

# Efficient parallel programming on ccNUMA nodes

Performance characteristics of ccNUMA nodes

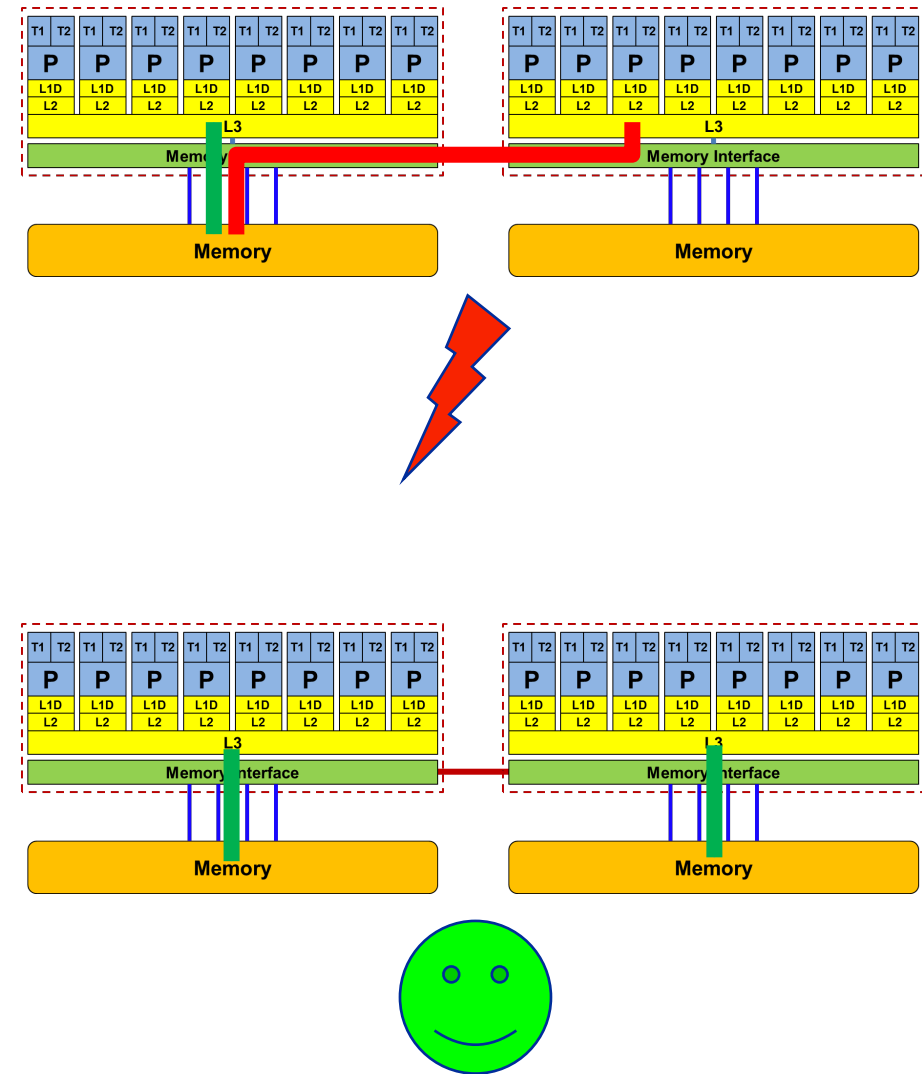
First touch placement policy



# ccNUMA – The “other affinity”

- ccNUMA:
  - Whole memory is **transparently accessible** by all processors
  - but **physically distributed** across multiple locality domains (LDs)
  - with **varying bandwidth and latency**
  - and **potential contention** (shared memory paths)
- How do we make sure that memory access is always as "local" and "distributed" as possible?

