"What Could Possibly Go Wrong Using OpenMP?"

Ruud van der Pas

Senior Principal Software Engineer

Oracle Linux and Virtualization Engineering

Oracle, USA

OpenMP Training Series

August 5, 2024

NERSC, Berkeley, CA, USA

$ whoishe

My background is in mathematics and physics

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

Currently I work in the Oracle Linux Engineering organization

I have been involved with OpenMP since the introduction

I am passionate about performance and OpenMP in particular

Copyright (©) 2024 by Ruud van der Pas

Outline

Prologue

Part I - What Could Possibly Go Wrong Using OpenMP?

Part II - The Joy Of Computer Memory

Q and (some) A

Copyright (©) 2024 by Ruud van der Pas

Prologue

Copyright (©) 2024 by Ruud van der Pas

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

Copyright (©) 2024 by Ruud van der Pas

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

Copyright (©) 2024 by Ruud van der Pas

Things You Need To Know

Copyright (©) 2024 by Ruud van der Pas

About Caches

Caches are fast buffers, used for data and instructions

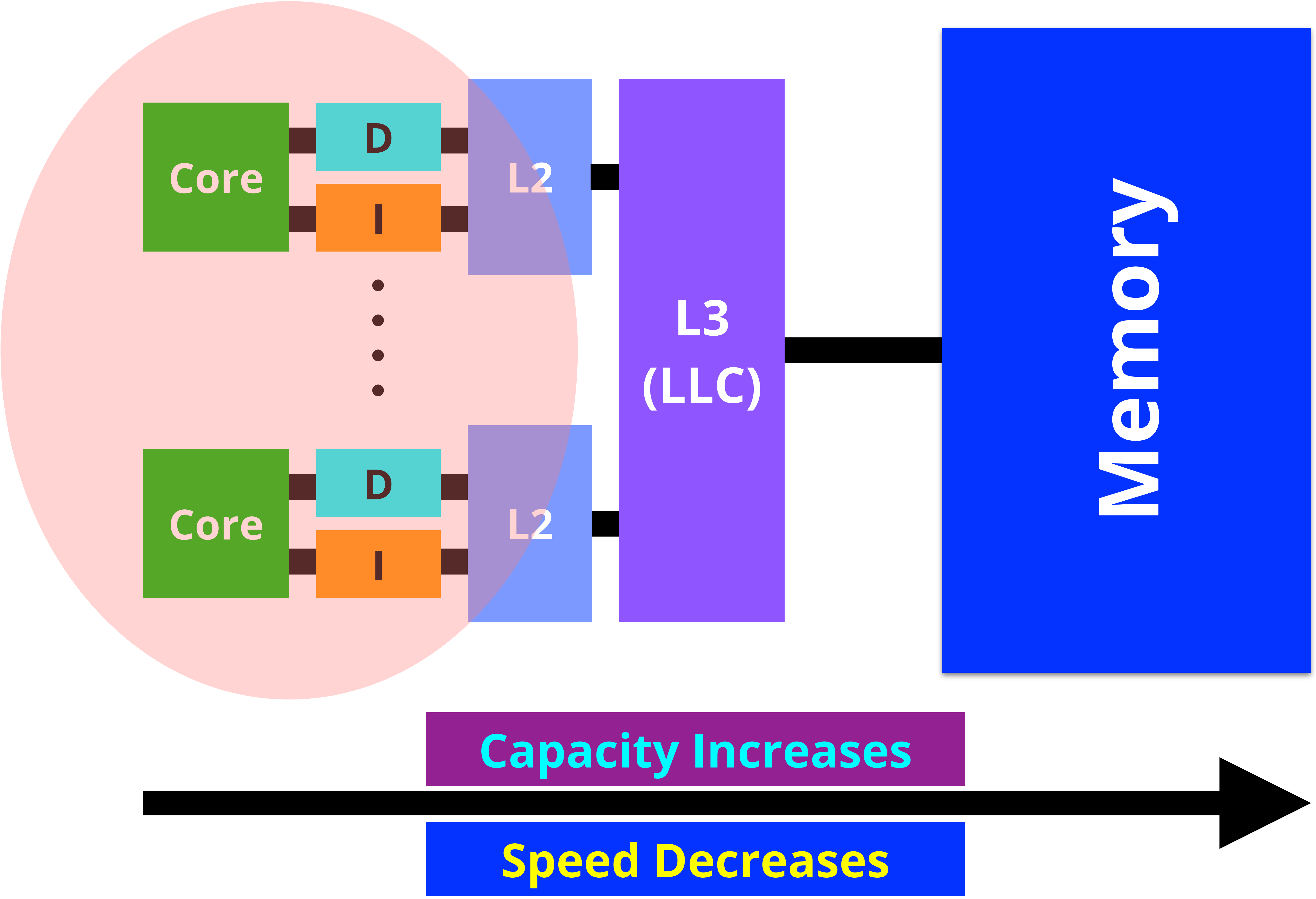
For cost and performance reasons, a modern processor has a hierarchy of caches

Some caches are private to a core, others are shared

Let’s look at a typical example

Copyright (©) 2024 by Ruud van der Pas

A Typical Memory Hierarchy

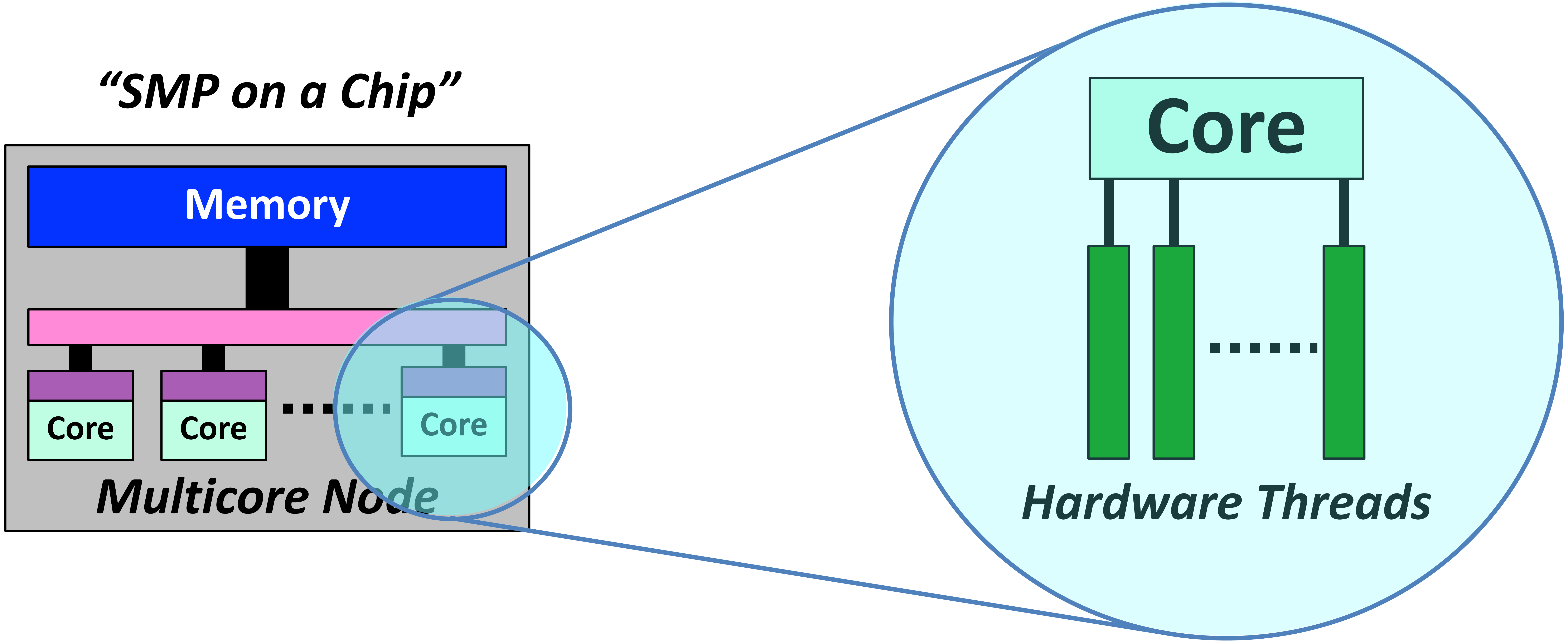


The unit of transfer is a "cache line"

A cache line contains multiple elements

Copyright (©) 2024 by Ruud van der Pas

Multicore and Hardware Threads



Copyright (©) 2024 by Ruud van der Pas

About Cores and Hardware Threads

A core may, or may not, support hardware threads

This is part of the design

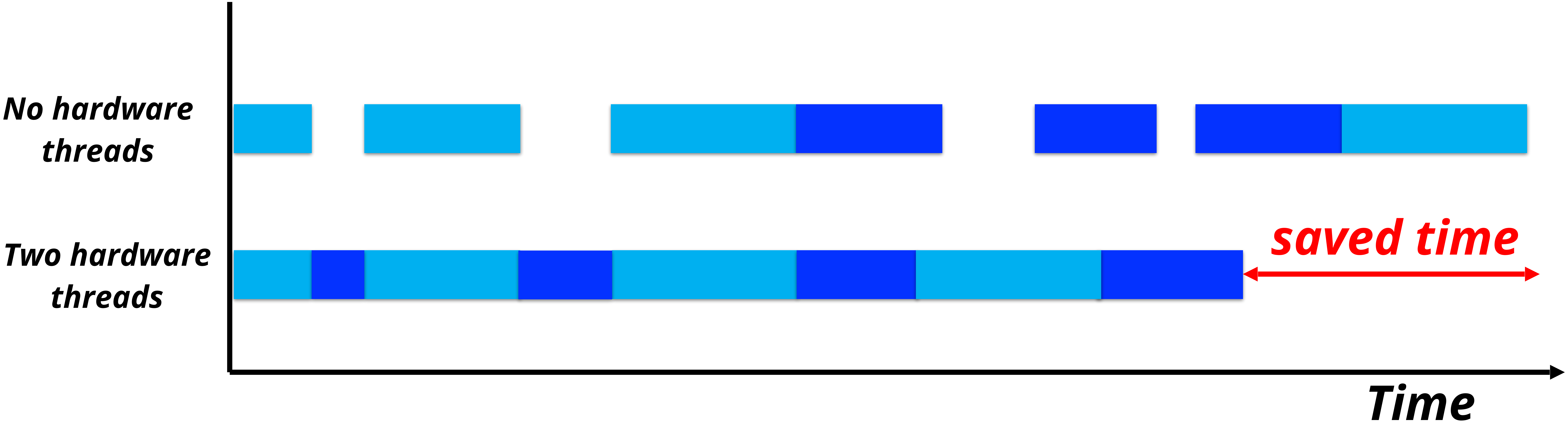
These hardware threads may accelerate the execution of a single application, or improve the throughput of a workload

