General description

OpenMP is a parallel programming technology for computers with shared memory. Standard 4.5 was adopted in November 2015.
One version of the program for parallel and sequential execution.
Any process consists of several threads that have a common address space, but different command sets and separate stacks.

`_OPENMP` macro is defined in the format yyyymm, where yyyy and mm are the year and month in which the supported OpenMP standard was adopted .

Conditional compilation:

```
#include<stdio.h>
int main()
{
#ifdef _OPENMP
   printf ("OpenMP is supported!\n");
#endif
}
```

Parallelization in OpenMP: inserting special directives into the program code, as well as calling auxiliary functions.
SPMD model (Single Program – Multiple Data) of parallel programming: the same code is used for all parallel threads.

The program starts from a sequential region: one thread works; upon entering the parallel region, several more threads are spawned, between which parts of the code are distributed.
When the parallel region completes, all threads except one (the master thread) terminate.
Any number of parallel and sequential regions. Parallel regions can be nested within each other.

In OpenMP, variables in parallel regions of a program are divided into two main classes:

- `shared` (shared; all threads see the same variable);
- `private` (local, private; each thread sees its own instance of this variable).

By default, all variables generated outside the parallel region remain common when entering it. The exception is for variables that are iteration counters in a loop. Variables created within a parallel region are local by default.

Parallel and sequential regions

OpenMP directives in C are specified by preprocessor instructions starting with `#pragma omp`.

```
#pragma omp directive-name [ option [, option ] ... ]
```

The object of most directives is a single statement or block that precedes the directive in the source code of the program.

To use the OpenMP runtime library functions, the omp.h header file must be included in the program (the omp_lib.h file or the omp_lib module for Fortran programs).

You need to set the number of threads executing parallel regions of the program by defining the value of the `OMP_NUM_THREADS` environment variable:

```
export OMP_NUM_THREADS=n
```

Functions for working with the system timer:

```
double omp_get_wtime(void);
```

Returns the astronomical time in seconds that has passed since some point in the past.

```
double omp _get_ wtick(void);
```

Returns the resolution of the timer in seconds to the calling thread.

Parallel and sequential regions:

```
#pragma omp parallel [ option [, option] ... ]
```

New `OMP_NUM_THREADS-1` threads are spawned, each thread receives its own unique number, and the spawning thread receives number 0 and becomes the master thread of the group. When leaving the parallel region, implicit synchronization is performed and all threads except the master one are destroyed.

- `if(condition)` – execution of a parallel region according to a condition;
- `num_threads(integer expression)` – explicitly specifying the number of threads in the parallel region;
- `default(shared|none)` – `shared`: variables in the parallel region that are not explicitly assigned a class will be assigned the class `shared`; `none`: all variables are assigned a class explicitly;
- `private(list)` – variables for which a local copy is generated in each thread; initial value is undefined;
- `firstprivate(list)` – variables for which a local copy is generated in each thread; local copies are initialized with the values of these variables in the master thread;
- `shared(list)` – shared variables in all threads;
- `copyin(list)` – variables declared as `threadprivate`, which, when entering a parallel region, are initialized with the values of the corresponding variables in the master thread;

- `reduction(operator:list)` – specifies an operator and a list of shared variables; for each variable, local copies are created in each thread; they are initialized according to the operator type (for additive – 0 or similar, for multiplicative – 1 or similar); after executing all statements in the parallel region, the specified operator is executed; operator for C language: `+` , `*` , `-` , `&` , `|` , `^` , `&&` , `||`.

```
#include <stdio.h>
int main( int argc, char *argv[])
{
  printf("Sequential region 1\n");
#pragma omp parallel
  {
    printf("Parallel region\n");
  }
  printf("Sequential region 2\n");
}
```

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  int count = 0;
#pragma omp parallel reduction(+: count)
  {
    count ++;
    printf( "Current value of count: %d\n", count);
  }
  printf("Number of threads: %d\n", count);
}
```

If a parallel region contains only one parallel loop, one `sections` construct , or one `workshare` construct, then you can use the shorthand notation: `parallel for`, `parallel sections`, or `parallel workshare`.

Auxiliary functions

The omp_set_num_threads(num) function allows you to set the number of threads for subsequent parallel regions (where this number is not explicitly set with the num_threads option).

