EPFL

Parallel and High Performance Computing

Dr. Pablo Antolín

Solution Series 3

March 13 2025

Debugging, profiling, and thread level parallelism with OpenMP

1 Debugging

Exercise 1: ***Write overflow***

- The execution ends abnormally in a segmentation fault.
- In `gdb`, when you hit `run`, the debugger will intercept the error and let you inspect the variables. In this case, the error occurs on line 9, in the `main` function.
- If you print the value of `i` you should get `i = 0`. If you print `data` you will see that the pointer of `data` is `0x0` (a null pointer), and that `data` has size 0, i.e., `data` was not allocated.
- At line 6 you should specify a size for the vector: `data(N)`.

Exercise 2: ***Read overflow***

- This code runs fine and most of the times the result is correct.
- However, if you run with `valgrind`, you will get an output that looks like this:

```
==14921== Memcheck, a memory error detector
==14921== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==14921== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==14921== Command: ./read
==14921==
==14921== Invalid read of size 8
==14921==    at 0x400ABA: main (read.cc:14)
==14921==  Address 0x5ab4bc0 is 0 bytes after a block of size 8,000 alloc'd
==14921==    at 0x4C2A1E3: operator new(unsigned long)
↪  (vg_replace_malloc.c:334)
==14921==    by 0x400A41: allocate (new_allocator.h:111)
==14921==    by 0x400A41: allocate (alloc_traits.h:436)
```

```
==14921==    by 0x400A41: _M_allocate (stl_vector.h:296)
==14921==    by 0x400A41: _M_create_storage (stl_vector.h:311)
==14921==    by 0x400A41: _Vector_base (stl_vector.h:260)
==14921==    by 0x400A41: vector (stl_vector.h:416)
==14921==    by 0x400A41: main (read.cc:6)
==14921==
499500 == 499500
==14921==
==14921== HEAP SUMMARY:
==14921==     in use at exit: 0 bytes in 0 blocks
==14921==   total heap usage: 2 allocs, 2 frees, 80,704 bytes allocated
==14921==
==14921== All heap blocks were freed -- no leaks are possible
==14921==
==14921== For counts of detected and suppressed errors, rerun with: -v
==14921== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

- The way to read this is that there is and invalid operation at line 14. This is a read of size 8 bit just after an array of size 8000 bit. The last line tells you where the vector was allocated in our code. In this case, line 6.

  Line 6 contains `std::vector<double> data(N);` which is correct (`N = 1000` and `double` are 8 bit, so 8000 bit in total). This means it is the access that is wrong.

  If you look at the loop limits, it is $i \in [0, N]$, and should be $i \in [0, N[$, such that $i < N$.

- The information output for `./read` by the sanitizer is the same, but its presentation is different:

```
=================================================================
==1293390==ERROR: AddressSanitizer: heap-buffer-overflow on address
↪  0x625000002040 at pc 0x0000004011a1 bp 0x7ffec59f85f0 sp 0x7ffec59f85e8
READ of size 8 at 0x625000002040 thread T0
    #0 0x4011a0 in main
    ↪  /home/antolin/Teaching/exercises-2023/lecture_03/debugging/read⌋
    ↪  .cc:14
    #1 0x7fba6b2b2492 in __libc_start_main (/lib64/libc.so.6+0x23492)
    #2 0x400e0d in _start
    ↪  (/home/antolin/Teaching/exercises-2023/lecture_03/debugging/read+0x400e0d)

0x625000002040 is located 0 bytes to the right of 8000-byte region
↪  [0x625000000100,0x625000002040)
allocated by thread T0 here:
    #0 0x7fba6c0b7a77 in operator new(unsigned long)
    ↪  /tmp/scitasbuild/syrah.v1/tmp/spack-stage-gcc-11.3⌋
    ↪  .0-yrewh277xaqlocj7zcuxnv2m4fq4p3tw/spack-src/libsanitizer/asan/asan_new_delete⌋
    ↪  .cpp:99
```