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

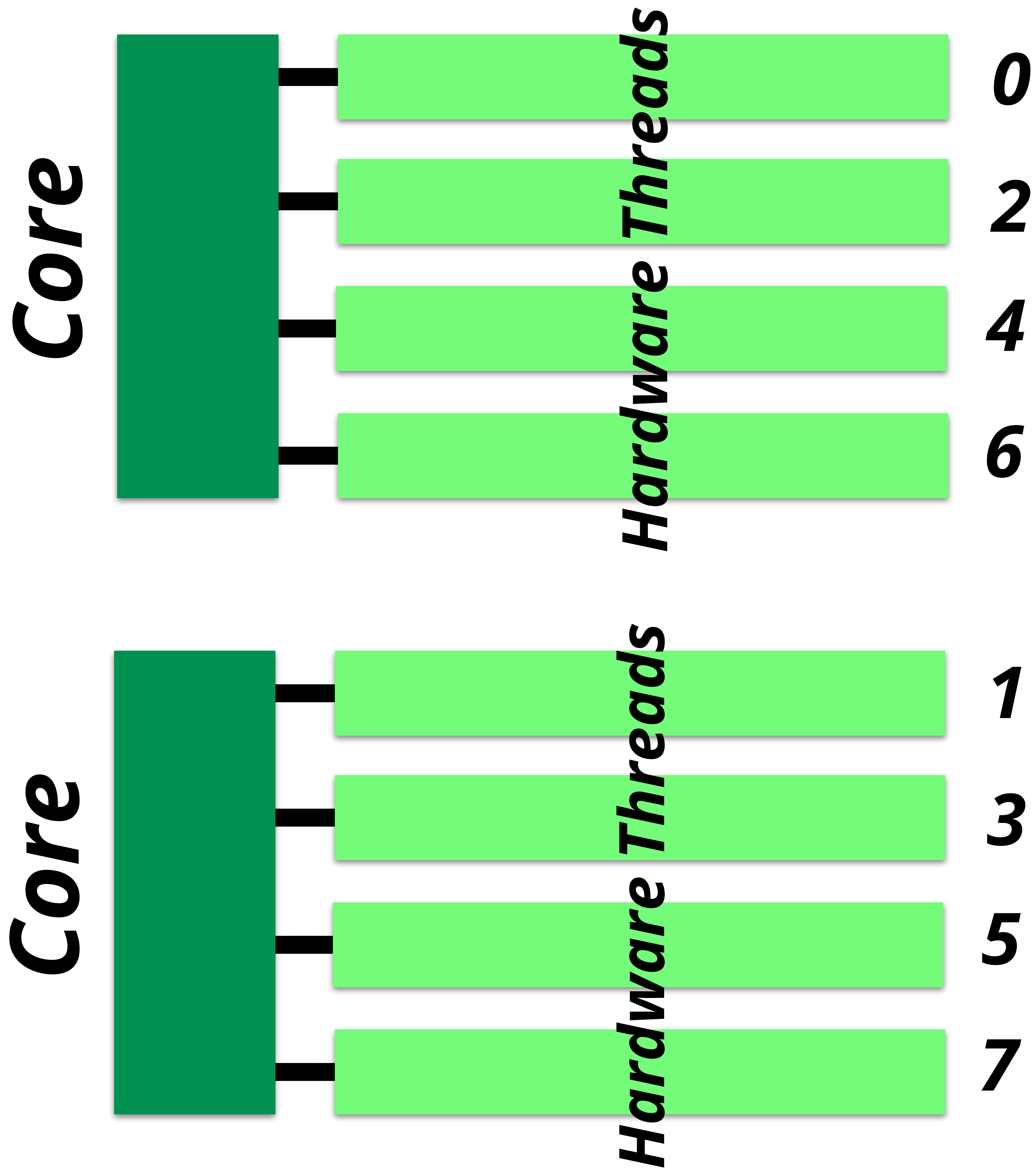
Copyright (©) 2024 by Ruud van der Pas

How Hardware Threads Work



Copyright (©) 2024 by Ruud van der Pas

Hardware Thread IDs



Copyright (©) 2024 by Ruud van der Pas

Part I - What Could Possibly Go Wrong?

Copyright (©) 2024 by Ruud van der Pas

Nothing

of course

or maybe ...

Copyright (©) 2024 by Ruud van der Pas

Where Could Things Go Wrong Then?

Copyright (©) 2024 by Ruud van der Pas

What Do You Actually Mean with “Wrong”?

There are three things that can go wrong

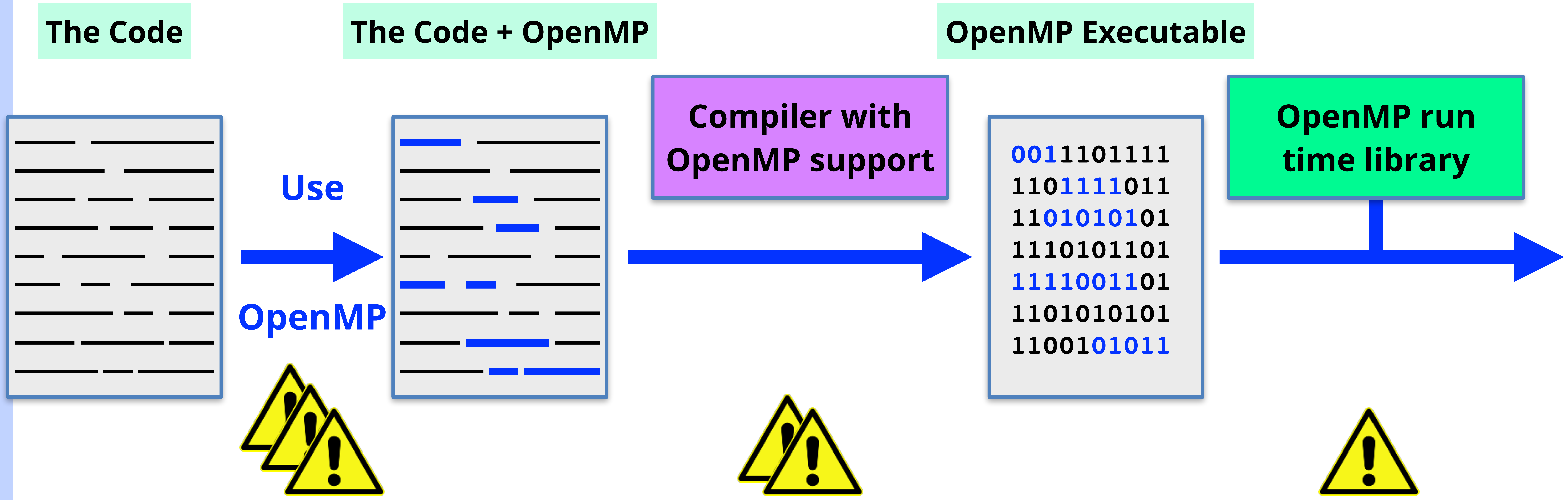
- ***An incorrect answer***
- ***Poor parallel performance***
- ***A wrong answer and terrible performance***

In this talk, we cover the first two categories*

****) The third category is too much for me to handle ;-)***

Copyright (©) 2024 by Ruud van der Pas

The Big OpenMP Picture



The Code

The Code + OpenMP

OpenMP Executable

Use OpenMP

Compiler with OpenMP support

OpenMP run time library

Copyright (©) 2024 by Ruud van der Pas

Wrong Answers - The Top Three

The code has been incorrectly parallelized

The scoping (private, shared, etc.) rules are violated

A data race has been introduced

Copyright (©) 2024 by Ruud van der Pas

Incorrect Parallelization - An Example

This loop has a data dependence

```
prev_val = a[0];
for (int i=1; i<n; i++)
 {
  a[i] = prev_val + b[i];
  prev_val = a[i];
 }
```

→

Still a data dependence …

```
for (int i=1; i<n; i++)
 {
  a[i] = a[i-1] + b[i];
 }
```

Copyright (©) 2024 by Ruud van der Pas

Incorrect Parallelization - An Example

Force the loop to execute in parallel

```
prev_val = a[0];
#pragma omp parallel for
for (int i=1; i<n; i++)
 {
  a[i] = prev_val + b[i];
  prev_val = a[i];
 }
```

```
$ gcc -fopenmp wrong.c
$ export OMP_NUM_THREADS=4
$ ./a.out
Loop length n = 10
Number of threads = 4
Number of errors = 6
a[1] =   3 ref[1] =   3
a[2] =   6 ref[2] =   6
a[3] =  10 ref[3] =  10
a[4] =  34 ref[4] =  15 *
a[5] =  40 ref[5] =  21 *
a[6] =  36 ref[6] =  28 *
a[7] =  44 ref[7] =  36 *
a[8] =  19 ref[8] =  45 *
a[9] =  29 ref[9] =  55 *
```

Copyright (©) 2024 by Ruud van der Pas

Incorrect Parallelization - Wrong Results (of course)

Force the loop to execute in parallel

```
#pragma omp parallel for
for (int i=1; i<n; i++)
 {
  a[i] = a[i-1] + b[i];
 }
```

```
$ gcc -fopenmp wrong.c
$ export OMP_NUM_THREADS=4
$ ./a.out
Loop length n = 10
Number of threads = 4
Number of errors = 4
a[1] =   3 ref[1] =   3
a[2] =   6 ref[2] =   6
a[3] =  10 ref[3] =  10
a[4] =  15 ref[4] =  15
a[5] =  21 ref[5] =  21
a[6] =  47 ref[6] =  28 *
a[7] =  55 ref[7] =  36 *
a[8] =  64 ref[8] =  45 *
a[9] =  74 ref[9] =  55 *
```

Copyright (©) 2024 by Ruud van der Pas

Incorrect Parallelization - Morale

Parallelize code that is not parallel => maybe wrong results

Yes, “maybe”. In the worse case, the results are sometimes ok

There is a simple trick, but it works only one way

If it is a loop, run the sequential version backwards

If the results are wrong, you know it is not parallel as written

If the results are correct, you still don’t know …

Copyright (©) 2024 by Ruud van der Pas

Incorrect Parallelization - Some Tips

Use a profiling tool to see if this code part actually matters

If this is the case, try to find a parallel version

But, be aware it is still efficient on a single thread

Isolate the sequential part and parallelize the remainder

In doing so, try to avoid excessive extra cache/memory traffic

Copyright (©) 2024 by Ruud van der Pas

Wrong Answers - Violation of the Scoping Rules

The previous example also included a wrong scoping case

Variable prev_val was implicitly scoped as “shared”

This is one of the common pitfalls, but not the only one

The most common mistake is about private variables

Recall that they are undefined outside of the parallel region

```
prev_val = a[0];
#pragma omp parallel for
for (int i=1; i<n; i++)
 {
  a[i] = prev_val + b[i];
  prev_val = a[i];
 }
```

