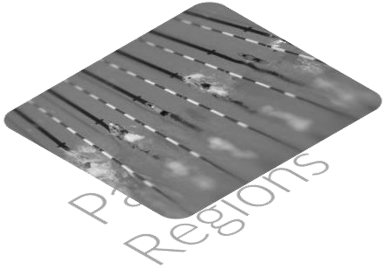
OpenMP
Threads Affinity

Luca Tornatore - I.N.A.F.

DATA SCIENCE & ARTIFICIAL INTELLIGENCE

2024-2025 @ Università di Trieste

OpenMP Outline



Advanced Parallelism

NUMA Awareness

NUMA Outline

- The problem: “Where? Who? What?”
- Touch-first and touch-by-all policy
- Threads affinity

Remind: get back to The typical NUMA architecture

Where do the threads run ?

As we have seen when we discussed the modern architectures, a unique central memory with a fixed bandwidth would be a major bottleneck in a system with a fast growing number of cores/sockets and sockets.

The problem is avoided by physically disjointing the memory in separated units (the *memory banks*) each of which is connected to a socket; All the sockets are inter-connected so that each core can access all the memory and a cache-coherency system "glues" the data.
This way, the resulting aggregated bandwidth scales as the number of sockets (although, we know, the cache-coherency becomes the new limiting factor).

However, the major drawback is that the access time is no more uniform.
This has severe consequences on how you have to write and run your codes.

Where do the threads run ?

OpenMP and the OS offer the capability to decide where each thread have to run, i.e. on which core and/or how the threads have to distribute on the available cores.

We know that each core may have the capability of running more than one thread, which is called (*) Simultaneous MultiThreading (SMT).
In the next slides, let's call ***strands*** or ***hwthreads*** *(hardware threads)* the different threads that a physical core could run, as opposed to "*swthreads*" (software threads) that indicates the OpenMP threads.

The placement of OpenMP threads on cores is called "**threads affinity**".

(*)The *Hyper-Threading* is the proprietary technology, and trademark, of Intel's SMT

Threads affinity

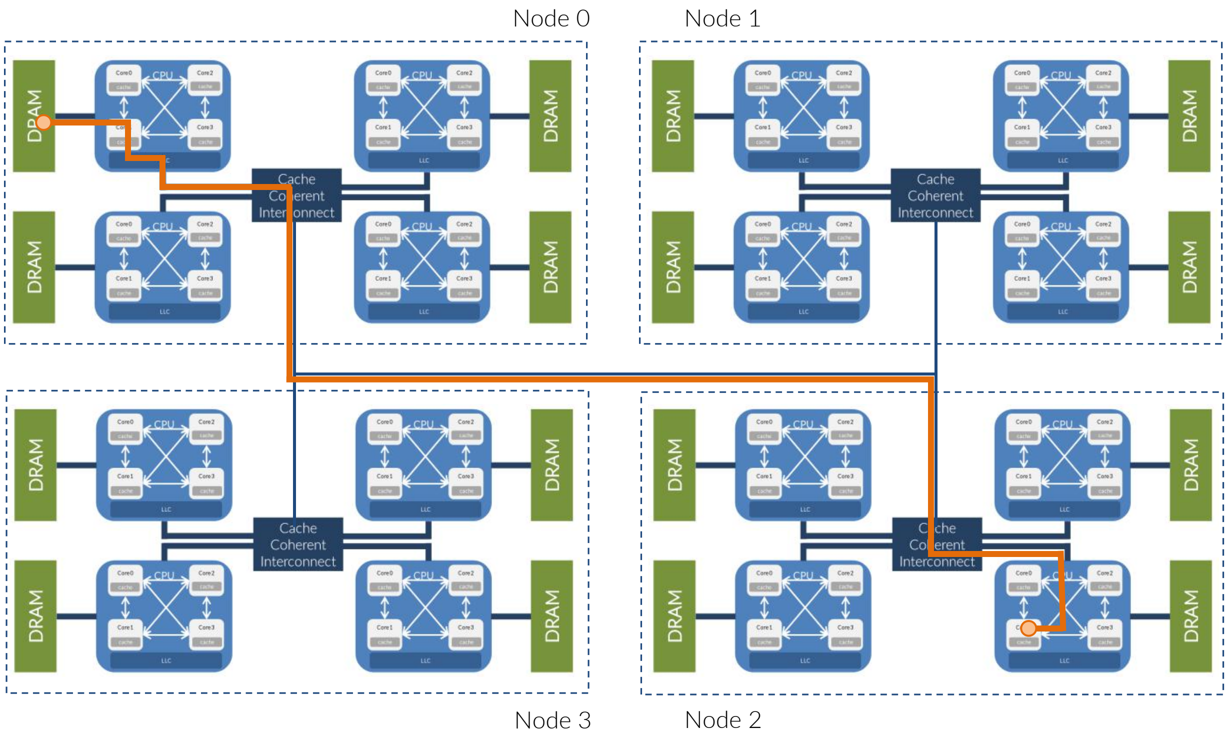
The *Threads affinity* is defined as the mapping of the threads on the underlying cores. The goal is to maximize the efficiency of the memory access on a strongly hierarchical memory system.

As usual, what “efficient” is depends on the details of each specific case.

For instance, if *n* threads work on shared data, it would be more efficient if they share the L2 cache – or in other words, they run on the same socket – so that frequently used data are at hands for all of them.

Conversely, if *n* threads work on independent memory segments, the most efficient choice is to maximize the memory bandwidth over the shortest core-to-ram path.

Where? Who? What?



The aim is to have as few *remote memory accesses* as possible. That depends on

- **Where:** i.e. in what memory bank the data are;
- **Who:** is accessing them, i.e. which thread → how are the threads distributed on the cores;
- **What:** how is the work-load distributed among the threads;

NUMA rationale

In principle, you want to be able to distribute the work in an optimal way, i.e. without any resource (computational power, caches and memory) contention.

To do that, you must be able to place each OpenMP *swthread* to a dedicated computational resource, and to grant it the fastest possible access to “its own” data.

So, you need to:

- explicitly bind the threads to “cores”, i.e. *hwthreads*
- explicitly allocate memory on the best suited physical memory
- minimize the remote memory access
- In case, to migrate memory and/or *swthreads* to one NUMA node to another, or to one *hwthreads* to another respectively

Threads affinity - RATIONALE

What is the “optimal” way to place the swthreads in a node depends on the nature of the algorithm and the data you are dealing with.

Having the swthreads “**distant”** from each other:

- may increase the aggregate bandwidth – i.e. each hwthread could fully exploit its available bandwidth – *if* the data are placed accordingly;
- may result in a better utilization of each core’s cache, because it would be reserved to a single swthread’s data;
- may worsen the performance of synchronization constructs.
- may dispel the cache advantage if the swthreads are reading the same data

Symmetrically, having the swthreads “**close**” to each other:

- may decrease the latency of synchronization constructs;
- may decrease the aggregated bandwidth;
- may worsen/enhance the cache performance depending on what operations are performed on the data.

