

Getting OpenMP Up To Speed

Ruud van der Pas

Distinguished Engineer

Oracle Linux and Virtualization Engineering

Oracle, Santa Clara, CA, USA

Guest Lecture

02614 High-Performance Computing Class

DTU, January 12, 2022

Agenda

<i>Part I - Tips and Tricks</i>	09:00 - 09:45
<i>A well deserved break</i>	09:45 - 10:00
<i>Part II - The Joy of Computer Memory</i>	10:00 - 10:45
<i>Q (and some A)</i>	10:45 - 11:00

Your OneStop Place for OpenMP

<https://www.openmp.org>

The screenshot shows the OpenMP website homepage. At the top, there is a navigation bar with links for Home, Specifications, Community, Resources, News & Events, About, and a search icon. Below the navigation bar, a large banner features the text "OpenMP 5.2 Released with Improvements and Refinements". A subtext below it says "Syntax is refined and several API features...". A blue "READ MORE" button is visible. To the right of the banner, a large callout box contains the text "A printed copy of the 5.2 OpenMP specs is also available on Amazon (669 pages, 1.6 kg)". Below the banner, there is a section titled "Latest News" with several news items. One news item is highlighted with a pink circle and contains the text "OpenMP 5.2 Released". Another news item in the circle is "OpenMP API Speeds Up Autonomous Driving Codes". On the far right, there is a sidebar with a tweet from the OpenMP ARB (@OpenMP_ARB) about the availability of the 5.2 API specification on Amazon.

Home Specifications Community ▾ Resources ▾ News & Events ▾ About ▾ Q

OpenMP 5.2 Released with Improvements and Refinements

Syntax is refined and several API features...

[READ MORE](#)

A printed copy of the 5.2 OpenMP specs is also available on Amazon (669 pages, 1.6 kg)

Latest News

The Pawsey Supercomputer Research Centre joins

OpenMP
OpenMP 5.2 Released

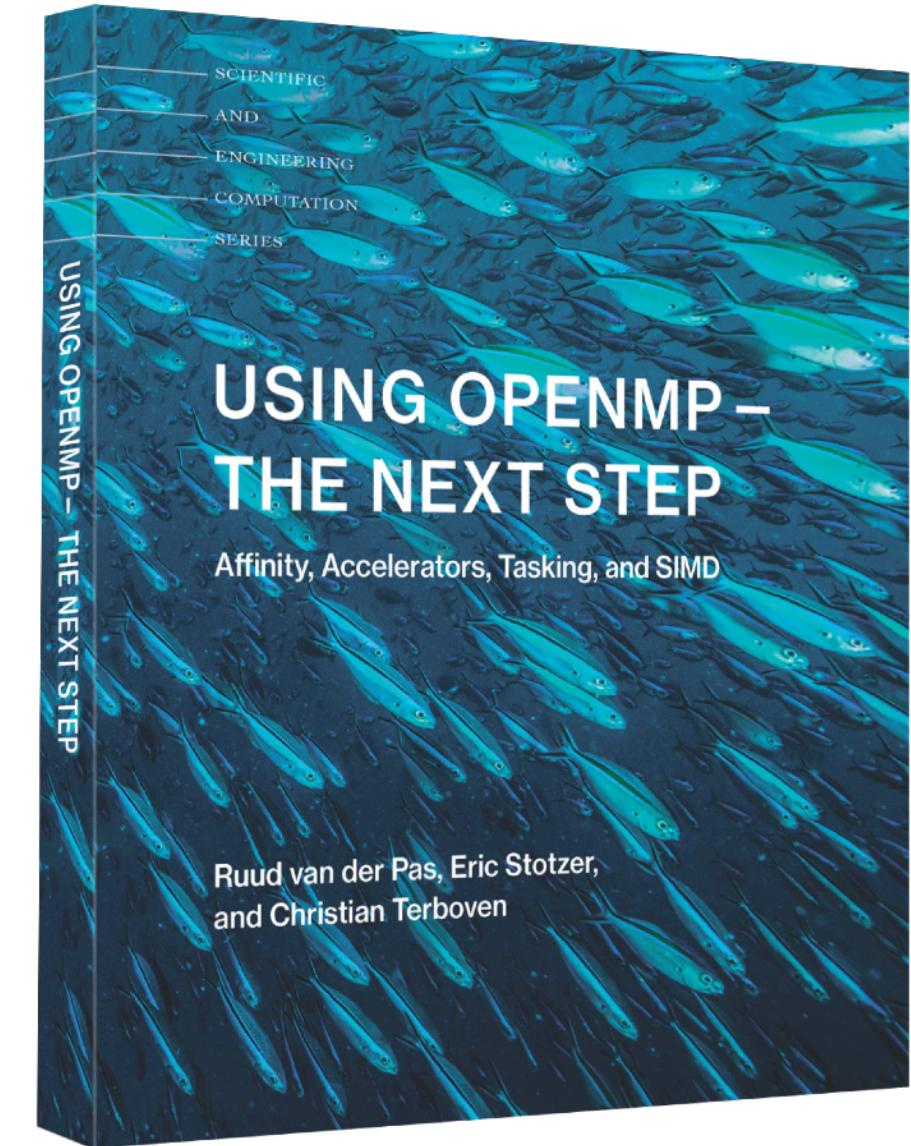
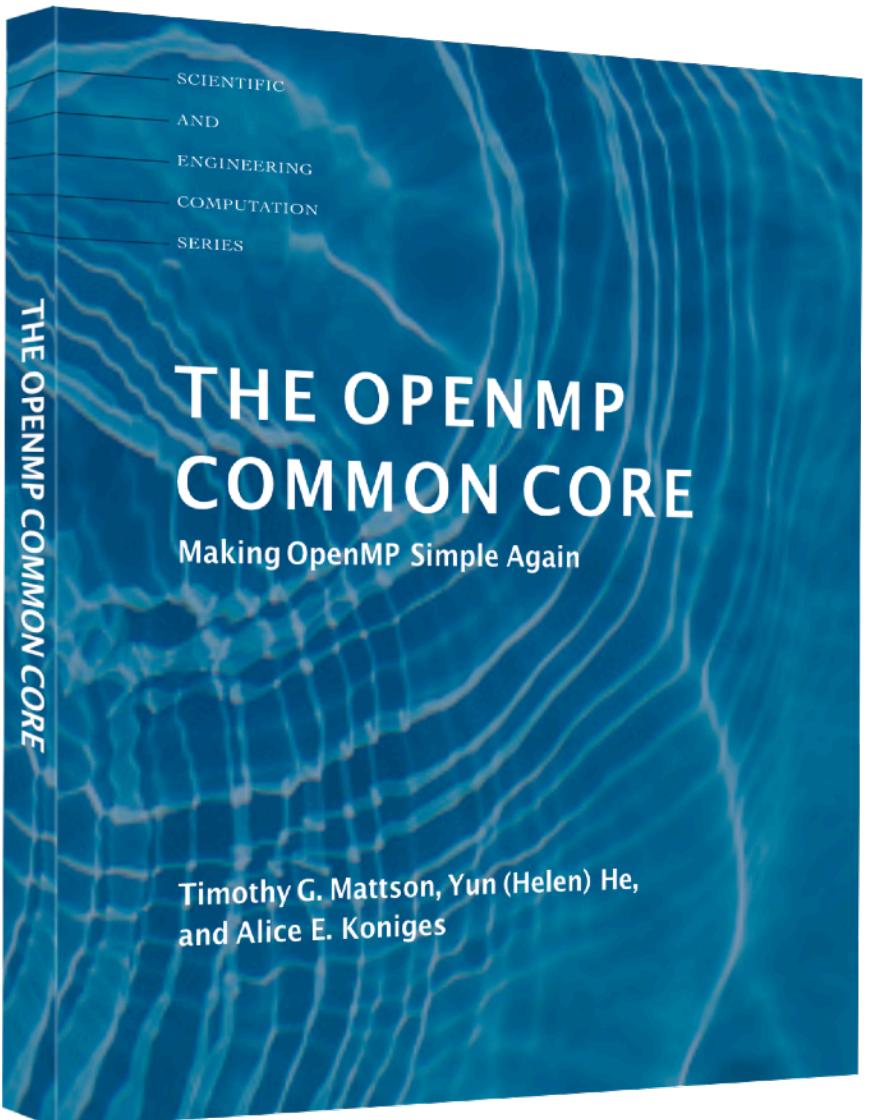
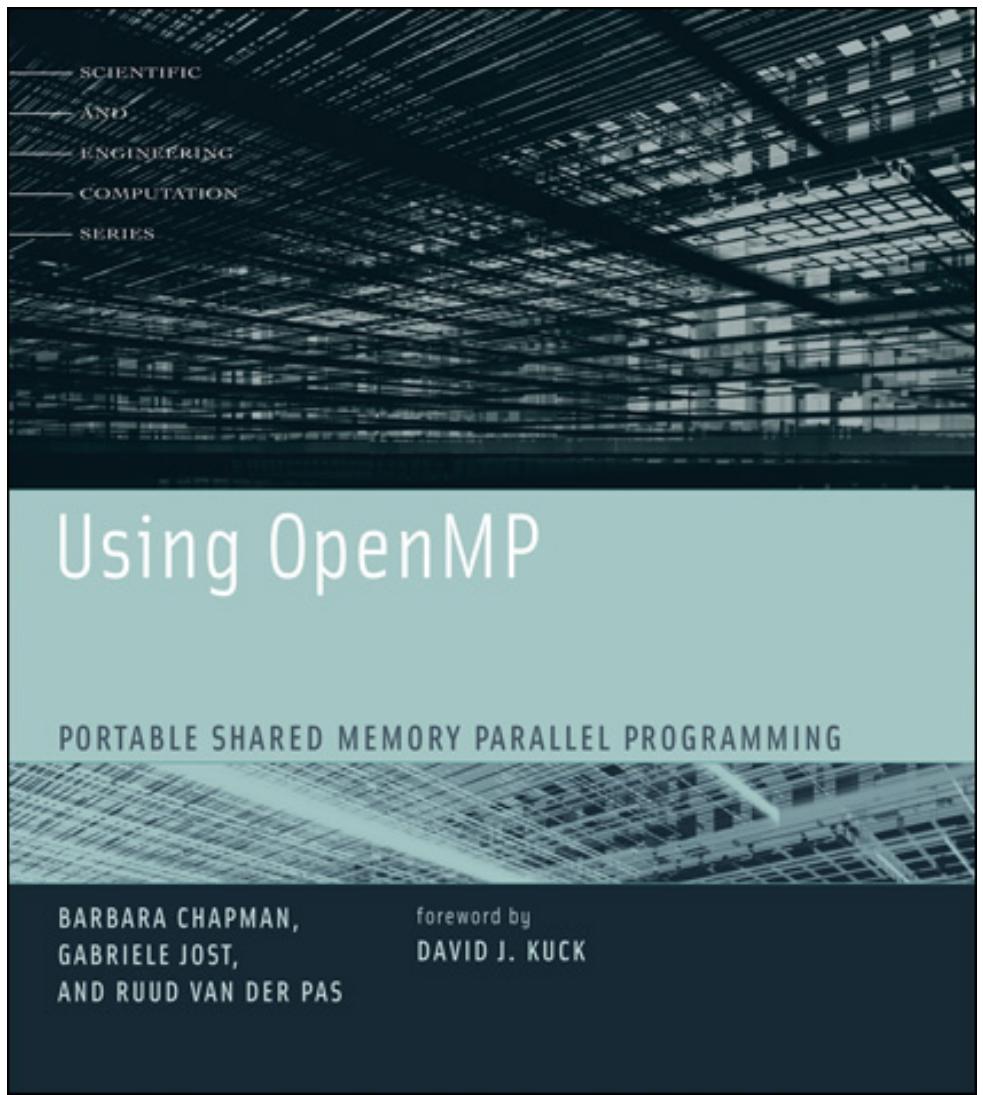
OpenMP ARB Releases OpenMP 5.2 Version 5.2 refactors the OpenMP specification to improve consistency

BLOG
OpenMP API Speeds Up Autonomous Driving Codes

OpenMP ARB @OpenMP_ARB Did you know that you can get a softcover version of the #OpenMP 5.2 API specification from Amazon for a ridiculous price? Go have a look at:amazon.com/Application-Pr... \$ 15#HPC #Embedded

 DTU 

Food for the Eyes and Brains



*OpenMP 2.5 and intro
Parallel Computing*

*Covers the OpenMP
Basics to get started*

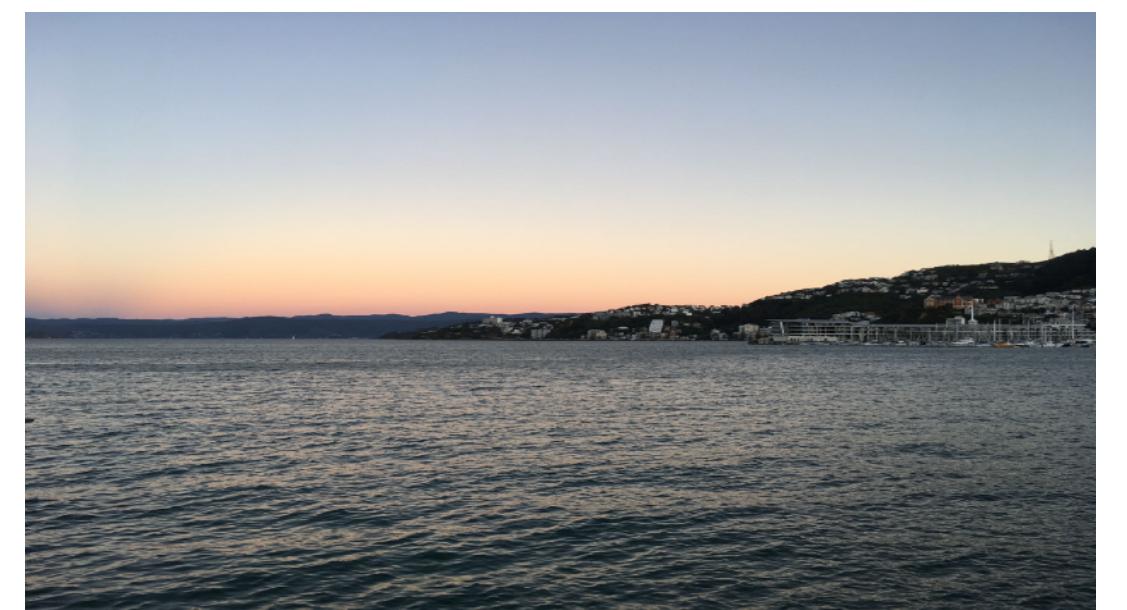
*Focus on the Advanced
Features*

*What goes on under
the hood*

OpenMP®

DTU

Part I - Tips and Tricks



OpenMP and Performance

You can get good performance with OpenMP

And your code will scale

If you do things in the right way

Easy -ne Stupid

The OpenMP Performance Court

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*

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

If you write dumb code, you ~~probably~~ will get dumb performance

*Just don't blame OpenMP, please**

My Preferred Tuning Strategy

In terms of complexity, use the most efficient algorithm

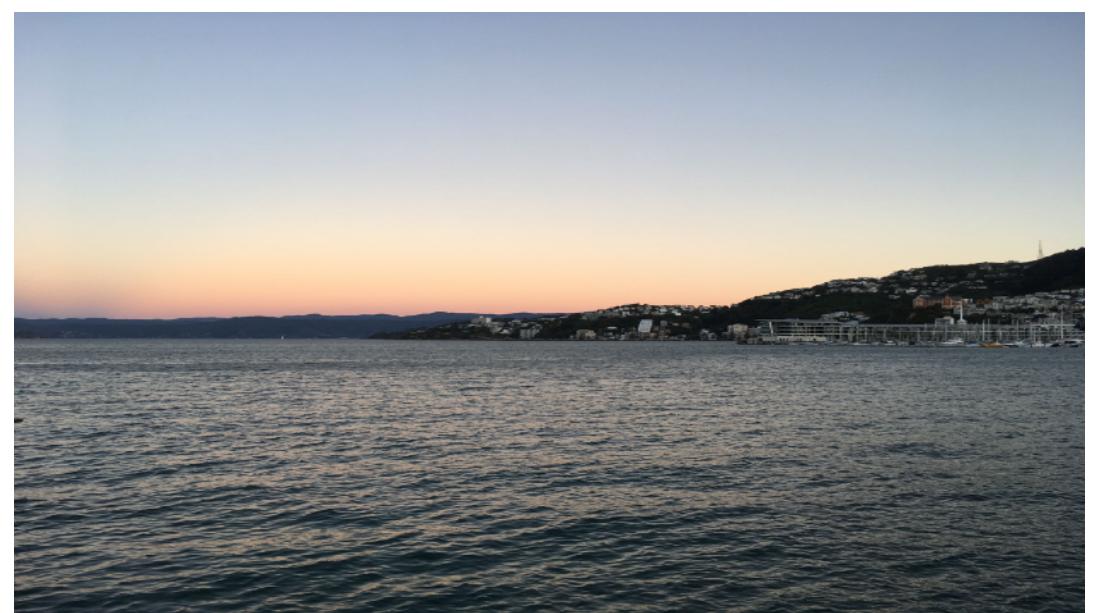
Select a profiling tool

*Find the highest level of parallelism
(this should however provide enough work to use many threads)*

*Use OpenMP **in an efficient way***

Be prepared to have to do some performance experiments

Things You Need To Know



About Caches

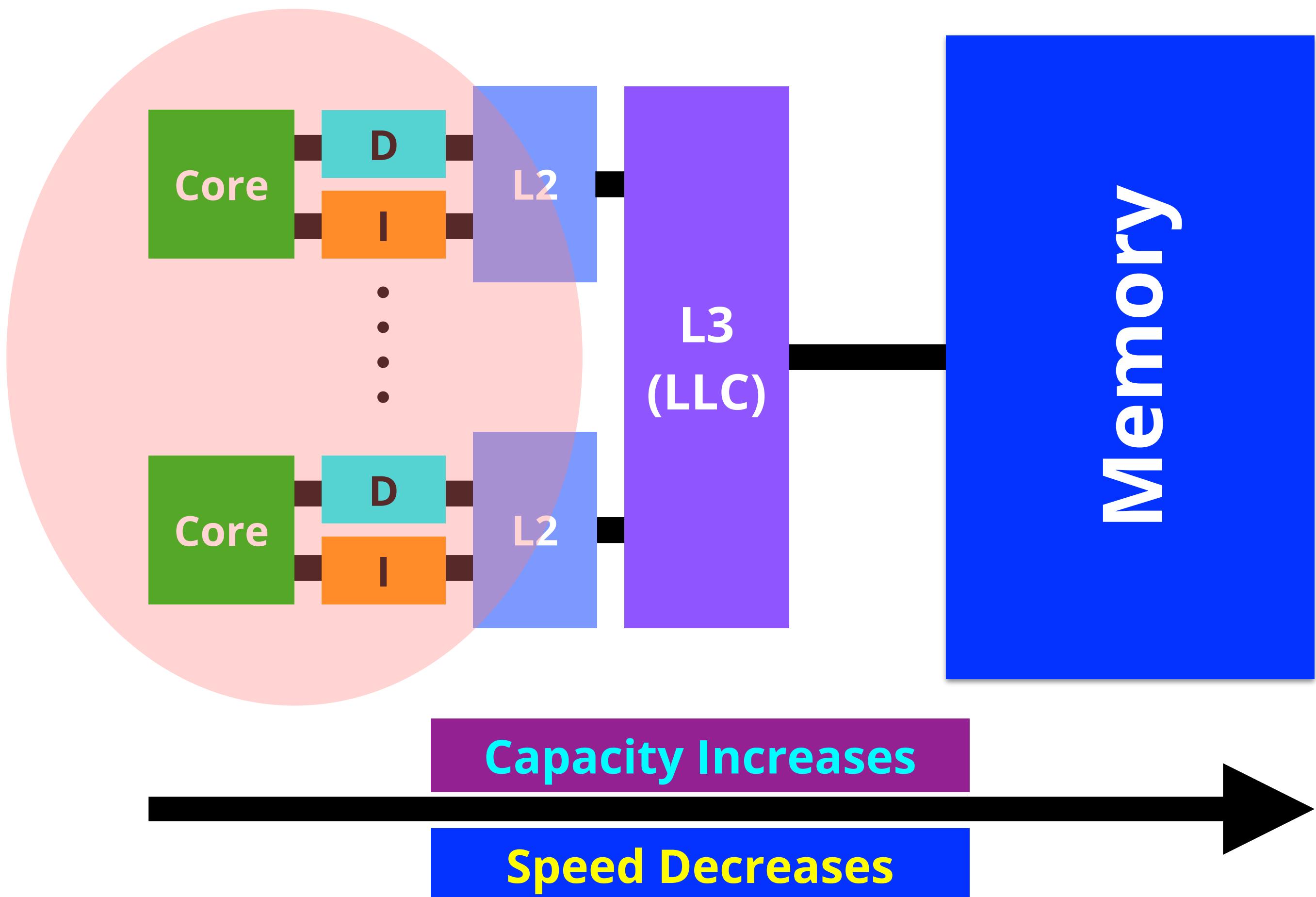
Caches are fast buffers, used for data and instructions