Copyright (©) 2024 by Ruud van der Pas

Incorrect Scoping - Another Example

```
int my_var = 10;
#pragma omp parallel for private(my_var)
for (int i=0; i<n; i++)
 {
  a[i] = my_var + b[i];
 }
```

Variable my_var is undefined

Even if this might work today, there is no guarantee for tomorrow

Copyright (©) 2024 by Ruud van der Pas

Incorrect Scoping - The Solution

```
int my_var = 10;
#pragma omp parallel for firstprivate(my_var)
for (int i=0; i<n; i++)
 {
  a[i] = my_var + b[i];
 }
```

Variable my_var *is implied to be private*

Each thread has a local copy with an initial value of 10

Copyright (©) 2024 by Ruud van der Pas

Incorrect Scoping - Morale

Declare variables local to a code block where possible

They are automatically privatized

Specify the scope of the remaining variables yourself

This is not as hard as it may seem

Extremely rewarding when it comes to avoiding bugs

Copyright (©) 2024 by Ruud van der Pas

Wrong Answers - Data Races

A data race occurs if all the following conditions are met

- ***Multiple threads access the same memory location concurrently***
- ***At least one of the accesses modifies the contents of this location***
- ***There is no control to guarantee exclusive access to this location***

A data race may lead to silent data corruption

The wrong results are also non-deterministic

Yes, the results may vary, even across identical runs

Copyright (©) 2024 by Ruud van der Pas

Incorrect Code - A Data Race Example

```
int my_shared_var = 0;
#pragma omp parallel for shared(my_shared_var)
for (int i=0; i<n; i++)
 {
  my_shared_var += a[i];
 }
```

WRONG!

The above code meets all 3 conditions

At any moment, multiple threads may read and write my_shared_var

Copyright (©) 2024 by Ruud van der Pas

Incorrect Code - Fixing the Data Race Example

```
int my_shared_var = 0;
#pragma omp parallel for reduction(+:my_shared_var)
for (int i=0; i<n; i++)
 {
  my_shared_var += a[i];
 }
```

As simple as it looks, the reduction clause generates non-trivial code that avoids the data race

Copyright (©) 2024 by Ruud van der Pas

Data Races - Morale

Data races are very nasty

Luckily, OpenMP provides high level constructs to avoid them

In less common cases, use alternatives that avoid data races:

- ***Atomic operations***
- ***Critical regions***
- ***Barriers***
- ***Locks***

Please us these!

These help to make it easier to avoid data races

Copyright (©) 2024 by Ruud van der Pas

Poor Performance - The Top Three

Too much parallel overhead

- ***Consolidate as much work as possible in a single parallel region***

Load balancing

- ***Consider the schedule clause and tasking***

Non-Uniform Memory Acces (NUMA)

- ***Experiment with the affinity related environment variables***

Copyright (©) 2024 by Ruud van der Pas

Summary Part I

We covered some major mistakes made

Unfortunately, these, or others could happen to you too

What helps, is to regularly check for correctness

The performance issues mentioned are the tip of the iceberg

But, it is a big tip :-)

Make sure to use a profiling tool to guide you with the tuning

Copyright (©) 2024 by Ruud van der Pas

Part II - The Joy Of Computer Memory

Copyright (©) 2024 by Ruud van der Pas

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, 2024

Copyright (©) 2024 by Ruud van der Pas

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

Copyright (©) 2024 by Ruud van der Pas

What is False Sharing?

A corner case, but it may affect you

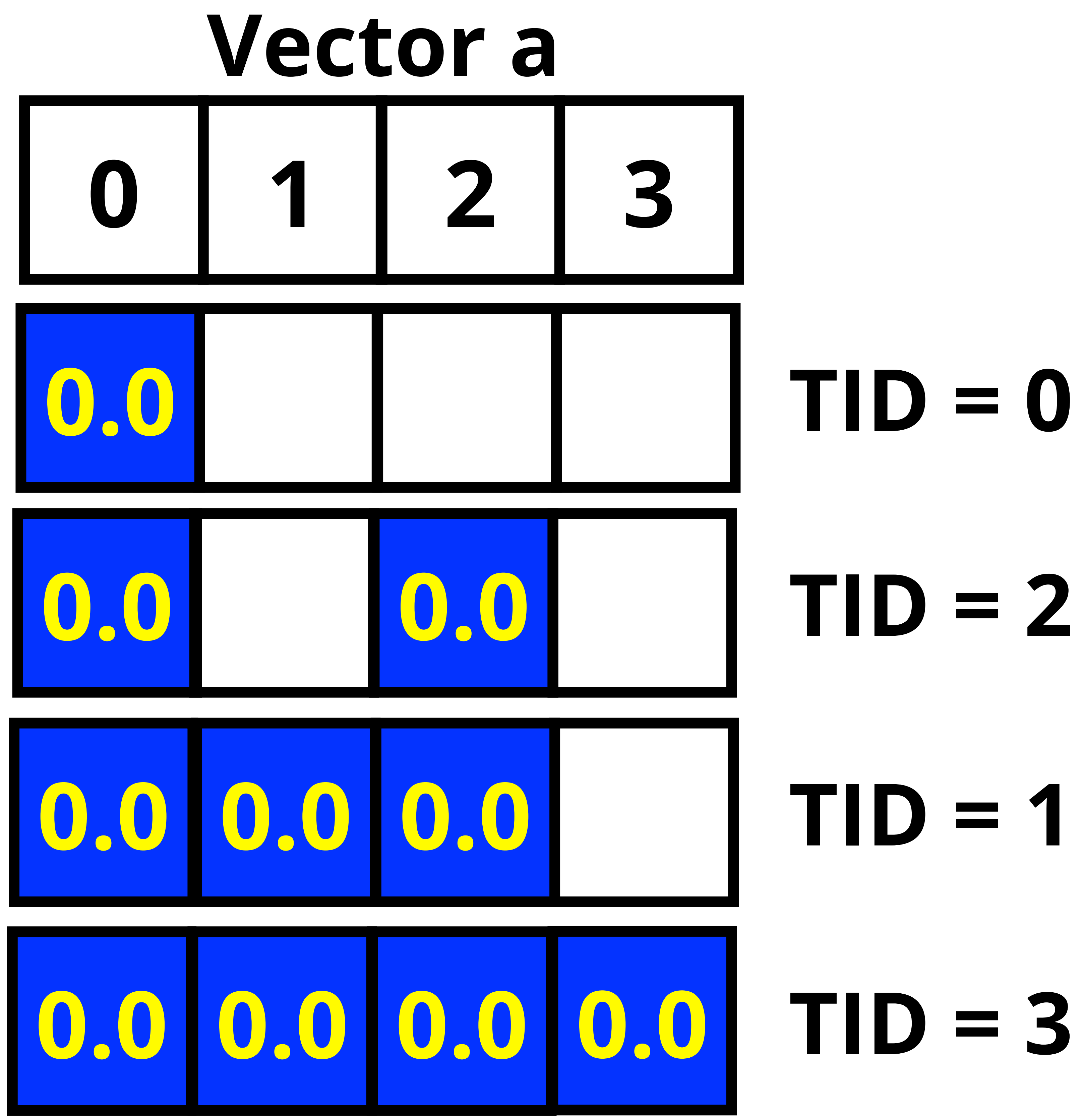
Happens when multiple threads modify the same cache line at the same time

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

Copyright (©) 2024 by Ruud van der Pas

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

A data race induces false sharing
(so the program will run much slower)

Copyright (©) 2024 by Ruud van der Pas

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)

Copyright (©) 2024 by Ruud van der Pas

NUMA in Contemporary Systems

Copyright (©) 2024 by Ruud van der Pas

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

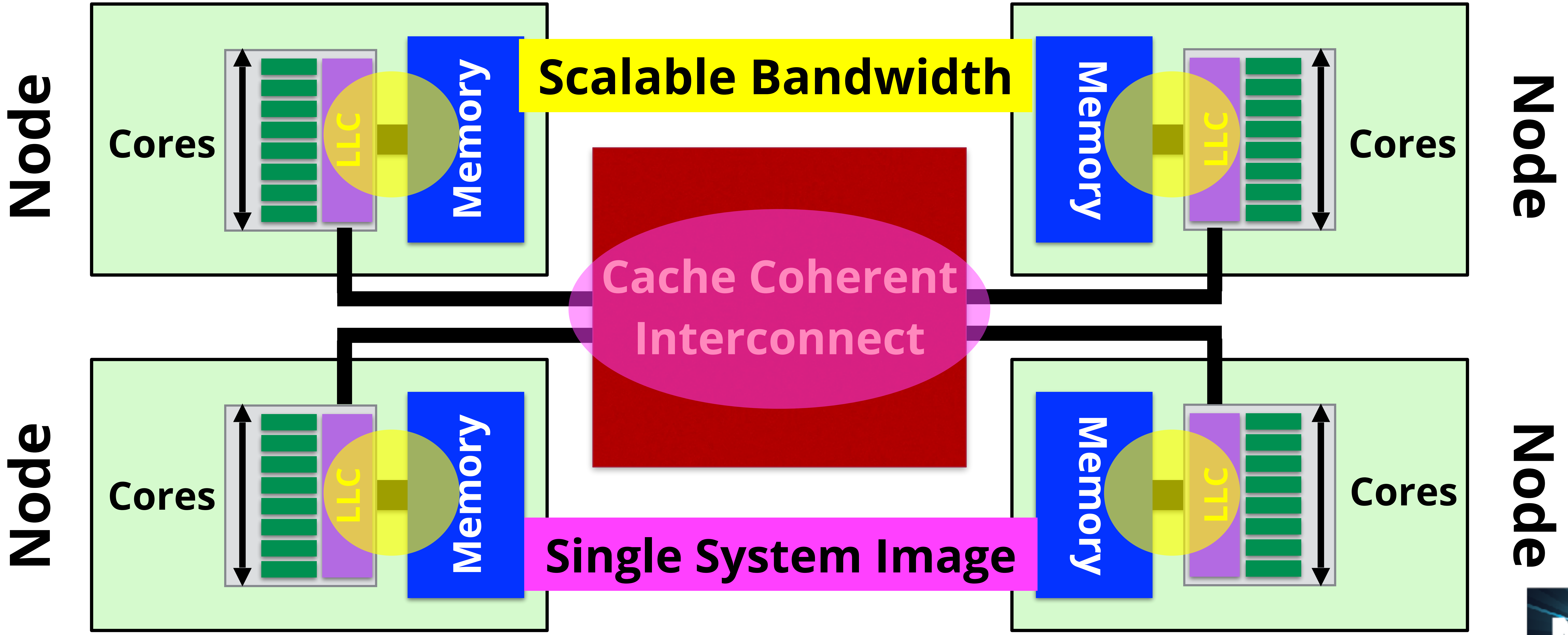
The tricky thing is that "things just work"

But do you know how efficiently your code performs?

