

# Programming OpenMP

## *NUMA*

**Christian Terboven**

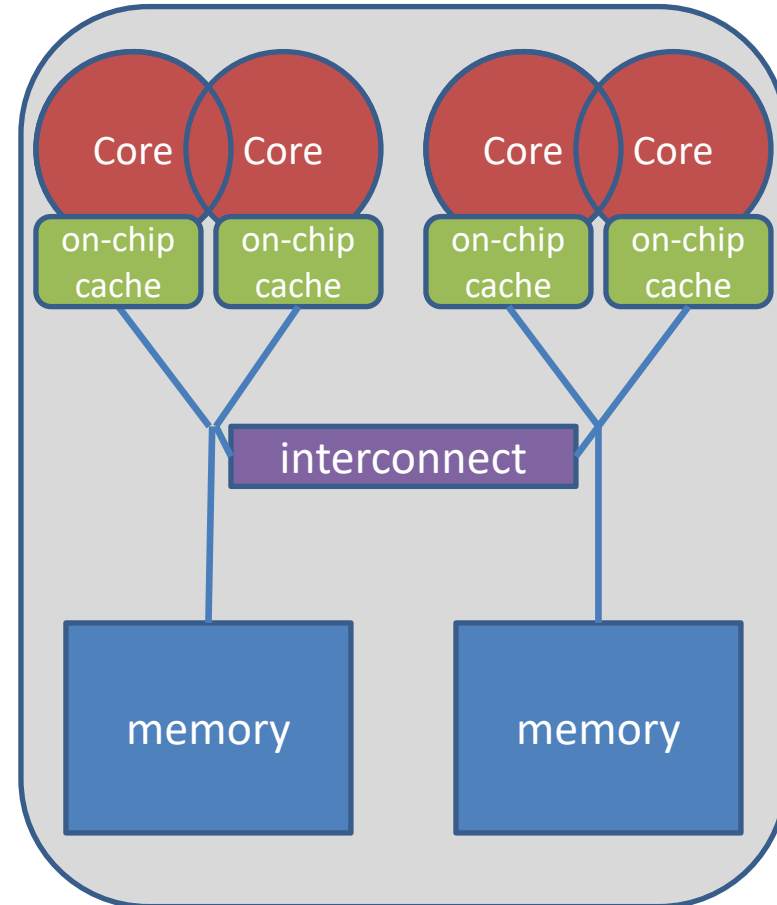


# *OpenMP: Memory Access*

## How To Distribute The Data ?

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

```
for (int i = 0; i < N; i++) {  
    A[i] = 0.0;  
}
```

