

An aerial view from an airplane window showing a city and a large body of water. The city is built on a hillside overlooking the water. The sky is blue with some clouds. The airplane's wing and fuselage are visible in the foreground.

Speed Up Your OpenMP Application Without Doing Much

Ruud van der Pas

Director, OpenMP Architecture Review Board

PPCES 2026

March 16-20, 2026

RWTH Aachen University

\$ whoishe

My background is in mathematics and physics

Previously, I worked at Philips Electronics, the University of Utrecht, Convex Computer, SGI, Sun Microsystems, and Oracle

I have been involved with OpenMP since the introduction

I am passionate about performance and OpenMP in particular



Agenda

Prologue

Part I - Tips and Tricks

A Well Deserved 10 Minute Break

Part II - The Joy of Computer Memory

Q and (some) A



Prologue



Your OneStop Place for OpenMP

<https://www.openmp.org>

New OpenMP API 6.0 Examples Document

Provides programming examples of the OpenMP API, including new features found in the OpenMP 6.0 Specification.

DOWNLOAD

Latest News and Events

OpenMP ARB announces new Board Members van der Pas and Stotzer

New Board Members van der Pas and Stotzer

The OpenMP ARB welcomes Ruud van der Pas and Eric Stotzer to its Board of Directors.

Tell us what you think about the OpenMP Examples book*

*Download the book for free on the Specifications page. Survey closes on March 22.

OpenMP Example Book Survey

If you have used our OpenMP API Examples book, take our survey to tell us how we can improve it.

ANACONDA
python

New Python Subcommittee, and Anaconda joins OpenMP ARB

Anaconda has joined the OpenMP ARB, and the OpenMP ARB has formed a new Python subcommittee to add Python as the fourth supported language in the OpenMP API.



**Coming up:
Support for Python
in OpenMP!**



The Number of Members Continues to Increase

A total of 33 members

Members

The OpenMP Architecture Review Board (ARB) includes the following members:

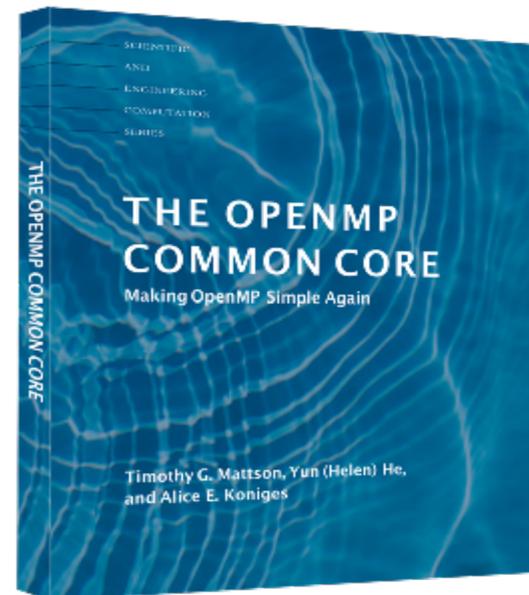
 AMD Greg Rodgers	 Anaconda Stanley Selbert	 Argonne National Laboratory Kalyan Kumaran	 ASC/Lawrence Livermore National Laboratory Bronis R. de Supinski	 Barcelona Supercomputing Center Xavier Martorell	 Leibniz Supercomputing Centre Viktor Weinberg	 Los Alamos National Laboratory Jenai Muli-Yusuf	 Micron Tandy Meyer	 NASA Henry Jin	 NCSA Roland Haas
 BayLibre Tobias Dornus	 CEA Adrien Roussel	 CSC - IT Center for Science Vartti Louhivuori	 Edinburgh Parallel Computing Centre Mark Bull	 Fujitsu Takahiro Kawashima	 NEC Shinichi Otsu	 NextSilicon Ilan Toyari	 NVIDIA W. D. A. Jeff Hammond	 Oak Ridge National Laboratory Swaroop Pophale	 Pawsey Supercomputing Research Centre Alexis Espinosa
 Hewlett-Packard Enterprise Deepak Eachampati	 IBM Kevin Li	 Inria Marzin Schreiber	 Intel Xinmin Tian	 Lawrence Berkeley National Laboratory Helen He	 RWTH Aachen University Christian Terboven	 Sandia National Laboratory Stephen Olivier	 SiFive Alexey Tataev	 Stony Brook University Tony Curtis	 SUSE Michael Matr
					 Texas Advanced Computing Center Kent V. Feld	 University of Bristol Tim Mannon	 UNC Charlotte Yinghong Yan		



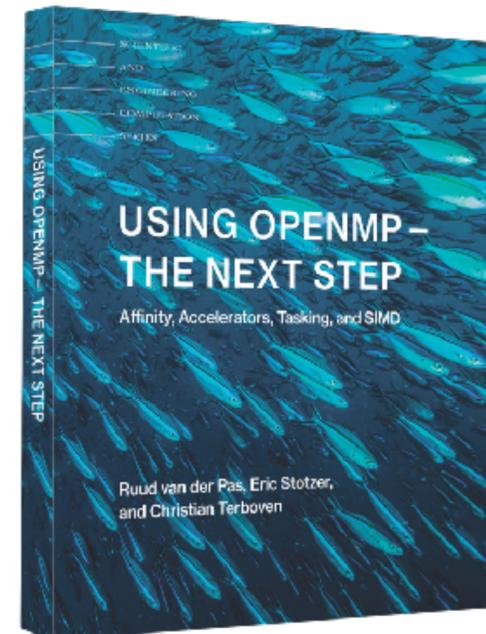
Food for the Eyes and Brains



**OpenMP 2.5 and
intro Parallel
Computing**



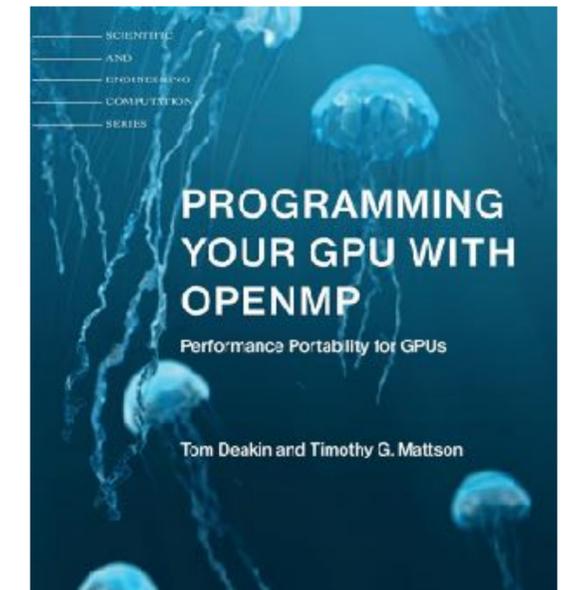
**Covers the OpenMP
Basics to get started**



**Focus on the
Advanced Features**



**What goes on
under the hood?**



**Must read for users
of GPUs in OpenMP**



About gprofng

There are many profiling tools available

Select the one that you prefer

*The **gprofng** profiling tool is part of GNU binutils*

It is the tool that I used for the profiling views in this talk

And yes, I am involved with the development ;-)



More Info

<https://blogs.oracle.com/linux/post/gprofng-the-next-generation-gnu-profiling-tool>

Linux Toolchain & Tracing

gprofng: The Next Generation GNU Profiling Tool

January 26, 2023 | 10 minute read



Elena Zannoni

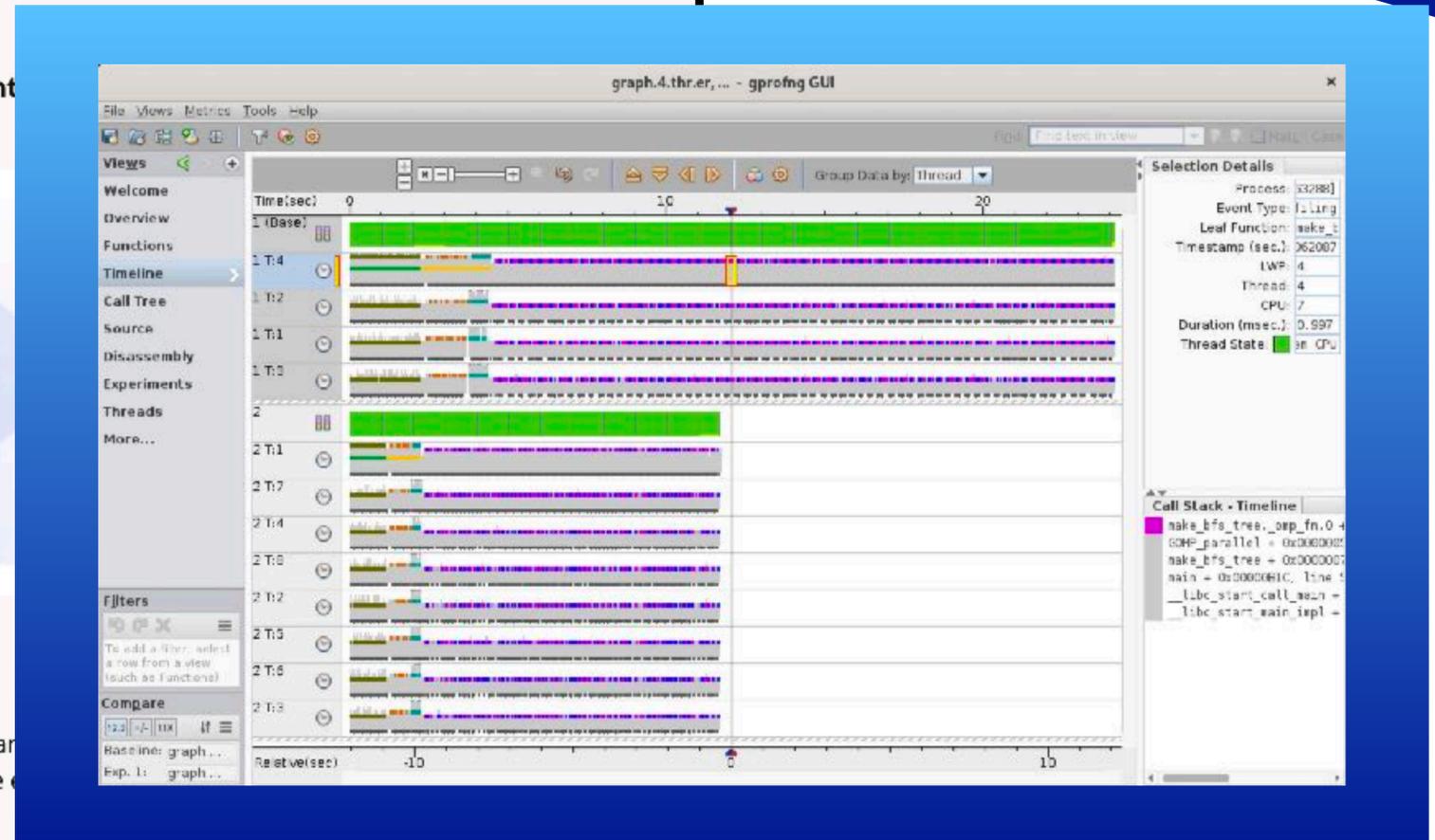
This blog entry was contributed by: Ruud van der Pas, Kurt Goebel, Vladimir Mezentsev, and the Linux Toolchain Team and are involved with gprofng on a daily basis.



What is gprofng?

Gprofng is a next generation application profiling tool. It supports the profiling of programs running on systems using processors from Intel, AMD, Arm, or compatible vendors. The tool is highly configurable, but the basic views are always available.

Two distinct steps are needed to produce a profile. In the first step, the performance data is collected. This information is stored in a directory called the experiment directory. There are several tools available to display and analyze the information stored in this directory.



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**

**) It is fine to blame the weather, or politicians, or both though*



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

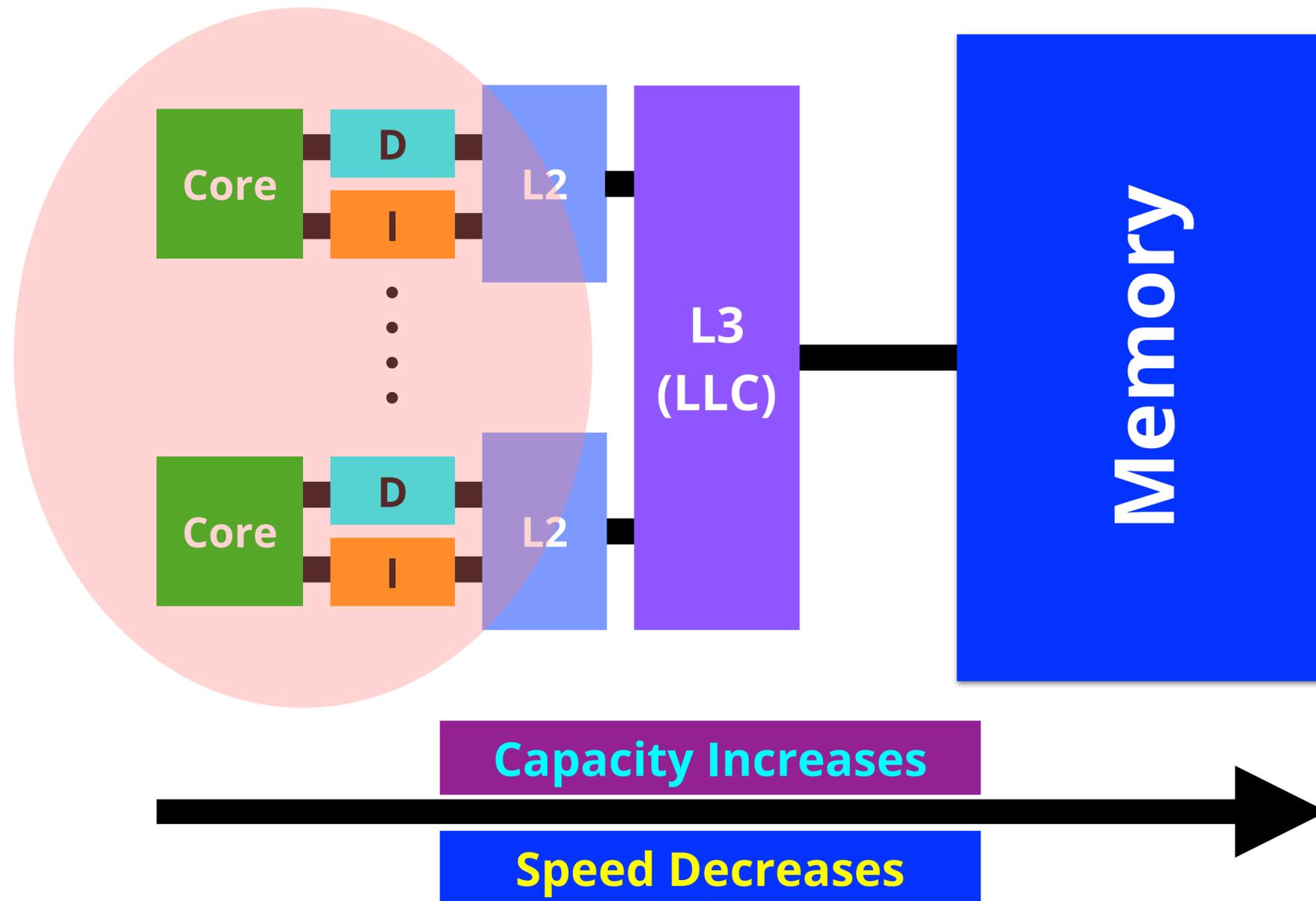
For cost and performance reasons, a modern processor has a hierarchy of caches

