Parallel Programming of High-Performance Systems

A collaborative course of NHR@FAU and LRZ Garching

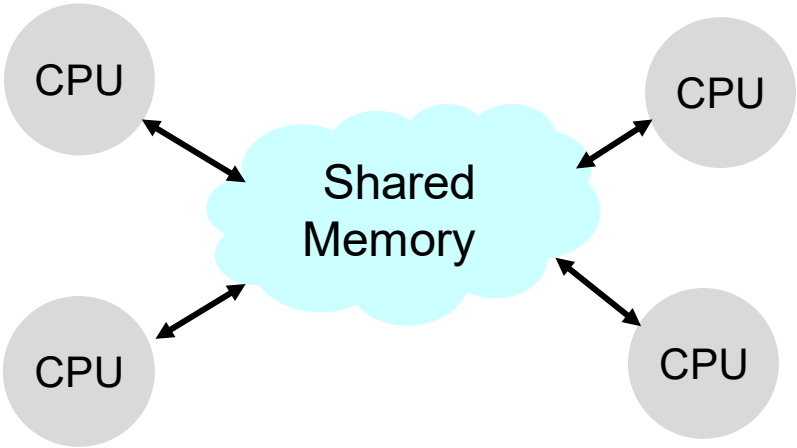
Georg Hager, Volker Weinberg, Ayesha Afzal, Markus Wittmann

Shared-Memory Computer Architecture

High Performance Computing

Shared memory

- Single address space for all processors/cores
- Cache coherent, i.e., changes in one cache will be communicated to all others for consistency



- Two basic variants: UMA and ccNUMA

UMA vs. ccNUMA

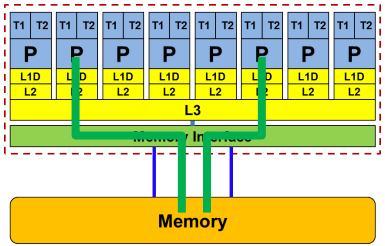
[cache-coherent]
Uniform **M**emory **A**ccess

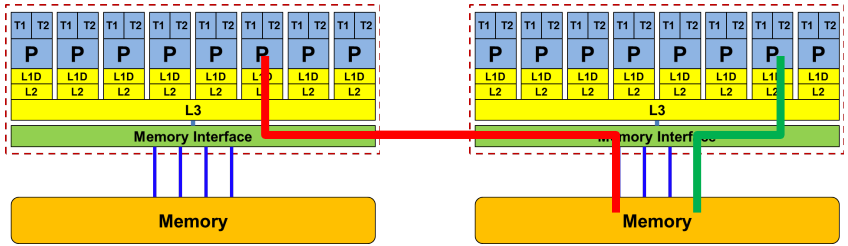
All memory accessible by all cores with same latency and bandwidth

cache-**c**oherent
Non-**U**niform **M**emory **A**ccess

Latency and bandwidth vary depending on mutual position of core and memory

But why???





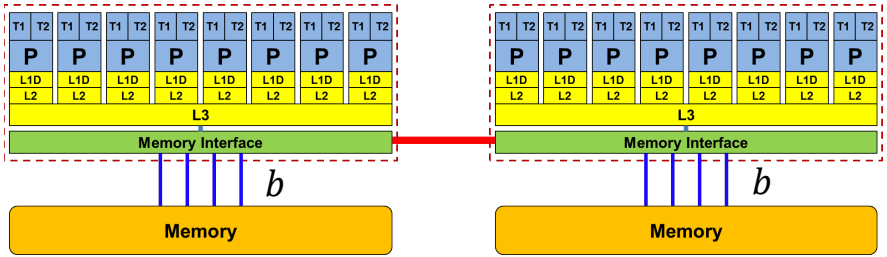
Why ccNUMA?

- Many algorithms rely on high Memory bandwidth:

$$b = \frac{V}{T}$$

V data transferred over memory bus [byte]

