NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Advanced OpenMP

Selected performance issues

Data locality & ccNUMA

Page migration

Prof. Dr. G. Wellein[a,b], Dr. G. Hager[a]

[a] Erlangen National High Performance Computing Center (NHR@FAU)

[b] Department für Informatik

Friedrich-Alexander-Universität Erlangen-Nürnberg, Sommersemester 2024

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Efficient programming of ccNUMA nodes

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Advanced OpenMP

Selected performance issues

Data locality & ccNUMA

Page migration

Conditional parallelism

- Sometimes we want a flexible means to avoid parallelism
  - Barrier cost, cost of waking up the team of threads, scheduling cost
- `if` clause takes any valid condition in the base language
  - Can be applied to various constructs, including `task`

```
#pragma omp parallel if(n>8000)
{
#pragma omp for
  for(int i=0; i<n; ++i)
    a[i] = b[i] + c[i] * d[i];
}
```

Example: suppress nested parallelism in a library routine

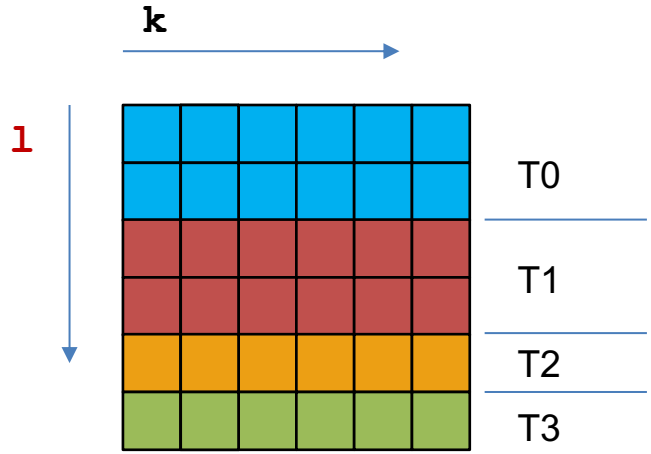
```
#pragma omp parallel \
  if(!omp_in_parallel())
{
  ... ! parallel region
}
```

- Less extreme option: `num_threads(n)` clause to reduce # of threads in region

Coarse granularity

- Even if there is enough work in a parallel loop, granularity may cause imbalance
- Example: load imbalance if M is “small,” i.e., comparable to number of threads

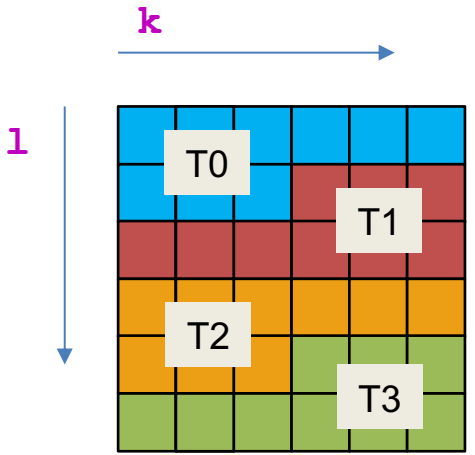
```
double a[M][N];
#pragma omp parallel for \
  schedule(static) reduction(+:res)
  for(int l=0; l<M; ++l)
    for(int k=0; k<N; ++k)
      res += a[l][k];
```



Coarse granularity

- `collapse(n)` clause coalesces perfect n-way loop nest

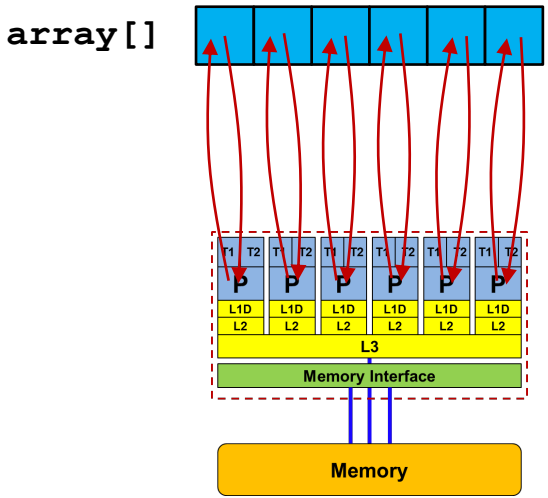
```
double a[M][N];
#pragma omp parallel for \
  schedule(static) reduction(+:res) \
  collapse(2)
  for(int l=0; l<M; ++l)
    for(int k=0; k<N; ++k)
      res += a[l][k];
```



