NumaMMA
NUMA MeMory Analyzer

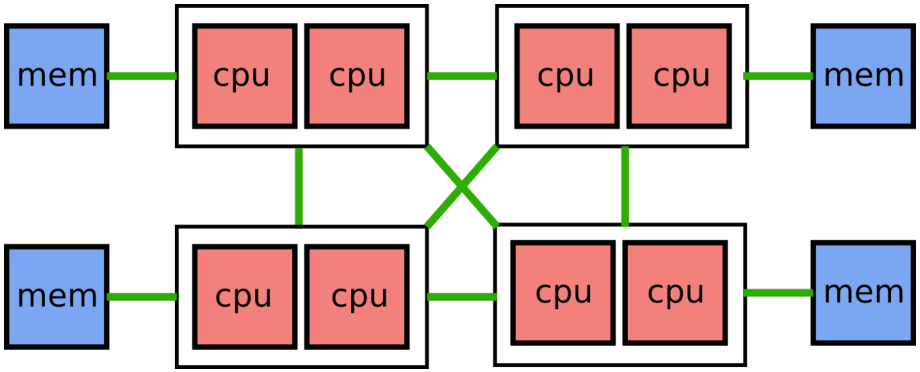
François Trahay

Parallel & Distributed systems group

Inp@ct seminar – 16 oct. 2019

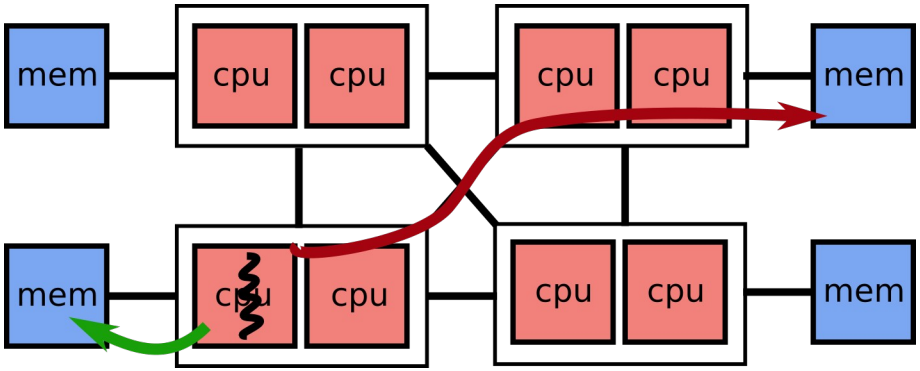
NUMA architecture

- **NUMA architectures are now common**
  - Multi-socket systems
  - Multicore CPUs
    - AMD Infinity Fabric (Zen CPU family)
    - Intel sub-NUMA clustering (Skylake family)
  - Increase the available memory bandwidth



Impact of locality

- **NUMA = Non Uniform Memory Access**
  - Fast access to the local memory
  - Slower access to remote NUMA nodes



→ **the locality of memory access impacts the performance**

eg. impact on NPB LU on a 48-core machine: up to 27%

→ **need to allocate pages on the right NUMA node**

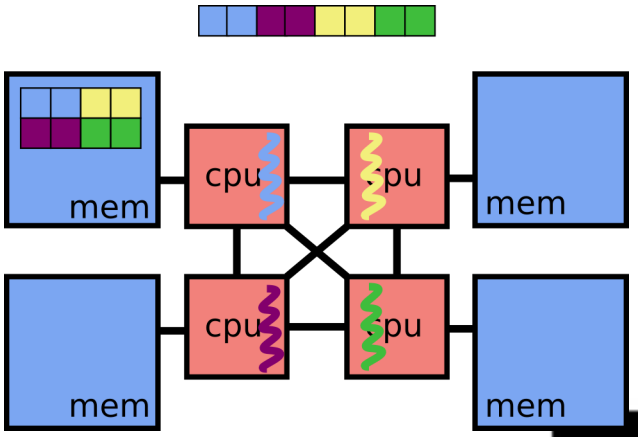
Memory allocation strategies
First-touch policy

- ***First-touch* policy**
  - Default policy on Linux
  - Lazy allocation policy
  - Allocate a page locally when a thread touches it
    - Assumption: this thread is probably the one that will use the page the most
      - Assumption may be wrong !