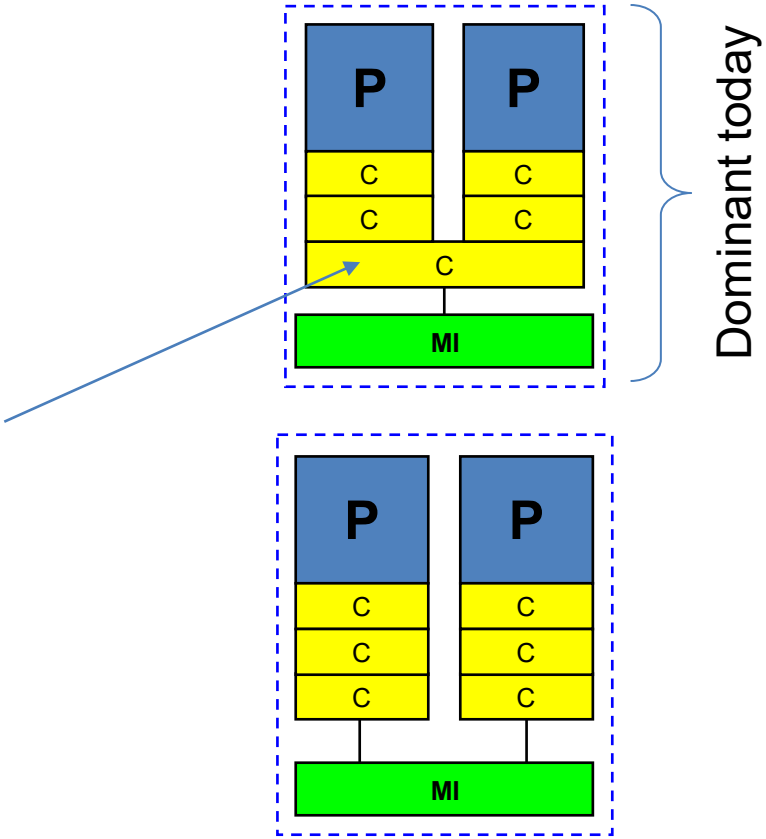
T wallclock time [s]



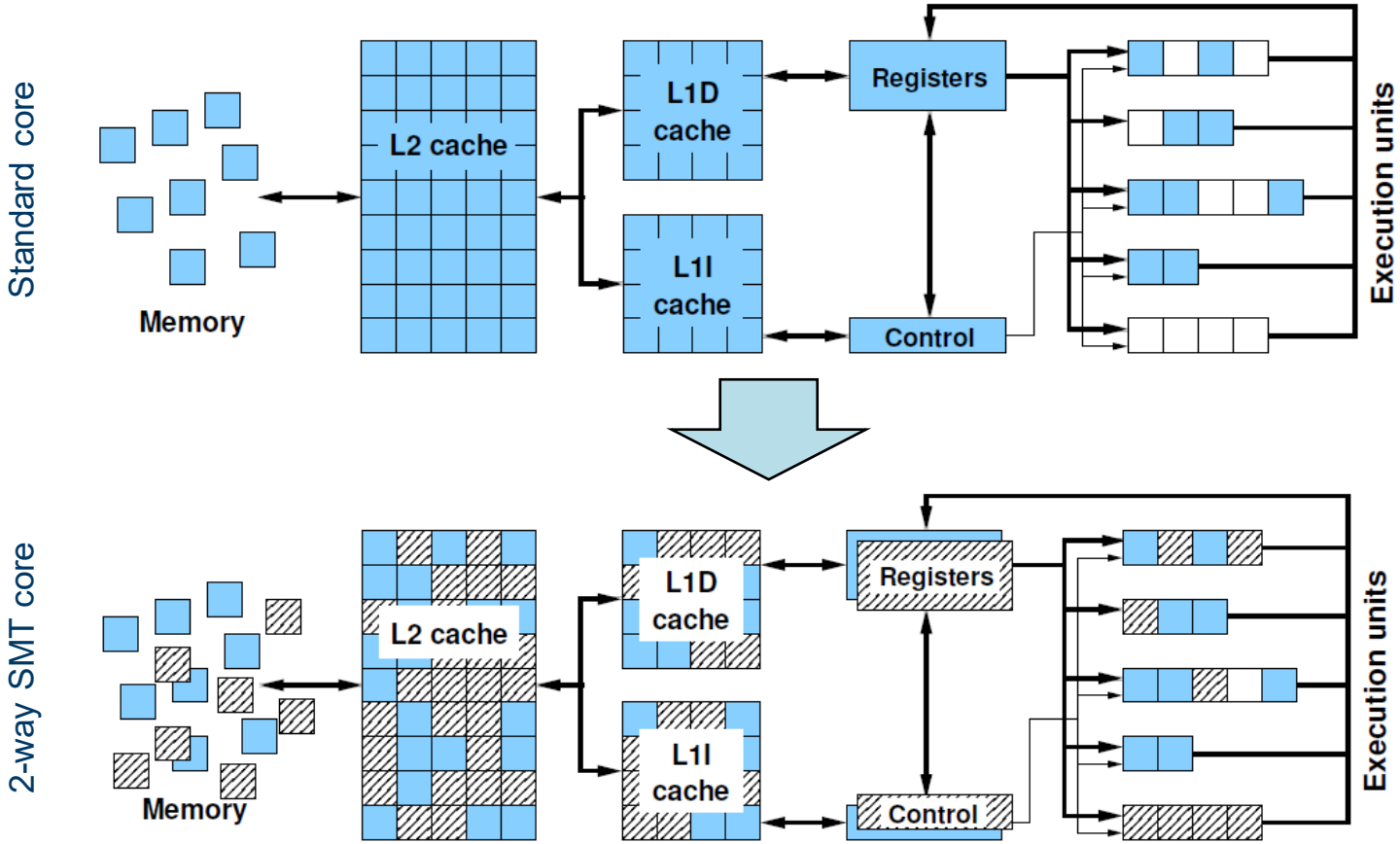
- Advantage: Easier (cheaper) to build multiple domains with smaller bandwidth than one UMA domain with high bandwidth
- Disadvantage: Adds “topology” (non-uniformity in memory access, need to know where my threads are running)

