PROGRAMACIÓN PARALELA
Basic OpenMP Tutorial

In the accompanying archive, there is a compressed directory called `Lab4`. Once the archive is uncompressed change to that directory. Inside it there is a `Makefile`. Use `make` to compile the programs in that directory. You may also override the definition of the `PROGRAM` macro given in `Makefile` by typing a command line like `make PROGRAM=<prog>`. This will generate an executable image from the source code inside `prog.c`.

To compile an OpenMP program you will need to pass the compiler `-fopenmp`, which enables code generation using the OpenMP API.

The first program you should execute is called `omp0`. It simply illustrates the mechanics of and OpenMP program. First, the initial thread prints a message. Then, when the compiler encounters the `parallel` directive it will generate code to create a team of threads that will execute the parallel region. Typically, the number of threads to create corresponds to the number of logical cores available on the machine we will run the program minus one. All the created threads and the initial thread execute the parallel region. Once all the threads have finished the parallel region, the created threads are terminated, and the initial thread prints an exit message. What if we use `time` to measure execution times? The `user` and `system` time amount to the total time that all logical cores together spent executing the program. Therefore, the sum of the user and system time may be bigger than the real time, i.e., the elapsed or wall-clock time. Hence, in this case, various parts of the program must have run on several logical cores simultaneously. You can check the number of logical cores by entering the `lscpu` command.

The next program to test is called `omp1`. It illustrates how to each thread can determine its id as well as the number of threads in the job.

What if we want to use variables to hold the previous two values? The next program to test is called `omp2`. It illustrates how using threads may lead to data races.

The next program to test is called `omp3`. It makes use of the `private` clause to declare that `t_id`, `t`, and `p`, have a private scope. By using the `private` clause, each thread will have a copy of the variables inside the parentheses to the right of the `private` keyword.

Actually, in some cases we may not need to use the `private` clause. `omp4` shows how to declare per thread private variables without resorting to the `private` clause.

In all the previous cases, the value for the `t` variable is the same for every thread; therefore, we do not need a per thread copy of the variable. Then a single thread may get the number of threads and print it. `omp5` makes use of the `single` directive to print the common value.

However, there may be a problem with using `single`. The directive just specifies that a single thread (i.e., any thread) must execute the structured block beneath the directive. If we want the master thread to execute a particular code portion inside the parallel region, we need to use the `master` directive, as illustrated in `omp6`.

`omp7` shows another way to use the master thread to print the common variable.

`omp8` is quite similar to `omp7`, but now the barrier directive is used to synchronise all the threads before the master thread prints the common variable.

The set of `for?` programs illustrates using OpenMP parallelisation at the loop level.

In `for0`, iterations are divided among and executed by the team of threads running on all the logical cores: a number of threads is created, one per logical core or as specified explicitly (minus one that the initial thread runs on). Then,

- iterations of the `parallel for` loop are divided among threads where each iteration is executed by the thread it has been assigned to, and
- once all iterations have been executed, all threads in the team are synchronised at the implicit barrier at the end of the `parallel for` loop and all the created threads are terminated.

Notice that:

1. the program does not specify how the iterations should be divided among threads. In such cases, most OpenMP implementations divide the entire iteration space into chunks where each chunk containing a subinterval of all iterations is executed by one thread.
2. once the iteration space is divided into chunks, all iterations of an individual chunk are executed sequentially, one iteration after another.
3. the `parallel for` loop variable `i` is made private in each thread executing a chunk of iterations as each thread must have its own copy of `i`. On the other hand, variable `max` can be shared by all threads as it is set before and is only read within the parallel region.
4. the overall task of printing out all integers from `1` to `max` in no particular order can be divided into t totally independent subtasks of almost the same size.

`for1` shows how to add two vectors by using the `parallel for` directive.

`for2` shows how to initialise a matrix by using the `parallel for` directive.

`for3` shows how to initialise two vectors by using the `parallel sections` directive.

`races` illustrates the effect of data races.

`critical` shows how to use the `critical` directive.

`atomic` shows how to use the `atomic` directive.

`reduction` shows how to use the `reduction` clause.

We can set the `OMP_NUM_THREADS` environment variable to modify the number of logical cores by entering `export OMP_NUM_THREADS=<cores>`