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

for(int i=0; i<N; i++) {
  array[i] = something(i);
}

#pragma omp parallel for
for(int i=0; i<N; i++) {
  double value = array[i];
  /* ... */
}
```



Challenges

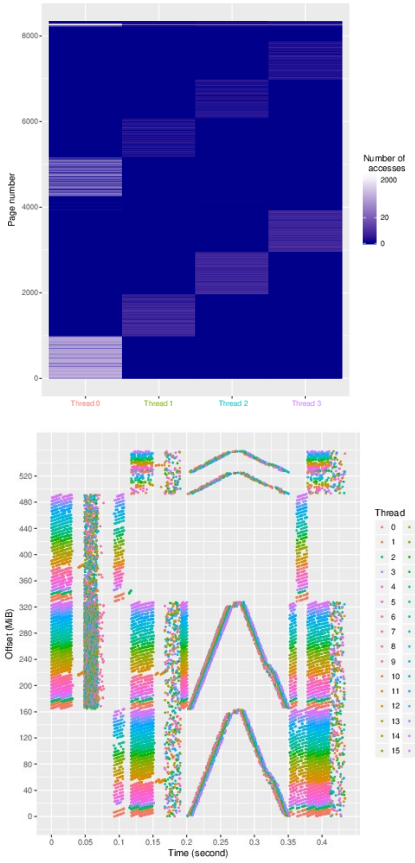
- **Choosing the right allocation policy**
  - First-touch ? Interleaved ?
  - May depend of the memory access pattern to the object

- **How to detect the memory access pattern ?**
  - Instrumentation-based approaches
    - eg. NUMAProf
      + accuracy of the memory access collection
      - High overhead (~ 27x)
  - Sampling-based approaches
    - eg. MemProf
      - coarse grain view of memory access
      + low overhead (~5%)

NumaMMA

NumaMMA – NUMA MeMory Analyzer

- Analyze the memory access pattern of parallel applications
  - Low overhead collection of memory access
- Report:
  - The most accessed memory objects
  - Which thread access which part of an object
  - The evolution of access patterns over the time
- Freely available as open-source
  - https://github.com/numamma



NumaMMA
Collecting memory access information

- **Hardware memory sampling**
  - eg. Intel PEBS, AMD IBS
  - Every X instructions, the CPU collects a sample
    - Address of the memory load/store
    - Thread/Instruction that issued the memory access
    - Where the data is stored (cache, RAM, remote RAM, …)
    - Cost of the memory access (ie. latency)
  - Information stored in a buffer
  - Low overhead sampling (eg. < 1-2 %)

#tid	timestamp	address	mem_level	latency
0	5835423725112	0x557735cf07c8	L2 Hit	24
0	5835456302591	0x557736353ef8	Local RAM Hit	779
0	5835466068752	0x55773642a0c0	Local RAM Hit	657
0	5835471131886	0x5577362726e8	L2 Hit	23
0	5835566865010	0x557735d1fd28	L3 Hit	52
0	5835567586835	0x557735d04710	L3 Hit	64
0	5835604540592	0x557736320900	Local RAM Hit	1585
0	5835605025940	0x557735c39900	Local RAM Hit	265
0	5835618194705	0x557735f0e428	L2 Hit	24
0	5835693753719	0x557735f16a78	L2 Hit	23
0	5835709318658	0x557736260f00	Local RAM Hit	266

NumaMMA
Identifying memory objects

- **Static memory object**
  - eg. global variables
  - Search for symbols in the ELF binaries
- **Dynamic memory objects**
  - malloc, realloc, calloc, free, …
  - Intercept dynamic allocations with LD_PRELOAD
- **For each object, NumaMMA knows**
  - The allocation/de-allocation timestamp
  - The start/end address

NumaMMA
Matching samples

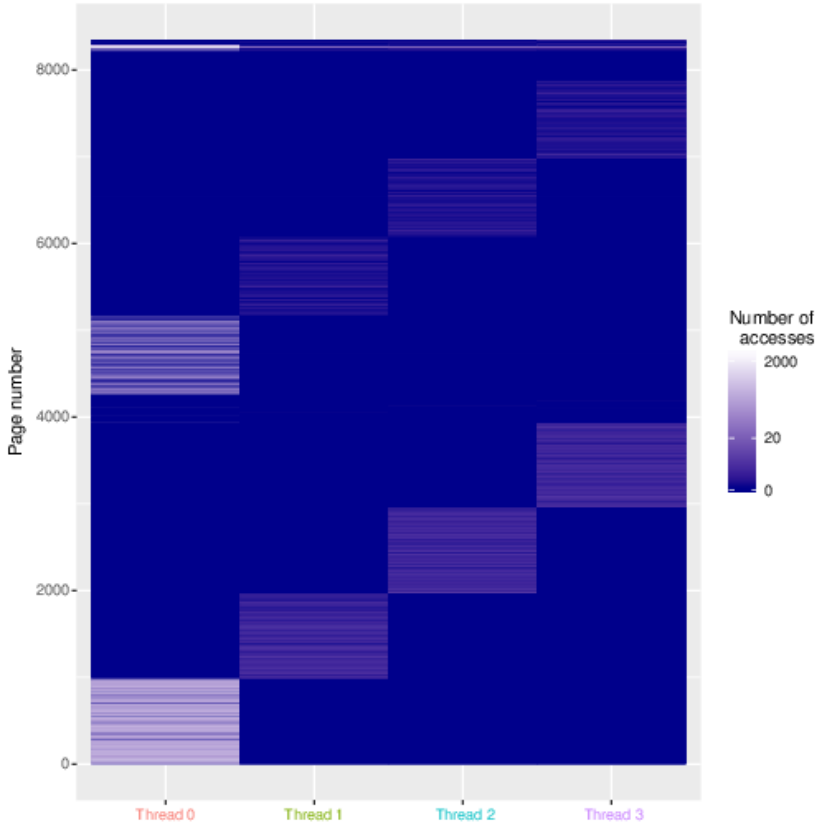
- **For each sample, find the matching memory object**
- **Once found, update counters**
  - Number of read/write accesses
  - Total cost of memory accesses to the object
- **For large objects, counters are computed per page**

- **Generate a summary of the most accessed objects**