Shared vs. private caches

- Shared cache
  - Fast communication path between cores
    - Can also reduce synchronization overhead
  - Less coherence overhead between cores connected to the same cache
  - More cache for sequential applications
  - Shared bandwidth → potential bottleneck
- Private caches
  - No cache bandwidth bottleneck
  - More overhead for cache coherence
  - Single-threaded workloads leave a lot of cache unused



Simultaneous multi-threading (SMT)

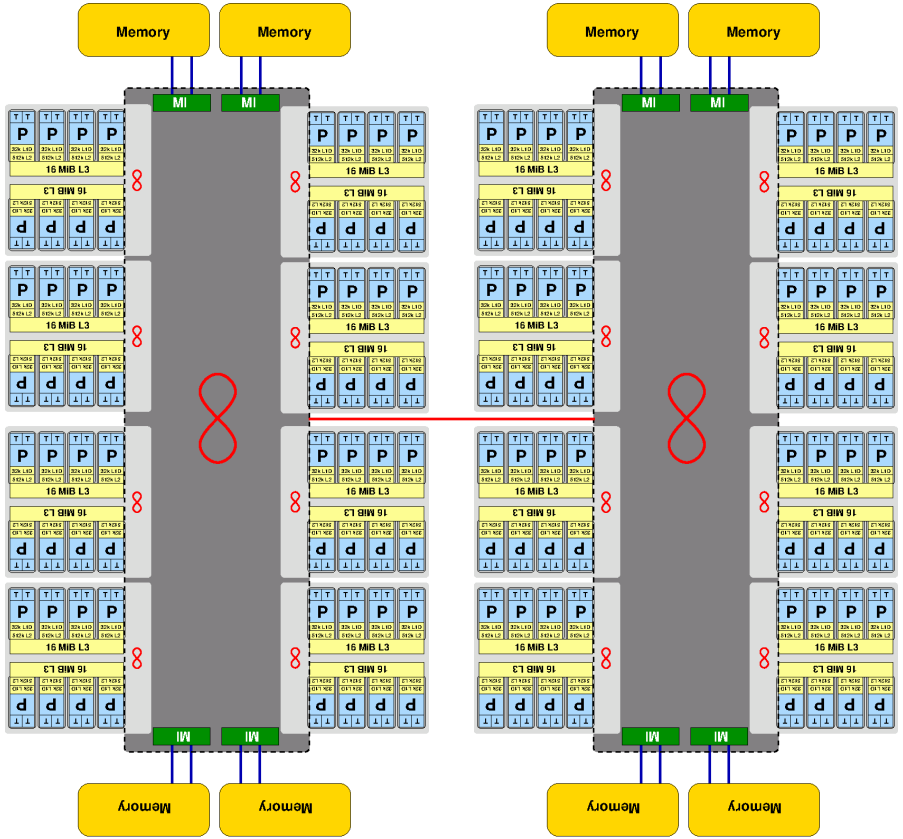


SMT benefits and caveats

- Can provide better throughput if there is parallelism in the code
  - i.e., more instructions executed per second
  - This is not automatic – code must have multiple threads/processes
  - “If in doubt, give it a try!”
- Almost all chip resources are shared among hardware threads
  - Execution units, caches, memory interface
  - Sharing these resources may prevent SMT from improving performance or even give a performance hit
- SMT introduces another layer of topology on top of it all
  - Learn how to ignore it if necessary

