Notebook 4: asymmetric execution

Sometimes one is better than many

Gabriele Gaetano Fronzé

INFN Istituto Nazionale di Fisica Nucleare

VIRGO

Variable access modifiers

Until now we left OpenMP decide the access modifiers for our variables.

Several access modifiers are available in OpenMP:

- `Shared`: each variable defined in outer scopes is accessible to any thread at the same memory location. Race conditions can arise;
- `Private`: each thread has a private copy of the variable. References of the outer variable are replaced with references to the cloned variable. The variable can be uninitialized;
- `Firstprivate`: like private, but the value is initialized with the previous value of the cloned variable.

Variable access modifiers

This is useful to reduce multiple accesses to the same memory location by several threads, for example by creating initialized private copies of a constant.

Keep in mind that the default for OpenMP is shared.

Thinking question: how does the previously introduced pragmas work? What access modifiers do they use on the variables (`reduction`)?

Asymmetric execution

OpenMP parallel for construct is devoted to single instructions applied symmetrically to the data.

Two of the available constructs allow for a branching which limits a part of the code to be executed by a single thread:

- `#pragma omp single`: the first thread getting to the `single` section executes the section. Following threads skip the section;
- `#pragma omp master`: like a single section, but the thread **must** be the one with ID=0;

#pragma omp section(s)

OpenMP offers a way to differentiate even more the processed instructions.

Via the pragma sections, called inside a parallel scope, the thread pool can be divided in sub-pools.

Each sub-pool will handle one of the sections(s) independently.

Remember to use:

`omp_set_nested(1)`

To tell OpenMP you are aware of what you are doing!


```cpp
#include <omp.h>
#include <iostream>
#include <thread>
#include <chrono>

using namespace std::chrono;

int main(){

	const long maxIteration = 100000;

	omp_set_nested(1);

	int outputCounter = 0;
	bool done = false;

#pragma omp parallel sections shared(outputCounter) num_threads(2)
	{
		#pragma omp section
		{
			while(!done){
				printf("%d/%d\r",outputCounter,maxIteration);
			}
			printf("%d/%d\n",outputCounter,maxIteration);
		}

		#pragma omp section
		{
			#pragma omp parallel for
			for( int i=0; i<maxIteration; i++){
				std::this_thread::sleep_for(microseconds(100));
				#pragma omp critical
				{
					outputCounter++;
				}
			}
			done = true;
		}
	}

	return 0;
}
```

Once again… Make the code rain!

Upgrade your solutions creating an asynchronous status monitor
and optimizing variables access and initialization