**Note:** Page placement is implemented in units of OS pages (often 4 KiB, possibly more)

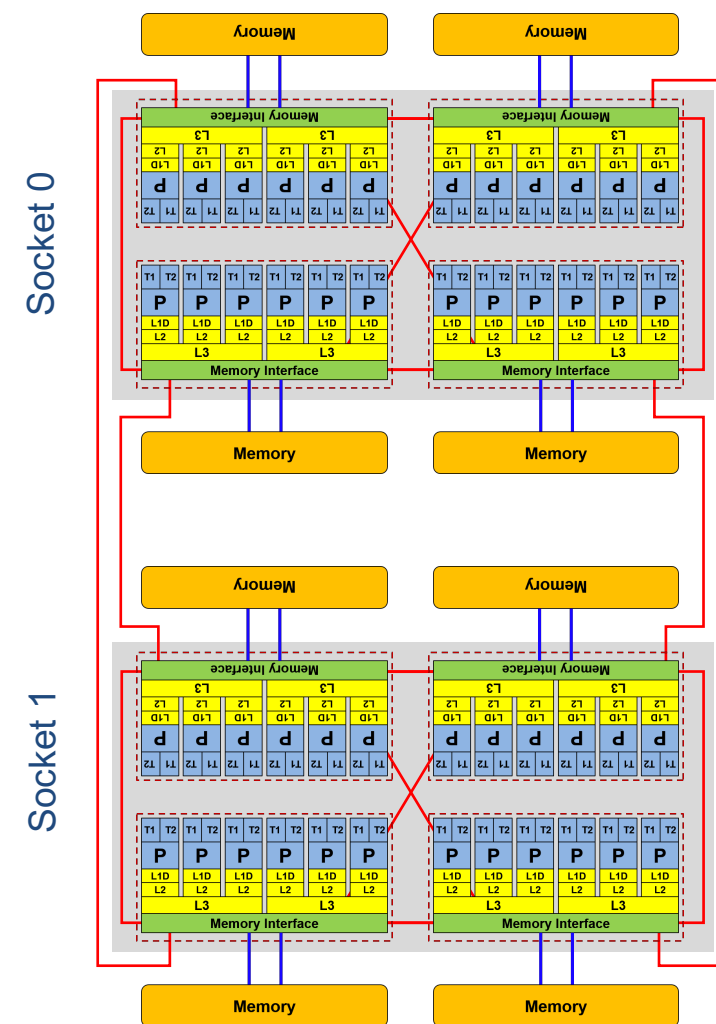


# How much does nonlocal access cost?

Example: AMD “Naples” 2-socket system  
(8 chips, 2 sockets, 48 cores):

*STREAM Triad bandwidth measurements [Gbyte/s]*

CPU node	0	1	2	3	4	5	6	7
MEM node 0	32.4	21.4	21.8	21.9	10.6	10.6	10.7	10.8
1	21.5	32.4	21.9	21.9	10.6	10.5	10.7	10.6
2	21.8	21.9	32.4	21.5	10.6	10.6	10.8	10.7
3	21.9	21.9	21.5	32.4	10.6	10.6	10.6	10.7
4	10.6	10.7	10.6	10.6	32.4	21.4	21.9	21.9
5	10.6	10.6	10.6	10.6	21.4	32.4	21.9	21.9
6	10.6	10.7	10.6	10.6	21.9	21.9	32.3	21.4
7	10.7	10.6	10.6	10.6	21.9	21.9	21.4	32.5



## numactl as a simple ccNUMA locality tool : *How do we enforce some locality of access?*

- **numactl** can influence the way a binary maps its memory pages:

```
numactl --membind=<nodes> a.out      # map pages only on <nodes>
      --preferred=<node> a.out      # map pages on <node>
                                      # and others if <node> is full
      --interleave=<nodes> a.out     # map pages round robin across
                                      # all <nodes>
```

- Examples:

```
for m in `seq 0 7`; do
  for c in `seq 0 7`; do
    env OMP_NUM_THREADS=6 \
      numactl --membind=$m likwid-pin -c M${c}:0-5 ./stream
  done
done
```

ccNUMA map scan  
for Naples system

```
numactl --interleave=0-7 likwid-pin -c E:N:8:1:12 ./stream
```

- But what is the default without **numactl**?