A modern dual-socket node

- AMD “Rome” (Zen2) dual-socket system
  - 64 cores per socket (with SMT)
  - 8 cores per die, 8 dies per socket
  - Shared L3 cache for core quadruplets (half dies)
  - AMD “Infinity Fabric” between dies and sockets
- Up to four ccNUMA domains per node
  - Configurable NPS1, NPS2, NPS4
- Two DDR4 memory channels per ccNUMA domain



NHR FAU

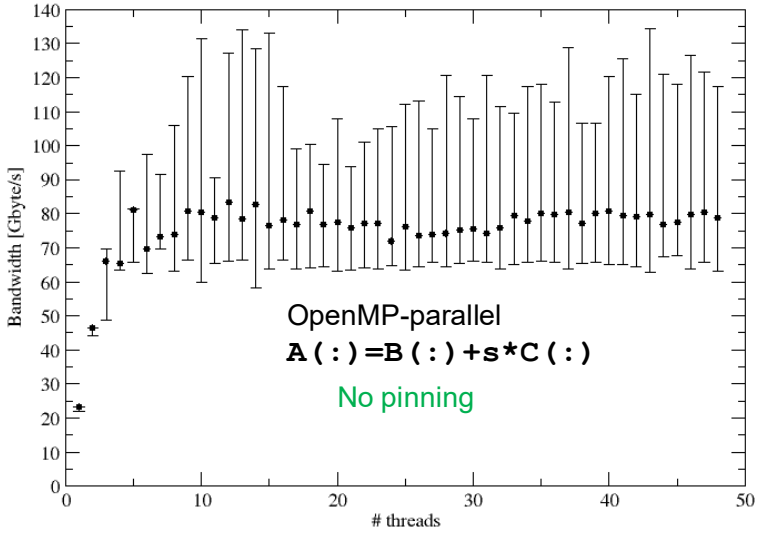
lrz

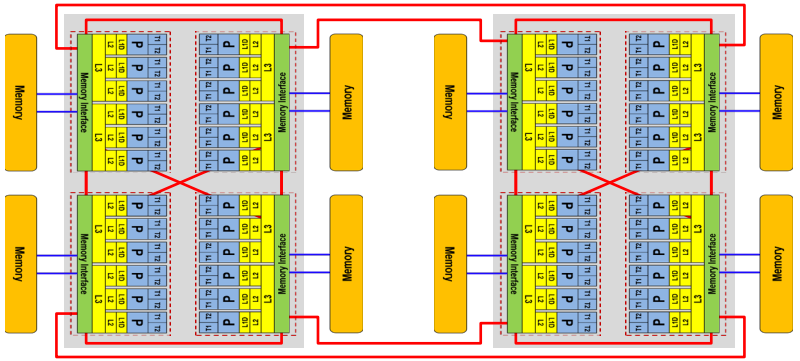
The role of thread/process affinity

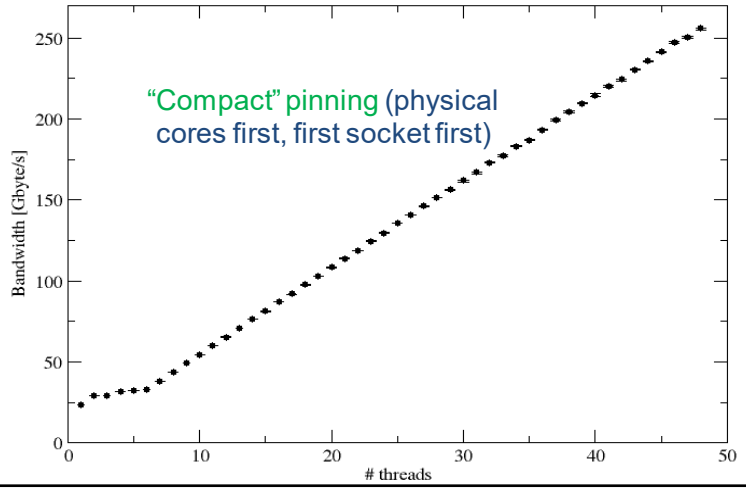
HPC High Performance Computing

STREAM benchmark on 2x24-core AMD Zen “Naples”

Anarchy vs. thread pinning







There are several reasons for caring about affinity:

- Eliminating performance variation
- Making use of architectural features
- Avoiding resource contention

