What Could Possibly Go Wrong Using OpenMP?

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

<table>
<thead>
<tr>
<th>My background is in mathematics and physics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Previously, I worked at Philips, the University of Utrecht, Convex Computer, SGI, and Sun Microsystems</td>
</tr>
<tr>
<td>Currently I work in the Oracle Linux Engineering organization</td>
</tr>
<tr>
<td>I have been involved with OpenMP since the introduction</td>
</tr>
<tr>
<td>I am passionate about performance and OpenMP in particular</td>
</tr>
</tbody>
</table>
What Could Possibly Go Wrong Using OpenMP?
Nothing

of course

or maybe ...
Where Could Things Go Wrong Then?
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

What Could Possibly Go Wrong Using OpenMP?

The Code

The Code + OpenMP

OpenMP Executable

Compiler with OpenMP support

OpenMP run time library

Use

OpenMP

The Code

The Code + OpenMP

OpenMP Executable

Compiler with OpenMP support

OpenMP run time library

What Could Possibly Go Wrong Using OpenMP?
Disclaimer

All the examples to follow next were found in real applications
<table>
<thead>
<tr>
<th>Wrong Answers - The Top Three</th>
</tr>
</thead>
<tbody>
<tr>
<td>The code has been incorrectly parallelized</td>
</tr>
<tr>
<td>The scoping (private, shared, etc.) rules are violated</td>
</tr>
<tr>
<td>A data race has been introduced</td>
</tr>
</tbody>
</table>
Incorrect Parallelization - An Example

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

This loop has a data dependence

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

Still a data dependence...

```c
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];
}
```

```bash
$ 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[8] = 19 ref[8] = 45 *
```
Incorrect Parallelization - Wrong Results (of course)

**Force the loop to execute in parallel**

```c
#include <stdio.h>

#define N 10

int a[N], b[N];

int main() {
    for (int i=1; i<N; i++) {
        a[i] = a[i-1] + b[i];
    }
    return 0;
}
```

```
$ gcc -fopenmp wrong.c
$ export OMP_NUM_THREADS=4
$ ./a.out
Loop length n = 10
Number of threads = 4
Number of errors = 4
```

<table>
<thead>
<tr>
<th>i</th>
<th>a[i]</th>
<th>ref[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>6</td>
<td>47</td>
<td>28 *</td>
</tr>
<tr>
<td>7</td>
<td>55</td>
<td>36 *</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>45 *</td>
</tr>
<tr>
<td>9</td>
<td>74</td>
<td>55 *</td>
</tr>
<tr>
<td>Incorrect Parallelization - Morale</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------------------</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **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 ...**
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];
}
```
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

```c
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**
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
A data race occurs if all the following conditions are met:

- Multiple threads access the same memory location concurrently.
- At least one of the accesses modifies the contents of this location.
- There is no control to guarantee exclusive access 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];
}

The above code meets all 3 conditions
At any moment, multiple threads may read and write my_shared_var
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 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 use 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 single parallel region**

**Load balancing**

**Consider the schedule clause and tasking**

**Non-Uniform Memory Access (NUMA)**

**Experiment with the affinity related environment variables**
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 API spec, videos, reference guides, and more

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