# CITS3402 High Performance Computing
# Laboratory Sheet 3
# First OpenMP program

**Note:** If Setonix (the Pawsey supercomputer) is down, you can practice this in your own computer if you have a C compiler.

The aim of this lab is to develop our first OpenMP program and write some non-trivial OpenMP programs. We will use a few OpenMP constructs and functions that are in fact very powerful for parallelising most C programs. Although we will use further constructs in the future labs and also in the project. You can compile openmp programs by using the compiler flag gcc -fopenmp file.c

For the time being, we will live with hyperthreading if it is enabled in the machine you are working on. If you are working at home, search the internet to see how to turn off hyperthreading. You can also search for a command that will enable you to see the number of (physical and logical) cores in your machine. We will soon have a machine that you will be able to ssh into and run your code without hyperthreading. You can also look here.

Use the `#pragma omp` parallel and `omp_get_thread_num()` in this exercise. Write a multi-threaded program in which each thread will write its thread number and some meaningful message. Run the program several time. Do you see the same ordering of the print statements? The scheduling of the CPUs are actually non-deterministic if there are other processes in the system. Hence the ordering of the print statements should be random (although there are other factors too).

Now modify the program so that one or more threads sleep for some time. Use the `sleep()` call. This is a system call that makes a thread to sleep. How do you choose a particular thread to sleep? You can do this in an if statement using the `omp_get_thread_num()` function. What is the ordering of the print statements now?

Time your C program. We have learnt how to time a C code in the last lab. You can print the time as a floating point number, e.g.,

```
{ printf("time spent=%10.6f\n",time_spent);}
```

, this means 10 total characters, and 6 characters after the decimal, which is sufficient for us.

Most probably you will get all 0s if you time your C code, as our code is doing almost nothing. We have to make our code do some computation to get some non-zero time.

Now write a C program where you will change the number of threads used by `omp_set_num_threads(n)` directive. This directive should appear before a `#pragma omp parallel` directive and then the parallel region will be executed by n threads, where n is an integer. There is usually a limit how many threads you can launch from a process in a linux system. I have tested up to 512 threads in my machine, but we will work with smaller number of threads.

The way to parallelize a for loop is to use the `#pragma omp` for directive. So your program will look like:

```
int main()
{
.....
omp_set_num_threads(n);
#pragma omp parallel
{
#pragma omp for
{
for(...)
}
}
}
```

You have to of course include `omp.h` along with other include files. OpenMP divides the for loop into equal parts (except the last part) depending on the number of threads you are using. Each part behaves like an independent for loop and is executed by a separate thread.

Now write a program that adds floating point numbers stored in a large array. This is our first non-trivial OpenMP program. Each thread will use a local variable called localSum to store the sum that it computes in its part of the for loop. Hence, we have to make the variable localSum a private variable for each thread. This is done by changing the `#pragma omp parallel` to `#pragma omp parallel private(localSum)`. The rest of the code will be similar to what you would otherwise write for summing numbers in a loop. Use the localSum variable to store the sum of each thread. Remember, each thread has its own private copy of localSum variable now.

Use large arrays by increasing the array size and filling it up with random floating point numbers. You will eventually get a run time error (most probably a segmentation fault) as there is a limit how large an array you can allocate in a C program. Remember all such allocations are done in the stack (see the third lecture). It is possible to allocate larger arrays through dynamic allocation.

Now print the local sums. Write a separate program and check whether the sum of all the localSum variables is indeed equal to the sum of the elements of the array. Time your program. You should get some reasonable time (non-zero) for very large arrays. Increase the number of threads, and check the timing. Do you see any change in timing with increasing number of threads? Why?

Write a separate program where you call a sorting function from inside a for loop with a large number of iterations. You can make the sorting program that you wrote in the last lab as a function and call that function from inside the for loop. This is to introduce more work for the threads. Each iteration of the for loop will sort the same array, as we are just interested in making the threads work harder. Time this program and do a similar analysis for sorting larger and larger arrays many many times (increasing number of iterations of the for loop), and also with diferent number of threads.

**Amitava Datta**
**August 2024**