False sharing

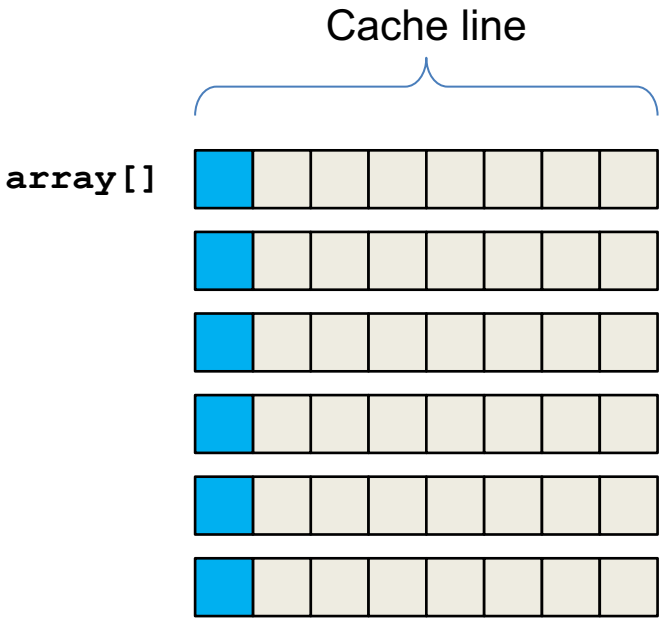
- If multiple threads frequently access the same cache line and at least one thread writes to it → false sharing

```
int a=0;
int array[omp_get_max_threads()];
#pragma omp parallel
{
  int id = omp_get_thread_num();
  array[id]=0;
  #pragma omp for
    for(int i=0; i<N; ++i) {
      int x = compute(i);
      array[id] += x;
    }
  #pragma omp critical
    a += array[id];
}
```



False sharing: two solutions

1. Padding: leave ≥ 1 cache line of room between adjacent entries




```
int a=0;
const int CL=8;
int array[omp_get_max_threads()*CL];
#pragma omp parallel
{
  int id = omp_get_thread_num();
  array[id*CL]=0;
  #pragma omp for
    for(int i=0; i<N; ++i) {
      int x = compute(i);
      array[id*CL] += x;
    }
  #pragma omp critical
    a += array[id*CL];
}
```

False sharing: two solutions

2. Privatization and reduction

```
int a=0;
#pragma omp parallel
{
  #pragma omp for reduction(+:a)
    for(int i=0; i<N; ++i) {
      int x = compute(i);
      a += x;
    }
}
```

If possible, prefer privatization over synchronization!

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Advanced OpenMP

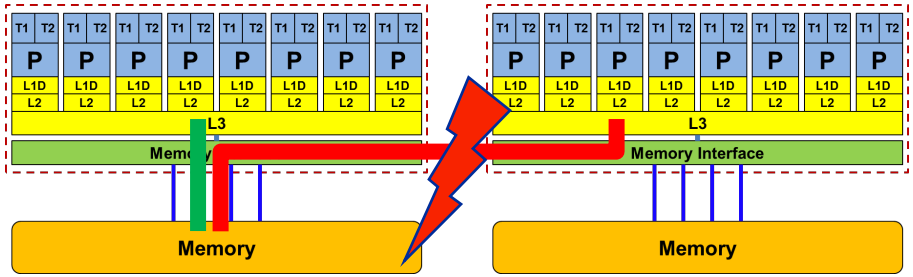
Selected performance issues

Data locality & ccNUMA

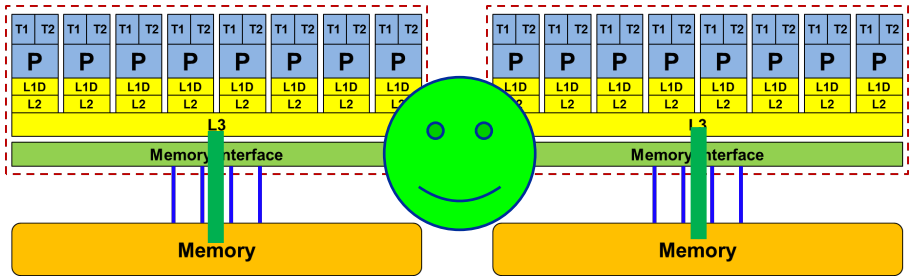
Page migration

ccNUMA – The other affinity to care about

- ccNUMA:
  - Whole memory is transparently accessible by all cores
  - 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 4kB, possibly more)

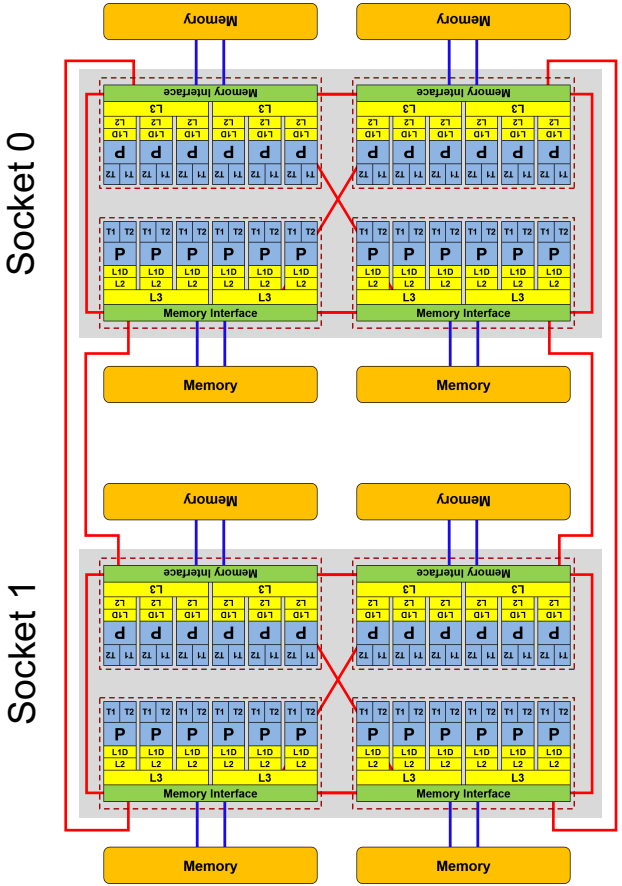


How much does nonlocal access cost?