```
    #1 0x400fb3 in __gnu_cxx::new_allocator<double>::allocate(unsigned long,
     ↪  void const*)
     ↪  /ssoft/spack/syrah/v1/opt/spack/linux-rhel8-x86_64_v2/gcc-8.5」
     ↪  .0/gcc-11.3」
     ↪  .0-yrewh277xaqlocj7zcuxnv2m4fq4p3tw/lib/gcc/x86_64-pc-linux-gnu/11」
     ↪  .3.0/../../../../include/c++/11.3.0/ext/new_allocator.h:127
    #2 0x400fb3 in std::allocator_traits<std::allocator<double>
     ↪  >::allocate(std::allocator<double>&, unsigned long)
     ↪  /ssoft/spack/syrah/v1/opt/spack/linux-rhel8-x86_64_v2/gcc-8.5」
     ↪  .0/gcc-11.3」
     ↪  .0-yrewh277xaqlocj7zcuxnv2m4fq4p3tw/lib/gcc/x86_64-pc-linux-gnu/11」
     ↪  .3.0/../../../../include/c++/11.3.0/bits/alloc_traits.h:464
    #3 0x400fb3 in std::_Vector_base<double, std::allocator<double>
     ↪  >::_M_allocate(unsigned long)
     ↪  /ssoft/spack/syrah/v1/opt/spack/linux-rhel8-x86_64_v2/gcc-8.5」
     ↪  .0/gcc-11.3」
     ↪  .0-yrewh277xaqlocj7zcuxnv2m4fq4p3tw/lib/gcc/x86_64-pc-linux-gnu/11」
     ↪  .3.0/../../../../include/c++/11.3.0/bits/stl_vector.h:346
    #4 0x400fb3 in std::_Vector_base<double, std::allocator<double>
     ↪  >::_M_create_storage(unsigned long)
     ↪  /ssoft/spack/syrah/v1/opt/spack/linux-rhel8-x86_64_v2/gcc-8.5」
     ↪  .0/gcc-11.3」
     ↪  .0-yrewh277xaqlocj7zcuxnv2m4fq4p3tw/lib/gcc/x86_64-pc-linux-gnu/11」
     ↪  .3.0/../../../../include/c++/11.3.0/bits/stl_vector.h:361
    #5 0x400fb3 in std::_Vector_base<double, std::allocator<double>
     ↪  >::_Vector_base(unsigned long, std::allocator<double> const&)
     ↪  /ssoft/spack/syrah/v1/opt/spack/linux-rhel8-x86_64_v2/gcc-8.5」
     ↪  .0/gcc-11.3」
     ↪  .0-yrewh277xaqlocj7zcuxnv2m4fq4p3tw/lib/gcc/x86_64-pc-linux-gnu/11」
     ↪  .3.0/../../../../include/c++/11.3.0/bits/stl_vector.h:305
    #6 0x400fb3 in std::vector<double, std::allocator<double>
     ↪  >::vector(unsigned long, std::allocator<double> const&)
     ↪  /ssoft/spack/syrah/v1/opt/spack/linux-rhel8-x86_64_v2/gcc-8.5」
     ↪  .0/gcc-11.3」
     ↪  .0-yrewh277xaqlocj7zcuxnv2m4fq4p3tw/lib/gcc/x86_64-pc-linux-gnu/11」
     ↪  .3.0/../../../../include/c++/11.3.0/bits/stl_vector.h:511
    #7 0x400fb3 in main
     ↪  /home/antolin/Teaching/exercises-2023/lecture_03/debugging/read.cc:6
    #8 0x401387
     ↪  (/home/antolin/Teaching/exercises-2023/lecture_03/debugging/read+0x401387)

SUMMARY: AddressSanitizer: heap-buffer-overflow
 ↪  /home/antolin/Teaching/exercises-2023/lecture_03/debugging/read.cc:14 in
 ↪  main
Shadow bytes around the buggy address:
  0x0c4a7fff83b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c4a7fff83c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
  0x0c4a7fff83d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c4a7fff83e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c4a7fff83f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0c4a7fff8400: 00 00 00 00 00 00 00 00[fa]fa fa fa fa fa fa fa
  0x0c4a7fff8410: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c4a7fff8420: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c4a7fff8430: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c4a7fff8440: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c4a7fff8450: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
  Shadow gap:              cc
==1293390==ABORTING
```

And for `./write`

```
AddressSanitizer:DEADLYSIGNAL
=================================================================
==1295964==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000
↪  (pc 0x000000400f9b bp 0x7ffe353d3510 sp 0x7ffe353d3510 T0)
==1295964==The signal is caused by a WRITE memory access.
==1295964==Hint: address points to the zero page.
    #0 0x400f9b in main
    ↪  /home/antolin/Teaching/exercises-2023/lecture_03/debugging/write」
    ↪  .cc:9
    #1 0x7f186b941492 in __libc_start_main (/lib64/libc.so.6+0x23492)
    #2 0x400dbd in _start
    ↪  (/home/antolin/Teaching/exercises-2023/lecture_03/debugging/write+0x400dbd)