Threads affinity

OpenMP offers 2 basic concepts to set and control the affinity:

PLACES

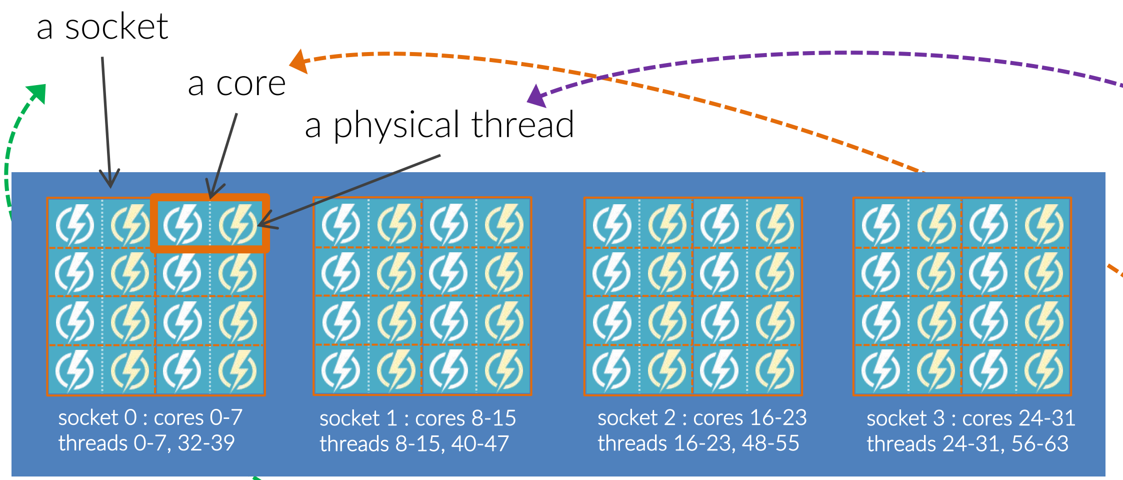
i.e. to what physical entities (*hwthread*) we are referring to with our affinity request: "where" the threads run.

BINDING

i.e. whether there is some relationship between threads and PLACES (in other words: between swthreads and hwthreads), and what relation is it.

NUMA Awareness | # Threads affinity - PLACES

OpenMP



A typical example of configuration for a multicore, multisocket node: 4 sockets, each with 8 cores, each with 2 hwthreads.
Physical threads are exposed as "cores", numbered in a round robin fashion.

PLACES are where swthreads run.
The names for PLACES are:

- **THREADS** each place corresponds to a hwthread, or *strand*, on cores
- **CORES** each place corresponds to a single core (which may have more strands) on sockets
- **SOCKETS** each place corresponds to a physical sockets, with its multiple cores

Threads affinity - examples


```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                96
On-line CPU(s) list:   0-95
Thread(s) per core:    2
Core(s) per socket:    12
Socket(s):             4
NUMA node(s):          4
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz
Stepping:              4
CPU MHz:               1000.073
CPU max MHz:           3200.0000
CPU min MHz:           1000.0000
BogoMIPS:              4600.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              16896K
NUMA node0 CPU(s):     0-11,48-59
NUMA node1 CPU(s):     12-23,60-71
NUMA node2 CPU(s):     24-35,72-83
NUMA node3 CPU(s):     36-47,84-95
```

First hardware threads on sockets.
Do exist also when SMT is switched off

Second hardware threads.
Depends on SMT being active

Socket 0

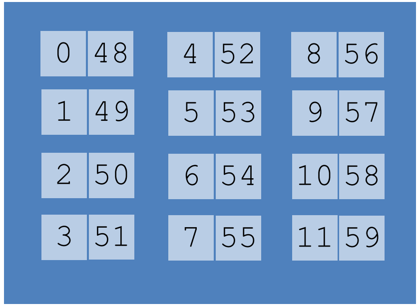
Socket 1

Socket 2

Socket 3

The following examples will refer to a node like the one reported here on the left:

4 sockets
12 cores / socket
2 hwthreads / core



Threads affinity - examples

Socket 0

0 48	4 52	8 56
1 49	5 53	9 57
2 50	6 54	10 58
3 51	7 55	11 59

Socket 1

12 60	16 64	20 68
13 61	17 65	21 69
14 62	18 66	22 70
15 63	19 67	23 71

Socket 2

24 72	28 76	32 80
25 73	29 77	33 81
26 74	30 78	34 82
27 75	31 79	35 83

Socket 3

36 84	40 88	44 92
37 85	41 89	45 93
38 86	42 90	46 94
39 87	43 91	47 95

On a node the computational resources are identified as the physical threads numbered in a round-robin way.

If there are $\mathtt{n_{sockets}}$ with $\mathtt{n_{cores\text{-}per\text{-}socket}}$ then there are

$$\mathtt{n_{cores}} = \mathtt{n_{sockets}} \times \mathtt{n_{cores\text{-}per\text{-}socket}}$$
$$\mathtt{n_{threads}} = \mathtt{n_{cores}} \times \mathtt{n_{SMT\text{-}threads}}$$

The $\mathtt{n_{threads}}$ are the computational resources available on the node; in the following examples we do refer to these IDs

Threads affinity - examples

To clarify the numbering: if the same system shown in the previous slide had `n`$_{\texttt{SMT-threads}}$ `= 4` instead of 2 the numbering would have been as in the right instead of as in the left, here below.

SMT 2	SMT 4
`NUMA node0 CPU(s):   0-11, 48-59`	`NUMA node0 CPU(s):   0-11, 48-59, 96-107, 144-155`
`NUMA node1 CPU(s):   12-23, 60-71`	`NUMA node1 CPU(s):   12-23, 60-71, 108-119, 156-167`
`NUMA node2 CPU(s):   24-35, 72-83`	`NUMA node2 CPU(s):   24-35, 72-83, 120-131, 168-179`
`NUMA node3 CPU(s):   36-47, 84-95`	`NUMA node3 CPU(s):   36-47, 84-95, 132-143, 180-191`

| Threads affinity - PLACES

HOW TO PASS TO OPENMP YOUR PLACES DEFINITION:

the easiest and most practical way is through the env. variable **OMP_PLACES**

```
:> export OMP_PLACES = { sockets | cores | threads }
```

The names listed in the brackets refer exactly to what they mean:

`threads` : refer to logical cores i.e. it takes into account the SMT threads
`cores` : refers to physical cores
`sockets` : refers to sockets

However, we can be much more detailed if needed, as described in the following slides

| Threads affinity - PLACES

Advanced

A "**place**" can be defined by an **unordered** set of comma-separated non-negative numbers enclosed in braces (*the numbers are the IDs of the smallest unit of execution on that hardware, a hwthread*).
"unordered" means that the OS and OpenMP are free to use all the resources specified in the set without any specific priority.

{ 0, 1 }	this defines a place made by hwt 0 and hwt 1 *in the frame of the previous examples, these are the hwt on core 0 and core 1 of socket 0*
{ 0, 48 }	this defines a place made by hwt 0 and hwt 48 *in the frame of the previous examples, these are the hwt and the SMT hwt on core 0 of socket 0*
{ 0, 12, 24, 36 }	this defines a place made by hwt 0, 12, 24, 36 *in the frame of the previous examples, these are the hwt on cores 0 of sockets 0, 1, 2 and 3*
{ 0,1 }, { 1 ,49 }	A list with two places

Advanced

OMP_PLACES can be defined as an explicit **ordered** list of comma-separated places (see the previous slide for a definition of "places").

Intervals can also be used, specified as `start:counter:stride` which results in the serie

`start, start+stride, start + 2×stride, …, start + (counter-1)×stride`

OMP_PLACES = { 0, 48} , {1, 49}	sets OMP_PLACES to 2 places *in the frame of the previous examples, these are the hwt and SMT hwt on cores 0 and 1, respectively, of socket 0*
OMP_PLACES = { 0:2:48}, {1:2:48}	the same than previous line
OMP_PLACES = { 0, 12, 24, 36 }	SET OMP_PLACES to 1 place *in the frame of the previous examples, these are the hwt on cores 0 of sockets 0, 1, 2 and 3*
OMP_PLACES = { 0:4:12 }	the same than previous line

Other examples of places definition by intervals:

{ 0 }:4:12	→ { 0 }, { 12 }, { 24 }, { 36 }
{ 0:4:1 }:4:12	→ { 0,1,2,3 }, { 12,13,14,15 }, { 24,25,26,27 }, { 36,37,38,39 }
{ 0:4}:4:4 { 0:4 }, { 4:4 }, { 8:4 }, { 12:4 }	→ { 0,1,2,3 }, { 4,5,6,7 }, { 8,9,10,11 }, { 12,13,14,15 }
{ 0:12 }:4:12	→ Equivalent to OMP_PLACES=`sockets` on a system with 4 sockets with 12 cores each

The `!` Operator can be used to *exclude* intervals.
The places are *static*: there is no way to change it while the program is running.
It some of the specified places is not available, the behaviour is implementation dependent.