Copyright (©) 2024 by Ruud van der Pas

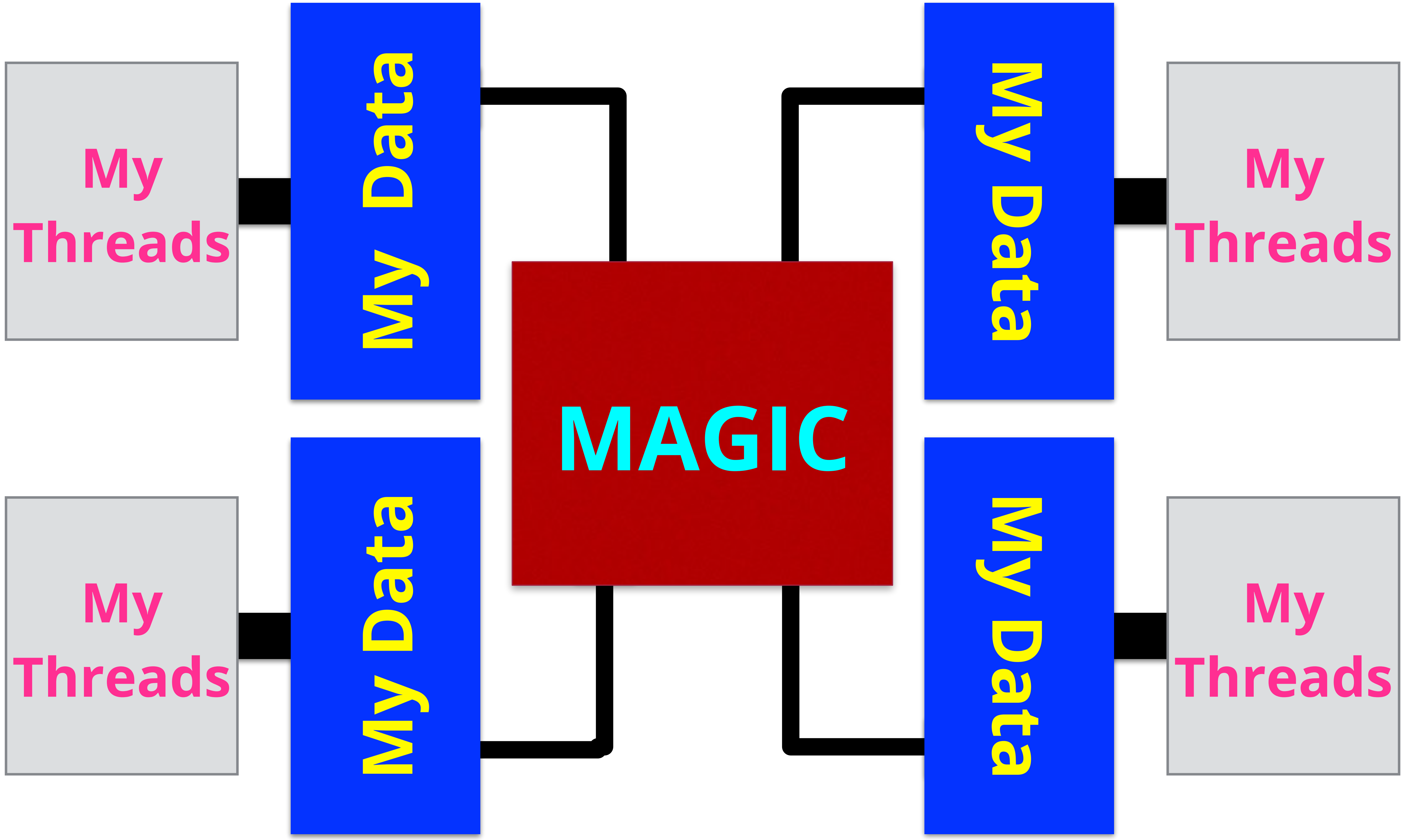
NUMA - The System Most of Us Use Today

A Generic, but very Common and Contemporary NUMA System



Copyright (©) 2024 by Ruud van der Pas

The Developer's View



Copyright (©) 2024 by Ruud van der Pas

The NUMA View

Memory is physically distributed, but logically shared

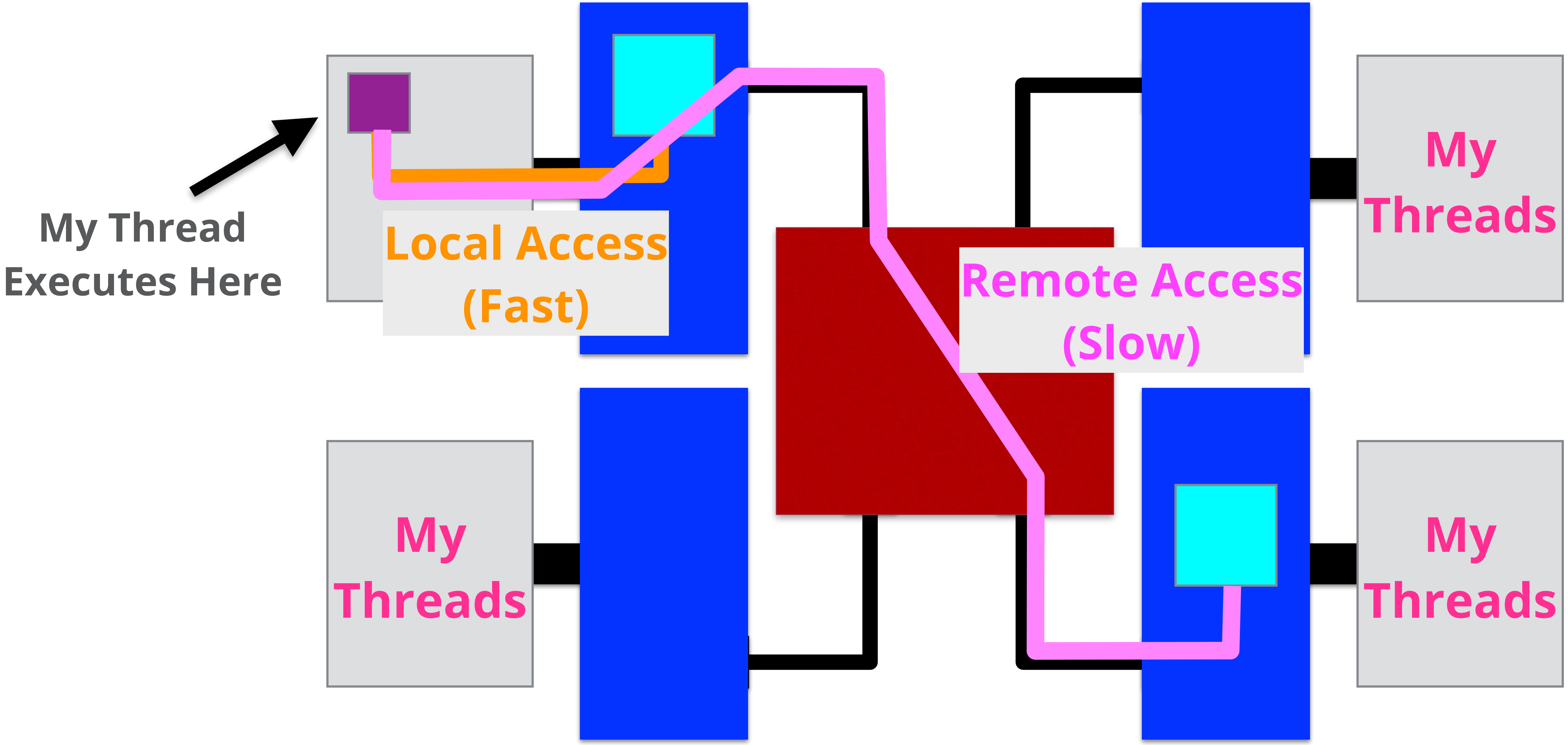
Shared data is accessible to all threads

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

Unless you care about performance …

Copyright (©) 2024 by Ruud van der Pas

Local Versus Remote Access Times



Copyright (©) 2024 by Ruud van der Pas

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)

