A JOURNEY OVER THE MEMORY MANAGEMENT STACK FOR HPC LARGE APPLICATIONS ON MODERN ACHITECTURES

Work coming from

Sébastien Valat

IXPUG - 25 sept 2019

Plan

Introduction

I. Analysis of OS paging policy

II. NUMA allocator for HPC applications

III. Cost of first touch handler

IV. MALT & NUMAPROF memory profilers

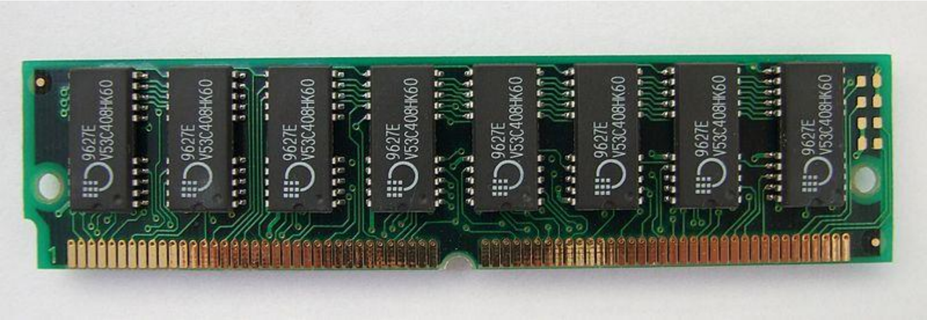
V. Conclusion

INTRODUCTION

The “new” memory context

- **Memory** becomes a **critical resource**
- Growing impact on **performance**
- **Data movements :** speed gap CPU / RAM, **memory wall**.
- **Management** : now have to handle close to **TB** of memory
- *Decreasing* ***per compute*** *(cores)* ***power ?***

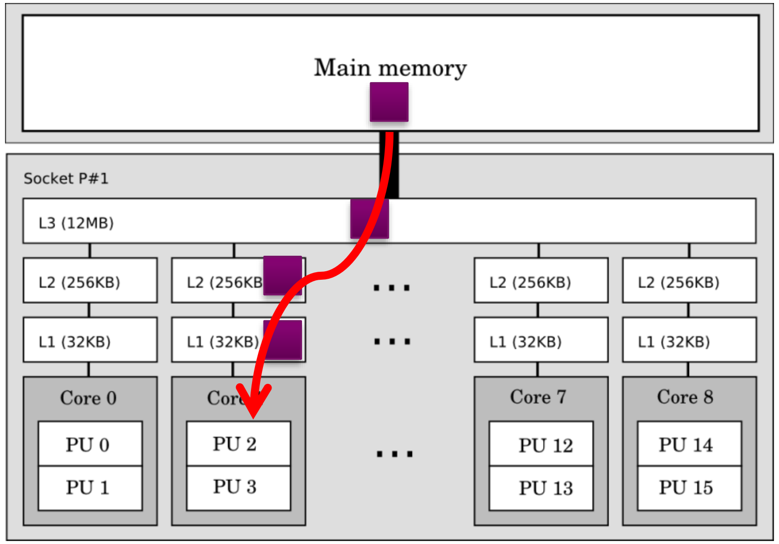
http://www.cea.fr/multimedia/Pages/galeries/defense/Tera-100.aspx

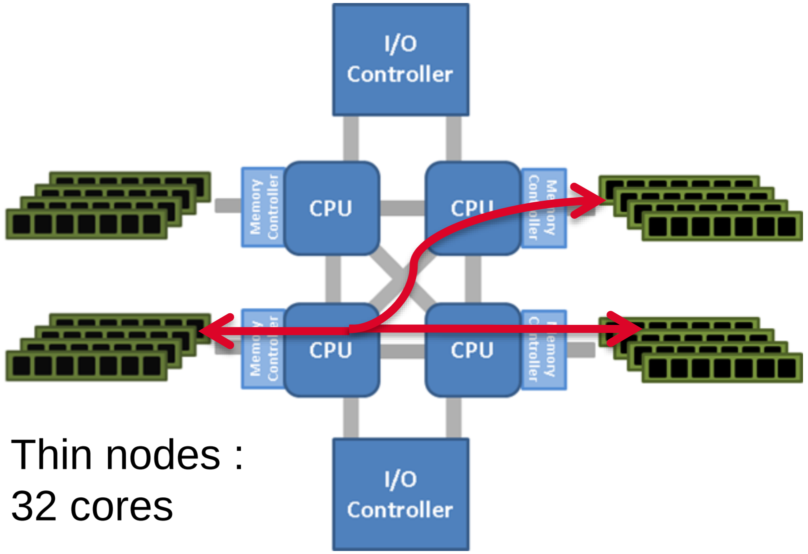


https://de.wikipedia.org/wiki/Datei:PS2_RAM_Module.jpg

Complex memory hierarchies

- Caches
- NUMA
- MCDRAM ?
- NVMe DIMMs ?

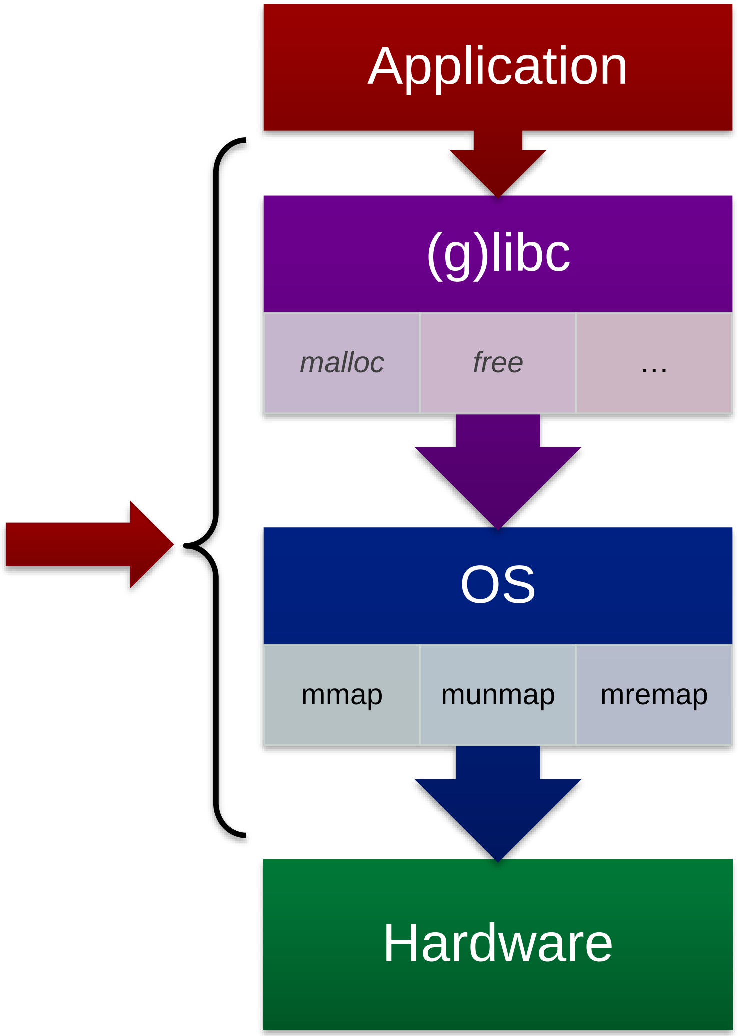




Thin nodes :
32 cores

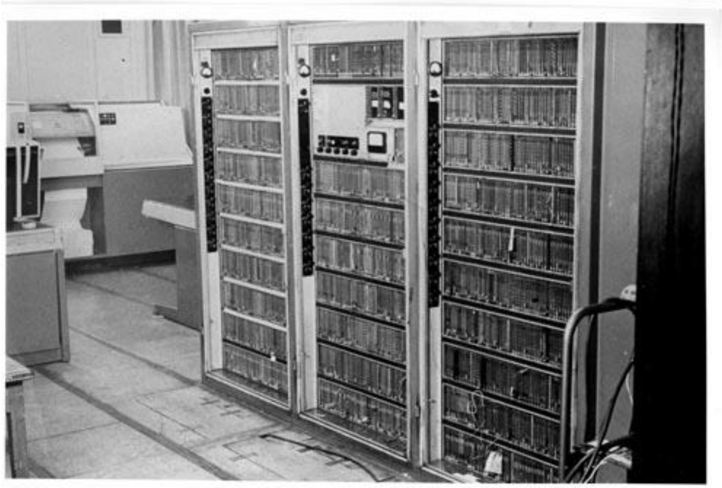
Software memory management layer

- **Impact of memory management mechanisms ?**
- Involving **two components** :
  - User space ***memory allocator*** (malloc)
  - **Operating System** (OS)
- Focus on :
  - Impact on **allocation time**
  - Impact on **access efficiency** (placement)

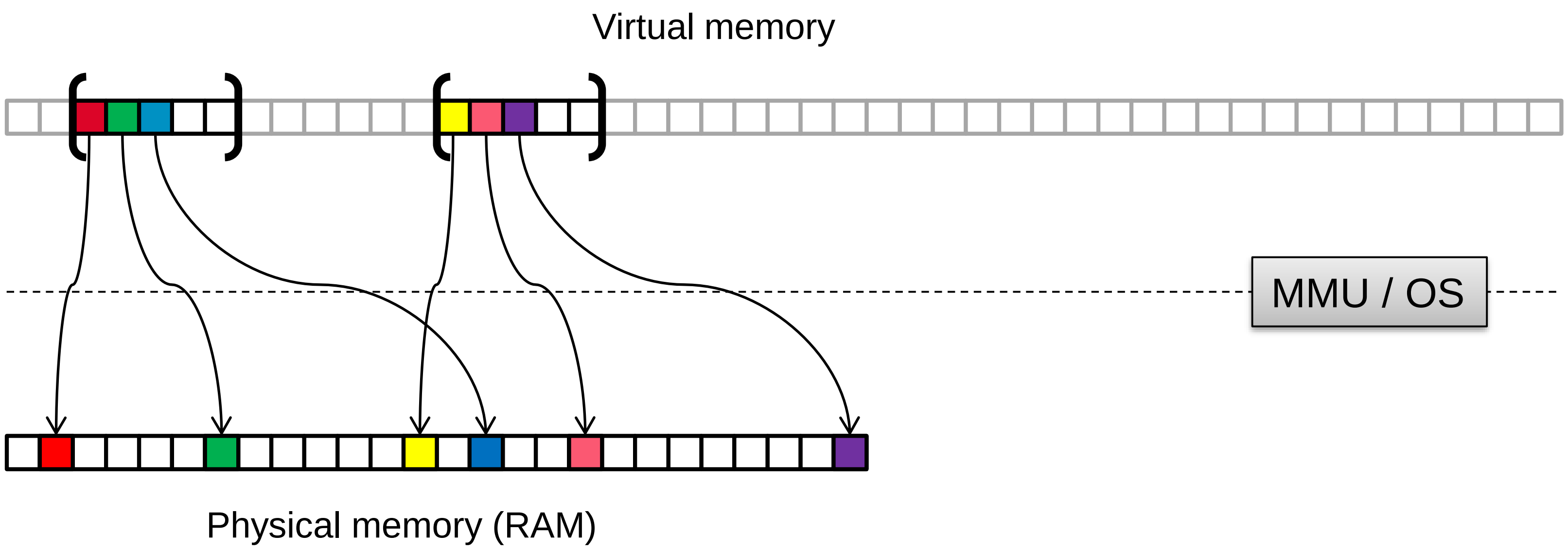


OS virtual / physical address spaces

- Two address spaces : **physical** + **virtual**
- **Paging** was first used in **1962** on the **ATLAS computer**
- **Area** creation with syscalls : **mmap / munmap / mremap**
- **Malloc** has the responsibility to **hide the pages to developers**

