Staying Close to Home with NUMA

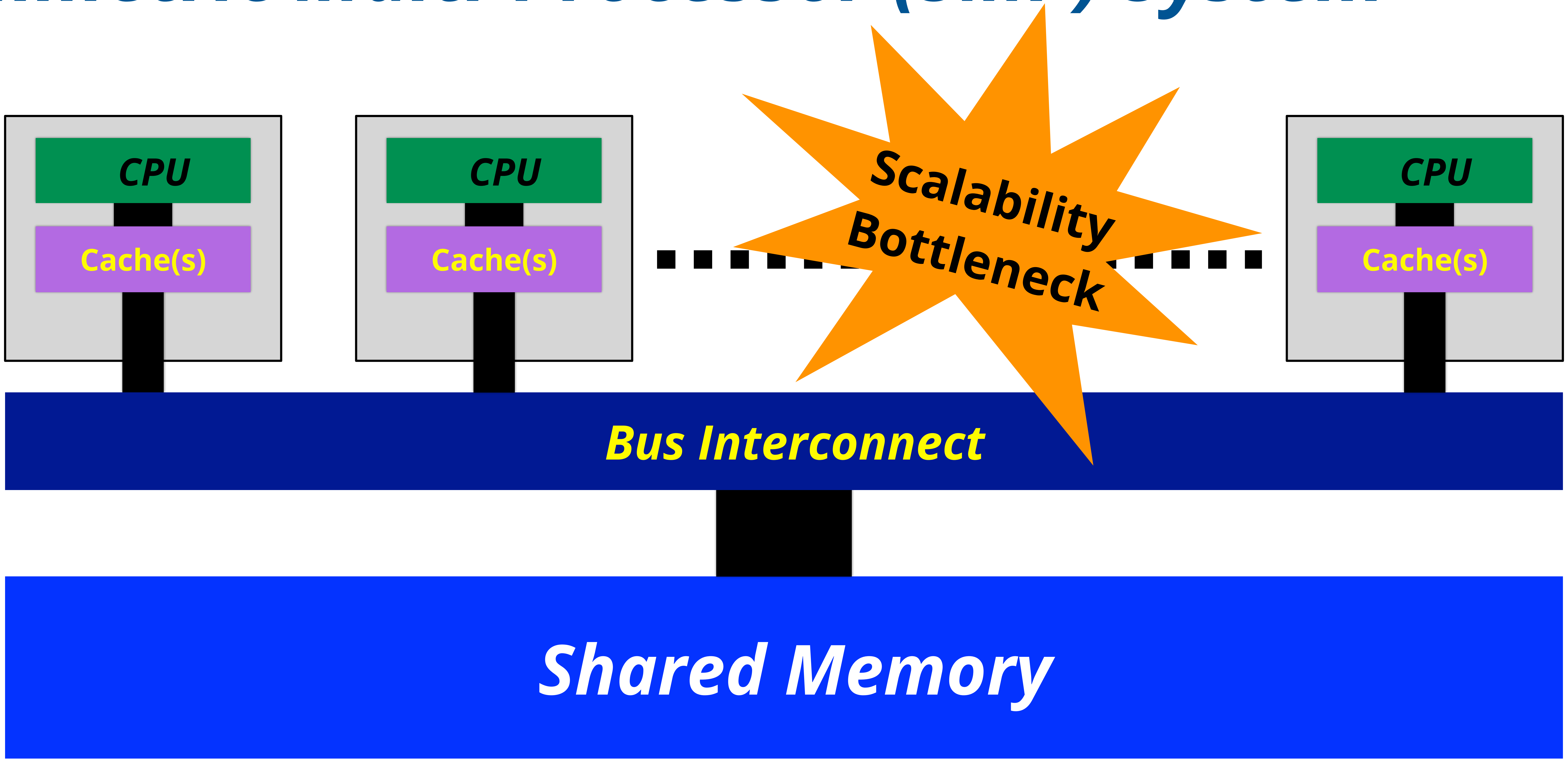
Ruud van der Pas
Senior Principal Software Engineer

Oracle Linux and Virtualization Engineering

Multicore World 2025
Christchurch, New Zealand, February 17-21, 2025

What is NUMA?

A Symmetric Multi-Processor (SMP) System



What is NUMA?

Non-Uniform Memory Access (NUMA)

What is NUMA - Who is Right?

"A great way to provide scalable memory performance"

- The computer architect

"No idea what you're talking about"

- The developer

"A curse and a pain to optimize for"

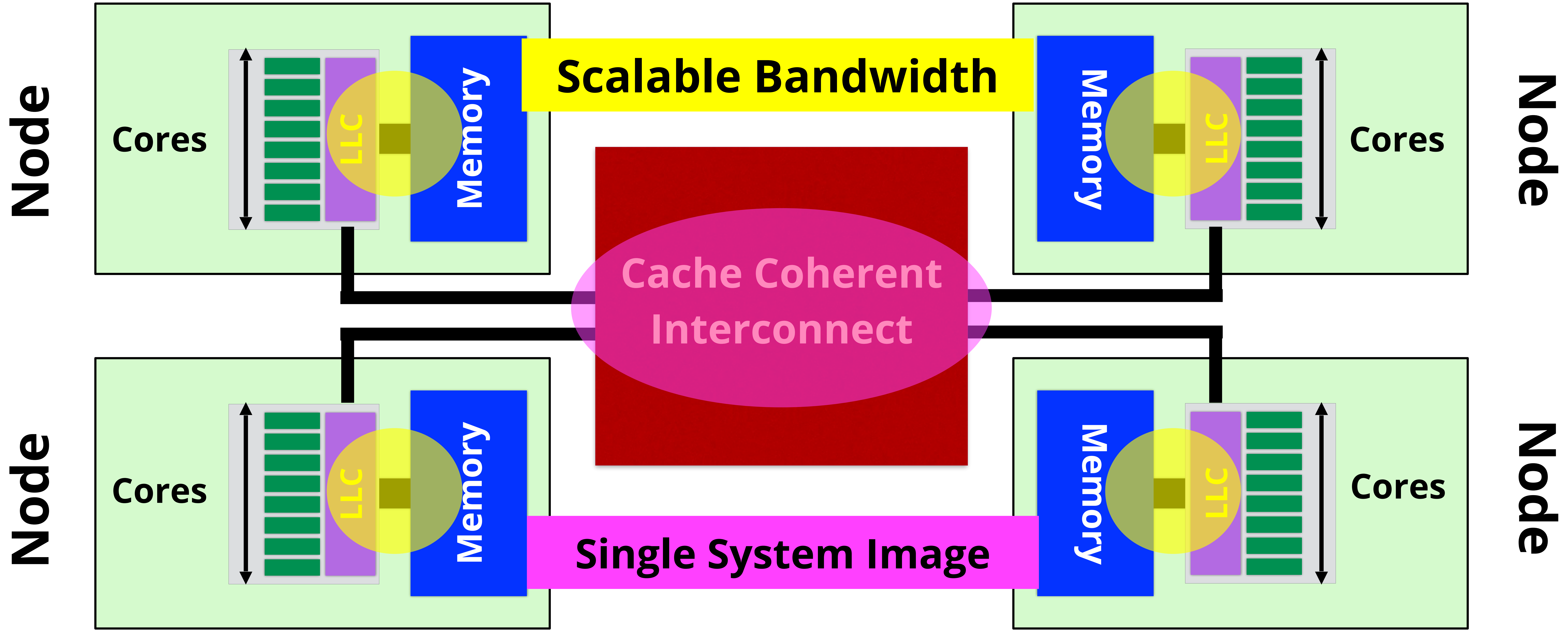
- The performance analyst

What is NUMA - Who is Right?