Copyright (©) 2024 by Ruud van der Pas

About NUMA and Data Placement

Copyright (©) 2024 by Ruud van der Pas

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

Copyright (©) 2024 by Ruud van der Pas

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

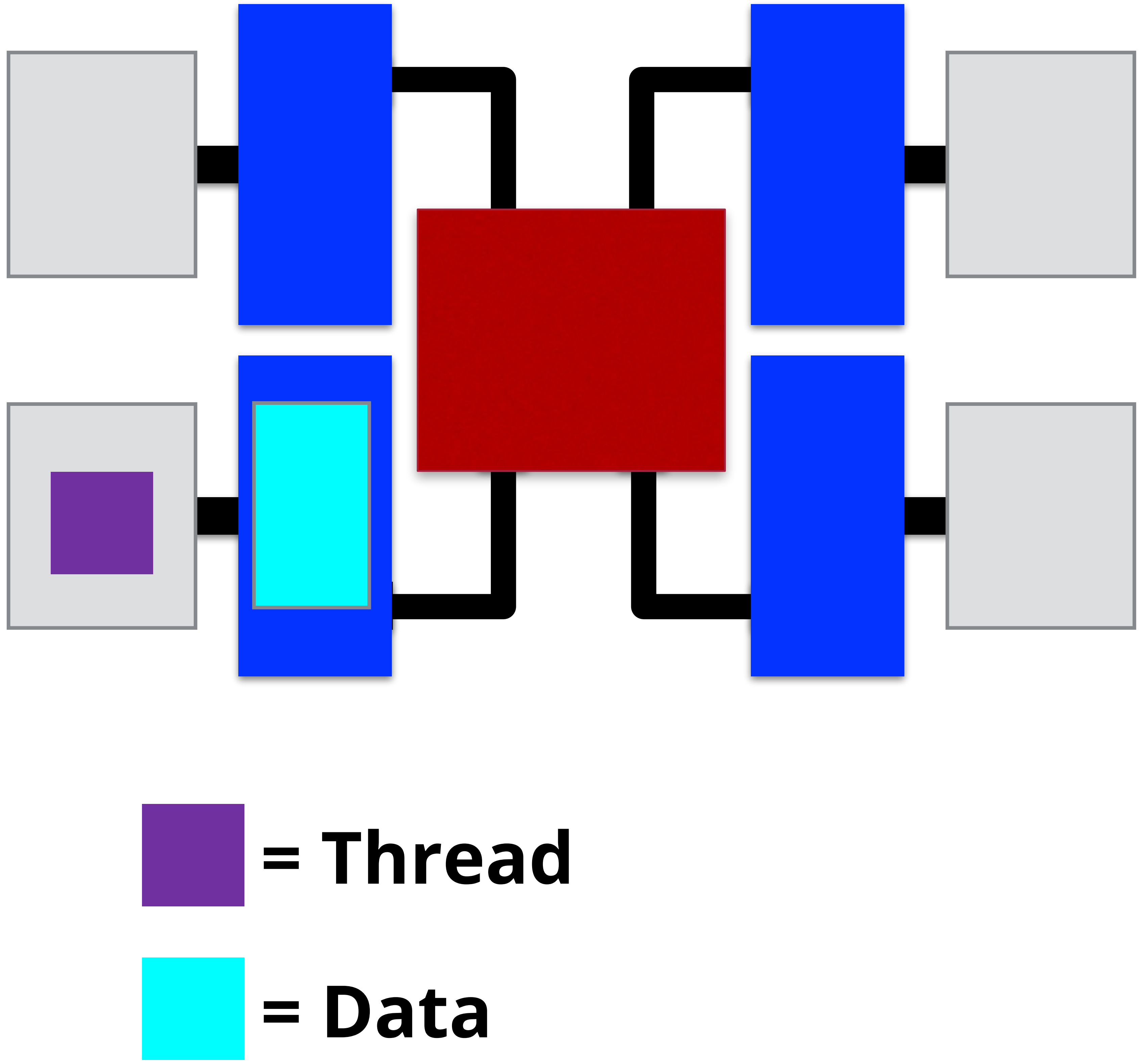
Copyright (©) 2024 by Ruud van der Pas

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



Note: The allocation is on a virtual memory page basis

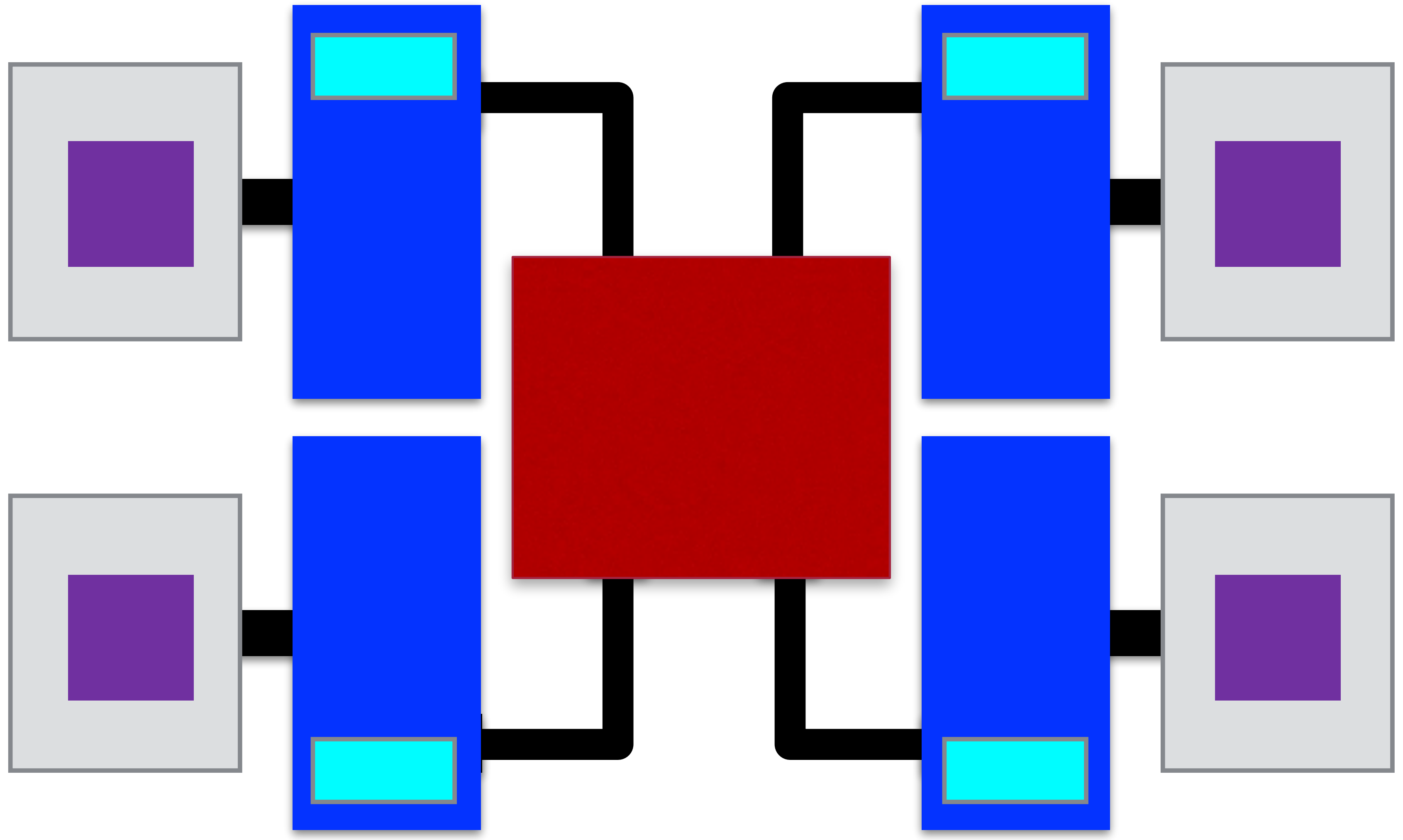
Leverage the First Touch Placement Policy

```
#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

Copyright (©) 2024 by Ruud van der Pas

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

Copyright (©) 2024 by Ruud van der Pas

OpenMP Support for NUMA Systems

Copyright (©) 2024 by Ruud van der Pas

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

Copyright (©) 2024 by Ruud van der Pas

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

Copyright (©) 2024 by Ruud van der Pas

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

Copyright (©) 2024 by Ruud van der Pas

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***

Copyright (©) 2024 by Ruud van der Pas

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}

Copyright (©) 2024 by Ruud van der Pas

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
```

Copyright (©) 2024 by Ruud van der Pas

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

Copyright (©) 2024 by Ruud van der Pas

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
```

Copyright (©) 2024 by Ruud van der Pas

Remember This Example?

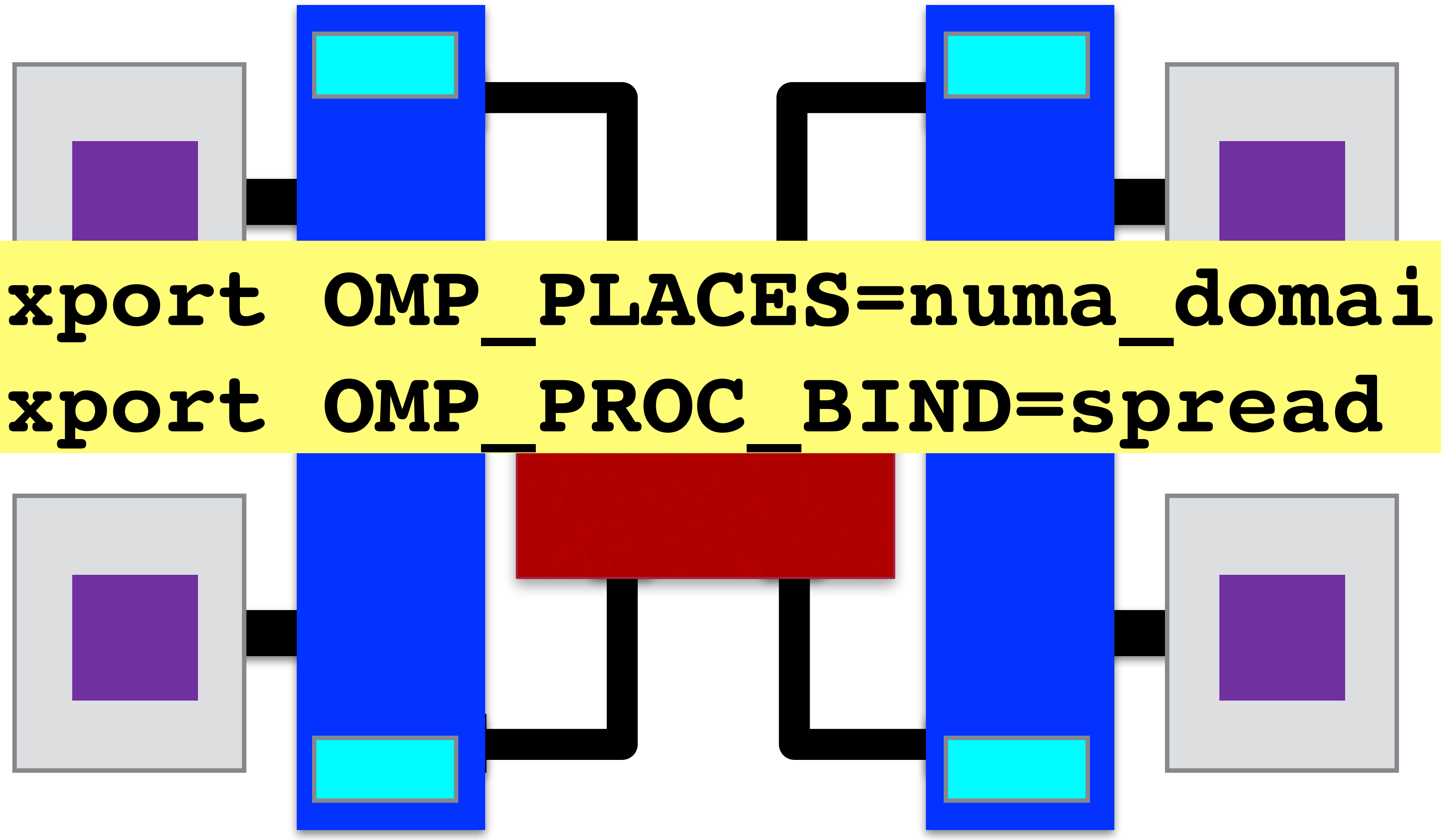
```
#pragma omp parallel for schedule(static)
for (int64_t i=0; i<n; i++)
     a[i] = 0;
```

Four threads execute this loop

Wishful Thinking

Data placement depends on where threads execute

Use Affinity Controls ✓



```
$ 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 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!

