Data races in Parallel and High Performance Computing

Simone Atzeni

simone@cs.utah.edu

February 3, 2016

What is a data race?

- A **data race** occurs when **two or more threads**:
  - **access the same memory location** (i.e. shared variable)
  - **at least one thread writes the memory location**
  - **the accesses are concurrent and unsynchronized**
- Leads to non-deterministic behavior (crashes, program exceptions, wrong results, etc.)
- Hard to find with traditional debugging tools

What is a data race?

PThreads

```
void Thread1() {  // Runs in Thread1.
  sum = sum + 1;
}
⋮
void ThreadN() {  // Runs in ThreadN.
 sum = sum + 1;
}
```

OpenMP

```
#pragma omp parallel num_threads(N)
   sum = sum + 1;
```

Why is finding data races important?

- Data race are undefined behavior:
  - Non-deterministic results
  - Program errors or exceptions
  - Critical consequences
- Data races affect several fields:
  - File systems
  - Security
  - Life-critical systems
- Compiler optimization can introduce races:
  - “Benign” races (there is no such things!)
  - Instruction reordering
- Data race prevention:
  - Too much synchronization → bad performance and deadlocks

OpenMP Data Races: Example 1

```
#pragma omp parallel for
  for (i = 1; i < N; i++) {
    a[i] = 2.0 * i * (i - 1);
    b[i] = a[i] - a[i - 1];
  }
```

- Solution

  Include accesses to array `a` within a critical section.

  or

  Recompute the second espression.

OpenMP Data Races: Example 1

```
#pragma omp parallel for
  for (i = 1; i < N; i++) {
    #pragma omp critical
    {
      a[i] = 2.0 * i * (i - 1);
      b[i] = a[i] - a[i - 1];
    }
}
```

OpenMP Data Races: Example 1

```
#pragma omp parallel for
  for (i = 1; i < N; i++) {
    a[i] = 2.0 * i * (i - 1);
    b[i] = a[i] - 2.0 * (i - 1) * (i - 2);
}
```

OpenMP Data Races: Example 2

```
#pragma omp parallel
  {
#pragma omp for nowait
    for (i = 1; i < N; i++) {
      a[i] = 3.0 * i * (i + 1);
    }
#pragma omp for
    for (i = 1; i < N; i++) {
      b[i] = a[i] - a[i - 1];
    }
  }
}
```

- Solution

  Remove the `nowait` clause from first for-loop.

OpenMP Data Races: Example 2

```
#pragma omp parallel
  {
#pragma omp for
    for (i = 1; i < N; i++) {
      a[i] = 3.0 * i * (i + 1);
    }
#pragma omp for
    for (i = 1; i < N; i++) {
      b[i] = a[i] - a[i - 1];
    }
  }
}
```

OpenMP Data Races: Example 3

```
#pragma omp parallel for
  for (i = 1; i < N; i++) {
    x = sqrt(b[i]) - 1;
    a[i] = x * x + 2 * x + 1;
  }
}
```

- Solution

  Add `private(x)` in omp pragma.

OpenMP Data Races: Example 3

```
#pragma omp parallel for private(x)
  for (i = 1; i < N; i++) {
    x = sqrt(b[i]) - 1;
    a[i] = x * x + 2 * x + 1;
  }
}
```

- Solution

  Add `private(x)` in omp pragma.

OpenMP Data Races: Example 4

```
#pragma omp parallel for
  for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
      a[i][j] = compute(i,j);
    }
  }
```

- Solution

  Add `private(x)` in omp pragma.

OpenMP Data Races: Example 4

```
#pragma omp parallel for private(j)
  for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
      a[i][j] = compute(i,j);
    }
  }
```

- Solution

  Add `private(x)` in omp pragma.

- Good Practice

  Use clause `default(none)`, explicitly share variables with `shared(…)`

OpenMP Data Races: Example 5

```
#pragma omp parallel for
  for (i = 1; i < N; i++)
    {
      sum = sum + a[i];
    }
}
```

- Solution

  Add `reduction(+:sum)` to omp pragma.

OpenMP Data Races: Example 5

```
#pragma omp parallel for reduction(+:sum)
  for (i = 1; i < N; i++)
    {
      sum = sum + a[i];
    }
}
```

- Solution

  Add `reduction(+:sum)` to omp pragma.