Some caches are private to a core, others are shared

Let's look at a typical example



A Typical Memory Hierarchy

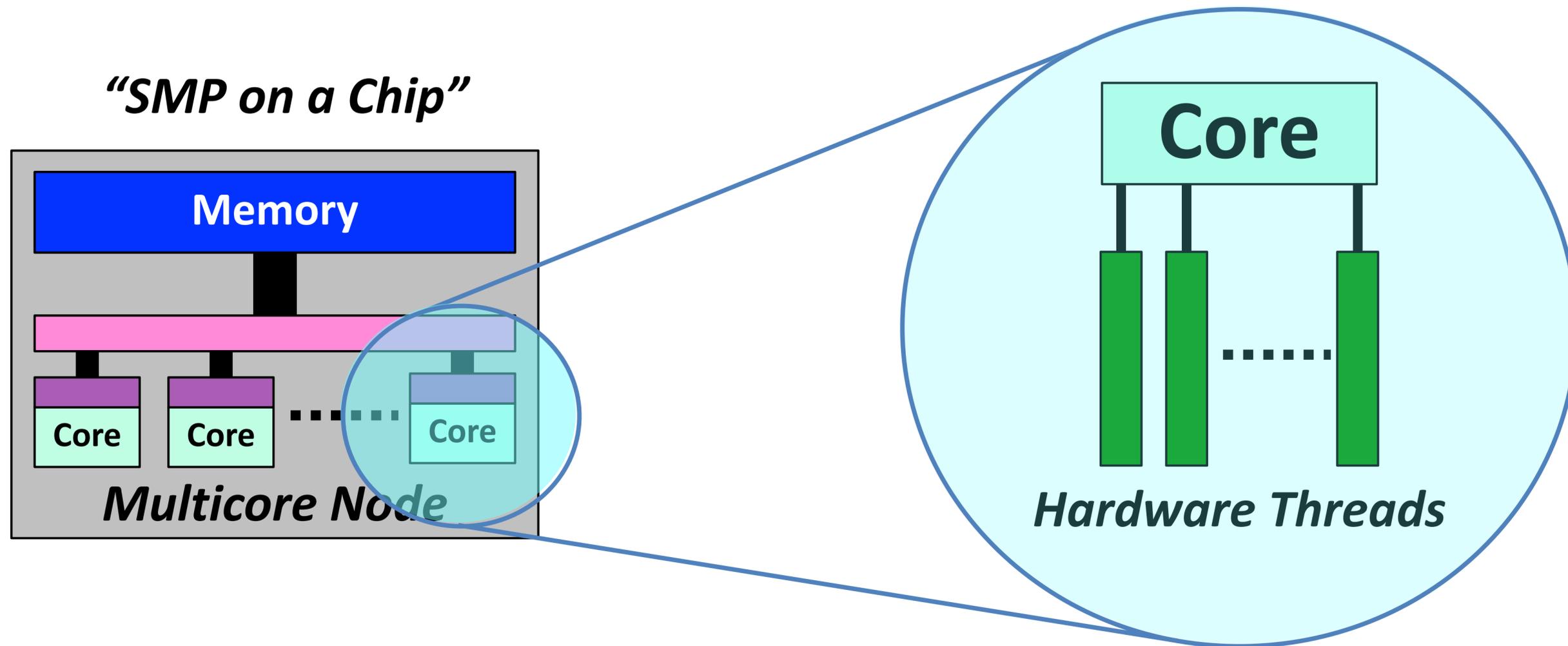


The unit of transfer is a "cache line"

A cache line contains multiple elements



Multicore and Hardware Threads



About Cores and Hardware Threads

A core may, or may not, support **hardware threads**

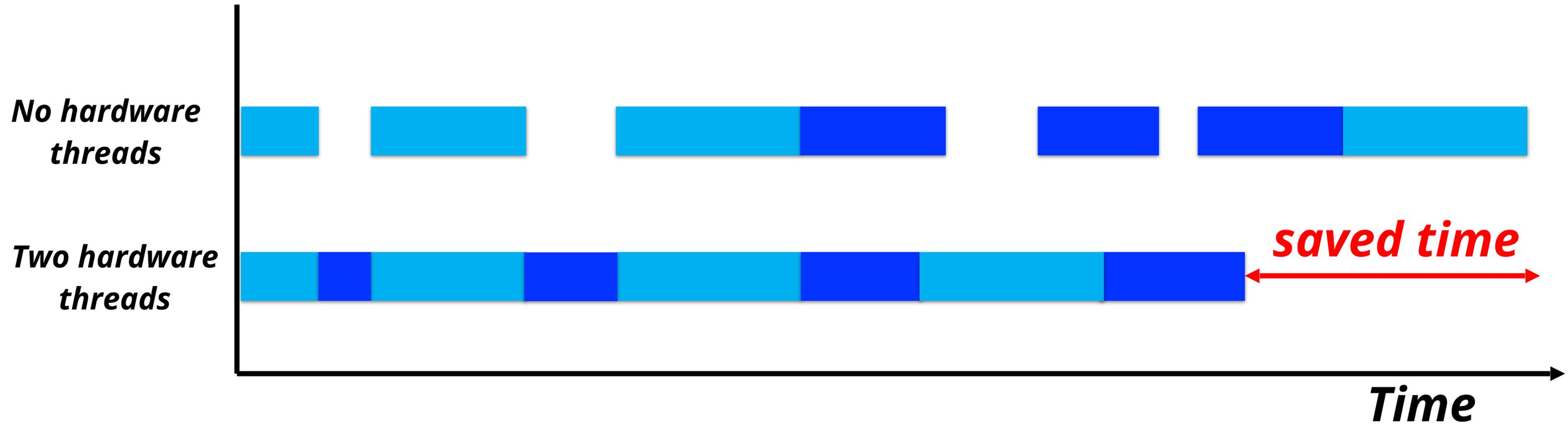
This is part of the design

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

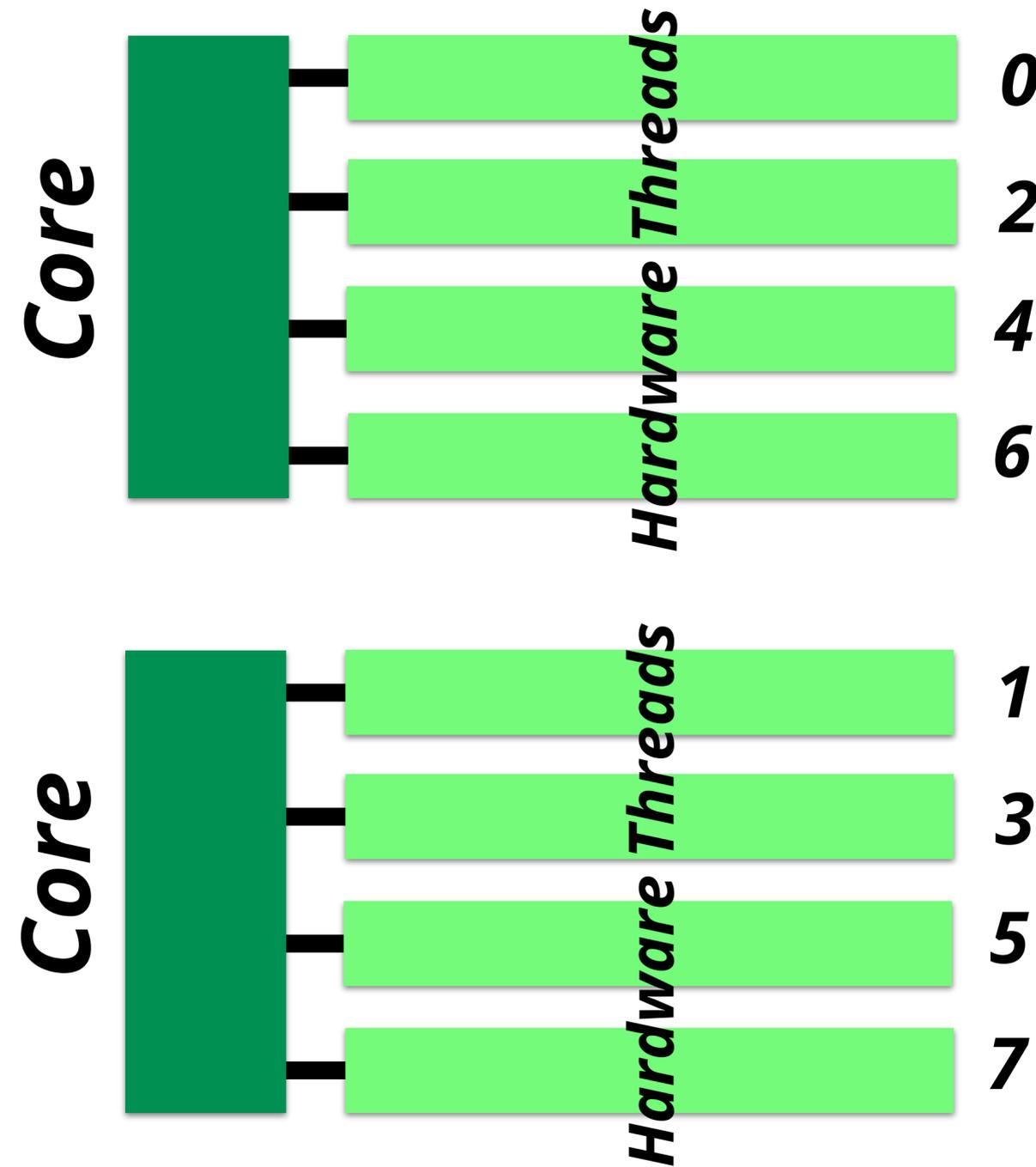
The idea is that the pipeline is used by another thread in case the current thread is idle

Each hardware thread has a unique ID in the system

How Hardware Threads Work



Hardware Thread IDs



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> }
```

*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*



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)***

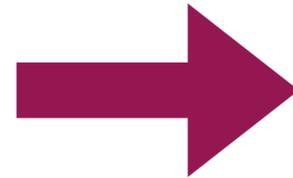


Another Example

```
#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
```



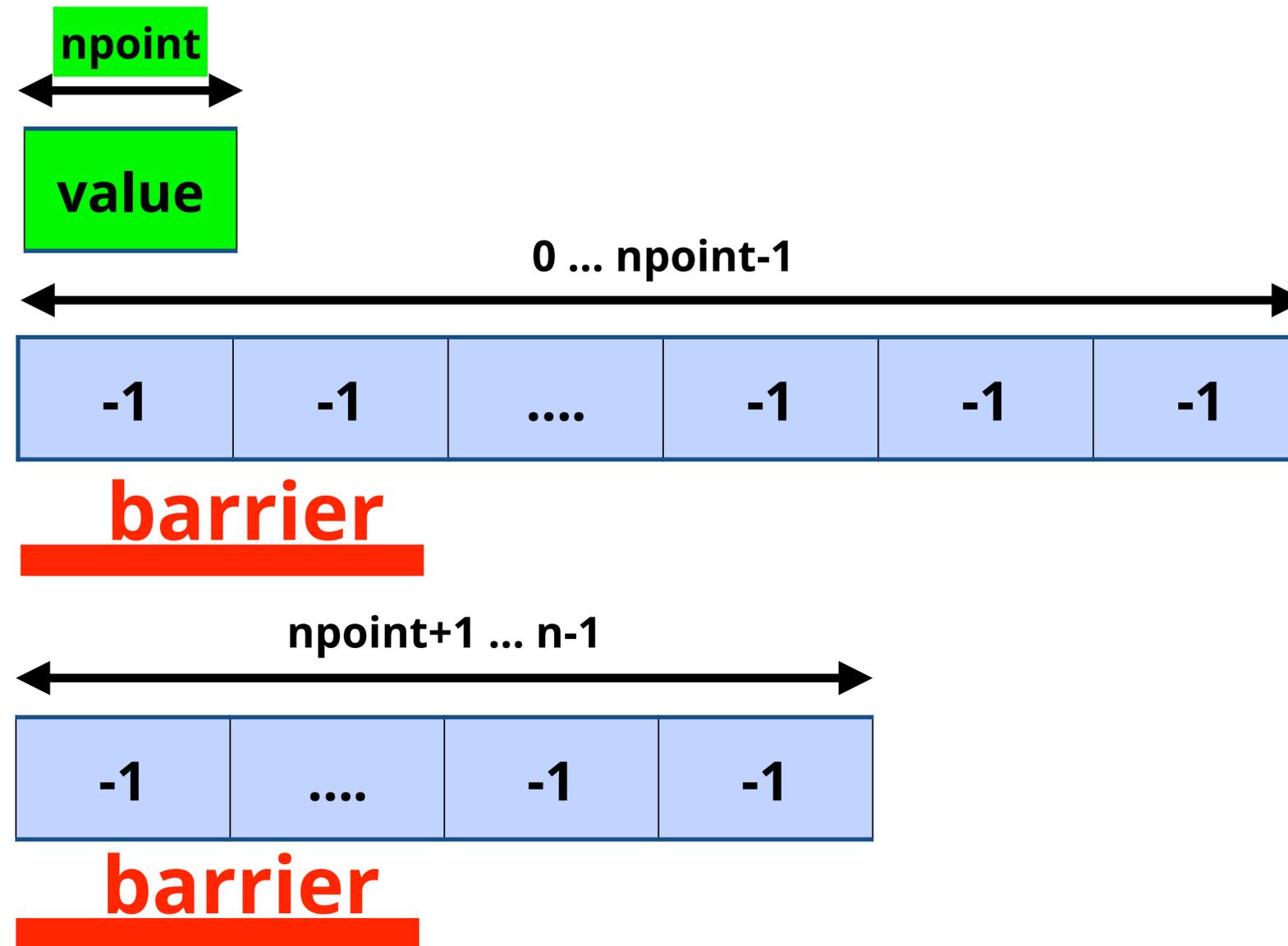
So What Is Wrong With This Then?