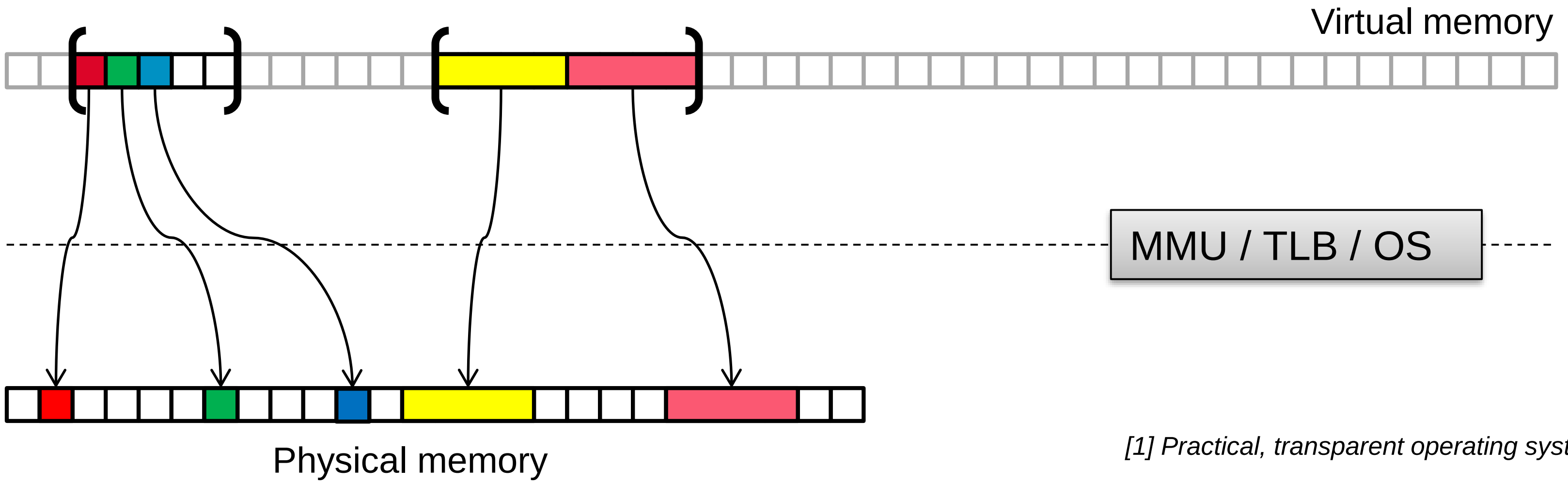


http://www.computerhistory.org/collections/catalog/102698470



Huge pages

- **Huge pages : 2 MB**
- **First real support : FreeBSD (superpages, 2002)** [1]
- Support **Linux : ***old HugeTLBfs* then now **Transparent Huge Pages (THP), 2011**

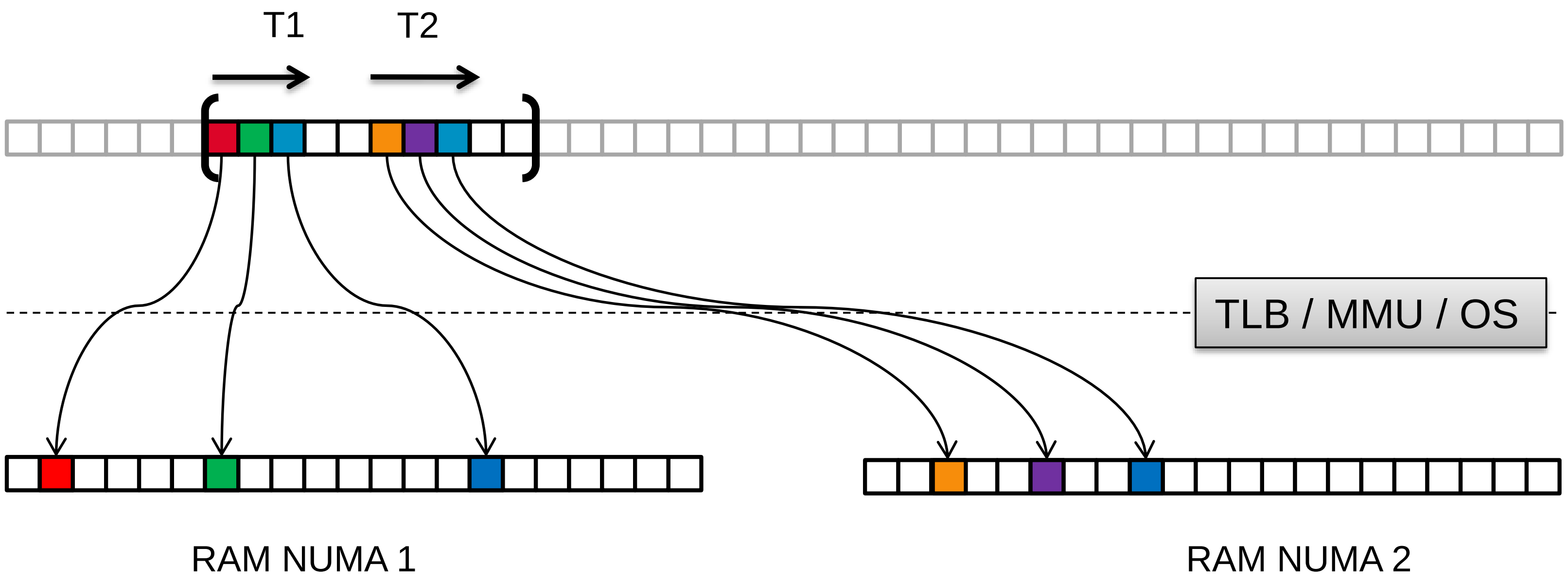


[1] Practical, transparent operating system support for superpages, 2002

Lazy page allocation

- **mmap** creates **pure virtual** area
- First touch creates a **page fault** for each virtual page
- OS provides **physical pages** on **first touch**
- **First touch implicitly** determines **NUMA placement** of the page

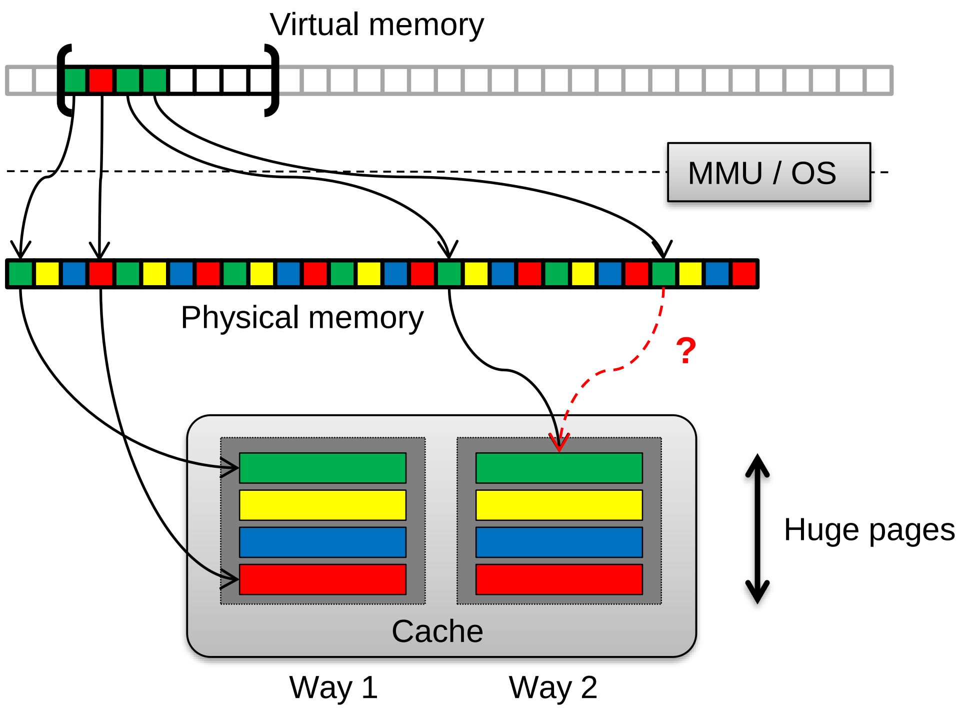
```
ptr = mmap(…,SIZE,…);
#pragma omp parallel for
for (i = 0 ; i < SIZE ; i++)
        ptr[i] = 0;
```