For cost and performance reasons, a modern processor has a hierarchy of caches

Some caches are private to a core, others are shared

Let's look at a typical example

A Typical Memory Hierarchy



*The unit of transfer
is a “cache line”*

*A cache line contains
multiple elements*

About Cores and Hardware Threads

A core may, or may not, support hardware threads

This is part of the design

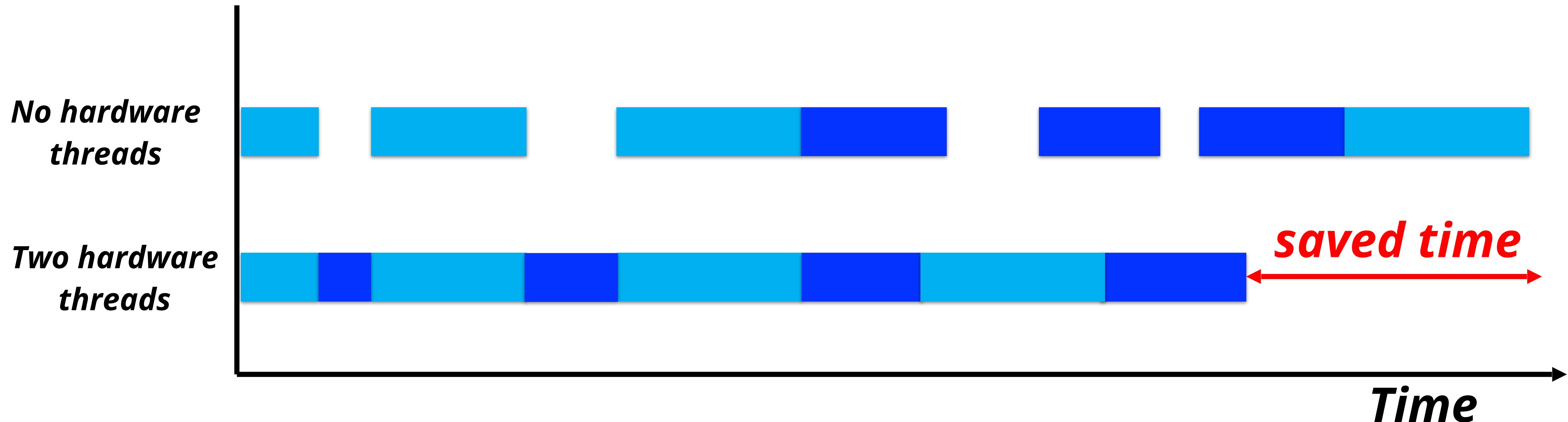
These hardware threads may accelerate the execution of a single application, or improve the throughput of a workload

The idea is that the pipeline is used by another thread in case the current thread is idle

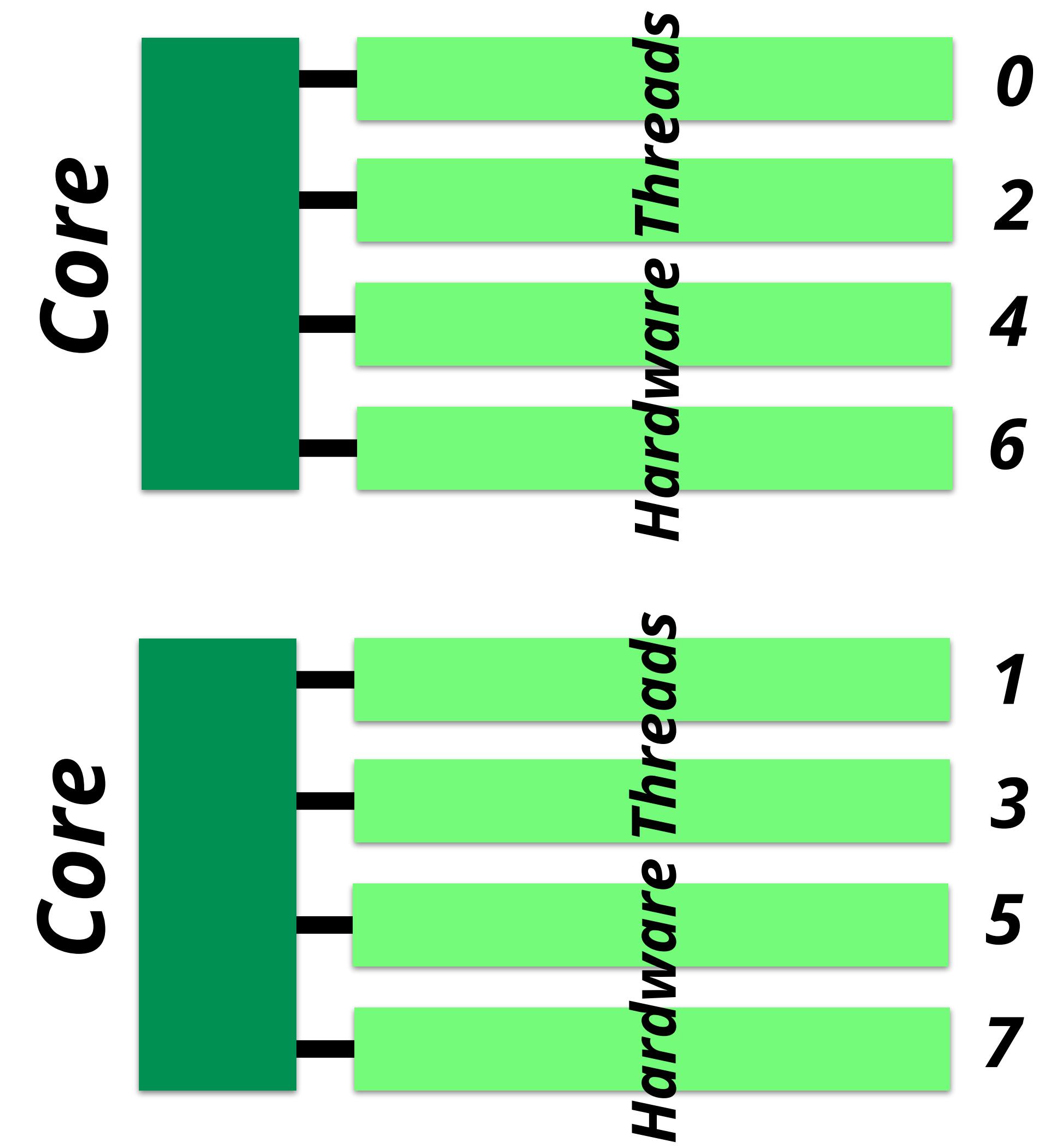
Each hardware thread has a unique ID in the system



How Hardware Threads Work



Hardware Thread IDs



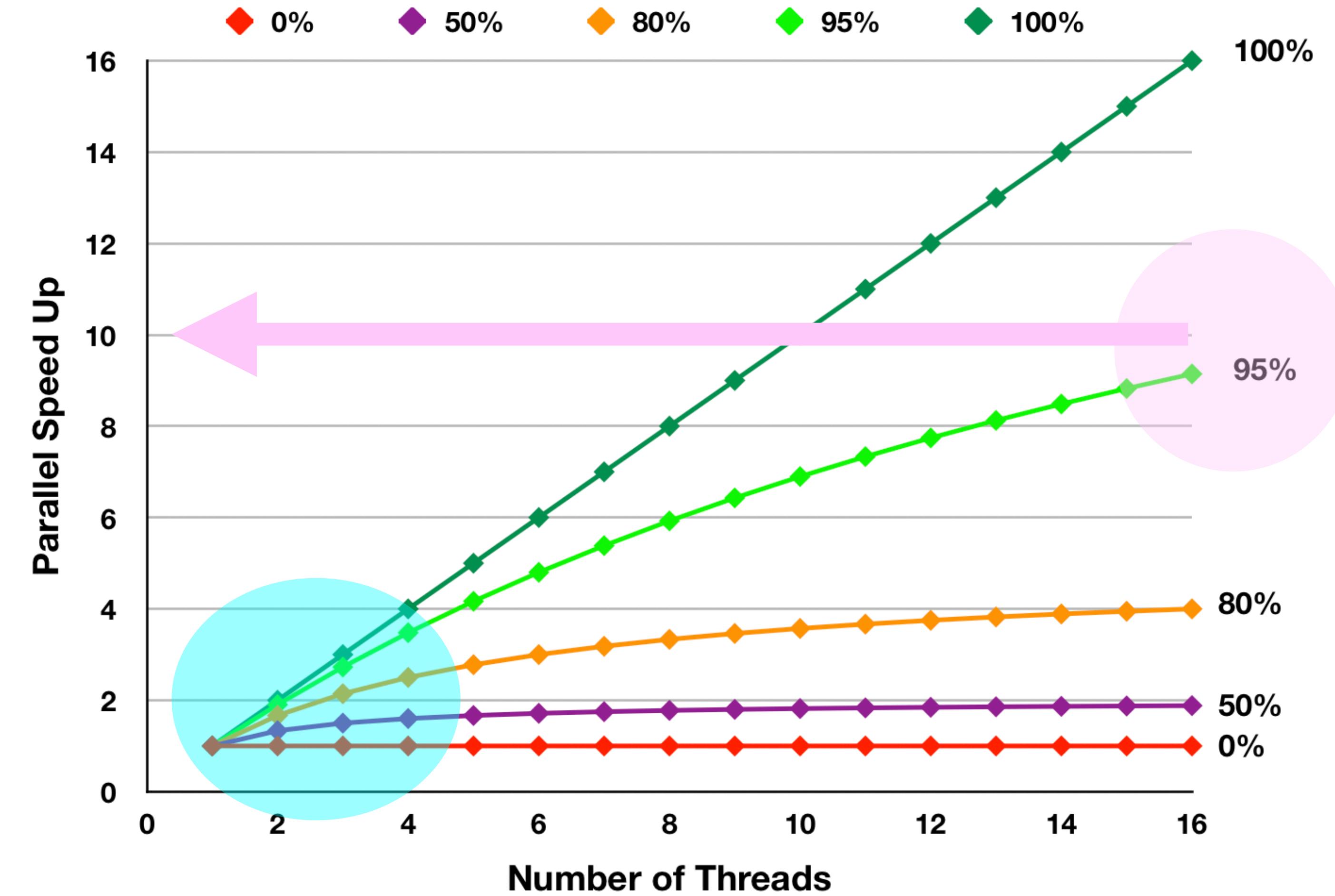
Amdahl's Law

Suppose your application needs 100 seconds to run

If 80% of this run time can execute in parallel, the time using 4 threads is $80/4+20 = 40$ seconds

This means that your program is 2.5x faster, not 4x

Amdahl's Law Using 16 Threads



Morale

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

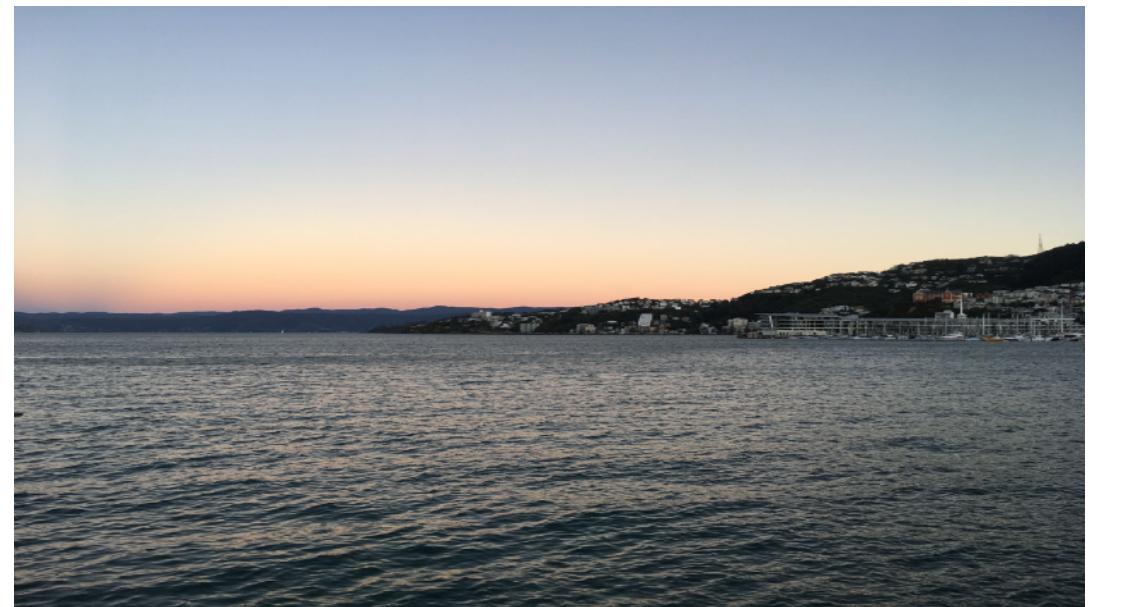
About Single Thread Performance and Scalability

You *have to pay* attention to single thread performance

Why? If your code performs badly on 1 core, what do you think will happen on 10 cores, 20 cores, ... ?

Scalability can mask poor performance!
(a slow code tends to scale better ...)

How To Not Write Dumb OpenMP Code



The Basics For All Users

Do not parallelize what does not matter

Never tune your code without using a profiling tool

***Do not share data unless you have to
(in other words, use private data as much as you can)***

Think BIG

(maximize the size of the parallel regions)

One “parallel for” is fine. More, back to back, is EVIL.



The Wrong and Right Way Of Doing Things

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

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

*Parallel region overhead repeated “n” times
No potential for the “nowait” clause*

*Parallel region overhead only once
Potential for the “nowait” clause*

More Basics

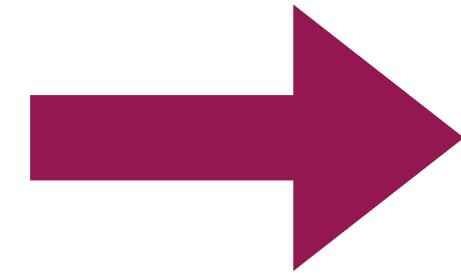
*Every barrier matters
(and please use them carefully)*

*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

```
#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 because the single construct has an implied barrier already (this second barrier will still take time though)

More Things to Consider

*Identify opportunities to use the **nowait** clause*

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

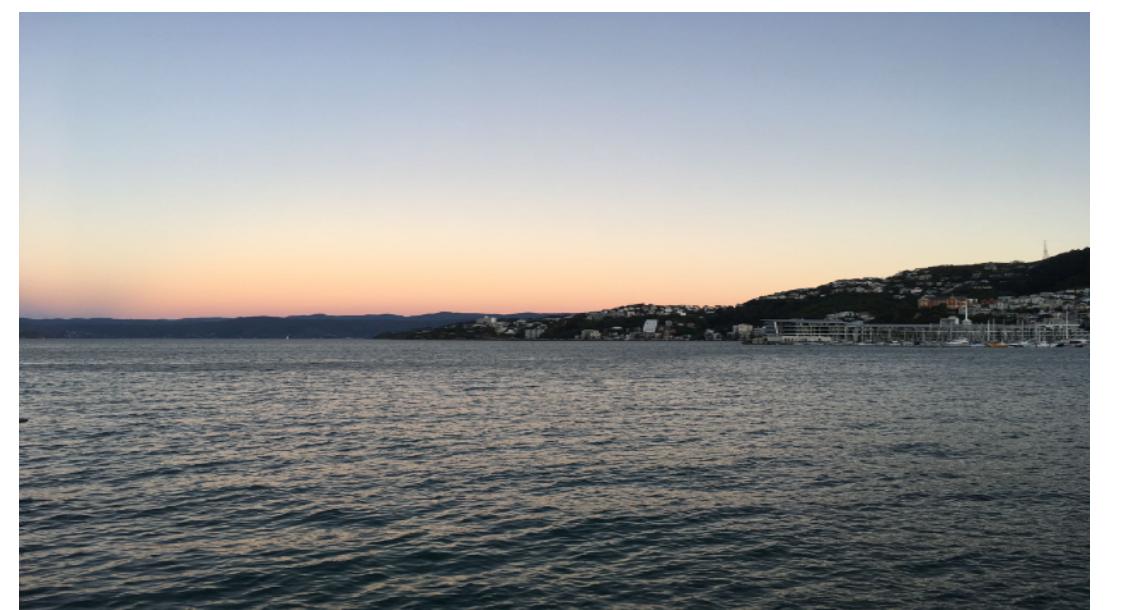
*Use the **schedule** clause in case of load balancing issues*

*Avoid nested parallelism
(the nested barriers really add up)*

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



Case Study - Do More Work and Save Time



A Very Time Consuming Part in the Code

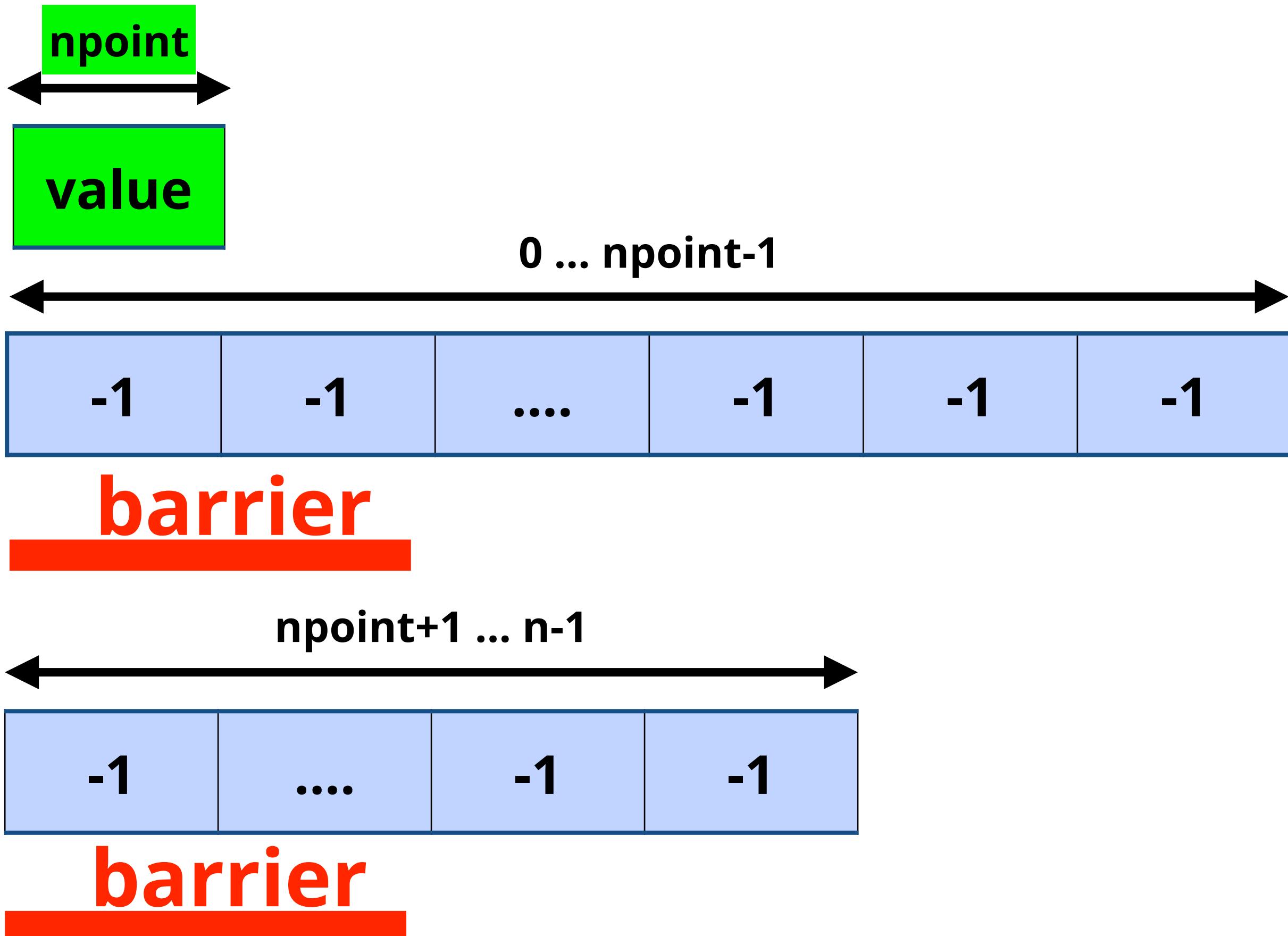
```
a[npoint] = value;  
#pragma omp parallel ...  
{  
    #pragma omp for  
    for (int64_t k=0; k<npoint; k++)  
        a[k] = -1;  
    #pragma omp for  
    for (int64_t k=npoint+1; k<n; k++)  
        a[k] = -1;  
  
