of its code.

- The `#pragma omp critical` directive specifies an anonymous critical region.
- If multiple anonymous critical regions appear in the code, the mutual exclusion principle applies **globally** across all of them.
- The `#pragma omp critical name` directive specifies one piece of code of the critical region `name`.
- Again, the named critical region can appear at multiple places in the code and the mutual exclusion principle applies **globally** across all of them.

An example of a critical region

Code 13: Multiple critical regions

```
#pragma omp parallel
  { ....
      #pragma omp critical listing
      { printf("Very important ");
        printf("and long print");
    } }
    ....
    if (report_is_needed) {
      #pragma omp critical listing
      { printf("Another very important ");
        printf("and long print");
    } }
  }
```

Question: How would the printings change if they were not part of a `critical` region?

Parallel reduction implemented with critical regions

```
int sum = 0, n = some_big_number, a[n];
#pragma omp parallel
{int myID = omp_get_thread_num(); int localSum = 0;
 #pragma omp for
 for (int i = 0; i < n; i++)
   localSum += a[i];
 #pragma omp critical
  {sum += localSum;
   printf("ID = %d:localSum = %d sum = %d", myID, localSum, sum);
 } } //end of parallel region
printf("Value of global sum %d", sum);
```

- Compare with the implementation of the parallel reduction using the `reduction` properties in Code 1 on Slide 14.
- Here, parallel threads compete for the shared `sum` $\Rightarrow$ higher overhead.
- In both cases $T(n, p) = O(n/p) + O(p)$ (OpenMP does not guarantee $O(\log p)$).

The atomic directive and its usage I

- The `#pragma omp atomic` directive is important for accessing the shared memory.
- It ensures that the access to a specific shared memory location representing data of scalar data type (integer, floating-point, ...) will be atomic, i.e., uninterruptible R/W/RMW (Read-Modify-Write) operation without any interleaving of accesses of multiple threads.
- The `#pragma omp atomic` construct ensures that a specific storage location is accessed atomically, rather than exposing it to the possibility of multiple, simultaneous reading and writing threads that may result in indeterminate values.
- The `atomic` Directive can be applied to memory operations `read`, `write`, `update`, and `capture`.
- Typical usage is shown in Code 2 on Slide 17.
- On simple examples, let us show the semantics of combinations of directives `private`, `shared`, and `atomic`.

The `atomic` directive and its usage II

```
int i = 10;
#pragma omp parallel private(i) num_threads(3)
i = omp_get_thread_num(); //parallel region
```

- The value of the variable i after the parallel region will be 10 again, since the updates of the variable values inside the region will be forgotten.

```
int i = 10;
#pragma omp parallel shared(i) num_threads(3)
i = omp_get_thread_num(); //parallel region
```

- Valid code: the threads perform the write into the shared variable i in **some order**.
- The value of i after the parallel region can be anything from $\{0, 1, 2\}$ (if the write operation is atomic).

The `atomic` directive and its usage III

- It is absolutely crucial for preserving the consistency of the shared memory to ensure the atomicity of memory operations of the Read-Modify-Write type, e.g., increments of a shared counter (see again Code 2 on Slide 17).
- Nonatomic implementations lead to **data racing errors**. For example:

```
int i = 10;
#pragma omp parallel shared(i) num_threads(3)
i += 2;
```

- This is a valid code where the threads execute the increments of the shared variable i in **some order**, but individual increments as **sequences** of 3 operations Read-Modify-Write can **arbitrarily interleave**.
- The value in i after the region can be anything from $\{12, 14, 16\}$.
- The `flush` directive after the increment does not help (all threads can read 10 at once and whichever can perform the write as the last one).

The atomic directive and its usage IV

- A consistent result of concurrent executions of three operations Read-Modify-Write can be guaranteed just by the atomic directive.
- The Read-Modify-Write operation will become uninterruptible and no data racing errors can appear.

```
int i = 10;
#pragma omp parallel shared(i) num_threads(3)
{
  #pragma omp atomic update
  i += 2;
}
```

- After the parallel region, the value in i will **always** be 16 regardless of in which order the threads were allowed to gain access to the shared i.
- We can of course use the critical region or the mutex semaphores.