- "A great way to provide scalable memory performance"
  - The computer architect
- "No idea what you're talking about"
  - The developer
- "A curse and a pain to optimize for"
  - The performance analyst

NUMA - The System Most of Us Use Today

A Generic, but very Common and Contemporary NUMA System



The NUMA View

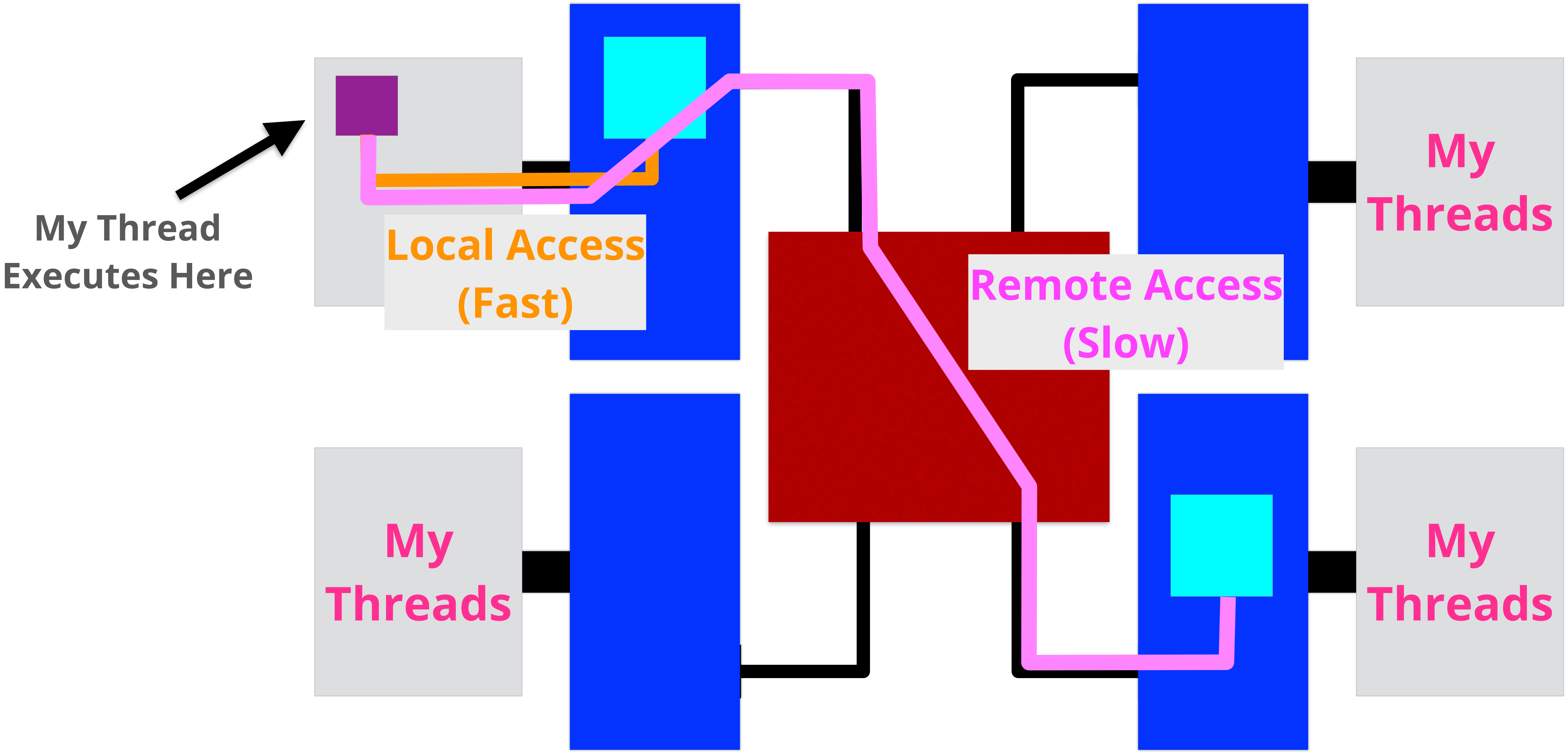
Memory is physically distributed, but logically shared