OpenMP Data Races: Example 6

```
#pragma omp parallel for private(local)
{
  #pragma omp master
    init = 10;
  local = init
}
```

- Solution

  `master` construct does not have an implied barrier better use `single`

OpenMP Data Races: Example 6

```
#pragma omp parallel for private(local)
{
  #pragma omp single
    init = 10;
  local = init
}
```

- Solution

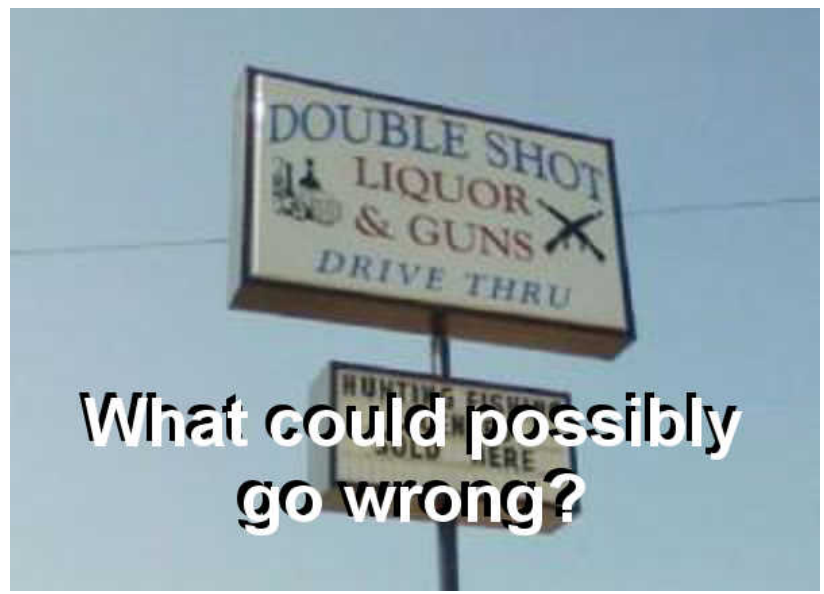
  `master` construct does not have an implied barrier better use `single`

“Benign” data races in OpenMP

```
int num_threads;

#pragma omp parallel
{
  num_threads = omp_get_num_threads();
}
```

- This is undefined behavior in C/C++.
- Compiler transformations on racy code might transform a “benign race” in a very bad behavior.



"Benign" data races in OpenMP

```
#pragma omp parallel
{
  foo = …;  // Store a pointer to function.
  num_threads = foo;  // Spill from register into the stop variable.
  …
  foo = num_threads;  // Restore from the stop variable into a register.
  call(foo) ...;  // Call function pointed by foo.
  num_threads = omp_get_num_threads();
}
```

- A compiler transformation could spill some temp (i.e. function pointer) value into `num_threads`:
  - One thread spills pointer to `write_file()` function.
  - Another thread spills pointer to `launch_nuclear_missile()` function.
- The first thread restores its pointer, it will get the wrong pointer to `launch_nuclear_missile()`.
- The "benign" *data race* just lead to an accidental missile launch.

"Benign" races are bad, find and fix them as any other type of data race![1]

[1]See Benign data races: what could possibly go wrong? by Dmitry Vyukov.

“Benign” data races in OpenMP

Real-World example at Lawrence Livermore National Laboratory

- HYDRA – Large multiphysics MPI/OpenMP application.
- Porting on one of the largest supercomputer in the world Sequoia.
- Non-deterministic crashes on a threaded version of Hypre library.
- Above certain optimization levels and certain scales (8K MPI processes).

Archer finds data races on HYDRA (Hypre library):

- Three data races found inside fairly complex OpenMP region.
- The races are “benign” (same value written multiple times in the same memory location).
- Crash manifests only at “> O0” optimization level.
- The compiler (IBM XL), assuming race-free code for optimizations, transforms “benign” races in harmful races.

Data Race Detection Techiniques

- Static Analysis
  - Reasons about all inputs/interleavings
  - Very imprecise, many false positives and miss races
  - Very scalable and fast (i.e. no runtime overhead)
- Dynamic Analysis
  - Very precise, no false positives
  - Reports races only in branches of the programs that are actually executed
  - Very high runtime and memory overhead

Data Race Detection Tools

- Not many OpenMP race detectors out there!
- Commercial tools:
  - Intel Static Secure Analysis (static analysis)
  - Intel Inspector XE (dynamic analysis)
  - Sun Studio Data-Race Detection Tool (dynamic analysis)
- Open-source tools:
  - ThreadSanitizer (only PThreads programs, dynamic analysis)
  - Archer, based on Clang/LLVM and ThreadSanitizer (static and dynamic analysis)

ThreadSanitizer: data race detector

- Error checking tool for
  - Memory errors
  - Threading errors (Pthreads)
- Based on LLVM/Clang
- Runtime analysis
- Available for Linux, Windows and Mac
- Supports C, C++, and Fortran
- More info: https://github.com/google/sanitizers