```
Summary of the call sites:
--------------------------
Sorting call sites
0       fields_ (size=2520000) - 34098 read access (total weight: 362881, avg weight: 10.642296). 66541 wr_access
1       [stack] (size=412316860415) - 47982 read access (total weight: 345827, avg weight: 7.207432). 60001 wr_access
2       constants_ (size=1272) - 589 read access (total weight: 5131, avg weight: 8.711375). 0 wr_access
3       /usr/lib/x86_64-linux-gnu/libgomp.so.1(+0x9b49) [0x7f6b06eb4b49] (size=192) - 96 read access (total weight: 672, avg weight: 7.
```

NumaMMA
Memory access patterns

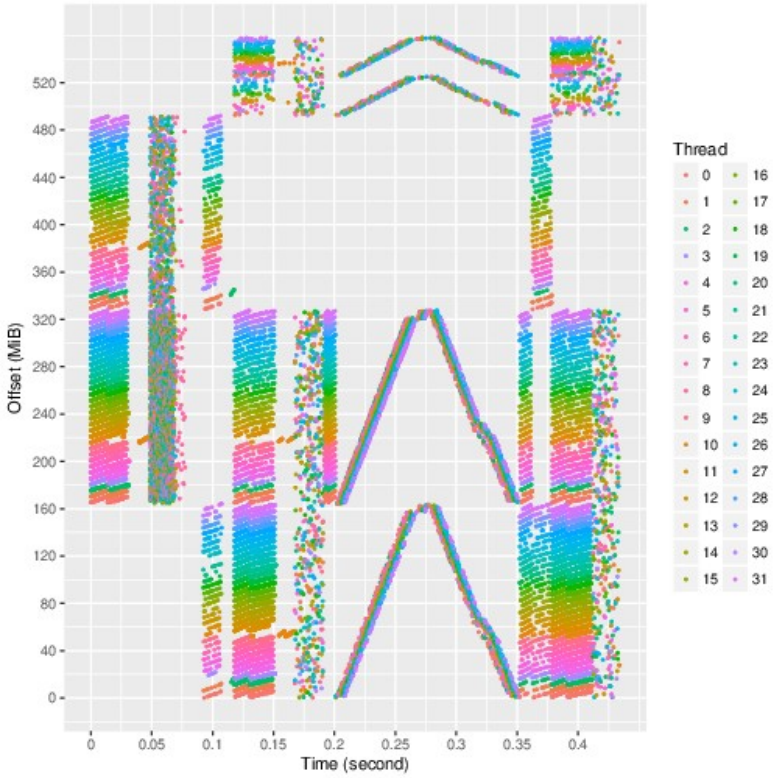
- **Access pattern to an object**
  - X-axis: threads
  - Y-axis: memory pages
- **Thread 0 access pattern**
  - Pages [0-992] [4267-5176]
- **Thread 3 access pattern**
  - Pages [993-1969] [5178-6074]
- **Due to sampling, some pages may not be detected**
  - Depends on the sampling frequency



main_flt_mem in NPB CG

NumaMMA
Evolution of access patterns over the time