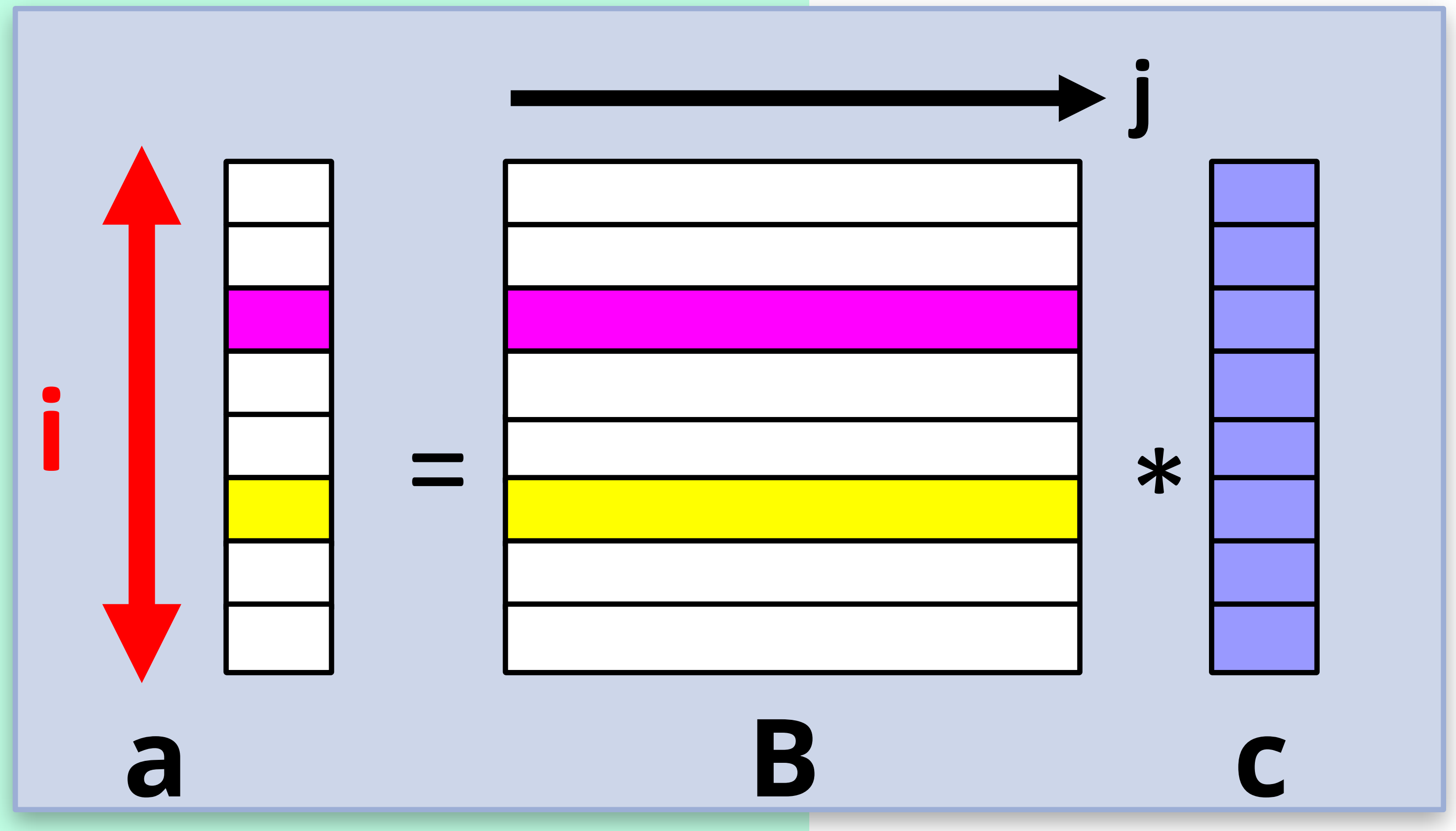
Copyright (©) 2024 by Ruud van der Pas

A Performance Tuning Example

Copyright (©) 2024 by Ruud van der Pas

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;
}
```



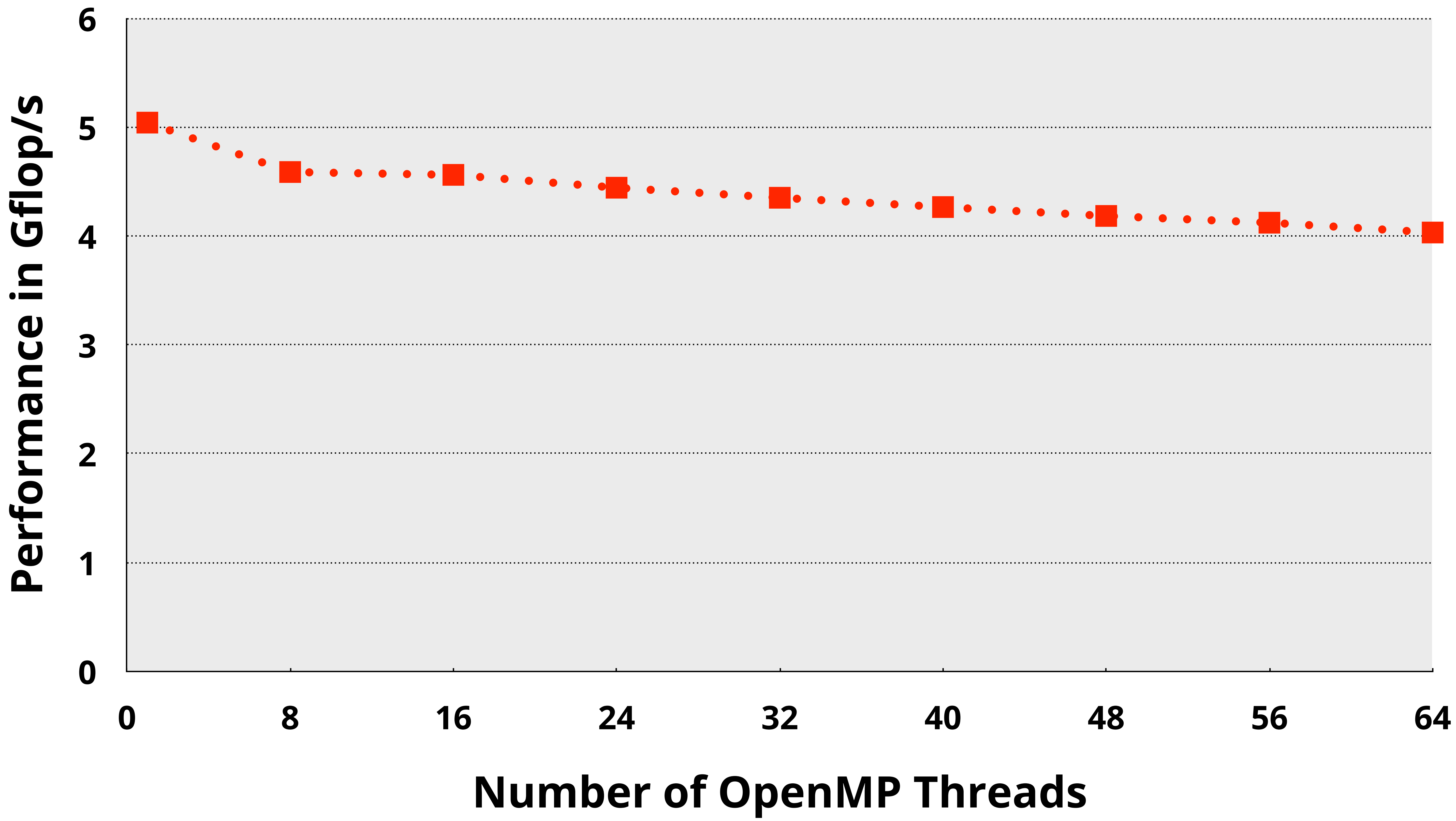
An embarrasingly parallel algorithm!

(on paper)

Copyright (©) 2024 by Ruud van der Pas

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*

Copyright (©) 2024 by Ruud van der Pas

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
```

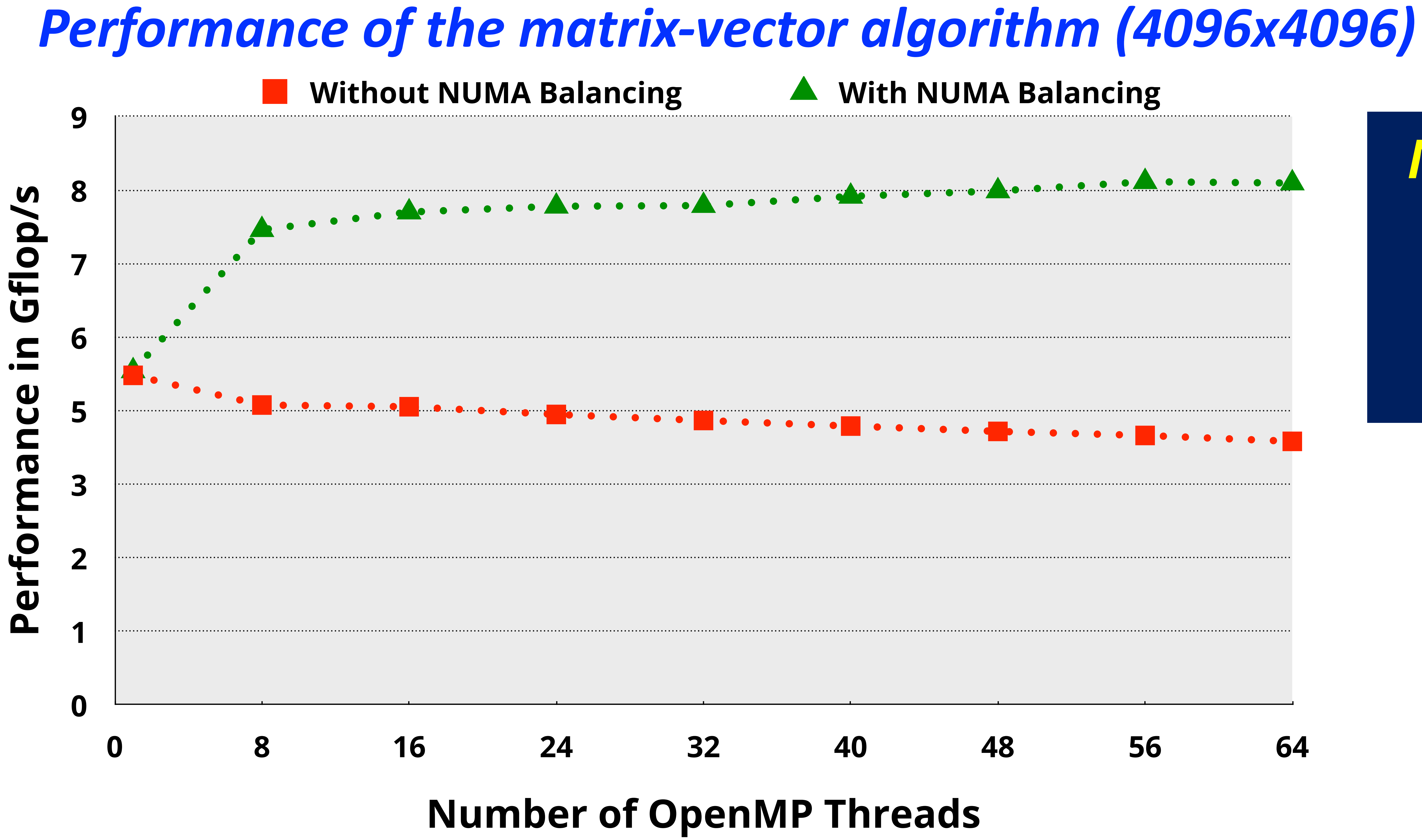
enable

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

disable

Copyright (©) 2024 by Ruud van der Pas

The Performance Using 64 Threads*



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

Copyright (©) 2024 by Ruud van der Pas

Let’s Check The System We Are Using!