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

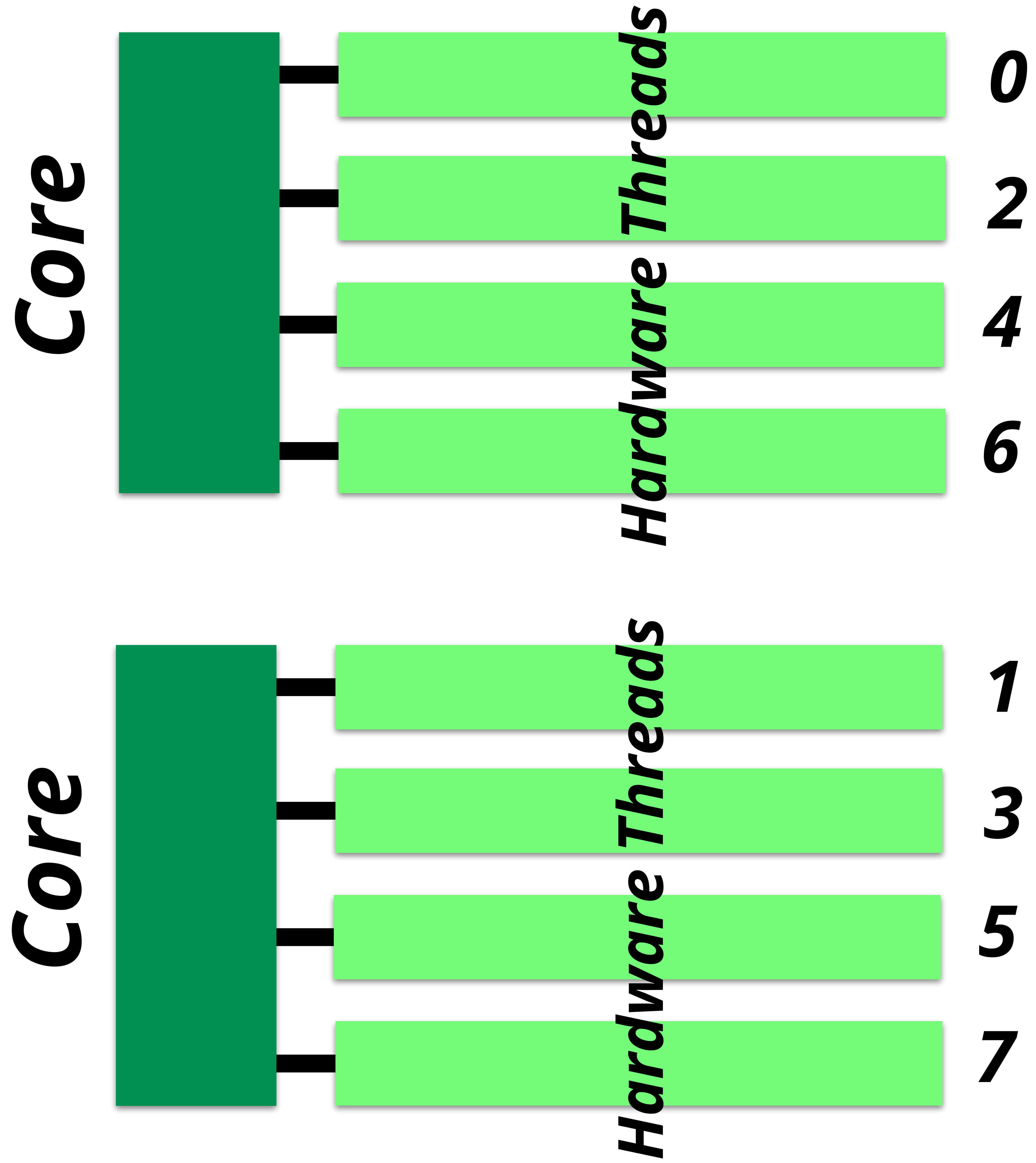


The Goal of Tuning for NUMA

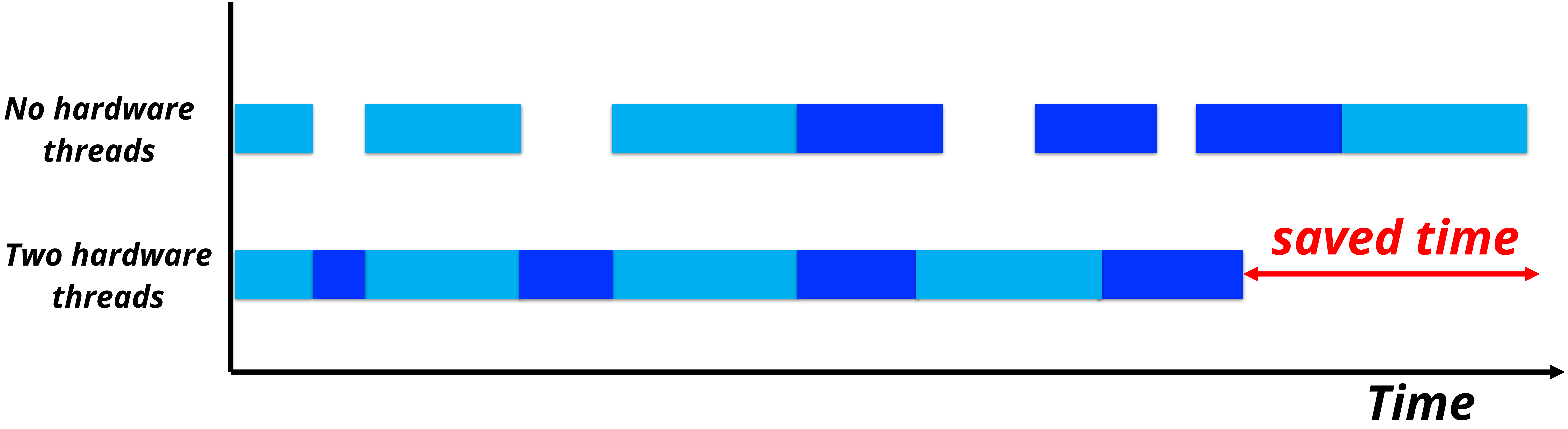
Keep Threads and Their Data Close

Intermezzo - Hardware Threads

Hardware Threads and Thread IDs



How Hardware Threads Work



About NUMA and Data Placement

The First Touch Data Placement Policy

Question: 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 defines the fixed Home Node for the particular page

The Goal of Tuning for NUMA

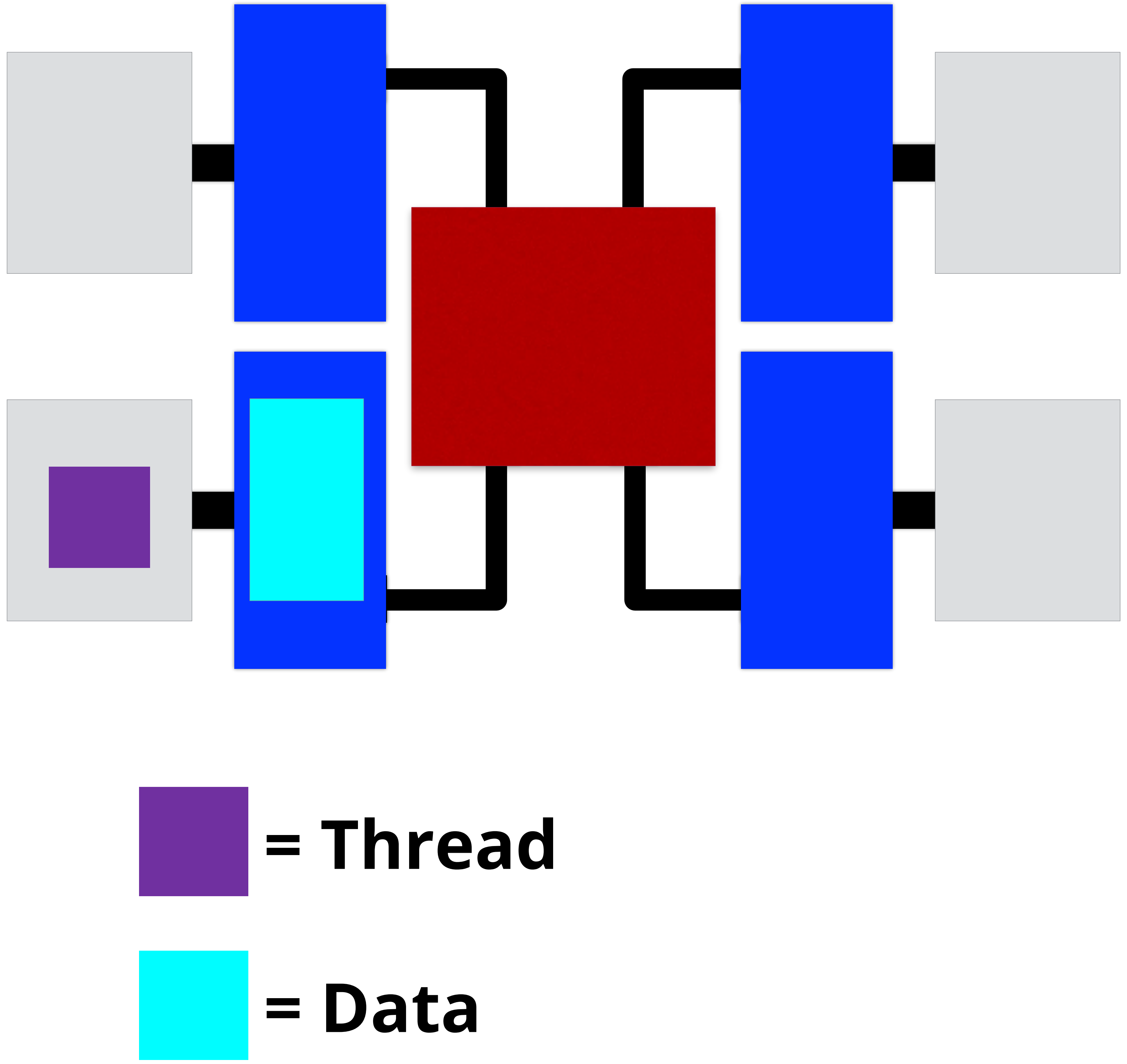
Keep Threads Close to the Home Node of Their Data