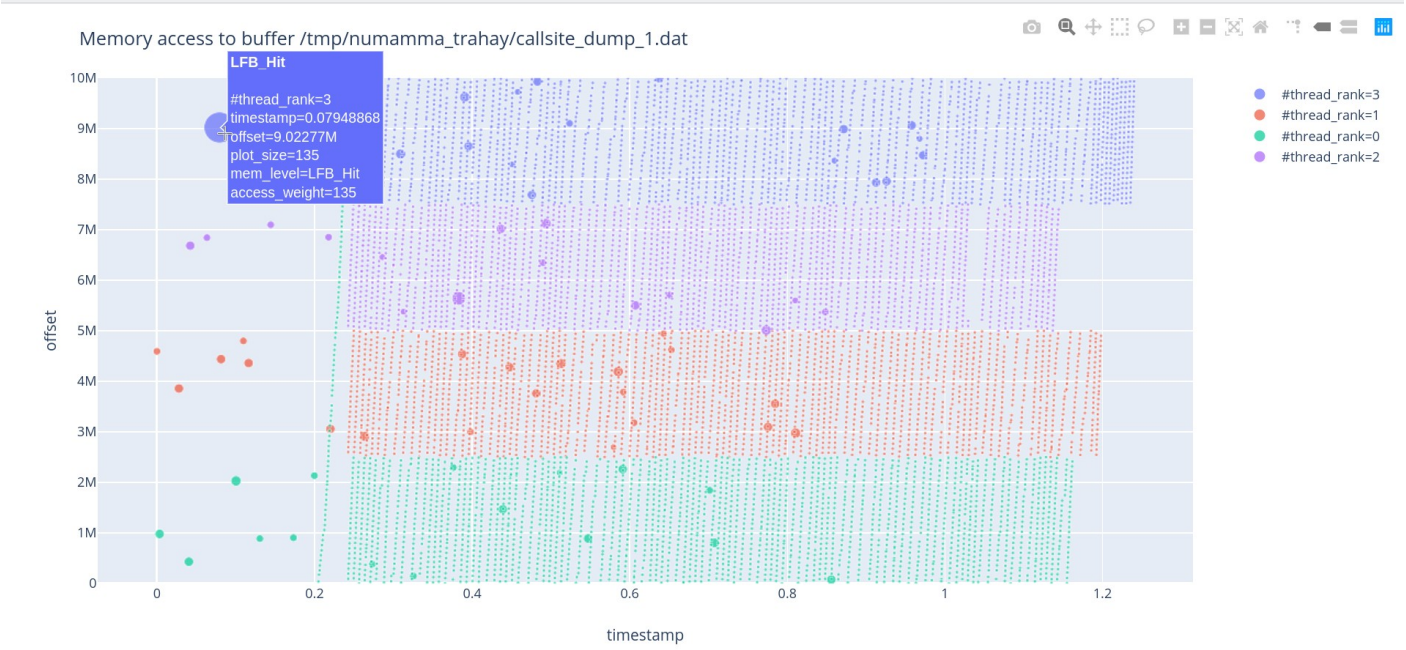
- **Access pattern to an object over the time**
  - X-axis: time
  - Y-axis: memory pages
  - Color: thread id
- **Allows to detect the phases of the application**



cvar in NPB LU

NumaMMA
Interactive view of the memory accesses

- **Gives additional information**
  - Level in the memory hierarchy
  - Memory access latency
  - read/write memory access
- **Allows to zoom in**

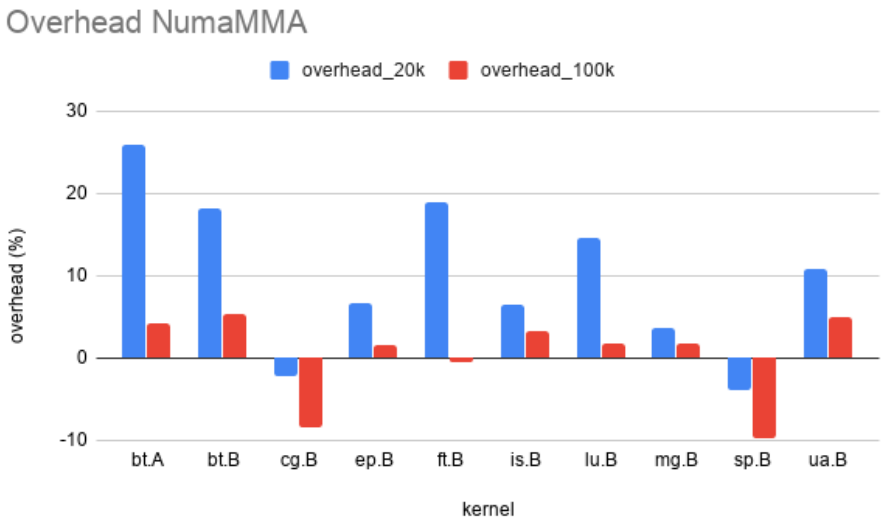


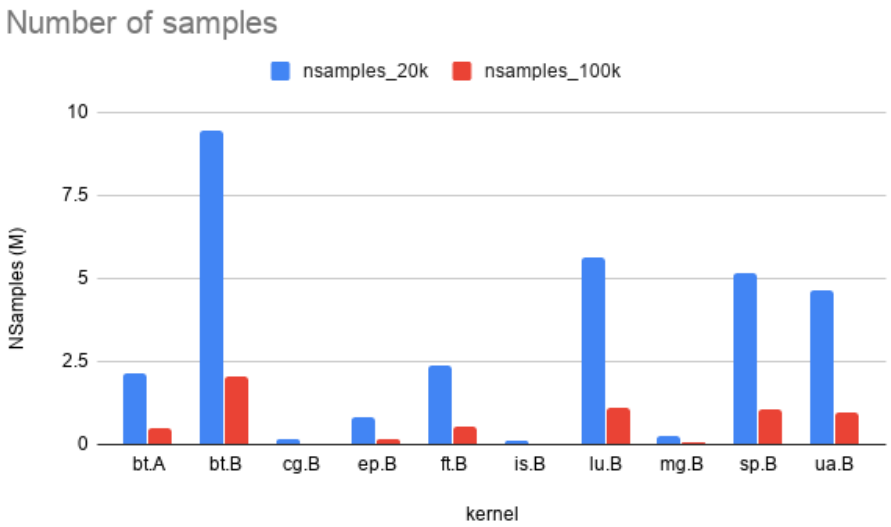
Evaluation

- **Run multithreaded applications from NAS Parallel Benchmark and Parsec**
  - Evaluate the overhead
  - How to use NumaMMA to improve the execution time of applications ?
- **Experiment setup**
  - Intel32:
    - 2 Intel Xeon E5-2630 v2 CPUs (16 cores/32 threads)
    - 32 GiB RAM (2 NUMA nodes)
    - Linux 4.11, GCC 6.3
  - AMD48:
    - 4 AMD Opteron 6174 CPUs (48 cores)
    - 128 GiB RAM (8 NUMA nodes)
    - Linux 4.10, GCC 6.3

Evaluation
Overhead

- **Overhead on NAS Parallel Benchmarks**