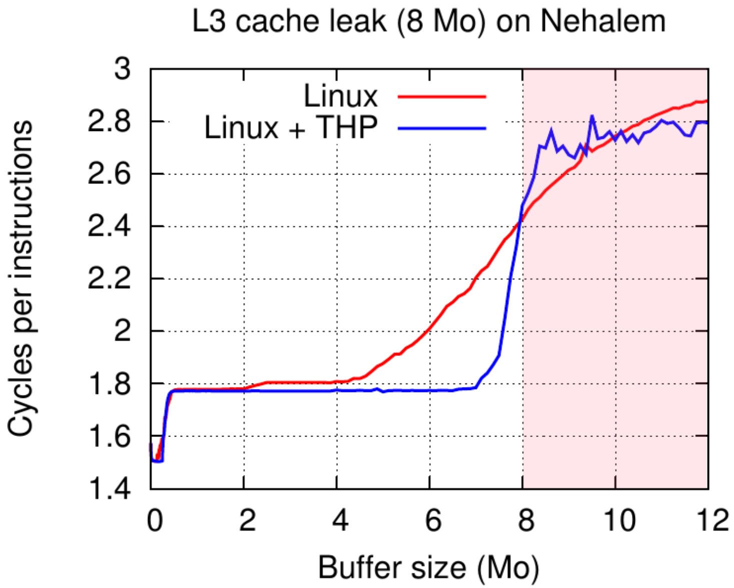


ANALYSIS OF OS PAGING POLICY

Cache associativity

- Data can only be placed in one of the **N lines associated to the address**
- Can create **conflicts** depending on the OS
- Linux "**randomly" chooses** the pages





OS strategies comparison (2010)

- Each **system** has its default paging **strategy**:

OS	Strategy
Linux	4K random
OpenSolaris	Page coloring
FreeBSD	Huge pages

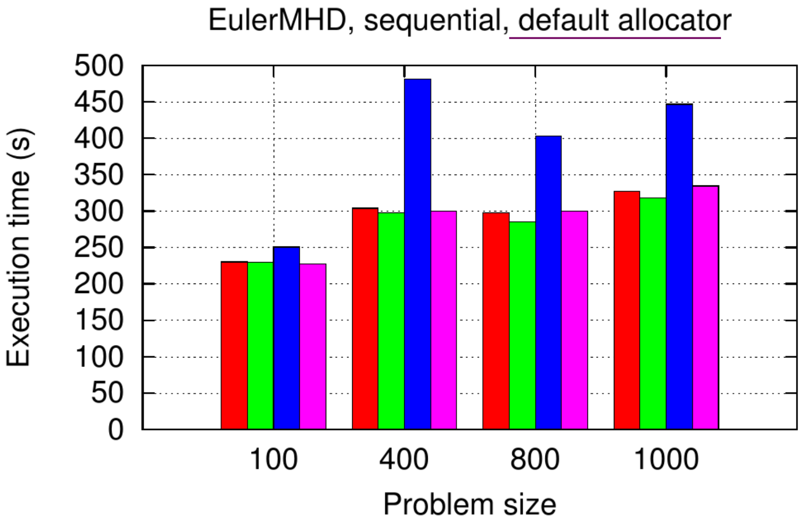
- Is **Linux** slower due to **random paging** ?
- Tested architecture : Intel **Nehalem bi-socket**
- Use a fixed compile chain : **GCC/Binutils/MPI/BLAS**
- **Focus a pathological case**

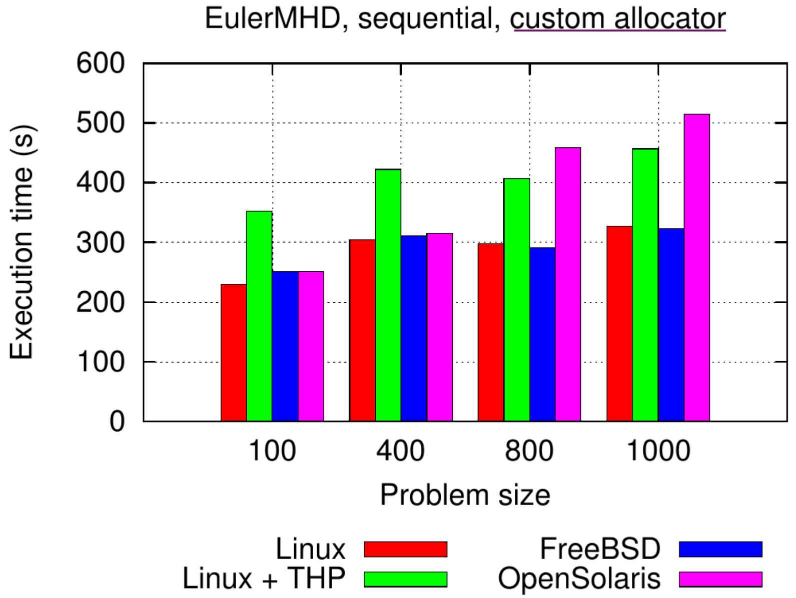
EulerMHD issue

- **EulerMHD (CEA) :**
  - **C++ /MPI**
  - Magnéto-hydrodynamic **stencil code**
- **FreeBSD :** slowdown of **1.5x,** up to **3x** in **parallel**
- Impacted function only do compute.
- Function with **9 arrays pre-allocated** at init. :

```
for (i = 0 ; i < SIZE ; i++)
          x1[i] = x2[i] + x3[i] … + x9[i]
```

- Change between OS's :
  - **User space memory allocator** (malloc).
  - **OS paging policy**
  - ~~*(Scheduler)*~~
- Effect can be controlled by **changing the allocator.**

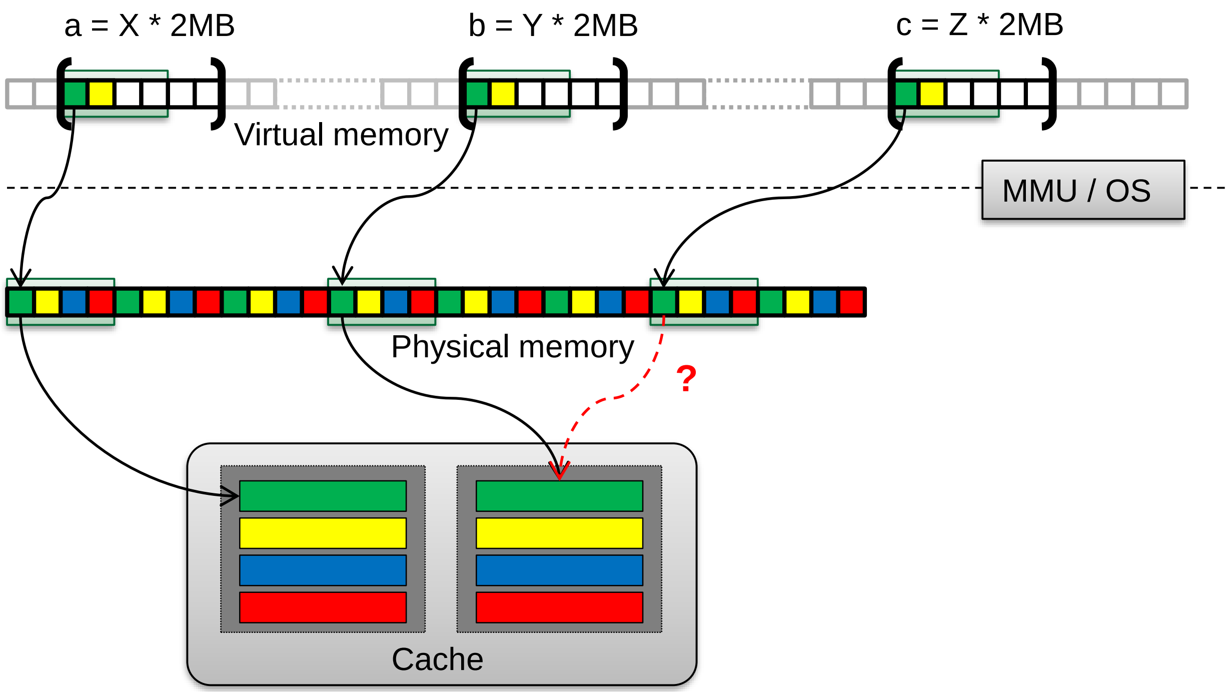




Alignment effect on regular coloring

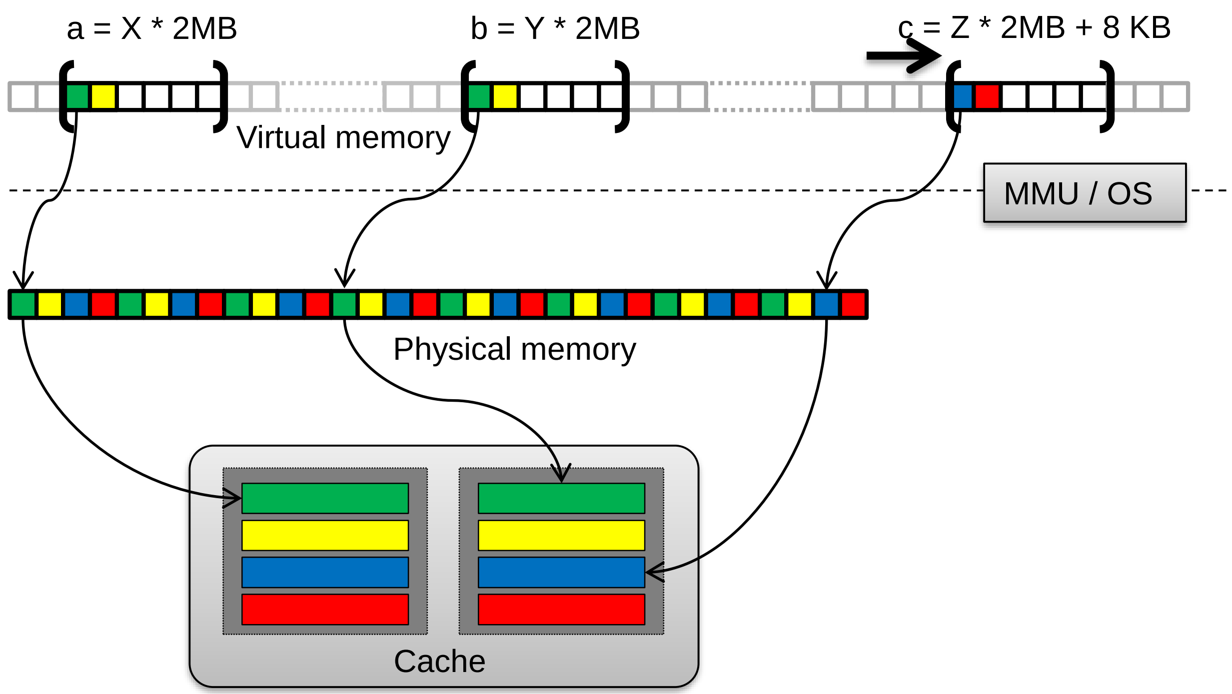
- Each **malloc** (OS) produces different **alignments**
- **FreeBSD** align **large segments** on **2 MB**
- **It interferes** with **regular patterns** generated by :
  - OpenSolaris coloration method (modulo)
  - Huge pages

```
for (i = 0 ; i < SIZE ; i++)
        a[i] = b[i] + c[i];
```