A Sequential Initialization

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

One thread executes this loop

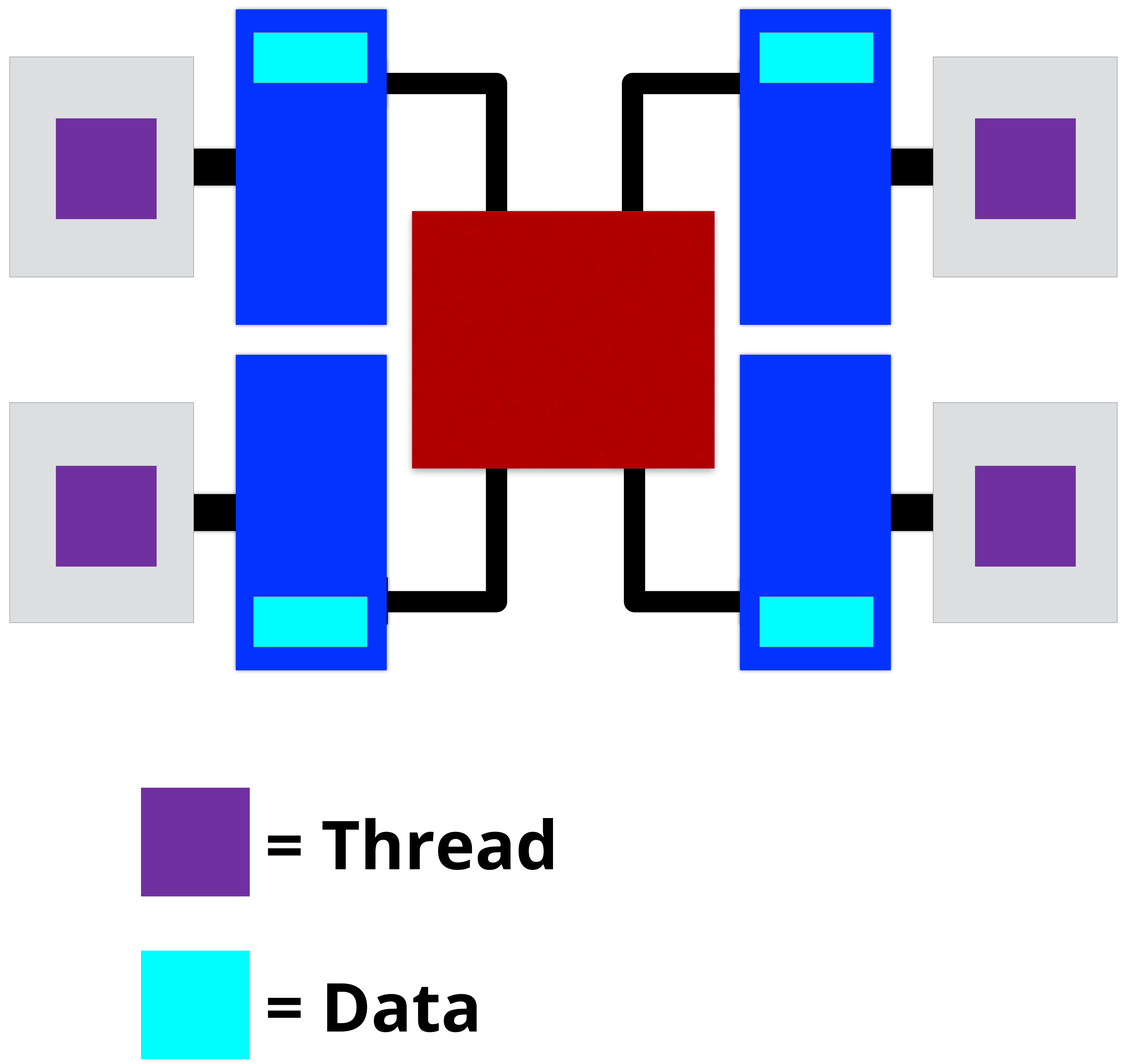
All of "a" is in a single node



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

Note: The allocation is on a virtual memory page basis

OpenMP Support for NUMA

Two NUMA OpenMP Environment Variables

OMP_PLACES
Defines the places where threads may run
OMP_PROC_BIND
Defines how threads map onto the OpenMP places (relevant if there are more places than threads)

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 with the same distance to that memory
sockets	A single socket

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

Hardware Thread ID Support to Define Places

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}

Multicore World XII 2025
12th Edition | 17 - 21 February 2025 | Christchurch, New Zealand

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

Remember this Example?