  - Running on Intel32
  - OpenMP Implementation
  - Threads are bound
- **2 settings**
  - NumaMMA 20K
  - NumaMMA 100K





→ **Low overhead**

→ **High precision/high overhead vs. low precision/low overhead**

Evaluation
Case study: NPB LU

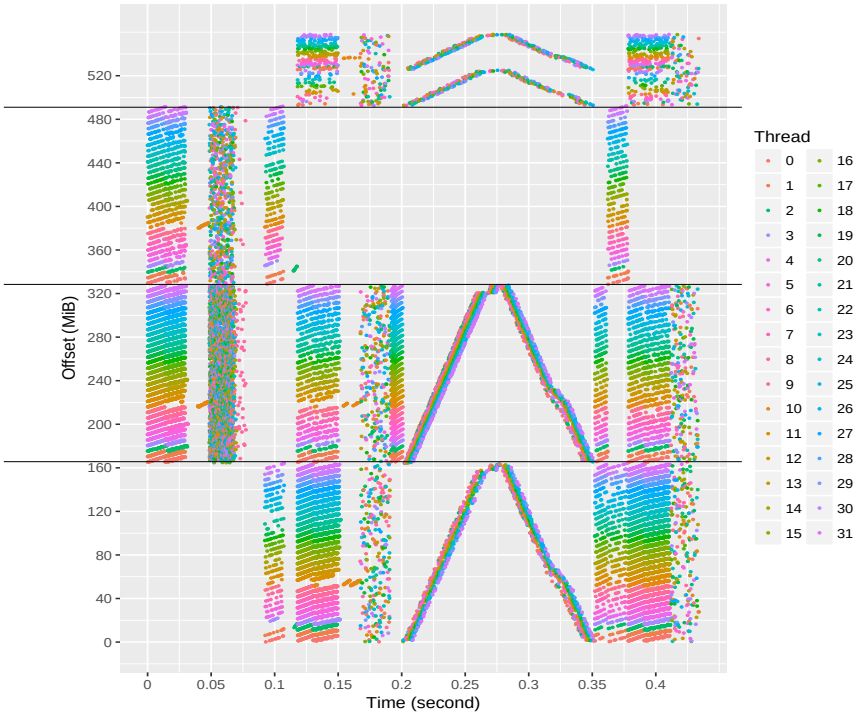
mem level	nb	percent	min cycles	max cyles	avg cycles	Total weight	Total weight (%)
L1 Hit	126979	91.88	6	886	8	1091015	59.91
L2 Hit	3017	2.18	6	159	20	61858	3.40
L3 Hit	448	0.32	6	1295	53	23771	1.31
LFB Hit	7568	5.48	12	3222	77	584637	32.10
Local RAM Hit	58	0.04	6	869	289	16763	0.92
Remote RAM Hit	108	0.08	35	1424	384	41517	2.28
Remote Cache Hit	6	0.00	32	249	194	1168	0.06
Uncached Memory hit	11	0.01	7	7	7	77	0.00

symbol	size(MiB)	nread	weight read	avg read weight	nwrite
cvar_	140.35	43647	1052451	24.11	1403571
[stack]	393,216.00	47084	332117	7.05	1657574
cjac_	8.02	19200	206177	10.74	1972931
/lib64/libgomp.so.1(+0x9	0.00	2436	14670	6.02	8
/lib64/libgomp.so.1(+0x9	0.00	19	114	6.00	0
/lib64/libc.so.6(+0x33ca	0.00	0	0	0.00	1

