*#pragma omp barrier*: Barrier synch. Each thread wait at the barrier until all threads arrive. Note: There is an implicit barrier at the end of each parallel block, and at the end of each sections, for and single statement, unless the nowait directive is used.

The barrier directive causes threads encountering the barrier to wait until all the other threads in the same team have encountered the barrier.

**Barrier construct example**:

#pragma omp parallel

{

int id=omp\_get\_thread\_num();

A[id]=funct1(id);

#pragma omp barrier //no thread will execute funct2, before A[] is stored

B[id]= funct2(id, A);

}

***#pragma omp nowait***: The nowait directive can only be attached to sections, for and single. It cannot be attached to the within-loop ordered clause, for example.nowait overrides the barrier implicit in a directive. Nowait example:

#pragma omp for

for (int n=0; n<10; ++n)

funct1();

#pragma omp for nowait // funct2 will be executed in parallel with funct1.

for (int n=0; n<10; ++n)

funct2();

----------------

***Single construct - #pragma omp single :*** The single construct specifies that the given statement/block is executed by only one thread. It is unspecified which thread. Other threads skip the statement/block and wait at an implicit barrier at the end of the construct. Do not assume that the single block is executed by whichever thread gets there first. According to the standard, the decision of which thread executes the block is implementation-defined.

Example:

#pragma omp parallel {

funct1(); //all threads execute this

#pragma omp single

{

funct2(); //just one thread executes this

} //other threads wait here for the single thread to finish

funct3(); //all threads execute this

}

***Master construct - #pragma omp master :*** The master construct is similar to single, except that the statement/block is run by the *master* thread, and there is no implied barrier; other threads skip the construct without waiting. The following two examples are equivalent.

*#pragma omp parallel //example1*

*{ funct1(); #pragma omp master { funct2(); }*

funct3(); }

*#pragma omp parallel //example2, equivalent to example1 { funct1();*if(omp\_get\_thread\_num() == 0) { funct2(); } funct3(); }

***Loop nesting and collapse clause***: Two or more loops are merged to one and parallelized. Used when the number of iterations is small. Use the collapse-clause to increase the total number of iterations that will be partitioned across the available number of OMP threads. An example follows

*#pragma omp parallel for collapse(2)*

*for ( i=0; i<15; i++)*

*for( j=0; j<80; j++)*

*func(i,j);*

*Below an inappropriate example follows:*

*#pragma omp parallel for for ( i=0; i<15; i++)*

*#pragma omp for* ***// ERROR, nesting like this is not allowed.*** *for( j=0; j<80; j++) func(i,j);*

*Furthermore, i*f there are nested loops, we need to specify the innermost loops as private on our own.

**Changing the storage attributes**

* ***shared*** (a): all threads can access ‘a’
* ***private*** (a): each thread creates an un-initialized copy of ‘a’
* ***firstprivate*** (a): each thread creates an initialized copy of ’a’
* ***lastprivate*** (a) : the value of ‘a’, of the last iteration of the loop, is stored back as global. If a loop goes from i=[0,N-1], then the thread that executed the iteration N-1, its value of tmp will be copied out to the global scope.
* ***default (private | shared | none)***. This is not available in C. The default clause forces a programmer to explicitly specify the data-sharing attributes of all variables in a parallel region. Using this clause then forces the programmer to think about data-sharing attributes. This is beneficial because the code is clearer and has less bugs. Default(shared) clause makes all the variables shared. Another usage of default(shared) clause is to specify the data-sharing attributes of the majority of the variables and then additionally define the private variables, e.g., #pragma omp parallel for default(shared) private(a, b). You can also write parallel regions with the default(none) clause and then specify the private and shared ones. The default clause is to check whether you have remembered to consider all variables as private/shared, using the default(none) setting.

Two examples follow. The first example is wrong as private(tmp) does not initialize the variable, it just creates it. However, firstprivate() will initialize it.

*void wrong(){*

*int tmp=0;*

*#pragma omp parallel for* ***private****(tmp) //create a var tmp that is private (****un-initialized****)*

*for (i=0; i<N; i++)*

*tmp+=j;*

*printf(tmp); //problem, will see the global tmp, not the private. The private tmp is disappeared*

*}*

*Void good(){*

*tmp=0;*

*#pragma omp parallel for* ***firstprivate****(tmp) ////create a var tmp that is private (in****itialized****)*

*for (i=0; i<N; i++) {*

*if ((I%2)==0) tmp++;*

*A[i]=tmp;*

*}*

*//the private tmp is dissapeared here*

*}*

**Task1:** Consider the following code. Are a,b,c local to each thread or shared inside the parallel region?

*int a=1,b=1, c=1;*

*#pragma omp parallel private(b) firstprivate(c)*

*{ }*

Answer: a is shared, while b and c are local

**Task2:** What are the a,c,b initial values inside the parallel region and after the parallel region, in the code above?

