OpenMP®

SC22 Booth Talk Series

SC22 Dallas, TX | hpc accelerates.

What Could Possibly Go Wrong Using OpenMP?

Ruud van der Pas
Senior Principal Software Engineer, Oracle Linux Engineering

$ whoishe

My background is in mathematics and physics

Previously, I worked at Philips, the University of Utrecht, Convex Computer, SGI, and Sun Microsystems

Currently I work in the Oracle Linux Engineering organization

I have been involved with OpenMP since the introduction

I am passionate about performance and OpenMP in particular

What Could Possibly Go Wrong Using OpenMP?

Nothing

of course

or maybe ...

Where Could Things Go Wrong Then?

What Do You Actually Mean with “Wrong”?

There are two things that can go wrong

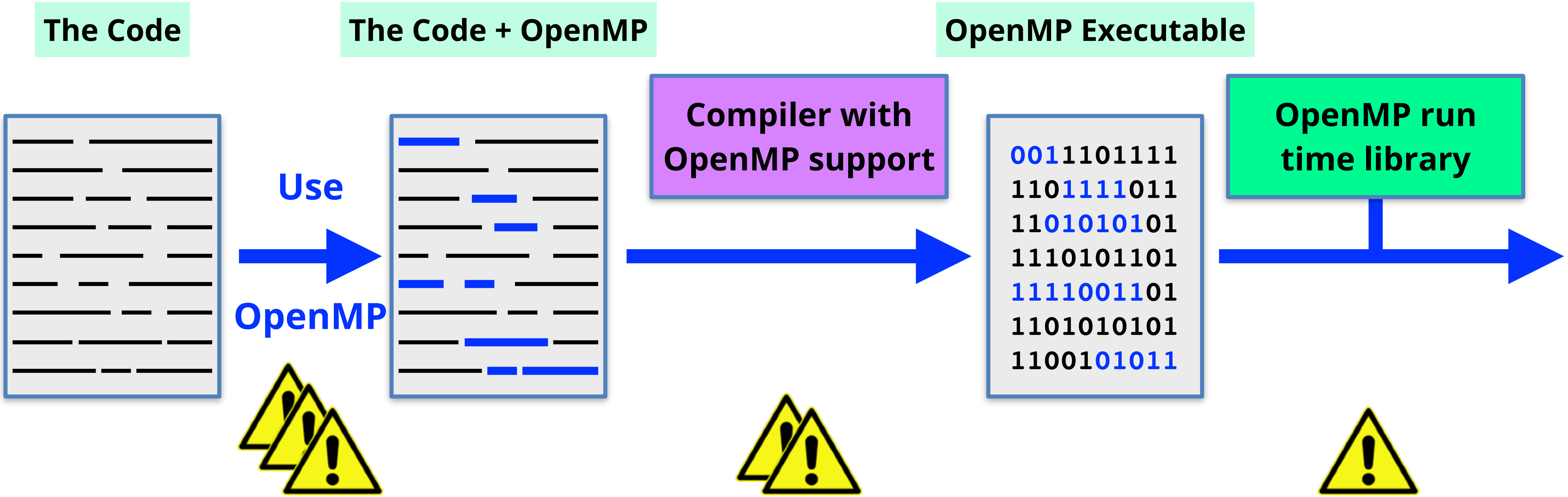
- ***An incorrect answer***
- ***Poor parallel performance***

In this talk, we cover the top 3 of both categories

The focus is on the first category though*

****) Several of my previously recorded SC OpenMP booth talks focus on performance and can be found at https://www.youtube.com/c/OpenMPARB***

The Big OpenMP Picture



Disclaimer

All the examples to follow next were found in real applications

Wrong Answers - The Top Three

The code has been incorrectly parallelized

The scoping (private, shared, etc.) rules are violated

A data race has been introduced

Incorrect Parallelization - An Example

This loop has a data dependence

```
prev_val = a[0];
for (int i=1; i<n; i++)
 {
   a[i] = prev_val + b[i];
   prev_val = a[i];
 }
```

Still a data dependence ...

```
for (int i=1; i<n; i++)
 {
   a[i] = a[i-1] + b[i];
 }
```

Incorrect Parallelization - An Example

Force the loop to execute in parallel

```
prev_val = a[0];
#pragma omp parallel for
for (int i=1; i<n; i++)
 {
   a[i] = prev_val + b[i];
   prev_val = a[i];
 }
```

```
$ gcc -fopenmp wrong.c
$ export OMP_NUM_THREADS=4
$ ./a.out
Loop length n = 10
Number of threads = 4
Number of errors = 6
a[1] =   3 ref[1] =   3
a[2] =   6 ref[2] =   6
a[3] =  10 ref[3] =  10
a[4] =  34 ref[4] =  15 *
a[5] =  40 ref[5] =  21 *
a[6] =  36 ref[6] =  28 *
a[7] =  44 ref[7] =  36 *
a[8] =  19 ref[8] =  45 *
a[9] =  29 ref[9] =  55 *
```

