Parallel and High Performance Computing

Dr. Pablo Antolín

Series 3

March 13 2025

Debugging, profiling, and thread level parallelism with OpenMP

This week the exercise serie consists of 3 distinct parts: debugging, profiling a serial code, and thread level parallelism with OpenMP.

1 Debugging

In this part we will see how to sanitize a code for the most commonly found bugs: buffer overflows.

Exercise 1: *Write overflow*

- In the `debugging` folder, execute `make` to build the executables.
- Execute `./write`
- Run the code with `gdb`:

```
$> gdb ./write
```

- Once inside `gdb`, run the code with `run`, as:

```
(gdb) run
```

  It should stop at the line where the `segfault` happens.
- Check at which line the `segfault` occurred and study the code `write.cc`
- You can print the value of the variables with `print`. For instance,

```
(gdb) print i
(gdb) print data
```

- Fix the bug and check that the code works properly now.
- Describe the bug.

Exercise 2: *Read overflow*

- Execute the `./read` executable.
- It might run fine but there is a bug.
- Run the code with valgrind:

```
$> valgrind ./read
```

- Check at which line the error occurred and study the code `read.cc`
- Describe the bug.
- You can also compile with special sanitize options:

```
$> module load gcc
$> make clean
$> CXXFLAGS='-fsanitize=address' make
```

Execute again the buggy version of `./read` (and `./write`)

In this case the bounds check is always done at execution time.

Note: The sanitizers are relatively new. They were initially implemented in `clang` *and ported to* `gcc`*. Therefore, you should use a relatively new version of the compilers. On EPFL's cluster it means loading the module for* `gcc` *(it is not implemented in* `Intel` *compilers). At execution it will try to allocate a significant amount of memory. On some system's this behavior is blocked and will make the sanitizer fail.*

2 Profiling 101

In this part we will use the code in the `poisson` folder. This code will be used in multiple series.

Exercise 3: *Sequential profiling with gprof*

`gprof` is a profiling tool based on function calls counting. Therefore, if there are no functions you will see nothing. And you cannot get a grain finner than function calls.

- To use `gprof` you have to add the `-pg` option at compilation. Due the fact this is a C++ code it is highly recommended to at least have one level of optimization.

```
$> CXXFLAGS='-pg -g -O1' make
```

- Run the code using a batch script, for instance:

```
#!/bin/bash -l
#SBATCH --qos=math-454
#SBATCH --account=math-454

./poisson
```

It should generate automatically a `gmon.out` file in addition to the normal output of the code.

- This file can be visualized with `gprof`:

```
$> gprof ./poisson
```

- Where is most of the time spent? What can you do about it?
- Remove this “unnecessary” call to this expensive function at every iteration.

Exercise 4: ***Sequential profiling with perf***

`perf` uses hardware counters present in the CPU. It does a statistical analysis of the execution, hence results may vary between runs.

- Make sure you commented the calls to the `dump` function.
- Change the size of the problem to `1024` (modify the line `#define N 256`).
- To do some profiling with `perf`, recompile your code with the following options:

```
$> make clean
$> CXXFLAGS='-g -pg -O2' make
```

You can also compile with more optimization `-O3` but the results will be a bit harder to interpret.

- Run you code adding the `perf` tool in front of the executable, as

```
#!/bin/bash -l
#SBATCH --qos=math-454
#SBATCH --account=math-454

perf record -o perf.data --call-graph dwarf ./poisson
```

This will ask `perf` to record the collected statistics for the call graph of the run.

- You can visualize the results in an interactive way using:

```
$> perf report -g
```

This will report in the form of a call graph the results written in the `perf.data` file. Try to navigate the results and inspect the instructions that are more time consuming.

- `perf` can also give you some stats. Run it from a script as:

```
#!/bin/bash -l
#SBATCH --qos=math-454
#SBATCH --account=math-454

perf stat ./poisson
```

- Copy the poisson executable to `./poisson_ji`

- In the compute step function, permute the order of the loops on i and j. Recompile and copy the executable to `./poisson_ij`
- Compare the `perf` stats of both executables.
- To get a better understanding on what is the difference between the two runs, re-run them with:

```
#!/bin/bash -l
#SBATCH --qos=math-454
#SBATCH --account=math-454

perf stat -e L1-dcache-loads,L1-dcache-load-misses ./poisson_ij
perf stat -e L1-dcache-loads,L1-dcache-load-misses ./poisson_ji
```

- What do you notice?

3 Thread Level Parallelism: OpenMP

In this part we will start with a simple code that computes an approximation of π using the relationship:

$$\frac{\pi}{4} = \int_0^1 \frac{1}{1+x^2}\,\mathrm{d}x\,.$$

Exercise 5: ***OpenMP: hello world***

- In the file `pi.cc` add a function call to get the number of threads. Both `omp_get_num_threads` and `omp_get_max_threads` are good candidates, but they do not do the same thing. Check OpenMP's documentation.
- Compile using the proper options for OpenMP.
- Test that it works by varying the number of threads `export OMP_NUM_THREADS`
- To vary the number of threads in a sbatch job you can set the number of threads to the number of cpus per task

```
#!/bin/bash
#SBATCH --qos=math-454
#SBATCH --account=math-454
#SBATCH -c <nthreads>

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
<my_openmp_executable>
```

Exercise 6: ***Parallelize the loop***

The way this exercise ask you to parallelize the code is willingly not correct. The purpose is to make you realize the effect of a race condition.

- Add a parallel for work sharing construct around the integral computation, with `default` for all variables context.
- Run the code
- Run the code
- Run the code
- You should observe different results due to a race condition.

Exercise 7: ***Naïve reduction***

Here again, it is not the optimal way to do it. But it is often the first idea you might have to solve the problem of the previous exercise. Actually, in bigger codes, you might do it without being so clear that the placement of `critical` *region can impact performances.*

- To solve the race condition from the previous exercise we can protect the computation of the `sum`. Add a `critical` directive to protect it.
- Run the code.
- What can you observe on the execution time while varying the number of threads?

Exercise 8: ***Naïve reduction ++***

This solution is perfectly good in terms of result and performance. It will match the results of Exercise 9, so it might provide us some information of how `reduction` is actually implemented.

- Create a local variable `partial_sum` per thread.
- Make each thread compute it's own `partial_sum` (`private`).
- After the computation of the integral use a `critical` directive to add the `partial_sum` to the shared `sum`.

Exercise 9: ***Reduction***

- Use the `reduction` clause.
- Compare the timings with the previous versions.

Exercise 10: ***Poisson***

- Now you can apply what you learned to the `poisson` code.
- Remember that 90% of the time is spend in the dumpers. So modify this behavior to dump only at the end of the simulation to get a validation image.