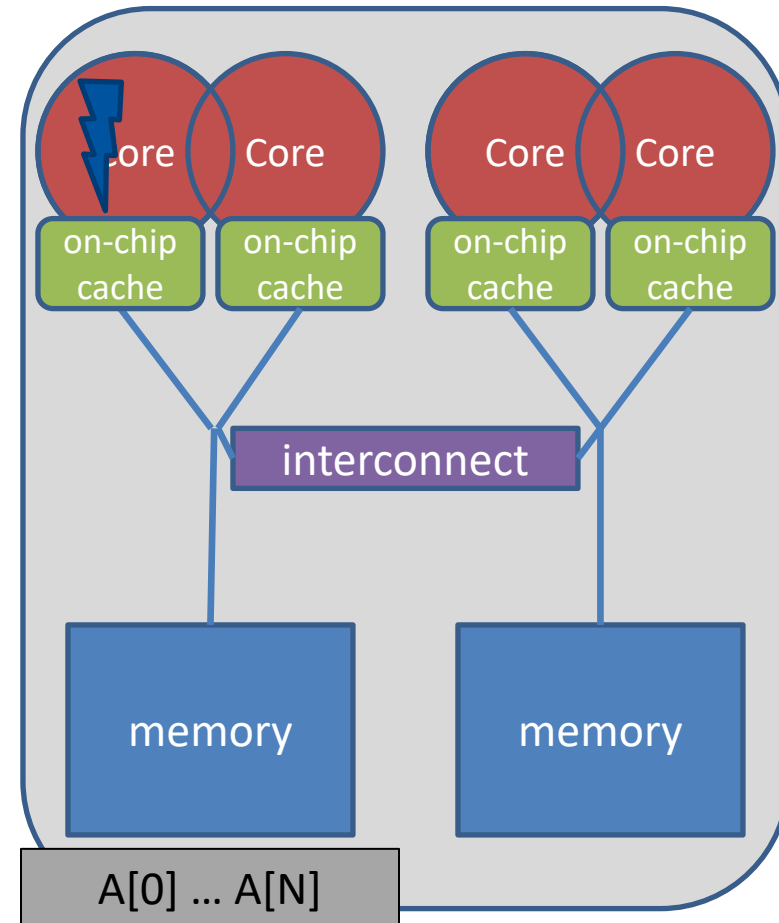


# Non-uniform Memory

- **Serial code: all array elements are allocated in the memory of the NUMA node closest to the core executing the initializer thread (first touch)**

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

```
for (int i = 0; i < N; i++) {  
    A[i] = 0.0;  
}
```



# About Data Distribution

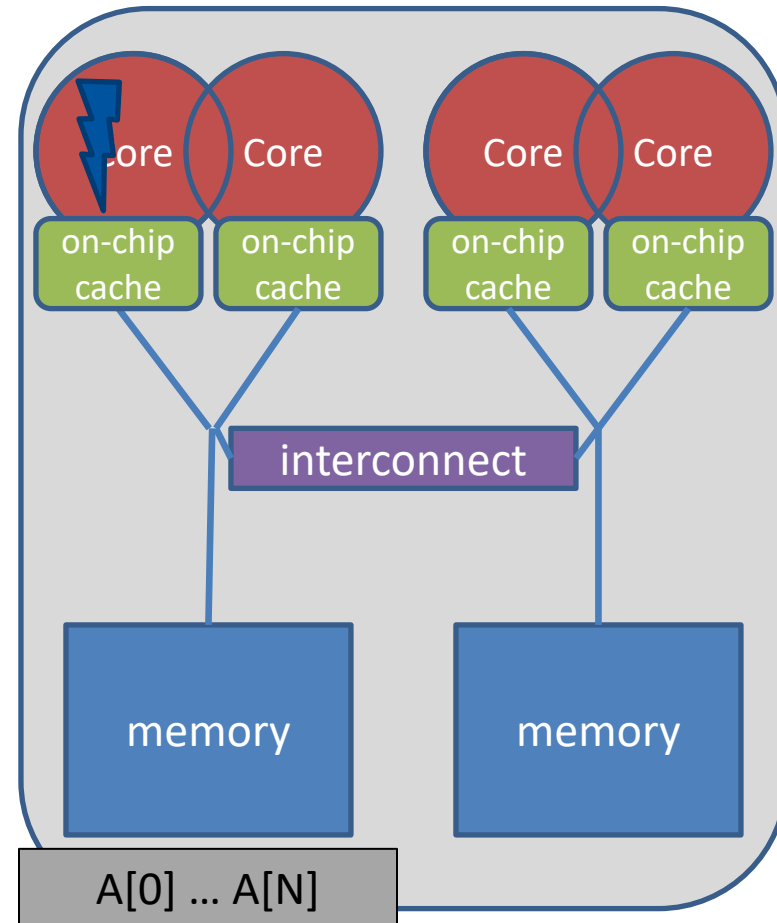
- Important aspect on cc-NUMA systems
  - If not optimal, longer memory access times and hotspots
- Placement comes from the Operating System
  - This is therefore Operating System dependent
- Windows, Linux and Solaris all use the “First Touch” placement policy by default
  - May be possible to override default (check the docs)

# Non-uniform Memory

- Serial code: all array elements are allocated in the memory of the NUMA node closest to the core executing the initializer thread (first touch)