Copyright (©) 2024 by Ruud van der Pas

The NUMA Information for 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
......
$
```

```
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
```

8 cores/node

8 NUMA Nodes

2 columns => 2 hardware threads/core

Copyright (©) 2024 by Ruud van der Pas

*The NUMA Structure of the System**

Consists of 8 NUMA nodes according to “lscpu”

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

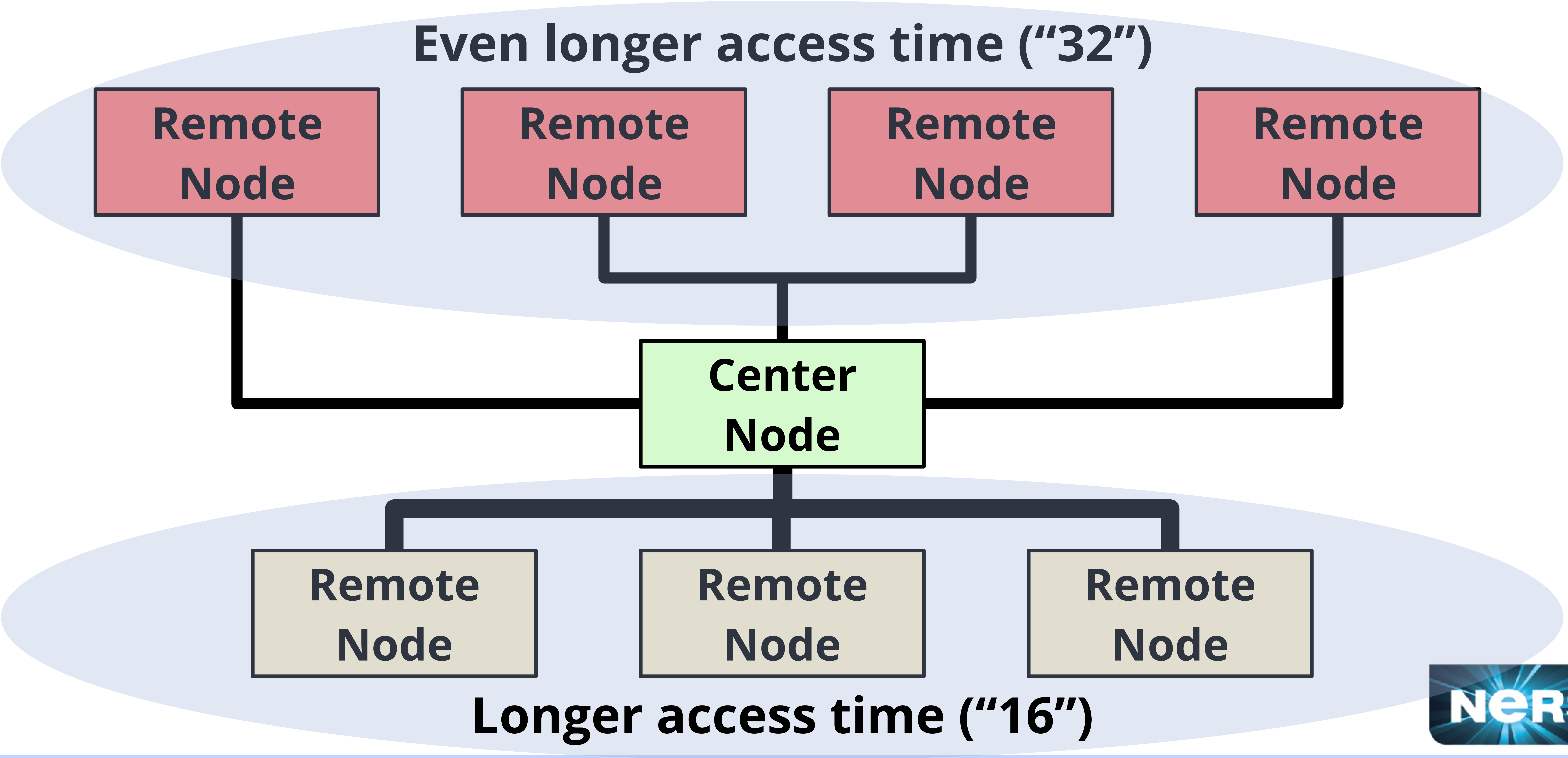
Each NUMA node has 8 cores with 2 hardware threads each

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 :-))*

Copyright (©) 2024 by Ruud van der Pas

The Abstract System Topology (numactl -H)



Even longer access time (“32”)

Remote Node

Remote Node

Remote Node

Remote Node

Center Node

Remote Node

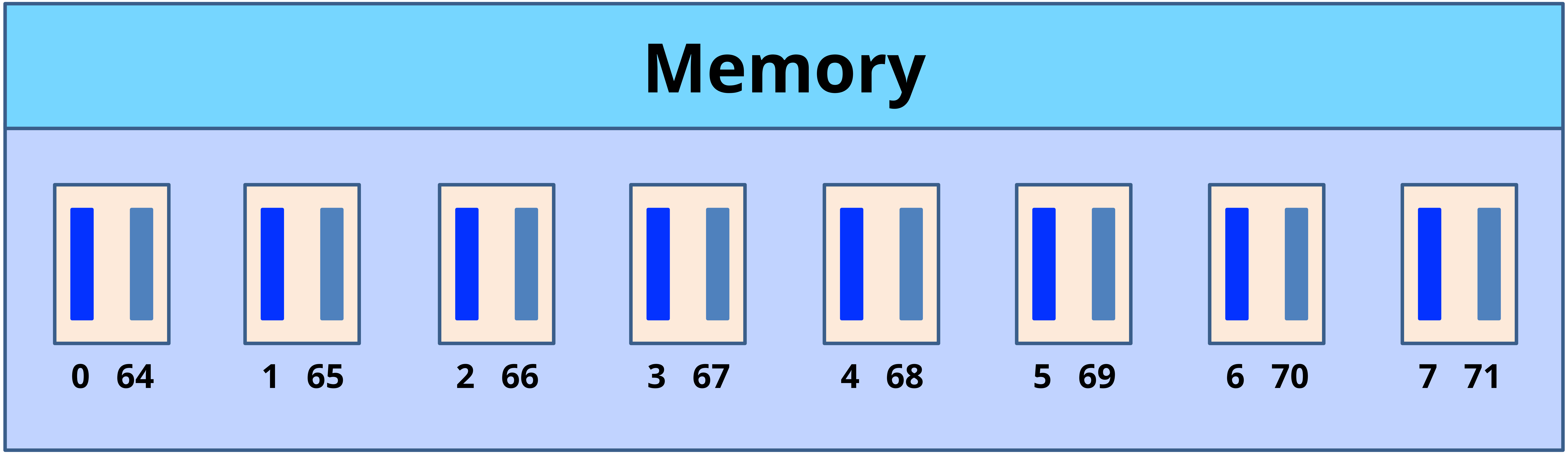
Remote Node

Remote Node

Longer access time (“16”)

Copyright (©) 2024 by Ruud van der Pas

Example - NUMA Node 0 (lscpu output)



Memory

0 64

1 65

2 66

3 67

4 68

5 69

6 70

7 71

8 cores

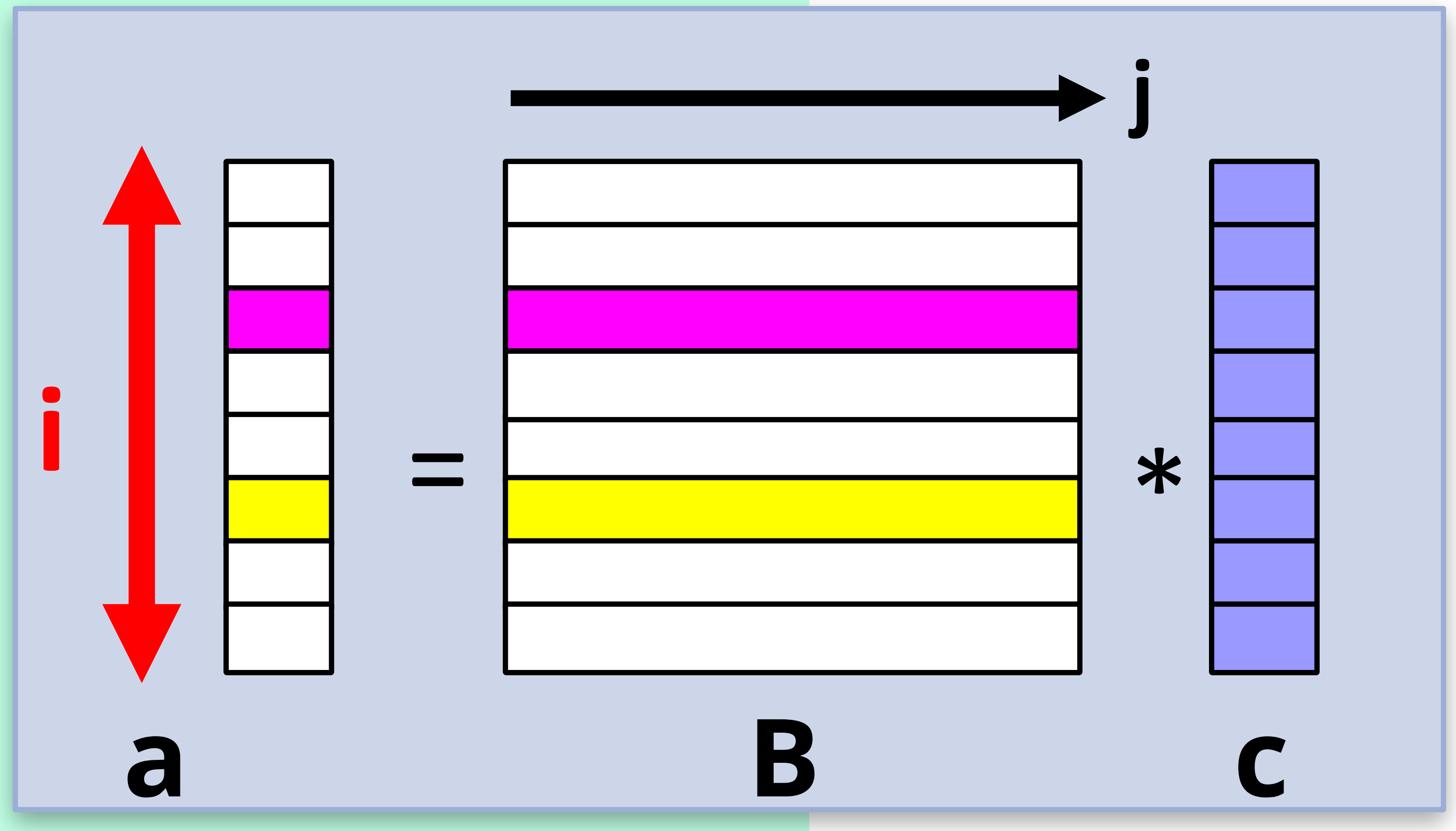
16 hardware threads

All cores and hardware threads share the memory in the node

Copyright (©) 2024 by Ruud van der Pas

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;
}
```