```
void omp_set_num_threads ( int num);
```

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
  omp_set_num_threads(2);
#pragma omp parallel num_threads(3)
  {
    printf("Parallel region 1\n");
  }
#pragma omp parallel
  {
    printf("Parallel region 2\n");
  }
}
```

The omp_get_max_threads() function returns the maximum number of threads allowed to be used in the next parallel region.

```
int omp_get_max_threads(void);
```

The omp_get_num_procs() function returns the number of processors available for use by the user program at the time of the call. Note that the number of available processors may change dynamically.

```
int omp_get_num_procs(void);
```

Parallel regions can be *nested*; By default, a nested parallel region is executed by a single thread. This is controlled by setting the OMP_NESTED environment variable.

```
export OMP_NESTED=true
```

Use the function omp_set_nested(nested) to set the OMP_NESTED variable.

```
void omp_set_nested(int nested);
```

Use the omp_get_nested() function to obtain the value of the OMP_NESTED variable.

```
int omp_get_nested(void);
```

The omp_in_parallel() function returns 1 if it was called from the active parallel region of the program.

```
int omp_in_parallel(void);
```

```
void mode(void)
{
  if(omp_in_parallel())
    printf ("Parallel region\n");
  else
    printf("Sequential region\n");
}

int main( int argc, char *argv[])
{
  mode();
#pragma omp parallel
  mode();
}
```

Directives for executing code once

If some section of code must be executed only once in a parallel region, then it must be supplied with `single` directive:

```
#pragma omp single [ option [, option] ... ]
```

`private(list);`
`firstprivate(list);`
`copyprivate(list)` – new values of list variables will be available to all private variables of the same name (`private` and `firstprivate`) in the parallel region; this option cannot be used in conjunction with the `nowait` option; list variables must not be listed in the `private` and `firstprivate` options of this `single` directive;
`nowait` – after a section is executed, implicit barrier synchronization of threads occurs: their further execution occurs only when everyone has reached this point; if this is not necessary, the `nowait` option should be used which allows threads that reach the end of the section to continue execution without synchronization.

```
#pragma omp parallel
{
  printf("Message 1\n");
#pragma omp single nowait
  {
    printf("One thread\n");
  }
  printf("Message 2\n");
}
```

The `master` directive defines a section of code that will only be executed by the master thread. The remaining threads skip this section and continue working from the next statement. Does not involve implicit synchronization.

```
#pragma omp master
```

```
#pragma omp parallel private(n)
{
  n=1;
#pragma omp master
```

```
    n=2;
  printf ("n value: %d\n", n);
}
```

Thread numbering

All threads in a parallel region are numbered with consecutive integers from 0 to N–1, where N is the number of threads executing this region.

Calling `omp_get_thread_num()` allows a thread to get its unique number in the current parallel region.

```
int omp_get_thread_num(void);
```

Calling `omp_get_num_threads()` allows the thread to get the number of threads in the current parallel region.

```
int omp_get_num_threads (void);
```

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
  int n=1;
  printf("In sequential region (1): %d\n", n);
#pragma omp parallel private( n)
  {
    printf("Value n on thread (1): %d\n", n);
    /* Assign variable n the number of the current thread */
    n= omp_get_thread_num();
    printf("Value n on thread (2): %d\n", n);
  }
  printf ("In the sequential region (2): %d\n", n);
}
```

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
  int n=1;
  printf("In the sequential region (1): %d\n", n);
#pragma omp parallel firstprivate(n)
  {
    printf("Value n on thread (1): %d\n", n);
    /* Assign variable n the number of the current thread */
    n= omp_get_thread_num();
    printf("Value n on thread (2): %d\n", n);
  }
  printf("In the sequential region (2): %d\n", n);
}
```

Loop parallelization

If a loop operator is encountered in a parallel region, then, according to the general rule, it will be executed by all threads of the current group, that is, each thread will execute *all* iterations of this loop. You can use the `for` directive to distribute loop iterations among different threads .

```
#pragma omp for [option [ , option] ... ]
```

This directive refers to the block that follows this directive, which is the `for` statement.

