Speed Up Your OpenMP Code Without Doing Much

Ruud van der Pas
Distinguished Engineer

Oracle Linux and Virtualization Engineering
Oracle, Santa Clara, CA, USA
“Speed Up Your OpenMP Code Without Doing Much”

Agenda

• Motivation
• Low Hanging Fruit
• A Case Study
Motivation
OpenMP and Performance

You can get good performance with OpenMP

And your code will scale

If you do things in the right way

Easy $\neq$ Stupid

“Speed Up Your OpenMP Code Without Doing Much”
Ease of Use?

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

Often, a small change can have a big impact

In this talk we show some of this low hanging fruit
Low Hanging Fruit

“Speed Up Your OpenMP Code Without Doing Much”
### About Single Thread Performance and Scalability

<table>
<thead>
<tr>
<th>You <strong>have to pay</strong> attention to single thread performance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Why?</strong> If your code performs badly on 1 core, what do you think will happen on 10 cores, 20 cores, … ?</td>
</tr>
<tr>
<td><strong>Scalability can mask poor performance!</strong></td>
</tr>
<tr>
<td>(a slow code tends to scale better …)</td>
</tr>
</tbody>
</table>
# The Basics For All Users

- Do not parallelize what does not matter
- Never tune your code without using a profiler
- Do not share data unless you have to (in other words, use private data as much as you can)
- One “parallel for” is fine. More is EVIL.
- Think BIG (maximize the size of the parallel regions)

**“Speed Up Your OpenMP Code Without Doing Much”**
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

“Speed Up Your OpenMP Code Without Doing Much”
## More Basics

| 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

```c
#pragma omp single
{
    <some code>
} // End of single region

#pragma omp barrier

<more code>
```

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

```c
a[npoint] = value;
#pragma omp parallel ...
{
    #pragma omp for
    for (int64_t k=0; k<npoint; k++)
        a[k] = -1;
    #pragma omp for
    for (int64_t k=npoint+1; k<n; k++)
        a[k] = -1;

    <more code>
}
// End of parallel region
```
What Is Wrong With This?

```c
a[npoint] = value;
#pragma omp parallel ...
{
    #pragma omp for
    for (int64_t k=0; k<npoint; k++)
        a[k] = -1;
    #pragma omp for
    for (int64_t k=npoint+1; k<n; k++)
        a[k] = -1;
    <more code>
} // End of parallel region
```

✓ There are 2 barriers
✓ Two times serial and parallel overhead
✓ Performance benefit depends on the value of “npoint” and “n”
The Sequence Of Operations

```
<table>
<thead>
<tr>
<th></th>
<th>npoint</th>
<th>value</th>
<th>npoint</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>....</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
```

```
0 ... npoint-1
```

```
barrier
```

```
npoint+1 ... n-1
```

```
barrier
```

“Speed Up Your OpenMP Code Without Doing Much”
The Actual Operation Performed

-1 -1 .... -1 -1 -1 value -1 ...

0 ... npoint-1 npoint npoint+1 ... n-1
The Modified Code

```c
#pragma omp parallel ...
{
    #pragma omp for
    for (int64_t k=0; k<n; k++)
        a[k] = -1;

#pragma omp single nowait
    {a[npoint] = value;}

<more code>
}
```
A Case Study

“Speed Up Your OpenMP Code Without Doing Much”
The Application

The OpenMP reference version of the Graph 500 benchmark

Structure of the code:

- Construct an undirected graph of the specified size
- Randomly select a key and conduct a BFS search
- Verify the result is a tree

For the benchmark score, only the search time matters

Repeat <n> times
## Testing Circumstances

The code uses a parameter `SCALE` to set the size of the graph

The value used for `SCALE` is 24 (~9 GB of RAM used)

All experiments were conducted in the Oracle Cloud ("OCI")

Used a VM instance with 1 Skylake socket (8 cores, 16 threads)

Operating System: Oracle Linux; compiler: gcc 8.3.1

The Oracle Performance Analyzer was used to make the profiles
The Dynamic Behaviour

Graph Generation

Search and verification for 64 keys

Note: The Oracle Performance Analyzer was used to generate this time line
The Performance For SCALE = 24 (9 GB)

- Search time reduces as threads are added
- Benefit from all 16 (hyper) threads
- But the 3.2x parallel speed up is disappointing
### Action Plan

<table>
<thead>
<tr>
<th>Used a profiling tool* to identify the time consuming parts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Found several opportunities to improve the OpenMP part</td>
</tr>
<tr>
<td>These are actually shown earlier in this talk</td>
</tr>
<tr>
<td>Although simple changes, the improvement is substantial:</td>
</tr>
</tbody>
</table>

*) Use your preferred tool; we used the Oracle Performance Analyzer
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
Are We Done Yet?

The 2x reduction in the search time is encouraging

The efforts to achieve this have been limited

The question is whether there is more to be gained

Let’s look at the dynamic behaviour of the threads:
A Comparison Between 1 And 2 Threads

Both phases benefit from using 2 threads

Note: The Oracle Performance Analyzer was used to generate this time line
Zoom In On The Second Thread

Note: The Oracle Performance Analyzer was used to generate this time line

“Speed Up Your OpenMP Code Without Doing Much”
How About 4 Threads?

Note: The Oracle Performance Analyzer was used to generate this time line

Adding threads makes things worse
The Problem

```c
#pragma omp for
for (int64_t k = k1; k < oldk2; ++k) {
    for (int64_t vo = XOFF(v); vo < veo; ++vo) {
        if (bfs_tree[j] == -1) {
            // More code and irregular
        } // End of if
    } // End of for-loop on vo
} // End of parallel for-loop on k
```

- Regular, structured parallel loop
- Irregular loop
- Unbalanced flow

"Speed Up Your OpenMP Code Without Doing Much"
The Solution

#pragma omp for schedule(runtime)
for (int64_t k = k1; k < oldk2; ++k) {
    for (int64_t vo = XOFF(v); vo < veo; ++vo) {
        if (bfs_tree[j] == -1) {
            // Even more code and irregular
        } // End of if
    } // End of for-loop on vo
} // End of parallel for-loop on k

$ export OMP_SCHEDULE="dynamic,25"
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 Improvement

Note: The Oracle Performance Analyzer was used to generate this time line

“Speed Up Your OpenMP Code Without Doing Much”
Conclusions

A good profiling tool is indispensable when tuning applications.

With some simple improvements in the use of OpenMP, a 2x reduction in the search time is realized.

Compared to the original code, adding dynamic scheduling reduces the search time by 3x.
Thank You And … Stay Tuned!

Ruud van der Pas
Distinguished Engineer
Oracle Linux and Virtualization Engineering
Oracle, Santa Clara, CA, USA