SC23 Booth Talk Series

Make OpenMP Go Fast

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

You can get good performance with OpenMP

And your code will scale

If you do things in the right way
In this talk we cover the basics how to get good performance

Follow the guidelines and the performance should be decent

An OpenMP compiler and runtime should Do The Right Thing

You may not get blazing scalability, but ...

The lawyers in the OpenMP Performance Court have no case against you
The ease of use of OpenMP is a mixed blessing (but I still prefer it over the alternative)

**Ideas are easy and quick to implement**

But some constructs are more expensive than others

**If you write dumb code, you probably get dumb performance**

Just don’t blame OpenMP, please*

*) It is fine to blame the weather, or politicians, or both though
My Preferred Tuning Strategy

<table>
<thead>
<tr>
<th>In terms of complexity, use the most efficient algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select a profiling tool</td>
</tr>
<tr>
<td>Find the highest level of parallelism</td>
</tr>
<tr>
<td>(this should however provide enough work to use many threads)</td>
</tr>
<tr>
<td>Use OpenMP in an efficient way</td>
</tr>
<tr>
<td>Be prepared to have to do some performance experiments</td>
</tr>
</tbody>
</table>
Two Things You Need To Know
A core may, or may not, support **hardware threads**

This is part of the design

These hardware threads may accelerate the execution of a single application, or improve the throughput of a workload

The idea is that the pipeline is used by another thread in case the current thread is idle
How Hardware Threads Work

No hardware threads

Two hardware threads

saved time

Time
Suppose your application needs 100 seconds to run

If 80% of this run time can execute in parallel, the time using 4 threads is \( 80/4+20 = 40 \text{ seconds} \)

This means that your program is 2.5x faster, not 4x
Amdahl’s Law Using 16 Threads
Amdahl’s Law shows that you need to parallelize a significant fraction of the run time to see a decent speed up for higher thread counts.

The BIG issue is that the serial, single thread, part of your code will dominate sooner than later.

And this is why you should pay attention to improve the single thread performance, before parallelizing your code.
How To Not Write Dumb OpenMP Code
**The Basics For All Users**

<table>
<thead>
<tr>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do not parallelize what does not matter</td>
</tr>
<tr>
<td>Never tune your code without using a profiling tool</td>
</tr>
<tr>
<td>Do not share data unless you have to</td>
</tr>
<tr>
<td>(in other words, use private data as much as you can)</td>
</tr>
<tr>
<td><strong>Think BIG</strong></td>
</tr>
<tr>
<td>(maximize the size of the parallel regions)</td>
</tr>
<tr>
<td>One “parallel for” is fine. More, back to back, is EVIL.</td>
</tr>
</tbody>
</table>
The Wrong and Right Way Of Doing Things

```
#pragma omp parallel for
{ <code block 1> }
```

```
#pragma omp parallel for
{ <code block n> }
```

Parallel region overhead repeated “n” times

No potential for the “nowait” clause

```
#pragma omp parallel
{
#pragma omp for
{ <code block 1> }
```

```
#pragma omp for nowait
{ <code block n> }
```

```
} // End of parallel region
```

Parallel region overhead only once

Potential for the “nowait” clause
Every barrier matters
(and please use them carefully)

The same is true for locks and critical regions
(use atomic constructs where possible)

EVERYTHING Matters
(minor overheads get out of hand eventually)
Another Example

The second barrier is redundant because the single construct has an implied barrier already (this second barrier will still take time though)
More Things to Consider

<table>
<thead>
<tr>
<th>Identify opportunities to use the <code>nowait</code> clause</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>(a very powerful feature, but be aware of data races)</em></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use the <code>schedule</code> clause in case of load balancing issues</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Avoid nested parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>(the nested barriers really add up)</em></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Consider tasking instead</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>(provides much more flexibility and finer granularity)</em></td>
</tr>
</tbody>
</table>
Case Study - Graph Analysis
The Scalability is Disappointing

- The data is for a 9 GB sized problem (SCALE 24)
- Search time reduces as threads are added
- Benefit from all 16 (hyper) threads
- The 3.2x parallel speed up is disappointing
- The parallel scalability is similar for larger graphs

System: A cloud instance with 8 Intel Xeon Platinum 8167M CPU @ 2.00GHz ("Skylake") cores, 16 hardware threads
Performance Of The Original and Modified Code

✓ A noticeable reduction in the search time at 4 threads and beyond
✓ The parallel speed up increases to 6.5x
✓ The search time is reduced by 2x

The modified version is 2x faster than the original code

“Make OpenMP Go Fast”
A Comparison Between 1 And 2 Threads

“Make OpenMP Go Fast”
Zoom In On The Second Thread

- Main thread is fine.
- Second thread has gaps.

“Make OpenMP Go Fast”
The Issue

```c
#pragma omp for
for (int64_t k = k1; k < oldk2; ++k) {
    const int64_t v = vlist[k];
    const int64_t veo = XENDOFF(v);
    for (int64_t vo = XOFF(v); vo < veo; ++vo) {
        const int64_t j = xadj[vo];
        if (bfs_tree[j] == -1) {
            if (int64_cas (&bfs_tree[j], -1, v)) {
                if (kbuf < THREAD_BUF_LEN) {
                    nbuf[kbuf++] = j;
                } else {
                    int64_t voff = int64_fetch_add (&k2, THREAD_BUF_LEN);
                    assert (voff + THREAD_BUF_LEN <= nv);
                    for (int64_t vk = 0; vk < THREAD_BUF_LEN; ++vk)
                        vlist[voff + vk] = nbuf[vk];
                    nbuf[0] = j;
                    kbuf = 1;
                }
            }
        }
    }
}
```

- **Fixed length loop**
- **Irregular length loop**
- **Irregular workload per k-iteration**
- **Irregular control flow**
The Performance With Dynamic Scheduling Added

✓ A 1% slow down on a single thread
✓ Near linear scaling for up to 8 threads
✓ The parallel speed up increased to 9.5x
✓ The search time is reduced by 3x

The modified version is 3x faster than the original code

“Make OpenMP Go Fast”
### Takeaways

There are many opportunities to improve the performance

If you follow the advice given, you should be fine

(in most of the cases, since there are always exceptions)

Use a profiling tool to guide you

Don’t guess, since it is likely you might be wrong
Thank You And ... Stay Tuned!

Bad OpenMP
Does Not Scale

Ruud van der Pas
SC23 OpenMP Booth Talk