NHR FAU

lrz

Cache coherence

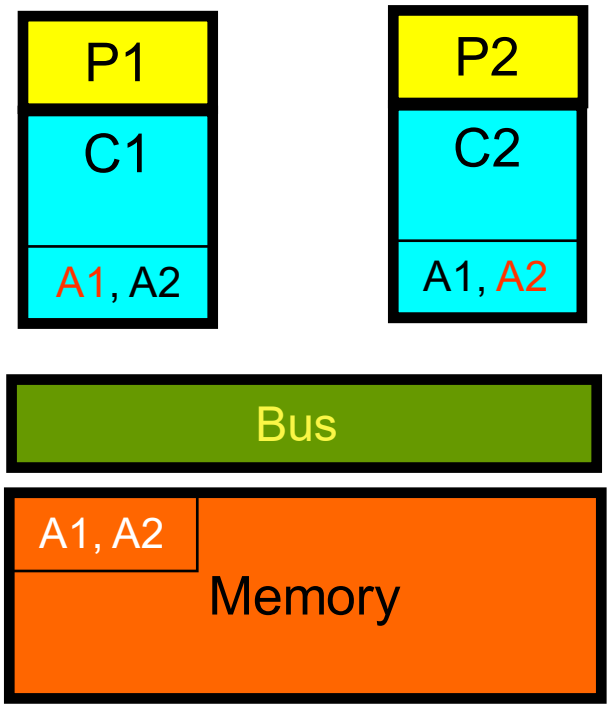
HPC High Performance Computing

Cache coherence in shared-memory computers

- Data in cache is only a copy of data in memory
  - Data is always cached in blocks (“cache lines”) of, e.g., 64 bytes
  - Multiple copies of same data on multiprocessor systems – consistency?
  - Without cache coherence, shared cache lines can become clobbered

- Cache coherence protocol keeps track of cache line (CL) status
  - Simple protocol: MESI
  - Cache line can be
    - Modified
    - Exclusive
    - Shared
    - Invalid

Without cache coherence protocol



P1	P2
`Load A1`	`Load A2`
`Write A1=0`	`Write A2=0`

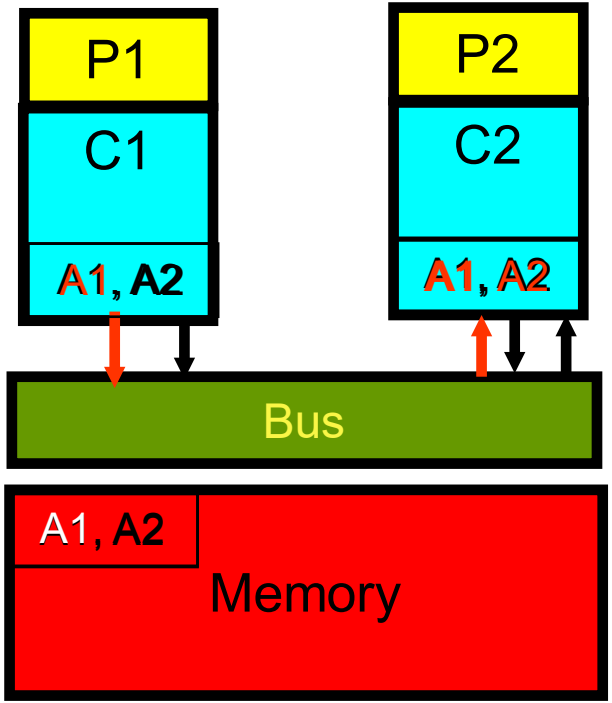
Write-back to memory leads to incoherent data

A1, A2 | A1, A2 | A1, A2

C1 & C2 entry can not be merged to:

A1, A2

With cache coherence protocol



P2

`Load A1`

`Load A2`

`Write A1=0:`

1. Request exclusive access to CL
2. Invalidate CL in C2
3. Modify A1 in C1

`Write A2=0:`

1. Request exclusive CL access
2. CL write back+ Invalidate
3. Load CL to C2
4. Modify A2 in C2

t

C2 is exclusive owner of CL

Cache coherence

- Cache coherence can cause substantial overhead
  - may reduce available bandwidth
  - “False sharing” when multiple cores modify same CL frequently
- Different implementations
  - Snoop: On modifying a CL, a CPU must broadcast its address to the whole system
  - Directory, “snoop filter”: Hardware (“network”) keeps track of which CLs are where and filters coherence traffic
- Directory-based ccNUMA can reduce pain of additional coherence traffic

- Multiple cores should never write frequently to the same cache line (“*false sharing*”)! Very bad performance may ensue.