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

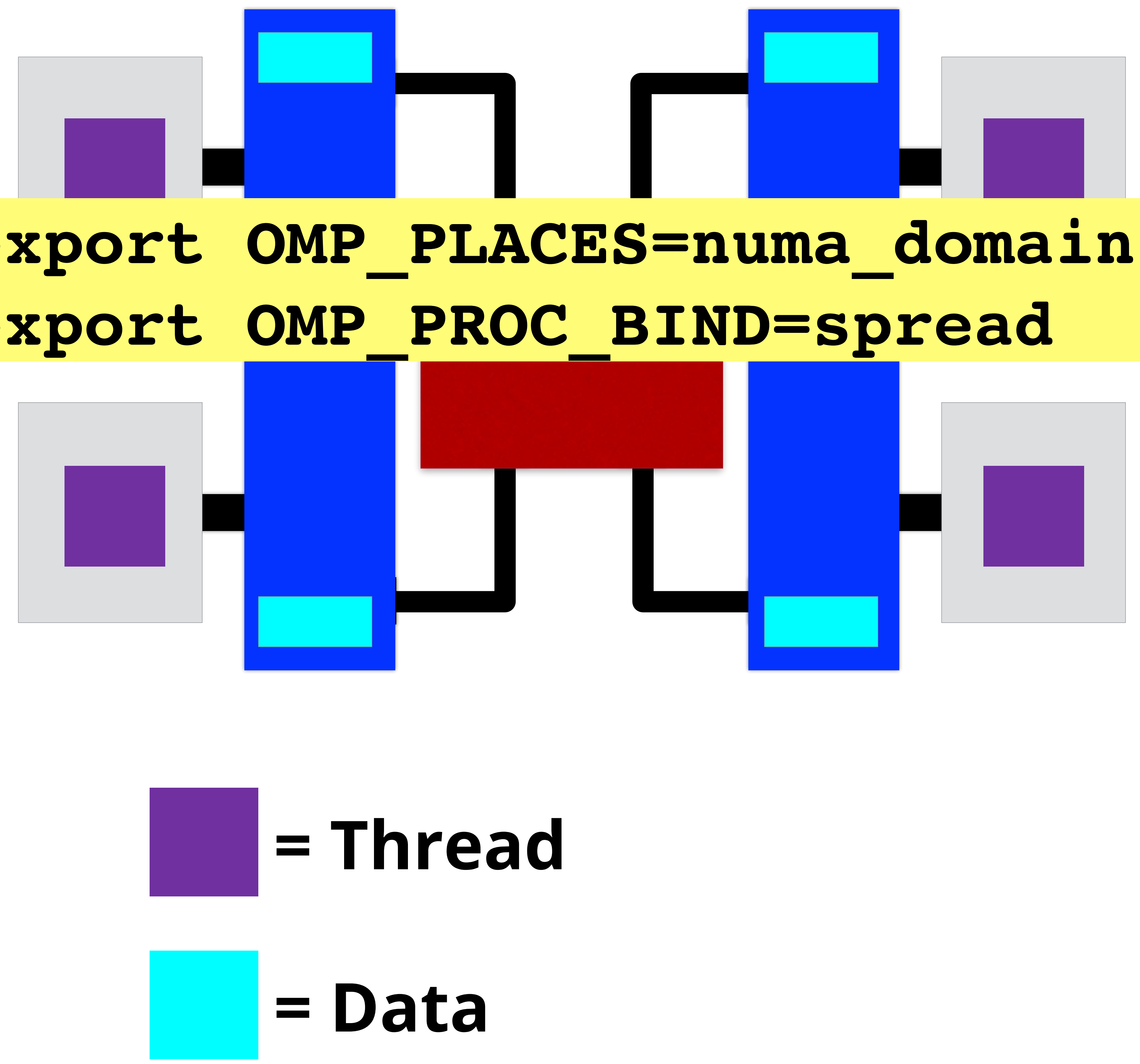
Four threads execute this loop

Wishful Thinking

Data placement depends on where threads execute

Use the NUMA Controls

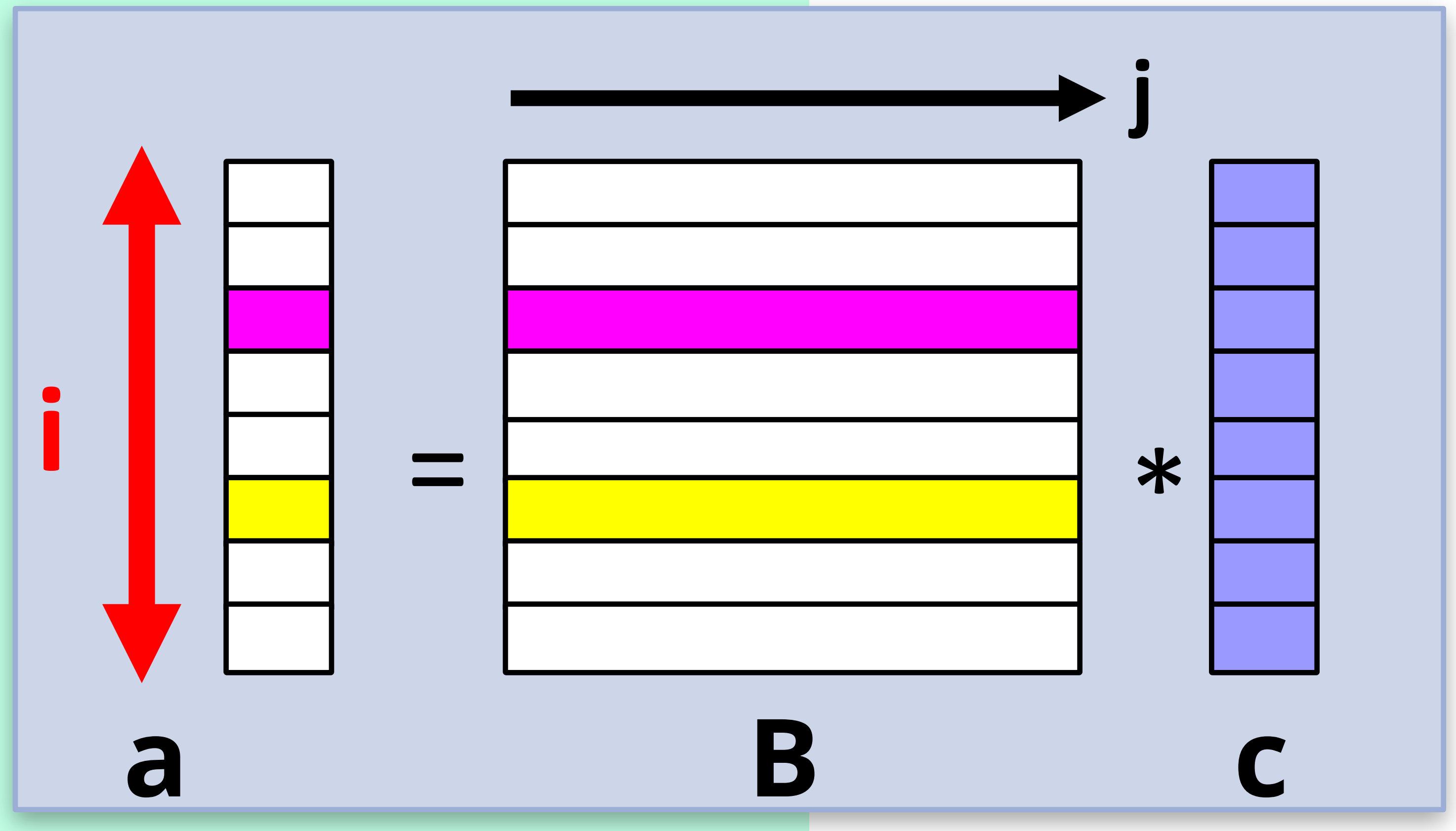
```
$ export OMP_PLACES=numa_domains
$ export OMP_PROC_BIND=spread
```



A Performance Tuning Example

Matrix Times Vector Multiplication: a = B*c

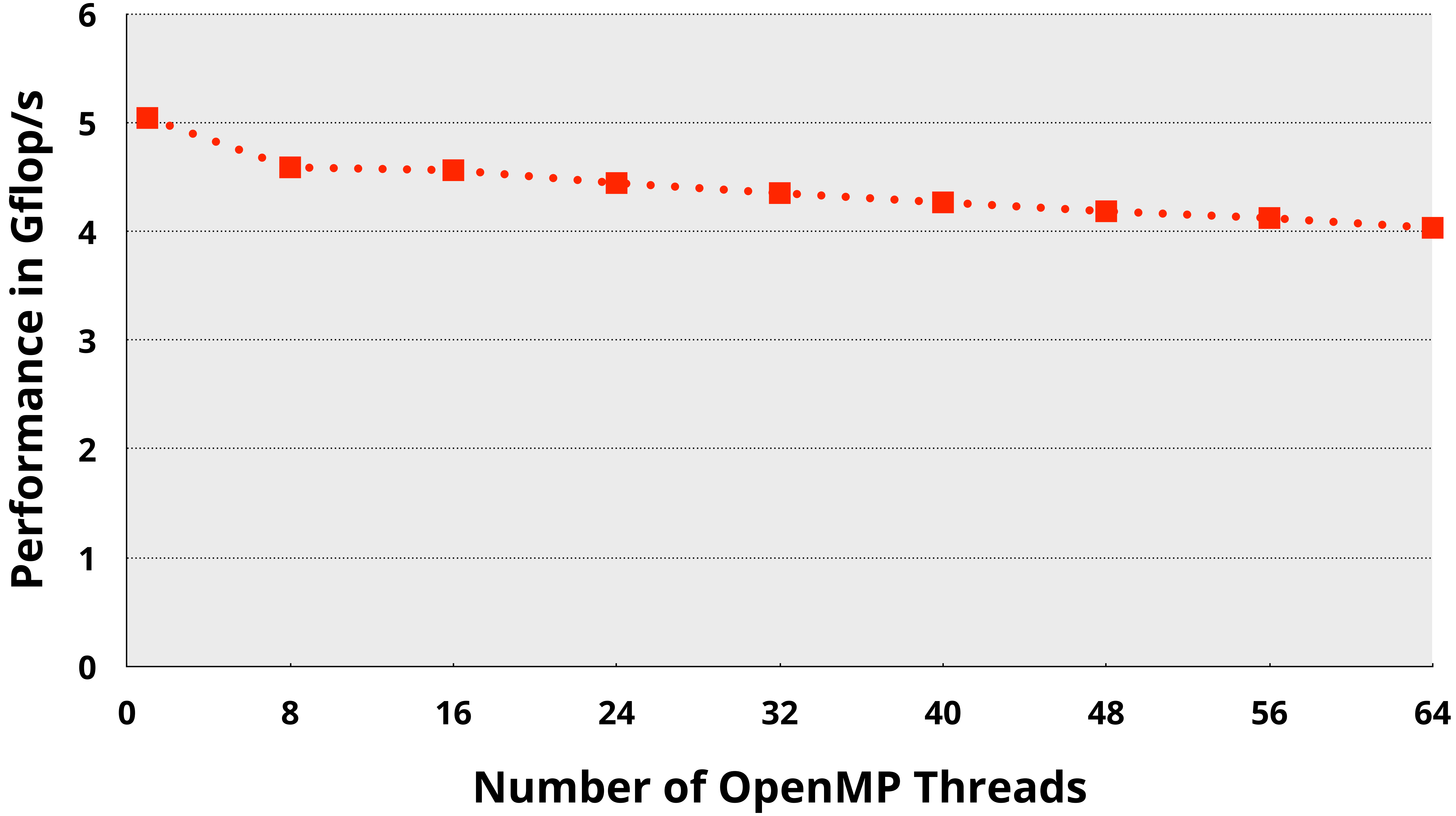
```
#pragma omp parallel for default(none) \
        shared(m,n,a,B,c) schedule(static)
for (int i=0; i<m; i++)
{
  double sum = 0.0;
  for (int j=0; j<n; j++)
     sum += B[i][j]*c[j];
  a[i] = sum;
}
```



As shown here, this algorithm is trivial to parallelize.
One single "omp parallel for" pragma causes
all dotproducts to execute in parallel.