HOW TO PASS TO OPENMP YOUR PLACES DEFINITION:

through the env. variable **OMP_PLACES**

```
:> export OMP_PLACES = { sockets | cores | threads }

:> export OMP_PLACES = "{0}:4:12"

:> export OMP_PLACES = "{0:11,48:11},{24:12,72:12}"
```

RESUMÉ:

We have just learnt how to define the ***places*** the swthreads will run during execution.
Each place is composed by 1 or more physical resources that are pretty equivalent from the point of view of the OS or OpenMP (i.e. all the resources in a place are equivalently good/“legal” for the placement of a swthread).

The next fundamental question is how the swthreads are placed among the available places ?
That is decided upon the **threads affinity policy**, which is defined through the “thread binding” on places: once the destination place for running has been decided, the swthreads are not allowed to move out of that place (it may be re-scheduled on a different resource in the same place, but still in the same place)

Threads affinity - BINDING

The **BINDING** defines how the swthreads are mapped onto the PLACES.

The names for BINDING are listed here on the right

- **NONE** the placement is up to the OS
- **CLOSE** the swthreads are placed onto places as close as possible to each other (assigned to consecutive places in a round-robin way)
- **SPREAD** the swthreads are placed onto places as evenly as possible, then the places are filled in a round-robin fashion
- **MASTER** the swthreads run onto the same place than master thread

Threads affinity - BINDING

HOW TO PASS TO OPENMP YOUR BINDING REQUEST:

(1) through the env. tariable **OMP_PROC_BIND**

```
:> export OMP_PROC_BIND = { false | true | master | close | spread }
```

this amounts to ask no policy (i.e. “none”), so that the O.S. will decide the placement, and to allow the O.S. to migrate the threads.

this amounts to ask no policy (i.e. “none”), so that the O.S. will decide the placement, BUT forbid the O.S. to migrate the threads.

these 3 options amount to ask a precise policsy and forbid the O.S. to migrate the threads.

Threads affinity - BINDING

(2) The binding can be specified in a non-persistent way for each parallel region *inside* the code:

```
#pragma omp parallel proc_bind(policy)
```

Once a swthread has been assigned to a hwthread, it is not allowed to migrate. If you have *nested parallelism*, you may define different behaviour for the nested regions

```
#pragma omp parallel proc_bind(spread)
{
   #pragma omp parallel for proc_bind(close)
   for ( int ii = 0; ii < local_N; ii++ )
}
```

- T ≤ P : there are sufficient places for a unique assignment. swthreads are assigned to *consecutive places* by their thread ID. The first place is the master's place. *"consecutive" must be intended in physical sense, so that if the places is "threads" the hwthreads on a same core are the closest to each other.*

CLOSE

- T > P : at least one place executes more than one swthread. swthread are splitted in P subsets St_i, so that

$$\texttt{floor(T/P)} \leq \texttt{St}_\texttt{i} \leq \texttt{ceiling(T/P)}$$

$\texttt{St}_0$ includes swt 0 and is assigned to the master's place.

Threads affinity - BINDING

SPREAD

- T ≤ P : place list is splitted in `T` subpartitions; each subpartition contains at least `floor(P/T)` and at most `ceiling(P/T)` *consecutive places*. A thread is then assigned to a subpartition, starting from the master thread. Then, assignment proceeds by thread ID, and the threads are placed in the first place of the next subpartition.

- T > P : place list is splitted in `P` subpartitions, each of which contains only 1 place and St_i *threads* with consecutive IDs.
  The number of threads St_i in each subpartition is chosen so that:
  $\texttt{floor(T/P)} \leq St_i \leq \texttt{ceiling(T/P)}$
  At least one place has more than one thread assigned to it.
  The first subset with St_0 contains thread `0` and runs on the place that hosts the master thread.

Threads affinity - examples

places / binding	THREADS	CORES	SOCKETS
CLOSE	swt are placed on close hwt, saturating all the SMT hwt in each core before using new cores	swt are placed on close hwt, using 1 hwt/core before starting to use SMT	swt are placed round-robin per socket, 1/core; after saturation, SMT is used by round-robin +1 hwt/socket
SPREAD	swt are placed round-robin sockets, onto free cores in sockets	similar to ← SMT is avoided until saturation	similar to ← swt are placed by round-robin sockets and hwt
MASTER	all swt are placed on the same hwt on the same core on the same socket	all swt are placed on the same core on the same socket, using all its hwt	all swt are placed on the same socket, saturing all hwt starting from SMT ones

Using the **abstract names** for places

note:
`swt = software threads`
`hwt = hardware threads`

Threads affinity - examples

	THREADS	CORES	SOCKETS
CLOSE			
SPREAD			
MASTER			

OMP_PLACES = threads

OMP_PROC_BIND = close

There are 96 places.
swt are placed on close hwt, saturating all the siblings SMT hwt in each core before using new cores

numa_awareness/
00_where_I_am.c

4 swthreads

0	48	4	52	8	56
1	49	5	53	9	57
2	50	6	54	10	58
3	51	7	55	11	59

Socket 0

7 swthreads

0	48	4	52	8	56
1	49	5	53	9	57
2	50	6	54	10	58
3	51	7	55	11	59

Socket 0

18 swthreads

0	48	4	52	8	56
1	49	5	53	9	57
2	50	6	54	10	58
3	51	7	55	11	59

Socket 0

25 swthreads

12	60	16	64	20	68
13	61	17	65	21	69
14	62	18	66	22	70
15	63	19	67	23	71

Socket 1 (S0 saturated)

	THREADS	CORES	SOCKETS
CLOSE	■		
SPREAD			
MASTER			

OMP_PLACES = threads

OMP_PROC_BIND = close

There are 96 places.
swt are placed on close hwt, saturating all the siblings SMT hwt in each core before using new cores

4 swthreads

	hwthreads	swthreads
node 0	00-11 , 48-59	0,48,1,49 ●●○○
node 1	12-23 , 60-71	
node 2	24-35 , 72-83	
node 3	36-47 , 84-95	

7 swthreads

	hwthreads	swthreads
node 0	00-11 , 48-59	0,48,1,49,2,50,3 ●●●●○○○
node 1	12-23 , 60-71	
node 2	24-35 , 72-83	
node 3	36-47 , 84-95	

18 swthreads

	hwthreads	swthreads
node 0	00-11 , 48-59	0,48,1,49,2,50, 3,51,4,52,5,53, 6,54,7,55,8,56 ●●●●●●●●● ○○○○○○○○○
node 1	12-23 , 60-71	
node 2	24-35 , 72-83	
node 3	36-47 , 84-95	

swthreads places are reported by ID order.
- ● *means hwthread*
- ○ *means SMT hwthread*

numa_awareness/
00_where_I_am.c

| Threads affinity - examples

	THREADS	CORES	SOCKETS
CLOSE	■		
SPREAD			
MASTER			

OMP_PLACES = threads

OMP_PROC_BIND = close

There are 96 places.
swt are placed on close hwt, saturating all the siblings SMT hwt in each core before using new cores

25 swthreads

	hwthreads	swthreads
node 0	00-11 , 48-59	SATURATED
node 1	12-23 , 60-71	12 ●
node 2	24-35 , 72-83	
node 3	36-47 , 84-95	

50 swthreads

	hwthreads	swthreads
node 0	00-11 , 48-59	SATURATED
node 1	12-23 , 60-71	SATURATED
node 2	24-35 , 72-83	24, 72 ●○
node 3	36-47 , 84-95	

numa_awareness/
00_where_I_am.c

| Threads affinity - examples

	THREADS	CORES	SOCKETS
CLOSE			
SPREAD			
MASTER			

OMP_PLACES = cores

OMP_PROC_BIND = close

There are 48 places now.
swt are placed on close hwt, using 1 hwt/core.
When a socket is full, placement continues with the next socket.