Answer: inside the region a=1, b=undefined, c=1. After the region a=b=c=1. b and c revert to their original values.

***omp sections***: The section construct is one way to distribute different tasks to different threads. An example follows

*#pragma omp sections*

*{*

*{ funct1(); }*

*#pragma omp section*

*{ funct2();*

*funct3(); }*

*#pragma omp section*

*{ funct4(); }*

*}*

The code above indicates that any of the routines funct1, (funct2, funct3) and funct4 may run in parallel, but that funct2 and funct3 must be run in sequence. Each function is executed exactly once. As usual, if the compiler ignores the pragmas, the result is still a correctly running program.

**Task3:** study the ‘sections.c’ program. Unlike to the previous examples, where they were based on loop parallelism, this example exploits task parallelism; two different loop kernels are executed in parallel. Pay attention to the ‘nowait’ clause. This allows for all the threads but the master, not to wait in the section. If you cannot understand this, remove the ‘nowait’ clause and run the program several times to see what the output is.

**Task4:** Study the ‘*mandel\_serial.c*’ program. The ‘mandel\_parallel.c’ contains an OpenMP parallel implementation of the ‘*mandel\_serial.c*’ program, but it does not work properly. Can you spot the problem?

**Task5:** Study the ‘find\_the\_problem1.c’ program. This program is problematic. Can you spot the problem? Tip: think of the variables scope (private vs shared).

**Task6:** Study the ‘find\_the\_problem2.c’ program. This program prints a ‘Segmentation fault (core dumped)’ error message, which means that memory has been violated. Can you spot the problem?

**Answer**: Array A is private, which means that every thread will try to allocate an array of size NxN. The memory segment that is used, is the stack, not the heap, as A[][] is a private array; the size of the array is very large and the program cannot allocate such space on the threads’ stack. This makes the program to crash.

***Divide and Conquer algorithm:*** In computer science, divide and conquer is an algorithm design paradigm based on multi-branched recursion. A divide-and-conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem [<https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm#Parallelism> ].

***Omp task****:* When a thread encounters ‘*#pragma omp task*’, the thread may immediately execute the task, or defer its execution for later. If deferred, the task is placed in a conceptual pool of tasks associated with the current parallel region. All team threads will take tasks out of the pool and execute them until the pool is empty. A thread that executes a task might be different from the thread that originally encountered it. The code associated with a task construct will be executed only once. **Omp task is an advanced and new OpenMP feature and it will not be assessed.**

**Task7**: Try to implement a parallel version of the PI program using the divide and conquer approach. Use omp tasks. Before you start working on it, read the two paragraphs above as well as the lecture slides.

**ENVIROMENT VARIABLES**. The **OpenMP** specification defines several **environment variables** that control the execution of **OpenMP** programs.

OMP\_NUM\_THREADS : Sets the number of threads to use during execution of a parallel region. You can override this value by a **NUM\_THREADS** clause, or a call to **OMP\_SET\_NUM\_THREADS()**.

OMP\_STACKSIZE : Sets the stack size for each thread. The value is in kilobytes.

OMP\_WAIT\_POLICY : The OMP\_WAIT\_POLICY environment variable provides a hint to an OpenMP implementation about the desired behavior of waiting threads. It can be either ACTIVE or PASSIVE. In active, the thread actively spins waiting for something to be available. This consumes CPU power. In passive, the thread is put into sleep. Putting a thread into sleep and waiting it up, costs a lot. If you believe your program is not going to wait long, then use active.

OMP\_PROC\_BIND : It can be either true or false. It sets the thread affinity policy to be used for parallel regions at the corresponding nested level. If the environment variable is set to false, the execution environment may move a thread to another CPU core. If it is true, threads are not shuffled among the cores. Use true for cache intensive algorithms.

To learn more about environment variables visit <https://www.openmp.org/spec-html/5.0/openmpch6.html> .

**Example**: to set OMP\_PROC\_BIND=true in Linux, type the following commands:

*export OMP\_PROC\_BIND=TRUE*

*echo $OMP\_PROC\_BIND*

Further Reading

1. Guide into OpenMP: Easy multithreading programming for C++, available at <https://bisqwit.iki.fi/story/howto/openmp/#ParallelConstruct>
2. OpenMP Application Programming Interface Examples, available at <https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwiOip2R-rrqAhX8XRUIHa5HC0QQFjAAegQIAxAB&url=https%3A%2F%2Fwww.openmp.org%2Fwp-content%2Fuploads%2Fopenmp-examples-4.5.0.pdf&usg=AOvVaw3BDlLKC3VhdJI1iTj1RE_p>
3. GNU libgomp available at <https://gcc.gnu.org/onlinedocs/libgomp/index.html>