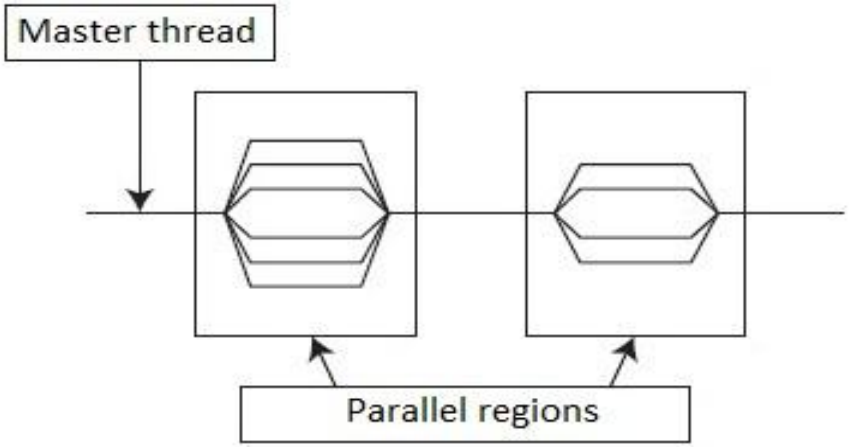
OpenMP: Open specifications for Multi-Processing

- What is OpenMP?
- Join\Fork model
- Variables
- Explicit parallelism
- High level parallelism
- Resume

What is OpenMP

- OpenMP (Open specifications for Multi-Processing) – one of the most popular parallel computing technologies for multi-processor/core computers with the shared memory architecture.
- OpenMP is based on traditional programming languages. OpenMP standard is developed for Fortran, C and C++ languages. All basic constructions for these languages are similar. Also, there are known cases of OpenMP implementation for MATLAB and MATHEMATICA.
- The OpenMP-based computer program contains a number of threads interacting via shared memory. OpenMP provides a number of special directives for a compiler, library functions and environment variables.
- Compiler directives are used for indicating segments of a code with the possibility for parallel processing.
- Utilizing *OpenMP* constructs *(compiler directives, procedures, environment variables),* a user can organize parallelism in their serial code*.*
- “Partial parallelization” is available by “step-by-step” adding OpenMP-directives. OpenMP-directives are ignored by standard compiler. So, the code stays workable on both single- and multi-processor platform.

OpenMP: Structure of the code



The program code consists of linear and parallel fragments. Only the master thread starts the execution of the code. Only Master thread executes all parts of the code.

Paralleizm is based on the **Fork\Join model:**

– On entry into the parallel section, a team of parallel threads is generated (*Fork*). After being generated, each of them gets its peculiar number; the master thread is always numbered as “0”. All threads execute the same code that corresponds to the parallel block.

– On exit from the parallel section, the master thread is waiting for the termination of other threads, and the further work is executed only by the master thread (*Join*).

– In C\C++, compiler directives beginning with **#pragma OMP** are used**.** All environment variables and functions referring to OpenMP begin with prefix **OMP_.**

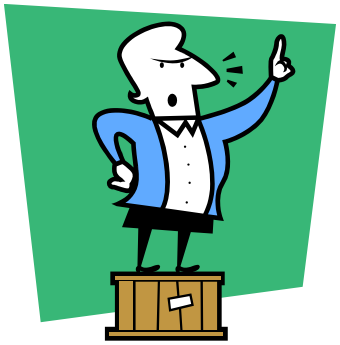
– In Fortran, all OpenMP directives are situated in comments and begin with one of the following combinations: !OMP, COMP or *$OMP (it is to be recalled that a line beginning with one of the symbols ‘!’, ‘C, or ‘*’is considered as a comment).