# ccNUMA default memory locality

"Golden Rule" of ccNUMA:

**A memory page gets mapped into the local memory of the processor that first touches it!**

(Except if there is not enough local memory available)

- **Caveat:** "to touch" means "to write," not "to allocate"
- Example:

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

```
for(i=0; i<N; i++) // or i+=PAGE_SIZE/sizeof(double)  
    huge[i] = 0.0;
```

Memory not mapped here yet

Mapping takes place here

- It is sufficient to touch a single item to map the entire page

# Coding for ccNUMA data locality

## Simplest case: explicit initialization

```
integer,parameter :: N=10000000
double precision, allocatable :: A(:), B(:)
allocate(A(N),B(N))
```

```
A=0.d0
```

```
!$OMP parallel do
do i = 1, N
  B(i) = function ( A(i) )
end do
!$OMP end parallel do
```



```
integer,parameter :: N=10000000
double precision, allocatable :: A(:), B(:)
allocate(A(N),B(N))
```

```
!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
  A(i)=0.d0
end do
!$OMP end do
...
!$OMP do schedule(static)
do i = 1, N
  B(i) = function ( A(i) )
end do
!$OMP end do
!$OMP end parallel
```

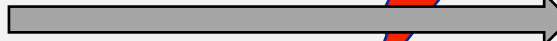


# Coding for ccNUMA data locality

Sometimes initialization is not so obvious: I/O cannot be easily parallelized, so “localize” arrays before I/O

```
integer,parameter :: N=10000000
double precision, allocatable :: A(:), B(:)
allocate (A(N),B(N))
```

READ(1000) A



```
!$OMP parallel do
do i = 1, N
  B(i) = function ( A(i) )
end do
!$OMP end parallel do
```

```
integer,parameter :: N=10000000
double precision, allocatable :: A(:), B(:)
allocate (A(N),B(N))
```

```
!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
  A(i)=0.d0
end do
!$OMP end do
!$OMP single
READ(1000) A
!$OMP end single
!$OMP do schedule(static)
do i = 1, N
  B(i) = function ( A(i) )
end do
!$OMP end do
!$OMP end parallel
```



# Coding for Data Locality

- Required condition: OpenMP **loop schedule** of initialization must be the same as in all computational loops
  - Only choice: **static!** Specify **explicitly** on all NUMA-sensitive loops, just to be sure...
  - Imposes some constraints on possible optimizations (e.g. load balancing)
  - Presupposes that all **worksharing loops** with the **same loop length** have the **same thread-chunk mapping**
  - If **dynamic scheduling/tasking** is unavoidable, the problem cannot be solved completely if a team of threads spans more than one LD
    - Static parallel first touch is still a good idea
    - OpenMP 5.0 will have rudimentary memory affinity functionality
- How about **global objects**?
  - Initialized before `main()` is called
  - If communication vs. computation is favorable, might consider **properly placed copies** of global data
- C++: Arrays of objects and `std::vector<>` are by default initialized sequentially
  - **STL allocators** provide an elegant solution



```
template <class T> class NUMA_Allocator {
public:
    T* allocate(size_type numObjects, const void
                *localityHint=0) {
        size_type ofs, len = numObjects * sizeof(T);
        void *m = malloc(len);
        char *p = static_cast<char*>(m);
        int i, pages = len >> PAGE_BITS;
#pragma omp parallel for schedule(static) private(ofs)
        for(i=0; i<pages; ++i) {
            ofs = static_cast<size_t>(i) << PAGE_BITS;
            p[ofs]=0;
        }
        return static_cast<pointer>(m);
    }
    ...
};
```