numa_awareness/
00_where_I_am.c

4 swthreads

0	48	4	52	8	56
1	49	5	53	9	57
2	50	6	54	10	58
3	51	7	55	11	59

Socket 0

7 swthreads

0	48	4	52	8	56
1	49	5	53	9	57
2	50	6	54	10	58
3	51	7	55	11	59

Socket 0

18 swthreads

0	48	4	52	8	56
1	49	5	53	9	57
2	50	6	54	10	58
3	51	7	55	11	59

Socket 0

Threads affinity - examples

	THREADS	CORES	SOCKETS
CLOSE			■
SPREAD			
MASTER			

OMP_PLACES = sockets

OMP_PROC_BIND = close

There are 4 places.
swt are placed round-robin per socket, 1/core; after saturation, SMT is used by round-robin +1 hwt/socket

▷ `numa_awareness/00_where_I_am.c`

4 swthreads

Socket 0

0 48	4 52	8 56
1 49	5 53	9 57
2 50	6 54	10 58
3 51	7 55	11 59

Socket 1

12 60	16 64	20 68
13 61	17 65	21 69
14 62	18 66	22 70
15 63	19 67	23 71

Socket 2

24 72	28 76	32 80
25 73	29 77	33 81
26 74	30 78	34 82
27 75	31 79	35 83

Socket 3

36 84	40 88	44 92
37 85	41 89	45 93
38 86	42 90	46 94
39 87	43 91	47 95

Threads affinity - examples

	THREADS	CORES	SOCKETS
CLOSE			
SPREAD			
MASTER			

OMP_PLACES = sockets

OMP_PROC_BIND = close

There are 4 places.
swt are placed round-robin per socket, 1/core; after saturation, SMT is used by round-robin +1 hwt/socket

4 swthreads

`numa_awareness/00_where_I_am.c`

NOTE: *for the sake of clarity in the previous slide we picked-up the first hwthread on each socket; however, since the place is the entire socket your threads may be placed wherever in each socket, like in this example*

0	48	4	**52**	8	56
1	49	5	53	9	57
2	50	6	54	10	58
3	51	7	55	11	59

Socket 0

12	60	16	64	20	68
13	61	17	65	21	69
14	62	**18**	66	22	70
15	63	19	67	23	71

Socket 1

24	72	28	76	32	80
25	73	29	77	33	81
26	74	30	78	34	82
27	75	31	79	35	83

Socket 2

36	84	40	88	44	92
37	85	41	89	45	**93**
38	86	42	90	46	94
39	87	43	91	47	95

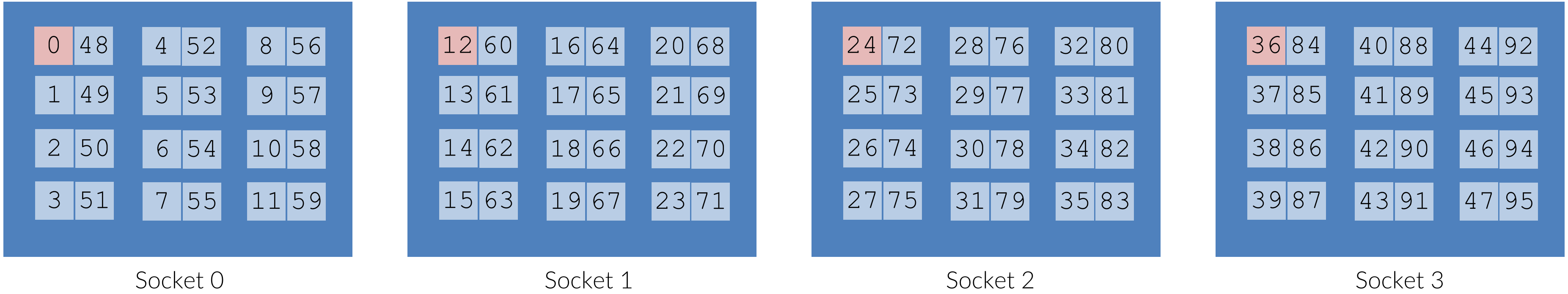
Socket 3

| Threads affinity - examples

OMP_PLACES = sockets

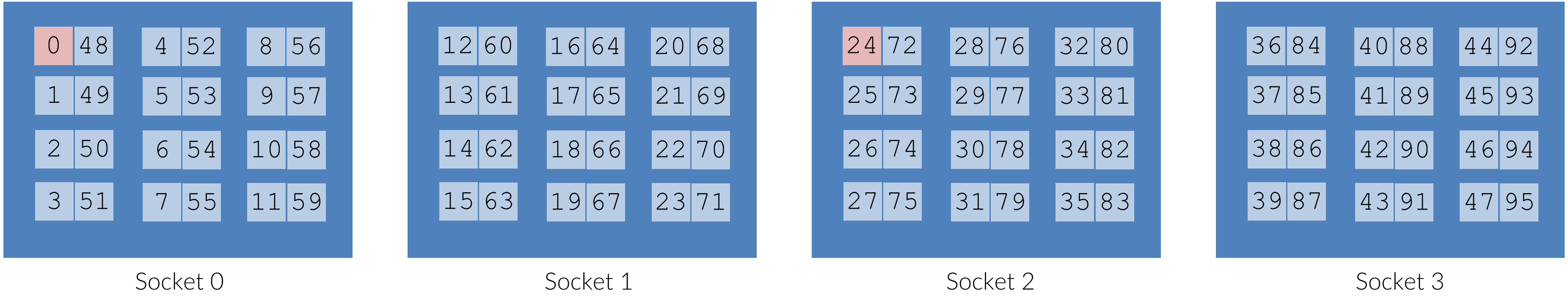
OMP_PROC_BIND = close

In this case (T=P) the `close` and `spread` policies produce the same distribution. To further clarify the difference between the two, let's examine the case T<P with T=2

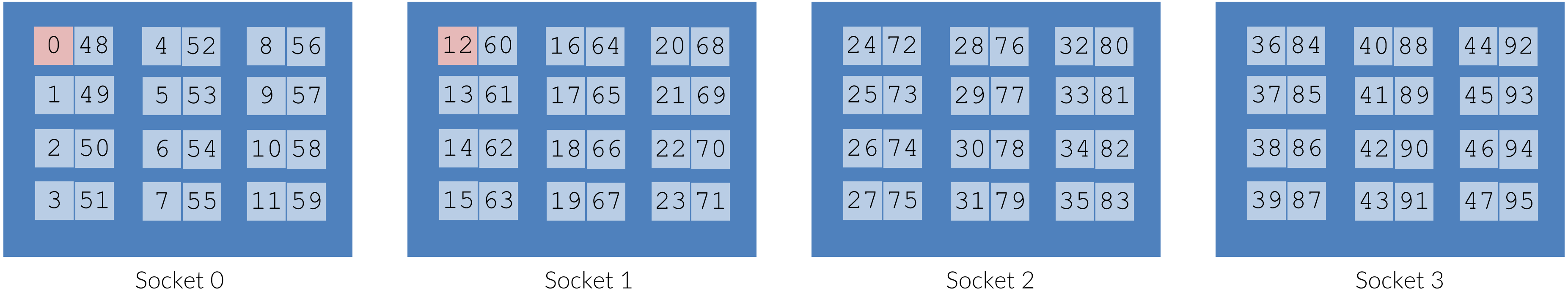


Threads affinity - examples

PLACES = sockets; BINDING = spread (two subpartitions {0,1} and {2,3}, a thread on each)



PLACES = sockets; BINDING = close (threads assigned to *consecutive* places by their thread id)