    <more code>  
}  
} // End of parallel region
```

So What Is Wrong With This Then?

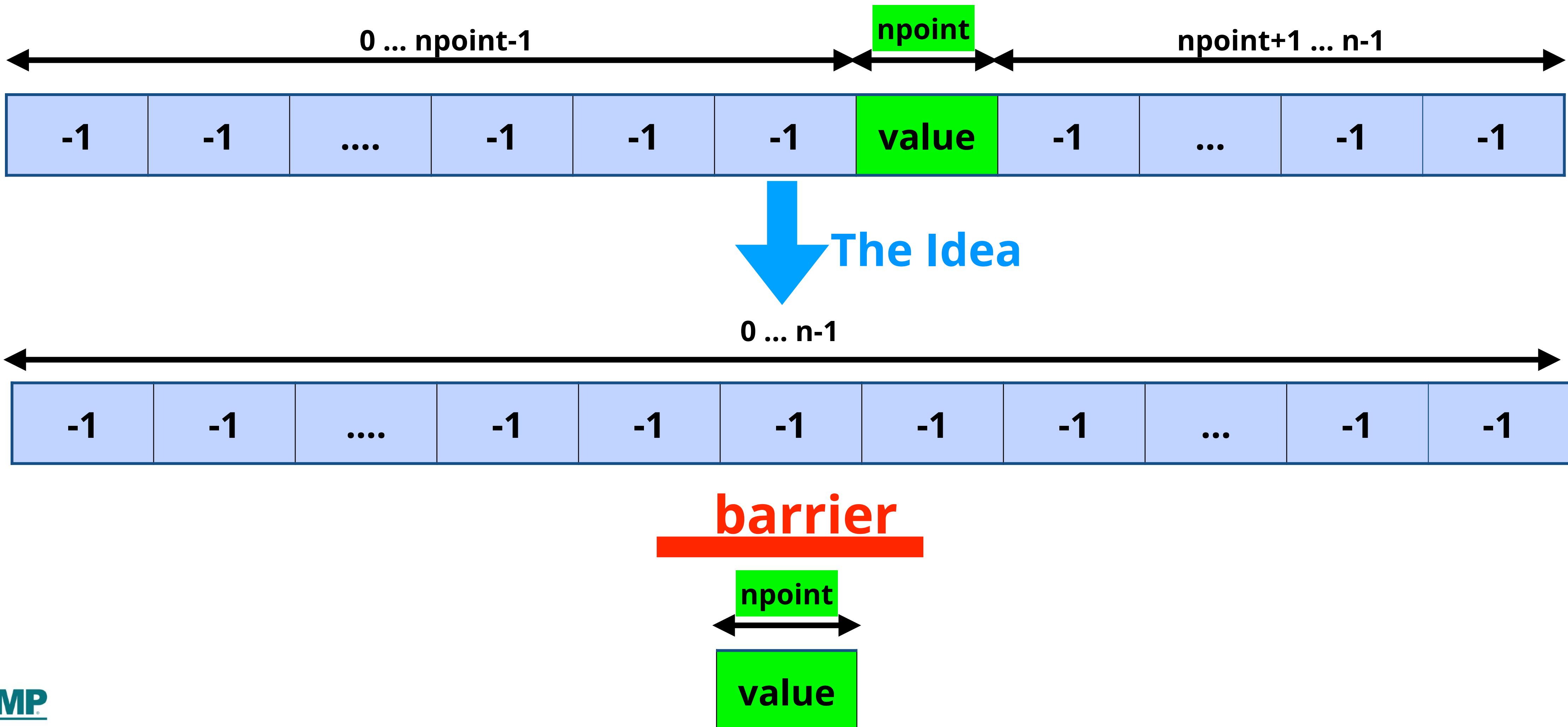
```
a[npoint] = value;  
#pragma omp parallel ...  
{  
    #pragma omp for  
    for (int64_t k=0; k<npoint; k++)  
        a[k] = -1;  
    #pragma omp for  
    for (int64_t k=npoint+1; k<n; k++)  
        a[k] = -1;  
  
<more code>  
  
} // End of parallel region
```

- ✓ *There are 2 barriers*
- ✓ *Two times the serial and parallel overhead*
- ✓ *Performance benefit depends on the value of variables “npoint” and “n”*

The Sequence Of Operations



The Final Result



The Modified Code

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

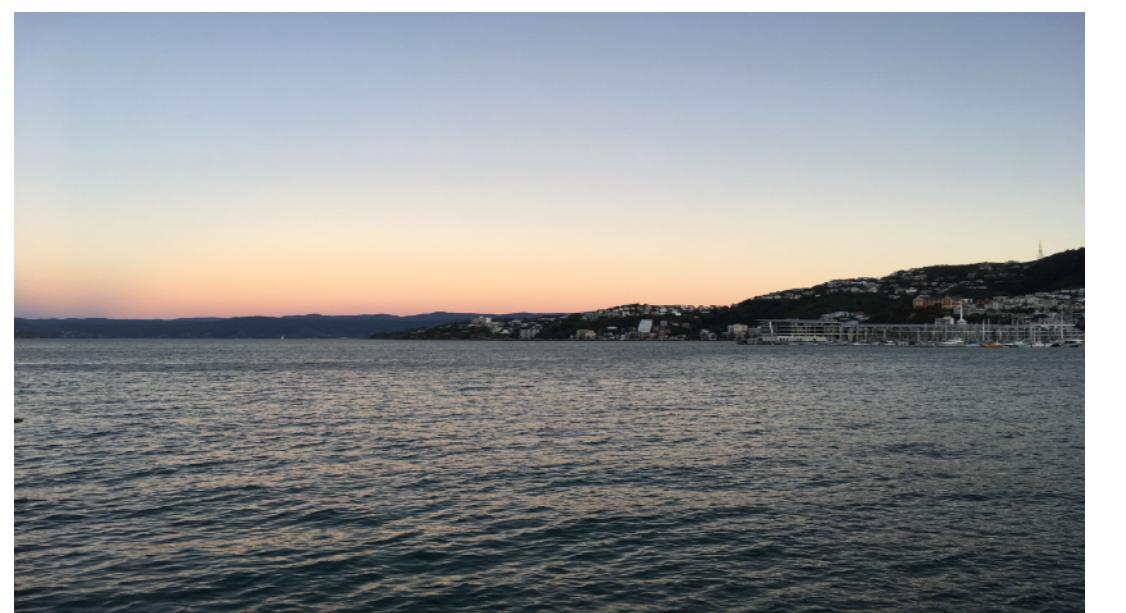
    #pragma omp single nowait
    {a[npoint] = value; }

    <more code>

} // End of parallel region
```

- ✓ **Only one barrier**
- ✓ **One time the serial and parallel overhead**
- ✓ **The performance benefit depends on the value of variable "n" only**

Case Study - Graph Analysis



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* — Repeat
- *Verify the result is a tree* —————— <n> times

For the benchmark score, only the search time matters

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)*

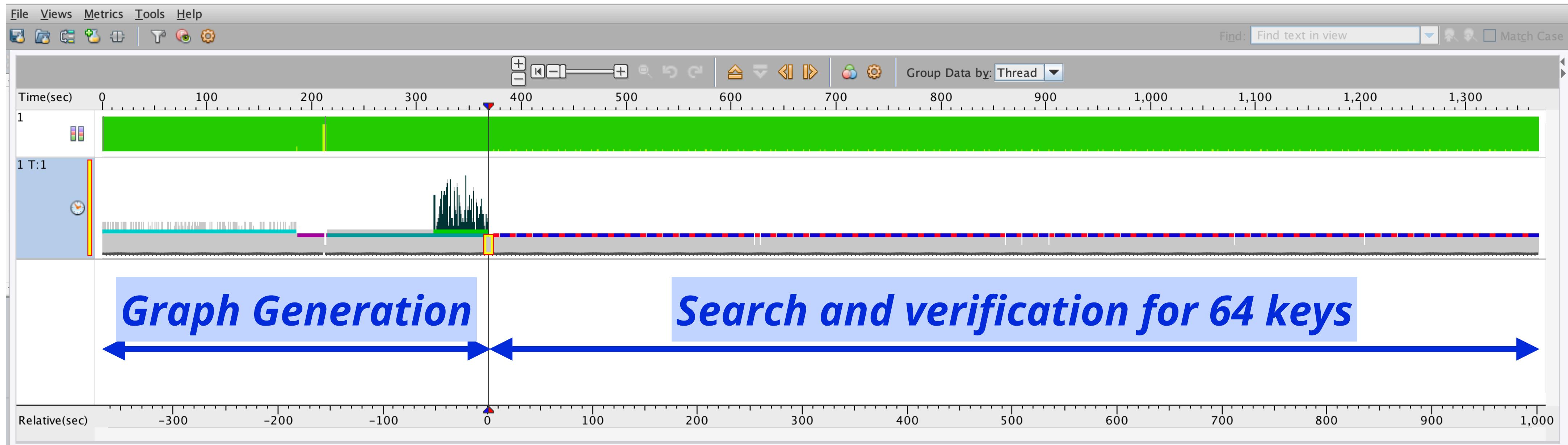
All experiments were conducted in the Oracle Cloud ("OCI")

Used a VM instance with 8 Intel Skylake cores (16 threads)

The Oracle Linux OS + gcc were used to build and run the jobs

The Oracle Performance Analyzer was used to make the profiles

The Dynamic Behaviour

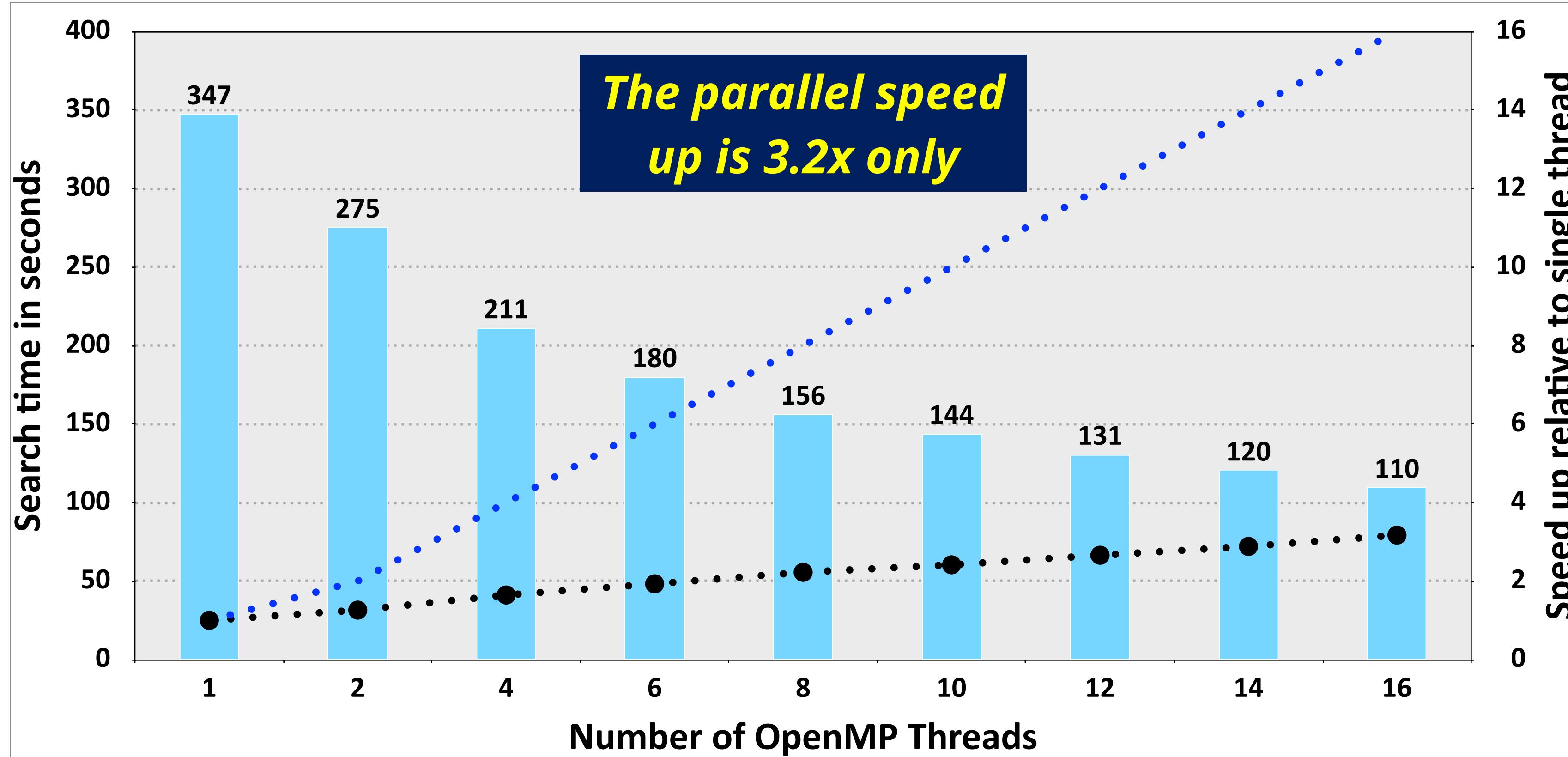


Search Verification

OpenMP

DTU

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 VM with 8 Intel Xeon Platinum 8167M CPU @ 2.00GHz ("Skylake") cores, 16 hardware threads

Action Plan

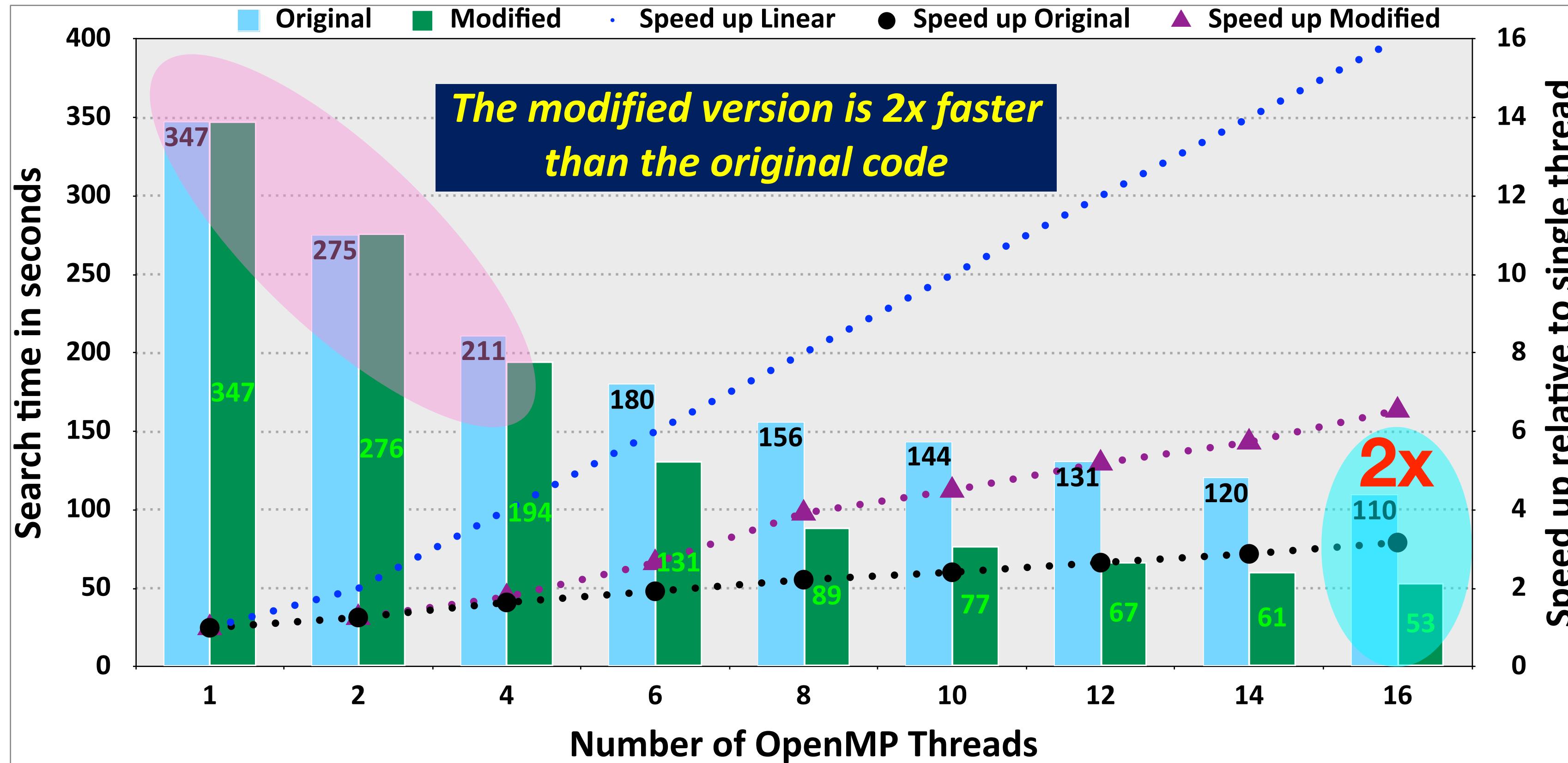
Used a profiling tool to identify the time consuming parts

Found several opportunities to improve the OpenMP part

These are actually shown earlier in this talk

Although simple changes, the improvement is substantial:

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 Tuning This Code?



OpenMP

DTU


Are We Done Yet?