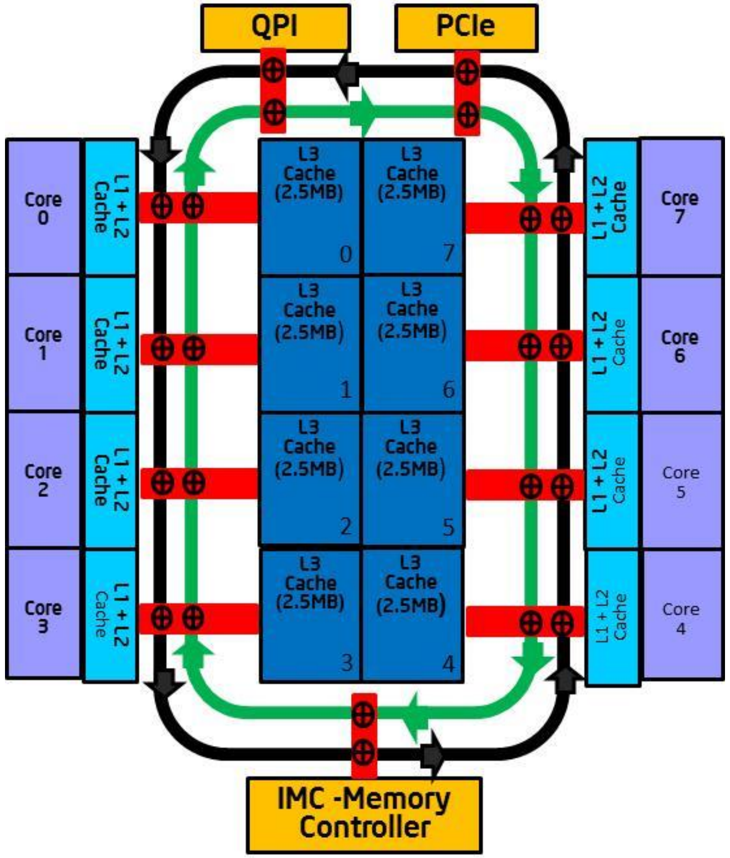
Solution

- Avoid segment **alignments** on **cache way size** (mmap / malloc).
- The **Linux random** approach **prevents pathological cases**



New intel L3 cache slices

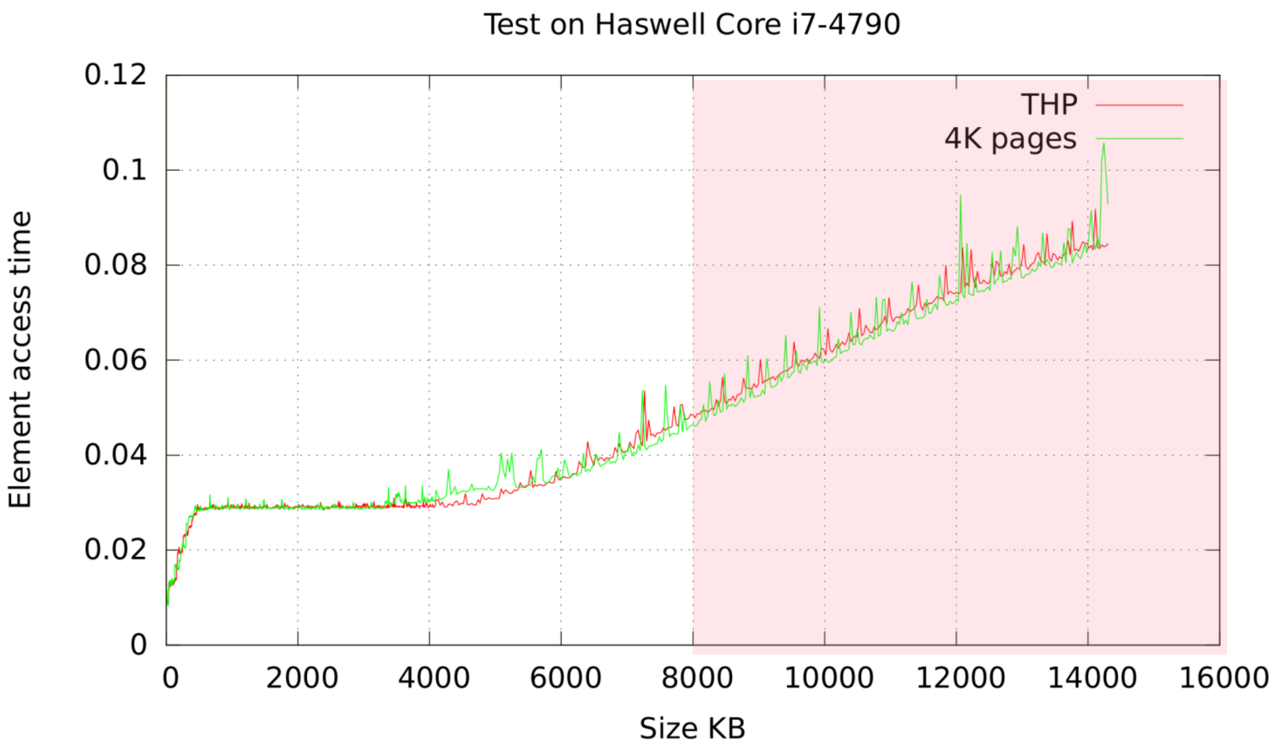
- Since **Sandy Bridge**
- L3 splits in **slices**
- **Slice** is selected by **hashing the address**
- **Each slice** has associativity with **16 ways**
- This **fix** the **coloring/alignment issue**



https://software.intel.com/en-us/articles/intel-xeon-processor-e5-26004600-product-family-technical-overview

On today CPUs

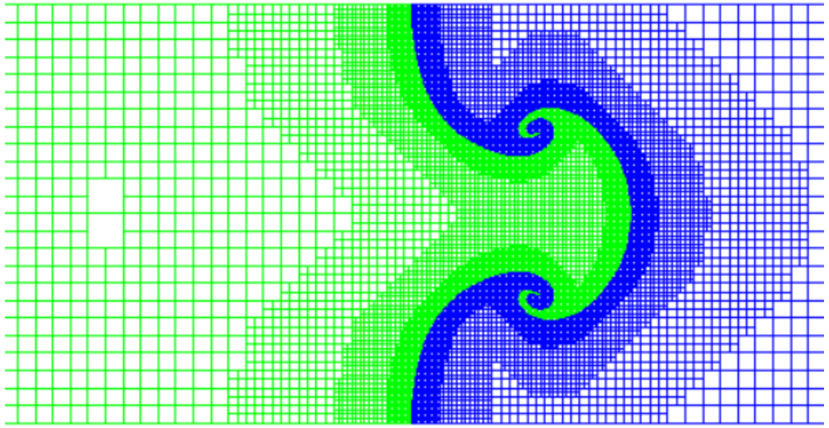
- **Not anymore** an issue **for Intel** L3 caches
  - **Change of topology : slices**
- **AMD Zen (Ryzen)**
  - Now also use **slices**
  - Should solve the issue
- **Still** an issue on **IBM power 8**
  - L3 cache has 8 ways for 8 MB
  - **Issue present**
  - **Power 9 ?** Also “regions” in LLC ?
- For **ARM** (v7/v8) ?
  - **L2** shared associative cache
  - Issue should be present
  - But I never tested
- Issue **for L2 of all processors** !
  - Think hyperthreading with 8 ways !

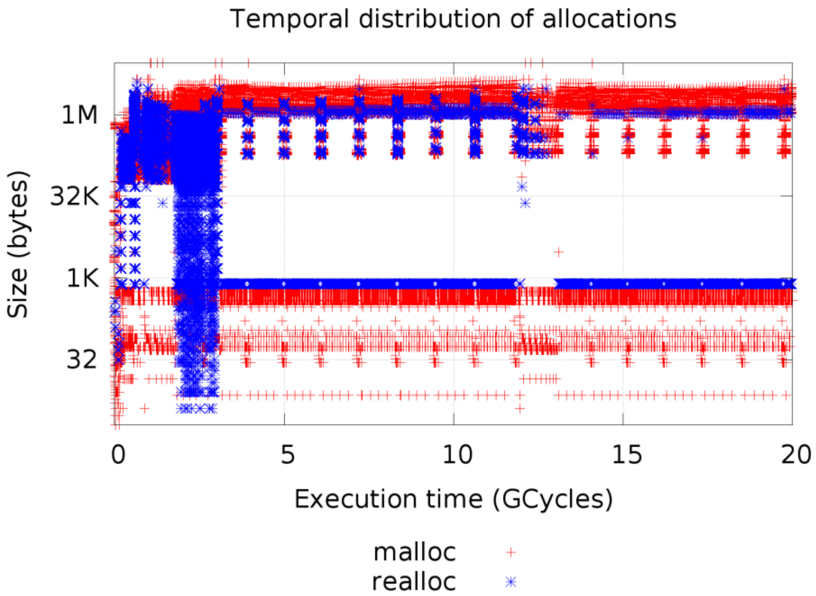


NUMA ALLOCATOR FOR HPC APPLICATIONS

Allocator performance on HPC applications

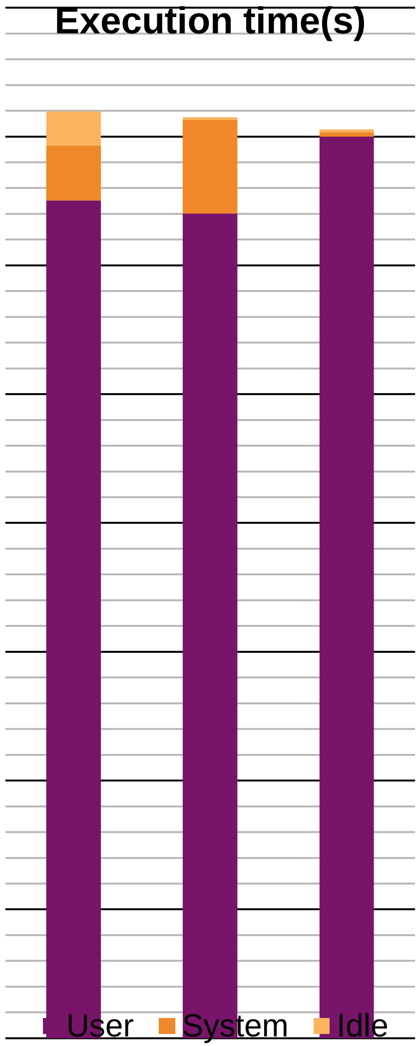
- Main interest : **malloc time cost**
- Test case : **Hera (CEA)**
  - **Adaptive Mesh Refinement** (AMR)
  - **Massive C++/MPI code** (~1 million lines).
- **Large number** of **memory allocations** (~75 millions / 5 minutes on 12 cores)
- **Large number** of **alloc/realloc** around **~20 MB**
- **Available allocators :**
  - **Doug Lea** / **PTMalloc** : libc Linux
  - **Jemalloc** : FreeBSD / Firefox / Facebook
  - **TCMalloc** : Google
  - *Hoard*