Threads affinity - examples

	THREADS	CORES	SOCKETS
CLOSE			
SPREAD			
MASTER			

OMP_PLACES = sockets

OMP_PROC_BIND = close

There are 4 places.
swt are placed round-robin per socket, 1/core; after saturation, SMT is used by round-robin +1 hwt/socket

numa_awareness/
00_where_I_am.c

14 swthreads

0	48	4	52	8	56
1	49	5	53	9	57
2	50	6	54	10	58
3	51	7	55	11	59

Socket 0

12	60	16	64	20	68
13	61	17	65	21	69
14	62	18	66	22	70
15	63	19	67	23	71

Socket 1

24	72	28	76	32	80
25	73	29	77	33	81
26	74	30	78	34	82
27	75	31	79	35	83

Socket 2

36	84	40	88	44	92
37	85	41	89	45	93
38	86	42	90	46	94
39	87	43	91	47	95

Socket 3

	THREADS	CORES	SOCKETS
CLOSE			
SPREAD			
MASTER			

OMP_PLACES = threads

OMP_PROC_BIND = spread

swt are placed round-robin sockets, onto free cores in sockets

`numa_awareness/00_where_I_am.c`

4 swthreads

Socket 0

0	48	4	52	8	56
1	49	5	53	9	57
2	50	6	54	10	58
3	51	7	55	11	59

Socket 1

12	60	16	64	20	68
13	61	17	65	21	69
14	62	18	66	22	70
15	63	19	67	23	71

Socket 2

24	72	28	76	32	80
25	73	29	77	33	81
26	74	30	78	34	82
27	75	31	79	35	83

Socket 3

36	84	40	88	44	92
37	85	41	89	45	93
38	86	42	90	46	94
39	87	43	91	47	95

	THREADS	CORES	SOCKETS
CLOSE			
SPREAD			
MASTER			

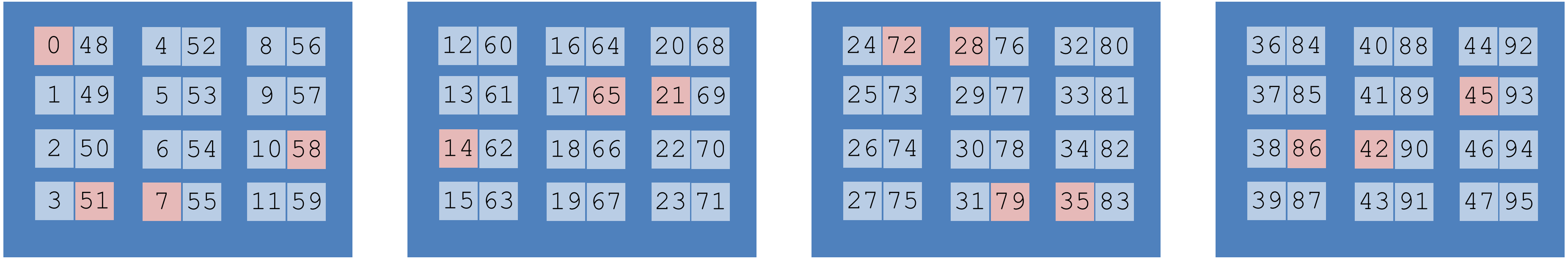
OMP_PLACES = threads

OMP_PROC_BIND = spread

swt are placed round-robin sockets, onto free cores in sockets

numa_awareness/00_where_I_am.c

14 swthreads



Socket 0 Socket 1 Socket 2 Socket 3

| Threads affinity - examples

	THREADS	CORES	SOCKETS
CLOSE			
SPREAD			
MASTER			

OMP_PLACES = cores

OMP_PROC_BIND = spread

	THREADS	CORES	SOCKETS
CLOSE			
SPREAD			
MASTER			

OMP_PLACES = sockets

OMP_PROC_BIND = spread

Similar to (thread, spread), just infer the differences from the table description.
And, run on your own `00_whare_I_am` to check what is happening.