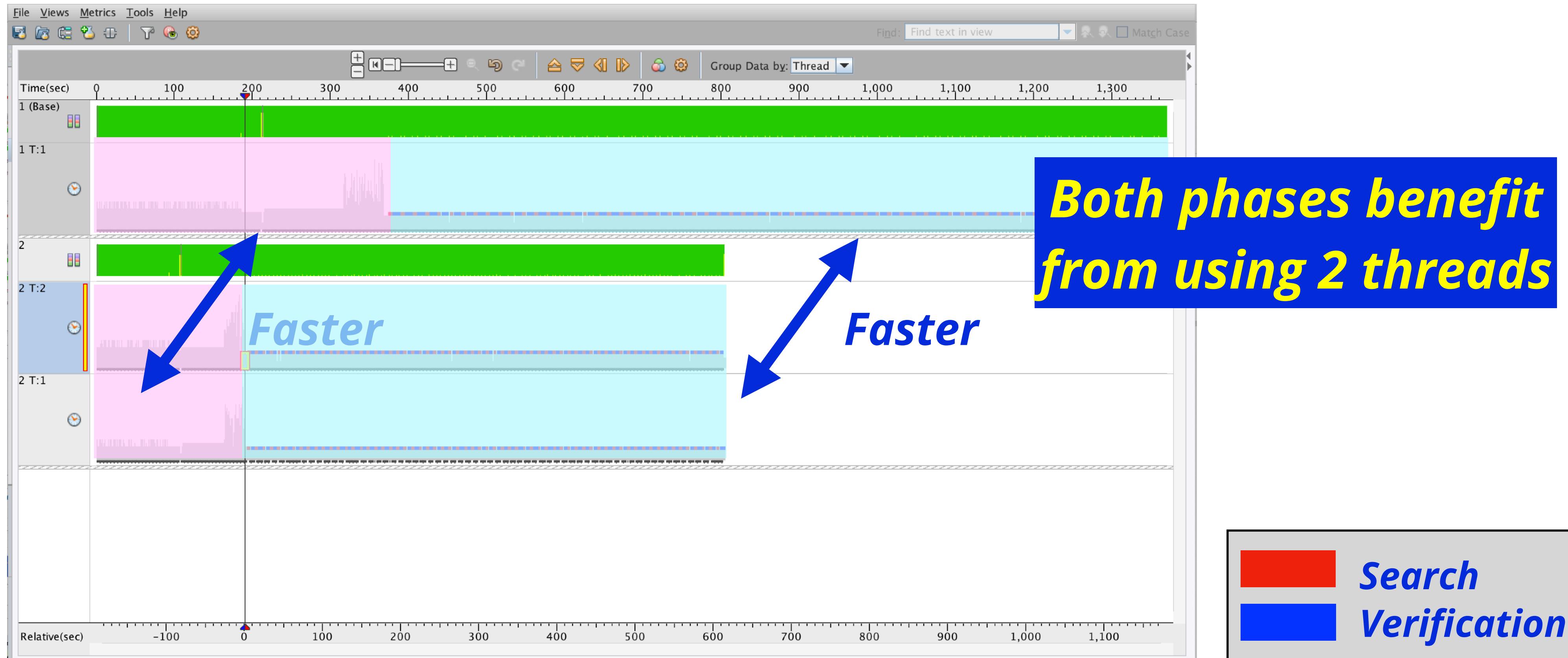
The 2x reduction in the search time is encouraging

The efforts to achieve this have been limited

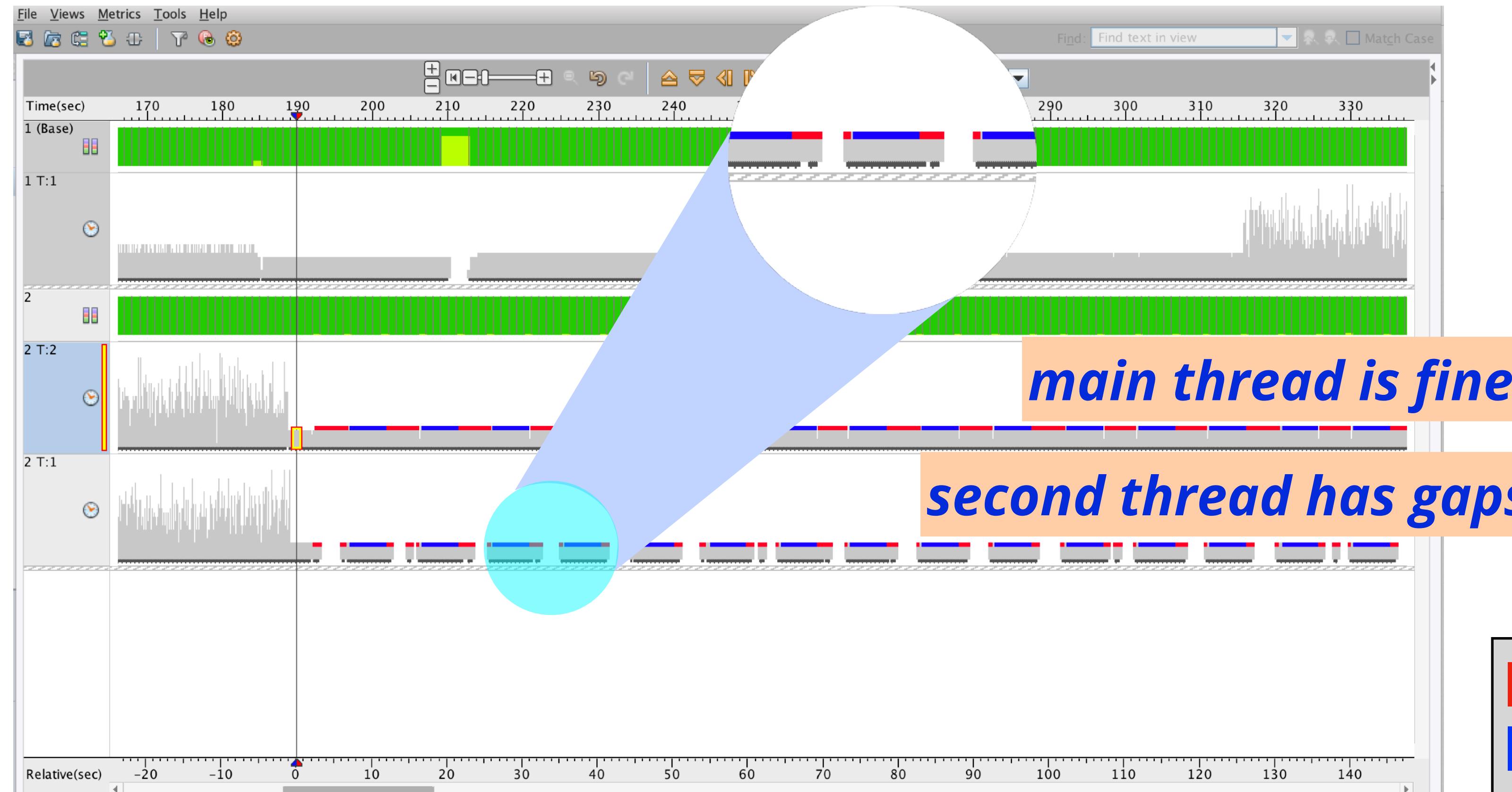
The question is whether there is more to be gained

Let's look at the dynamic behaviour of the threads:

A Comparison Between 1 And 2 Threads

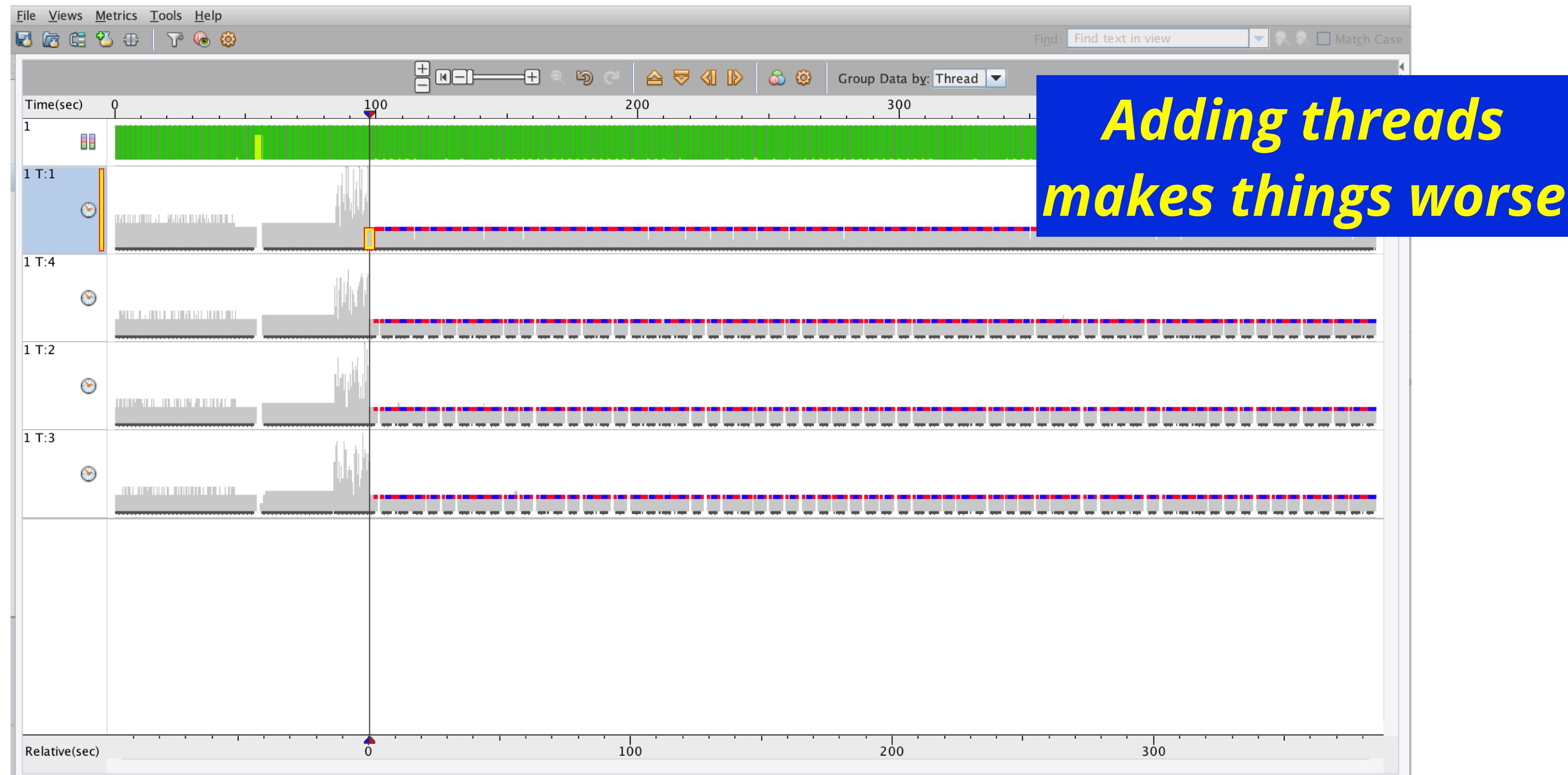


Zoom In On The Second Thread



OpenMP

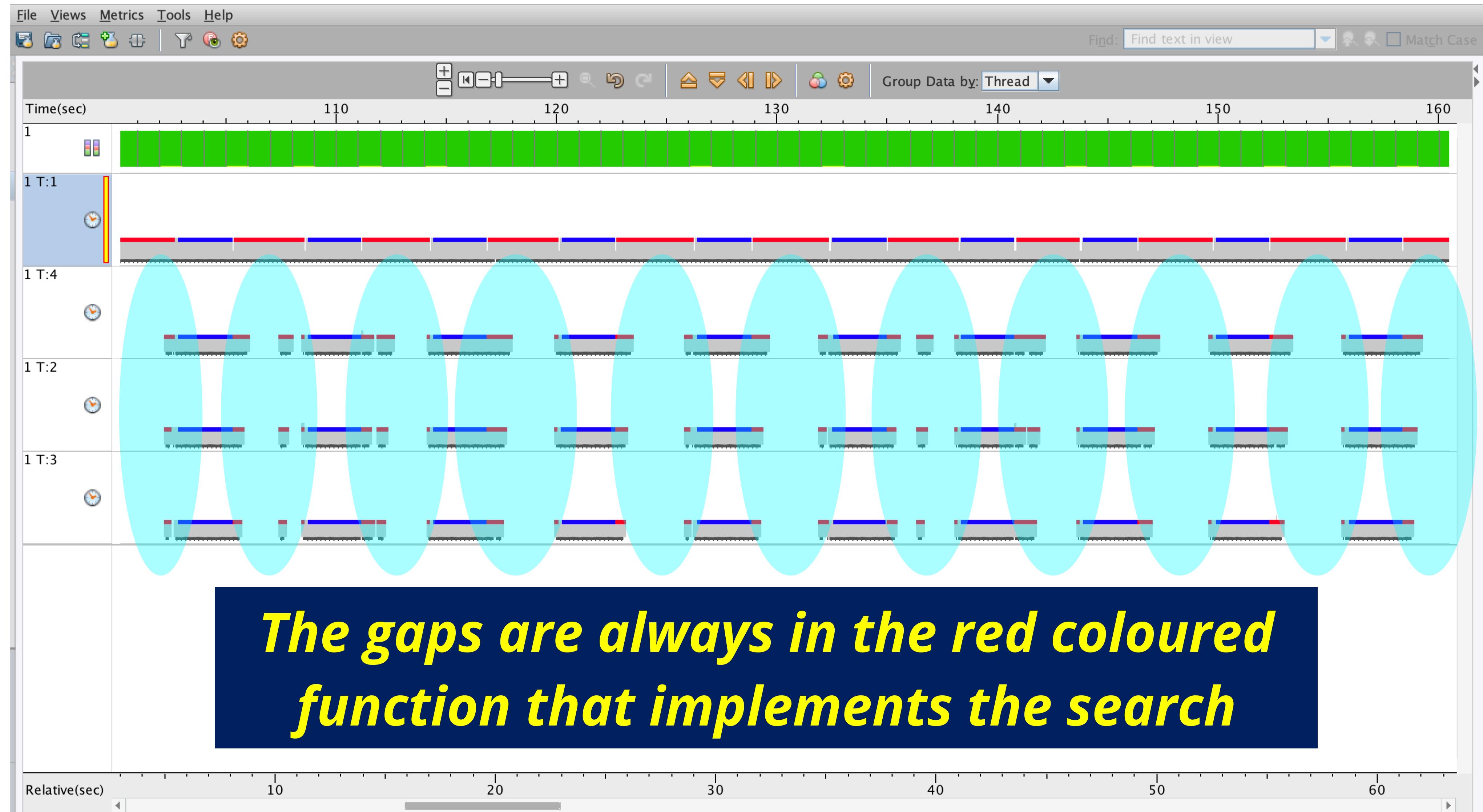
How About 4 Threads?



OpenMP

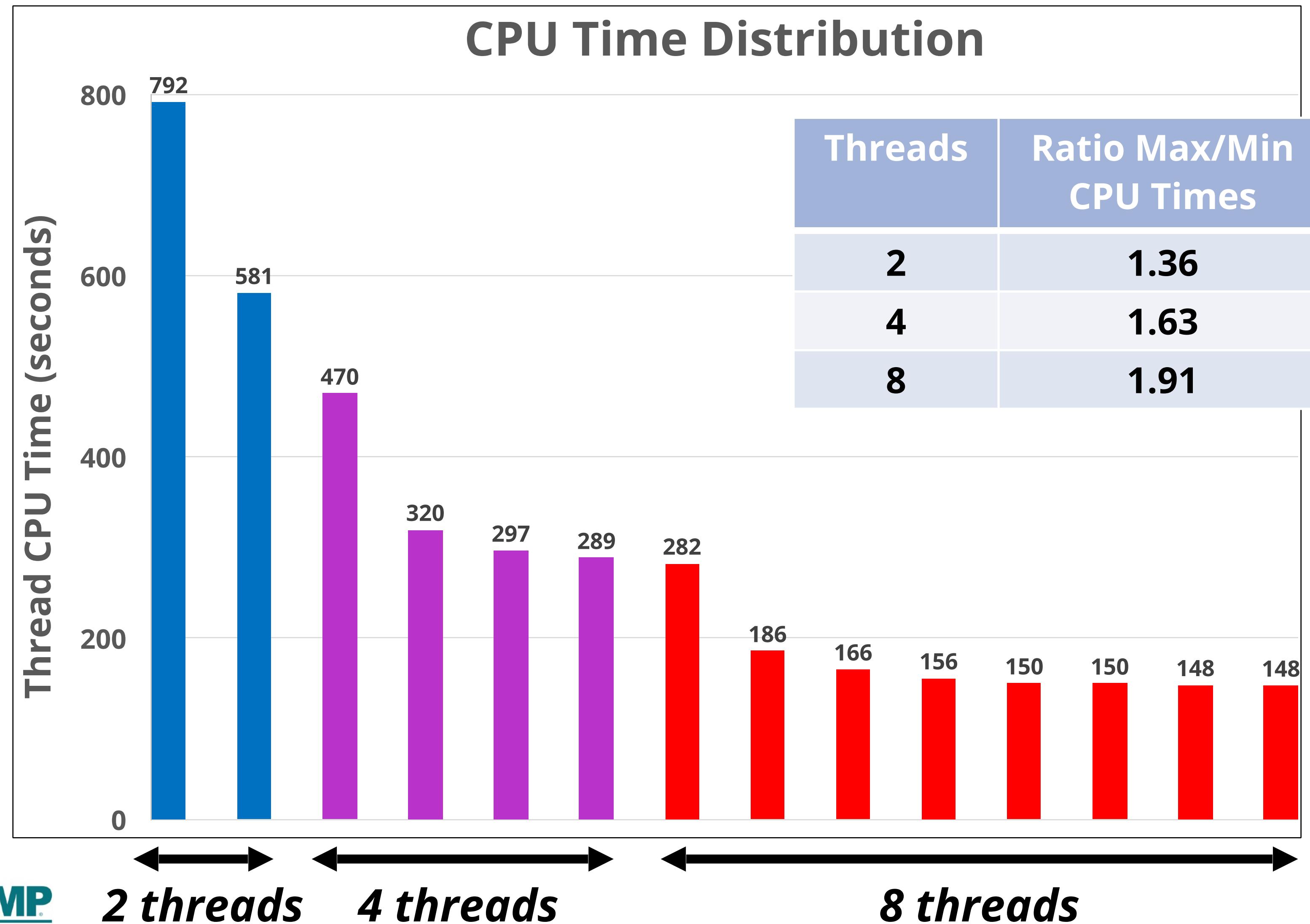


Zoom In Some More



OpenMP

CPU Time Variations



The load imbalance increases as the thread count goes up

The Issue

```
#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 = xad[vo];
        if (bfs_tree[j]) {
            if (int64_cas(&nbuf[kbuf], nbuf[0], kbuf)) {
                if (kbuf < BUF_LEN) {
                    nbuf[kbuf] = j;
                } else {
                    int64 assert();
                    for (int64_t vk = 0; vk < AD_LEN; ++vk)
                        vlist[vk] = nbuf[0];
                    nbuf[0] = j;
                    kbuf = 1;
                } // End of if-then-else
            } // End of if
        } // End of if
    } // End of for-loop on vo
} // End of parallel for-loop on k
```

Fixed length loop

Irregular length loop

Irregular control flow

Irregular workload per k-iteration

Observations and the Solution

The `#pragma omp for` loop uses default scheduling

The default is implementation dependent, but is “static” here

In this case, that 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"`

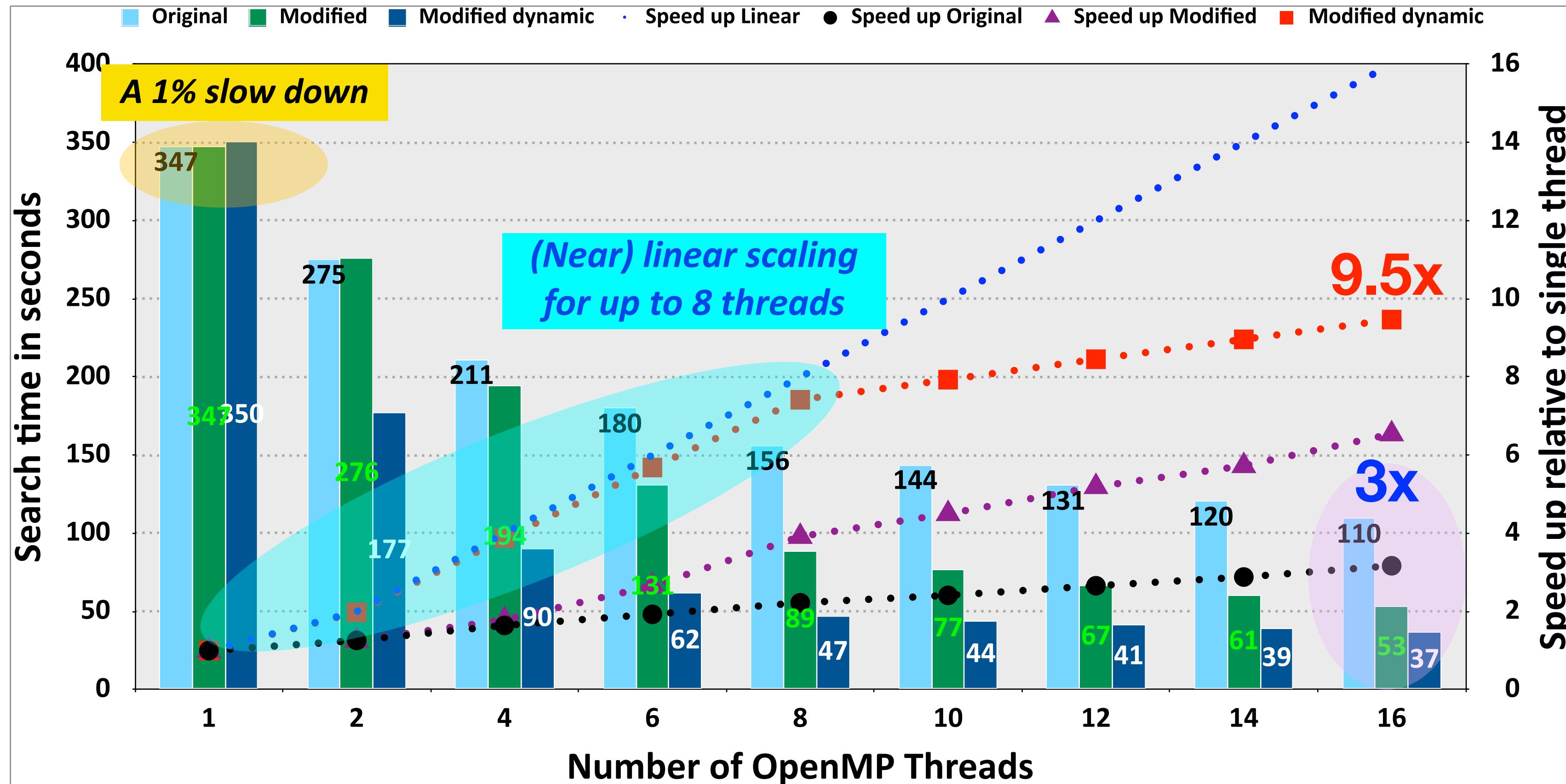
By The Way

How Do You Know The Chunk Size Should Be 25?