AddressSanitizer can not provide additional info.
```

```
SUMMARY: AddressSanitizer: SEGV
 ↪  /home/antolin/Teaching/exercises-2023/lecture_03/debugging/write.cc:9 in
 ↪  main
==1295964==ABORTING
```

2 Profiling 101

Exercise 3: ***Sequential profiling with gprof***

If you compile and run, with an annotated code for `gprof`, you will get something like this:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 71.73      0.48     0.48      345     1.39     1.39  DumperASCII::dump(int)
 28.39      0.67     0.19      345     0.55     0.55  Simulation::compute_step()
  0.00      0.67     0.00      346     0.00     0.00  DoubleBuffer::old()
  0.00      0.67     0.00      345     0.00     0.00  DoubleBuffer::swap()
  0.00      0.67     0.00      345     0.00     0.00  DoubleBuffer::current()
  0.00      0.67     0.00      345     0.00     0.00  Grid::m() const
  0.00      0.67     0.00      345     0.00     0.00  Grid::n() const
  0.00      0.67     0.00        3     0.00     0.00  Grid::clear()
  0.00      0.67     0.00        3     0.00     0.00  Grid::Grid(int, int)
  0.00      0.67     0.00        1     0.00     0.00
   ↪  _GLOBAL__sub_I__ZN10SimulationC2Eii
  0.00      0.67     0.00        1     0.00     0.00
   ↪  Simulation::set_epsilon(float)
  0.00      0.67     0.00        1     0.00     0.00
   ↪  Simulation::set_initial_conditions()
  0.00      0.67     0.00        1     0.00   670.80  Simulation::compute()
  0.00      0.67     0.00        1     0.00     0.00
   ↪  Simulation::Simulation(int, int)
  0.00      0.67     0.00        1     0.00     0.00
   ↪  DoubleBuffer::DoubleBuffer(int, int)
```

We can see that 70% of the time is spent in the `dump` function. This function creates an image on disk representing the solution. We can call it only once at the end instead of at every iteration of the Jacobi solver. Thus, removing those calls to `dump` we get:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.12      0.15     0.15      345     0.44     0.44  Simulation::compute_step()
  0.00      0.15     0.00      346     0.00     0.00  DoubleBuffer::old()
  0.00      0.15     0.00      345     0.00     0.00  DoubleBuffer::swap()
  0.00      0.15     0.00      345     0.00     0.00  DoubleBuffer::current()
  0.00      0.15     0.00        3     0.00     0.00  Grid::clear()
  0.00      0.15     0.00        3     0.00     0.00  Grid::Grid(int, int)
```

```
  0.00      0.15     0.00        1     0.00     0.00
  ↪  _GLOBAL__sub_I__ZN10SimulationC2Eii
  0.00      0.15     0.00        1     0.00     0.00
  ↪  Simulation::set_epsilon(float)
  0.00      0.15     0.00        1     0.00     0.00
  ↪  Simulation::set_initial_conditions()
  0.00      0.15     0.00        1     0.00   150.18  Simulation::compute()
  0.00      0.15     0.00        1     0.00     0.00
  ↪  Simulation::Simulation(int, int)
  0.00      0.15     0.00        1     0.00     0.00  DumperASCII::dump(int)
  0.00      0.15     0.00        1     0.00     0.00
  ↪  DoubleBuffer::DoubleBuffer(int, int)
  0.00      0.15     0.00        1     0.00     0.00  Grid::m() const
  0.00      0.15     0.00        1     0.00     0.00  Grid::n() const
```

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

With perf you will get something similar:

```
Samples: 742  of event 'cycles:u', Event count (approx.): 288228356
  Children      Self  Command  Shared Object        Symbol
-   97.84%     0.00%  poisson  poisson              [.] _start
     _start
     __libc_start_main
   - main
      + 95.62% Simulation::compute
      + 1.92% Simulation::set_initial_conditions
-   97.84%     0.00%  poisson  libc-2.28.so         [.] __libc_start_main
     __libc_start_main
   - main
      - 95.62% Simulation::compute
           90.10% Simulation::compute_step
         - 5.49% DumperASCII::dump
            - 2.33% ?? (inlined)
               + 1.96% ?? (inlined)
            - 1.33% std::ostream::_M_insert<long>
                 1.07% std::num_put<char, std::ostreambuf_iterator<char,
                  ↪  std::char_traits<char> > >::_M_insert_int<long>
      - 1.92% Simulation::set_initial_conditions
           __sin_fma
-   97.84%     0.00%  poisson  poisson              [.] main
   - main
      - 95.62% Simulation::compute
           90.10% Simulation::compute_step
         - 5.49% DumperASCII::dump
            + 2.33% ?? (inlined)
            + 1.33% std::ostream::_M_insert<long>
```

