UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Computing

Prof. Marco Bertini

Shared memory: C/C++ Thread sanitizer

Thread Sanitizer (TSan)

- The Thread Sanitizer, or TSan, is an LLVM based tool for C/C++ languages that detects data races at runtime.
  - Data races occur when multiple threads access the same memory without synchronization and at least one access is a write.
  - Data races are dangerous because they can cause programs to behave unpredictably, or even result in memory corruption.
- TSan also detects other threading bugs, including uninitialized mutexes and thread leaks. It can also detect deadlocks.

Thread Sanitizer (TSan)

- The Thread Sanitizer, or TSan, is an LLVM based tool for C/C++ languages that detects data races at runtime.

Works also on Go and Swift

  - Data races occur when multiple threads access the same memory without synchronization and at least one access is a write.

  - Data races are dangerous because they can cause programs to behave unpredictably, or even result in memory corruption.

- TSan also detects other threading bugs, including uninitialized mutexes and thread leaks. It can also detect deadlocks.

Data race

Difficile da capire con un debugger

- Two threads access the same shared variable
  - at least one thread modifies the variable
  - the accesses are concurrent, i.e. unsynchronized
- Leads to non-deterministic behavior
- Hard to find with traditional debugging tools

Other sanitizers

- There are also other sanitizers (that should be used) for C/C++:
  - Address Sanitizer (ASAN, `-fsanitize=address`): detect out-of-bounds memory accesses.
  - Memory Sanitizer (MSAN, `-fsanitize=memory`): detect reads of uninitialized memory.
  - Undefined Behaviour Sanitizer (UBSAN, `-fsanitize=undefined`): detect reads of uninitialized memory.
  - Leak Sanitizer (LSAN, `-fsanitize=leak`): detect memory leaks (recent Clang versions enable this by default when ASAN is used).

Other sanitizers

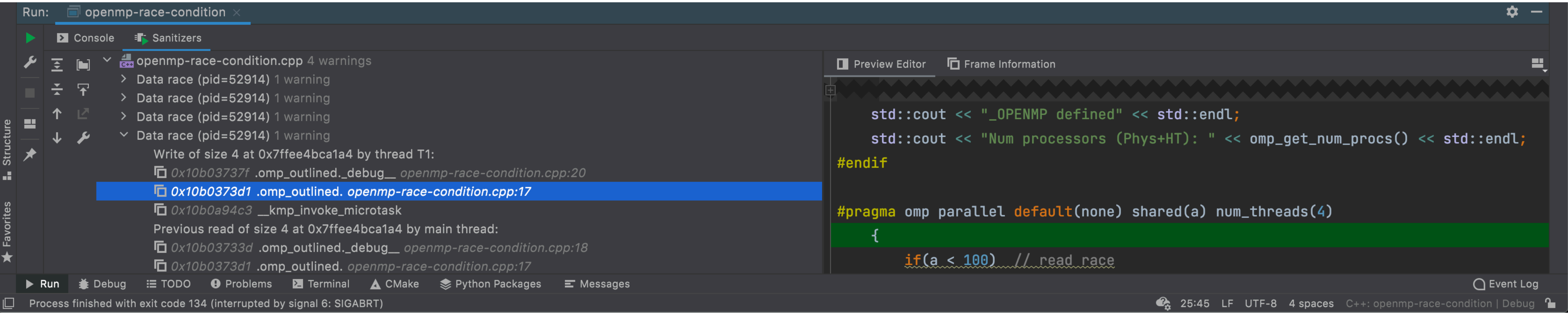
- There are also other sanitizers (that should be used) for C/C++:

All sanitizers (except MSAN) are available in GCC version >= 5

  - Address Sanitizer (ASAN, `-fsanitize=address`): detect out-of-bounds memory accesses.
  - Memory Sanitizer (MSAN, `-fsanitize=memory`): detect reads of uninitialized memory.
  - Undefined Behaviour Sanitizer (UBSAN, `-fsanitize=undefined`): detect reads of uninitialized memory.
  - Leak Sanitizer (LSAN, `-fsanitize=leak`): detect memory leaks (recent Clang versions enable this by default when ASAN is used).

IDE and sanitizers

- JetBrains CLion parses the output of the sanitizers and show them in the GUI



```
openmp-race-condition.cpp 4 warnings
  > Data race (pid=52914) 1 warning
  > Data race (pid=52914) 1 warning
  > Data race (pid=52914) 1 warning
  v Data race (pid=52914) 1 warning
      Write of size 4 at 0x7ffee4bca1a4 by thread T1:
        0x10b03737f .omp_outlined._debug__ openmp-race-condition.cpp:20
        0x10b0373d1 .omp_outlined. openmp-race-condition.cpp:17
        0x10b0a94c3 __kmp_invoke_microtask
      Previous read of size 4 at 0x7ffee4bca1a4 by main thread:
        0x10b03733d .omp_outlined._debug__ openmp-race-condition.cpp:18
        0x10b0373d1 .omp_outlined. openmp-race-condition.cpp:17
```

```
    std::cout << "_OPENMP defined" << std::endl;
    std::cout << "Num processors (Phys+HT): " << omp_get_num_procs() << std::endl;
#endif

#pragma omp parallel default(none) shared(a) num_threads(4)
    {
        if(a < 100)  // read race
```

```
Process finished with exit code 134 (interrupted by signal 6: SIGABRT)
```