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

STREAM Triad bandwidth measurements [Gbyte/s]

MEM node \ CPU node	0	1	2	3	4	5	6	7
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



Enforcing memory locality with `numactl`

- `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 ./a.out
  done
done
```

ccNUMA map scan

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

Advanced affinity enforcement with LIKWID → see separate lectures

- But what is the default without `numactl`?

ccNUMA default placement policy

"Golden Rule" of ccNUMA:

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

(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

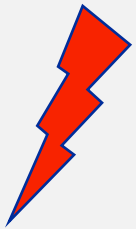
Mapping takes place here

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

Coding for ccNUMA data locality

Most simple case: explicit initialization

```
const int n=10000000;
a=(double*)malloc(n*sizeof(double));
b=(double*)malloc(n*sizeof(double));

...


for(int i=0; i<n; ++i)
    a[i] = 0.;
...

#pragma omp parallel for
for(int i=0; i<n; ++i)
    b[i] = function(a[i]);
```



```
const int n=10000000;
a=(double*)malloc(n*sizeof(double));
b=(double*)malloc(n*sizeof(double));
...

#pragma omp parallel
{
#pragma omp for schedule(static)
for(int i=0; i<n; ++i)
    a[i] = 0.;
...
#pragma omp for schedule(static)
for(int i=0; i<n; ++i)
    b[i] = function(a[i]);
}
```

Coding for ccNUMA data locality: DMVM

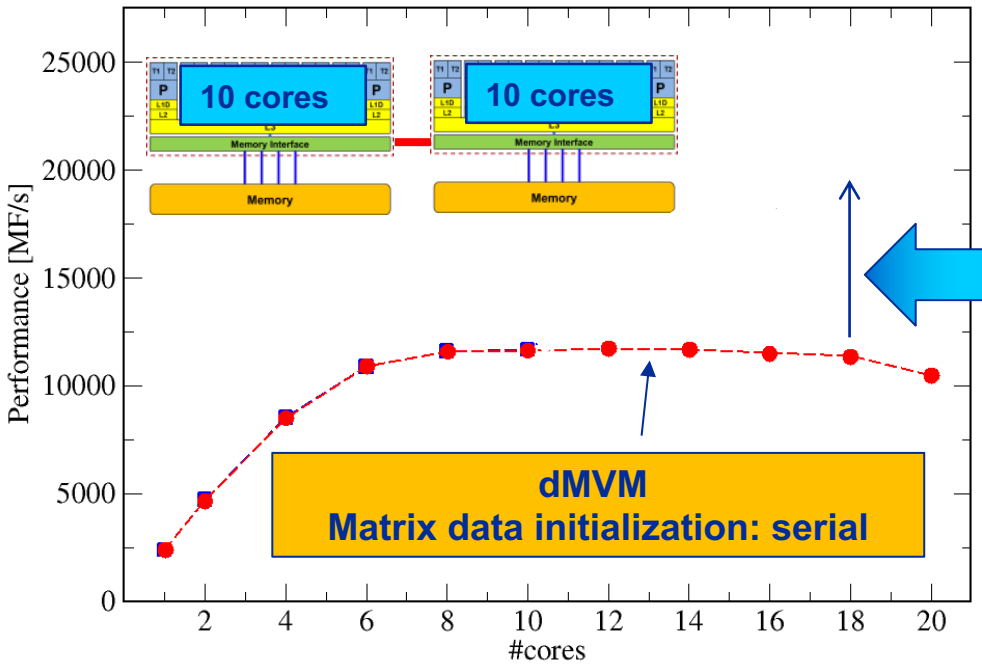
- Example: Dense Matrix-Vector-Multiplication (DMVM) subroutine

```
void dmvm(int n, int m, double *lhs, double *rhs, double *mat){
#pragma omp parallel for private(offset,c)schedule(static) {
  for(r=0; r<n; ++r) {
    offset=m*r;
    for(c=0; c<m; ++c)
      lhs[r] += mat[c + offset]*rhs[c];
}}
```

- DMVM routine is called from serial code

Coding for ccNUMA data locality: DMVM