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

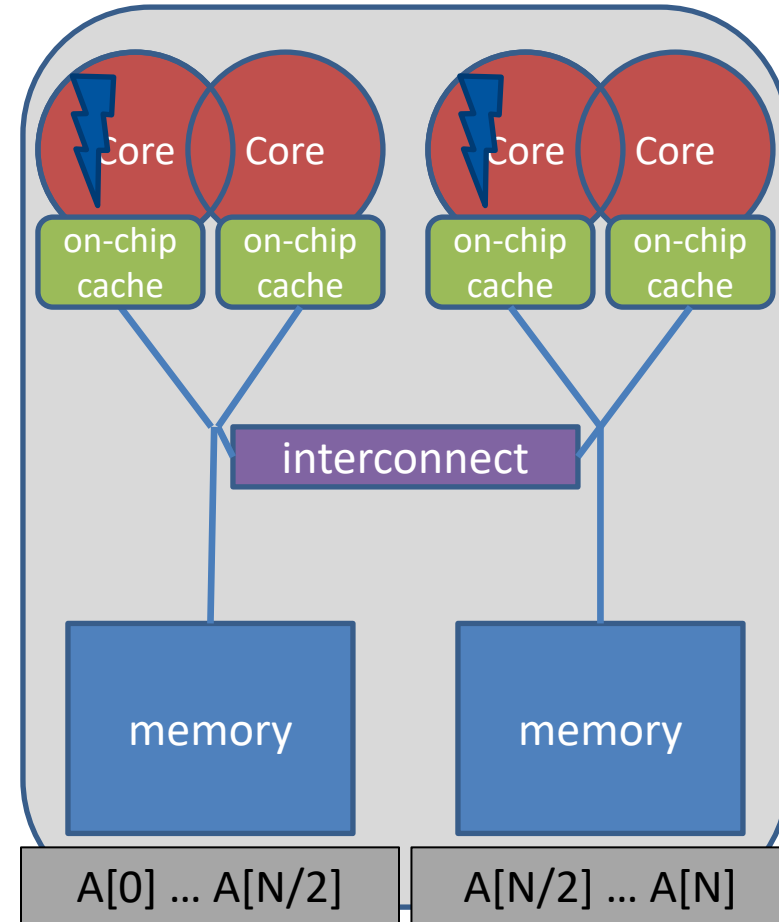
```
for (int i = 0; i < N; i++) {  
    A[i] = 0.0;  
}
```



# First Touch Memory Placement

- **First Touch w/ parallel code: all array elements are allocated in the memory of the NUMA node that contains the core that executes the thread that initializes the partition**

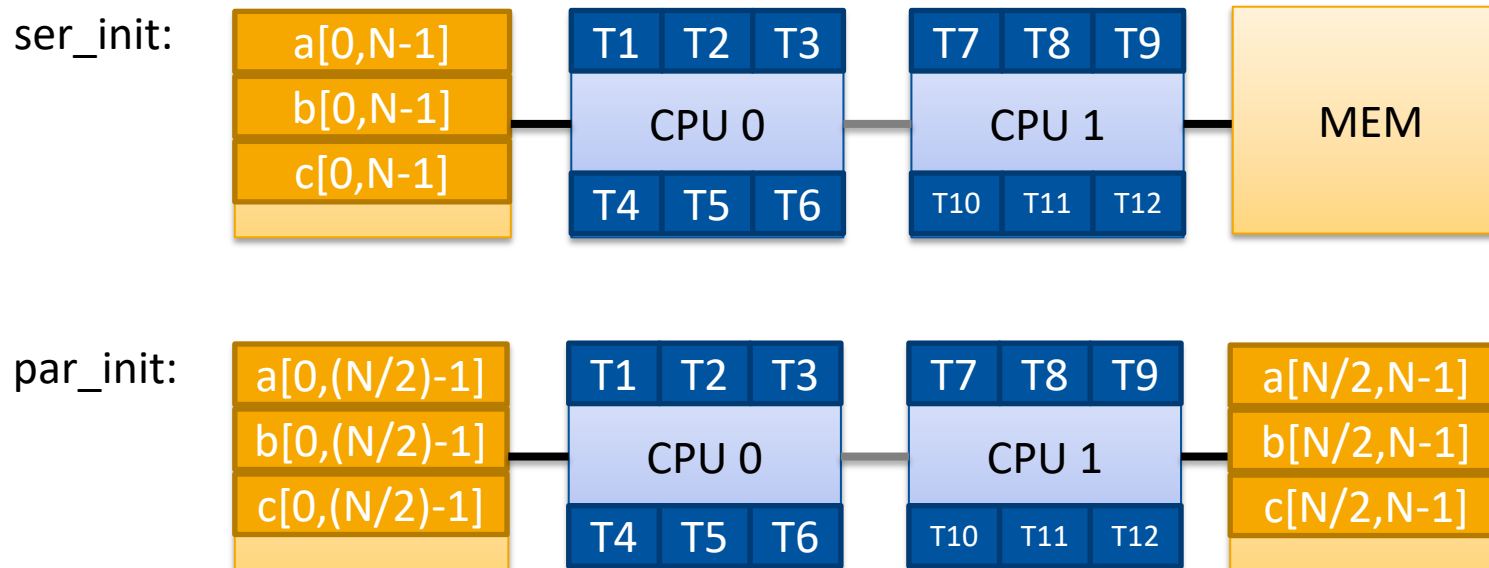
```
double* A;  
A = (double*)  
    malloc(N * sizeof(double));  
  
omp_set_num_threads(2);  
  
#pragma omp parallel for  
for (int i = 0; i < N; i++) {  
    A[i] = 0.0;  
}
```