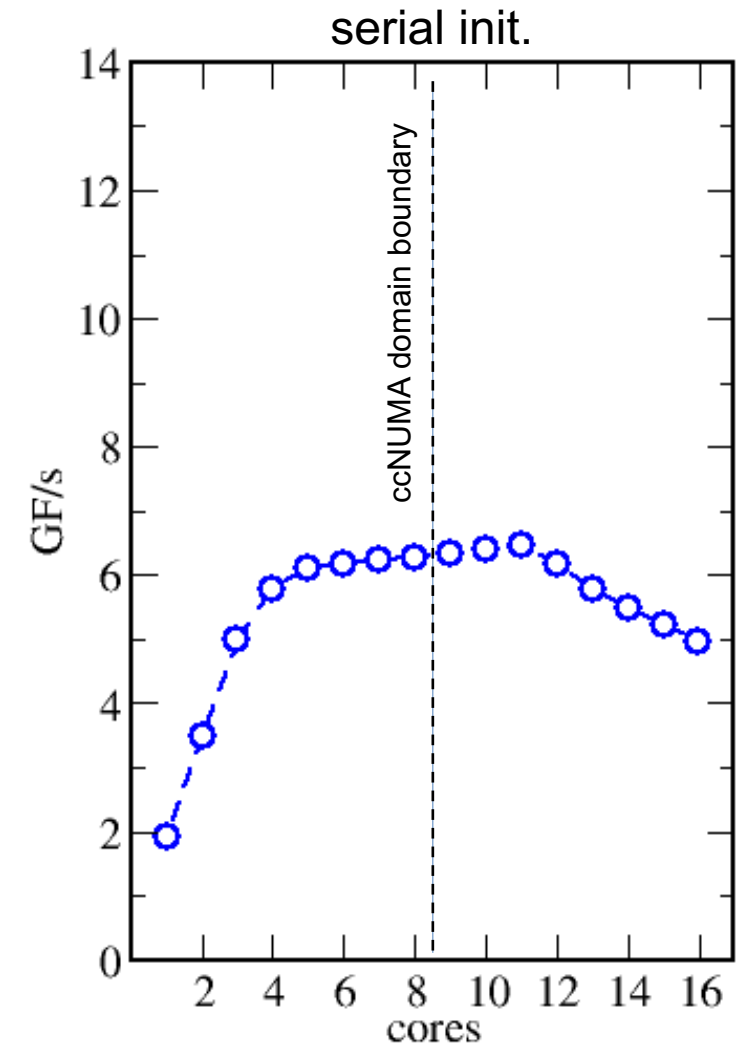
Application:

```
vector<double, NUMA_Allocator<double> > x(10000000);
```

# Diagnosing bad locality

- If your code is cache bound, you might not notice any locality problems
- Otherwise, bad locality **limits scalability** (whenever a ccNUMA node boundary is crossed)
  - **Just an indication, not a proof yet**
- Running with **numactl --interleave** might give you a hint
  - See later
- Consider using **performance counters**
  - **likwid-perfctr** can be used to measure non-local memory accesses
  - Example for Intel dual-socket system (Ivy Bridge, 2x10-core):

```
$ likwid-perfctr -g NUMA -C M0:0-4@M1:0-4 ./a.out
```



# Using performance counters for diagnosis

- Intel Ivy Bridge EP node (running 2x5 threads):  
measure NUMA traffic

```
$ likwid-perfctr -g NUMA -C M0:0-4@M1:0-4 ./a.out
```

- Summary output:

Metric	Sum	Min	Max	Avg
Runtime (RDTSC) [s] STAT	4.050483	0.4050483	0.4050483	0.4050483
Runtime unhalted [s] STAT	3.03537	0.3026072	0.3043367	0.303537
Clock [MHz] STAT	32996.94	3299.692	3299.696	3299.694
CPI STAT	40.3212	3.702072	4.244213	4.03212
Local DRAM data volume [GByte] STAT	7.752933632	0.735579264	0.823551488	0.7752933632
Local DRAM bandwidth [MByte/s] STAT	19140.761	1816.028	2033.218	1914.0761
Remote DRAM data volume [GByte] STAT	9.16628352	0.86682464	0.957811776	0.916628352
Remote DRAM bandwidth [MByte/s] STAT	22630.098	2140.052	2364.685	2263.0098
Memory data volume [GByte] STAT	16.919217152	1.690376128	1.69339104	1.6919217152
Memory bandwidth [MByte/s] STAT	41770.861	4173.27	4180.714	4177.0861