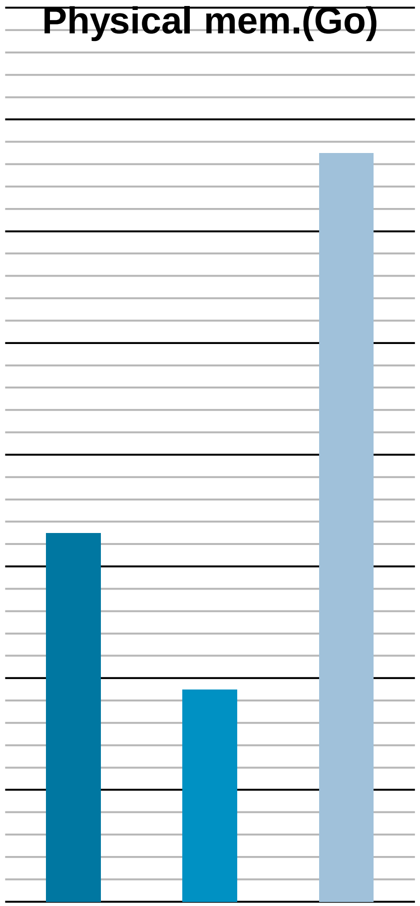




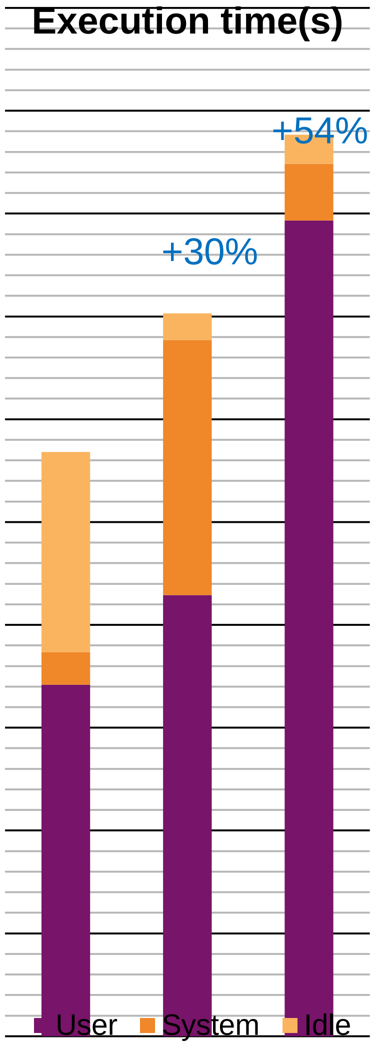
Hera preliminary results

12 cores





128 cores



How to measure malloc time

- Measurement method :

```
T0 = clock_start();
ptr = malloc(SIZE);
T1 = clock_end();
```

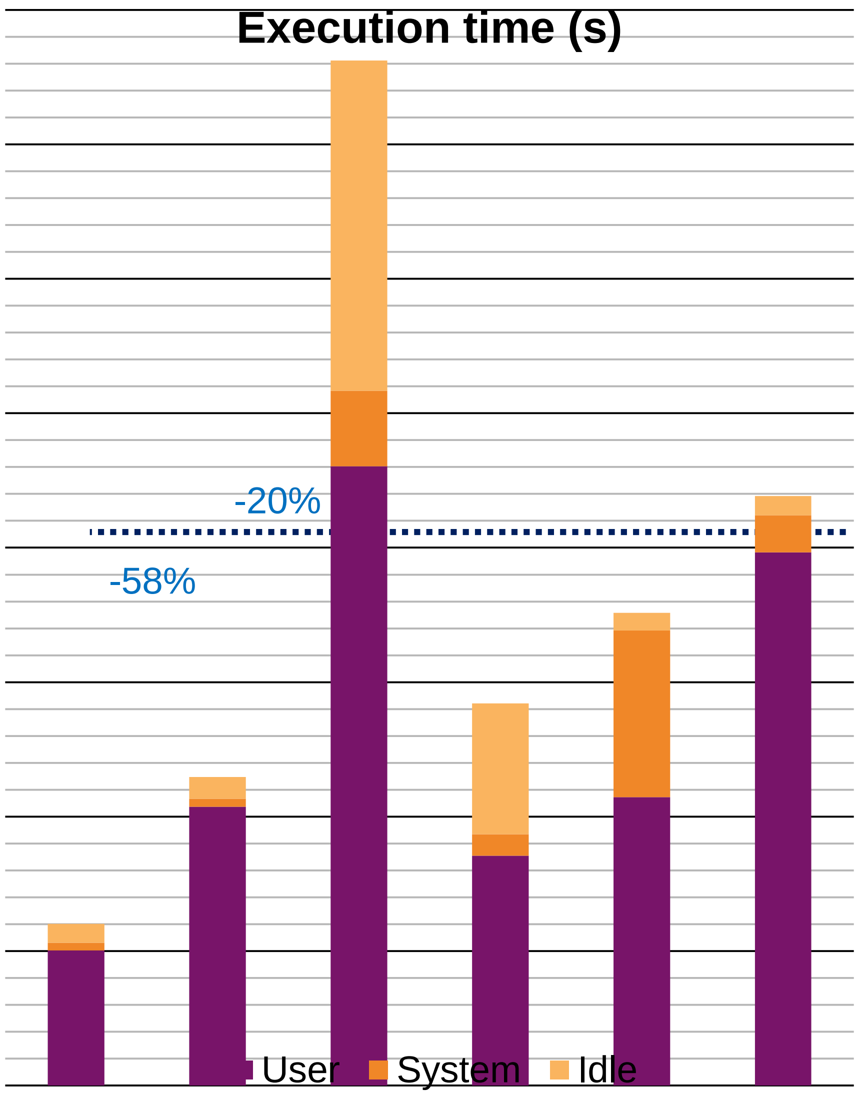
- Ok for **small blocks**, but not for **large** one :

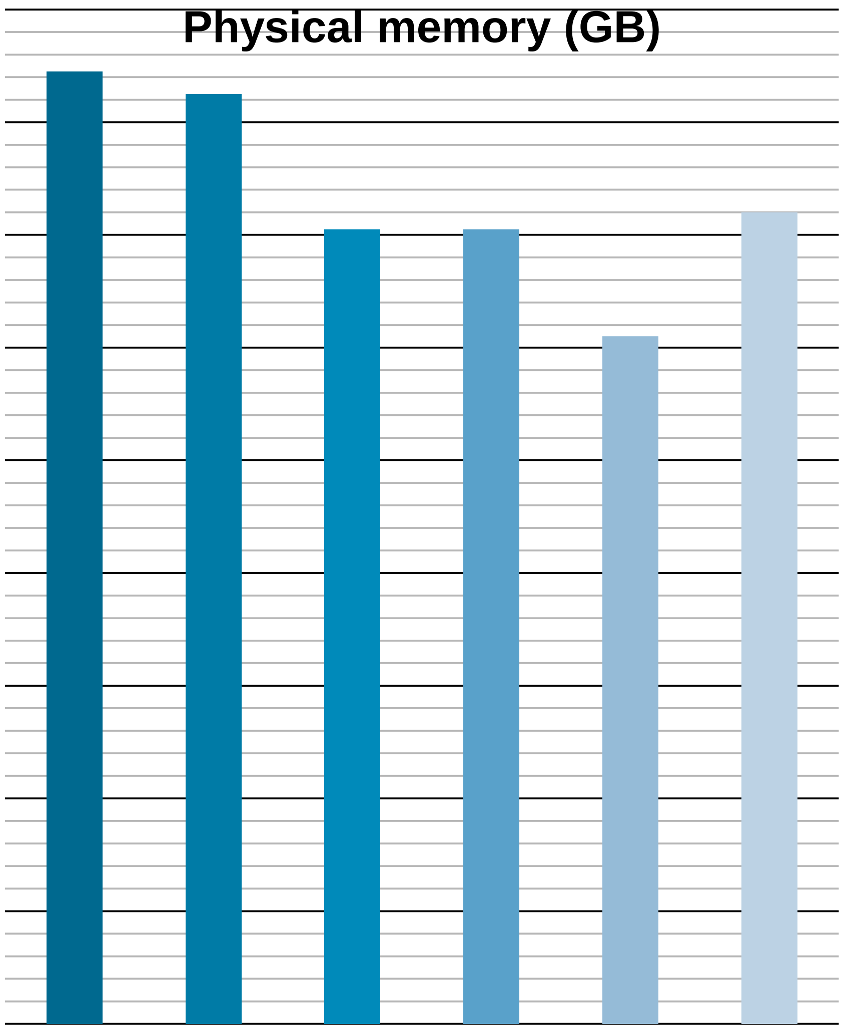
```
T0 = clock_start();
ptr = malloc(SIZE);
for ( i = 0 ; i < SIZE ; i += PAGE_SIZE)
         ptr[i] = 0;
T1 = clock_end();
```

- **Lazy page allocation**.
- **Page faults** on first access.