Crystal Ball

Trial And Error

The Performance With Dynamic Scheduling Added



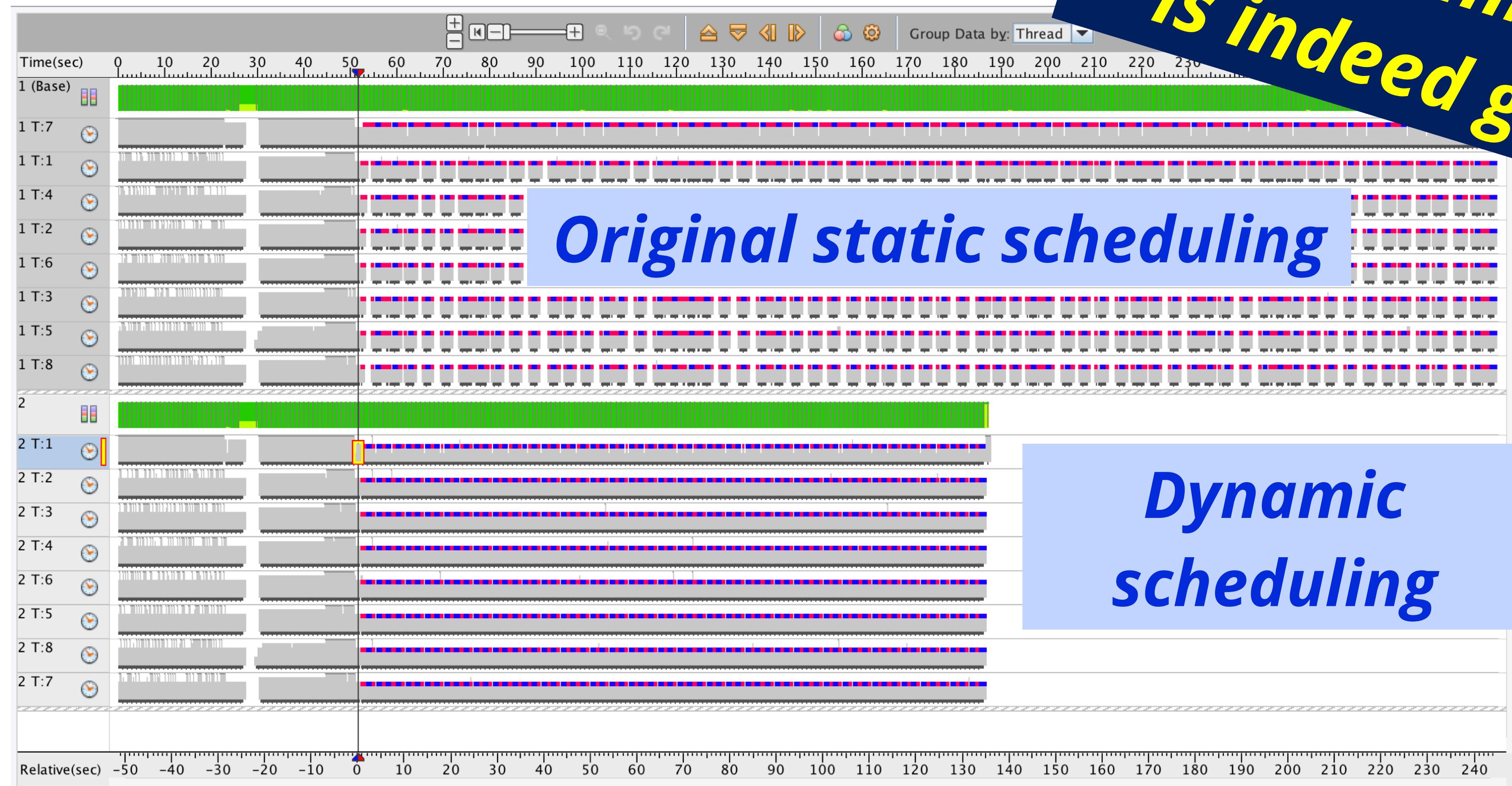
- ✓ A 1% slow down on a single thread
- ✓ Near linear scaling for up to 8 threads
- ✓ The parallel speed up increased to 9.5x
- ✓ The search time is reduced by 3x

*The modified version is 3x faster
than the original code*

Really Important

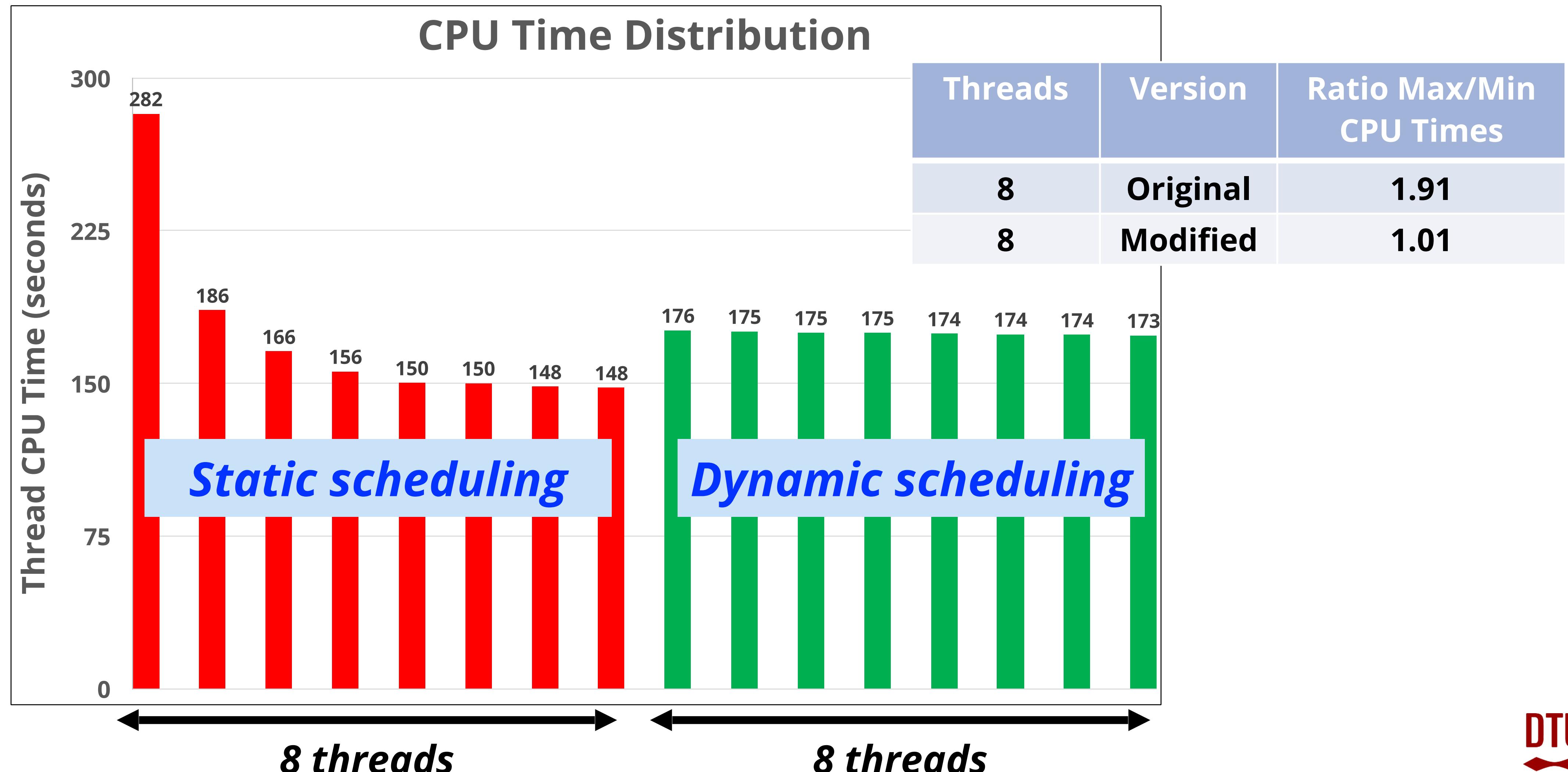
Always Verify the Behaviour!

Before and After (8 Threads)



The load imbalance
is indeed gone!

The Load Imbalance is Indeed Gone



Part I - Takeaways

There are many opportunities to improve the performance

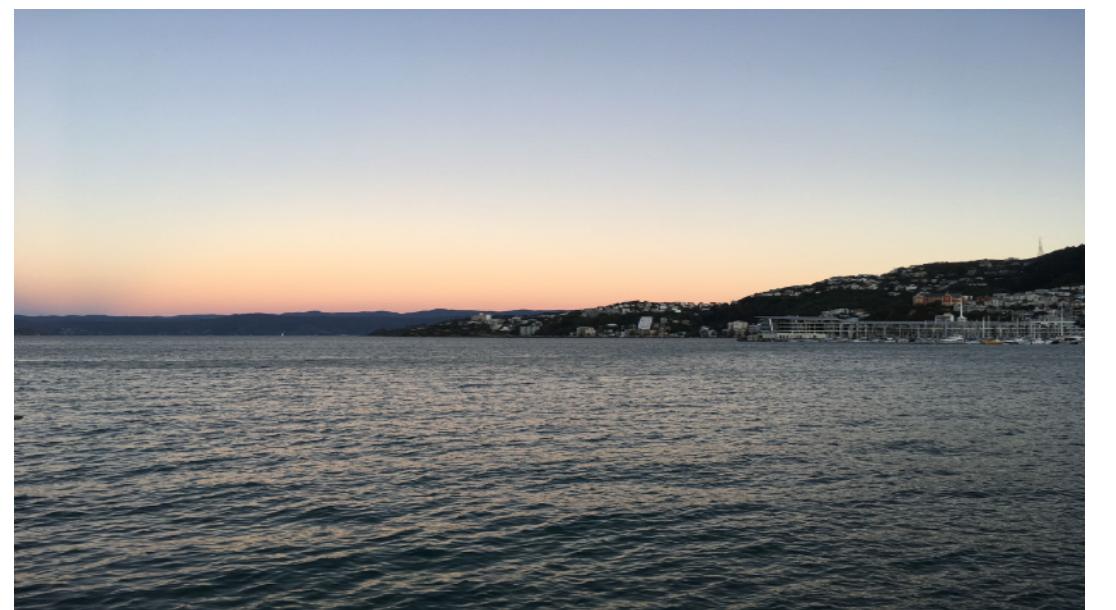
If you follow the advice given, you should be fine

(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

Part II - The Joy Of Computer Memory



Motivation Of This Work

Question: "Why Do You Rob Banks?"

Answer: "Because That's Where The Money Is"

Willie Sutton – Bank Robber, 1952

Question: "Why Do You Focus On Memory?"

Answer: "Because That's Where The Bottleneck Is"

Ruud van der Pas – Performance Geek, 2022

When Do Things Get Harder?

Memory Access “Just Happens”

There are however two cases to watch out for

NUMA and False Sharing

They have nothing to do with OpenMP though and are a characteristic of a shared memory architecture

What is False Sharing?

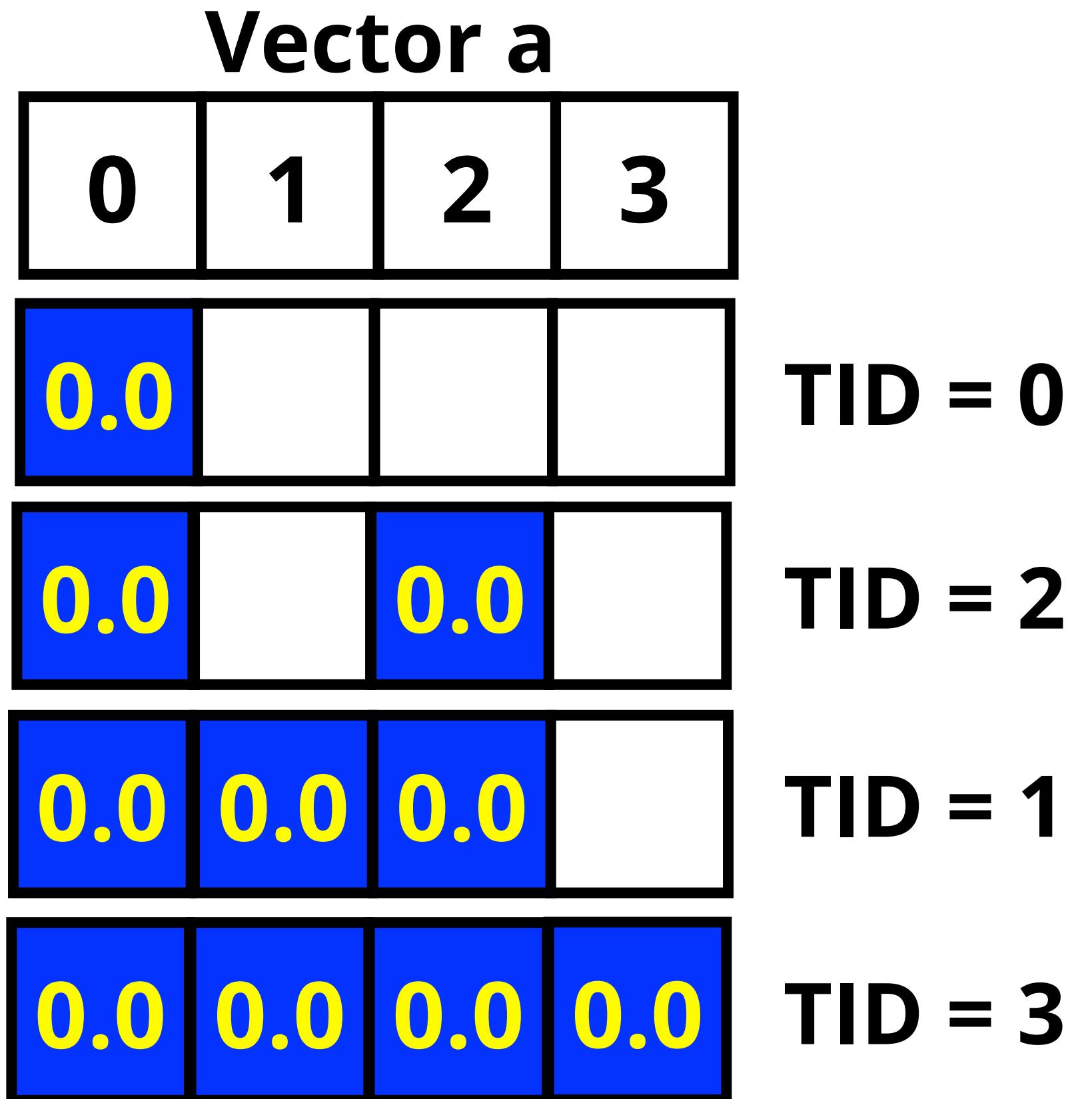
A corner case, but it may affect you

*Happens when multiple threads **modify** the same cache line at
the same time*

*This results in the cache line to move around
(plus the additional cost of the cache coherence)*

An Example of False Sharing

```
#pragma omp parallel shared(a)
{
    int TID = omp_get_thread_num()
    a[TID] = 0.0; // False Sharing
} // End of parallel region
```



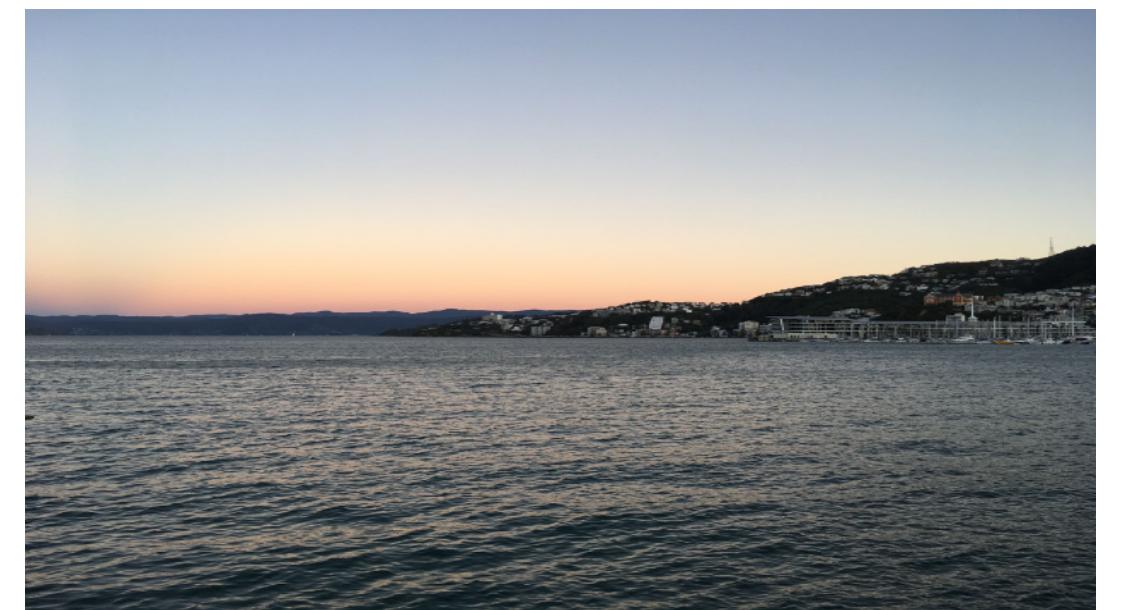
Now Things Are About To Get “Interesting”

False Sharing is important, but a corner case

*Non-Uniform Memory Access (“NUMA”) is much more general
and more likely to affect the performance of your code*

***The remainder of this talk is about NUMA
(you still have 10 seconds to leave, but please don't scream too loudly)***

NUMA in Contemporary Systems



Modern Times

Non-Uniform Memory Access (“NUMA”) used to be the realm of large servers only

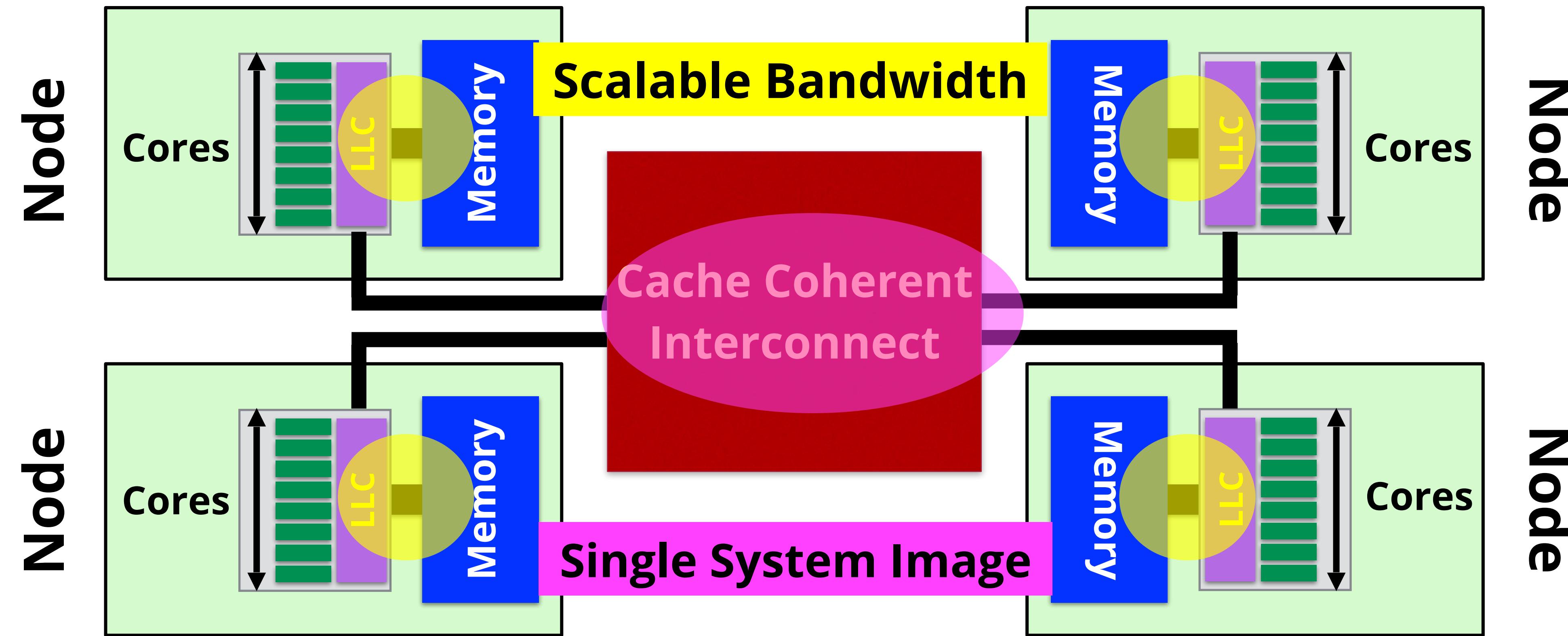
This is no longer true and therefore a concern to all

The tricky thing is that “things just work”

But do you know how efficiently your code performs?