```
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
```

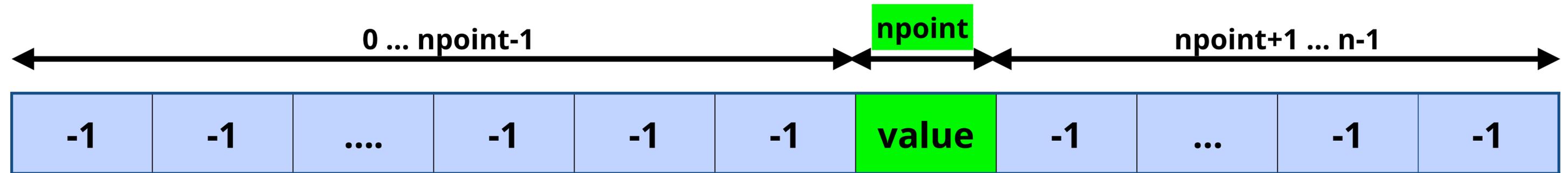
- ✓ *There are 2 barriers*
- ✓ *Two times the serial and parallel overhead*
- ✓ *Performance benefit depends on the value of variables "npoint" and "n"*



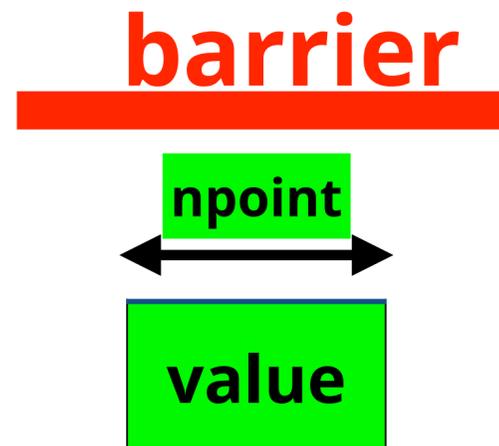
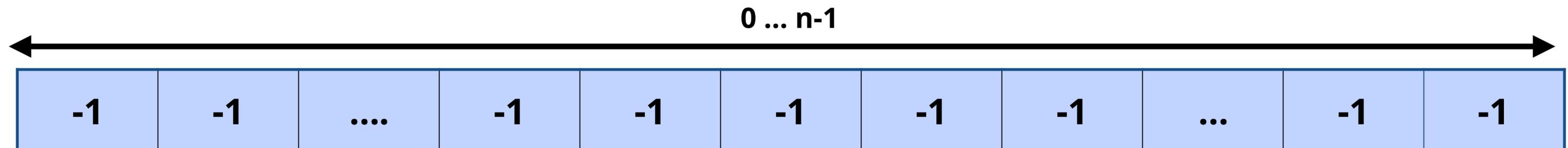
The Sequence Of Operations



The Final Result



↓ The Idea



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*
 - *Verify the result is a tree*
- Repeat <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")

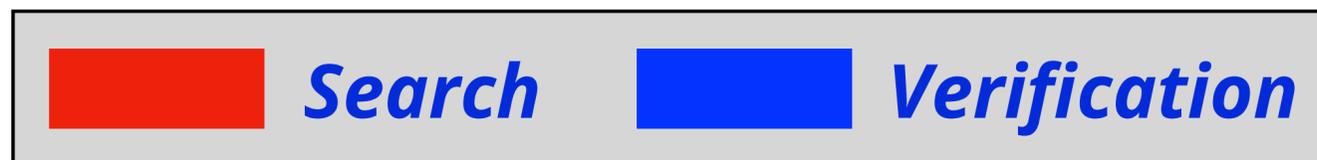
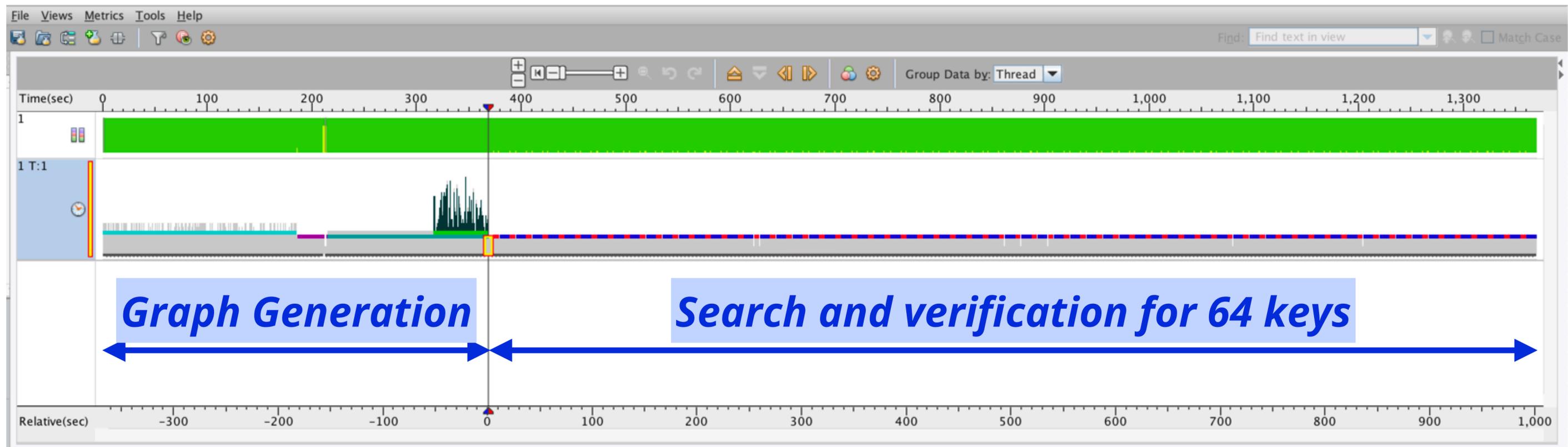
Used a VM instance with 8 Intel Skylake cores (16 threads)

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

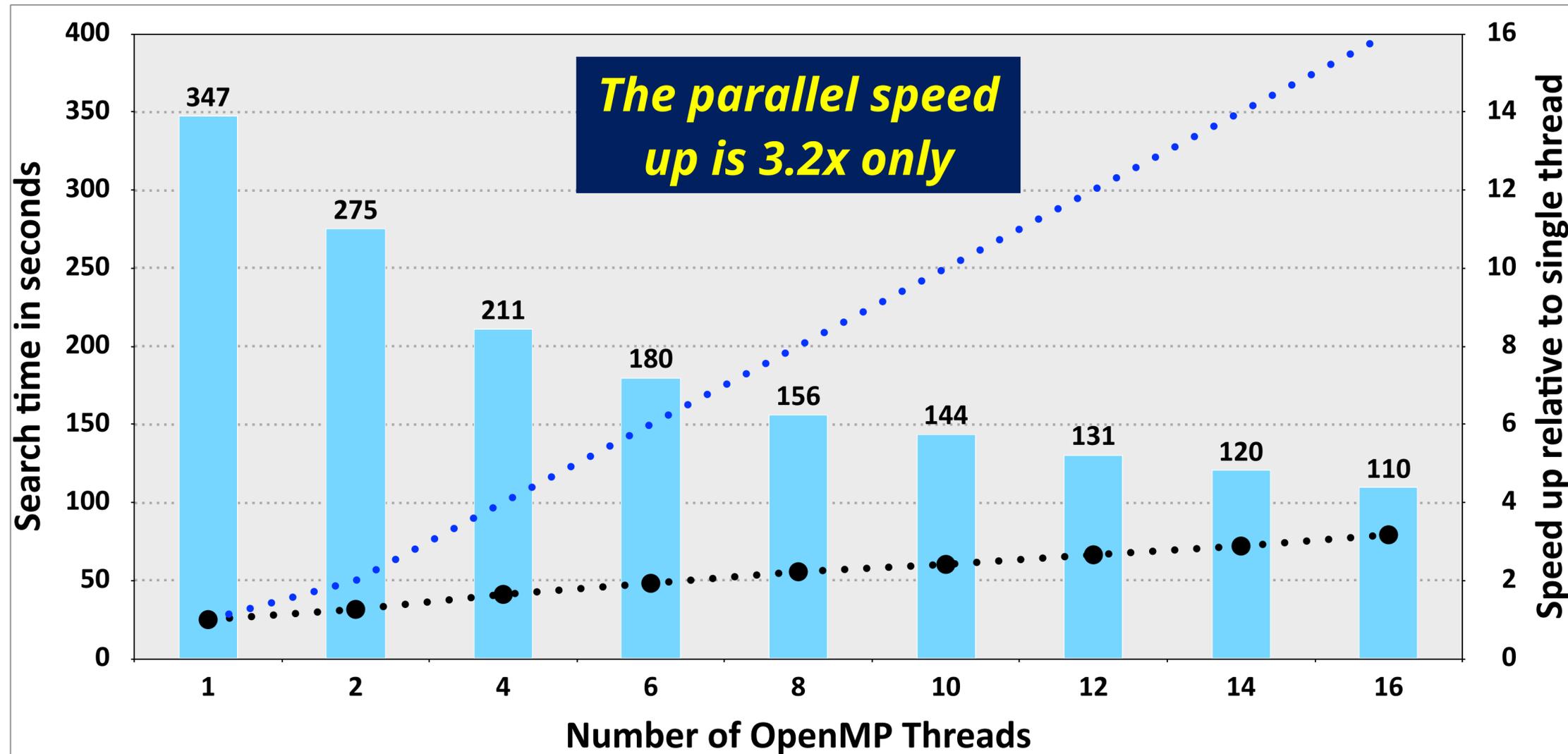
The gprofng profiling tool was used to make the profiles



The Dynamic Behaviour



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

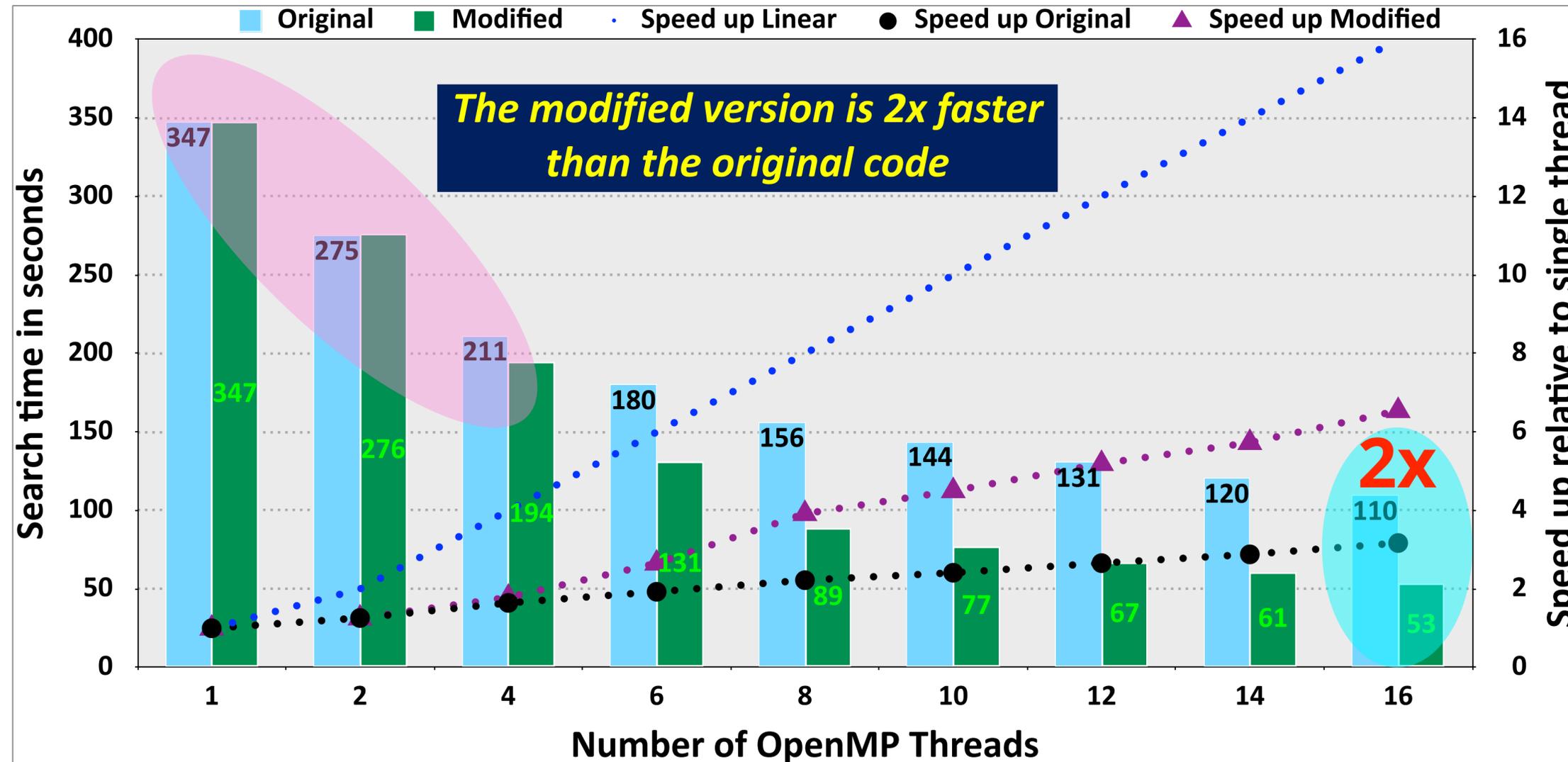
Found several opportunities to improve the OpenMP part

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?



Are We Done Yet?

The 2x reduction in the search time is encouraging