```
#pragma omp parallel for schedule(static) private(c) {
for(r=0; r<n; ++r)
    for(c=0; c<m; ++c) mat[c + m*r] = …; } }
```



Parallelization of matrix data initialization

Coding for ccNUMA data locality: DMVM (v2)

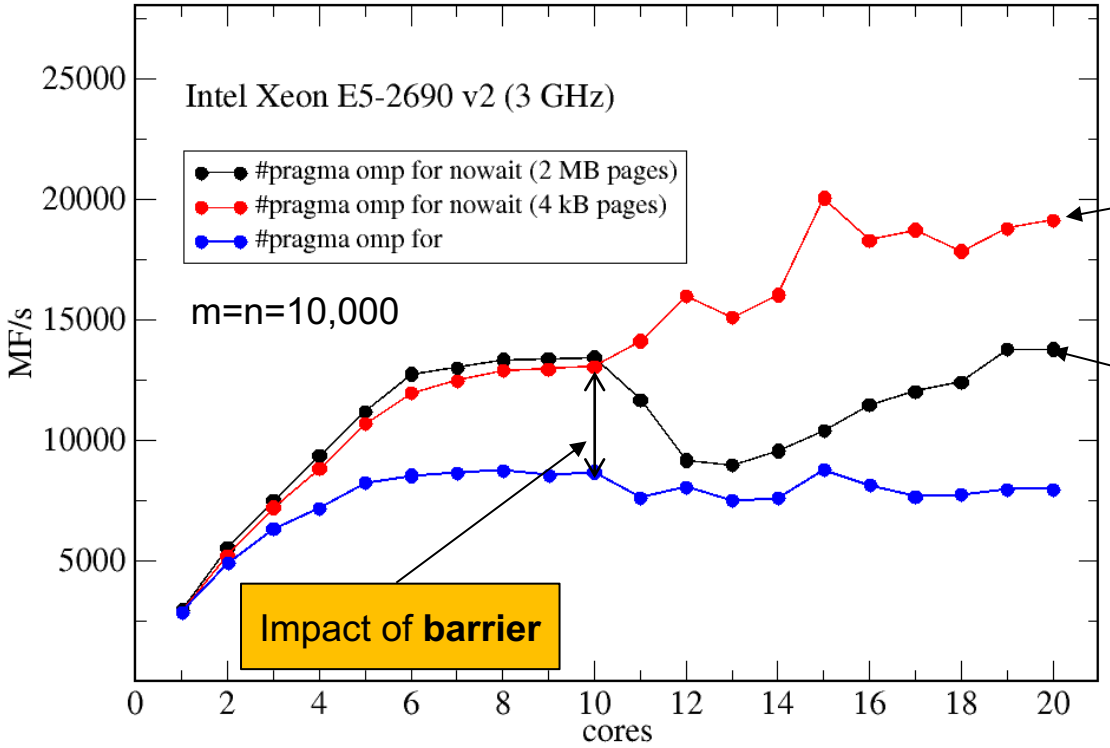
- DMVM with different formulation and inner loop parallelization

```
void dmvm2(int n, int m, double *lhs, double *rhs, double *mat){
for(r=0; r<n; ++r) {
    offset=m*r;
#pragma omp parallel for schedule(static) {
    for(c=0; c<m; ++c)
      lhs[c] += mat[c + offset]*rhs[r]; }
}
```

- First touch need to be adapted! (Parallelization changed)

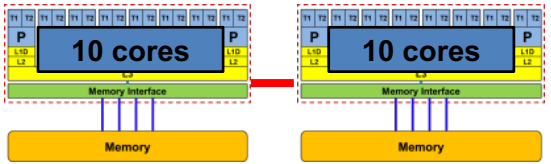
Coding for ccNUMA data locality: DMVM (v2)

```
for(r=0; r<n; ++r)
#pragma omp parallel for schedule(static) {
for(c=0; c<m; ++c) mat[c + m*r] = …; } }
```



NUMA problems despite placement?!

NUMA problems get worse with larger **page size (2 MB)**



Coding for ccNUMA data locality: DMVM (v2)

- Bad NUMA scaling: **NUMA Placement** is done **on page basis**
  - Data placement is "too fine"

```
for(r=0; r<n; ++r)
#pragma omp parallel for schedule(static) {
   for(c=0; c<m; ++c) mat[c + m*r] = …; } }
```

  - Remember all threads work on each row in parallel!
  - 4 KB pages → Single thread initializes 500 Entries
    → 20 threads initialize 1 row (20 * 500 = 10,000 = **m**)
  - 2 MB pages → Single thread initializes 500*500 Entries
    → 25 rows are initialized by a single thread

- What about placing lhs[] and rhs[] vectors?
  - **lhs[]** – exploit temporal locality: Accessed **r-1** times from cache
  - **rhs[]** – **r** memory accesses vs. **m*r** accesses for **mat[]** → lower order

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
- How about global objects?
  - 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

NUMA-aware allocator for C++ `std::vector<>`