25:45 LF UTF-8 4 spaces C++: openmp-race-condition | Debug

How TSan Works

- The Thread Sanitizer records the information about each memory access, and checks whether that access participates in a race. All memory accesses in the code is transformed by the compiler in the following way:
- Pseudocode for Thread Sanitizer memory access
- `// Before`

Guarda se vengono modificate da più variabili stesse zone di memoria

- `*address = ...;  // or: ... = *address;`
- `// After`
- `RecordAndCheckWrite(address);`
- `*address = ...;  // or: ... = *address;`
- Each thread stores its own timestamp and the timestamps for other threads in order to establish points of synchronization. Timestamps are incremented each time memory is accessed. By checking for consistency of memory access across threads, data races can be detected independent of the actual timing of the access. Therefore, the Thread Sanitizer can detect races even if they didn't manifest during a particular run.

How TSan Works

- The Thread Sanitizer records the information about each memory access, and checks whether that access participates in a race. All memory accesses in the code is transformed by the compiler in the following way:
- Pseudocode for Thread Sanitizer memory access
- `// Before`
- `*address = ...;  // or: ... = *address;`
- `// After`

Run disabling the ASLR (Address Space Layout Randomization) or TSAN will fail !

E.g.: `setarch $(name -m) -R ./my-executable`

disables (`-R`) the Linux ASLR

Occhio a disabilitare solo il programma e non l'intero sistema

- Each thread stores its own timestamp and the timestamps for other threads in order to establish points of synchronization. Timestamps are incremented each time memory is accessed. By checking for consistency of memory access across threads, data races can be detected independent of the actual timing of the access. Therefore, the Thread Sanitizer can detect races even if they didn't manifest during a particular run.

Performance Impact

- Running your code with Thread Sanitizer checks enabled can result in CPU slowdown of 2x to 20x, and an increase in memory usage by 5x to 10x.

- You can improve memory utilization and CPU overhead by compiling at the -01 / -02 optimization level.

- Example:

```
clang++ -fsanitize=thread -std=c++11
        -pthread -g -01
```

Performance Impact

- Running your code with Thread Sanitizer checks enabled can result in CPU slowdown of 2x to 20x, and an increase in memory usage by 5x to 10x.
- You can improve memory utilization and CPU overhead by compiling at the -O1 / -O2 optimization level.
- Example:

```
clang++ -fsanitize=thread -std=c++11
        -pthread -g -O1
```

-g to get file names and line numbers in the warning messages.

-O1 for performance

Performance Impact

- Running your code with Thread Sanitizer checks enabled can result in CPU slowdown of 2x to 20x, and an increase in memory usage by 5x to 10x.
- You can improve memory utilization and CPU overhead by compiling at the -01 / -02 optimization level.
- Example:

-fsanitize=thread is both a compiler and linker parameter !

```
clang++ -fsanitize=thread -std=c++11
        -pthread -g -01
```

-g to get file names and line numbers in the warning messages.

-01 for performance

Using TSan

- Errors detected by sanitizers are caught at runtime.
- Execute the program compiled with the sanitizer and check them, eg.:

```
% ./a.out
 WARNING: ThreadSanitizer: data race (pid=19219)
   Write of size 4 at 0x7fcf47b21bc0 by thread T1:
     #0 Thread1 tiny_race.c:4 (exe+0x00000000a360)
   Previous write of size 4 at 0x7fcf47b21bc0 by main thread:
     #0 main tiny_race.c:10 (exe+0x00000000a3b4)
   Thread T1 (running) created at:
     #0 pthread_create tsan_interceptors.cc:705 (exe+0x00000000c790)
     #1 main tiny_race.c:9 (exe+0x00000000a3a4)
```

Archer

- Archer is a data race detector for OpenMP programs.
- Starting with LLVM/10, the Archer runtime is included in LLVM releases.
- Add `-fsanitize=thread` to the OpenMP switches (compiler and linker)

Intel Inspector

- Intel Inspector is a tool with a graphical user interface that is effective at detecting race conditions and deadlocks in OpenMP code.
- It is an Intel proprietary tool, but it is now freely available. It can be installed from the OneAPI suite from Intel

Links

- https://clang.llvm.org/docs/ThreadSanitizer.html
- https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual
- https://www.intel.com/content/www/us/en/developer/tools/oneapi/inspector.html

Books

- Parallel and High Performance Computing, R. Robey, Y. Zamora, Manning - Chapt. 17