- `private(list)`;
- `firstprivate(list)`;
- `lastprivate(list)` – the variables listed in the list are assigned the result from the last iteration of the loop;
- `reduction(operator:list)`;
- `schedule(type[, chunk])` – the option specifies how loop iterations are distributed between threads;
- `collapse(n)` – the option specifies that n consecutive closely nested loops are associated with this directive; for loops, a common iteration space is formed, which is divided between threads; if the `collapse` option is not specified, then the directive applies only to the one cycle immediately following it;
- `ordered` – an option indicating that `ordered` *directives* may occur in the loop; in this case, a block is defined inside the body of the loop, which must be executed in the order in which the iterations occur in the sequential loop;
- `nowait`.

The strict restrictions are imposed on the type of parallel loops. In particular, it is assumed that a correct program should not depend on the *order* of parallel loop iterations. You cannot use explicit exit by `break` or `goto` statements from a parallel loop. The iteration block size specified in the `schedule` option should not change within the loop.

The format of parallel loops can be represented as follows:

```
for ([integer type] i = loop_invariant;
  i {,=,<=,>=} loop_invariant;
  i {+,-}= loop_invariant)
```

The iterative variable of a distributed loop must be local, so if it is specified as a general variable, it is implicitly made local when entering the loop. After the loop has completed, the value of the loop's iterative variable is undefined unless it is specified in the `lastprivate` option.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
  int A[10], B[10], C[10], i, n;
  for (i = 0; i < 10; i++)
  {
    A[ i ] = i;
    B[ i ] = 2 * i;
    C[i] = 0;
  }
#pragma omp parallel shared(A, B, C) private(i, n)
  {
    n= omp_get_thread_num();
#pragma omp for
    for (i = 0; i < 10; i++)
    {
      C[i] = A[i] + B[i];
      printf("Thread %d added elements %d\n", n, i);
    }
  }
}
```

Schedule option

In the `schedule` option, the `type` parameter specifies the following type of iteration distribution:

`static` – block-cyclic distribution of loop iterations; block size is equal to `chunk`. The first block of `chunk` iterations is performed by the first thread (thread 0), the second block by the next one, etc. until the last thread; then the distribution starts again from the first thread. If the `chunk` value is not specified, then the entire set of iterations is divided into continuous chunks of approximately the same size (the exact method depends on the implementation), and the resulting chunks of iterations are distributed between threads.

`dynamic` – dynamic distribution of iterations with a fixed block size: first, each thread receives a `chunk` of iterations (`chunk=1` by default), the thread that finishes executing its portion of iterations receives the first free portion of the `chunk` of iterations. The threads receive new portions of iterations until all portions are exhausted. The last portion may contain fewer iterations than all the others.

`guided` – dynamic distribution of iterations, in which the chunk size is reduced from a certain initial value to a `chunk` value (`chunk=1` by default) in proportion to the number of undistributed iterations divided by the number of threads executing the loop. The size of the block initially allocated is implementation dependent. In some cases, such a distribution makes it possible to more accurately divide the work and balance the load of threads. The number of iterations in the last chunk may be less than the `chunk` value.

auto – the method of distributing iterations is chosen by the compiler and/or runtime system. The chunk parameter is not specified in this case.

runtime – the method of distributing iterations is selected while the program is running based on the value of the OMP_SCHEDULE environment variable. The chunk parameter is not specified.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  int i;
#pragma omp parallel private(i)
  {
#pragma omp for schedule(static, 2)
    for (i = 0; i < 10; i++)
    {
      printf("Thread %d has iterated %d\n",
        omp_get_thread_num( ), i);
      sleep(1);
    }
  }
}
```

The default value of the OMP_SCHEDULE variable is implementation dependent. If the variable is set incorrectly, then the behavior of the program when the runtime option is set is also implementation dependent.
In Linux, you can set value of the OMP_SCHEDULE variable by means of the following command of the bash shell:

```
export OMP_SCHEDULE="dynamic,1"
```

You can change the value of the OMP_SCHEDULE variable from the program by calling the omp_set_schedule function.

```
void omp_set_schedule(omp_sched_t type, int chunk);
```

Values of constants of type omp_sched_t are declared in the file omp.h. At least the following options are available:

```
typedef enum omp_sched_t
{
  omp_sched_static = 1,
  omp_sched_dynamic = 2,
  omp_sched_guided = 3,
  omp_sched_auto = 4
} omp_sched_t;
```

By calling the omp_get_schedule function, the user can find out the current value of the OMP_SCHEDULE variable.

```
void omp_get_schedule(omp_sched_t *type, int *chunk);
```

When parallelizing a loop, you need to make sure that the iterations of this loop do not have information dependencies (in this case, its iterations can be performed in any order), in particular, in parallel. The compiler does not check this. If you instruct the compiler to parallelize a loop containing dependencies, the result may be incorrect.