Summary on shared-memory architecture

- Basic building block of all modern CPU-based clusters: shared-memory "compute node"
- Significant "topology" within the node
  - Simultaneous multi-threading (hyper-threading)
  - Shared/private caches
  - Memory interfaces
  - Sockets ("packages")
- Topology has important performance implications
  - Thread-core affinity (pinning) is decisive!
- Cache coherence mechanisms make programming easier
  - In general, nothing to worry about except when you have to ;-)

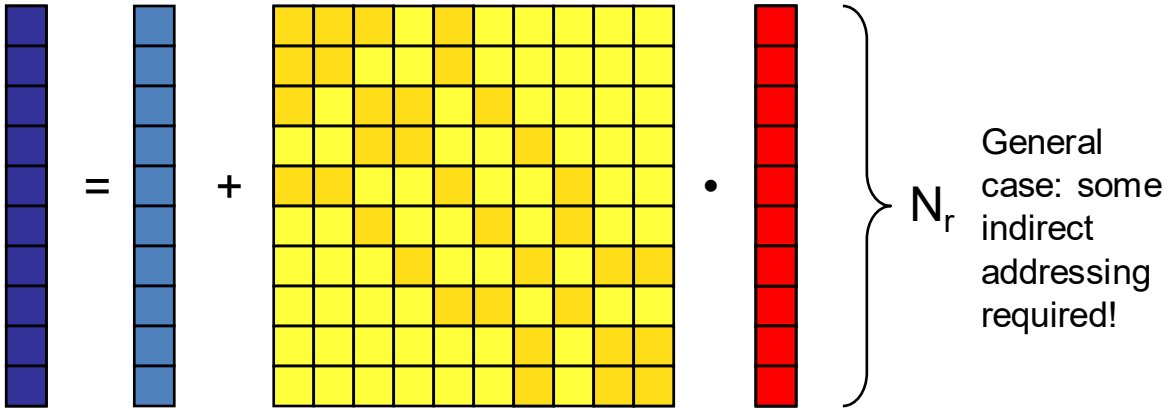
BACKUP:
OpenMP-parallel sparse matrix-vector multiplication

High Performance Computing

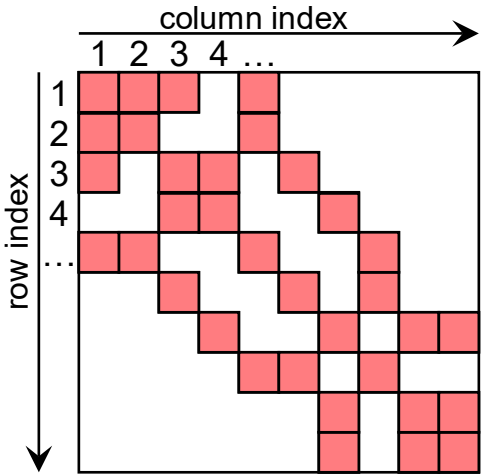
Sparse matrix-vector multiply (spMVM)

- Key ingredient in some matrix diagonalization algorithms
  - Lanczos, Davidson, Jacobi-Davidson

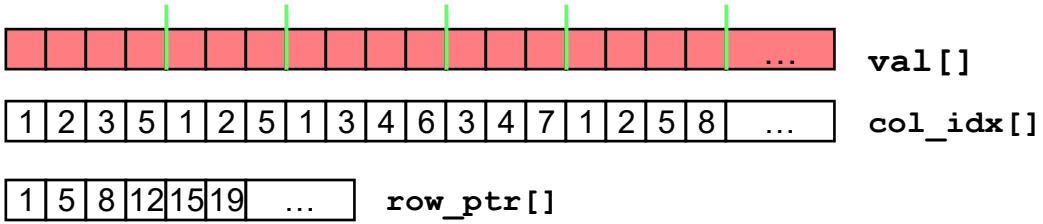
- Store only N_{nz} nonzero elements of matrix and RHS, LHS vectors with N_r (number of matrix rows) entries
- "Sparse": $N_{nz} \sim N_r$



CRS matrix storage scheme



- **`val[]`** stores all the nonzeros (length N_{nz})
- **`col_idx[]`** stores the column index of each nonzero (length N_{nz})
- **`row_ptr[]`** stores the starting index of each new row in **`val[]`** (length: N_r)