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
```

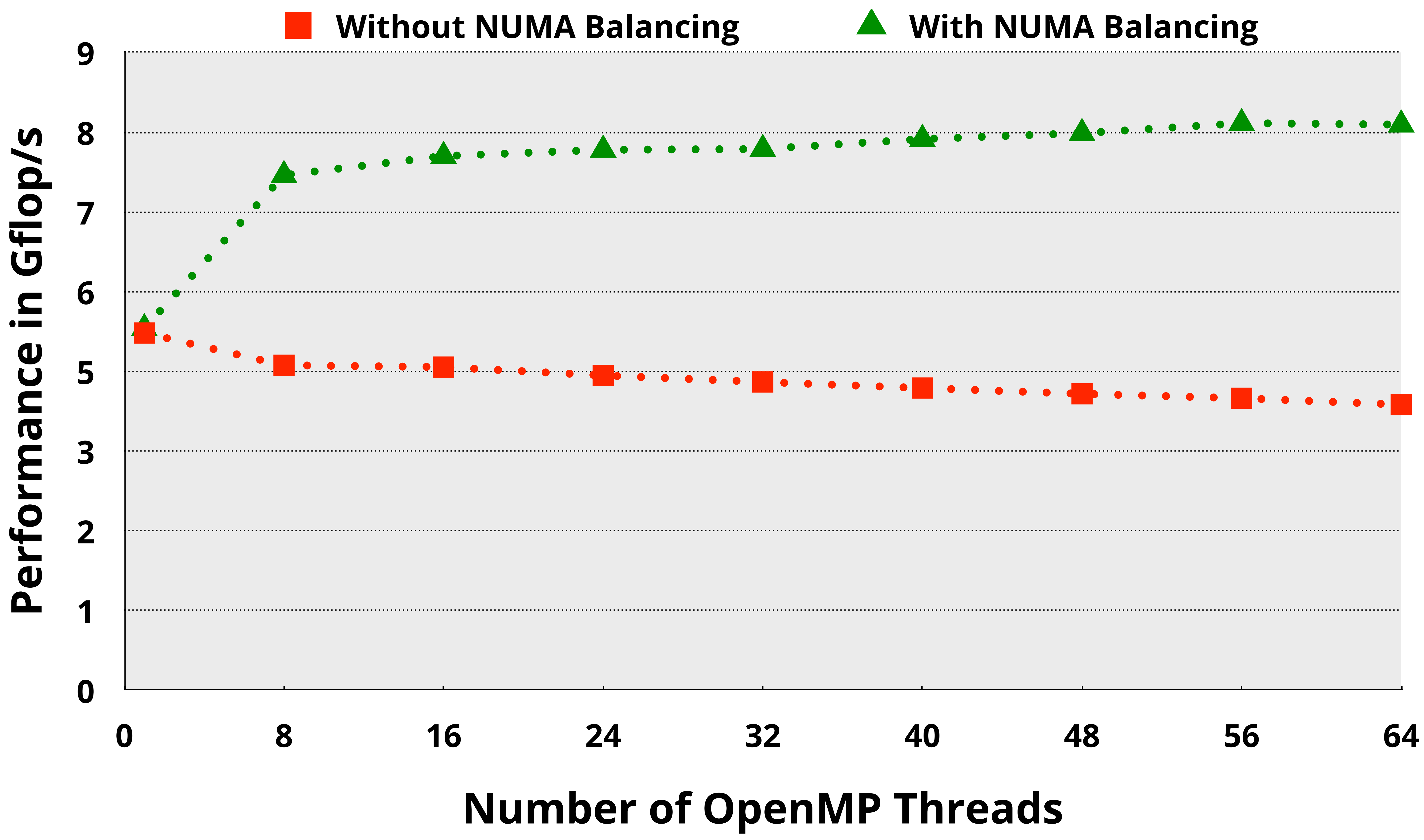
enable

```
# 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!