NUMA - The System Most of Us Use Today

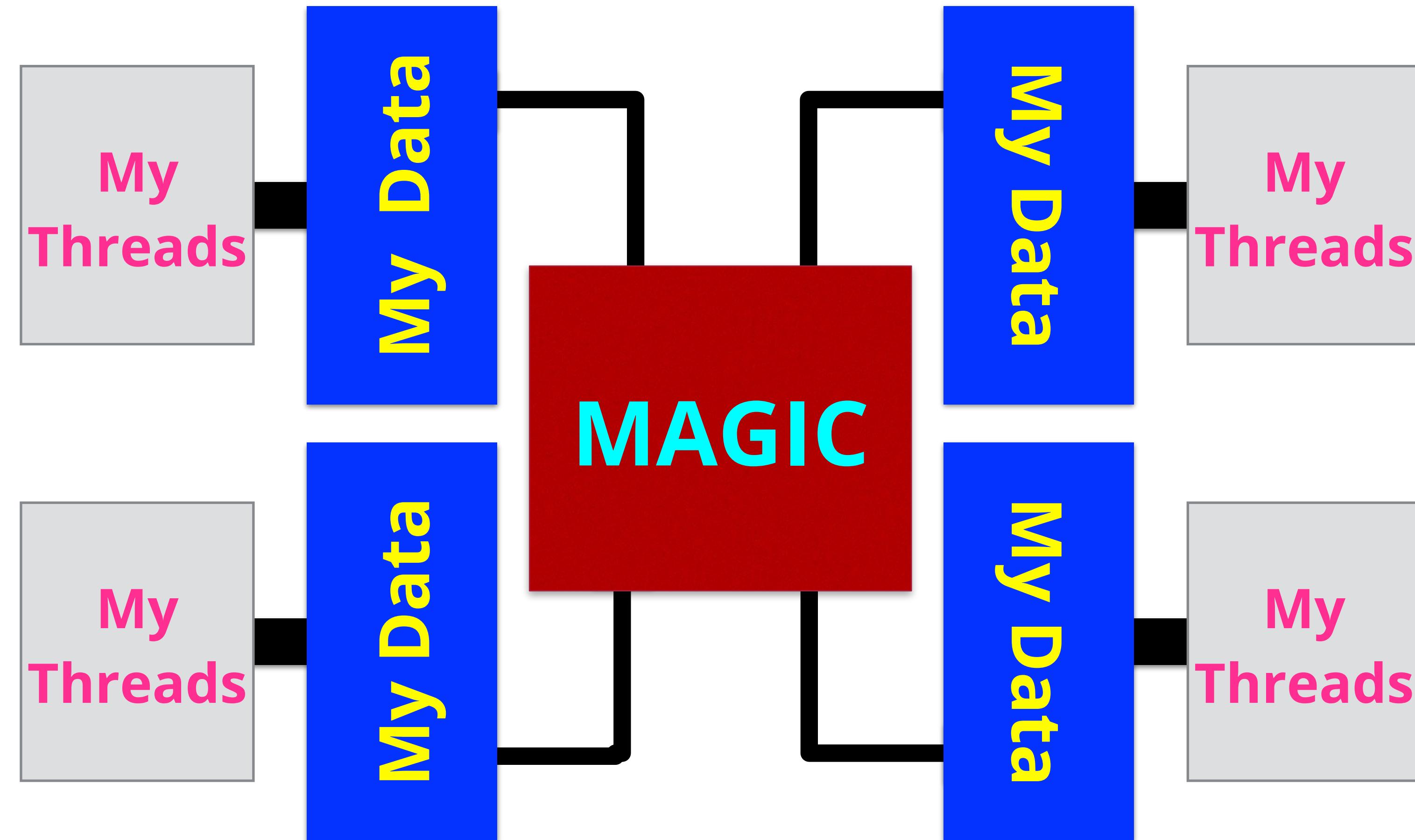
A Generic, but very Common and Contemporary NUMA System



OpenMP®



The Developer's View



The NUMA View

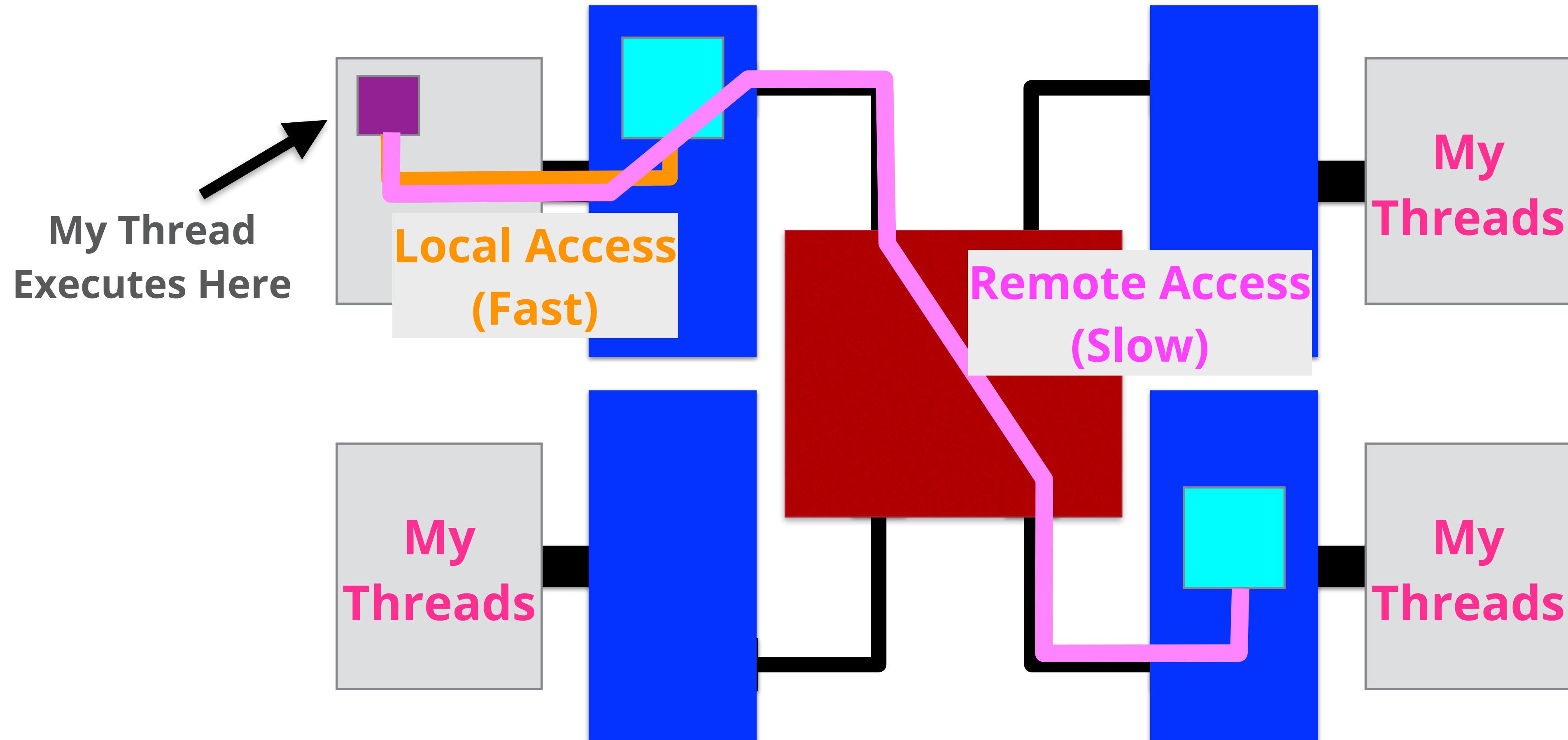
Memory is physically distributed, but logically shared

Shared data is accessible to all threads

You don't know where the data is and it doesn't matter

Unless you care about performance

Local Versus Remote Access Times



Tuning for a NUMA System

Tuning for NUMA is about keeping threads and their data close

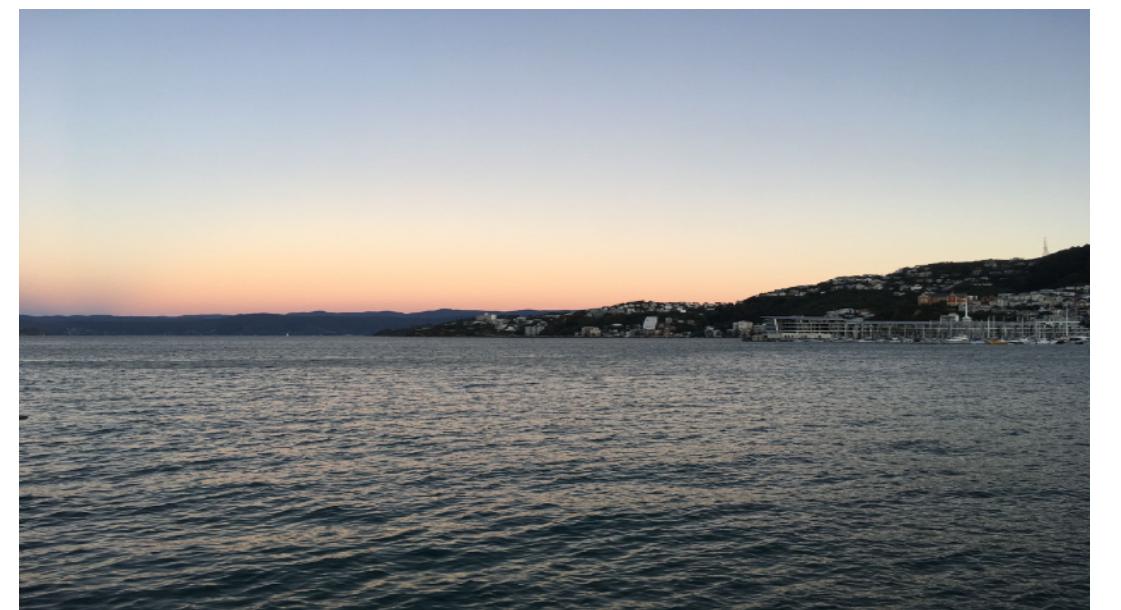
In OpenMP, a thread may be moved to the data

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 it right
(in this context, "right" is not about correctness, but about the performance)*

About NUMA and Data Placement



The First Touch Data Placement Policy

So where does data get allocated then?

*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 and Parallel Computing

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

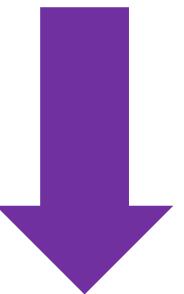
*This increases access times for certain threads
(and may also cause congestion on the network)*

Luckily, the solution is (often) surprisingly simple

A Sequential Initialization

```
for (int64_t i=0; i<n; i++)  
    a[i] = 0;
```

One thread executes this loop

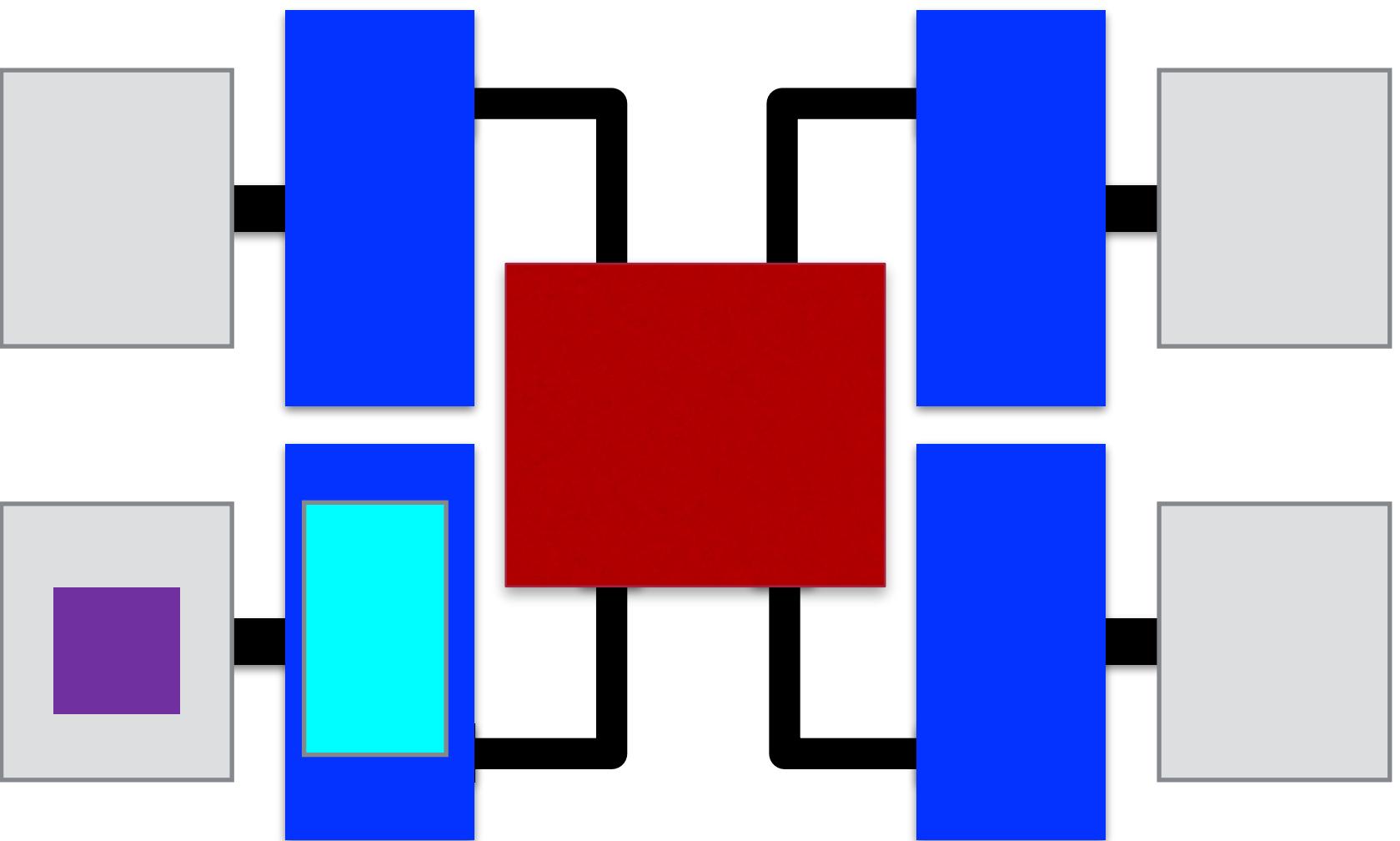


All of "a" is in a single node



= Thread

= Data

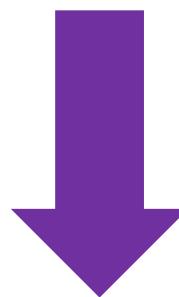


Note: The allocation is on a virtual memory page basis

Leverage the First Touch Placement Policy

```
#pragma omp parallel for schedule(static)
for (int64_t i=0; i<n; i++)
    a[i] = 0;
```

Four threads execute this loop



The data is spread out



- = Thread
- = Data

Note: The allocation is on a virtual memory page basis

The Tricky Part

Q: How about I/O ?

A: Add a redundant parallel initialization before reading the data

Q: What if the data access pattern is irregular?