```
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);
  }
...
};
```

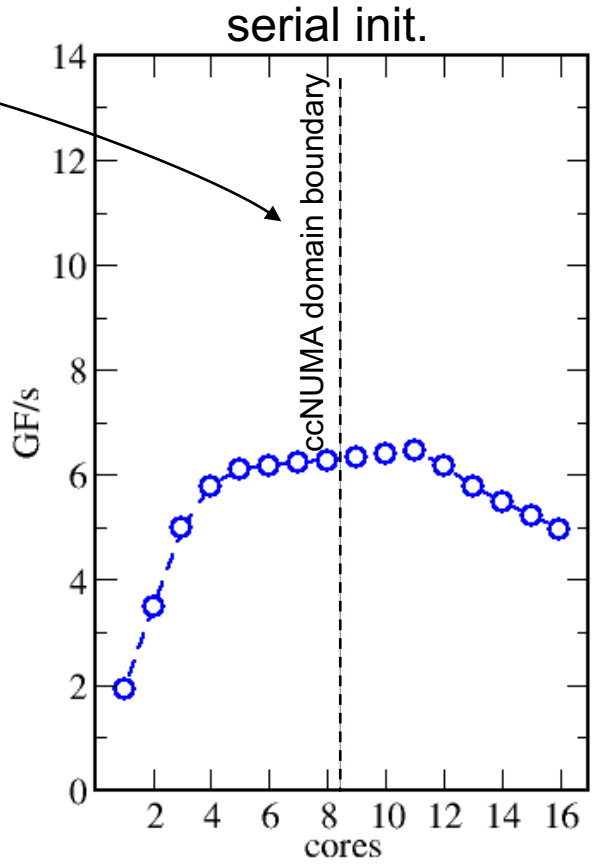
Application:

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

Diagnosing bad ccNUMA locality

- 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

- Important:
  This is all only relevant if the code is actually sensitive to memory access!



Using performance counters for diagnosis

- Intel Ivy Bridge EP node (running 2x5 threads): measure NUMA traffic per core with **`likwid-perfctr`**

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

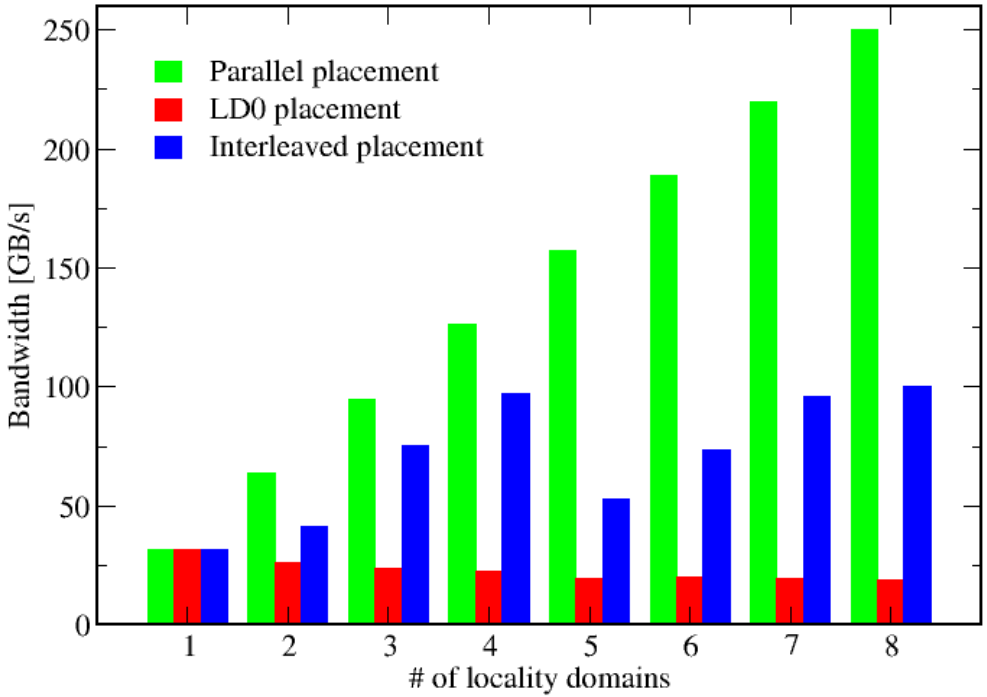
- Summary output:

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	0.735579		2
Local DRAM bandwidth [MByte/s] STAT	19140.761	1816.0		
Remote DRAM data volume [GByte] STAT	9.16628352	0.86682		2
Remote DRAM bandwidth [MByte/s] STAT	22630.098	2140.0		
Memory data volume [GByte] STAT	16.919217152	1.690376		2
Memory bandwidth [MByte/s] STAT	41770.861	4173.27	4180.714	4177.0861