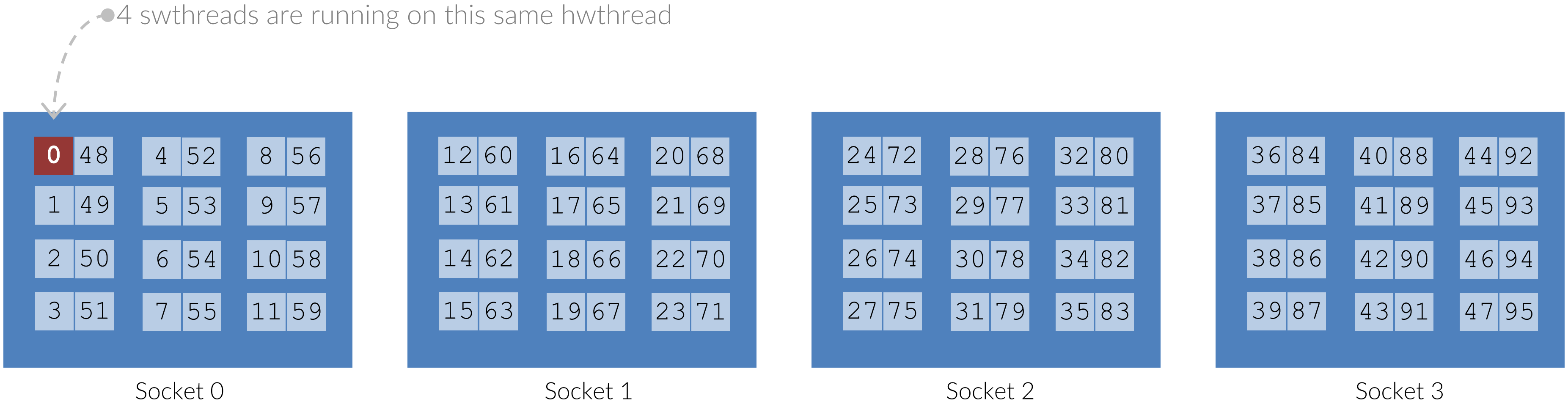
numa_awareness/
00_where_I_am.c

	THREADS	CORES	SOCKETS
CLOSE			
SPREAD			
MASTER			

OMP_PLACES = threads

OMP_PROC_BIND = master

4 swthreads



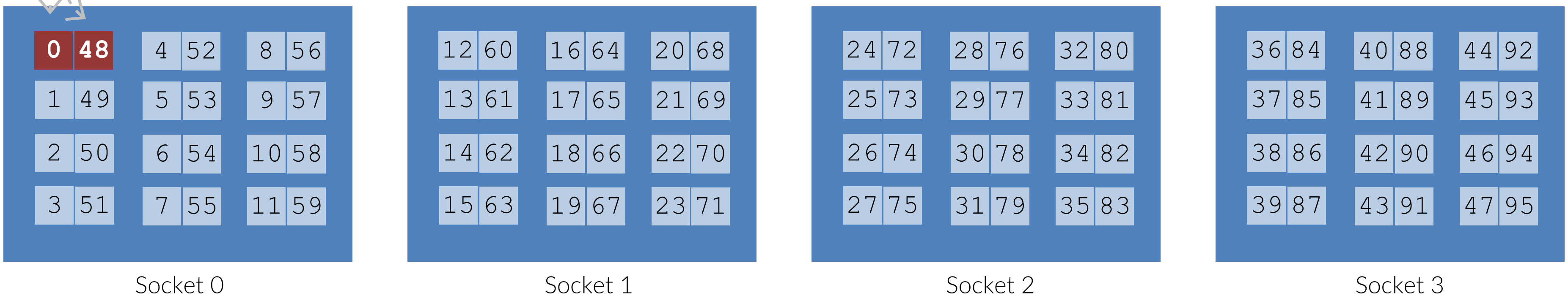
	THREADS	CORES	SOCKETS
CLOSE			
SPREAD			
MASTER		■	

OMP_PLACES = cores

OMP_PROC_BIND = master

4 swthreads

2 swthreads are running on each of these 2 hwthreads

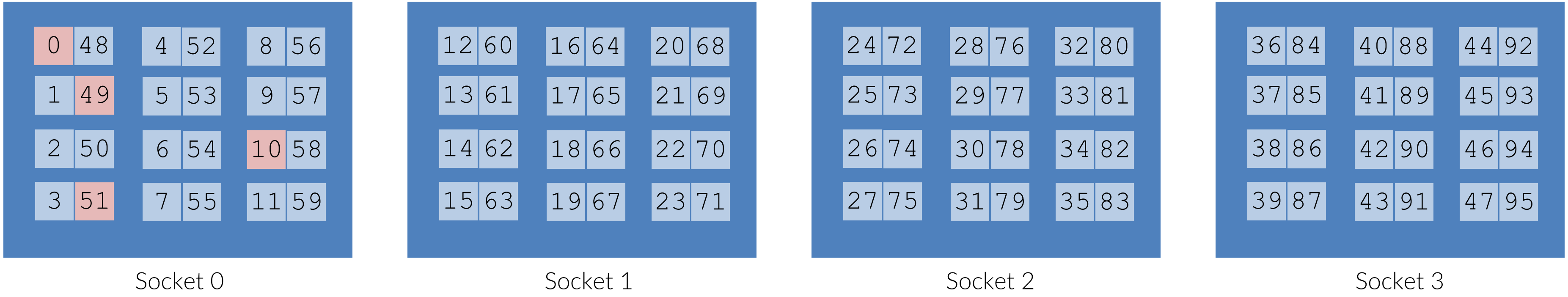


	THREADS	CORES	SOCKETS
CLOSE			
SPREAD			
MASTER			

OMP_PLACES = sockets

OMP_PROC_BIND = master

4 swthreads



| Threads affinity - examples

	THREADS	CORES	SOCKETS
CLOSE			
SPREAD			
MASTER			

OMP_PLACES = sockets

OMP_PROC_BIND = master

20 swthreads

0	48	4	52	8	56
1	49	5	53	9	57
2	50	6	54	10	58
3	51	7	55	11	59

Socket 0

12	60	16	64	20	68
13	61	17	65	21	69
14	62	18	66	22	70
15	63	19	67	23	71

Socket 1

24	72	28	76	32	80
25	73	29	77	33	81
26	74	30	78	34	82
27	75	31	79	35	83

Socket 2

36	84	40	88	44	92
37	85	41	89	45	93
38	86	42	90	46	94
39	87	43	91	47	95

Socket 3

| Threads affinity - examples

	THREADS	CORES	SOCKETS
CLOSE			
SPREAD			
MASTER			

OMP_PLACES = sockets

OMP_PROC_BIND = master

35 swthreads

When all the hwthreads are saturated, more than 1 swthread is placed on hwthreads by round-robin, on the same socket

Socket 0

0	48	**4**	52	**8**	56
1	**49**	**5**	53	9	**57**
2	50	**6**	54	**10**	58
3	**51**	7	**55**	11	**59**

Socket 1

12	60	16	64	20	68
13	61	17	65	21	69
14	62	18	66	22	70
15	63	19	67	23	71

Socket 2

24	72	28	76	32	80
25	73	29	77	33	81
26	74	30	78	34	82
27	75	31	79	35	83

Socket 3

36	84	40	88	44	92
37	85	41	89	45	93
38	86	42	90	46	94
39	87	43	91	47	95

Threads affinity

If you compile `00_whare_I_am.c` with `-DSPY` , it will load the hwthreads with some amount of work so that meanwhile you can inspect what is happening by using the either the htop or the top utility:

On my laptop using 2 swthreads

```
top - 14:11:56 up  5:29,  4 users,  load average: 3.01, 2.10, 1.81
Threads: 899 total,   3 running, 828 sleeping,   0 stopped,   0 zombie
%Cpu0  : 16.4 us,  7.0 sy,  0.0 ni, 76.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :  9.8 us,  3.9 sy,  0.0 ni, 86.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 27.7/16241208 [
KiB Swap:  0.0/35639292 [

  PID  PPID   UID USER     RUSER    TTY          TIME+ %CPU %MEM S COMMAND
20037 15584  1000 luca     luca     pts/1      1:17.54 99.9  0.0 R 00_where_I_am_s
20036 15584  1000 luca     luca     pts/1      1:14.71 99.9  0.0 R 00_where_I_am_s
 3271  3240  1000 luca     luca     ?          5:24.05  5.3  0.7 S kwin_x11
 2660  2656     0 root     root     tty1      11:13.78  4.3  1.0 S Xorg
 4181  3223  1000 luca     luca     ?          4:05.04  3.9  1.8 S Wavebox
 3279     1  1000 luca     luca     ?          3:20.64  3.0  3.0 S plasmashell
```

Threads affinity – omp functions

The OpenMP standard offers several `omp_` library functions to deal with the affinity.

You can study heir usage in the source files that are in the `day17/examples/` folder

`numa_awareness/01_where_I_am_omp.c`

Setting the affinity	`proc_bind` clause
Get the affinity	`omp_get_proc_bind( )`
Get details on places	`omp_get_num_places( )`
	`omp_get_place_num( )`
	`omp_get_place_num_procs( )`
	`omp_get_place_proc_ids( )`
Display affinity	`omp_display_affinity( )`
	`omp_get_affinity_format(…)`
	`omp_set_affinity_format(…)`
	`omp_capture_affinity(…)`

Memory allocation

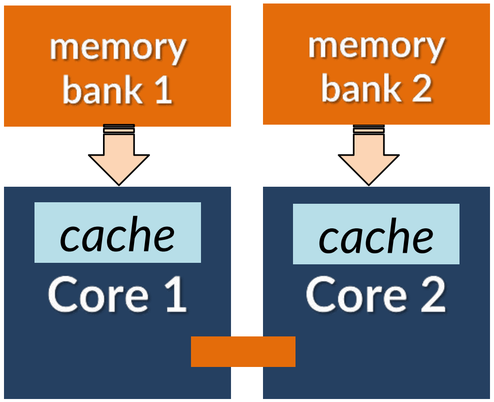
It is possible to control on what physical memory your data will reside by:

1. By carefully touching data
2. By changing default memory allocation with `numactl`
3. By explicit memory migration

*We're **not** gonna cover this*

1. Careful data touching

“touch-first” policy



Suppose that you are operating on a SMP system similar to the one depicted here on the left.

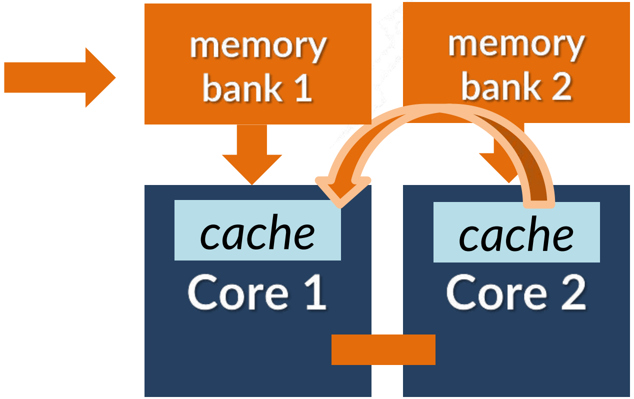
Each socket is physically connected to a RAM bank, and then physically connected to other socket. This way, the memory access is *not uniform*: the bandwidth for a core to access a memory bank not physically connected to it is likely to be significantly smaller than that to access the closest bank.

