Notebook 3: race conditions

It was too easy, isn't it?

Gabriele Gaetano Fronzé

INFN
Istituto Nazionale di Fisica Nucleare
UNIVERSITA' DEGLI STUDI DI TORINO
ELEMENTO MODULAR CLOUD

First results

One simple question:

If the previous exercises executable is run several times, is pi value always the same? If not, why?

Race condition!

Every thread instanced by OpenMP is writing to the same variable:

`double sum`

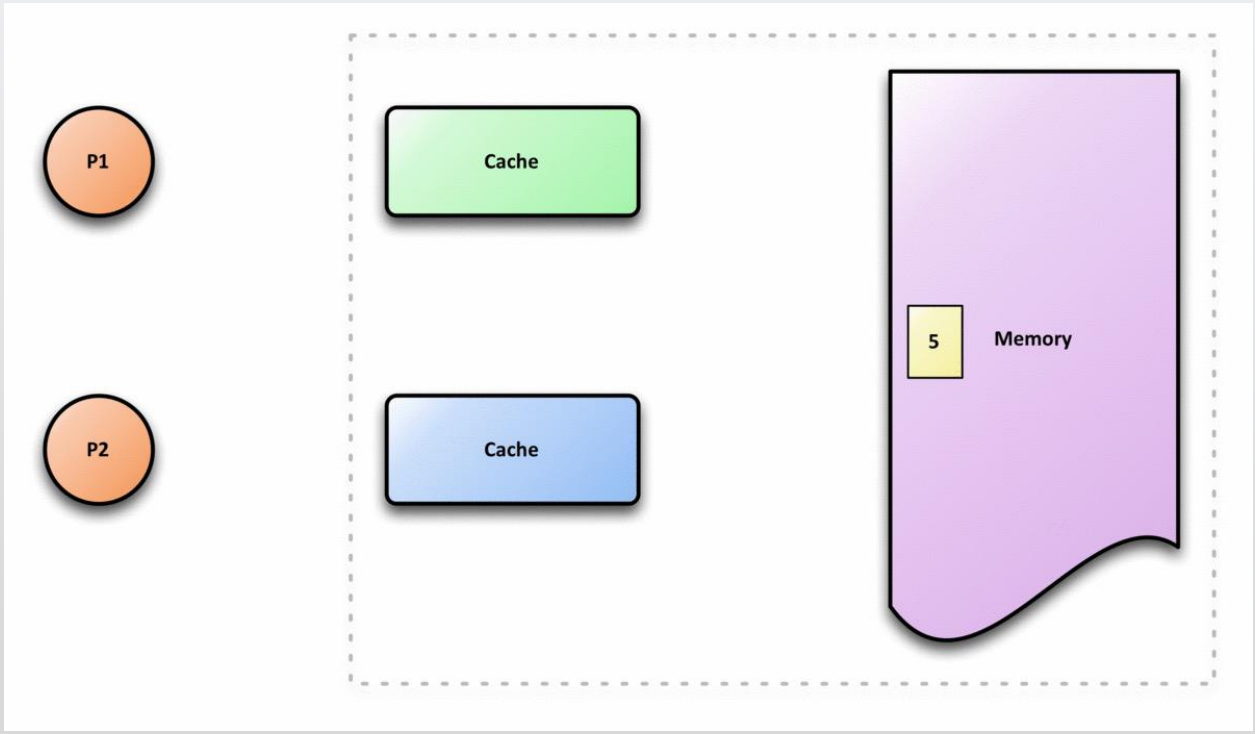
In order to add the new result to the variable, sum is read by the updating thread, stored, added to the local result and wrote back in memory.

While the sum value is stored locally, the common value can be changed by other threads.

The following update shadows previous updates!


```
#include "StopWatch.h"
#include <omp.h>
#include <iostream>

const long num_steps = 500000000; //number of x bins

int main()
{
	StopWatch stopWatch;

	double x, pi, sum = 0.0;
	step = 1.0/(double) num_steps; //x-step
	int n_threads=1;

	#pragma omp parallel
	{
		n_threads = omp_get_num_threads();

		#pragma omp for
		for (long i=1; i<=num_steps; i++) {
			x = (i - 0.5) * step; //computing the x value
			sum += 4.0 / (1.0 + x * x); //adding to the cumulus
		}
	}

	pi = step * sum;

	printf("Pi value: %f\n
	Number of steps: %d\n
	Number of threads: %d\n",
	pi,num_steps,n_threads;
	return 0;
}
```

Race condition!




```
#include "StopWatch.h"
#include <omp.h>
#include <iostream>

const long num_steps = 500000000; //number of x bins

int main()
{
	StopWatch stopWatch;

	double x, pi, sum = 0.0;
	step = 1.0/(double) num_steps; //x-step
	int n_threads=1;

	#pragma omp parallel
	{
		n_threads = omp_get_num_threads();

		#pragma omp for
		for (long i=1; i<=num_steps; i++) {
			x = (i - 0.5) * step; //computing the x value
			sum += 4.0 / (1.0 + x * x); //adding to the cumulus
		}
	}

	pi = step * sum;

	printf("Pi value: %f\n
	Number of steps: %d\n
	Number of threads: %d\n",
	pi,num_steps,n_threads;
	return 0;
}
```

Race condition!