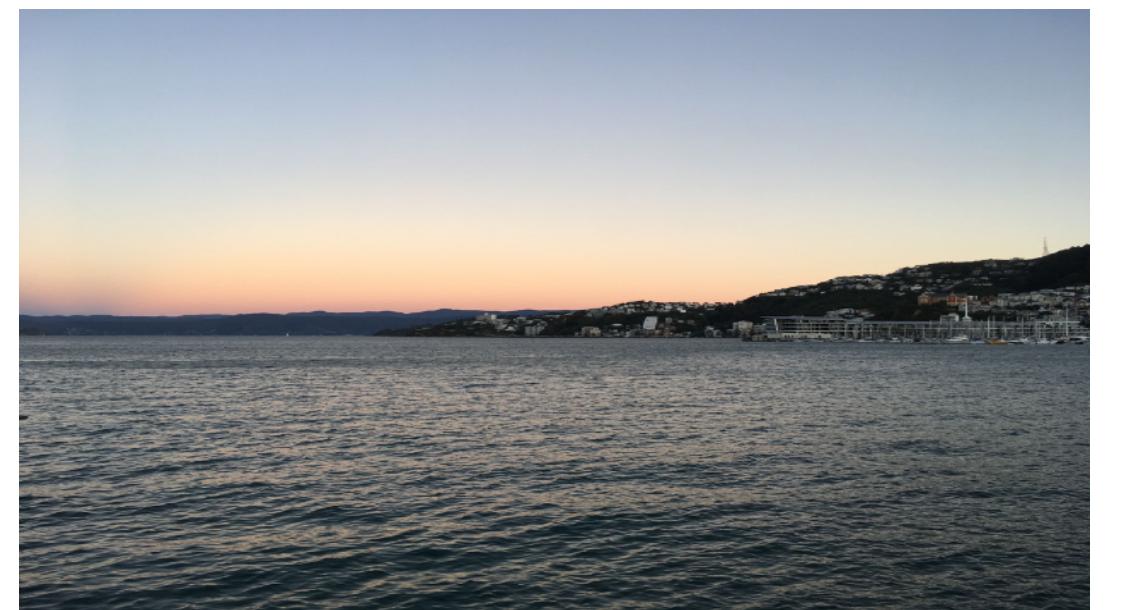
A: Randomize the data placement

About Memory Allocations

Do not use “calloc” for global memory allocation

Okay to use within a single thread

OpenMP Support for NUMA Systems



OpenMP Support For Thread Affinity

Philosophy:

- *The data is where it happens to be*
- *Move a thread to the data it needs most*

There are two environment variables to control this

The Affinity Related OpenMP Environment Variables

OMP_PLACES

Defines where threads may run

OMP_PROC_BIND

Defines how threads map onto the OpenMP places

Note: Highly recommended to also set OMP_DISPLAY_ENV=verbose

Placement Targets Supported by OMP_PLACES

Keyword	Place definition
threads	A hardware thread
cores	A core
ll_caches	A set of cores that share the last level cache
numa_domains	A set of cores that share a memory and have the same distance to that memory
sockets	A single socket

Hardware Thread ID Support to Define Places

The abstract names are preferred

*The **OMP_PLACES** variable also supports hardware thread IDs*

Places can be defined using any sequence of valid numbers

A compact set notation is supported as well

Notation: {start:total:increment}

For example: {0:4:2} expands to {0,2,4,6}

Examples How to Use OMP_PLACES

Threads are scheduled on the numa_domains in the system:

```
$ export OMP_PLACES=numa_domains
```

Use Hardware Thread IDs 0, 8, 16, and 24:

```
$ export OMP_PLACES="{0},{8},{16},{24}"
```

```
$ export OMP_PLACES={0}:4:8
```

Map Threads onto Places

Use variable `OMP_PROC_BIND` to map threads onto places

The settings define the mapping of threads onto places

***The following settings are supported:
`true`, `false`, `primary`, `close`, or `spread`***

The definitions of `close` and `spread` are in terms of the place list

An Example Using Places and Binding

Threads are scheduled on the cores in the system:

```
$ export OMP_PLACES=cores
```

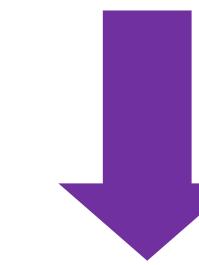
*And they should be placed on cores as far away from each other
as possible:*

```
$ export OMP_PROC_BIND=spread
```

Remember This Example?

```
#pragma omp parallel for schedule(static)
for (int64_t i=0; i<n; i++)
    a[i] = 0;
```

Four threads execute this loop



Wishful Thinking

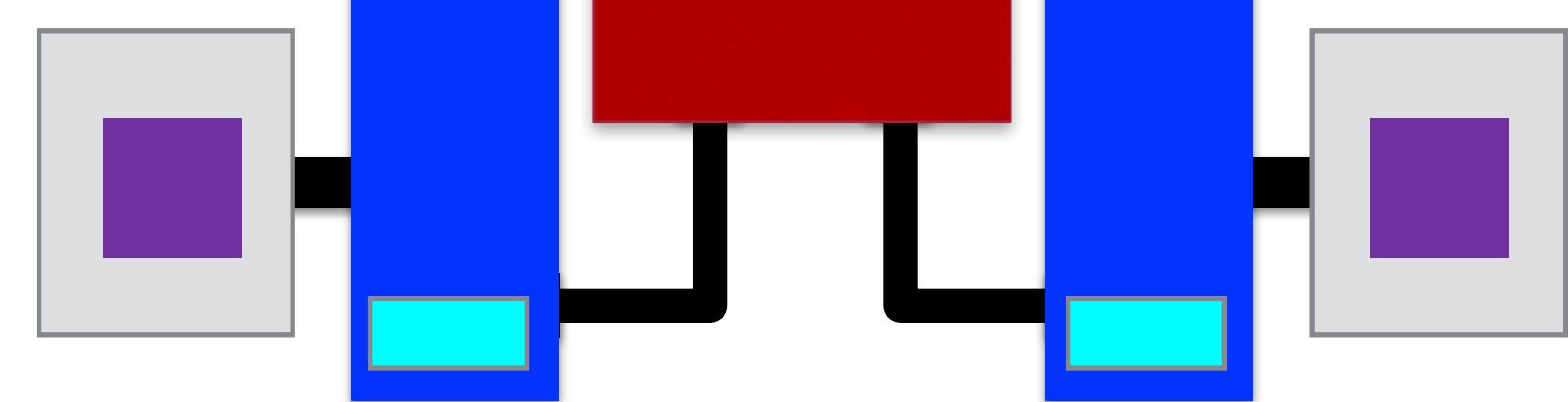
*Data placement depends on
where threads execute*

Use Affinity Controls

Open....



```
$ export OMP_PLACES=numa_domains
$ export OMP_PROC_BIND=spread
```



= Thread

= Data



NUMA Diagnostics

It is very easy to make a mistake with the NUMA setup

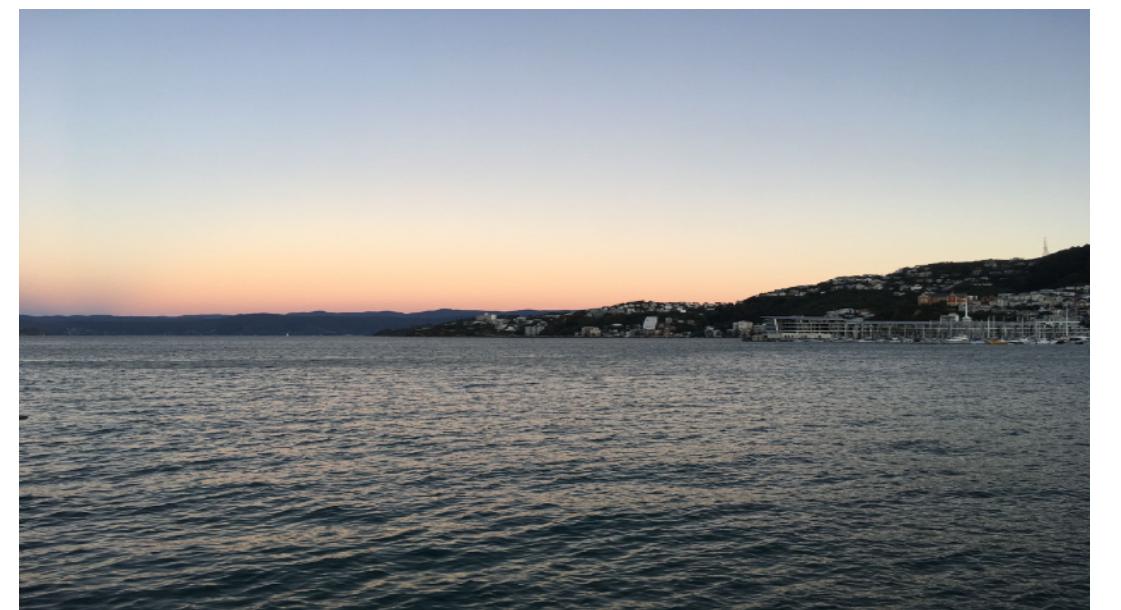
Two very simply, but yet powerful features to assist:

*Variable **OMP_DISPLAY_ENV** echoes the initial settings*

*Variable **OMP_DISPLAY_AFFINITY** prints information at run time*

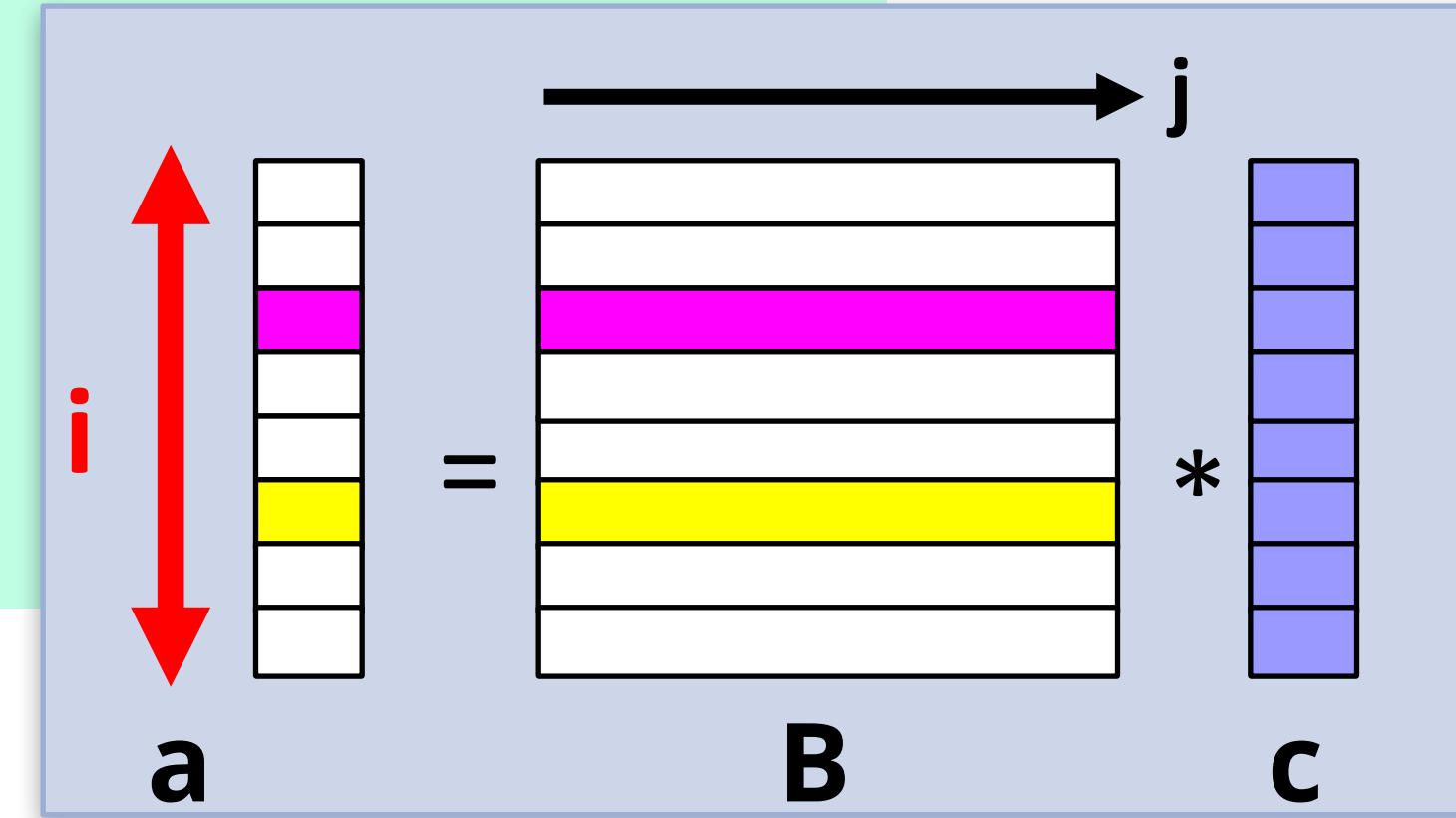
Highly recommended to use these diagnostic features!

A Performance Tuning Example



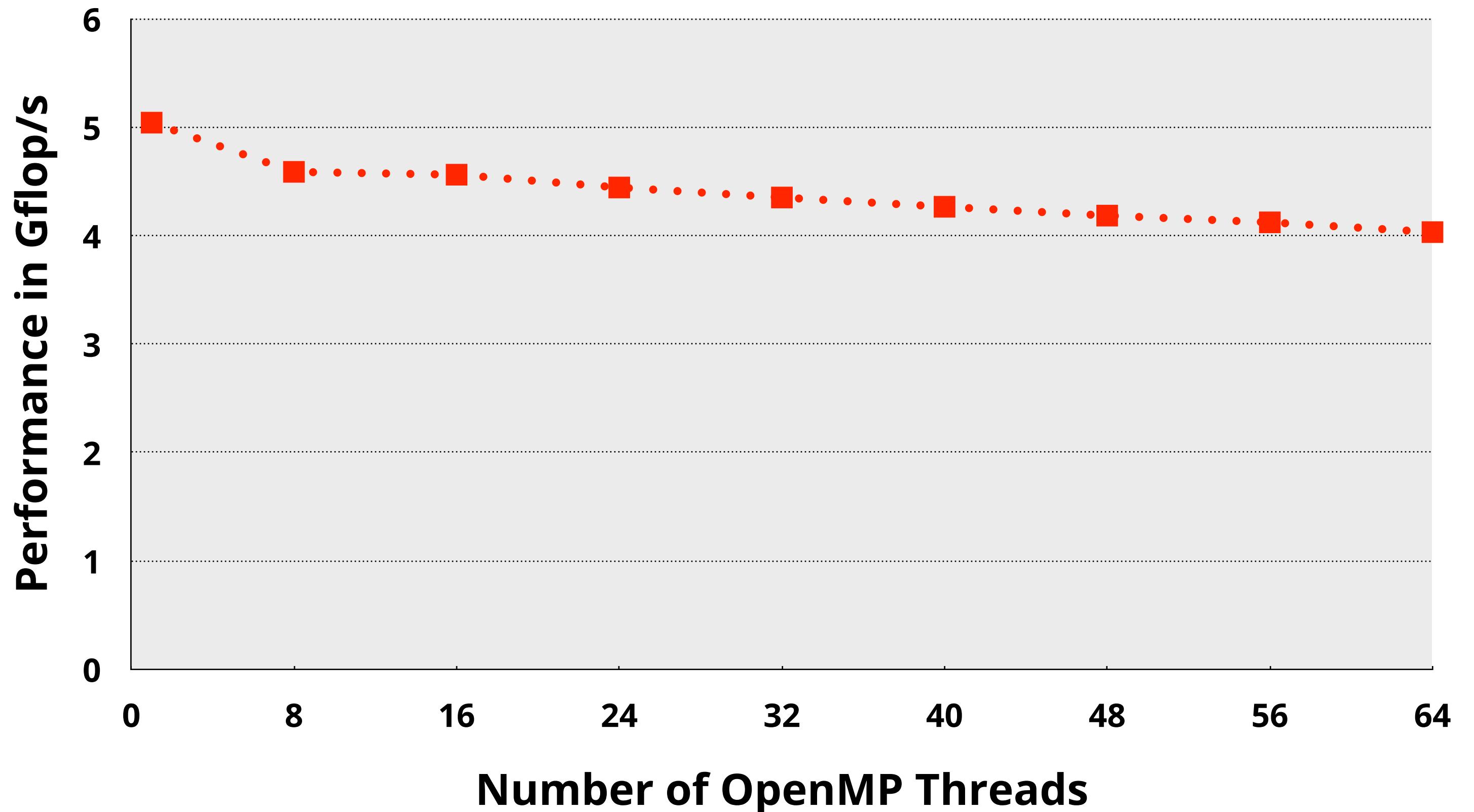
*Matrix Times Vector Multiplication: $a = B*c$*

```
#pragma omp parallel for default(none) \
    shared(m,n,a,b,c)
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;
}
```



*The Performance Using 64 Threads**

Performance of the matrix-vector algorithm (4096x4096)



This is a highly parallel algorithm, but adding threads degrades the performance!

Automatic NUMA Balancing in Linux

This is an interesting feature available in Linux

“Automatic NUMA balancing moves tasks (which can be threads or processes) closer to the memory they are accessing. It also moves application data to memory closer to the tasks that reference it. This is all done automatically by the kernel when automatic NUMA balancing is active.”

“Virtualization Tuning and Optimization Guide”, Section 9.2, Red Hat documentation

```
# echo 1 > /proc/sys/kernel/numa_balancing
```

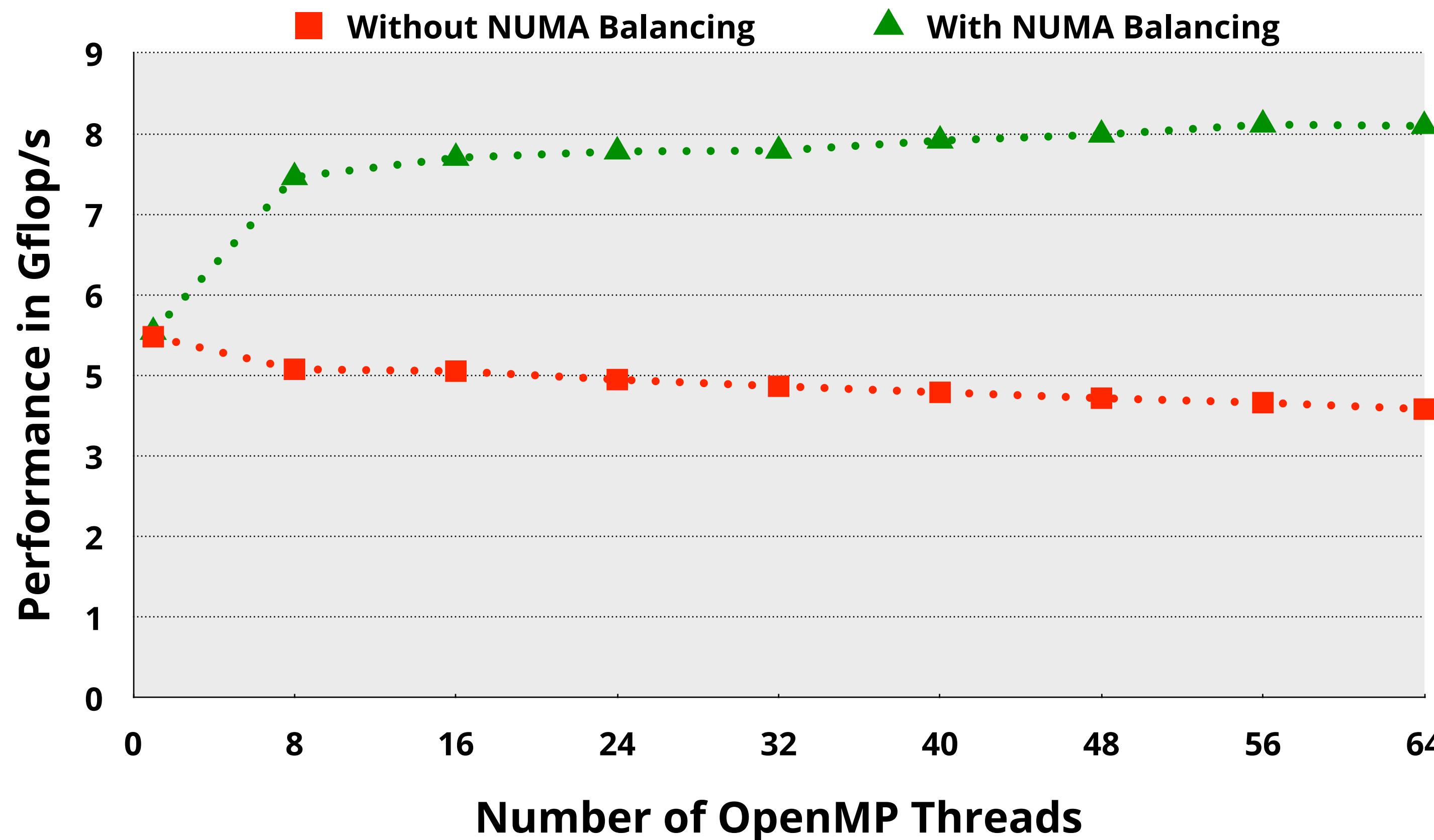
enable

```
# echo 0 > /proc/sys/kernel/numa_balancing
```

disable

The Performance Using 64 Threads*

Performance of the matrix-vector algorithm (4096x4096)



NUMA balancing gives a 1.6x improvement, but the performance is still rather poor

Let's Check The System We Are Using!