OpenMP: Classes of variables

- There are two main classes of variables: shared and private ones.
- **The shared variable** always exists only in a single instance for the whole program and is available for all threads under the same name.
- The declaration of **the private variable** causes generation of its own instance of the given variable for each thread. Change of a value of a thread's private variable does not influence the change of this local variable value in other threads.
- There are also "intermediate" types that provide interconnection between parallel and consistent sections.
- Thus, if a variable that is a part of a consistent code preceding a parallel section, is declared **Firstprivate**, then in the parallel section this value is assigned to private variables under the same name for each thread.
- Likewise, a variable of **Lastprivate** type after termination of a parallel block saves the value obtained in the latest completed parallel thread.

Compiler command: gcc –fopenmp text.c

OpenMP: Construct PARALLEL



Fortran:

```
!$OMP PARALLEL
< parallel block of the code>
!$OMP END PARALLEL
```

C\C++:

```
#pragma OMP parallel [declaration of variables]
{ <program's parallel block> }
```

- For the parallel block execution, the team of OMP_NUM_THREADS-1 threads is generated additionally to the master-thread. The number of threads is determined by the environment variable OMP_NUM_THREADS, or by means of special functions.
- Each thread has got a number from 0 to OMP_NUM_THREADS-1. A process (master thread) initiating generation of parallel section always gets number '0'.
- All threads execute the code in a parallel section. In general, there is no synchronization, and it is impossible to predict thread termination. At the end of a parallel block, implicit synchronization of all threads is carried out automatically; and as soon as all threads reach that point, master-thread continues execution of the following part of the program; other threads are destroyed.
- The necessity to generate a parallel region can be defined dynamically by means of an option IF in the directive. If the condition is not fulfilled, the parallel block is not activated and execution is continued in serial mode.

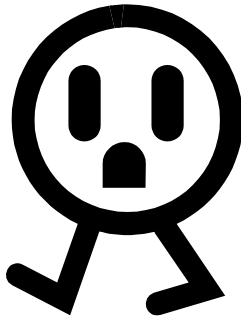
OpenMP functions. Explicit (low level) parallelism

- Function OMP_GET_THREAD_NUM returns number of the thread.
- Function OMP_GET_NUM_THREADS returns quantity of parallel threads in the team.
- Function OMP_SET_NUM_THREAD sets quantity of parallel threads in the team.

EXAMPLE. Explicit parallelism:
distribution of calculations in dependence on the thread number

```
#pragma omp parallel
{
myid = omp_get_thread_num ( );
If (myid == 0)
        do_something ( );
else
        do_something_else ( );
}
```

EXAMPLE:
Creation of parallel block



```
#include <stdio.h>
int main(int argc, char *argv[])
{
printf("Only master thread is working \n");
#pragma omp parallel
{   printf("Hello world from parallel block! \n");  }
printf("Again, only master thread is working \n");
}
```

Result of executing:

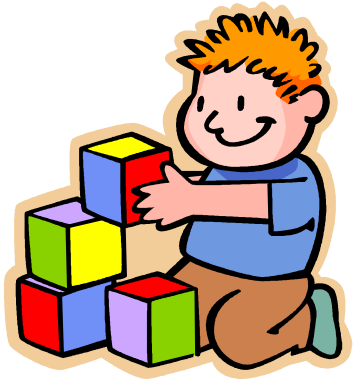
- Master-thread displays the text **Only master thread is working**
- Then pragma *parallel* generates new threads; each thread displays «**Hello world from parallel block!**»
- So this writing will be diaplayed OMP_NUM_THREADS times.
- Then the join occurs and only master-thread displays «**Again, only master thread is working**».

Example: quantity of threads
function omp_set_num_threads();
option num_threads

```
#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 block 1 \n");
}
#pragma omp parallel
{
printf("Parallel block 2 \n");
}
}
```

- Before 1st parallel section, function **omp_set_num_threads(2)** sets number of parallel threads **2**.
- **BUT!** Option **num_threads(3)**, inspires three parallel threads in the 1st parallel block. Hence writing **"Parallel block 1"** is displayed **3** times.
- No options in the 2nd parallel block. Therefore the setting of fuction **omp_set_num_threads(2)** is actual.
- So, writing **" Parallel block 2 "** will be displayed by **2** threads.