Case study: Sparse matrix-vector multiply

- Strongly memory-bound for large data sets
  - Streaming, with partially indirect access:

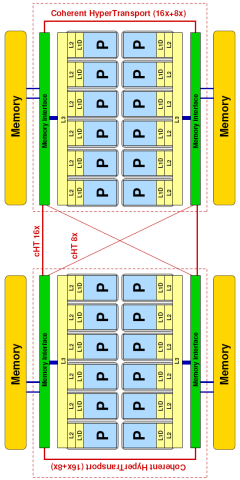
```
!$OMP parallel do
do i = 1,Nr
 do j = row_ptr(i), row_ptr(i+1) - 1
  c(i) = c(i) + val(j) * b(col_idx(j))
 enddo
enddo
!$OMP end parallel do
```

  - Usually many spMVMs required to solve a problem

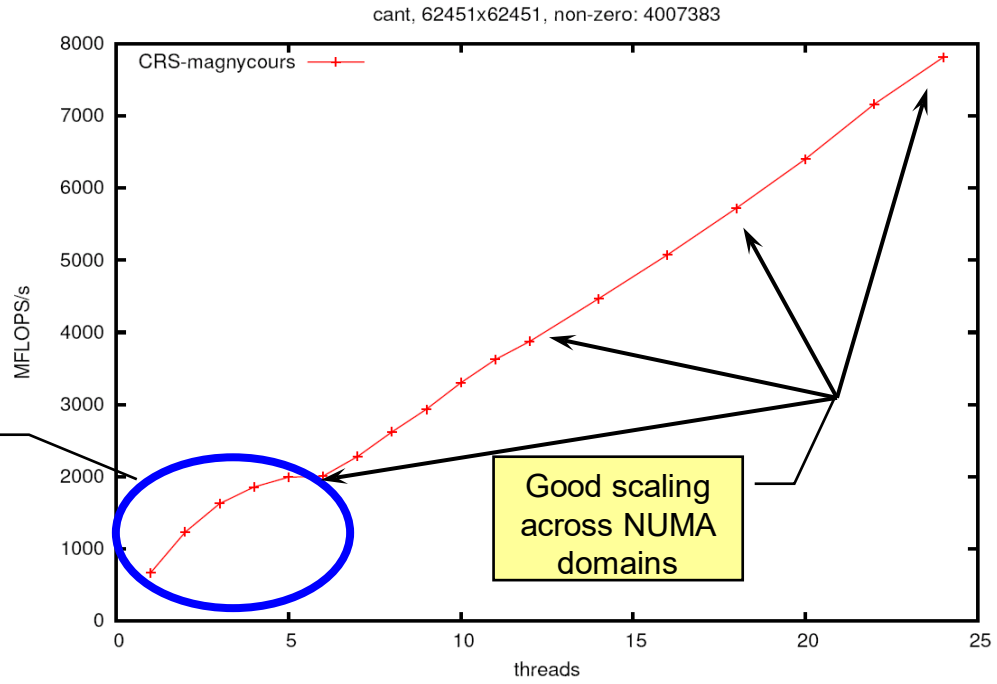
- Following slides: Performance data on one 24-core AMD Magny Cours node