OpenMP

DTU


The NUMA Information for the System

\$ lscpu

8 cores/node

```
.....  
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 8 NUMA Nodes (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  
.....
```

2 columns => 2 threads/core

Op.....

node distances:

node	0	1	2	3	4	5	6	7
0:	10	16	16	16	32	32	32	32
1:	16	10	16	16	32	32	32	32
2:	16	16	10	16	32	32	32	32
3:	16	16	16	10	32	32	32	32
4:	32	32	32	32	10	16	16	16
5:	32	32	32	32	16	10	16	16
6:	32	32	32	32	16	16	10	16
7:	32	32	32	32	16	16	16	10

*The NUMA Structure of the System**

Consists of 8 NUMA nodes according to “lscpu”

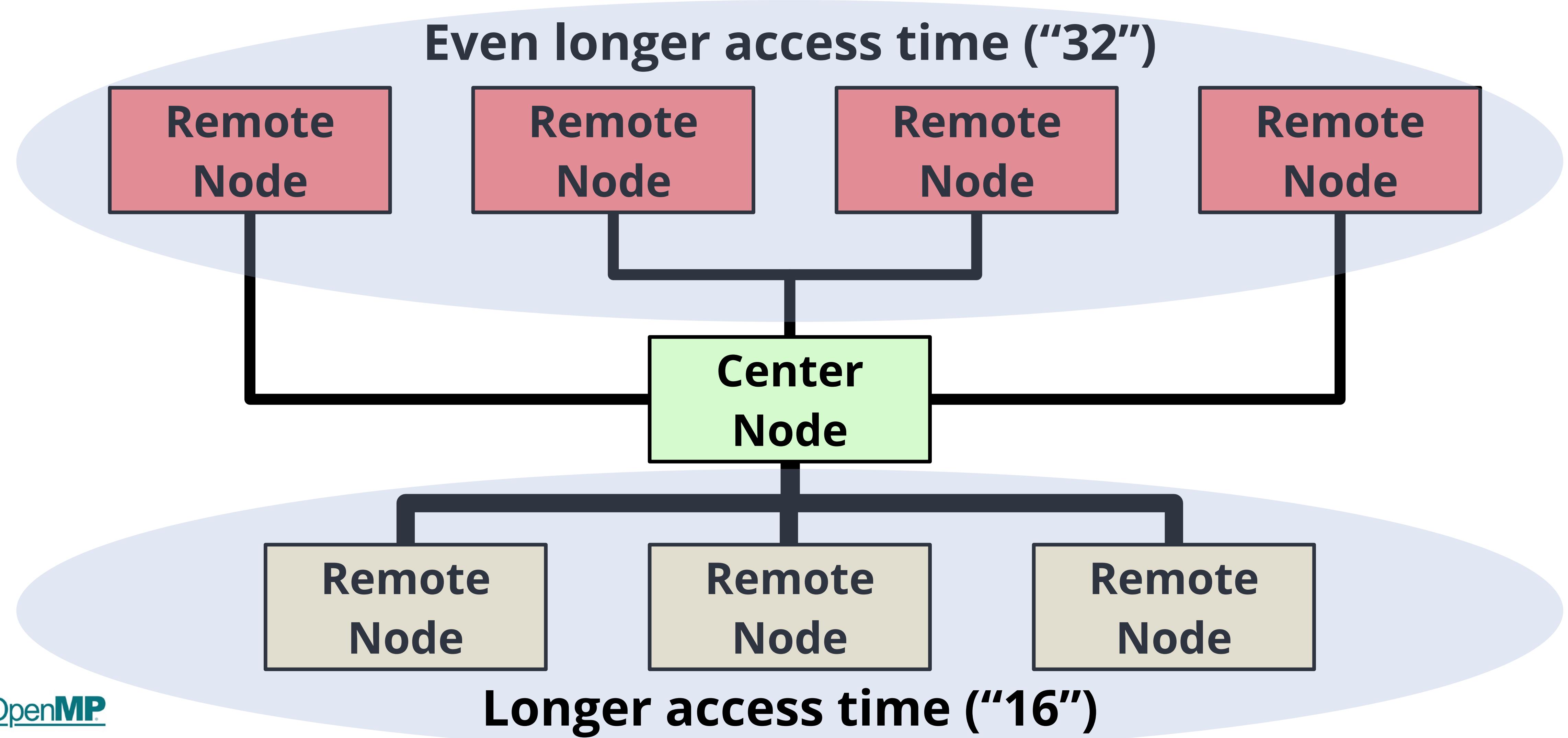
There are two levels of NUMA (“16” and “32”)

Each NUMA node has 8 cores with 2 hardware threads each

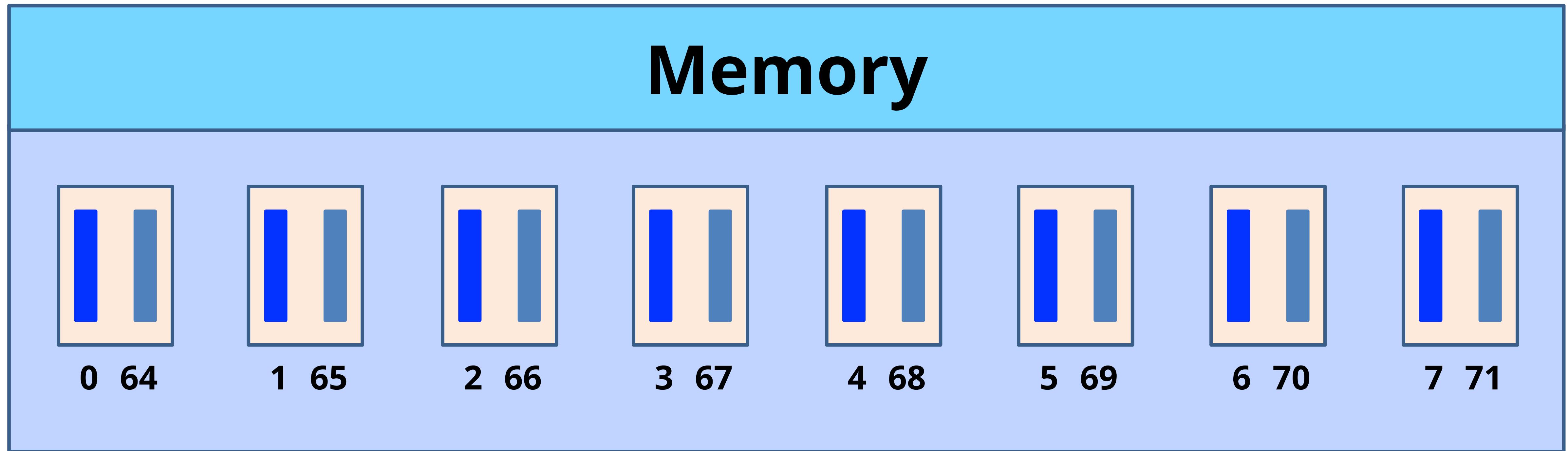
In total the system has 64 cores and 128 hardware threads

**) This is an AMD EPYC “Naples” 2 socket server (yes, I know, it is relatively old :-))*

The Abstract System Topology (`numactl -H`)



Example - NUMA Node 0 (lscpu output)



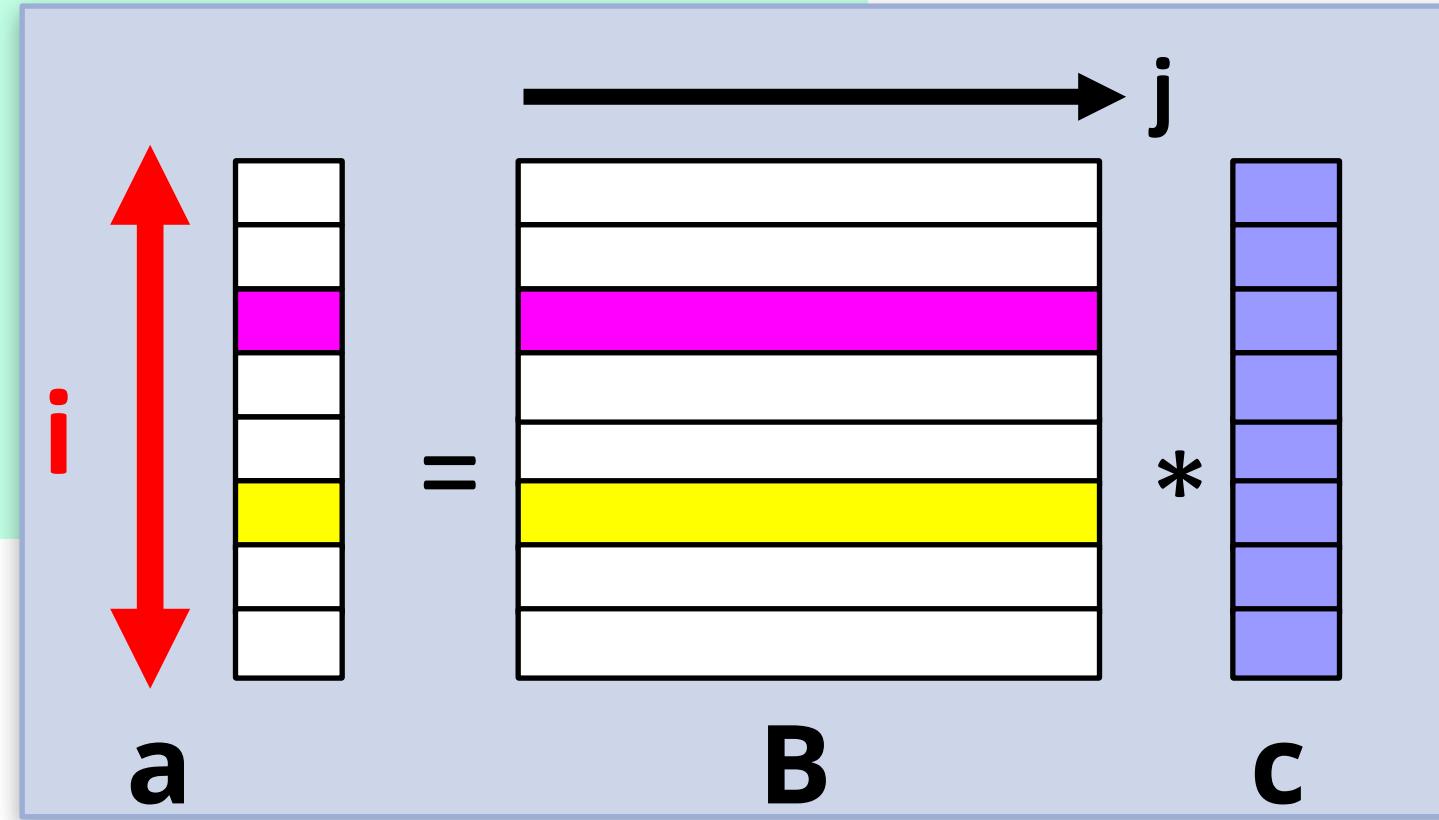
*8 cores
16 hardware threads*

All cores and hardware threads share the memory in the node



*Recall the Code Used Here ($a = B*c$)*

```
#pragma omp parallel for default(none) \
    shared(m,n,a,b,c)
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;
}
```



Is There Anything Wrong Here?

Nothing wrong with this code

But this code is not NUMA aware

The data initialization is sequential

Therefore, all data ends up in the memory of a single node

This is a more NUMA friendly data initialization:

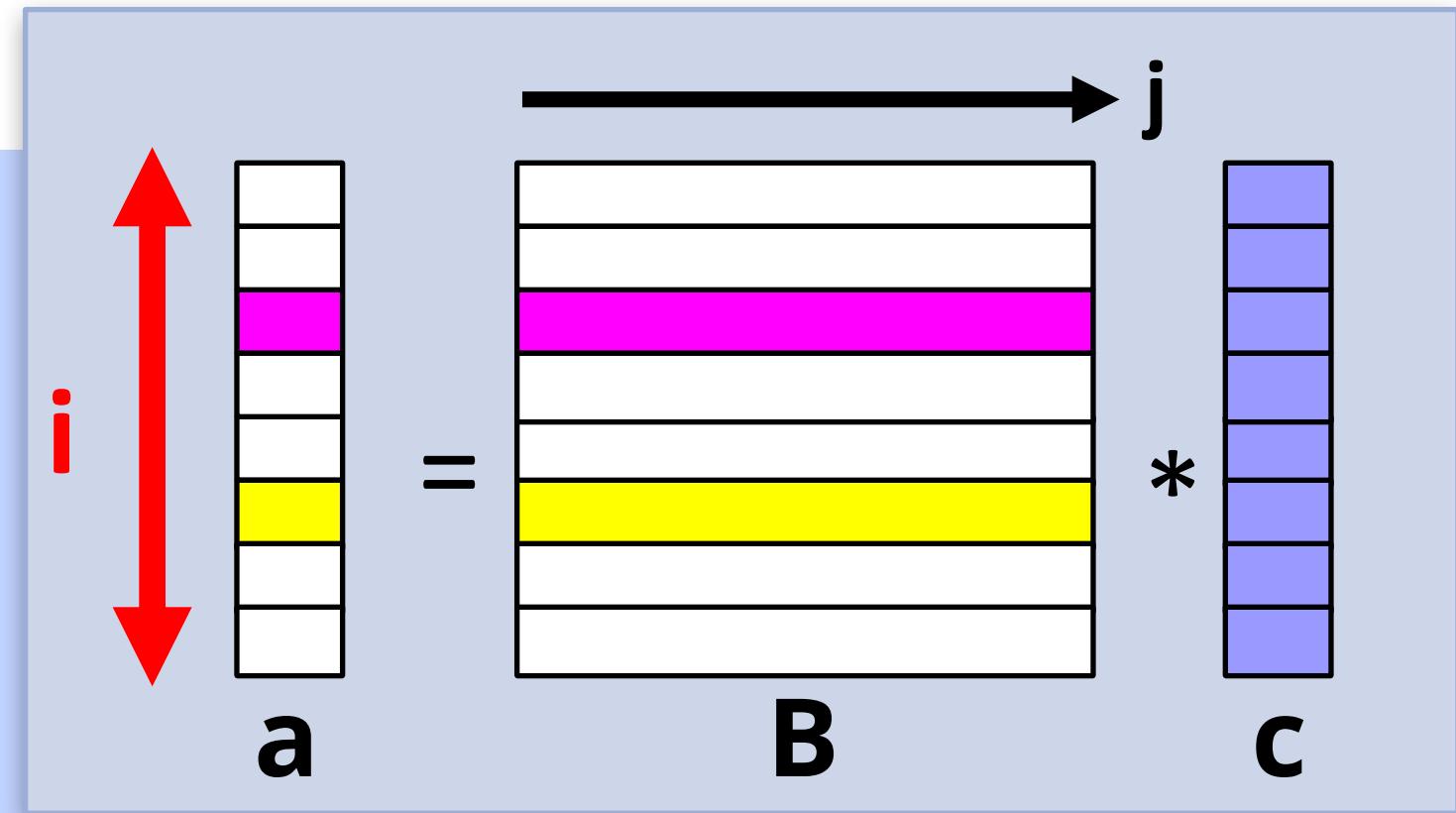
The Original Data Initialization

```
for (int64_t j=0; j<n; j++)
    c[j] = 1.0;

for (int64_t i=0; i<m; i++) {
    a[i] = -1957;
    for (int64_t j=0; j<n; j++)
        B[i][j] = i;
}
```

A NUMA Friendly Data Initialization

```
#pragma omp parallel
{
    #pragma omp for
    for (int64_t j=0; j<n; j++)
        c[j] = 1.0;
    #pragma omp for
    for (int64_t i=0; i<m; i++) {
        a[i] = -1957;
        for (int64_t j=0; j<n; j++)
            B[i][j] = i;
    }
} // End of parallel region
```

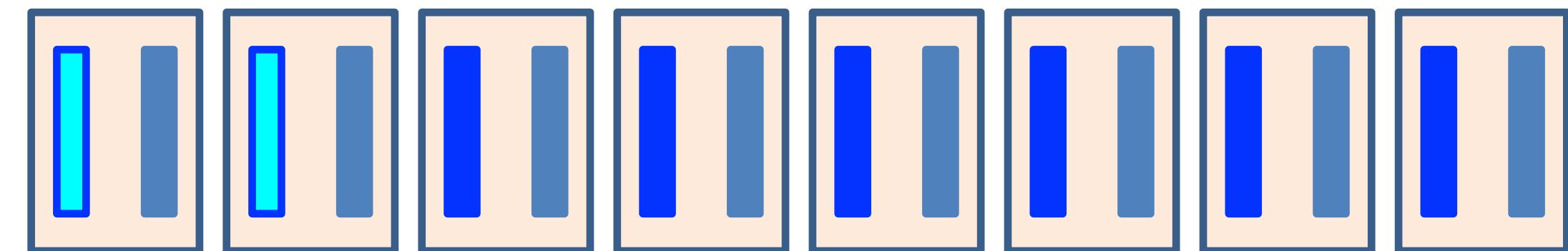


Control the Mapping of Threads

The Thread Placement 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 - The Target Hardware Thread Numbers



OpenMP



An Example How to Use OpenMP Affinity

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
```

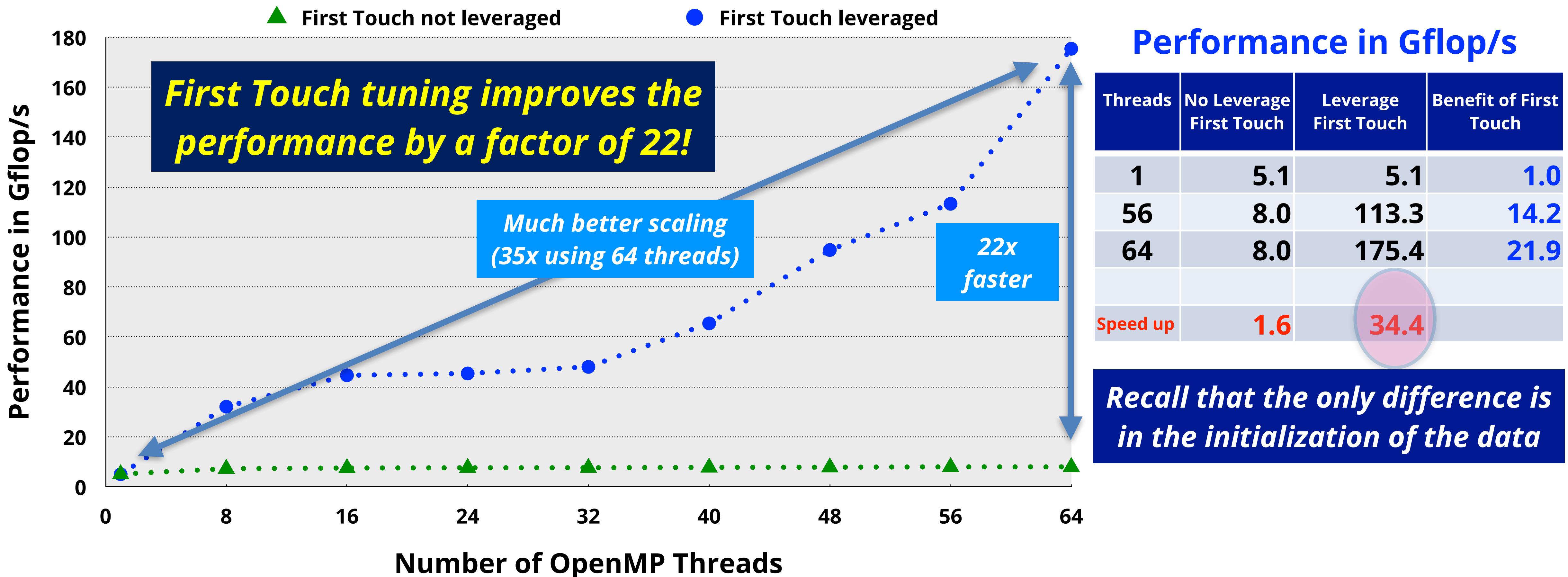
```
$ export OMP_PROC_BIND=close
```

```
$ export OMP_NUM_THREADS=16
```

```
$ ./a.out
```

```
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
```

The Performance for a 4096x4096 matrix



OpenMP

Part II - Takeaways

Data and thread placement matter (a lot)

Important to leverage First Touch Data Placement

OpenMP has elegant, yet powerful, support for NUMA

The NUMA support in OpenMP continues to evolve and expand

Wrapping Things Up

Think Ahead

Follow the tuning guidelines given in this talk

Always use a profiling tool to guide the tuning efforts

Performance tuning is an iterative process

In many cases, a performance “mystery” is explained by NUMA effects, False Sharing, or both

Thank You And ... Stay Tuned!

*Bad OpenMP
Does Not Scale*

Ruud van der Pas

Guest Lecture DTU, January 12, 2022

OpenMP