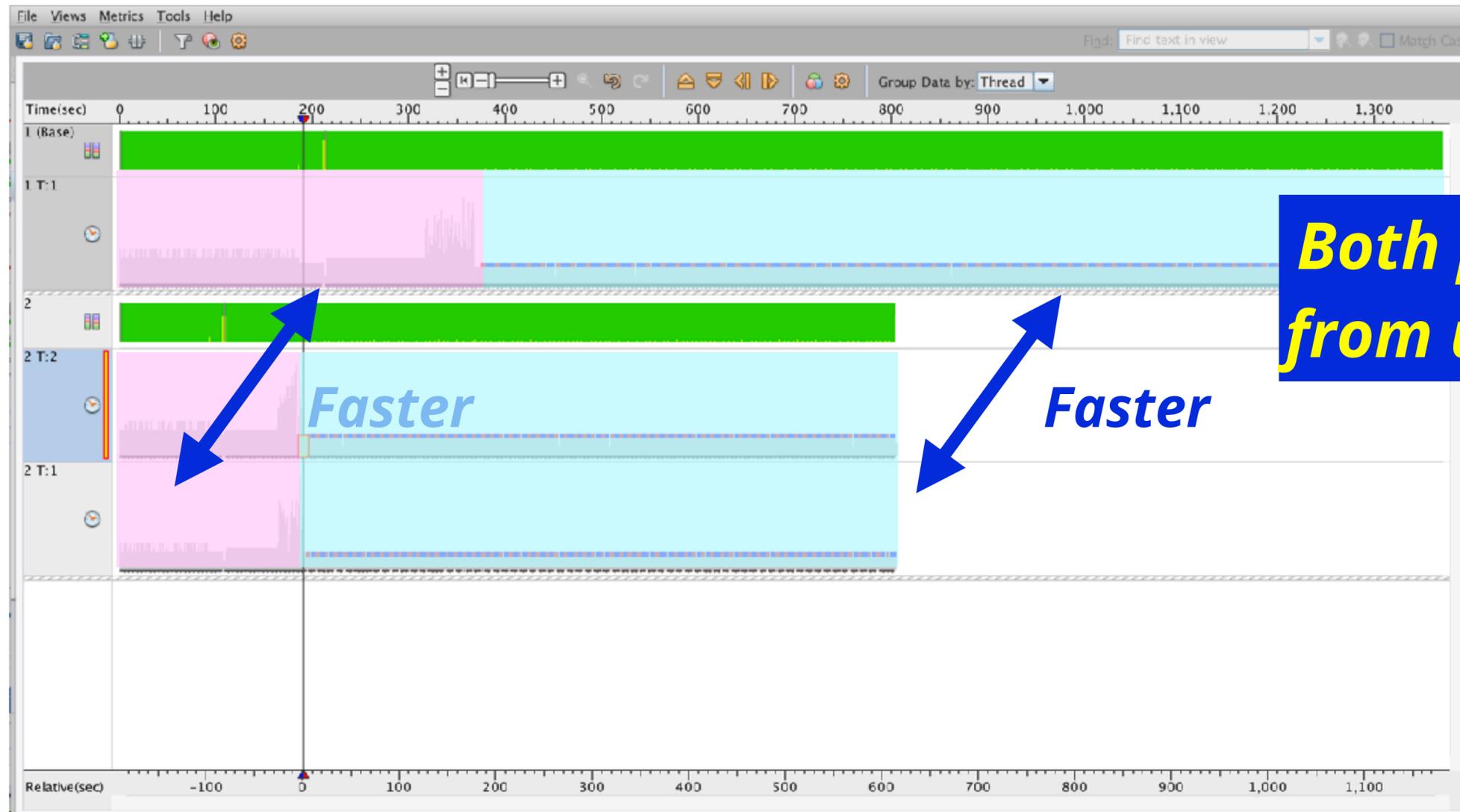
The efforts to achieve this have been limited

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

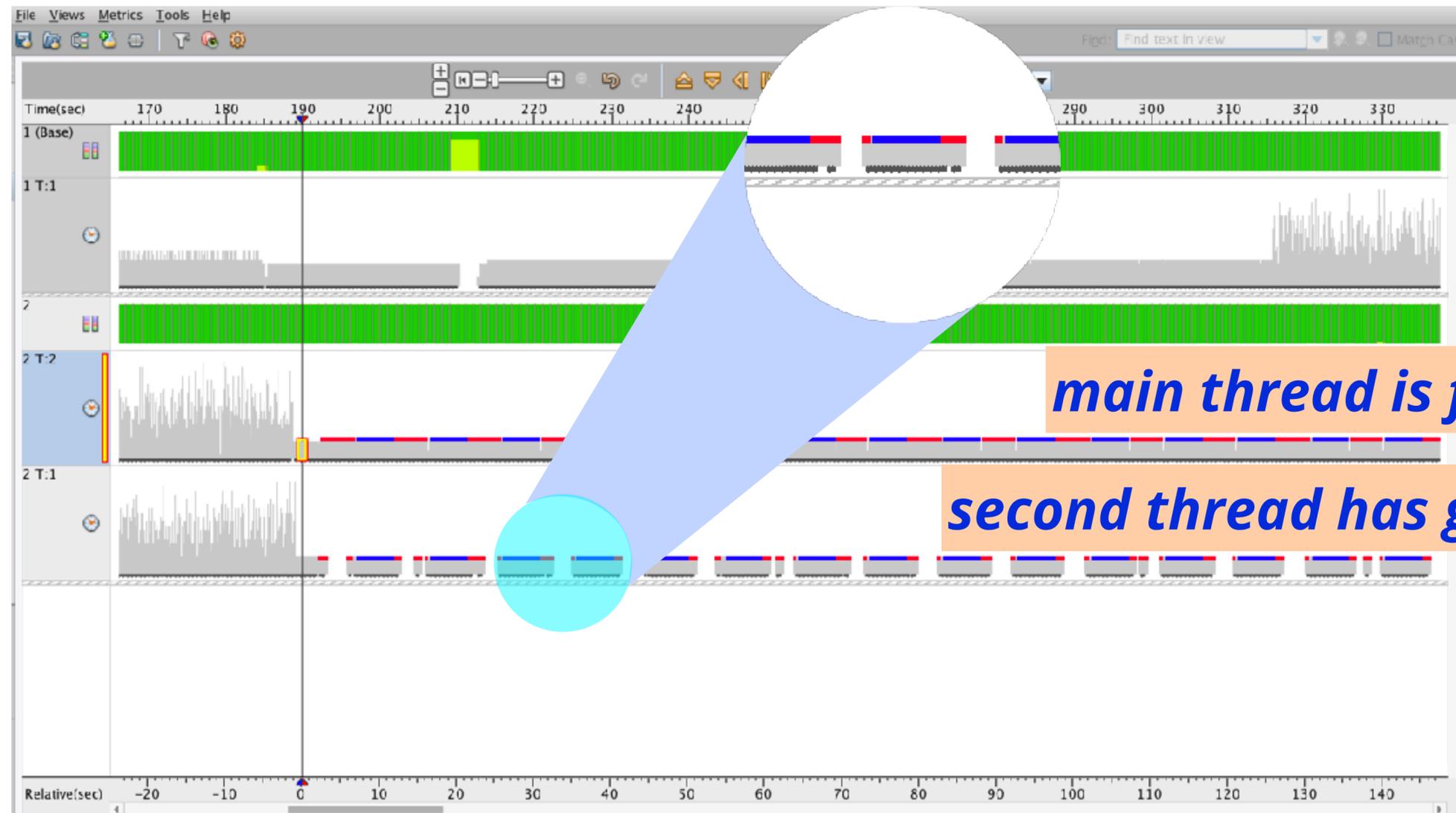
Faster

Faster

Search
Verification



Zoom In On The Second Thread

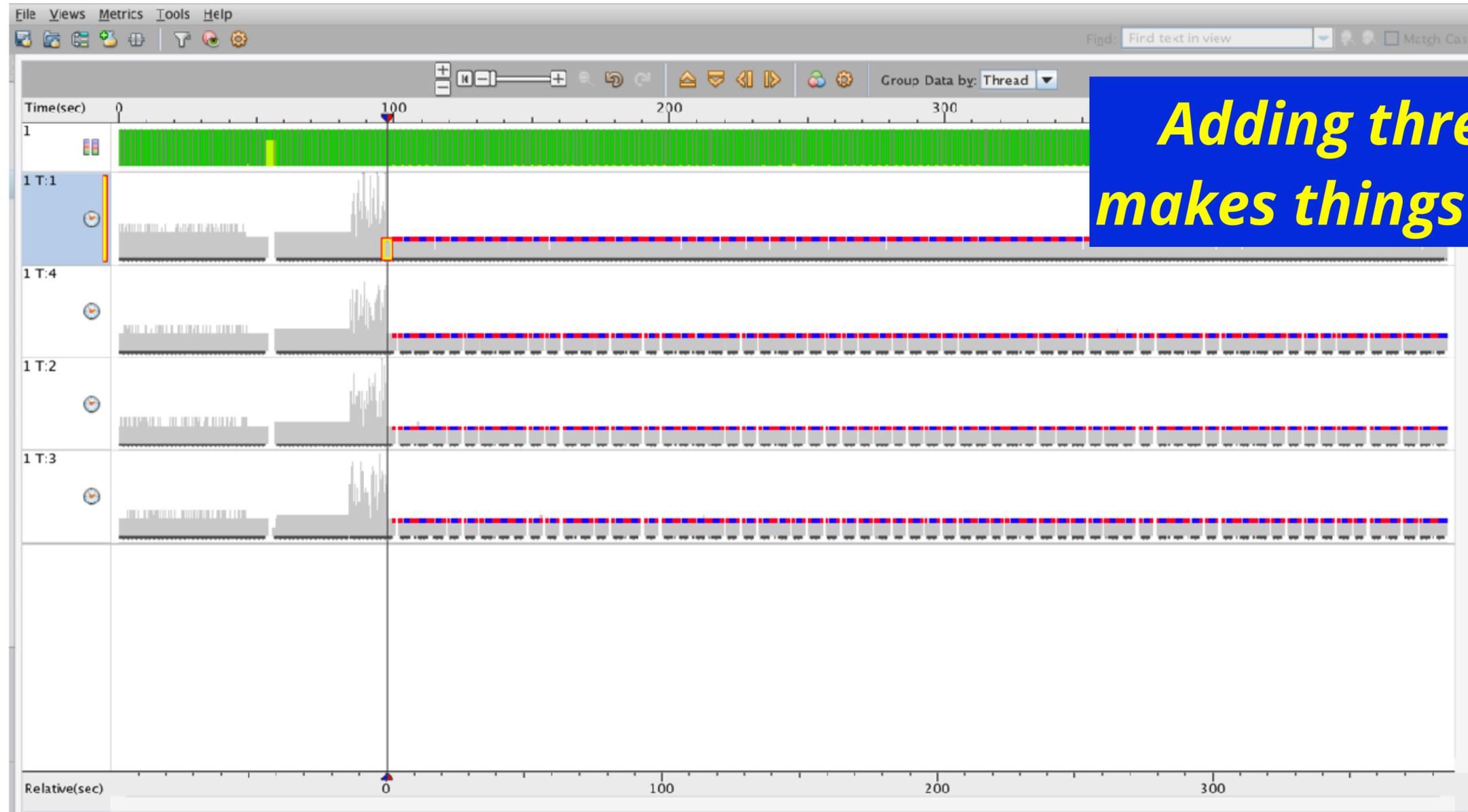


main thread is fine

second thread has gaps



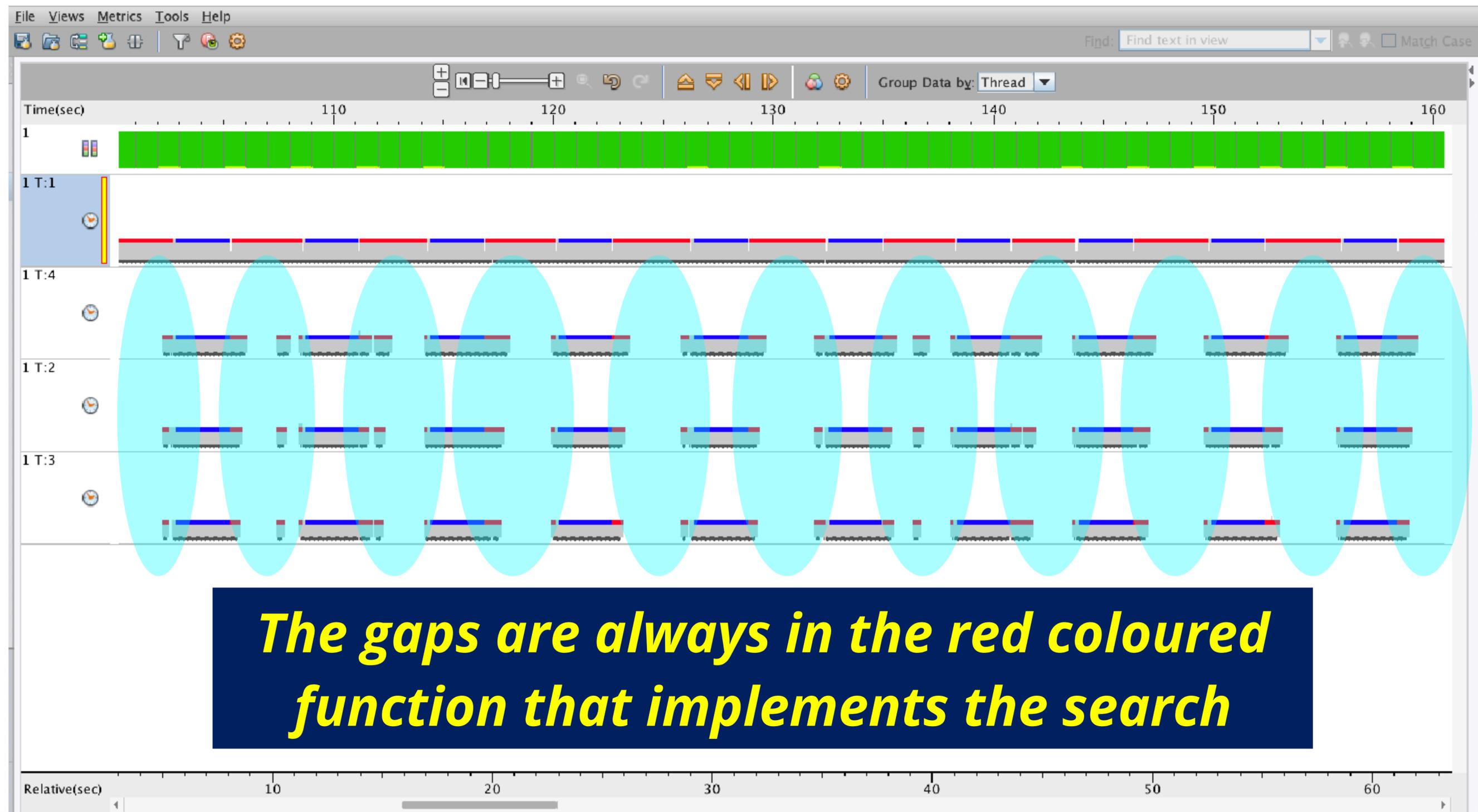
How About 4 Threads?



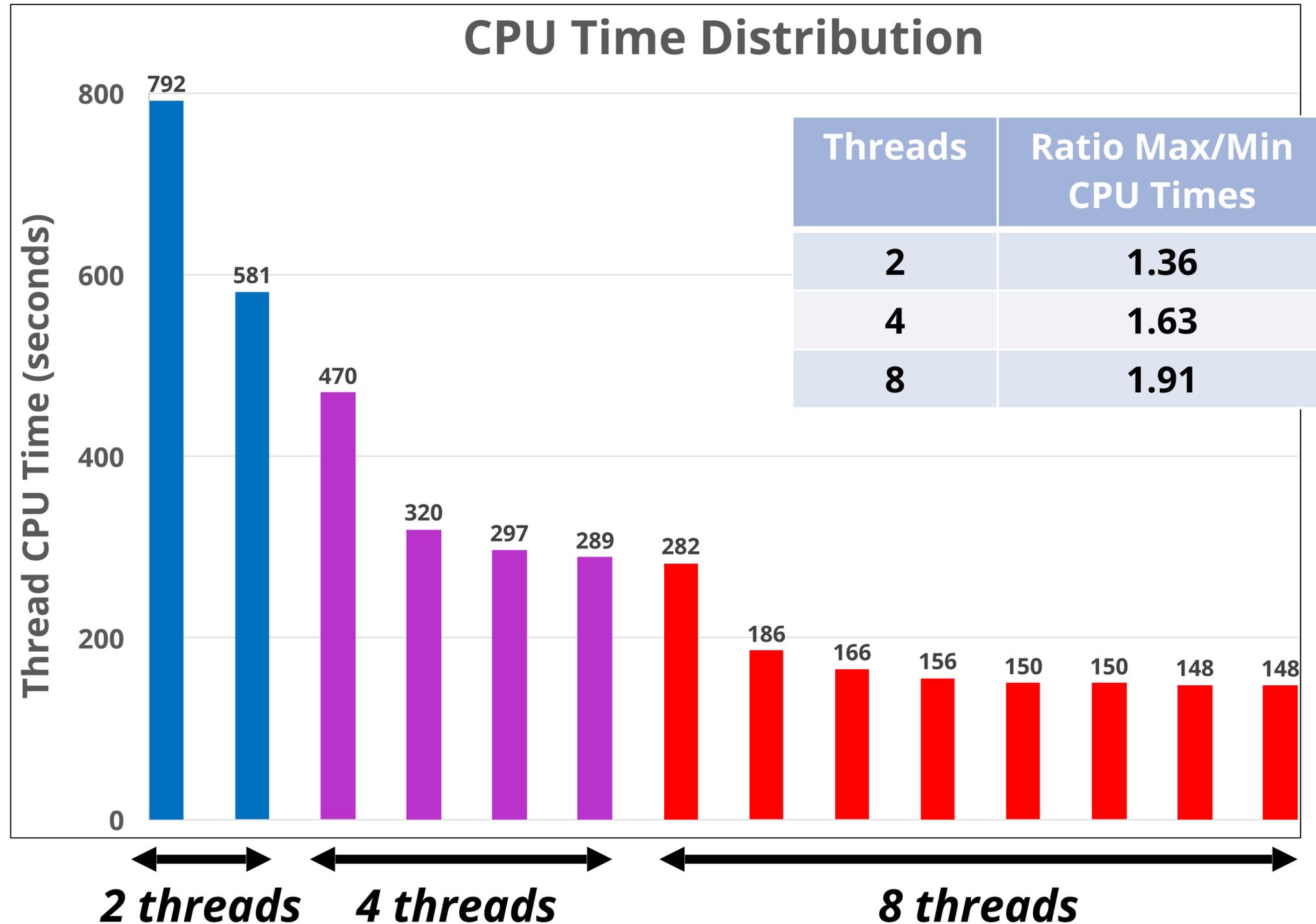
**Adding threads
makes things worse**



Zoom In Some More



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 = xadj[vo];
        if (bfs_tree[j] == 1) {
            if (int64_t cas = bfs_tree[j]) {
                if (kbuf < nbuf[kbuf]) {
                    nbuf[kbuf] = cas;
                } else {
                    int64_t cas2 = nbuf[kbuf];
                    assert(cas2 < cas);
                    for (int64_t i = 0; i < ADJ_LEN; ++vk) {
                        vlist[kbuf] = cas;
                        if (bfs_tree[kbuf] == 1) {
                            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
    } // End of for-loop on k
}
```