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

OpenMP STREAM triad on AMD Epyc 7451 (6 cores per LD)

- Parallel init: Correct parallel initialization
- LD0: Force data into LD0 via `numactl -m 0`
- Interleaved:
  `numactl --interleave ...`



NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Advanced OpenMP

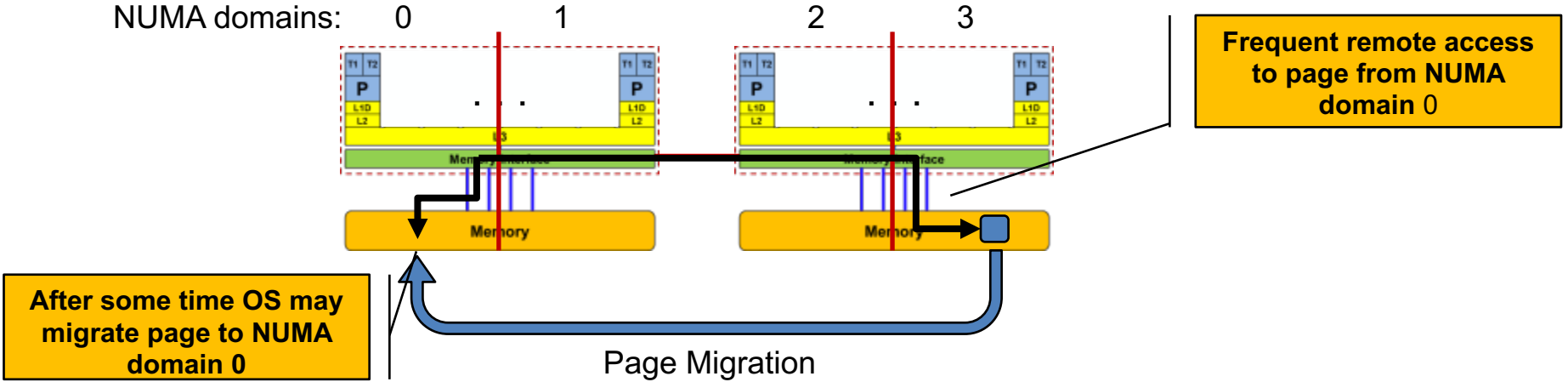
Selected performance issues

Data locality & ccNUMA

Page migration

ccNUMA: OS may try to solve locality problem

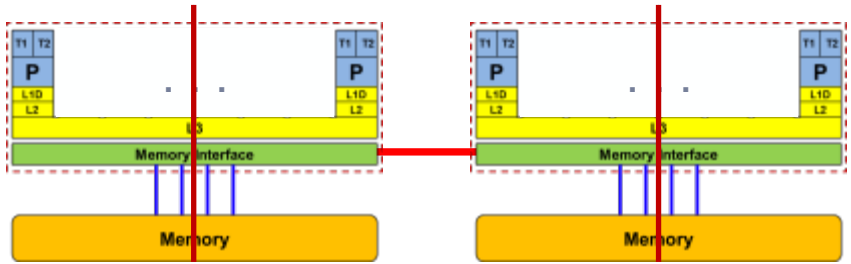
- Operating system may support „page migration“
- Page Migration: If a page is frequently requested by a remote NUMA domain it is migrated to the respective NUMA domain
- Assume 2-socket Haswell (14 cores per socket) system with Cluster on Die enabled → 4 NUMA domains with 7 cores each



ccNUMA: OS driven page migration

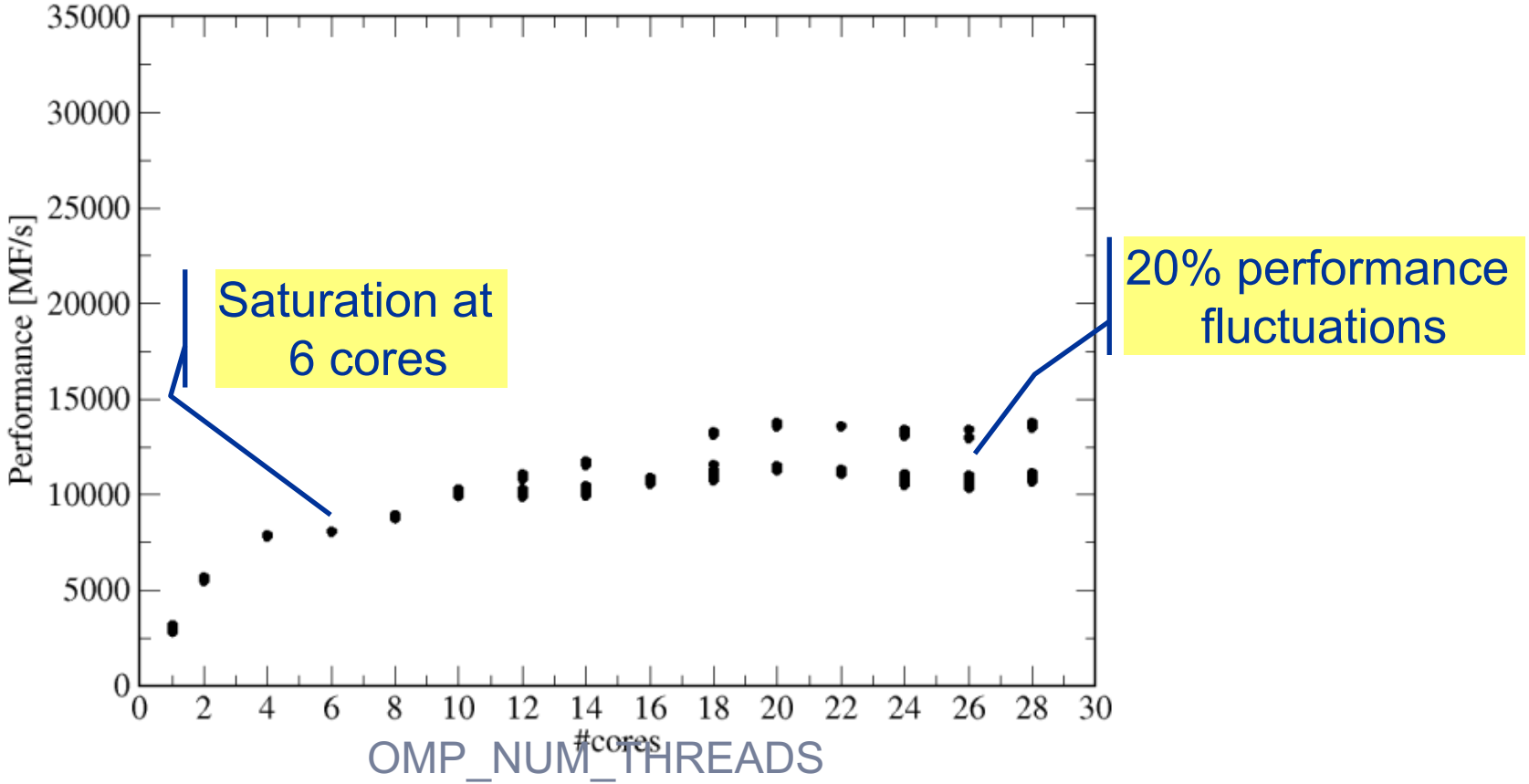
```
void dmvm(int n, int m, double *lhs, double *rhs, double *mat){