For 4GB	Malloc	First access
Time (M cycles)	0,008	1 217

Hera on Nehalem-EP (128 : 4*4*8 cores)





COST OF FIRST TOUCH HANDLER

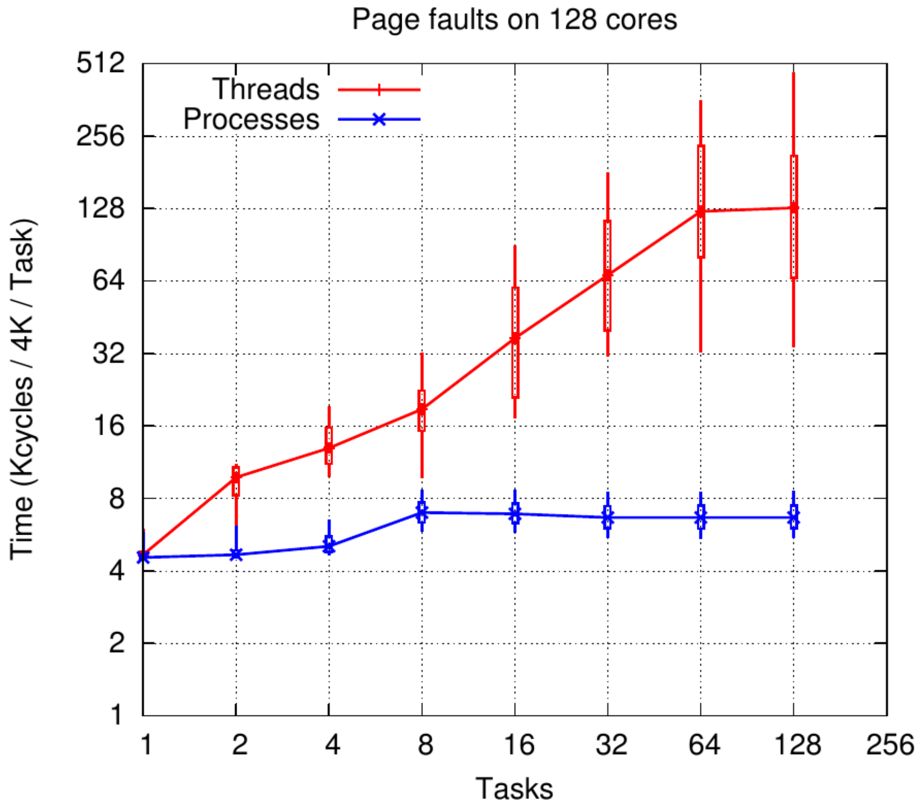
Benchmarking page faults

- **Page faults** are an issue for **allocation performance**
- We **previously limit them** with **large segment recycling**
- Can we **improve fault performance** of **large allocations**?
- **Micro-benchmark** :

```
ptr = mmap(SIZE);
#pragma omp parallel for
for ( i = 0 ; i < SIZE ; i += PAGE_SIZE)
{
        TIME_DISTRIBUTION(ptr[i] = 0);
}
```

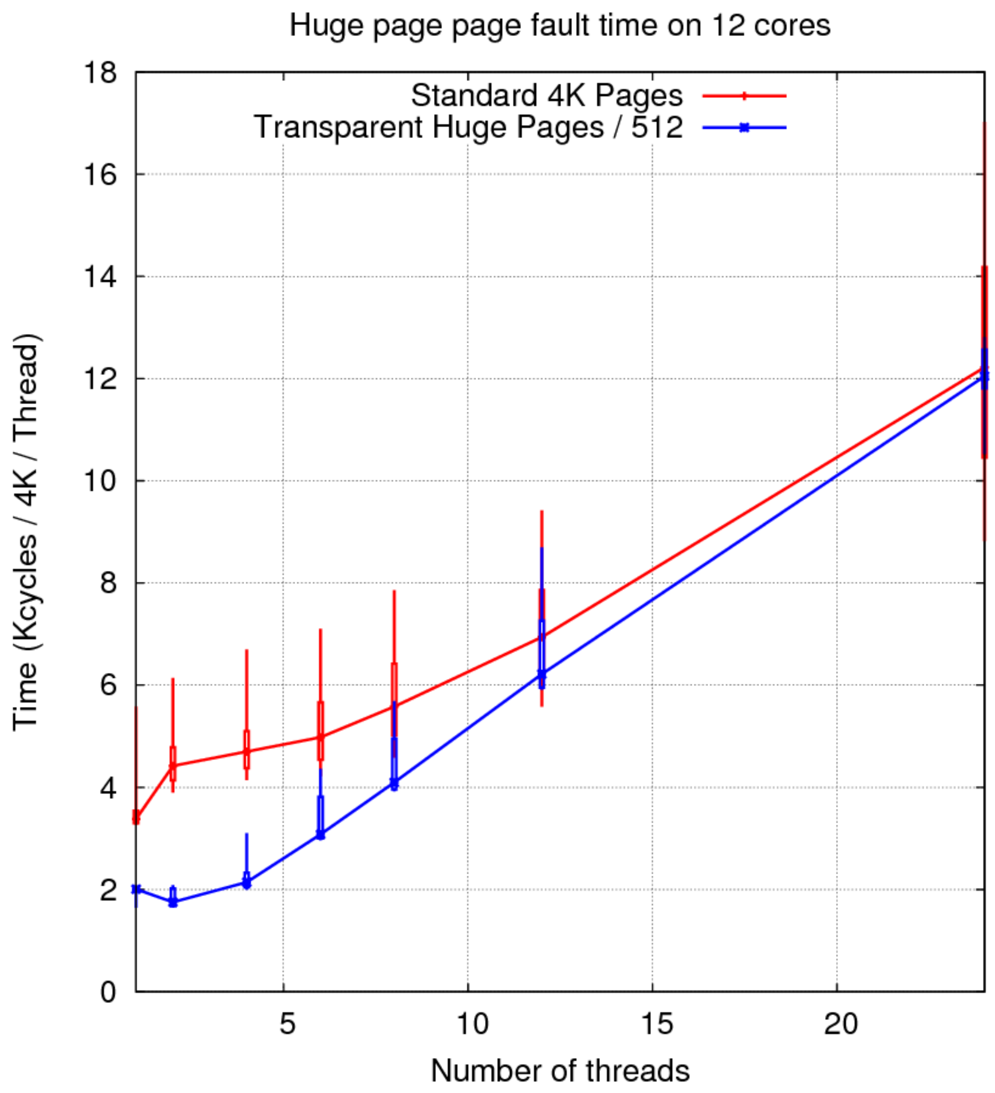
Page fault scalability

- **Are page faults scalable** ? Over threads or processes.
- Mesurement on **4*4 Nehalem-EP** (128 cores) and on **Xeon Phi** (60 cores)
- **Get scalability issue !**



Can huge pages solve this issue ?

- Standard pages: **4K**
- Huge pages (x86_64): **2M**
- **Divide number of faults by 512**
- Impact on performance ?
  - Sequential : **only 40%**
  - Parallel **: No**
- **Why ?**



What happens on first touch page fault ?

- Hardware generates an interruption to the OS
- Take **locks** on **page table**
- Check reason of the fault
- Is **first touch** from **lazy allocation**
- Request a free page to NUMA **free lists**
- **Clear the page content**
- Map the page, update the **page table**
- **Release locks**

Possible issue on large NUMA domains

~1400 / 3400 cycles 40%
99% for THP !

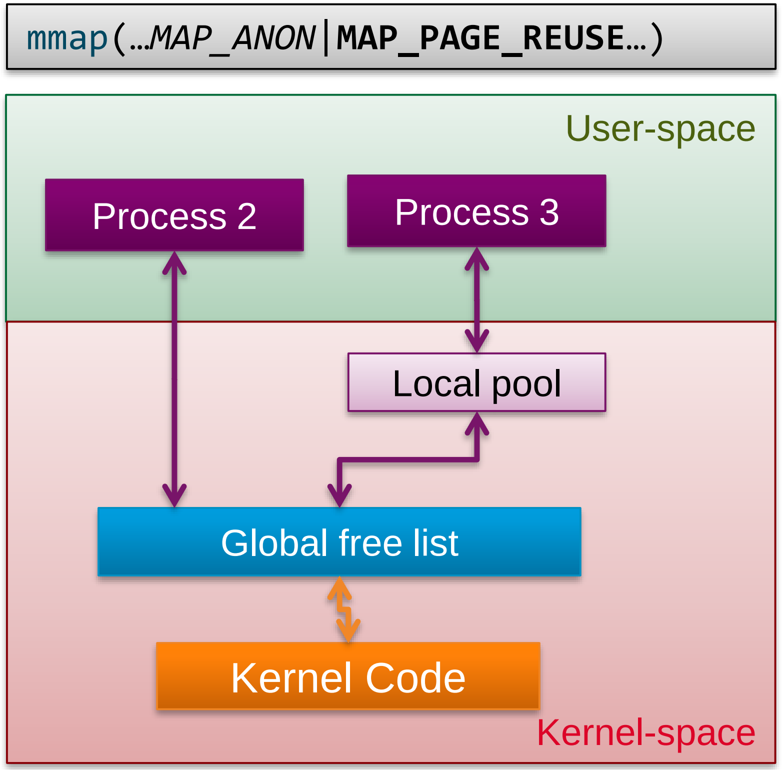
Locks, but hard to fix
(some work from
A.T. Clement ASPLOS12)

How to avoid page zeroing cost ?