Understanding Your System

The NUMA Information for a System

8 NUMA Nodes

$ 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
......
```

2 columns => 2 hardware threads/core

$ numactl -H

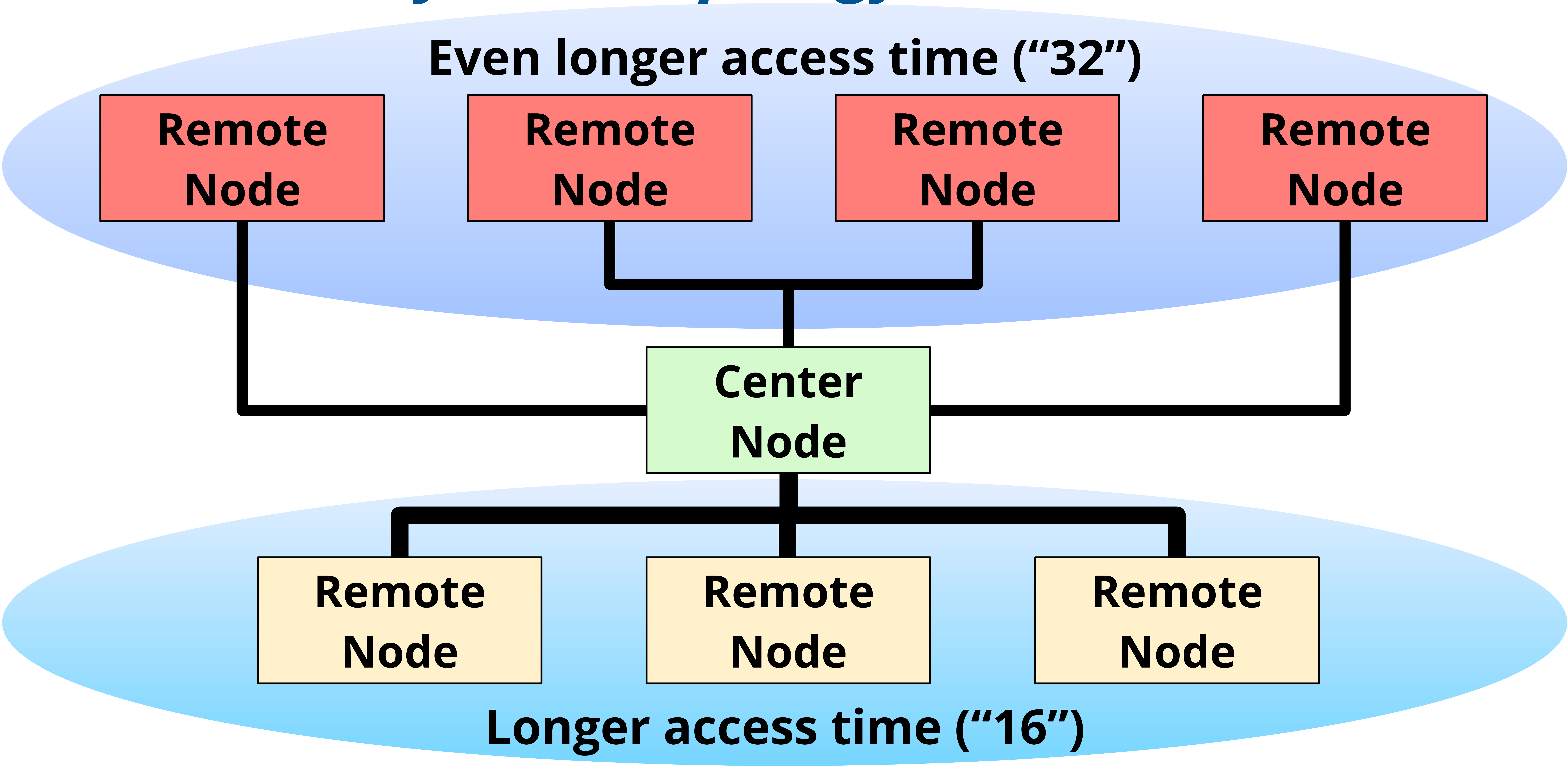
node distances:

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

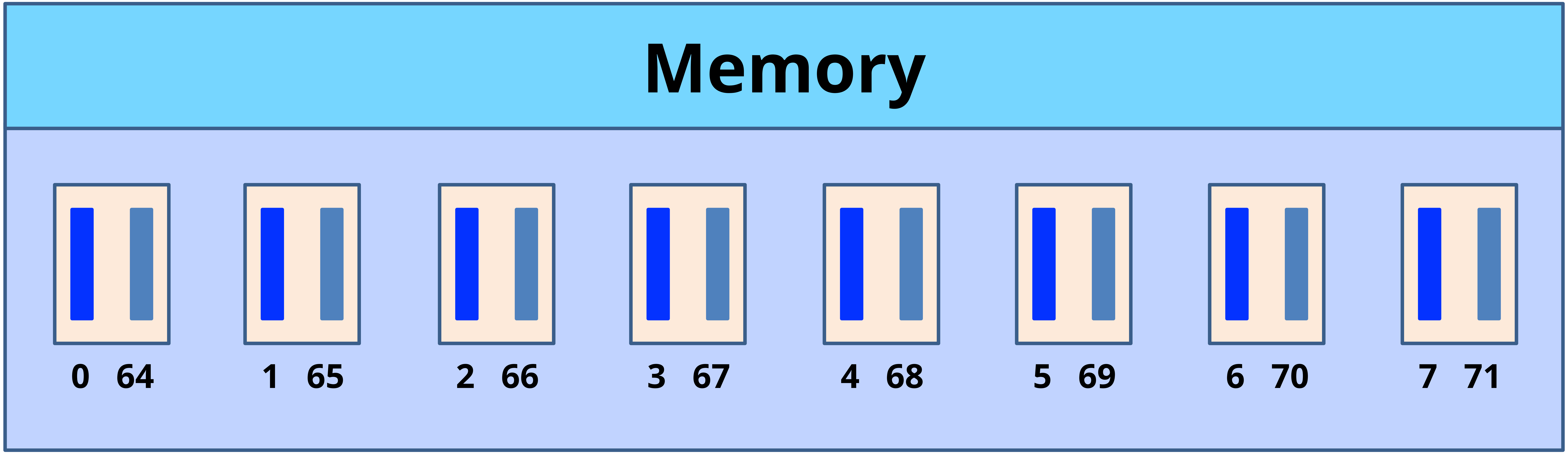
The NUMA Structure of the System

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)



8 cores

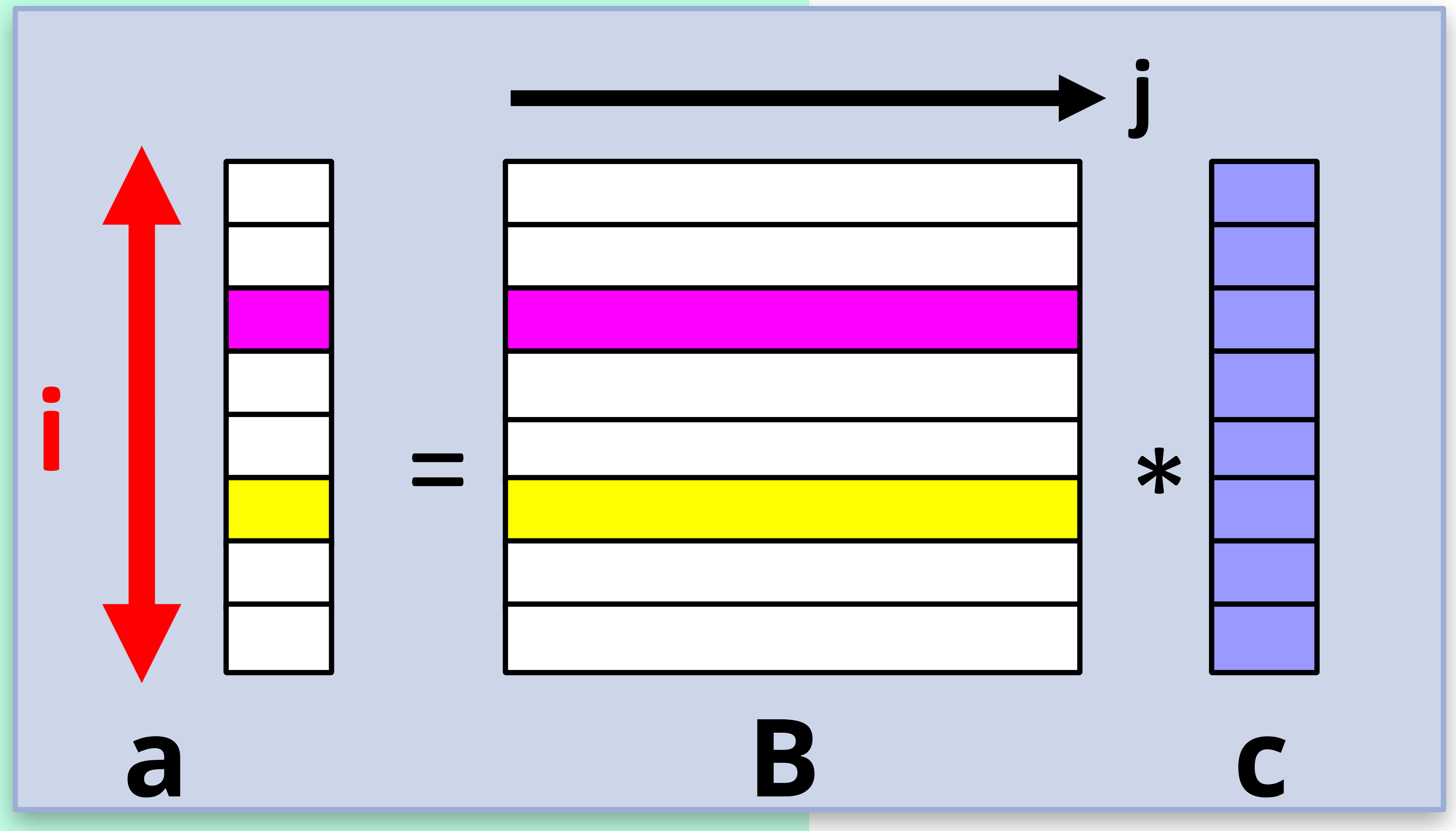
16 hardware threads

All cores and hardware threads share the memory in the node

Improving the Performance

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

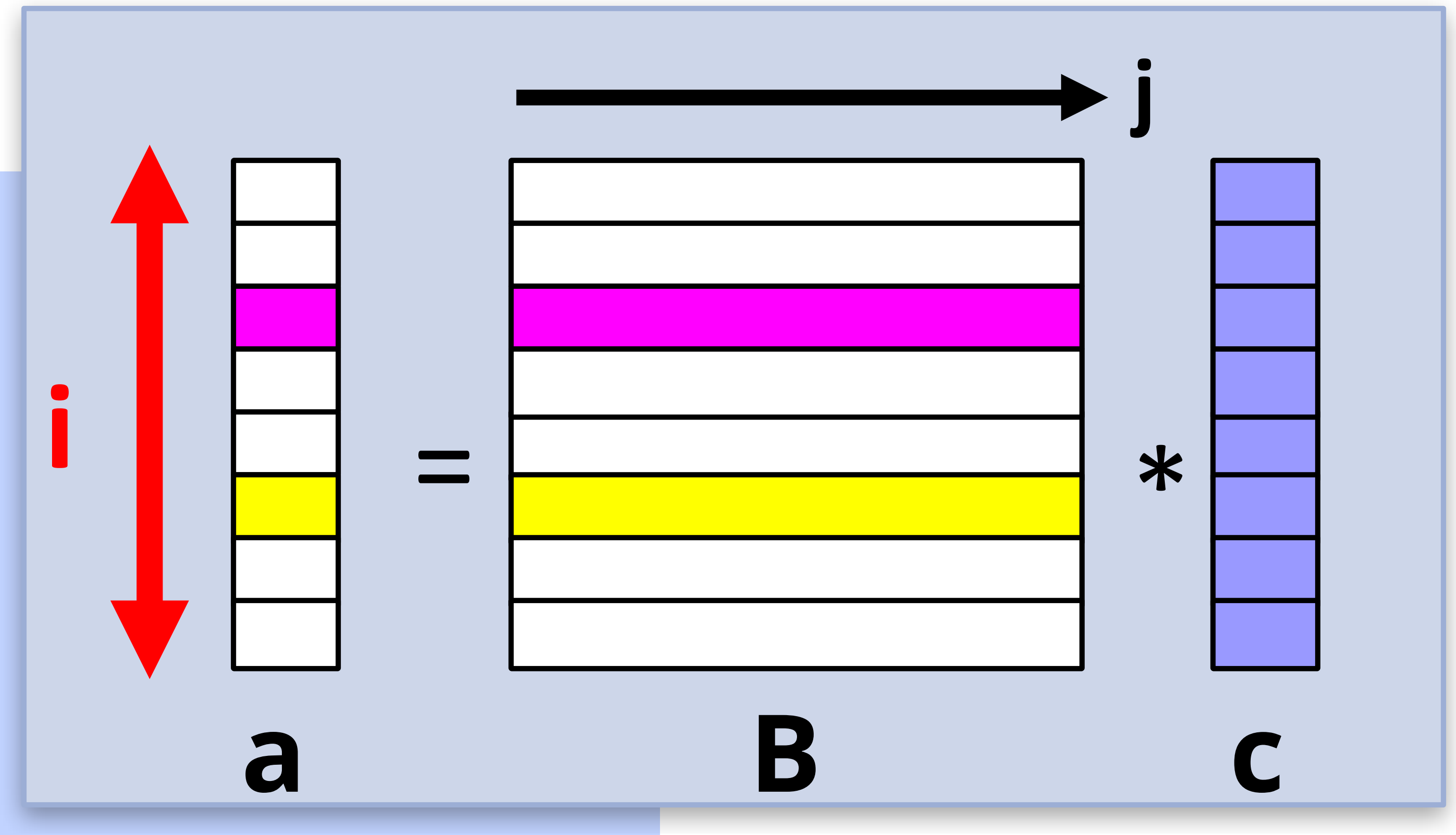
Let's look at a more NUMA friendly data initialization

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
```