EXAMPLE:
option reduction



Calculation of quantity of threads in the parallel block.

```
#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);
}
```

- In parallel block, each thread inspires a local variable **count = 0**.
- Then, each thread increases its **count** by 1 and displays this value.
- Hence, writing "**current value of count:** 1" is displayed OMP_NUM_THREADS times.
- In exit of parallel section, all local variables **count** summarize.
- The resulting value is equal number of threads. It is written to **count** in the serial part of the code and is displayed only one time.

High level parallelism: directive SECTIONS

Low level parallelism: the work is distributed between threads by means functions OMP_GET_THREAD_NUM (returns number of thread) and OMP_GET_NUM_THREADS (returns quantity of parallel threads).

EXAMPLE:

```
if( OMP_GET_THREAD_NUM() ==3 )
< code for the thread number 3 >;
else
< code for all another threads >;
```

EXAMPLE of high level parallelism (parallel independent sections):

```
#pragma omp sections […[parameters…]]
{
#pragma omp section
    < block 1>
#pragma omp section
    < block 2>
}
```

In case two blocks are independent we can arrange parallel executing of sections.

Each of Block 1 and Block 2 in this example will be carried out by one of parallel treads .

High level parallelism: parallel loops

For distribution of iteration cycle between various threads it is necessary to use directives !$OMP DO (Fortran) and #pragma omp for (C\C++) which refer to the following loop.

EXAMPLE:

```
#pragma omp parallel shared (a,b) private (j)
{
#pragma omp for
for (j=0; j<N; j++)
a[j] = a[j]+b[j]
}
```

Automatically at the end of cycle, barrier synchronization that can be canceled by means of **nowait** option is carried out.

Example: pragma omp parallel for

```
#include <stdio.h>
#include <omp.h>
int main()
{
  int idx = 100;
  int main_var = 2120;
  #pragma omp parallel for private(idx)
  for (idx = 0; idx < 120; ++idx)
  {
    main_var = idx * idx;
    printf("In the thead number %d: idx = %d, main_var = %d\n",
      omp_get_thread_num(), idx, main_var);
  }
  printf("After the parallel loop finish: main_var = %d\n", main_var);
return 0;
}
```

Open MP: synchronization constructs (1)

OpenMP holds a wide class of directives (pragma) for synchronization of threads in parallel block

SINGLE: is used for a single execution of a part of the code if in a parallel section some part of the code must be executed only once.

#pragma omp single{ <single block> }

<Single block> of the code will be carried out (only once) by a thread that reaches a particular program point first.

BARRIER: **#pragma omp barrier**

All threads reaching that directive stop and wait till other threads reach that point; after that all threads continue the work.

MASTER indicates a part of the code that must be executed only by the master-thread.

#pragma omp master
{ <fragment for master-thread> }

Other threads ignore this fragment.

Open MP: synchronization constructs (2)

Critical section

#pragma omp critical [name]

{ <critical block> }

At any time a critical section may contain only one thread. If the critical section is being executed by a thread, all other threads that have been executing a directive for the section with this name, will be blocked till the indicated thread terminates the execution of the given critical section. As soon as the thread terminates the execution of a critical section, one of the blocked threads enters the section. If on entering the critical section there were several threads, one of them is chosen at random; other blocked threads continue waiting.

Atomic refers to the following operator. This directive is working like the above **critical** but critical block consists of only one operator.

Example.