- Microsoft approach **:**
  - **Windows** uses a **system thread** to clear the memory
  - So its done **out** of **critical path**

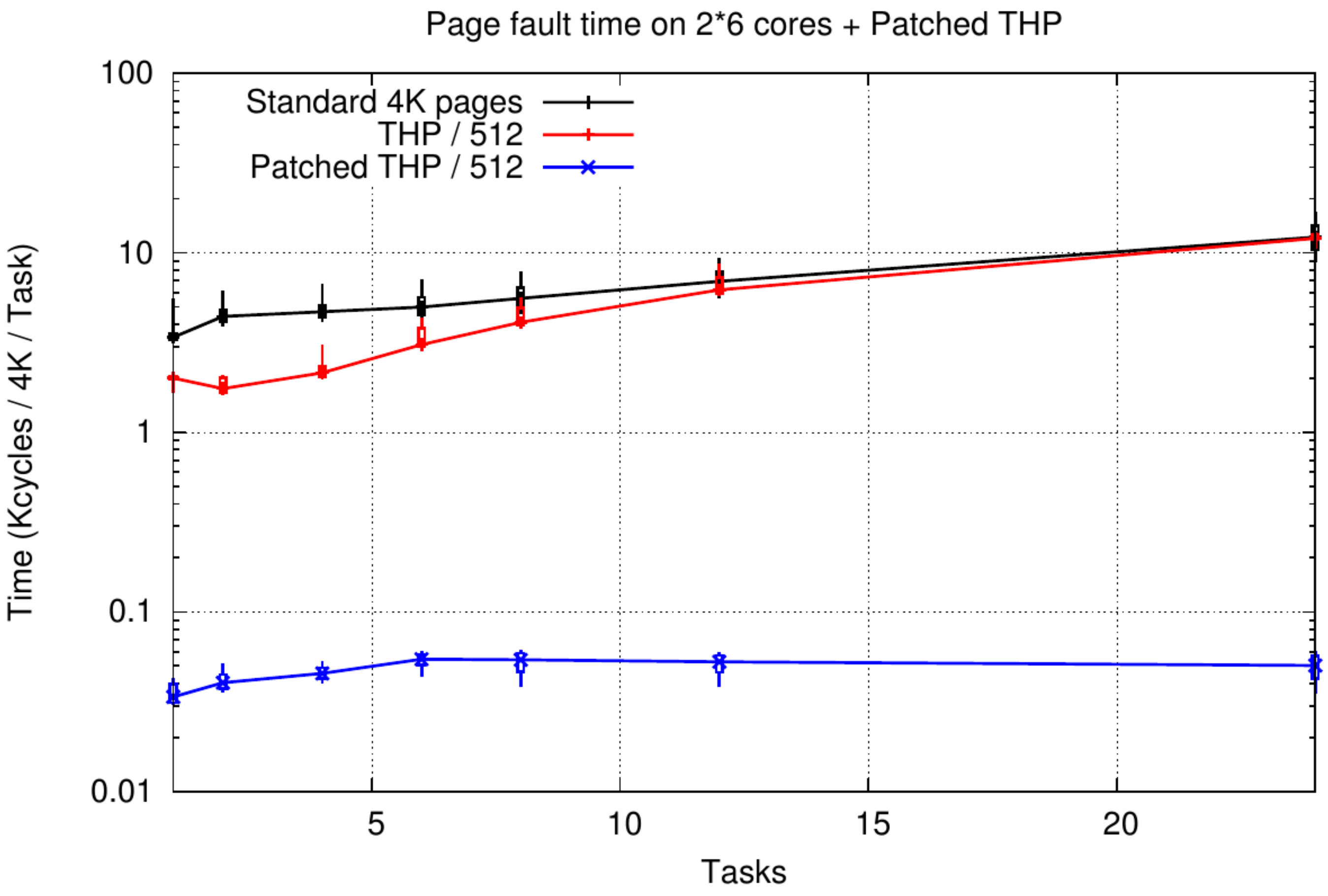
- But **zeroing**:
  - Implies **useless work**
  - Consumes CPU **cycles** so **energy**
  - Consumes **memory bandwidth**

- Why not **avoid them ?**
  - **Skip them (security ?)**
  - Use a **per process memory** in **kernel (published)**
  - Do in DIMM hardware



Performance impact on huge pages

- **Huge pages** (2 MB) faults become **47** times faster, **60** in parallel.
- **New interest** for huge pages.



MALT - A MALLOC TRACKER

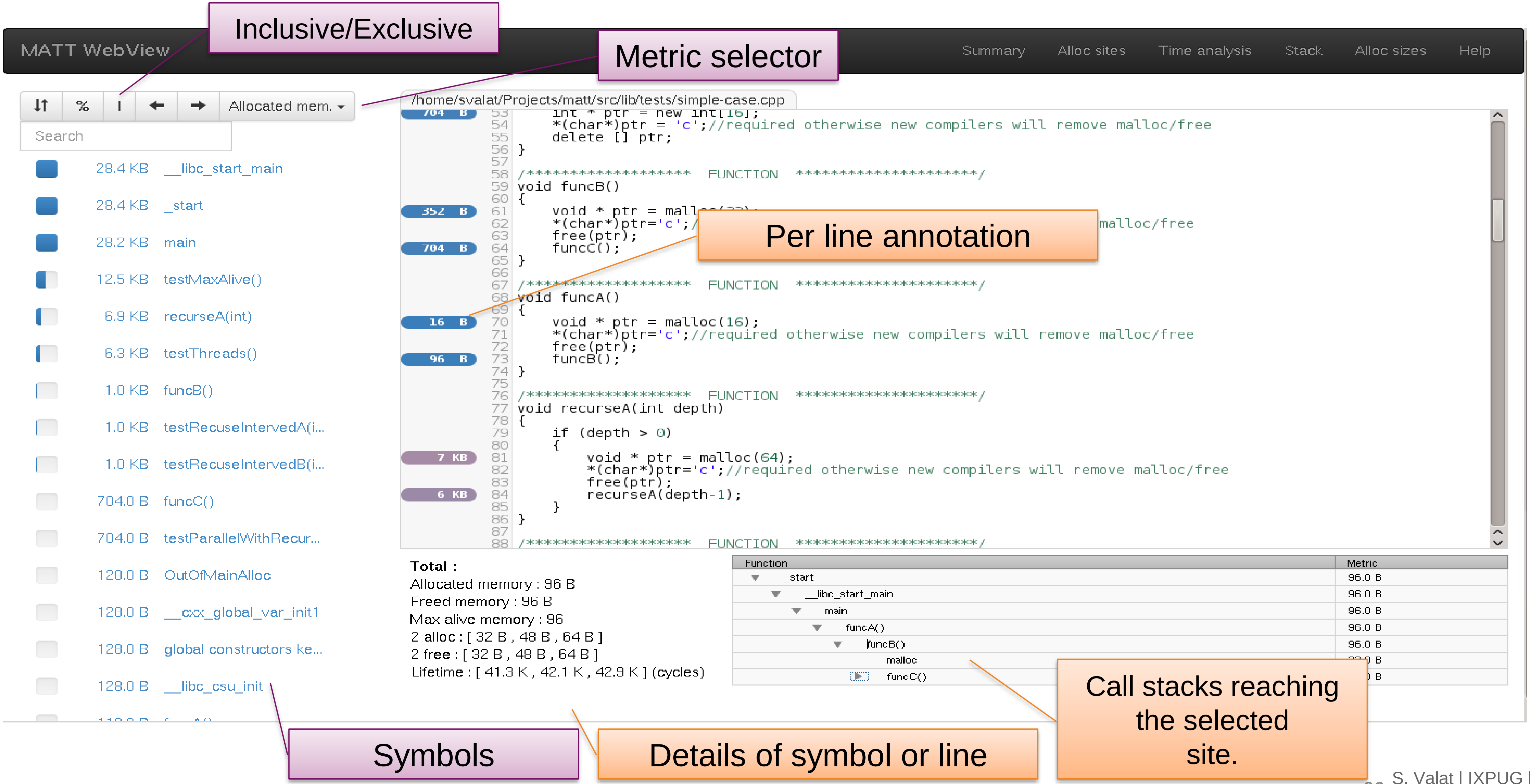
The question

- We want to point :
  - **Where** memory is allocated.
  - **Properties** of allocated chunks.
  - **Bad** allocation **patterns** for performance.

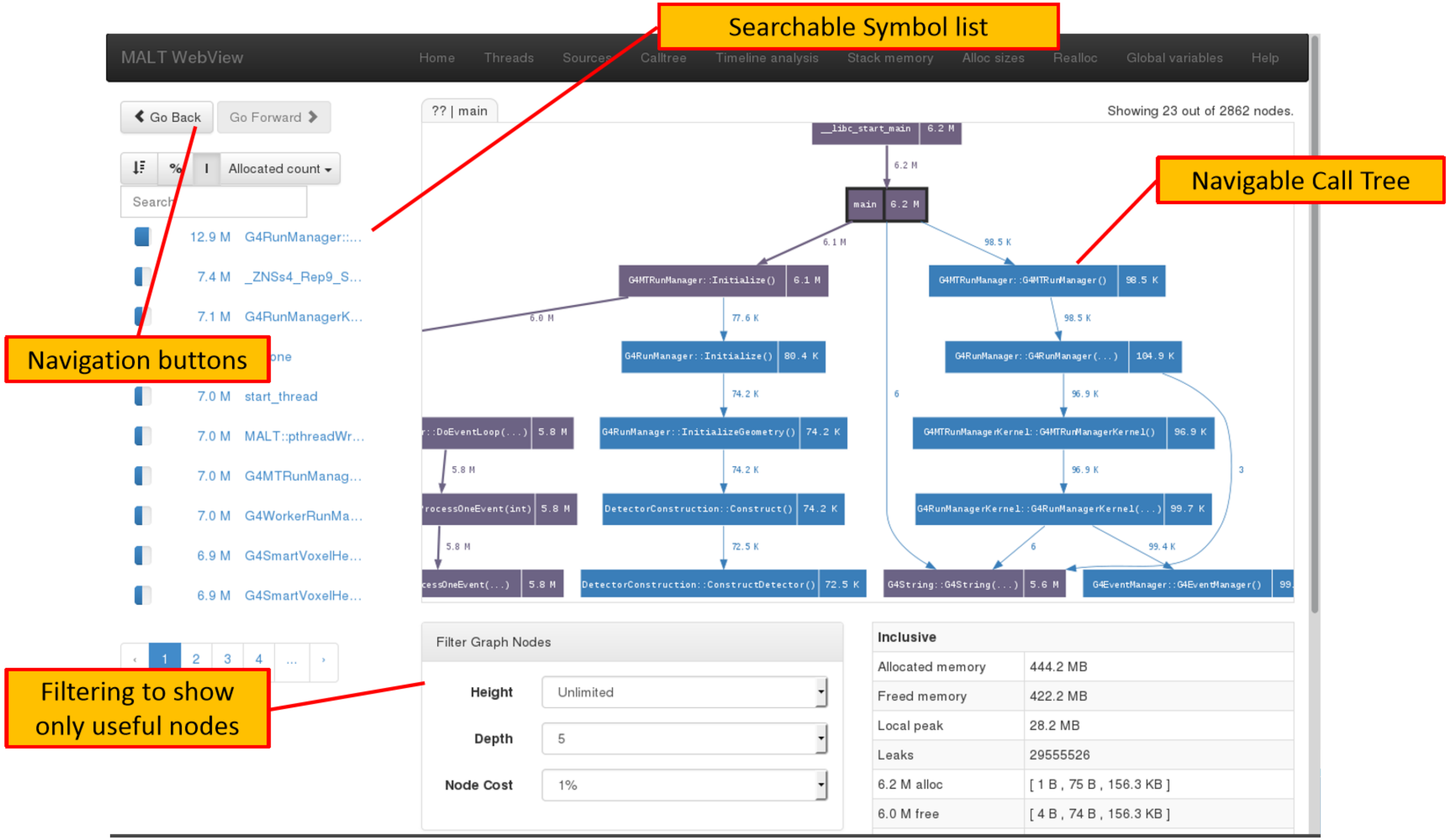
```
__thread Int gblVar[SIZE];
int * func(int size)
{
        child_func_with_allocs();
        void * ptr = new char[size];
        double* ret = new double[size*size*size];
        for (auto it : list)
        {
                double* buffer = new double[size];
                //short and quick do stuff
                delete [] buffer;
        }
        return ret;
}
```