# Serial vs. Parallel Initialization

- Stream example on 2 socket system with Xeon X5675 processors, 12 OpenMP threads:

	copy	scale	add	triad
ser_init	18.8 GB/s	18.5 GB/s	18.1 GB/s	18.2 GB/s
par_init	41.3 GB/s	39.3 GB/s	40.3 GB/s	40.4 GB/s





# Get Info on the System Topology

- Before you design a strategy for thread binding, you should have a basic understanding of the system topology. Please use one of the following options on a target machine:
  - Intel MPI's `cpuinfo` tool
    - `cpuinfo`
    - Delivers information about the number of sockets (= packages) and the mapping of processor ids to cpu cores that the OS uses.
  - `hwloc`'s `hwloc-ls` tool
    - `hwloc-ls`
    - Displays a graphical representation of the system topology, separated into NUMA nodes, along with the mapping of processor ids to cpu cores that the OS uses and additional info on caches.

# Decide for Binding Strategy

- Selecting the „right“ binding strategy depends not only on the topology, but also on application characteristics.
  - Putting threads far apart, i.e., on different sockets
    - May improve aggregated memory bandwidth available to application
    - May improve the combined cache size available to your application
    - May decrease performance of synchronization constructs
  - Putting threads close together, i.e., on two adjacent cores that possibly share some caches
    - May improve performance of synchronization constructs
    - May decrease the available memory bandwidth and cache size

# Places + Binding Policies (1/2)

## ■ Define OpenMP Places

- set of OpenMP threads running on one or more processors
- can be defined by the user, i.e. `OMP_PLACES=cores`

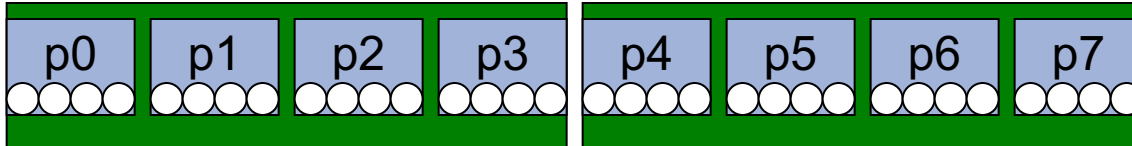
## ■ Define a set of OpenMP Thread Affinity Policies

- SPREAD: spread OpenMP threads evenly among the places, partition the place list
- CLOSE: pack OpenMP threads near master thread
- MASTER: collocate OpenMP thread with master thread

## ■ Goals

- user has a way to specify where to execute OpenMP threads
- locality between OpenMP threads / less false sharing / memory bandwidth

- Assume the following machine:



→ 2 sockets, 4 cores per socket, 4 hyper-threads per core

- Abstract names for OMP\_PLACES:

→ threads: Each place corresponds to a single hardware thread on the target machine.

→ cores: Each place corresponds to a single core (having one or more hardware threads) on the target machine.

→ sockets: Each place corresponds to a single socket (consisting of one or more cores) on the target machine.

→ ll\_caches: Each place corresponds to a set of cores that share the last level cache.

→ numa\_domains: Each place corresponds to a set of cores for which their closest memory is: the same memory; and at a similar distance from the cores.