Application: Sparse matrix-vector multiply

Case 1: Large matrix

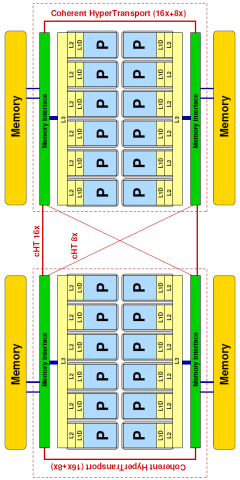


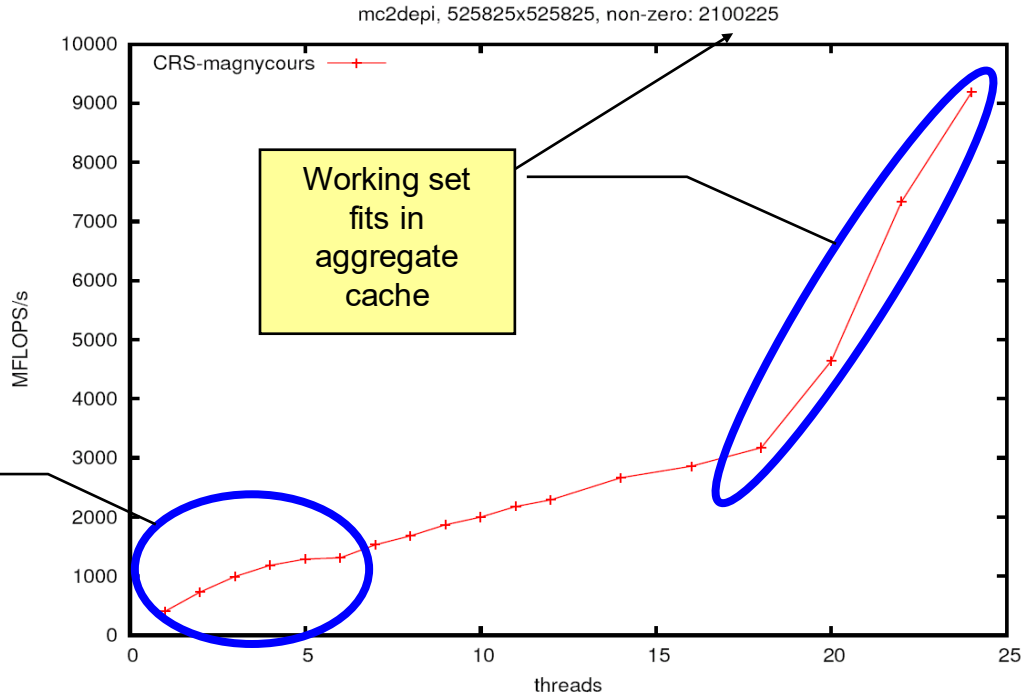
Intrasocket bandwidth bottleneck



Good scaling across NUMA domains

Case 2: Medium size



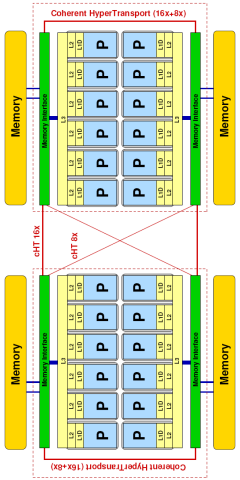


Working set fits in aggregate cache

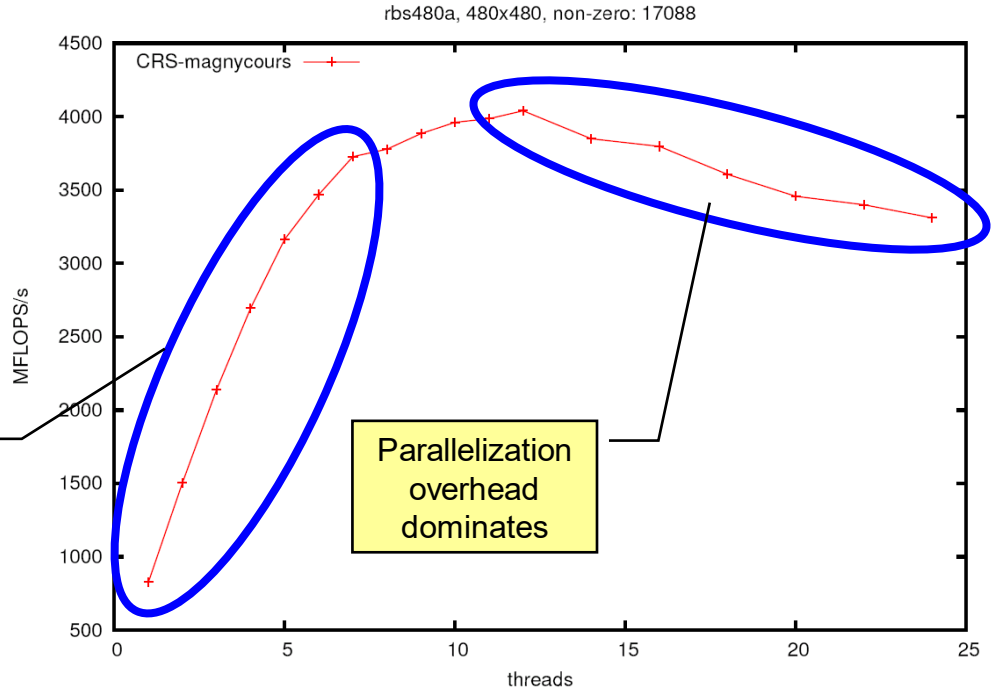
Intrasocket bandwidth bottleneck

Application: Sparse matrix-vector multiply

Case 3: Small size



No bandwidth bottleneck



Parallelization overhead dominates