Working with sections and tasks

The `sections` directive defines a block with a set of independent *sections* of code, each of which is executed by its own thread.

```
#pragma omp sections [ option [, option] ... ]
```

- `private(list)`;
- `firstprivate(list)`;
- `lastprivate(list)` – variables are assigned the result from the last section;
- `reduction(operator:list)`;
- `nowait`.

The `section` directive specifies a section of code within the `sections` block to be executed by one thread.

```
#pragma omp section
```

Before the first piece of code in the `sections` block, the `section` directive is optional. It is not specified, which threads will be used to execute each section. If the number of threads is greater than the number of sections, then some of the threads will not get any section. If the number of threads is less than the number of sections, then some (or all) threads will get more than one section.

```
int n;
#pragma omp parallel private(n)
{
  n= omp_get_thread_num();
#pragma omp sections
  {
#pragma omp section
    printf("First section, process %d\n", n);
#pragma omp section
    printf("Second section, process %d\n", n);
  }
  printf("Parallel region, process %d\n", n);
}
```

```
int n = 0;
#pragma omp parallel
{
#pragma omp sections lastprivate(n)
```

```
  {
#pragma omp section
    n = 1;
#pragma omp section
    n = 2;
  }
  printf("Value of n on thread %d: %d\n",
      omp_get_thread_num(), n);
}
printf("Value of n at the end: %d\n", n)
```

The `task` directive is used to select a separate independent *task*.

```
#pragma omp task [ option [, option] ... ]
```

The current thread interprets the block of statements defined by the `task` directive as a *task*. A task can be executed immediately after creation, or it can be deferred and executed in parts. The size of such parts, as well as the order in which the parts of different deferred tasks are executed, are determined by the implementation.

- `if(condition)` – generation of a new task only if some condition is satisfied; if the condition is not satisfied, then the task will be executed immediately by the current thread;
- `untied` – the option means that the deferred the task can be executed by any thread of the parallel region; if this option is not specified, then the task can only be executed by the thread that created it;
- `default(shared|none)`;
- `private(list)`;
- `firstprivate(list)`;
- `shared(list)`.

The `taskwait` directive is used to guarantee completion of all running tasks at the point of its call.

```
#pragma omp taskwait
```

The thread that has executed this directive is suspended until all tasks previously launched by this thread are completed.

Thread synchronization

The most common synchronization method in OpenMP is a *barrier*. It is defined with the `barrier` directive.

```
#pragma omp barrier
```

When threads of the current parallel region have reached this directive, then they stop and wait until *all* threads reach this point in the program, after which they unblock and continue working. In addition, to unlock, all threads must complete all their tasks.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
```

```
{
#pragma omp parallel
  {
    printf( "Message 1\n");
    printf( "Message 2\n");
#pragma omp barrier
      printf ( "Message 3\n");
  }
}
```

The `ordered` directive defines a block within the body of a loop that must be executed in the order in which iterations occur in a sequential loop.

```
#pragma omp ordered
```

It refers to the innermost of the enclosing loops, and the `ordered` option must be specified in the `for` directive. The thread that executes the first iteration of the loop immediately performs the operations of this block. A thread executing any subsequent iteration must first wait for all threads executing the previous iterations to perform all the operations of the block. This directive can be used, for example, to organize output from parallel threads.

```
int i, n;
#pragma omp parallel private(i, n)
{
  n=omp_get_thread_num();
#pragma omp for ordered
  for (i = 0; i < 5; i++)
  {
    printf("Thread %d, iteration %d\n", n, i);
#pragma omp ordered
    {
      printf("ordered: Thread %d, iteration %d\n", n, i );
    }
  }
}
```

Using the `critical` directive, the *critical section* of the program is defined.