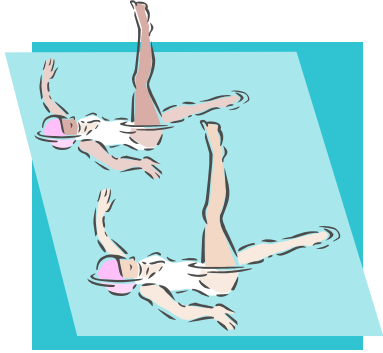
Each thread sequentially increases value of Expr.

```
#pragma omp atomic
        Expr++;.
```

Example: pragma omp critical

```
#include <stdio.h>
#include <omp.h>
int main()
{
  int sum = 0;
  int expr = 0;
  #pragma omp parallel num_threads(3) private(expr) shared(sum)
  {
    expr =omp_get_thread_num()+1;
  printf("in thread number %d expr = %d \n", omp_get_thread_num(),expr);
  #pragma omp critical
  {
  sum = sum+expr;
  }
  printf("after critical section\n");
  printf("In thead number %d: expr = %d, sum = %d\n",
      omp_get_thread_num(), expr, sum);
  }
  printf("After the parallel section closing: sum = %d\n", sum);
return 0;
}
```

Open MP: synchronization constructs (3)



FLUSH. Synchronization of this type is used for the update of private variables. The FLUSH directive identifies a synchronization point at which the implementation must provide a consistent view of memory. Thread-visible variables are written back to memory at this point.

Because of a complex structure and memory hierarchy in modern parallel computing systems, a user should have guarantees that at any required time each thread will detect united consisted memory image.

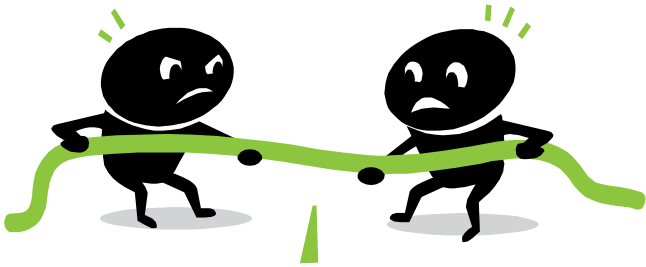
#pragma omp flush (var1, [var2,…])

Its execution presupposes that all variables' values temporally stored in registers, will be feed into the main memory; all changes of variables carried out by the threads during their operation, will be identified by other threads, if there is some information is stored in output buffers, the buffers will be reset and so on. Thus, after execution of a given directive, all variables in it have the same value for all parallel threads. Execution of the directive in full may cause serious cost overalls.

OpenMP: Concluding remarks

- OpenMP is designed in the way so that a user can work with a universal text for parallel and sequential programs because a standard compiler on a sequential machine ignores OpenMP-directives.
- Another advantage of OpenMP is the possibility of a gradual parallelization of a code. Basing on serial code, a user step by step adds new constructs that describe new parallel sections. There is no need to write a parallel program at once; parallel optimization is carried out gradually and that makes easier the process of coding and debugging.
- On the other side, it is important to take into account that after termination of parallel threads, the control is delegated to the master-thread. In this case there may appear a problem of correctness of data transfer from parallel threads to the master thread. In solving this problem, an synchronization of parallel threads plays important role. However, it can lead to a delaying and hence to a loss of efficiency.
- Synchronization and initialization operations of parallel threads are equivalent to the labor efficiency of execution of 1000 arithmetic operations. Therefore, in the process of selection of parallel sections, the labor efficiency should be not less than 2000 operations.
- It is important to avoid the use of shared variables without necessity and to monitor memory consistent in order to avoid computations' ambiguity (so called data race).

OpenMP: Example of data racing



Race condition.

The result in this fragment depends on the sequence of the parallel threads' execution. As soon as there is no synchronization, every time a different result is available. Note, the compiler cannot not provide with any diagnostic operation. Here, A,B,C are assumed to be shared variables.

```
#pragma omp sections
{
# pragma omp section
{…        C=A+B;
    …}
#pragma omp section
{…        B=C+A;
…}
#pragma omp section
{…        A=B+C;
…}
}
```