```
      - 1.92% Simulation::set_initial_conditions
           __sin_fma
-   95.62%     0.00%  poisson  poisson              [.] Simulation::compute
   - Simulation::compute
        90.10% Simulation::compute_step
      - 5.49% DumperASCII::dump
         + 2.33% ?? (inlined)
         + 1.33% std::ostream::_M_insert<long>
-   90.10%    90.10%  poisson  poisson              [.] Simulation::compute_step
     _start
     __libc_start_main
     main
     Simulation::compute
     Simulation::compute_step
-    5.49%     1.60%  poisson  poisson              [.] DumperASCII::dump
   - 3.89% DumperASCII::dump
      + 2.18% ?? (inlined)
      + 1.33% std::ostream::_M_insert<long>
   - 1.60% _start
        __libc_start_main
        main
        Simulation::comput
        DumperASCII::dump
```

Changing the loops ordering from `ji` to `ij`, the computation time reduces from 32.4 s to 5.1 s. With just `perf stat` we do not get much insight to understand what is happening. But, if we rerun asking explicitly for the L1 cache access and misses we get:

```
$>srun [...]  perf stat -e L1-dcache-loads,L1-dcache-load-misses  ./poisson_ij
nb steps 2600 [l2 = 0.00499909]

 Performance counter stats for './poisson_ij':

    22,013,505,236      L1-dcache-loads:u
       549,583,312      L1-dcache-load-misses:u   #    2.50% of all L1-dcache
       ↪  accesses

       7.317822412 seconds time elapsed

       5.118036000 seconds user
       2.166764000 seconds sys

$>srun [...]  perf stat -e L1-dcache-loads,L1-dcache-load-misses  ./poisson_ji
nb steps 2600 [l2 = 0.00499905]

 Performance counter stats for './poisson_ji':
```

```
   22,002,938,745      L1-dcache-loads:u
   11,951,467,511      L1-dcache-load-misses:u   #   54.32% of all L1-dcache
    ↪  accesses

     34.776462873 seconds time elapsed

     32.475311000 seconds user
      2.185647000 seconds sys
```

So, as it can be seen above, the `ji` presents a 54.3% of cache misses, compared to the 2.5% of the `ij` version, what explains the difference in performance between both versions.

3 Thread Level Parallelism: OpenMP

All the correction codes are in the corresponding `solution` sub-folders.

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

By calling, `omp_get_max_threads` the maximum number of threads to be used is computed. Check, for instance, the code in `pi_for_wrong.cc`

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

The file `pi_for_wrong.cc` contains the solution code of this exercise. If you run it multiple times, the value of `pi` should be wrong and "random". It is also worth observing that the first goal was achieved, as this code scales nicely. The "random" values obtained come from the access to the `sum` variable: `sum = sum + f(x)` will result in a race condition, if not properly protected.

Exercise 7: ***Naïve reduction***

The file `pi_critical.cc` contains the solution code of this exercise. The access to `sum` needs to be protected. The naïve way of doing this is to add a `critical` section protecting its access. If you run it you should see that the more threads you add, the slower it becomes. This comes from the need of synchronization between threads inside the `critical` section. Only one thread can execute the `critical` code at a time, all the others have to wait and will "fight" to enter the `critical` region, what causes a bottleneck.

Exercise 8: ***Naïve reduction ++***

The file `pi_critical_correct.cc` contains the solution code of this exercise. By having one variable by thread we remove the constrain that was serializing the execution. We still need to be careful on the way we sum the local contributions of all threads. Having one `critical` region at the end will generate a really small serial execution between threads.

Exercise 9: ***Reduction***

The file `pi_reduction.cc` contains the solution code of this exercise. Of course, all the previous exercises where just fiddling around a capability provided by OpenMP. You can still notice that the execution times are similar to the previous exercise. This might indicate that the previous exercise is how the `reduction` is implemented in OpenMP.

Exercise 10: ***Poisson***

A solution is provided in the `solution` sub-directory. There are no particular difficulties for this code. We should mainly be careful on the `reduction` of `l2`. And remove the call to the `dump` function and sort the loops in the "right" order. This solution is the first step to a OpenMP implementation: For small grids it will not scale well due to the permanent fork/join when the parallel region starts and ends. Nevertheless, when the grid is large enough, this effect will become negligible in comparison to the computation time.