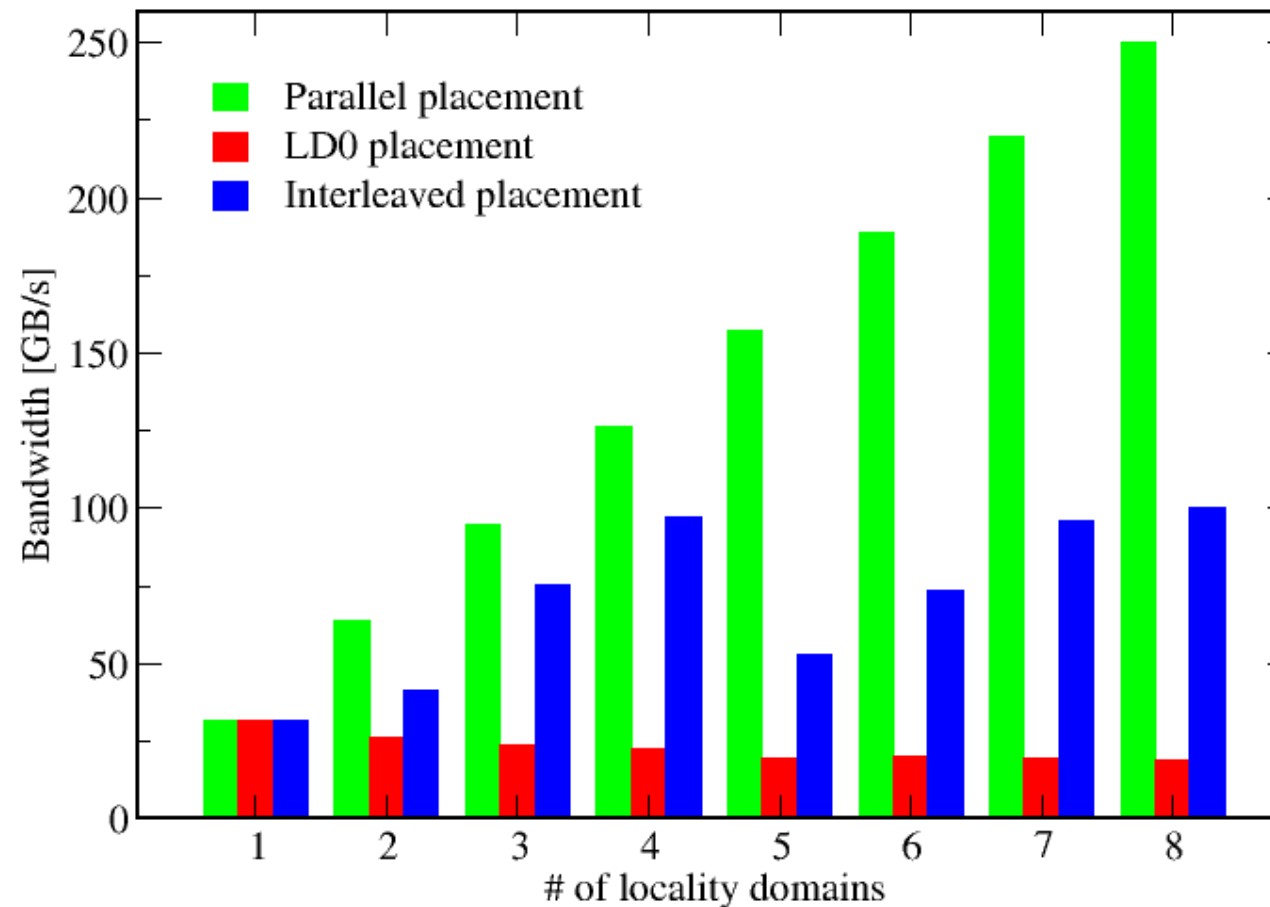
**Caveat:** NUMA metrics vary strongly between CPU models

**On ICX:** UPI groups

About half of the overall memory traffic is caused by the remote domain!