#pragma omp for private(offset,c) schedule(static) {
  for(r=0; r<n; ++r)

    offset=m*r;

    for(c=0; c<m; ++c)

      lhs[r] += mat[c + offset]*rhs[c];
} }
```

- Serial initialization of matrix data
- Run scaling experiment with OMP_NUM_THREADS = 1 (,2,4,6,8,…,28)
  Cluster-on-Die enabled: 4 NUMA domains
- Fill socket first / No SMT

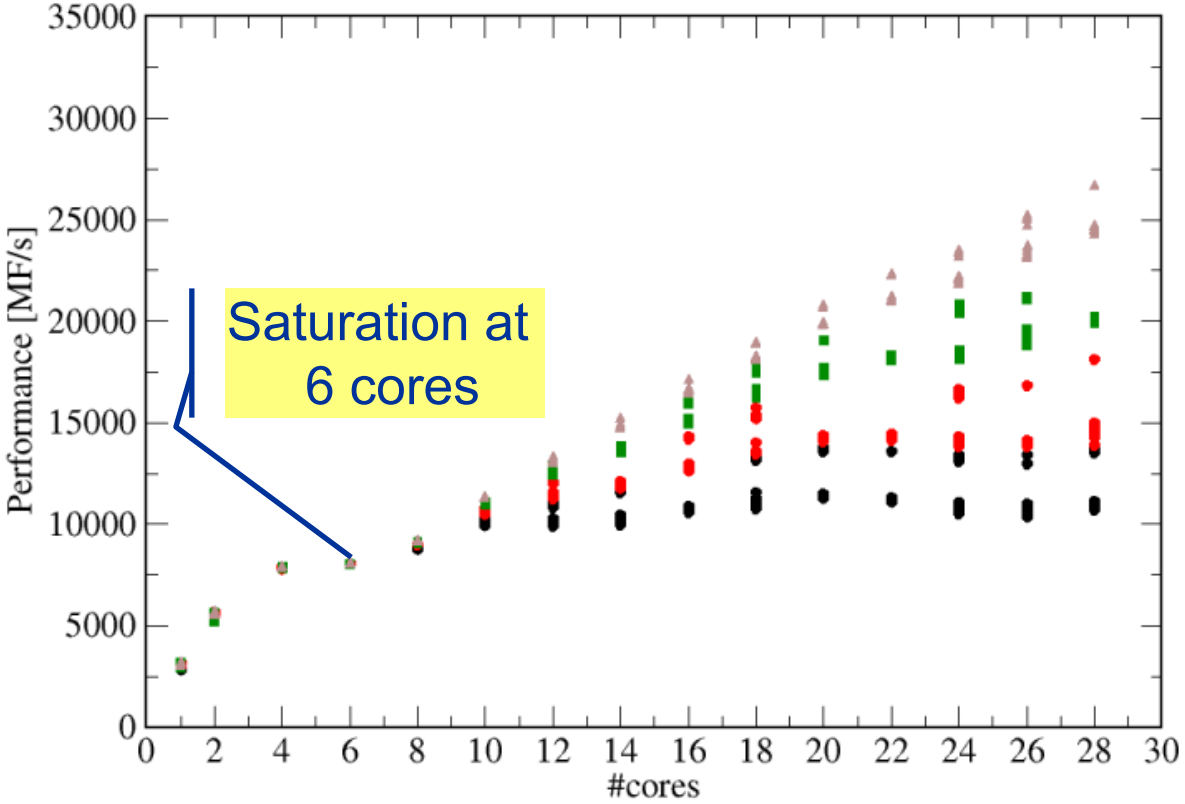


- Run 10 tests
  (with `iter` dmvm calls each)

ccNUMA: OS driven page migration (`iter=200`)



ccNUMA: OS driven page migration



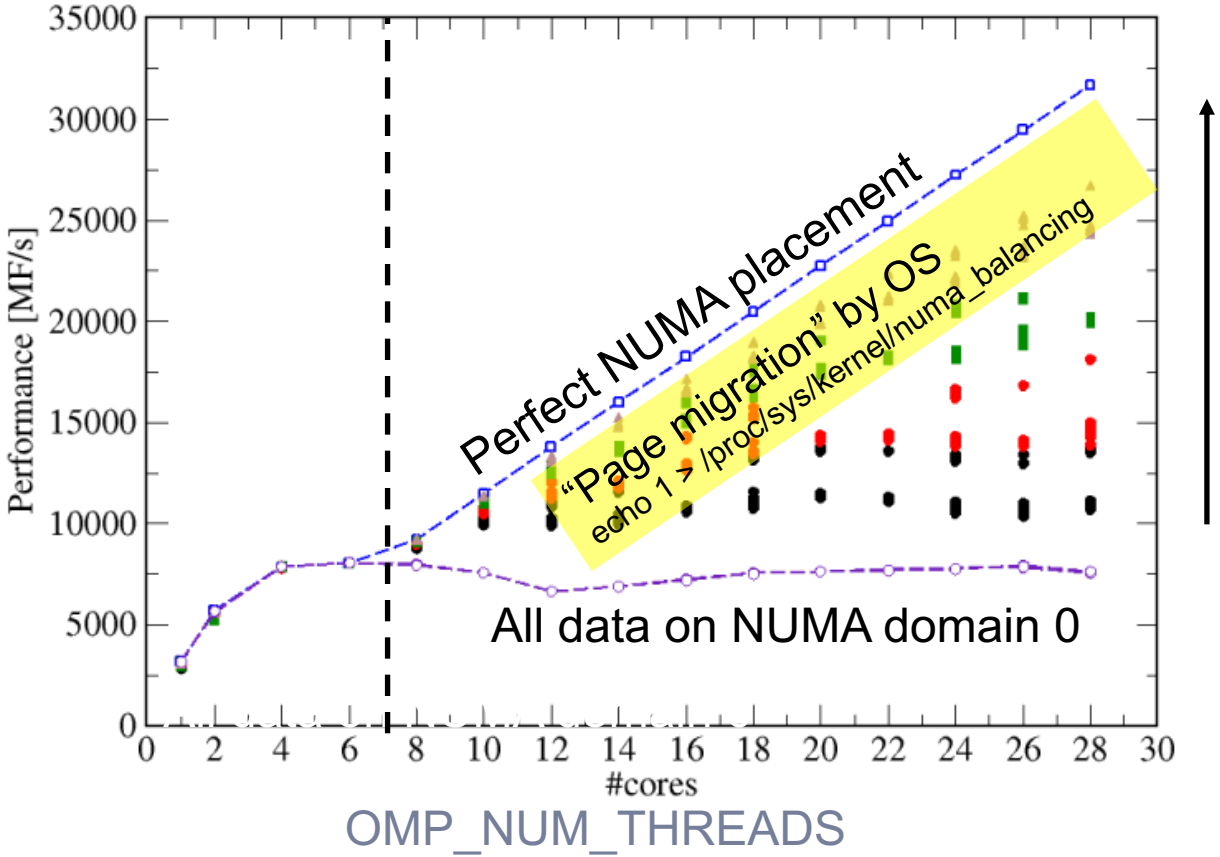
iter=2000

iter=1000

iter=500

iter=200

ccNUMA: OS driven page migration



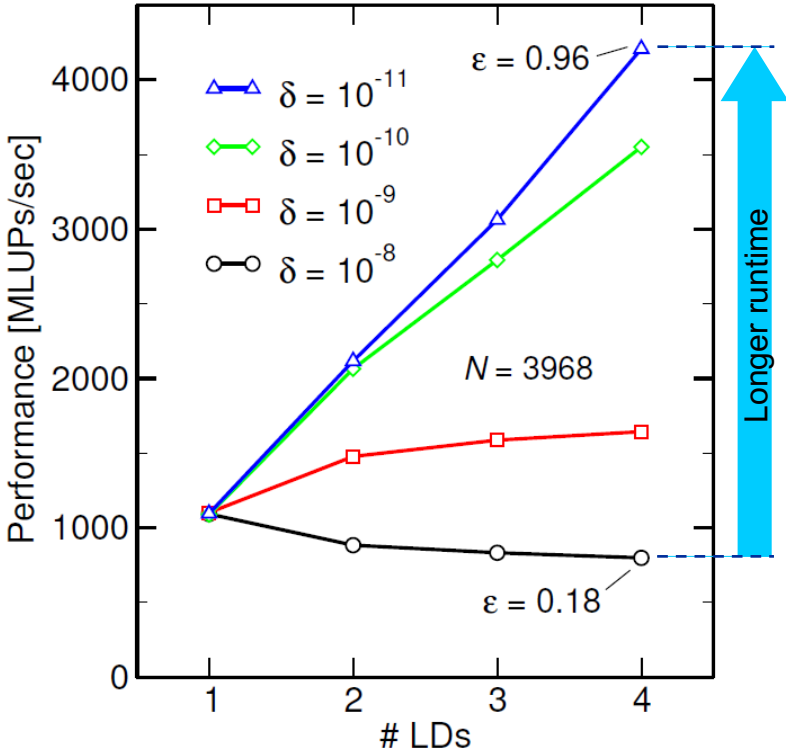
Page Migration „takes time“

Migration threshold: OS parameter

Works fine for regular access patterns

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 δ determines runtime
  - The smaller δ, the longer the run
- Observation
  - No scaling across LDs @ large δ (runtime 0.5 s)
  - Scaling gets better with smaller δ
    up to almost perfect efficiency ε (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 current Linux distributions
- 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” → it takes some time to “kick in”
- 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`
- Apply first-touch placement
  - Look at 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
  - 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
  - Still a good idea to do parallel first touch