**Irregular workload
per k-iteration**

Fixed length loop

Irregular length loop

Irregular control flow



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

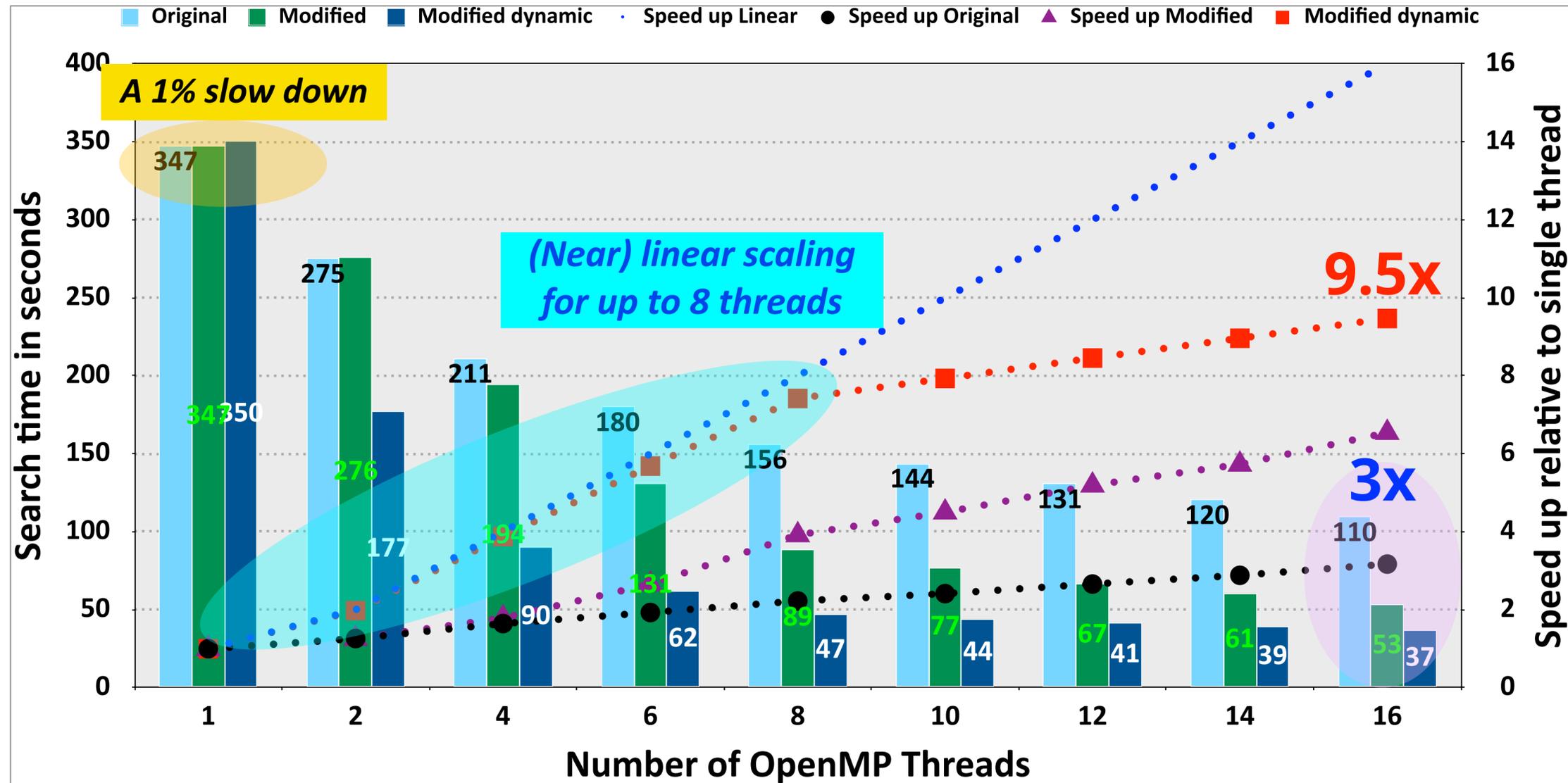
How Do You Know The Chunk Size Should Be 25?

~~***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



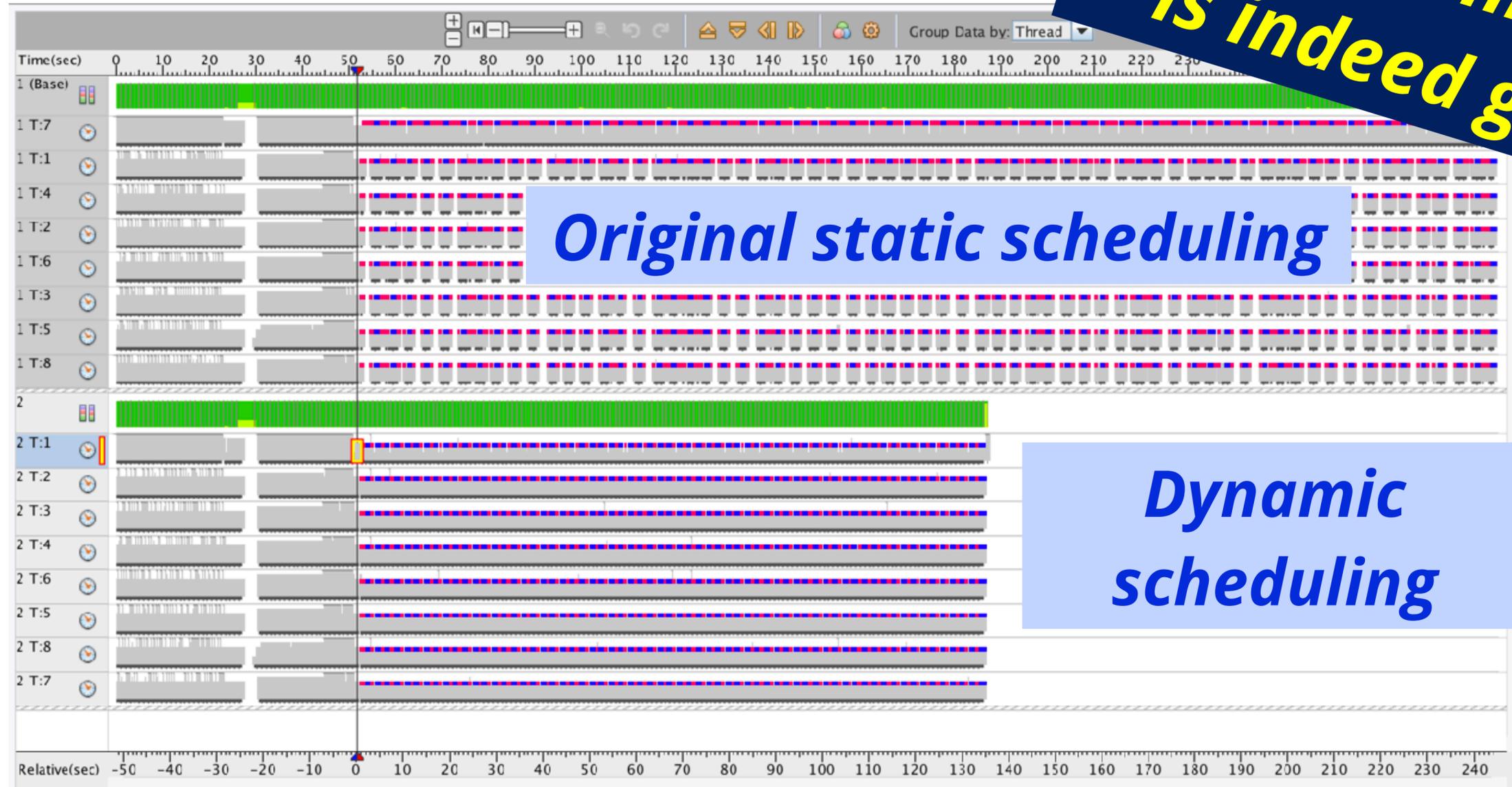
Really Important

Always Verify the Behaviour!

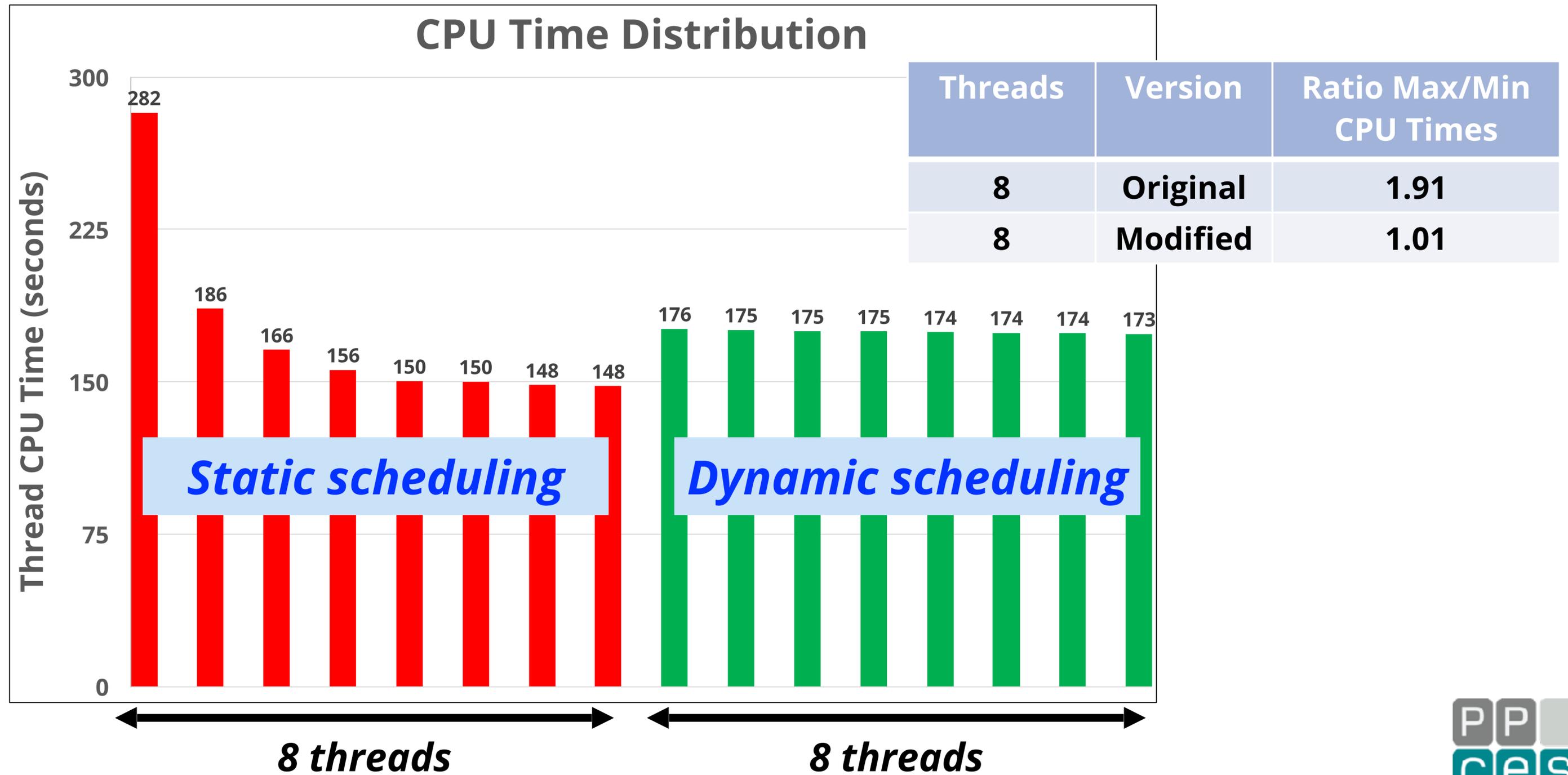


Before and After (8 Threads)