Incorrect Parallelization - Wrong Results (of course)

Force the loop to execute in parallel

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

→

```
$ gcc -fopenmp wrong.c
$ export OMP_NUM_THREADS=4
$ ./a.out
Loop length n = 10
Number of threads = 4
Number of errors = 4
a[1] =   3 ref[1] =   3
a[2] =   6 ref[2] =   6
a[3] =  10 ref[3] =  10
a[4] =  15 ref[4] =  15
a[5] =  21 ref[5] =  21
a[6] =  47 ref[6] =  28 *
a[7] =  55 ref[7] =  36 *
a[8] =  64 ref[8] =  45 *
a[9] =  74 ref[9] =  55 *
```

Incorrect Parallelization - Morale

Parallelize code that is not parallel => maybe wrong results

Yes, "maybe". In the worse case the results are sometimes ok

There is a simple trick, but it works only one way

If it is a loop, run the sequential version backwards

If the results are wrong, you know it is not parallel as written

If the results are correct, you still don't know …

Incorrect Parallelization - Some Tips

Use a profiling tool to see if this code part actually matters

If this is the case, try to find a parallel version

But, be aware it is still efficient on a single thread

Isolate the sequential part and parallelize the remainder

In doing so, try to avoid excessive extra cache/memory traffic

Wrong Answers - Violation of the Scoping Rules

The previous example also included a wrong scoping case

Variable prev_val was implicitly scoped as "shared"

This is one of the common pitfalls, but not the only one

The most common mistake is about private variables

Recall that they are undefined outside of the parallel region

```
prev_val = a[0];
#pragma omp parallel for
for (int i=1; i<n; i++)
 {
  a[i] = prev_val + b[i];
  prev_val = a[i];
 }
```

Incorrect Scoping - Another Example

```
int my_var = 10;
#pragma omp parallel for private(my_var)
for (int i=0; i<n; i++)
 {
  a[i] = my_var + b[i];
 }
```

Variable my_var is undefined

Even if this might work today, there is no guarantee for tomorrow

Incorrect Scoping - The Solution

```
int my_var = 10;
#pragma omp parallel for firstprivate(my_var)
for (int i=0; i<n; i++)
 {
   a[i] = my_var + b[i];
 }
```

Variable my_var is implied to be private
Each thread has a local copy with an initial value of 10

Incorrect Scoping - Morale

Declare variables local to a code block where possible
They are automatically privatized
Specify the scope of the remaining variables yourself
This is not as hard as it may seem
Extremely rewarding when it comes to avoiding bugs

Wrong Answers - Data Races

A data race occurs if all the following conditions are met

- ***<u>Multiple</u> threads access the <u>same</u> memory location <u>concurrently</u>***
- ***At least one of the accesses <u>modifies</u> the contents of this location***
- ***There is <u>no control to guarantee exclusive access</u> to this location***

A data race may lead to silent data corruption

The wrong results are also non-deterministic

Yes, the results may vary, even across identical runs

Incorrect Code - A Data Race Example

```
int my_shared_var = 0;
#pragma omp parallel for shared(my_shared_var)
for (int i=0; i<n; i++)
 {
  my_shared_var += a[i];
 }
```

WRONG!

The above code meets all 3 conditions

At any moment, multiple threads may read and write my_shared_var

Incorrect Code - Fixing the Data Race Example

```
int my_shared_var = 0;
#pragma omp parallel for reduction(+:my_shared_var)
for (int i=0; i<n; i++)
 {
  my_shared_var += a[i];
 }
```

As simple as it looks, the reduction clause generates non-trivial code that avoids the data race

Data Races - Morale

Data races are very nasty

Luckily, OpenMP provides high level constructs to avoid them

In less common cases, use alternatives that avoid data races:

- ***Atomic operations***
- ***Critical regions***
- ***Barriers***
- ***Locks***

Please us these!

These help to make it easier to avoid data races

Poor Performance - The Top Three

Too much parallel overhead
Consolidate as much work as possible in a <u>single parallel region</u>
Load balancing
Consider the schedule clause and tasking
Non-Uniform Memory Acces (NUMA)
Experiment with the affinity related environment variables

Summary

We covered some major mistakes made

Unfortunately, these, or others could happen to you too

What helps, is to regularly check for correctness

The performance issues mentioned are the tip of the iceberg

But, it is a big tip :-)

Make sure to use a profiling tool to guide you with the tuning

OpenMP®

SC22 Booth Talk Series

openmp.org OpenMP API spec, videos, reference guides, and more

link.openmp.org/sc22 Videos and PDFs of OpenMP SC22 presentations

Thank You And … Stay Tuned!

Bad OpenMP Does Not Scale

Ruud van der Pas

SC22 OpenMP Booth Talk