ThreadSanitizer: Usage and Limitations

- Compile the program with the following command:
  - **`clang -g -fsanitize=thread myprog.c -o myprog`**
- Runtime check
  - Error detection only in software branches that are executed
- Low runtime overhead
  - Roughly 2x - 20x
  - Detect races only in PThreads applications
  - No false positives
- Compiler instrumentation
  - Slower compilation process (apply different passes on the source code to identify race free regions of code, instruments only the rest), faster and more precise at runtime

ThreadSanitizer: Result Summary

```
5  int var;
6
7  void Thread1() {
8    var++;
9  }
10
11 void Thread2() {
12   var++;
13 }
```

WARNING: ThreadSanitizer: data race

Write of size 4 at 0x0000014b2e90 by thread T2:

    #0 Thread2 race1.c:12:6

Previous write of size 4 at 0x0000014b2e90 by thread T1:

    #0 Thread1 race1.c:8:6

Location is global 'var' of size 4

Archer: Features and Limitations

- Static Analysis
  – Only for OpenMP programs
  – Exclude race free regions and sequential code from runtime analysis to reduce overhead
- Runtime check
  – Error detection only in software branches that are executed
- Low runtime overhead
  – Roughly 2x - 20x
  – Detect races in large OpenMP applications
  – No false positives
- Compiler instrumentation
  – Slower compilation process (apply different passes on the source code to identify race free regions of code, instruments only the rest), faster and more precise at runtime

Archer: Usage

- Compile the program with the **–g** compiler flag
  - **`clang-archer myprog.c –o myprog`**
- Run the program under control of Archer Runtime
  - **`export OMP_NUM_THREADS=...`**
    `./myprog`
  - Detects problems only in software branches that are executed
- Understand and correct the threading errors detected
- Edit the source code
- Repeat until no errors reported

Archer: Result Summary

```
6     #pragma omp parallel
7     {
8         for(int i = 0; i < 100; i++)
9             a[i] = a[i] * a[i];
10    }
11
12    #pragma omp parallel
13    {
14        if (a < 100) {
15            #pragma omp critical
16            a++;
17        }
18    }
```

- Excluded from them runtime check because it is race free.

WARNING: ThreadSanitizer: data race

- Read of size 4 at 0x7fffffffdcdc by thread T2:

  #0 .omp_outlined. race.c:14 (race +0x0000004a6dce)

  #1 __kmp_invoke_microtask <null> (libomp_tsan.so)

- Previous write of size 4 at 0x7fffffffdcdc by main thread:

  #0 .omp_outlined. race.c:16 (race +0x0000004a6e2c)

  #1 __kmp_invoke_microtask <null> (libomp_tsan.so)

Archer: OpenMP data race detector

- Error checking tool for
  - Memory errors
  - Threading errors (**OpenMP**, Pthreads)
- Based on LLVM/Clang and ThreadSanitizer
- Static and Runtime analysis
- Available for Linux, Windows and Mac
- Supports C, C++
- More info: https://github.com/PRUNER/archer

Archer: Features and Limitations

- Static Analysis
  - Only for OpenMP programs
  - Exclude race free regions and sequential code from runtime analysis to reduce overhead
- Runtime check
  - Error detection only in software branches that are executed
- Low runtime overhead
  - Roughly 2x - 20x
  - Detect races in large OpenMP applications
  - No false positives
- Compiler instrumentation
  - Slower compilation process (apply different passes on the source code to identify race free regions of code, instruments only the rest), faster and more precise at runtime

Archer: Usage

- Compile the program with the **–g** compiler flag
  - **`clang-archer myprog.c –o myprog`**
- Run the program under control of Archer Runtime
  - **`export OMP_NUM_THREADS=...`**
    `./myprog`
  - Detects problems only in software branches that are executed
- Understand and correct the threading errors detected
- Edit the source code
- Repeat until no errors reported

Archer: Result Summary

```
6     #pragma omp parallel
7     {
8       for(int i = 0; i < 100; i++)
9           a[i] = a[i] * a[i];
10    }
11
12    #pragma omp parallel
13    {
14      if (a < 100) {
15          #pragma omp critical
16          a++;
17      }
18    }
```

Excluded from them runtime check because it is race free.

WARNING: ThreadSanitizer: data race

Read of size 4 at 0x7fffffffdcdc by thread T2:

#0 .omp_outlined. race.c:14 (race +0x0000004a6dce)

#1 __kmp_invoke_microtask <null> (libomp_tsan.so)

Previous write of size 4 at 0x7fffffffdcdc by main thread:

#0 .omp_outlined. race.c:16 (race +0x0000004a6e2c)

#1 __kmp_invoke_microtask <null> (libomp_tsan.so)