Evaluation
NPB LU: Analyzing the access pattern

- **`cvar`**
  - 558MiB buffer
  - Accessed by slices of 160MiB

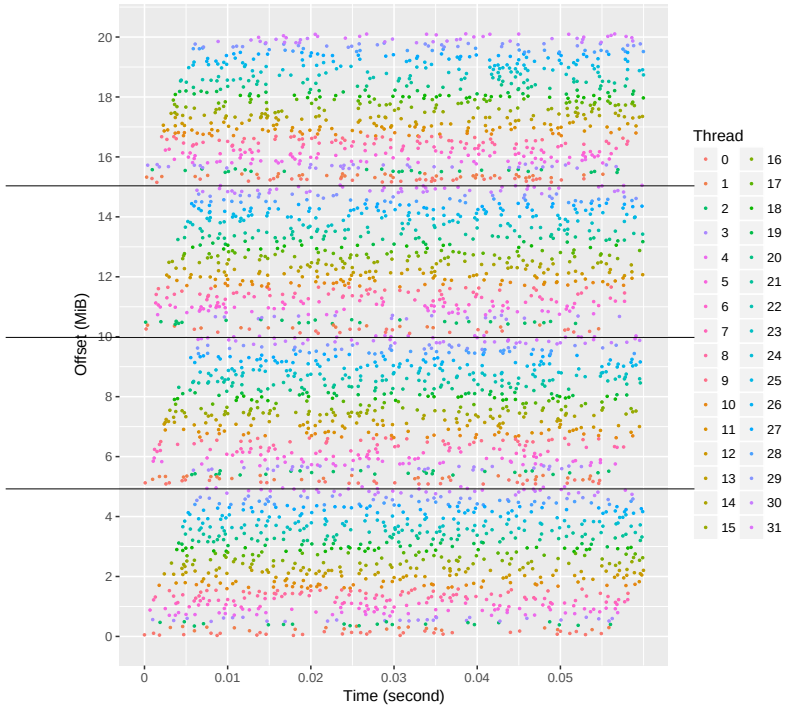
  → **block-cyclic distribution**



Access pattern to cvar

- **`cjac`**
  - 20MiB buffer
  - Accessed by slices of 5MiB

  → **block-cyclic distribution**



Access pattern to cjac

Evaluation
NPB LU: optimizing memory placement

- **Evaluation on AMD48**
- **Comparing different memory placement**
  - First-touch: default policy
  - Interleaved: interleave the memory pages of `cvar` and `cjac`
  - Block-naive: use a block distribution for `cvar` and `cjac`
  - NumaMMA: place `cvar` and `cjac` using a block-cyclic distribution

policy	execution time(s)	speedup
first-touch	102.53	1
interleaved	106.86	0.96
block-naive	109.88	0.93
NumaMMA	81.05	1.27

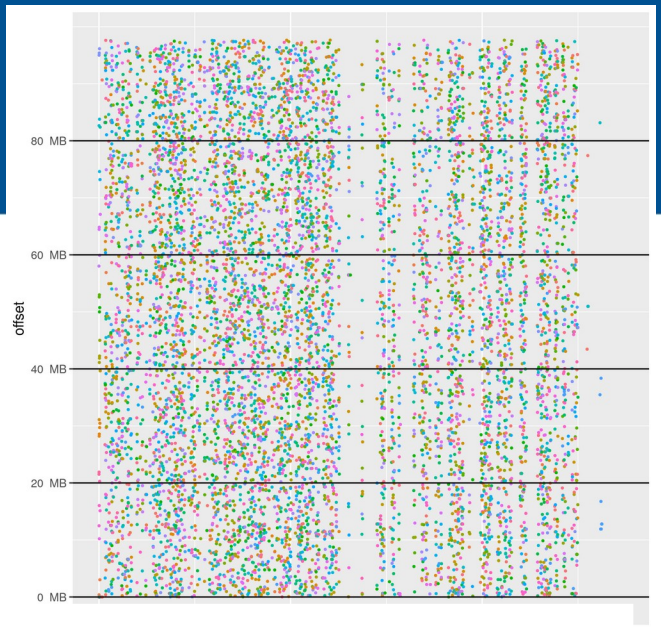
→ 27% performance improvement

Evaluation
Case study: Streamcluster

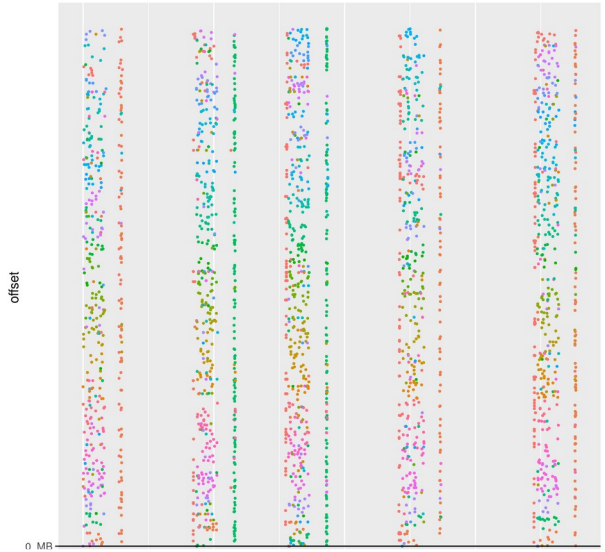
- **Application: Parsec Streamcluster**
- **Most accessed objects**
  - `block` – 98 MiB buffer (66% of the samples)
    - Evenly distributed accesses
      → interleave pages
  - `points` – 6 MiB buffer (31% of the samples)
    - Each thread accesses a part of the buffer
      → block distribution
- **Evaluation**

policy	execution time(s)	speedup
first-touch	93.72	1
interleaved	76.76	1.22
block-naive	79.75	1.17
NumaMMA	73.32	1.28

- → 28 % improvement



Access pattern to block



Access pattern to points

Conclusion & future work

- **Memory placement is important for performance**
- **NumaMMA: NUMA MeMory Analyzer**
  - Use hardware sampling to collect memory access samples
  - Report the most accessed memory objects
  - Report the threads access pattern over the time
  - Available as open-source: https://numamma.github.io/numamma/
- **Evaluation**
  - Low overhead ([2 - 20]%)
  - Reported information can be used for improving performance by up to 28%
- **Future work**
  - Port over AMD cpus
  - Automate memory placement