The atomic directive and its usage V

```
int a[10];
#pragma omp parallel num_threads(3)
 { int my_index = omp_get_thread_num() + 1; a[0] = a[my_index]; }
```

- 3 threads compete for **writing** into `a[0]`.
- The **write** operation is atomic at some architectures (e.g., x86-64), but in general, this does not hold.
- Therefore, in OpenMP there are atomic `read` and `write` operations.
- See the example below: Since `atomic` cannot be applied to two memory locations simultaneously, we need an auxiliary variable.

```
int a[10];
#pragma omp parallel num_threads(3)
{ int temp; int my_index = omp_get_thread_num() + 1;
  #pragma omp atomic read
  temp = a[my_index];
  #pragma omp atomic write
  a[0] = temp;            }
```

The atomic directive and its usage VI

- The capture operation extends the update operation with acquiring the value of the given variable before or after the Modify part.
- This operation is useful, e.g., in dynamic partitioning of shared data among threads.
- Example: dynamic assignment of disjoint segments of a shared array to threads so that each thread at any given time is assigned an exclusive array segment of the same size.

```
int *ptr = (int*) malloc(....); //pointer to a shared array
#pragma omp parallel shared(ptr) num_threads(3)
{ int *my_ptr;
  #pragma omp atomic capture
  { my_ptr = ptr; ptr += BLOCK_SIZE;}
}   //each thread has now an exclusive access into this segment
    //my_ptr, ..., my_ptr + BLOCK_SIZE - 1
```

- This is a way to implement schedule(dynamic,BLOCK_SIZE).

User controlled escape from a parallel region

- The `cancel` directive provides an easy and safe mechanism to leave prematurely a parallel region.
- The thread after executing `cancel` sends a signal to other threads to terminate the computation and moves to the final barrier to wait for the others.
- Other threads approaching the `cancel` operation do then the same.
- The remaining threads that passed already its `cancel` complete the execution of the parallel region in a standard way.

```
#pragma omp parallel
{ ................
  #pragma omp cancel construct [if (expr)]
  ................
}
```

where `construct`::=[`parallel`|`for`|`taskgroup`|`sections`].

An example of a user controlled escape from a parallel region

- The first thread for which eureka is satisfied, sends a signal to other threads to terminate the computations and jumps to the final barrier.
- The remaining threads leave the parallel region without any exception be executing the cancel operation.

```
#pragma omp parallel for private(eureka)
for (i = 0; i < n; i++)
 {  eureka = testing(i,...)
    #pragma omp cancel parallel if (eureka)
    ..... some other work to do if not heureka .....     }
```

- The next Slide shows how to use the cancel construct to finish efficiently a multithreaded binary tree search implemented using task parallelism.
- The thread that finds the searched value first cancels the search of all threads bound to the subtree search tasks generated so far.

```
typedef struct binary_tree_s { int value;
                               struct binary_tree_s *left, *right;
                             } binary_tree_t;
binary_tree_t *search_tree(binary_tree_t *tree, int value, int level) {
  binary_tree_t *found = NULL;
  if (tree) { if (tree->value == value) { found = tree;}
              else {
                    #pragma omp task shared(found) if(level < 10)
                    { binary_tree_t *found_left = NULL;
                      found_left = search_tree(tree->left, value, level + 1);
                      if (found_left) {
                        #pragma omp atomic write
                        found = found_left;
                        #pragma omp cancel taskgroup
                     }}
                     #pragma omp task shared(found) if(level < 10)
                    { binary_tree_t *found_right = NULL;
                      found_right = search_tree(tree->right, value, level + 1);
                      if (found_right) {
                        #pragma omp atomic write
                        found = found_right;
                        #pragma omp cancel taskgroup
                     }}
                     #pragma omp taskwait
                     }}
   return found;
}
binary_tree_t *search_tree_parallel(binary_tree_t *tree, int value) {
binary_tree_t *found = NULL;
#pragma omp parallel shared(found, tree, value)
#pragma omp master
#pragma omp taskgroup
      found = search_tree(tree, value, 0);
```

Further internet OpenMP references

- https://computing.llnl.gov/tutorials/openMP
- https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
- http://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf
- https://www.youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG