Mastering OpenMP Performance
Get OpenMP To Perform Well With Minimal Efforts

Ruud van der Pas
Webinar, March 18, 2021
My background is in mathematics and physics

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

Currently I am in the Oracle Linux Engineering organization

I have been involved with OpenMP since the introduction

Passionate about performance and OpenMP in particular
Your Onestop Place for OpenMP

https://www.openmp.org

A printed copy of the 5.1 OpenMP specs is also available on Amazon (700+ pages, 1.7 kg)
Food for the Eyes and Brains

- Intro Parallel Computing
- Covers the OpenMP Basics
- Covers all of the OpenMP 4.5 features
Part I - Tips and Tricks
**“OpenMP Does Not Scale”**

<table>
<thead>
<tr>
<th>A common and persistent Myth</th>
</tr>
</thead>
<tbody>
<tr>
<td>A <em>programming model in itself can not “Not Scale”</em></td>
</tr>
</tbody>
</table>

Some things that may impact scalability:

The tools you use (e.g. the compiler, libraries, etc.), or a mismatch between the system and the resource requirements

Or ... You
In this talk we cover the basics how to get good performance

If you follow these guidelines, you should expect decent performance

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

Ideas are easy and quick to implement

But some constructs are more expensive than others

If you write dumb code, you will likely get dumb performance

Just don’t blame OpenMP, please

*) It is fine to blame the weather, or politicians, or both though
How To Not Write Dumb OpenMP Code
About Single Thread Performance

<table>
<thead>
<tr>
<th>You <strong>have to</strong> pay 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, or 20 cores, or … ?</td>
</tr>
<tr>
<td><strong>Remember, scalability can mask poor performance</strong></td>
</tr>
<tr>
<td>(a slow code tends to scale better, but is often still slower)</td>
</tr>
</tbody>
</table>
The Basics for All Users

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

```
#pragma omp parallel for
{<for loop 1> }
```

**Parallel region cost repeatedly incurred**
No potential for the “nowait” clause

```
#pragma omp parallel
{
    #pragma omp for
    {<for loop 1> }
    #pragma omp for nowait
    {<for loop n> }
} // End of parallel region
```

**Parallel region cost incurred only once**
Potential for the “nowait” clause
**Identify opportunities to use the `nowait` clause**

*(a very powerful feature, but be aware of data races)*

**Use the `schedule` clause in case of load balancing issues**

**A good profiling tool is indispensable to find out more!**
An Example from Graph Analysis

Note: The Oracle Studio Performance Analyzer was used to generate this timeline
The Timeline using 4 Threads

There are gaps during the parallel execution
The gaps are always in the red coloured function
CPU Time Variations

The load imbalance increases

<table>
<thead>
<tr>
<th>Threads</th>
<th>Ratio Max/Min CPU Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.36</td>
</tr>
<tr>
<td>4</td>
<td>1.63</td>
</tr>
<tr>
<td>8</td>
<td>1.91</td>
</tr>
</tbody>
</table>

Mastering OpenMP Performance
The Issue

Fixed length loop

Irregular length loop

Irregular workload per k-iteration

Irregular control flow

#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;
                } // End of if-then
            } // End of if
        } // End of for-loop on vo
    } // End of parallel for-loop on k
Observations and the Solution

The #pragma omp for loop uses default scheduling

The default is implementation dependent, but is static scheduling

In this case this leads to load balancing issues

The solution: #pragma omp for schedule(dynamic)

Or an even better solution: #pragma omp for schedule(runtime)

Our setting: $ export OMP_SCHEDULE=“dynamic,25”
How Do You Know The Chunk Size Should Be 25?

Crystal Ball

Trial And Error
With Dynamic Scheduling Added*

A 1% slow down

(Near) linear scaling for up to 8 threads

The modified version is 3x faster than the original code

Search Time and Speed Up

- Original
- Modified dynamic
- Linear (Near) linear scaling for up to 8 threads

* OMP_SCHEDULE="dynamic,25"

Mastering OpenMP Performance
Really Important

Always Verify the Behaviour!
Before and After (8 Threads)

The load imbalance is indeed gone!
The Load Imbalance is Indeed Gone

CPU Time Distribution

<table>
<thead>
<tr>
<th>Threads</th>
<th>Version</th>
<th>Ratio Max/Min CPU Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Original</td>
<td>1.91</td>
</tr>
<tr>
<td>8</td>
<td>Modified</td>
<td>1.01</td>
</tr>
</tbody>
</table>

Static scheduling

Dynamic scheduling

8 threads

8 threads
Beyond the Basics, but Don’t Forget!

Every barrier matters
(needed, but please use them with care)

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

EVERYTHING Matters
(Amdahl’s Law: minor overheads get out of hand quickly)
More Things to Remember

Avoid nested parallelism
(the nested barriers really add up)

Consider tasking instead
(provides much more flexibility and finer granularity)

Consider taskloop as an alternative to a non-static loop iteration scheduling algorithm (e.g. dynamic)
When Do Things Get Harder?

<table>
<thead>
<tr>
<th>Memory access “just happens”</th>
</tr>
</thead>
<tbody>
<tr>
<td>There are however two things to watch out for:</td>
</tr>
<tr>
<td>Non-Uniform Memory Access (NUMA) and False Sharing</td>
</tr>
<tr>
<td>They have nothing to do with OpenMP as such and are a characteristic of using a shared memory architecture</td>
</tr>
<tr>
<td>They may impact the performance though</td>
</tr>
</tbody>
</table>
NUMA is covered extensively in the second part

So for now we will stop talking about NUMA and end part I with a topic called “False Sharing”
### What is False Sharing?

False Sharing occurs when multiple threads **modify** the same cache line at the same time.

This results in the cache line to move through the system (plus the additional cost of the cache coherence updates).

- *It is okay if this happens once in a while*
- *It is **not okay** if this happens very frequently*
#pragma omp parallel shared(a) 
{
    int TID = omp_get_thread_num();
    a[TID] = 0.0; // False Sharing
} // End of parallel region

With each update of “a”, the cache line moves to the cache of the thread executing the update
A Tuning Strategy

Follow the guidelines just given and in this order

Where applicable, give it a try

Always make a profile before and after

Details sometimes make all the difference

In many cases, a performance “mystery” is explained by NUMA effects, False Sharing, or both
Part II – OpenMP In The Real World
In this second part we want to show several tuning examples

These examples are all based upon real-life experiences

The first challenge is to recognize the underlying issue(s)

We start with some isolated cases

The last example is more elaborate and is all about NUMA (NUMA systems are common and our info may come handy)
More Examples
An Example – What is Wrong Here?

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

<more code>
```

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

<more code>
```

The second barrier is redundant

The single construct has an implied barrier already
(the second barrier will be quick, but still takes time)
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?

```
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 overhead from the “omp for”**
- **Performance benefit depends on the value of variables npoint and n**
- **If npoint is small, the overhead on the first loop is high**
- **If npoint is large, the overhead on the second loop is high**
The Sequence of Operations

-1 -1 ... -1 -1 -1

Barrier

value

npoint

0 ... npoint-1

npoint+1 ... n-1

Barrier
The Actual Operation Performed

0 \ldots \text{npoint}-1 \quad \text{npoint} \quad \text{npoint+1} \ldots \text{n-1}

-1 \quad -1 \quad \ldots \quad -1 \quad -1 \quad -1 \quad \text{value} \quad -1 \quad \ldots \quad -1 \quad -1

\text{0} \ldots \text{n-1}

-1 \quad -1 \quad \ldots \quad -1 \quad -1 \quad -1 \quad -1 \quad -1 \quad \ldots \quad -1 \quad -1

\text{Barrier}

\text{value} \quad \text{npoint}

The Idea
The Implementation

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

- Only one barrier
- Reduced parallel overhead
- Performance depends on the value of n only
We have just shown the basic hurdles to take

Each one of these speak for themselves

The complicating factor is that they may be mixed

Don’t panic though, be patient, and tackle them one by one

But be prepared for this to be an iterative process

Also, be prepared that sometimes good ideas turn out to be not such good ideas
NUMA from Beginning to End
<table>
<thead>
<tr>
<th>Memory is physically distributed, but logically shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared data is transparently accessible to all threads</td>
</tr>
<tr>
<td>You don’t know where the data is and it doesn’t matter</td>
</tr>
</tbody>
</table>

*Unless you care about performance ...*
A Generic, but very Common and Contemporary NUMA System
NUMA - The Developer’s View
NUMA - Local Versus Remote Access Times

My Thread Executes Here

Local Access (Fastest)

Remote Access (Slower)
Tuning for NUMA is about keeping threads and their data close

In OpenMP, a thread may be moved to the data

Not the other way round, because that is more expensive

The affinity constructs in OpenMP control where threads run

This is a powerful feature, but it is up to you to get this right
There are two NUMA related environment variables

**OMP_PLACES** defines where threads are allowed to execute

*Choices are: sockets, cores, threads, or hardware thread IDs*

**OMP_PROC_BIND** controls the mapping of threads onto places

*Choices are: master, close, spread*
An Example

# Use 16 threads:
export OMP_NUM_THREADS=16

# Use 2 sockets to place those threads:
export OMP_PLACES=sockets(2)

# Spread them as far apart as possible:
export OMP_PROC_BIND=spread

# Run the code:
./a.out
The **First Touch Placement Policy** allocates the data page in the memory closest to the thread accessing this page for the first time.

This policy is the default on Linux and other OSes.

It is the right thing to do for a sequential application.

But this may not work so well in a parallel application ...
First Touch works fine, but what if a single thread initializes most, or all of the data?

Then all the data ends up in the memory of a single node.

This increases access times for certain threads and may cause congestion at the memory controller.

Luckily, the solution is (often) surprisingly simple.
Parallelize the data initialization part!

```c
#pragma omp parallel for schedule(static)
for (int i=0; i<n; i++)
a[i] = 0;
```

Each thread has a slice of “a” in its local memory*

*) The allocation is on a virtual memory page basis
#pragma omp parallel for default(none) \
shared(m,n,a,b,c) schedule(static)
for (int i=0; i<m; i++)
{
    double sum = 0.0;
    for (int j=0; j<n; j++)
        sum += b[i][j]*c[j];
    a[i] = sum;
}

for (i=0; i<m; i++)
    for (j=0; j<n; j++)
        b[i][j] = i;

for (j=0; j<n; j++)
c[j] = 1.0;
### Is There Anything Wrong Here?

<table>
<thead>
<tr>
<th>Nothing wrong with this code</th>
</tr>
</thead>
<tbody>
<tr>
<td>But this code is not NUMA aware</td>
</tr>
<tr>
<td>The data initialization is sequential</td>
</tr>
<tr>
<td>Therefore, all data ends up in the memory of a single node</td>
</tr>
<tr>
<td>This is a more NUMA friendly data initialization:</td>
</tr>
</tbody>
</table>
```c
#pragma omp parallel default(none) \ 
    shared(m,n,a,b,c) private(i,j)
{
    #pragma omp for schedule(static)
    for (i=0; i<m; i++)
    {
        a[i] = -1957.0;
        for (j=0; j<n; j++)
            b[i][j] = i;
    } // End of omp for