```
#pragma omp critical[(critical_section_name)]
```

At any given time, there can be no more than one thread in the critical section. All other threads that have executed the directive for the section with the given name will block until the entering thread finishes executing. As soon as the working thread comes out critical section, one of the blocked threads will enter it. If there were several waiting threads, then one of them is randomly selected, and the rest continue to wait.

All unnamed critical sections are conventionally associated with the same name. Critical sections with the same name are considered as a *single* critical section, even if they are located in different parallel regions. Side entries and exits from the critical section are prohibited.

```
int n;
#pragma omp parallel
{
#pragma omp critical
  {
    n = omp_get_thread_num();
    printf( "Thread %d\n", n);
  }
}
```

`#pragma omp atomic`

This directive applies to the assignment operator immediately following it and ensures correct work with the shared variable located on its left side. During the execution of the operator, access to this variable is blocked for all currently running threads, except for the thread performing the operator. Only work with a variable on the left side of the assignment operator is atomic, while calculations on the right side do not have to be atomic.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
  int count = 0;
#pragma omp parallel
  {
#pragma omp atomic
    count++;
  }
  printf("Number of threads: %d\n", count);
}
```

Special type of synchronization: locks

One of the synchronization options in OpenMP is implemented through the *locks* mechanism. Shared variables are used as locks. These variables should only be used as parameters of synchronization primitives.
The lock can be in one of three states: *uninitialized*, *unlocked*, or *locked*. The unlocked lock may be *set* by some thread. At the same time, it goes into a locked state. The thread has set the lock and only it can *unset* it, after which the lock returns to its unlocked state.

There are two types of locks: *simple locks* and *nestable locks*. A nestable lock can be set multiple times by a single thread before being released, while a simple lock can only be set once. For a nestable lock, the concept of *nesting count* is introduced. Initially it is set to zero; it increases by one with each subsequent setting, and it decreases by one with each unsetting. A nestable lock is considered unlocked if its nesting count is zero.

To initialize a single or nestable lock, use the omp_init_lock and omp_init_nest_lock functions, respectively.

```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

After performing the function, the lock is in the unlocked state. For a nestable lock, the nesting count is set to zero.

The omp_destroy_lock and omp_destroy_nest_lock functions are used to put a single or nestable lock into an uninitialized state.

```
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
```

The omp_set_lock and omp_set_nest_lock functions are used to set a lock.

```
void omp_set_lock(omp_lock_t *lock);
void omp _set_nest_lock(omp_nest_lock_t *lock);
```

The thread that calls this function waits until the lock is unset and then sets it. As s result, the lock is switched to a locked state. If a nestable lock is already set by a given thread, then the thread is not blocked, and the nesting count is increased by one.

To unset the lock, the omp_unset_lock and omp_unset_nest_lock functions are used.

```
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_lock_t *lock);
```

Calling this function unsets a simple lock if it was set by the calling thread. For nestable locks, calling this function reduces the nesting count by one, and if the nesting count becomes zero, the lock is unset. If, after unsetting, there are threads waiting to the set this lock, then the lock will be immediately set by one of the waiting threads.

```
omp_lock_t lock;
int n;
omp_init_lock(&lock);
#pragma omp parallel private(n)
{
  n = omp_get_thread_num();
  omp_set_lock(&lock);
  printf("The lock is set, %d\n", n);
  sleep(5);
  printf("The lock is unset, %d\n", n);
  omp_unset_lock(&lock);
}
omp_destroy_lock ( & lock );
```

For a non-blocking attempt to set a lock, the `omp_test_lock` and `omp_test_nest_lock` functions are provided.

```
int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_lock_t *lock);
```

This function tries to set the specified lock. If this succeeds, then the function returns 1 for a simple lock, and it returns the new nesting count for a nestable lock. If the lock could not be set, the zero is returned in both cases.

```
omp_lock_t lock;
int n;
omp_init_lock(&lock);
#pragma omp parallel private(n)
{
  n = omp_get_thread_num();
  while (!omp_test_lock(&lock))
  {
    printf("The lock cannot be set, %d\n", n);
    sleep(2);
  }
  printf("The lock is set, %d\n", n);
  sleep(5);
  printf("The lock is unset, %d\n", n);
  omp_unset_lock(&lock);
}
omp_destroy_lock(&lock);
```