A race condition can be cured in several ways:

- Using per-thread private copies of the `sum` variable, organised in an array, in order to perform the global sum out of the `parallel` section;
- Making the sum variable `atomic` via `#pragma omp atomic` to avoid interrupts inside updates of the variable;
- Using `#pragma omp critical` to allow only one thread at the same time in the same scope;
- More?

```
#include "StopWatch.h"
#include <omp.h>
#include <iostream>

const long num_steps = 500000000; //number of x bins

int main()
{
	StopWatch stopWatch;

	double x, pi, sum = 0.0;
	step = 1.0/(double) num_steps; //x-step
	int n_threads=1;

	#pragma omp parallel
	{
		n_threads = omp_get_num_threads();

		#pragma omp for
		for (long i=1; i<=num_steps; i++) {
			x = (i - 0.5) * step; //computing the x value
			sum += 4.0 / (1.0 + x * x); //adding to the cumulus
		}
	}

	pi = step * sum;

	printf("Pi value: %f\n
	Number of steps: %d\n
	Number of threads: %d\n",
	pi,num_steps,n_threads;
	return 0;
}
```

Solution 0: private copies with manual sum

Each thread has its own accumulation variable, which has been instanced outside the parallel section.

No race condition rises, since each thread loads and store the updated values in a private memory address.

At the end of the parallel section a sequential sum of all the private copies has to be performed.

Bottom line: the added sequential sum at the end can be awful for the Amdahl's Law!

Solution 1: `#pragma omp critical`

This pragma is similar to a singleton.

A variable accessible by all threads is set to `true` whenever a thread enters the critical section. The variable is reset to `false` when a thread exits the section.

Another thread willing to enter the critical section must wait the control variable to be set to `false` before proceeding.

Bottom line: It is simple, but causes a lot of sequential execution!

Solution 2: `#pragma omp atomic`

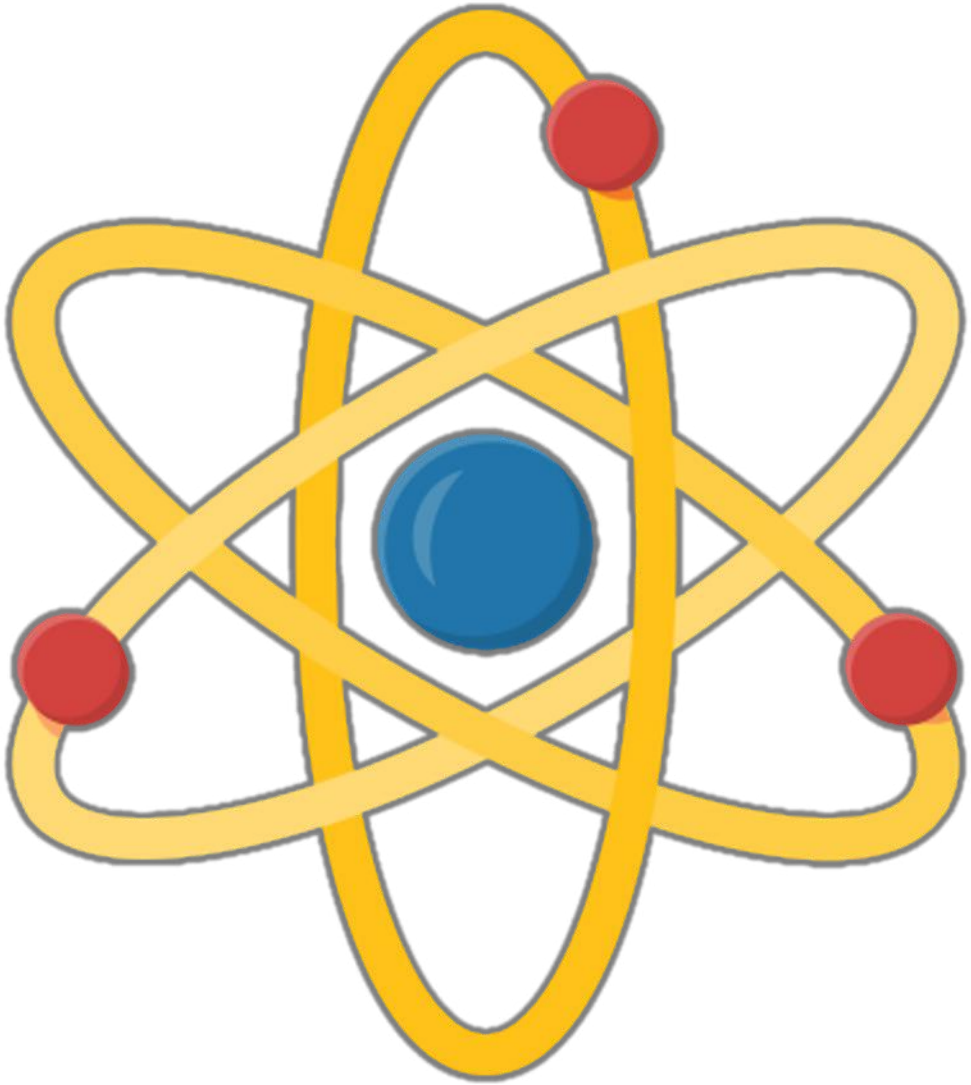
This pragma is more specific than a critical section.

Surrounding an operation with `#pragma omp atomic` causes that operation to become atomic. This pragma needs a specification to "understand" what kind of operation it is supposed to protect:

```
read, write, update, capture
```

An atomic operation is executed by a thread without interrupts between the needed uops.

Bottom line: less general approach, much less overhead wrt. `#pragma omp critical`



Solution 3: `#pragma omp reduction`

OpenMP provides a programmatic way to create private copies of the shared reduction variable.

The `reduction` keyword inserted after an `omp for`, followed by the specification of the reduction variable and the kind of operation to perform, protects the accumulation variable from any kind of race condition, offloading the task to the compiler.

For example, the constructs:

```
#pragma omp for reduction(+:sum)
```

Can solve the race condition rising from previous exercises.

It's your turn to... Make the code rain!

Try to solve the race conditions using the provided options