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, 2026



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
```

Vector a

0	1	2	3	
0.0				TID = 0
0.0		0.0		TID = 2
0.0	0.0	0.0		TID = 1
0.0	0.0	0.0	0.0	TID = 3



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***

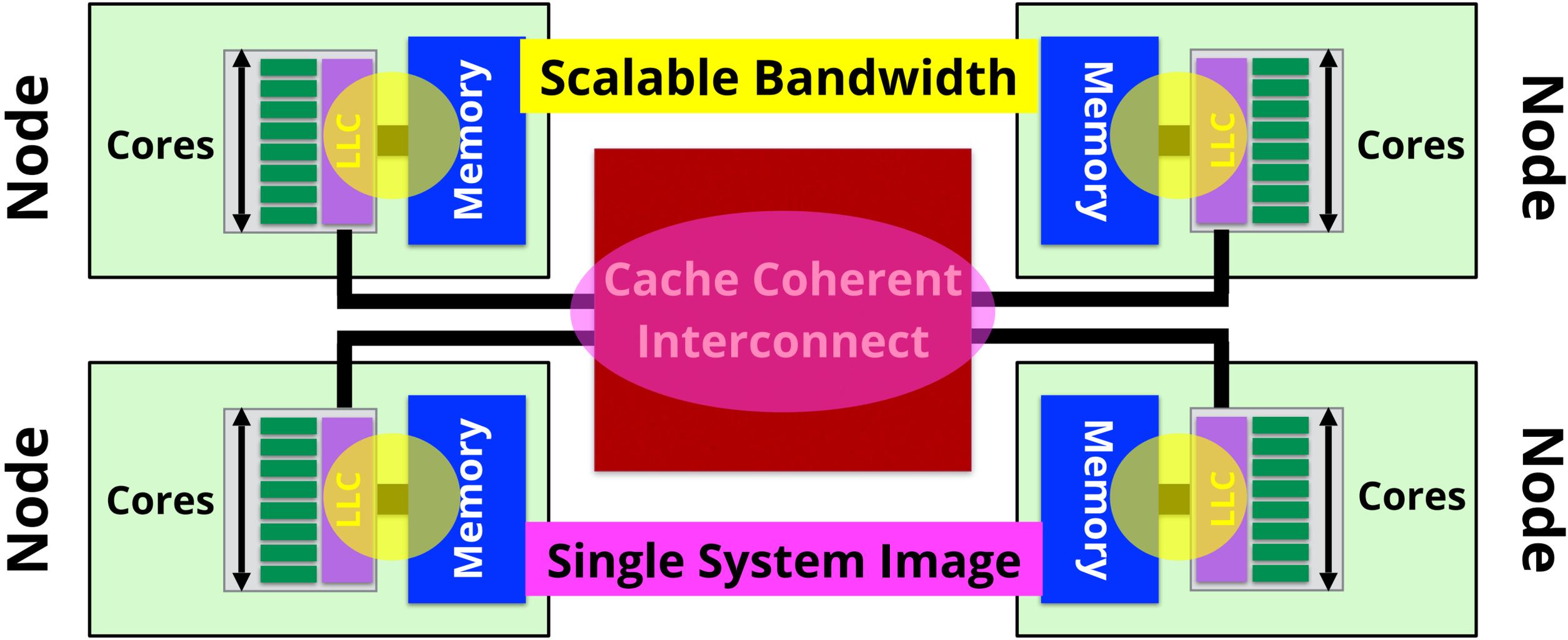
The tricky thing is that “things just work”

But do you know how efficiently your code performs?

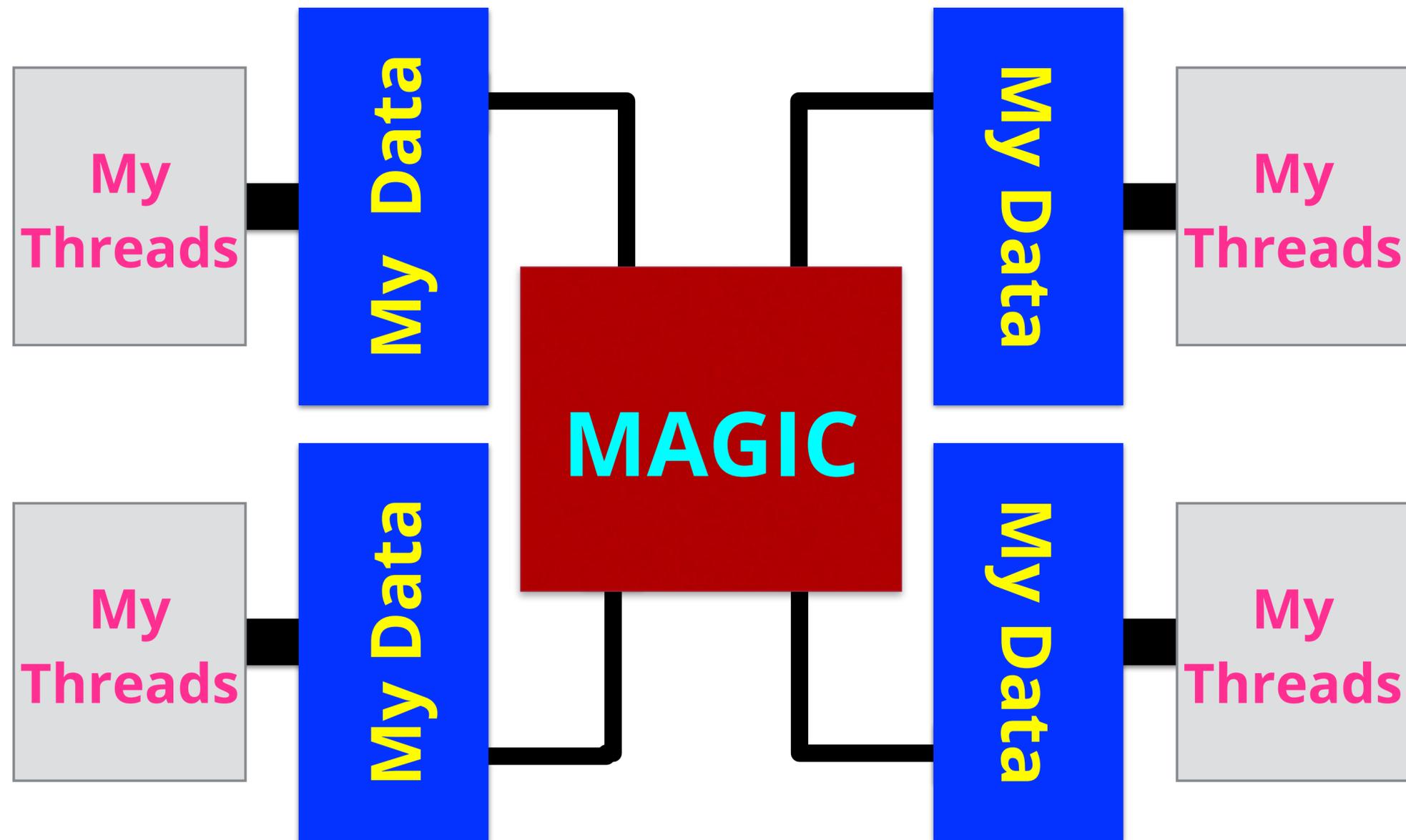


NUMA - The System Most of Us Use Today

A Generic, but very Common and Contemporary NUMA System



The Developer's View



The NUMA View

Memory is physically distributed, but logically shared

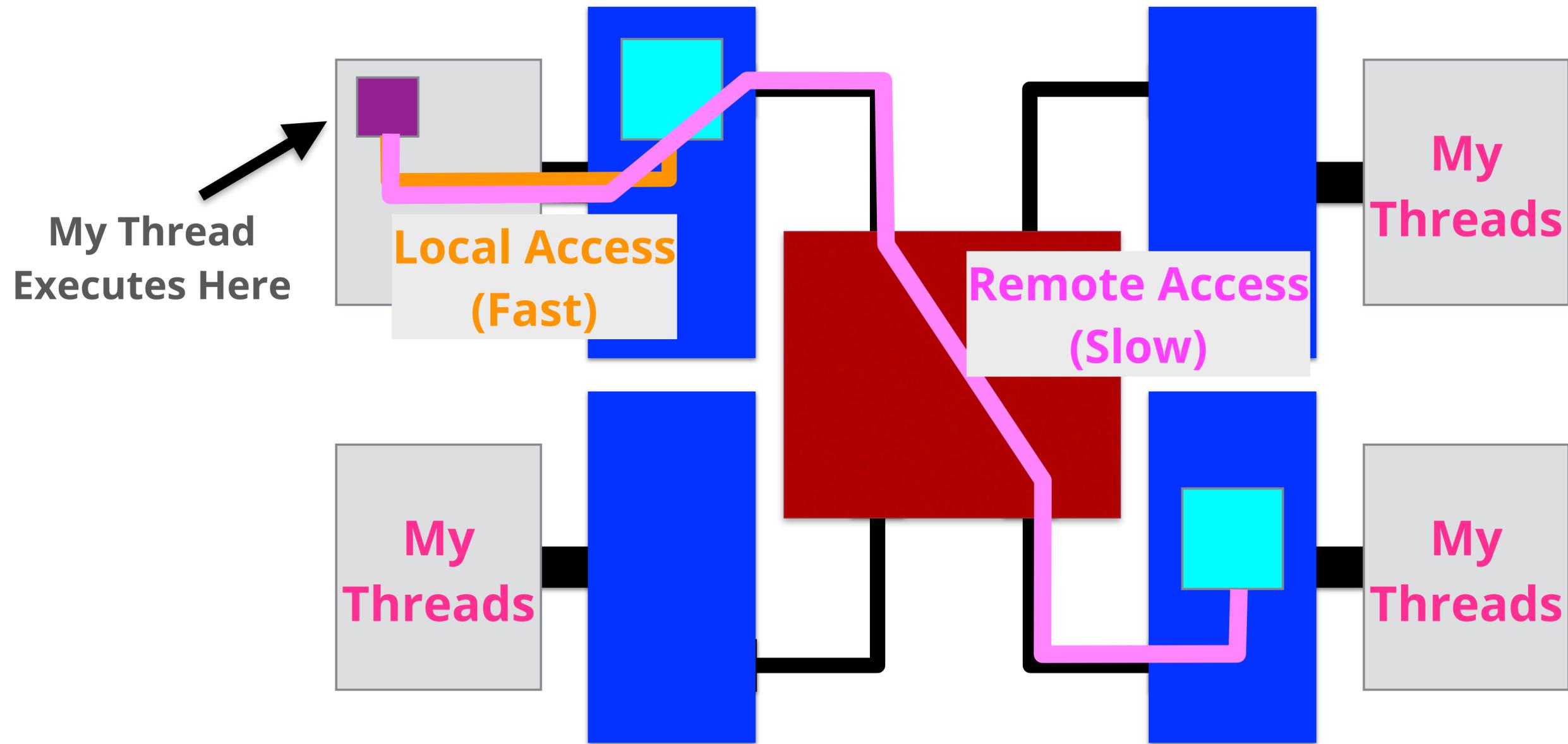
Shared data is accessible to all threads

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 memory access times for certain threads (and may also cause congestion on the network)

Luckily, the solution is (often) surprisingly simple



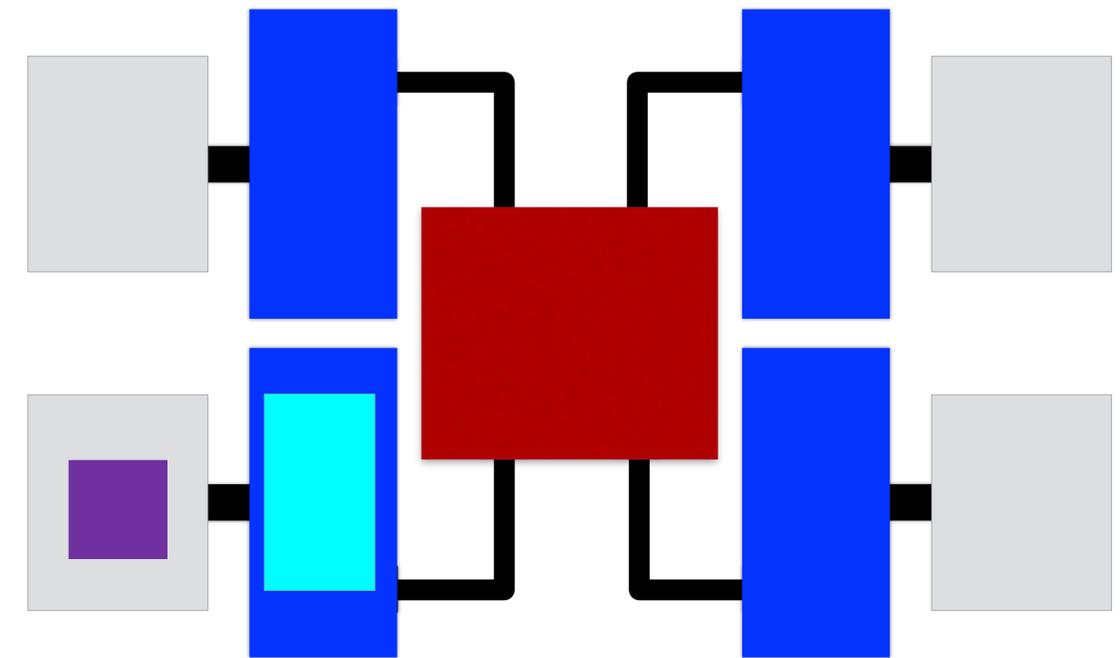
A Sequential Initialization

```
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



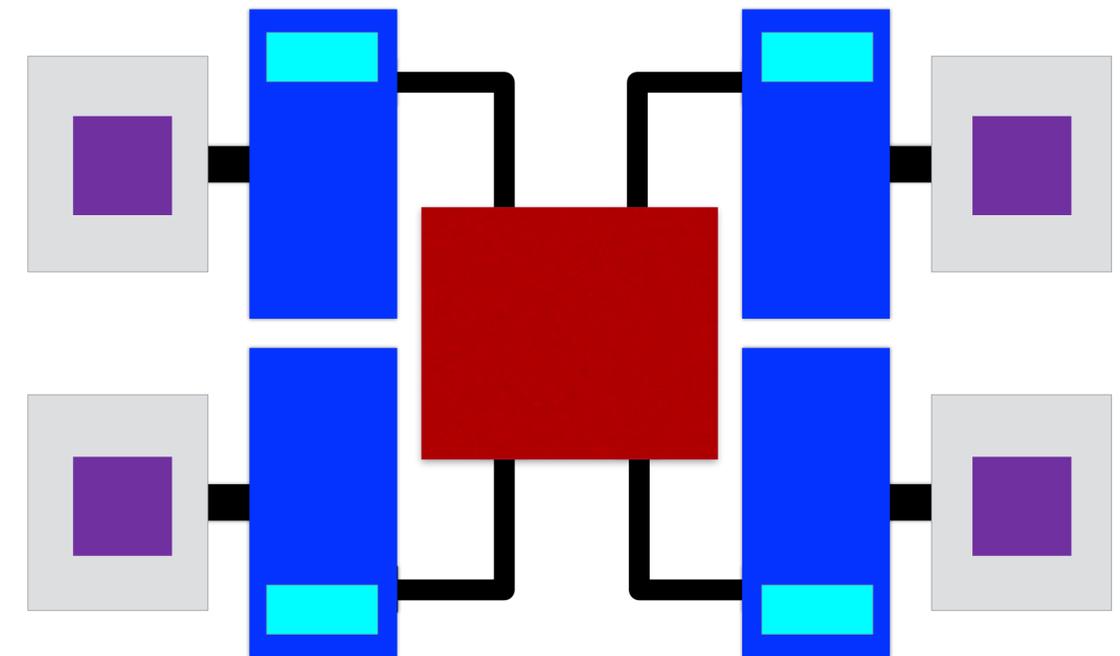
Leverage the First Touch Placement Policy

```
#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 (e.g. use the numactl tool)



About Memory Allocations

Do not use *calloc* for *global* memory allocation

Okay to use within a single thread



OpenMP Support for NUMA Systems



OpenMP Places

In a NUMA system, it matters where your threads and data are

*In OpenMP, **places** are used to **define where threads may run***

A place is defined by a symbolic name, or a set of numbers:

- *An example of a symbolic name: **cores***
- *An example of a set: **1, 5, 7, 11, 13***

Note that a mix of these two concepts is not allowed



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
<i>threads</i>	<i>A hardware thread</i>
<i>cores</i>	<i>A core</i>
<i>ll_caches</i>	<i>A set of cores that share the last level cache</i>
<i>numa_domains</i>	<i>A set of cores that share a memory with the same distance to that memory</i>
<i>sockets</i>	<i>A single socket</i>

Note: The number of places may be restricted - For example: `cores(4)`



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;
```