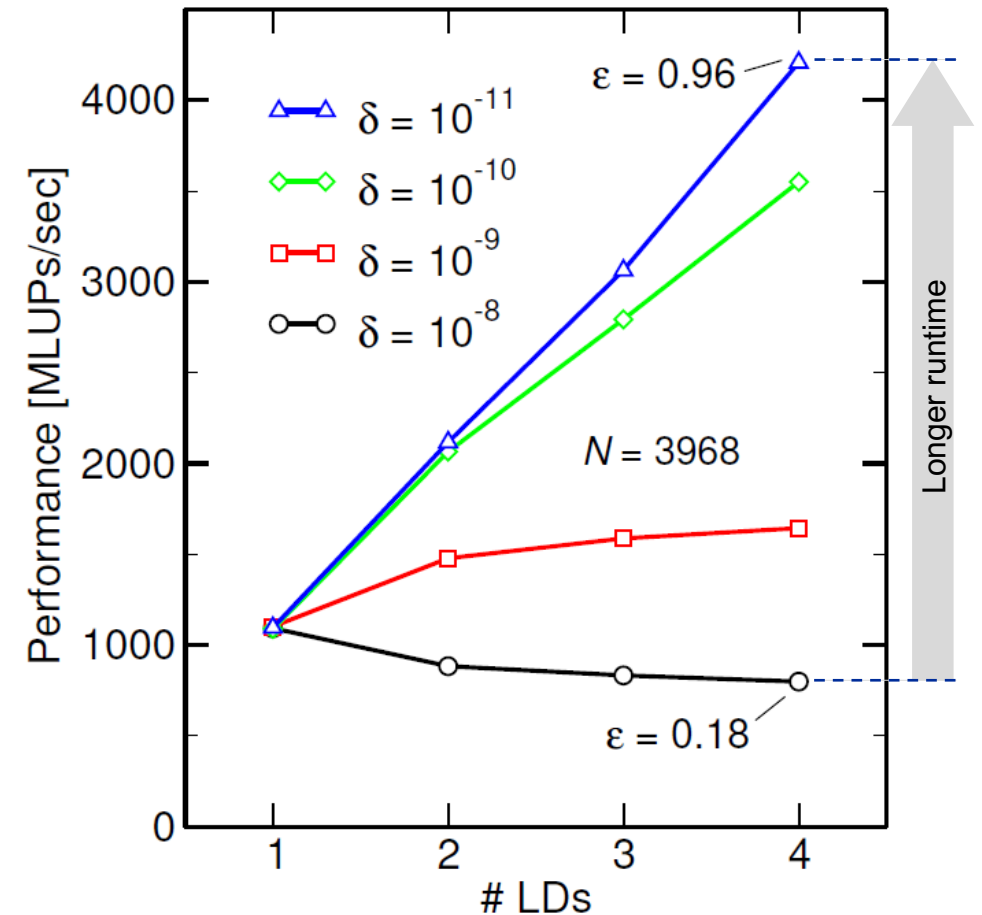
# OpenMP STREAM triad on a dual AMD Epyc 7451 (“Naples”) (6 cores per LD)

1. **Parallel init:** Correct parallel initialization
2. **LD0:** Force data into LD0 via `numactl -m 0`
3. **Interleaved:** `numactl --interleave <LD range>`



# A weird observation

- Experiment: **memory-bound** Jacobi solver with sequential data initialization
  - **No parallel data placement** at all!
  - Expect no scaling across LDs
- Convergence threshold  $\delta$  determines the runtime
  - The smaller  $\delta$ , the longer the run
- Observation
  - **No scaling** across LDs for large  $\delta$  (runtime 0.5 s)
  - **Scaling** gets **better** with **smaller**  $\delta$  up to almost perfect efficiency  $\varepsilon$  (runtime 91 s)
- **Conclusion**
  - Something seems to “**heal**” the **bad access locality** on a time scale of tens of seconds



# Riddle solved: NUMA balancing

- Linux kernel supports **automatic page migration**

```
$ cat /proc/sys/kernel/numa_balancing
0
$ echo 1 > /proc/sys/kernel/numa_balancing # activate
```

- Active on all current Linux distributions, some performance impact for single core execution
- Parameters control aggressiveness

```
$ ll /proc/sys/kernel/numa*
-rw-r--r-- 1 root root 0 Jun 26 09:16 numa_balancing
-rw-r--r-- 1 root root 0 Jun 26 09:16 numa_balancing_scan_delay_ms
-rw-r--r-- 1 root root 0 Jun 26 09:16 numa_balancing_scan_period_max_ms
-rw-r--r-- 1 root root 0 Jun 26 09:16 numa_balancing_scan_period_min_ms
-rw-r--r-- 1 root root 0 Jun 26 09:16 numa_balancing_scan_size_mb
```

- Default behavior is “take it slow”
- Do not rely on it! Parallel first touch is still a good idea!**

# Summary on ccNUMA issues

- Identify the problem
  - Is ccNUMA an issue in your code?
  - Simple test: run with `numactl --interleave`
  - Consider performance counters if available
- Apply first-touch placement in initialization loops
  - Consider loop lengths and static scheduling
  - C++ and global/static objects may require special care
- NUMA balancing is active on many Linux systems today
  - Automatic page migration
  - Slow process, may take many seconds (configurable)
  - Not a silver bullet
  - Still a good idea to do parallel first touch
- If dynamic scheduling cannot be avoided
  - Consider round-robin placement as a quick (but non-ideal) fix
  - OpenMP 5.0 has some data affinity support