The matter is: who "owns" the data?

```
double *a = (double*)calloc( N, sizeof(double);

for ( int i = 0; ii < N; ii++ ) {
   a[i] = initialize(i);

#pragma omp parallel for reduction(+: sum)
for ( int i = 0; i < N; i++ )
      sum += a[i];
```



In this way, *all* the data are physically paged in the memory bank of the core on which the master thread runs; its cache is also warmed-up; the other thread must access the memory bank1 which is not the most suited for the bandwidth

"touch-first" policy

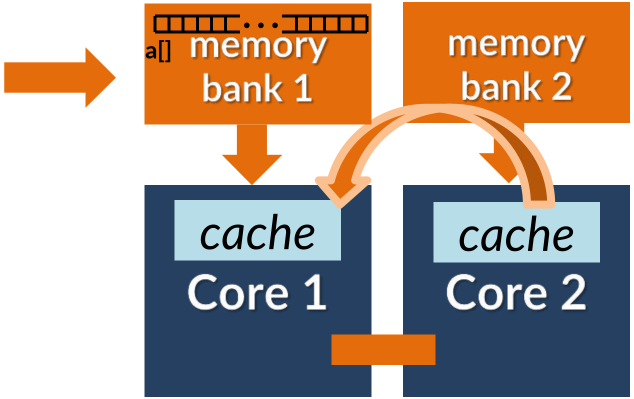
parallel_loops/
01_array_sum.c

The matter is: who "owns" the data?

```
double *a = (double*)calloc( N, sizeof(double);

for ( int i = 0; ii < N; ii++ ) {
   a[i] = initialize(i);

#pragma omp parallel for reduction(+: sum)
for ( int i = 0; i < N; i++ )
      sum += a[i];
```



In this way, the cache of the thread that initialize (first touch) the data is warmed-up **and the data are allocated in the memory connected to it.**

"touch-first" policy

In the "touch-first" policy, the data pages are allocated in the physical memory that is the closest to the physical core which is running the thread that access the data first.
If a single thread is initializing all the data, then all the data will reside in its memory and the number of remote accesses will be maximized.

"touch-by-all" policy

The matter is: who "owns" the data?

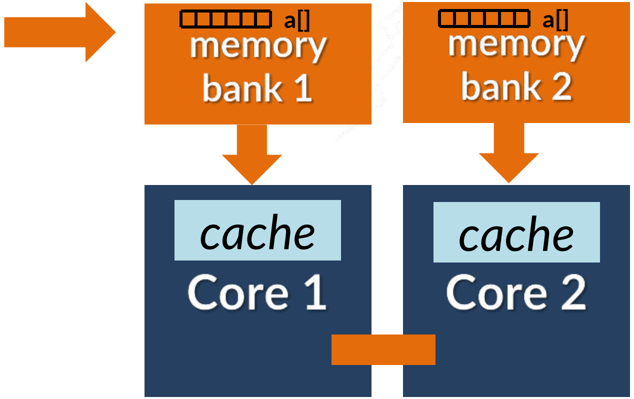
parallel_loops/06_touch_by_all.c

why did I change from `calloc` to `malloc`?

```
double *a = (double*)malloc(N*sizeof(double));

#pragma omp parallel for
for ( int i = 0; ii < N; ii++ ) {
   a[i] = initialize(i);

#pragma omp parallel for reduction(+: sum)
for ( int i = 0; i < N; i++ )
      sum += a[i];
```



In this way, the cache of each thread is warmed-up with the data it will use afterwards **and the data are allocated into each thread's memory** (the scheduling must be the same!)

"touch-by-all" policy

The matter is: who "owns" the data?

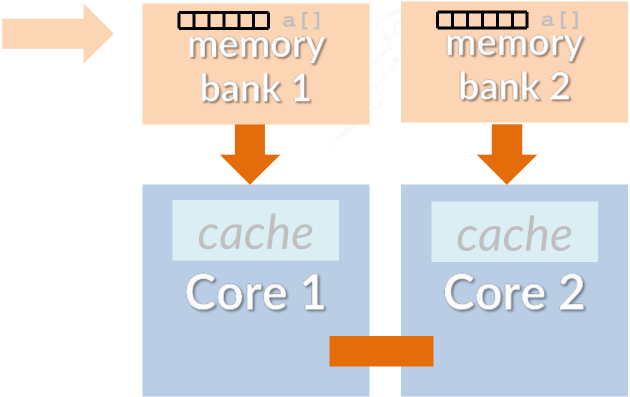
parallel_loops/06_touch_by_all.c

why did I change from calloc to malloc?

```
double *a = (double*)malloc(N*sizeof(double));

#pragma omp parallel for
for ( int i = 0; ii < N; ii++ ) {
   a[i] = initialize(i);

#pragma omp parallel for reduction(+: sum)
for ( int i = 0; i < N; i++ )
      sum += a[i];
```



In this way, the cache of each thread is warmed-up with the data it will use afterwards **and the data are allocated into each thread's memory** (the scheduling must be the same!)

"touch-by-all" policy

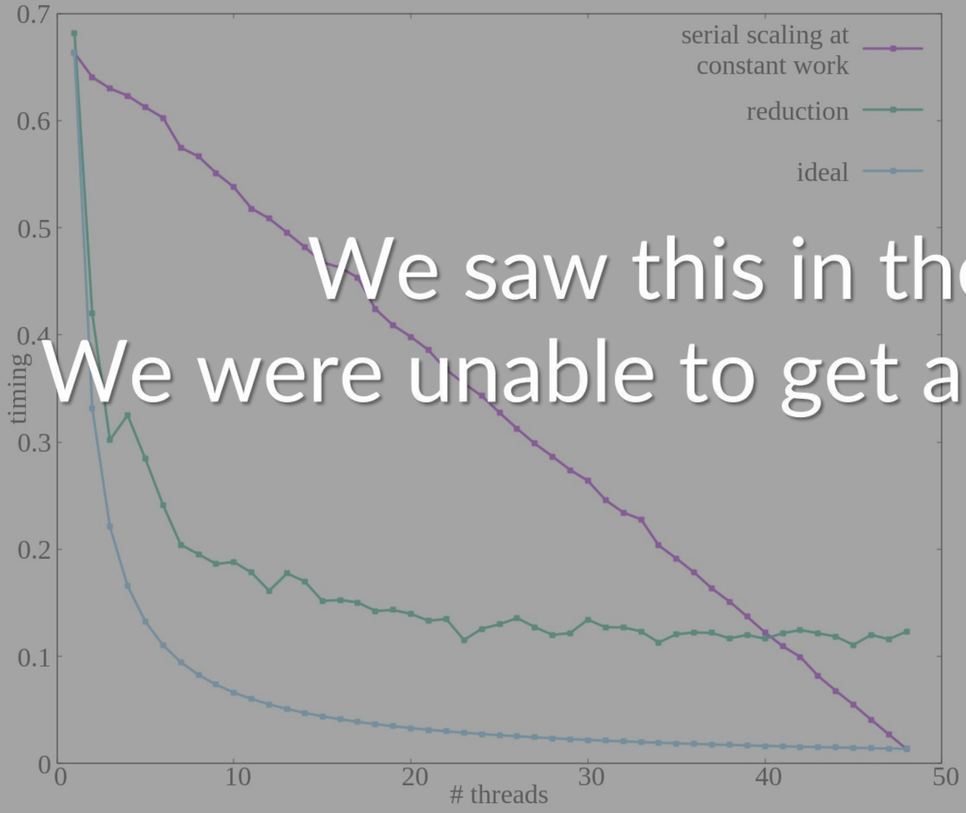
The difference between malloc and calloc