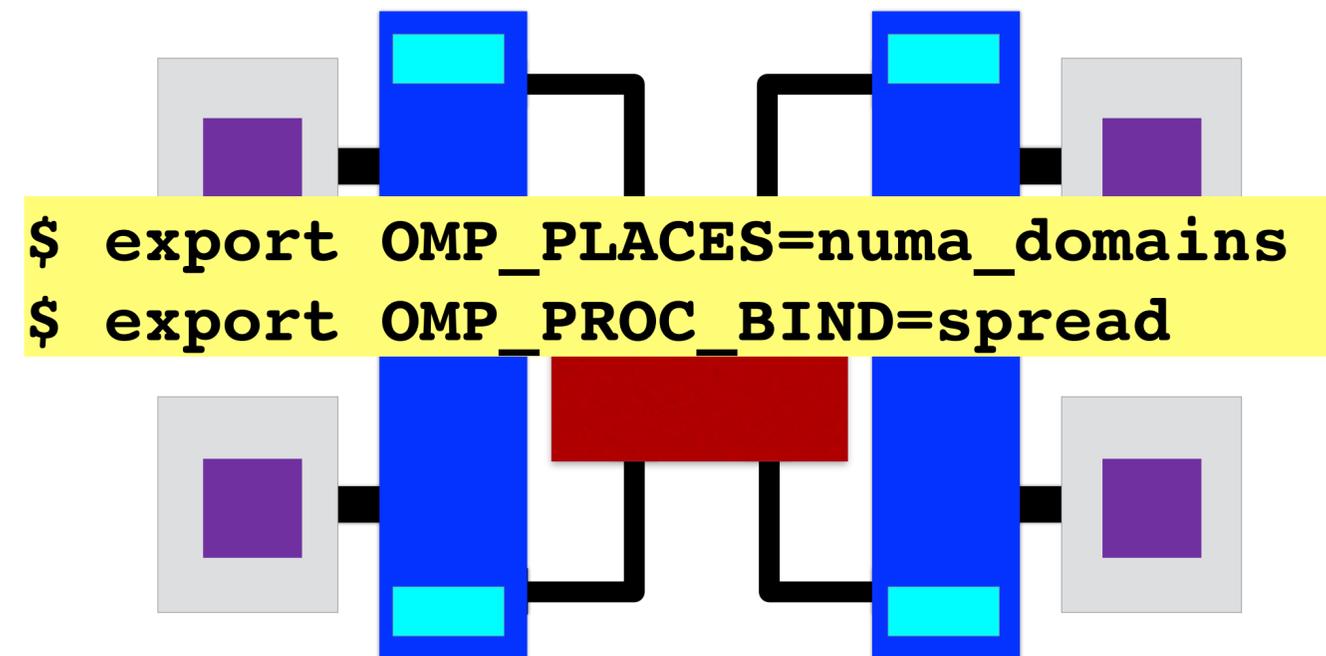
Four threads execute this loop



Wishful Thinking

*Data placement depends on
where threads execute*

Use the NUMA Controls



 = Thread

 = Data



NUMA Diagnostics

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

Two very simple, 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!

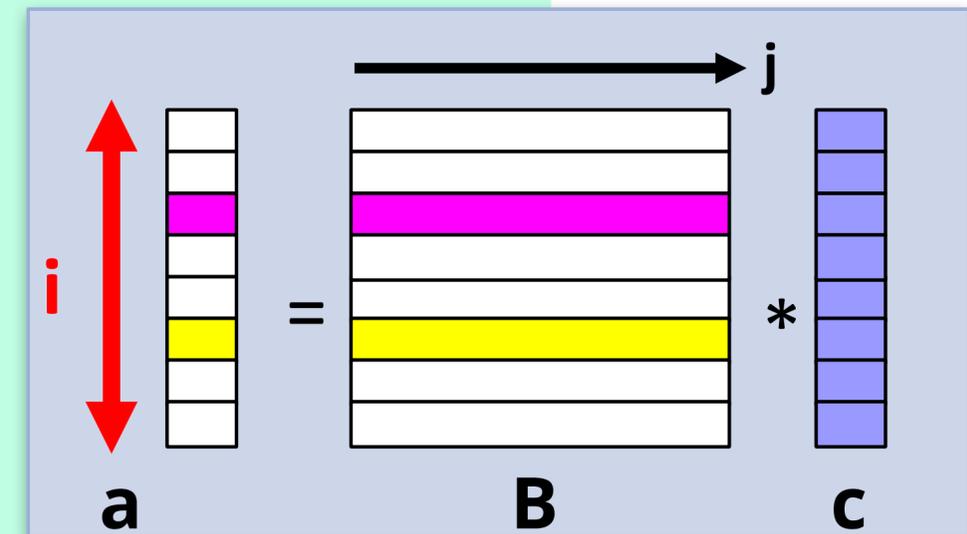


A Performance Tuning Example



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

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

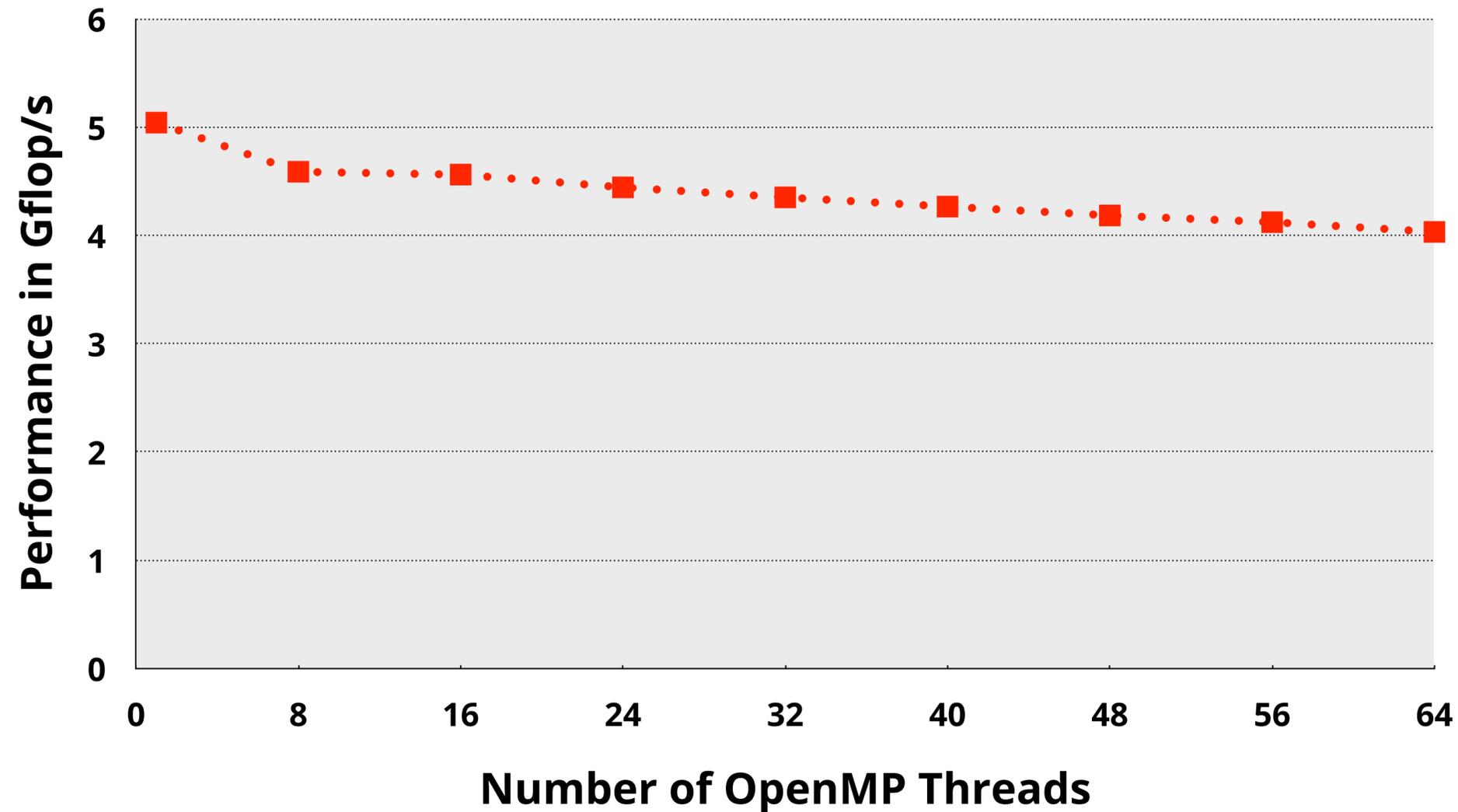


An embarrassingly parallel algorithm!
(on paper)



The Performance Using 64 Threads*

Performance of the matrix-vector algorithm (4096x4096)



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

**) The machine characteristics will be disclosed shortly*



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

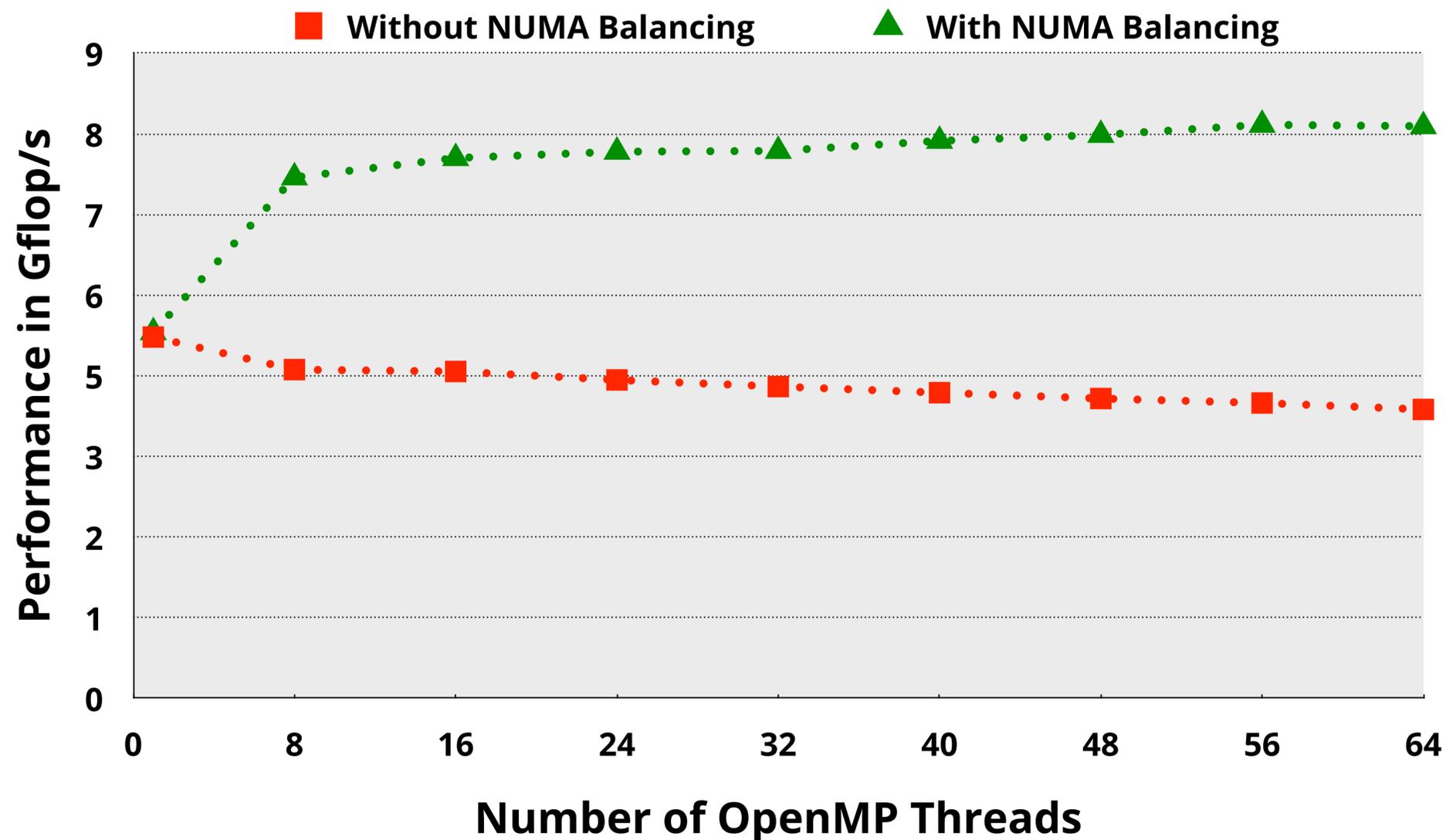
```
# echo 0 > /proc/sys/kernel/numa_balancing
```

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!