    #pragma omp for schedule(static)
    for (j=0; j<n; j++)
    {
        c[j] = 1.0;
    }
} // End of parallel region
```
Two very useful tools to help understand the NUMA topology

**lscpu** displays many things, including details on the node(s)

**numactl** –H displays the nodes and the relative latency

(lscpu is part of util-linux and numactl is a standalone package)

*) There are more tools than shown here (e.g. numastat)
AMD EPYC “Naples” based server*

How can you tell what the NUMA topology is?

*) I know, this is an older AMD system 😊
The NUMA Nodes in the System

$ lscpu$

**NUMA node0 CPU(s):** 0–7, 64–71
**NUMA node1 CPU(s):** 8–15, 72–79
**NUMA node2 CPU(s):** 16–23, 80–87
**NUMA node3 CPU(s):** 24–31, 88–95
**NUMA node4 CPU(s):** 32–39, 96–103
**NUMA node5 CPU(s):** 40–47, 104–111
**NUMA node6 CPU(s):** 48–55, 112–119
**NUMA node7 CPU(s):** 56–63, 120–127

Output from “numactl –H”

<table>
<thead>
<tr>
<th>Node</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>10</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>16</td>
<td>10</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>10</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>10</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>16</td>
<td>10</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>16</td>
<td>16</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>7</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>10</td>
</tr>
</tbody>
</table>

*) The lay-out information has been slightly reformatted

2 hardware threads per core
There are 8 NUMA nodes

Each NUMA node has 8 cores

Each core has 2 hardware threads

In total there are 64 cores and 128 hardware threads

There are 2 clusters with ”remote” nodes (”16” and ”32”)

Each node has 3 neighbors with a longer access time (”16”) and 4 neighbors with an even longer access time (”32”)

---

This is the NUMA Structure
The Abstract System Topology (numactl –H)

Even longer access time ("32")

Remote Node

Remote Node

Remote Node

Remote Node

Center Node

Remote Node

Remote Node

Remote Node

Longer access time ("16")
Example – NUMA Node 0 (lscpu output)

<table>
<thead>
<tr>
<th>Memory</th>
<th>0</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>71</td>
</tr>
</tbody>
</table>

8 cores
16 hardware threads
All cores and hardware threads share the memory in the node
An OpenMP place defines a set where a thread may run

OpenMP supports abstract names for places: sockets, cores, and threads

These abstract names are preferred to be used

But sometimes more explicit control is needed

We will now show an example how to use the explicit controls
Example - Control the Mapping of Threads

The Goal

Distribute the OpenMP threads evenly across the cores and nodes

As an example, use the first hardware thread of the first two cores of all the nodes
Example - Control the Mapping of Threads

Node 0
0 1

Node 1
8 9

Node 2
16 17

Node 3
24 25

Node 4
32 33

Node 5
40 41

Node 6
48 49

Node 7
56 57

Mastering OpenMP Performance
Expands to the first hardware thread on the first 2 cores on each node:
{0}, {8}, {16}, {24}, {32}, {40}, {48}, {56}, {1},{9},{17},{25},{33},{41},{49},{57}

$ export OMP_PLACES={0}:8:8,{1}:8:8
$ export OMP_PROC_BIND=close
$ export OMP_NUM_THREADS=16
$ ./a.out

Note: Setting OMP_DISPLAY_ENV=verbose is your friend here!
Performance – 2 Sockets/64 Cores

Performance of the matrix-vector algorithm (4096x4096)

- Without First Touch
- With First Touch

**Much better scaling**
(35x using 64 threads)

**Very poor scaling**

First Touch improves the performance by a factor of 22x

Oracle Linux with the gcc compiler
NUMA balancing on
NUMA Case Study - Takeaways

<table>
<thead>
<tr>
<th>Data and thread placement matters (a lot)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Important to leverage First Touch Data Placement</td>
</tr>
<tr>
<td>OpenMP has elegant, yet powerful, support for NUMA</td>
</tr>
<tr>
<td>More NUMA support has been added in OpenMP 5.0 and 5.1!</td>
</tr>
</tbody>
</table>
### Summary - The Tuning Strategy

#### Think Ahead

<table>
<thead>
<tr>
<th>Advice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always use a profiling tool to guide you</td>
</tr>
<tr>
<td>Never forget to tune for serial performance</td>
</tr>
<tr>
<td>Consider your data structures</td>
</tr>
<tr>
<td>Find and address the low hanging fruit</td>
</tr>
<tr>
<td>Do not forget to consider data placement</td>
</tr>
</tbody>
</table>
Thank You And ... Stay Tuned!

Bad OpenMP
Does Not Scale

Ruud van der Pas
Webinar, March 18, 2021