# Places + Binding Policies (2/2)

## ■ Example's Objective:

→ separate cores for outer loop and near cores for inner loop

## ■ Outer Parallel Region: `proc_bind(spread) num_threads(4)`

## Inner Parallel Region: `proc_bind(close) num_threads(4)`

→ spread creates partition, compact binds threads within respective partition

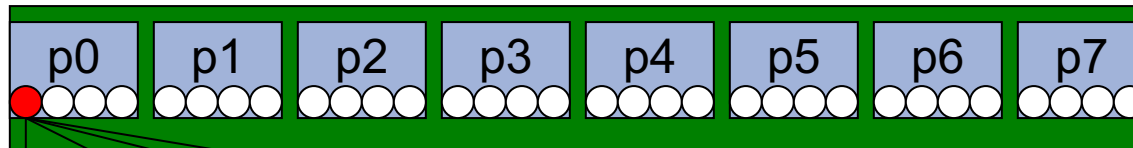
```
OMP_PLACES=(0,1,2,3), (4,5,6,7), ... = (0-3):8:4 = cores
```

```
#pragma omp parallel proc_bind(spread) num_threads(4)
```

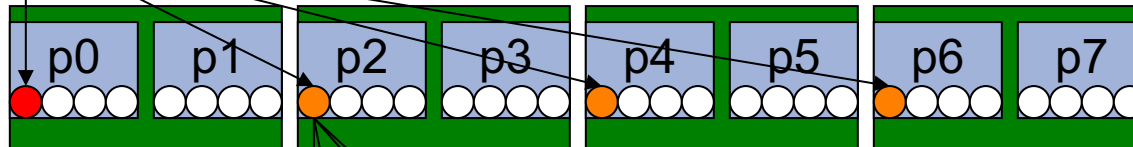
```
#pragma omp parallel proc_bind(close) num_threads(4)
```

## ■ Example

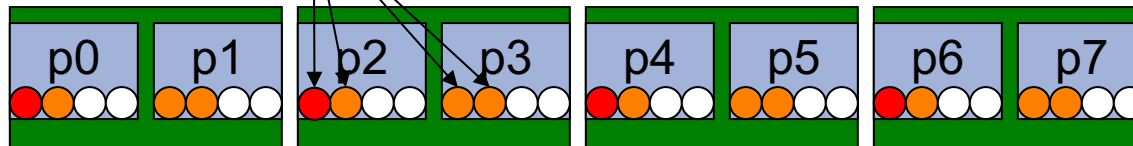
→ initial



→ spread 4

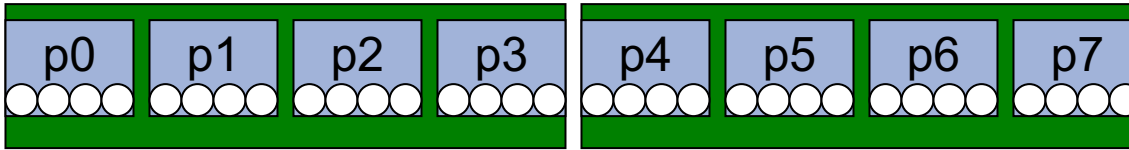


→ close 4



# More Examples (1/3)

- Assume the following machine:



→ 2 sockets, 4 cores per socket, 4 hyper-threads per core

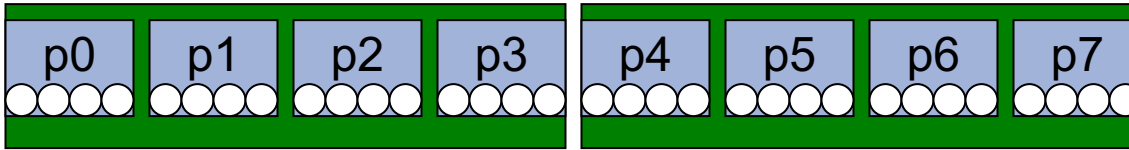
- Parallel Region with two threads, one per socket

→ `OMP_PLACES=sockets`

→ `#pragma omp parallel num_threads(2) proc_bind(spread)`

# More Examples (2/3)

- Assume the following machine:



- Parallel Region with four threads, one per core, but only on the first socket

→ `OMP_PLACES=cores`

→ `#pragma omp parallel num_threads(4) proc_bind(close)`

# More Examples (3/3)

- Spread a nested loop first across two sockets, then among the cores within each socket, only one thread per core

→ `OMP_PLACES=cores`

→ `#pragma omp parallel num_threads(2) proc_bind(spread)`

→ `#pragma omp parallel num_threads(4) proc_bind(close)`



# *Working with OpenMP Places*

# Places API (1/2)

- 1: Query information about binding and a single place of all places with ids 0 ... `omp_get_num_places()`:
- `omp_proc_bind_t omp_get_proc_bind()`: returns the thread affinity policy (`omp_proc_bind_false`, `true`, `master`, ...)
- `int omp_get_num_places()`: returns the number of places
- `int omp_get_place_num_procs(int place_num)`: returns the number of processors in the given place
- `void omp_get_place_proc_ids(int place_num, int* ids)`: returns the ids of the processors in the given place

# Places API (2/2)

- 2: Query information about the place partition:
- `int omp_get_place_num()`: returns the place number of the place to which the current thread is bound
- `int omp_get_partition_num_places()`: returns the number of places in the current partition
- `void omp_get_partition_place_nums(int* pns)`: returns the list of place numbers corresponding to the places in the current partition

# Places API: Example

- Simple routine printing the processor ids of the place the calling thread is bound to:

```
void print_binding_info() {
    int my_place = omp_get_place_num();
    int place_num_procs = omp_get_place_num_procs(my_place);

    printf("Place consists of %d processors: ", place_num_procs);

    int *place_processors = malloc(sizeof(int) * place_num_procs);
    omp_get_place_proc_ids(my_place, place_processors)

    for (int i = 0; i < place_num_procs - 1; i++) {
        printf("%d ", place_processors[i]);
    }
    printf("\n");

    free(place_processors);
}
```

# OpenMP 5.0 way to do this

## ■ Set `OMP_DISPLAY_AFFINITY=TRUE`

→ Instructs the runtime to display formatted affinity information

→ Example output for two threads on two physical cores:

```
nesting_level= 1,  thread_num= 0,  thread_affinity= 0,1
nesting_level= 1,  thread_num= 1,  thread_affinity= 2,3
```

→ Output can be formatted with `OMP_AFFINITY_FORMAT` env var or corresponding routine

→ Formatted affinity information can be printed with

```
omp_display_affinity(const char* format)
```

# Affinity format specification

t	omp_get_team_num()	a	omp_get_ancestor_thread_num() at level-1
T	omp_get_num_teams()	H	hostname
L	omp_get_level()	P	process identifier
n	omp_get_thread_num()	i	native thread identifier
N	omp_get_num_threads()	A	thread affinity: list of processors (cores)

## ■ Example:

```
OMP_AFFINITY_FORMAT="Affinity: %0.3L %.8n %.15{A} %.12H"
```

## → Possible output:

```
Affinity: 001          0          0-1,16-17          host003
Affinity: 001          1          2-3,18-19          host003
```

# *A first summary*

# A first summary

- Everything under control?
- In principle Yes, but only if
  - threads can be bound explicitly,
  - data can be placed well by first-touch, or can be migrated,
  - you focus on a specific platform (= OS + arch) → no portability
  
- What if the data access pattern changes over time?
  
- What if you use more than one level of parallelism?



- **First Touch:** Modern operating systems (i.e., Linux  $\geq$  2.4) decide for a physical location of a memory page during the first page fault, when the page is first „touched“, and put it close to the CPU causing the page fault.
- **Explicit Migration:** Selected regions of memory (pages) are moved from one NUMA node to another via explicit OS syscall.
- **Automatic Migration:** Limited support in current Linux systems.
  - Not made for HPC and disabled on most HPC systems.

# User Control of Memory Affinity

## ■ Explicit NUMA-aware memory allocation:

- By carefully touching data by the thread which later uses it
- By changing the default memory allocation strategy
  - Linux: `numactl` command
  - Windows: `VirtualAllocExNuma()` (limited functionality)
- By explicit migration of memory pages
  - Linux: `move_pages()`
  - Windows: no option

## ■ Example: using `numactl` to distribute pages round-robin:

- `numactl -interleave=all ./a.out`