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 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
```

8 NUMA Nodes

2 columns => 2 hardware threads/core

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

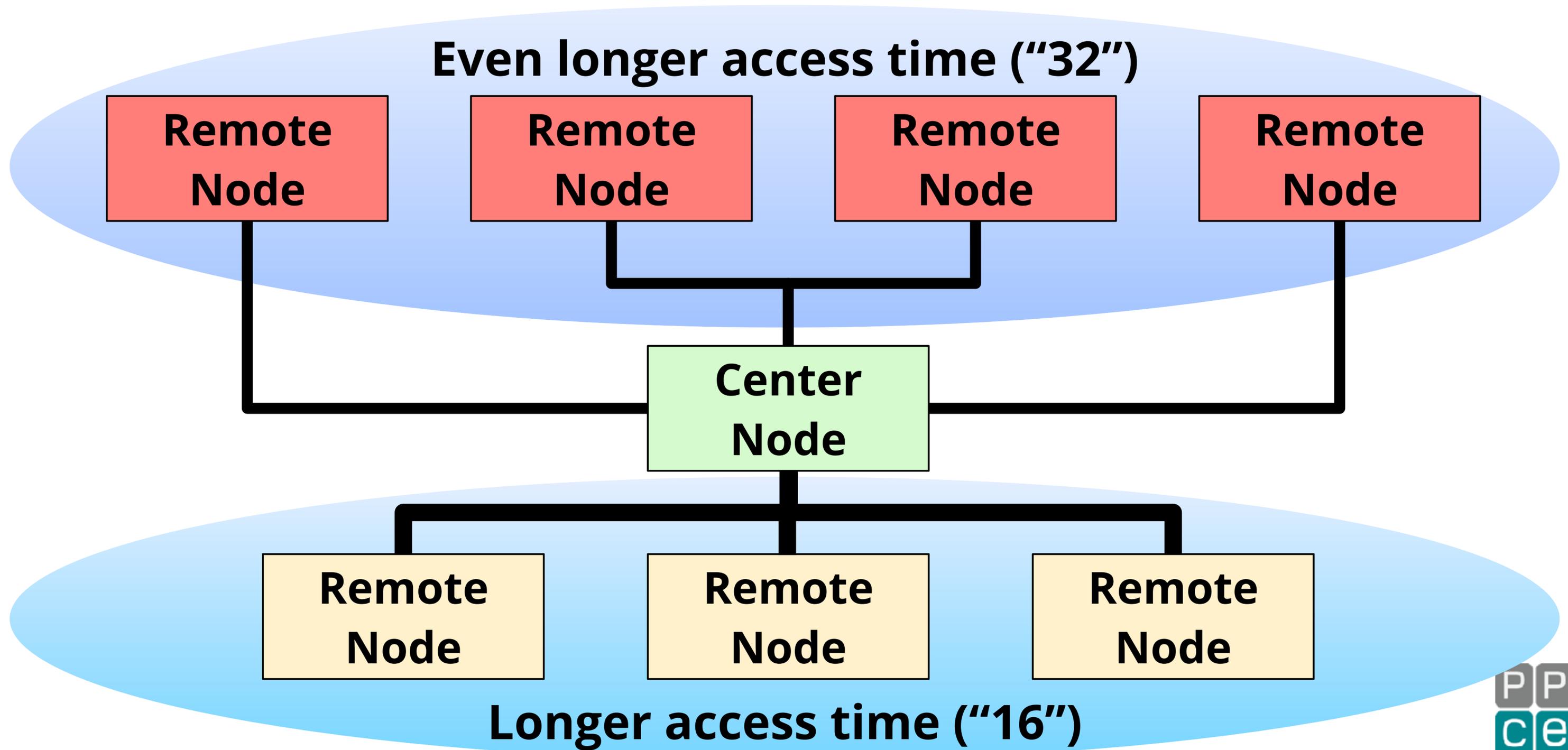


The NUMA Structure of the System

lscpu	There are 8 NUMA nodes
lscpu	There are 8 cores per node
lscpu	Each core has 2 hardware threads
numactl -H	Two levels of NUMA ("16" and "32")

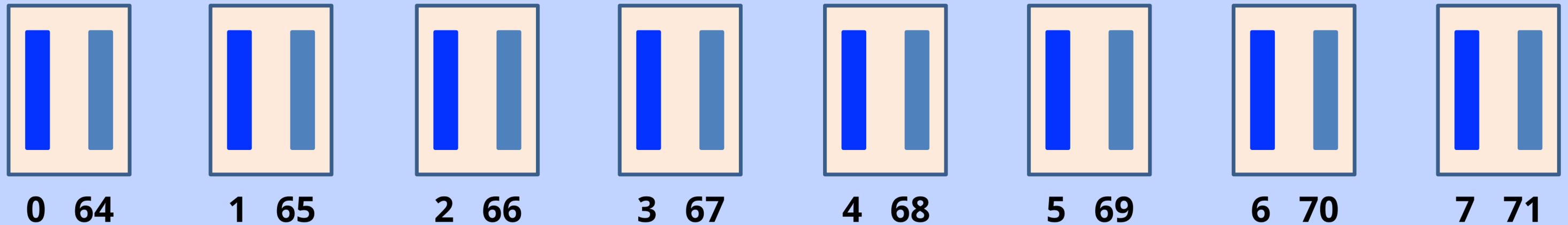


The Abstract System Topology



Example - NUMA Node 0 (lscpu output)

Memory



8 cores

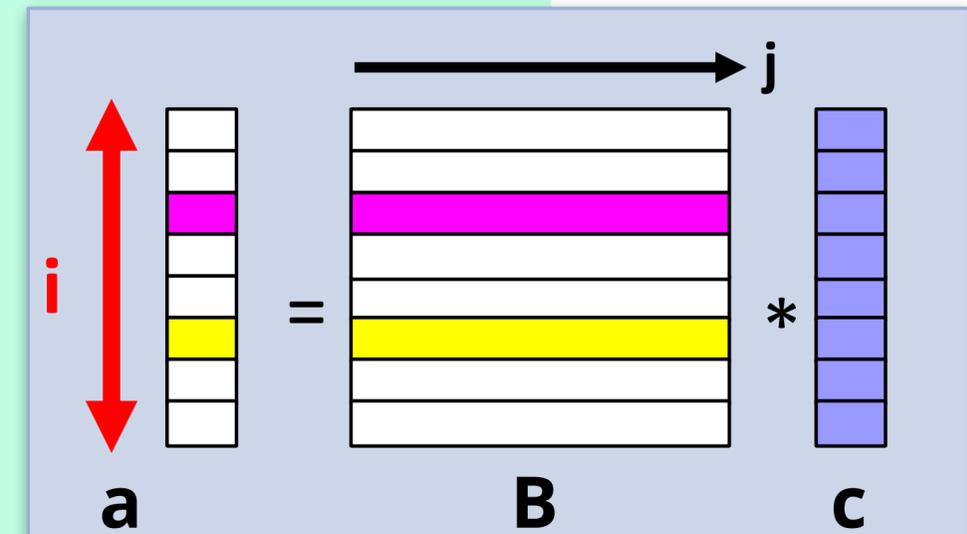
16 hardware threads

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) 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;  
}
```



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

Let's look at a more NUMA friendly data initialization



The Original Data Initialization

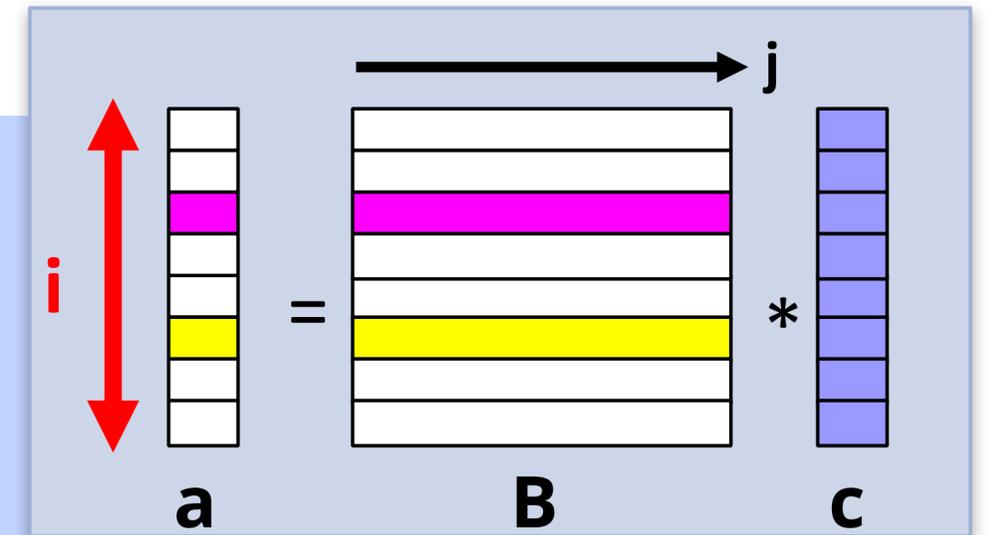
```
for (int64_t j=0; j<n; j++)
    c[j] = 1.0;

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 schedule(static)
    for (int64_t j=0; j<n; j++)
        c[j] = 1.0;
    #pragma omp for schedule(static)
    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
```

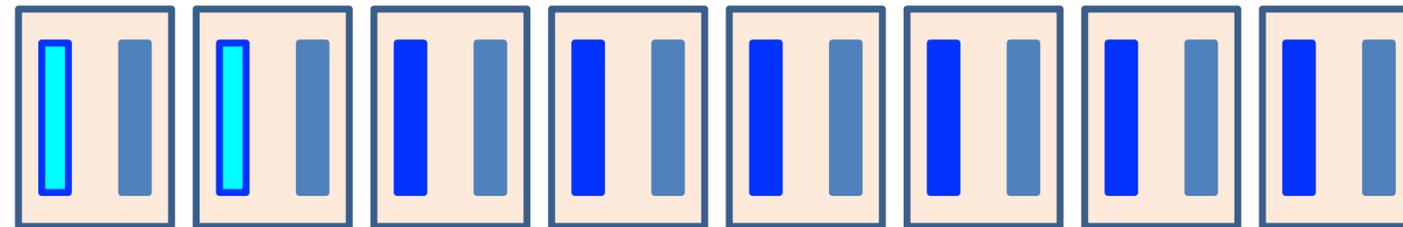


Control the Mapping of Threads

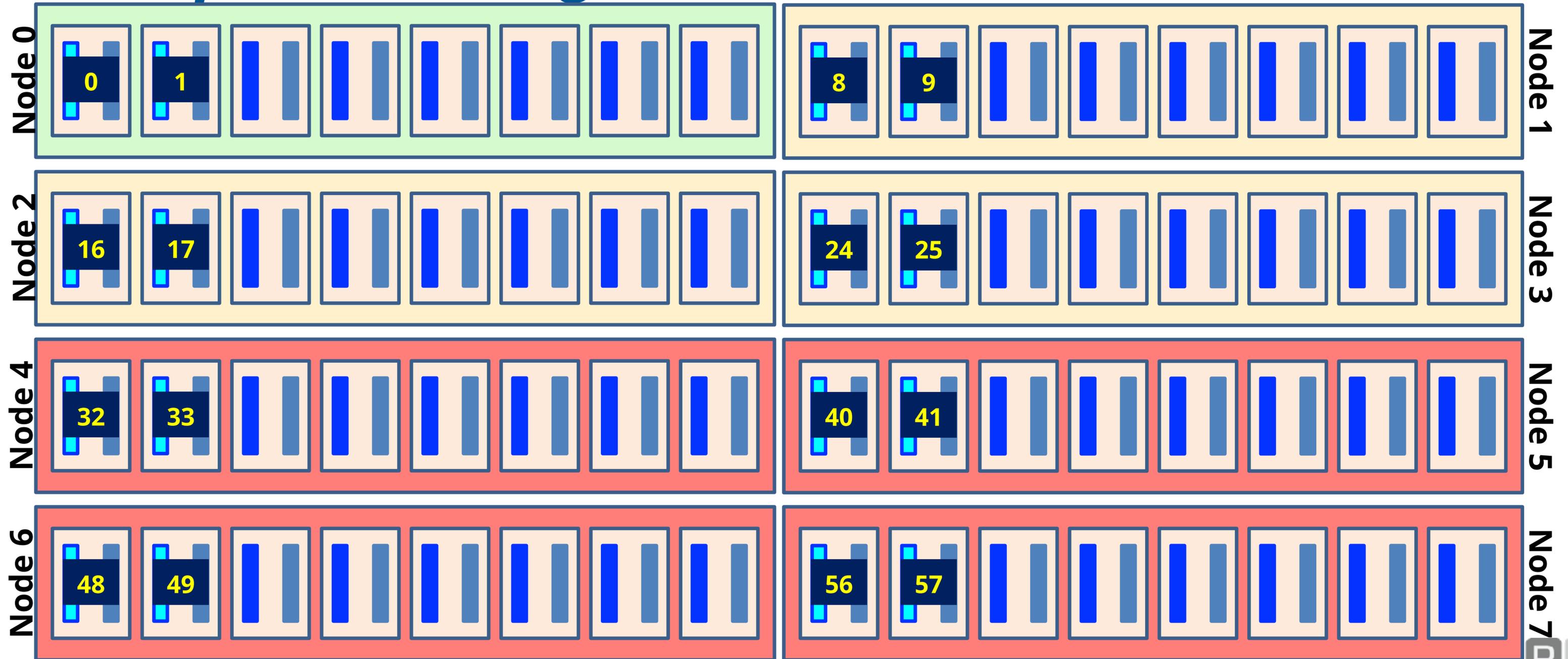
The Thread Placement Goal

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



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}*

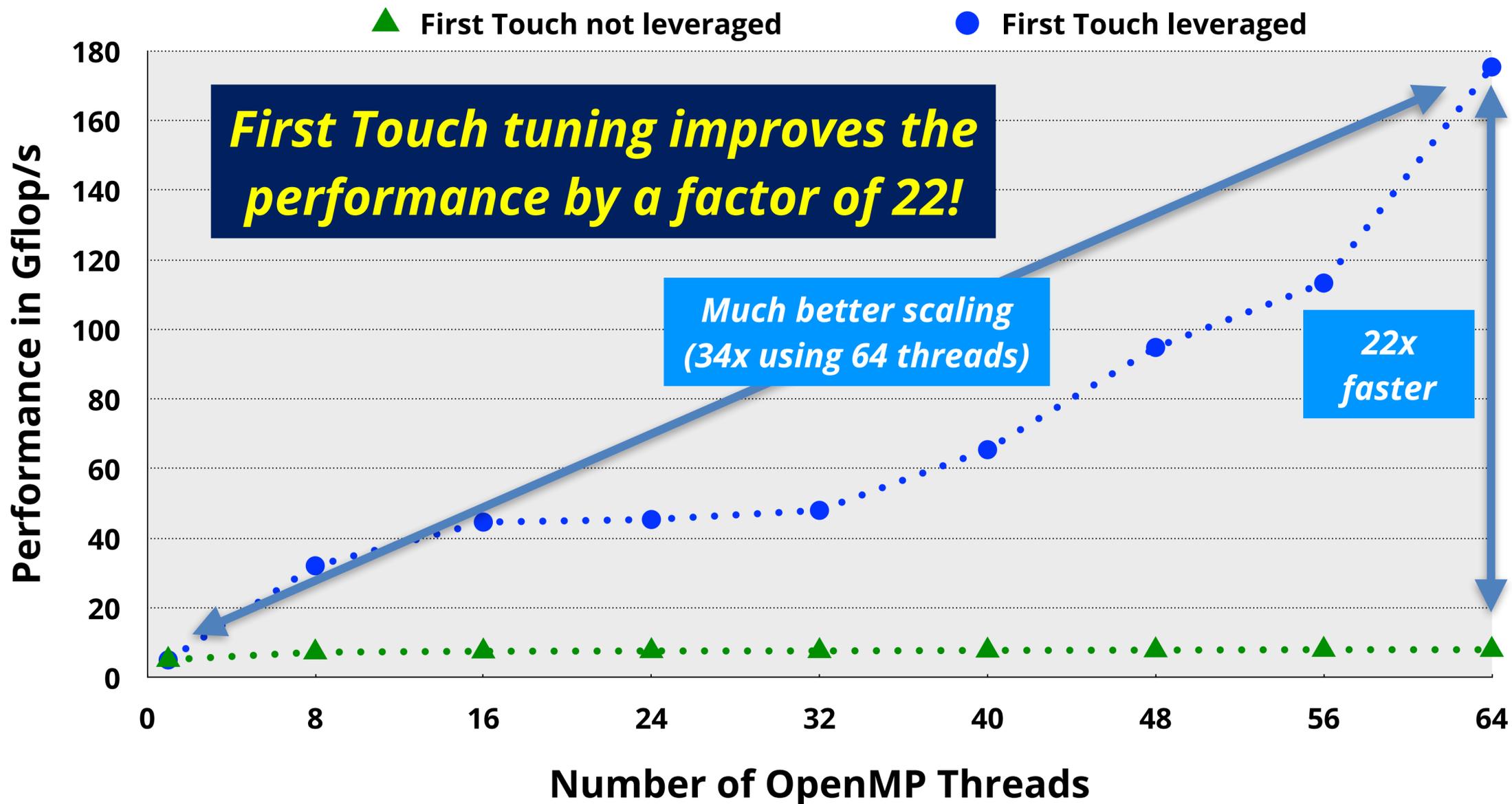
```
$ export OMP_PLACES={0}:8:8, {1}:8:8
$ export OMP_PROC_BIND=close
$ export OMP_NUM_THREADS=16
$ ./a.out
```

```
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
```

Note: Setting OMP_DISPLAY_ENV=verbose is your friend here!



The Performance for a 4096x4096 matrix



Performance in Gflop/s

Threads	No Leverage First Touch	Leverage First Touch	Benefit of First Touch
1	5,1	5,1	1,0
56	8,0	113,3	14,2
64	8,0	175,4	21,9
Speed up	1,6	34,4	

Recall that the only difference is in the initialization of the data

Oracle Linux with the gcc compiler
 2 socket system (2 AMD EPYC 7551 with 64 cores)
 NUMA balancing on; negative scaling for version without FT and balancing off



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 a frustrating and 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