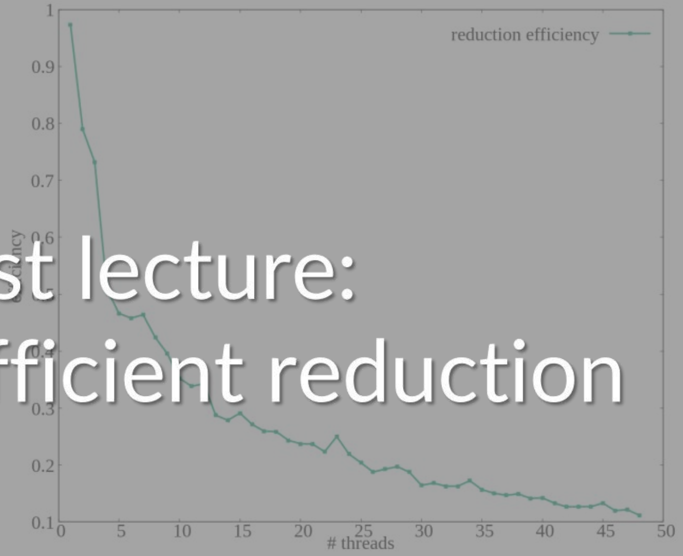
`malloc`	Notifies that the required amount of memory will be used, and the memory occupancy of the process in the heap is grown accordingly. However, the actual mapping of the memory pages into the physical memory does *not* happen until the pages are actually "touched" (i.e. read or written). Moreover, the mapping is done only for the touched pages, not for the entire amount of memory.
`calloc`	As for malloc, but with two fundamental differences: **(1)** the memory is required to be *physically contigous* (that is what the starting "c" means), and hence entirely on the same physical location; **(2)** all the memory is initialized to zero as a way to immediately "touch" it so that it is mapped onto a physical bank as soon as it is required.

“touch-by-all” policy

If each thread “touches” as first the data it will operate on subsequently, those data – by the “touch-first” policy – are allocated in the physical memory that is the closest.
Hence, each thread will have its data placed in the most convenient memory and the remote accesses will be minimized

Solving the reduction / 6

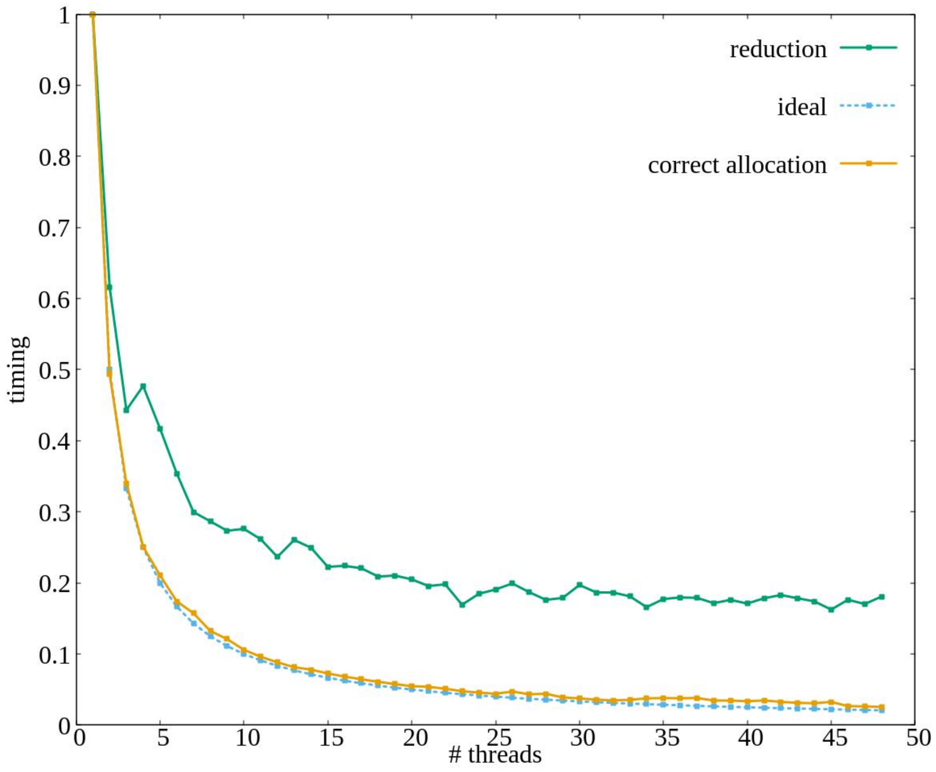


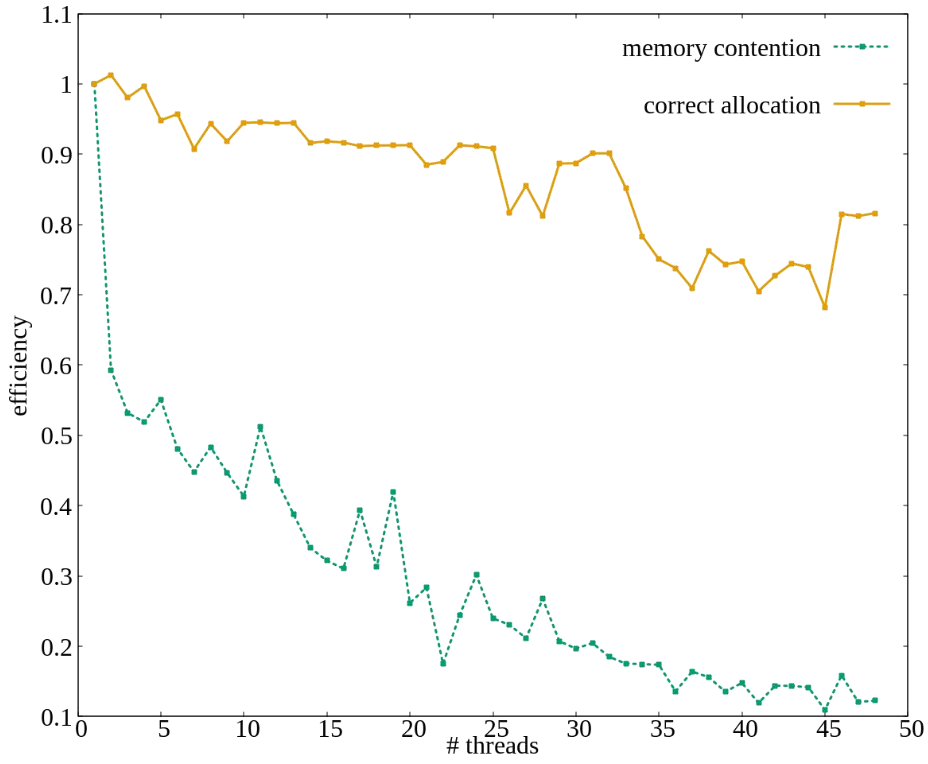


We saw this in the last lecture:
We were unable to get an efficient reduction

It seems that, after all, our reduction efficiency is very poor. Although one could conclude that OpenMP is somehow a bad affair, hidden in these plots there is a very important issue in multi-threading that we inquire in the next lectures.

“touch-by-all” policy





Discover your topology

Lots of tools are usually available on HPC platforms.
We'll see the details in the last lectures devoted to special topics

numactl
```
numactl --hardware  a summary of the topology
this may change your general policy for membinding
```

lscpu

lstopo
```
lstopo -s         a summary of the topology
lstopo -v         more verbose details
lstopo --only [core, socket, cache, pu, .. ]
```
check the man page.. :)

hwloc
```
hwloc-info, hwloc-distances, hwloc-ps, hwloc-ls, …
```

likwid
```
likwid-topology [-g]
likwid-pin        <- this lets you pin your threads
```

/proc/cpuinfo

/sys/devices/system/

How to pin from command line

At least two handy tools:

taskset	-a *pid*	set/retrieve the CPU affinity for all the threads of a given PID
	-c <mask>	(hexadecimal) mask for cores (both physical and logical) 0x00000001 is cpu #0 49 is cpu #0,#4 and #5
	--cpu-list <list>	List of cores, may contain ranges 0-4,15-19 is cpu #0 to #4 and #15 to #19 0-12:2 is cpu #0, #2, #4, .. the :2 is the stride
numactl	--cpunodebind *n*	binds the execution to the NUMA nodes *n* (multiple nodes may be specified, see the man page)
	--membind *n*	binds the memory allocation to DRAM associated to NUMA nodes *n* (multiple nodes may be specified, see the man page)

that’s all, have fun