Copyright (©) 2024 by Ruud van der Pas

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

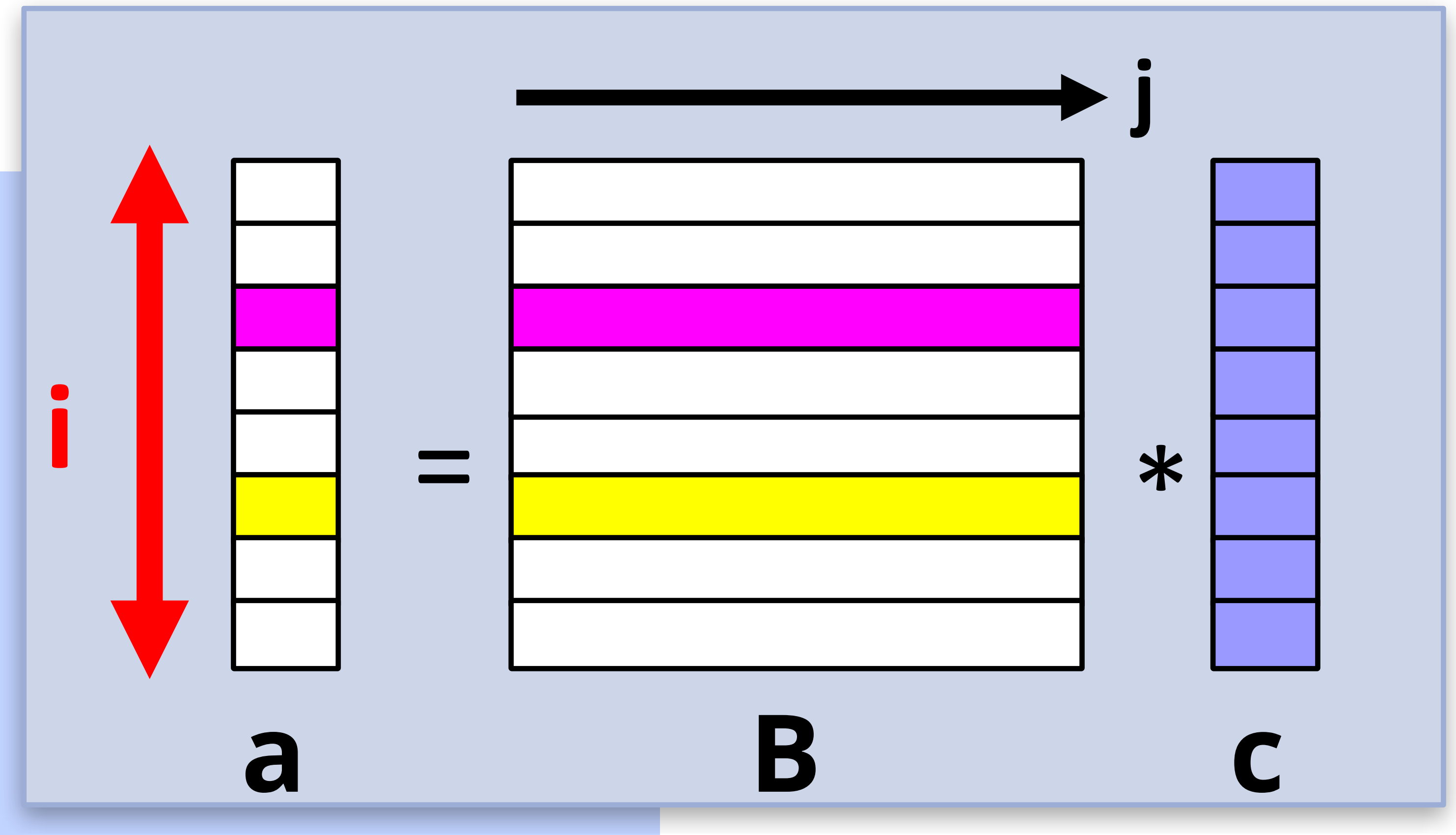
Copyright (©) 2024 by Ruud van der Pas

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;
}
```

Copyright (©) 2024 by Ruud van der Pas

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
```

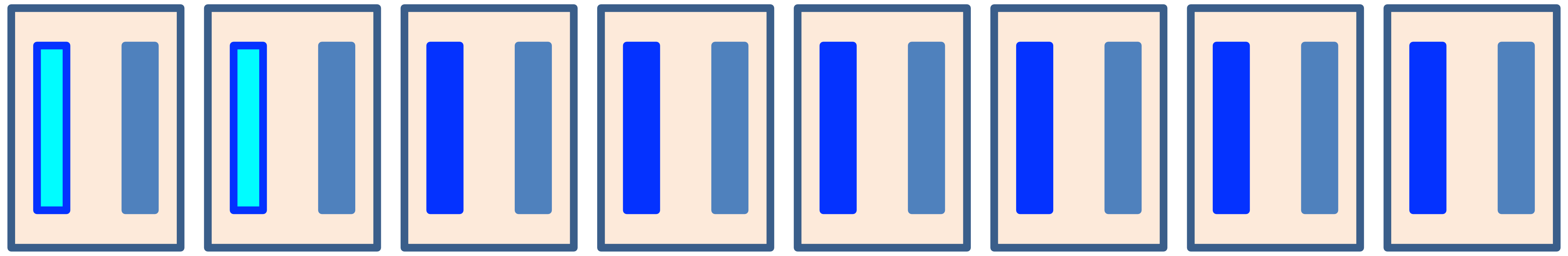
Copyright (©) 2024 by Ruud van der Pas

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



Copyright (©) 2024 by Ruud van der Pas

Example - The Target Hardware Thread Numbers

Copyright (©) 2024 by Ruud van der Pas

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}

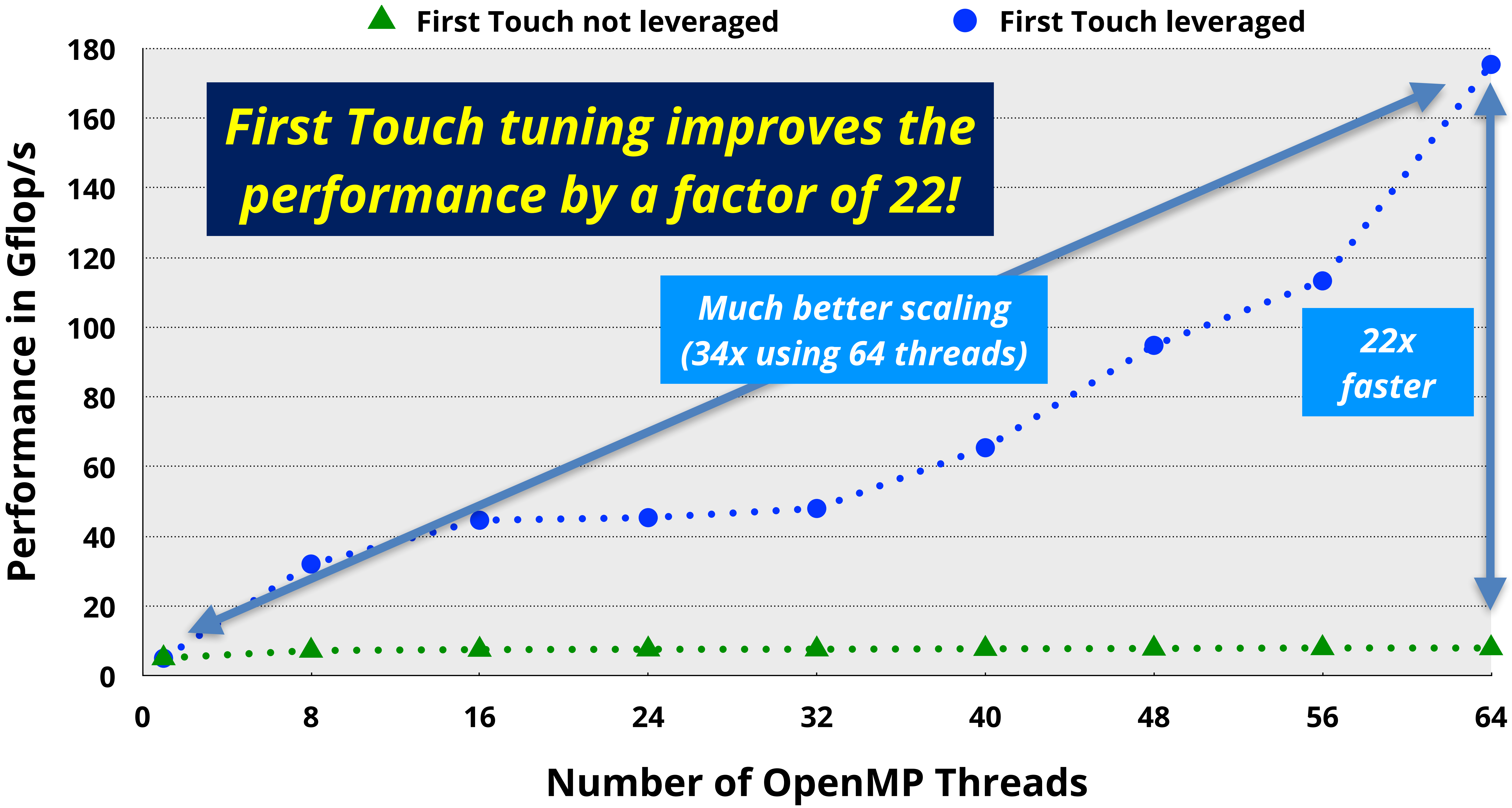
```
$ export OMP_PLACES={0}:8:8,{1}:8:8

$ 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!

Copyright (©) 2024 by Ruud van der Pas

The Performance for a 4096x4096 matrix



First Touch tuning improves the performance by a factor of 22!

Much better scaling (34x using 64 threads)

22x faster

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

Copyright (©) 2024 by Ruud van der Pas

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

Copyright (©) 2024 by Ruud van der Pas

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 may cases, a performance "mystery" is explained by NUMA effects, False Sharing, or both

Copyright (©) 2024 by Ruud van der Pas

Thank You And ... Stay Tuned!

Bad OpenMP Does Not Scale

Ruud van der Pas