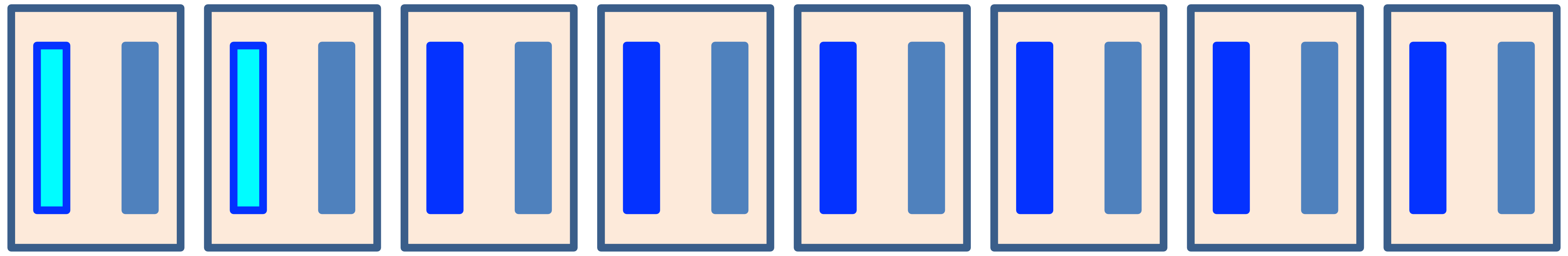


Control the Mapping of Threads

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

2025 MulticoreWorldXII
12th Edition | 17 - 21 February 2025 | Christchurch, New Zealand

An Example How to Use OpenMP Affinity

Expands to the first hardware thread on the first 2 cores on each node:

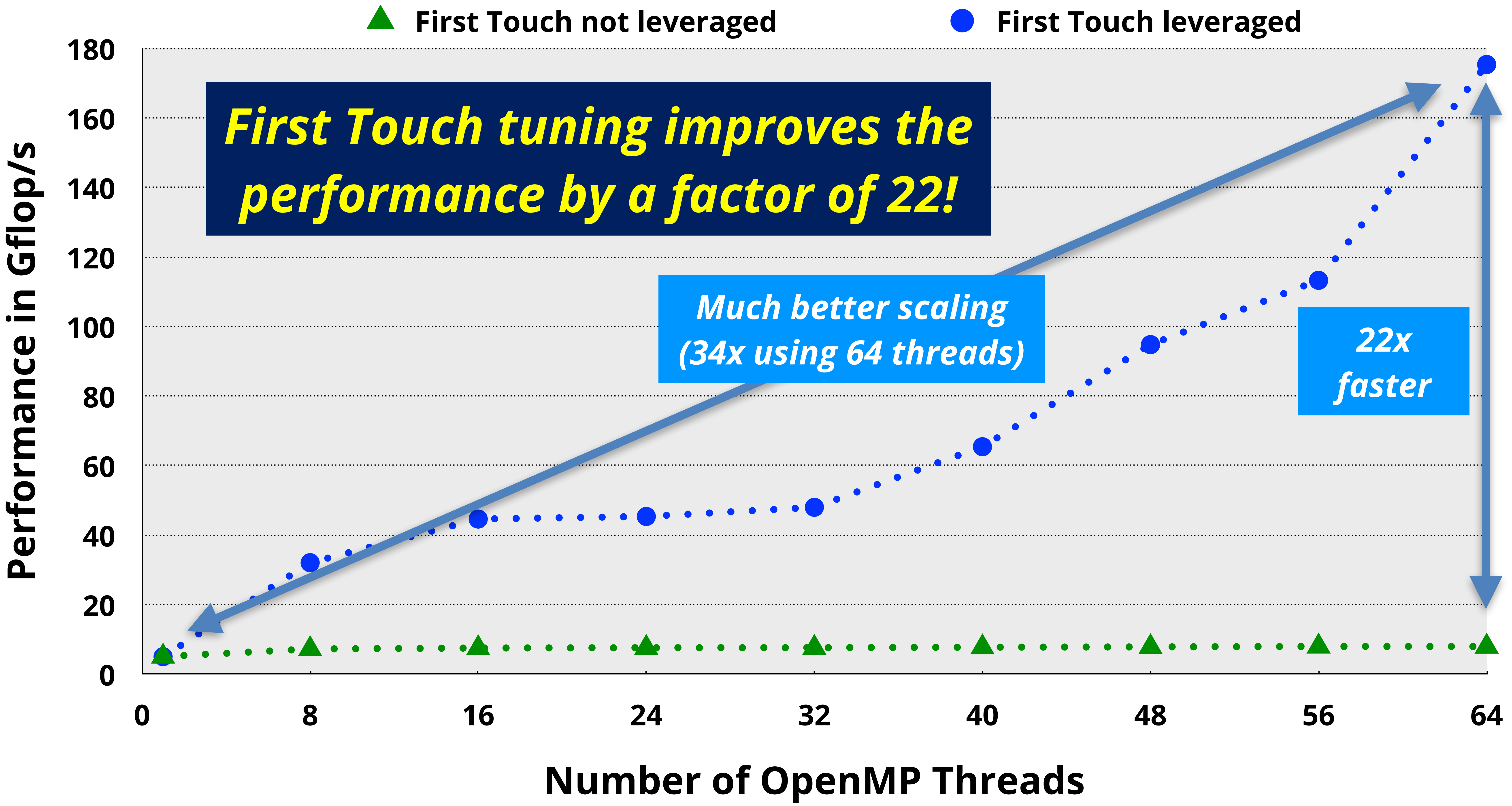
{0}, {8}, {16}, {24}, {32}, {40}, {48}, {56}, {1},{9},{17},{25},{33},{41},{49},{57}

```
$ 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

My Frustration Slide

Started to address NUMA

Paper presented at SC99

Performance Experiences on Sun's WildFire[1] Prototype

Lisa Noordergraaf
High End Server Engineering
Sun Microsystems
Burlington, MA
lisa.noordergraaf@sun.com

Ruud van der Pas
European HPC Team
Sun Microsystems
Geneva, Switzerland
ruud.vanderpas@sun.com

Abstract

This paper presents performance results from work done on Sun's WildFire system. WildFire is a codename for a prototype shared memory multiprocessor developed by Sun Microsystems™ consisting of up to four unmodified Sun Enterprise™ x000 series symmetric multiprocessors (SMPs). A goal of the WildFire system is to evaluate the effectiveness of leveraging large SMPs in the construction of even larger systems.

We have conducted several performance experiments with a shared memory parallelized finite difference solver. Our work demonstrates the key features of the WildFire system, including automatic page migration and read/write replication.

Our results show that the dynamic page migration algorithms used by the WildFire system are effective in automatically optimizing data placement at runtime. Performance comparisons between the WildFire system and currently available SMPs show that the system exhibits good scalability characteristics, and actually outperforms SMPs on this particular application.

which distinguish it from more traditional cc-NUMA machines include its use of large multiprocessors as nodes, and its ability to dynamically migrate and replicate data based on memory access patterns. Data is migrated and replicated at page granularity, but coherence among replicated data is maintained at cache-line granularity.

There are a number of possible advantages associated with dynamic migration and replication; two of the most obvious are using dynamic memory placement policies to relieve the user of having to control data placement explicitly, and also their use in repositioning data after processes are rescheduled on different nodes.

In the work described by this paper we used a standard finite difference solver to explore the impact of dynamic migration and replication. One goal was to determine whether these features are able to improve performance of running applications, and how effective they are at mitigating remote access latencies.

A number of previous studies have shown that dynamic migration and replication can improve overall system and application performance, but such work has generally been


Thank You And … Stay Tuned!