- Global variables and TLS
- Indirect allocations
- Leak
- Might lead to swap for large size
- “compiler added allocations”
- Short life allocations

Source annotations

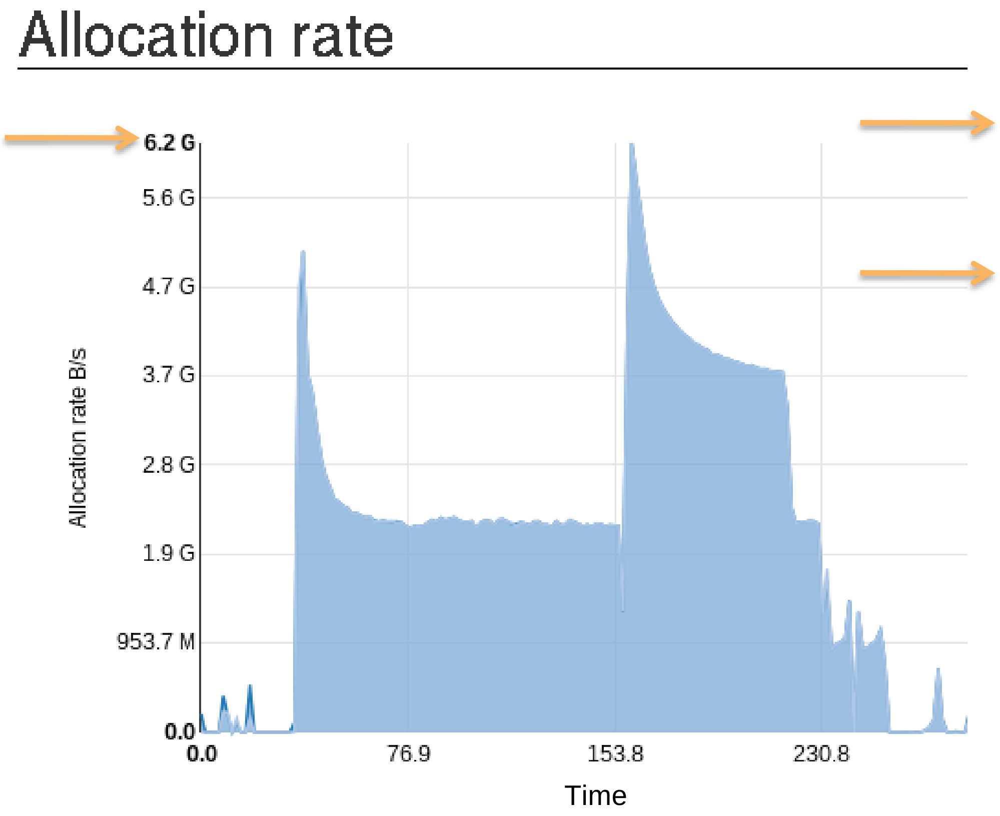


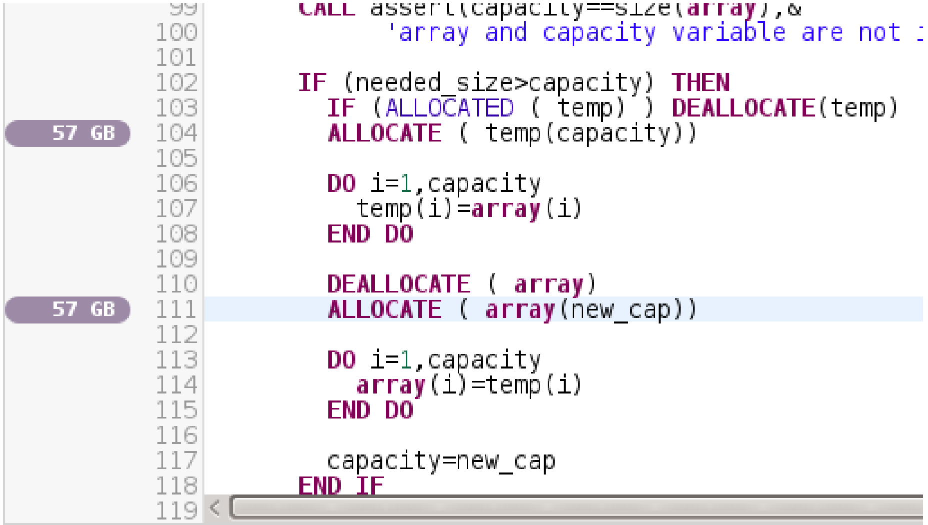
Call tree view



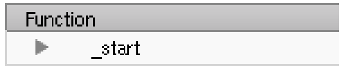
The question

- Issue with **reallocation** on init
- Detected with **allocation rate** & **cumulated allocatated mem.**





Total :
Allocated memory : 56.8 GB
Max alive memory : 135.7 M
3.5 K alloc : [16.0 KB , 16.3 MB , 33.7 MB]
Lifetime : [107.8 K , 26.7 M , 476.7 M] (cycles)



Own :
Allocated memory : 56.8 GB
Max alive memory : 135.7 M
3.5 K alloc : [16.0 KB , 16.3 MB , 33.7 MB]
Lifetime : [107.8 K , 26.7 M , 476.7 M] (cycles)

CERN-IT - MALT, Sébastien Valat

NUMAOROF - A NUMA PROFILER

Typical NUMA example

- Make first **init outside of OpenMP** (in thread 1)

- So **each pages** will be first touched **on NUMA 1**

```
#pragma omp parallel for
for (int i = 0 ; i < SIZE ; i++)
	array[i] = 0;
```

- Then access

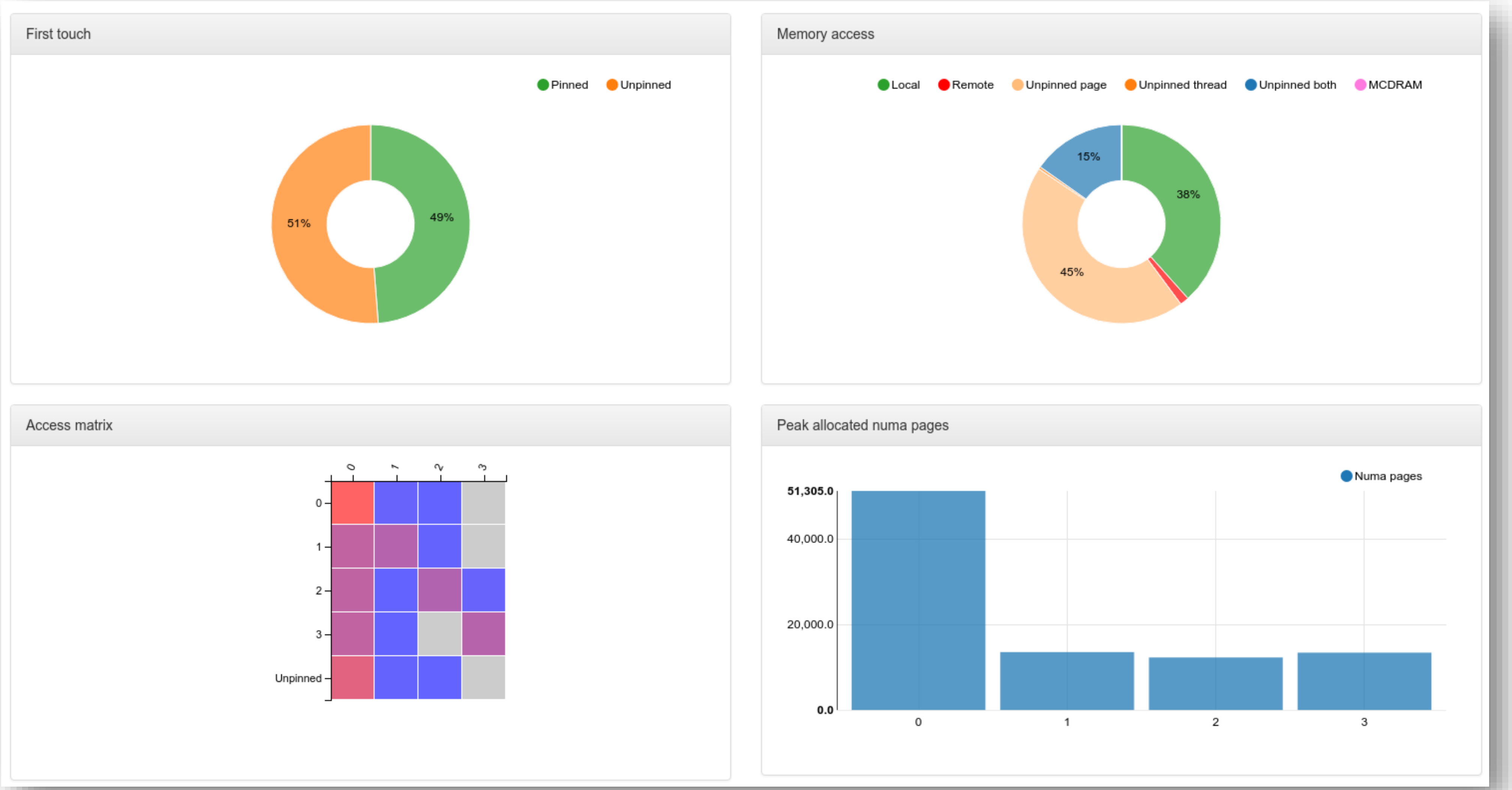
```
#pragma omp parallel for
for (int i = 0 ; i < SIZE ; i++)
	array[i]++;
```

- **Bad performance** due to remote accesses !

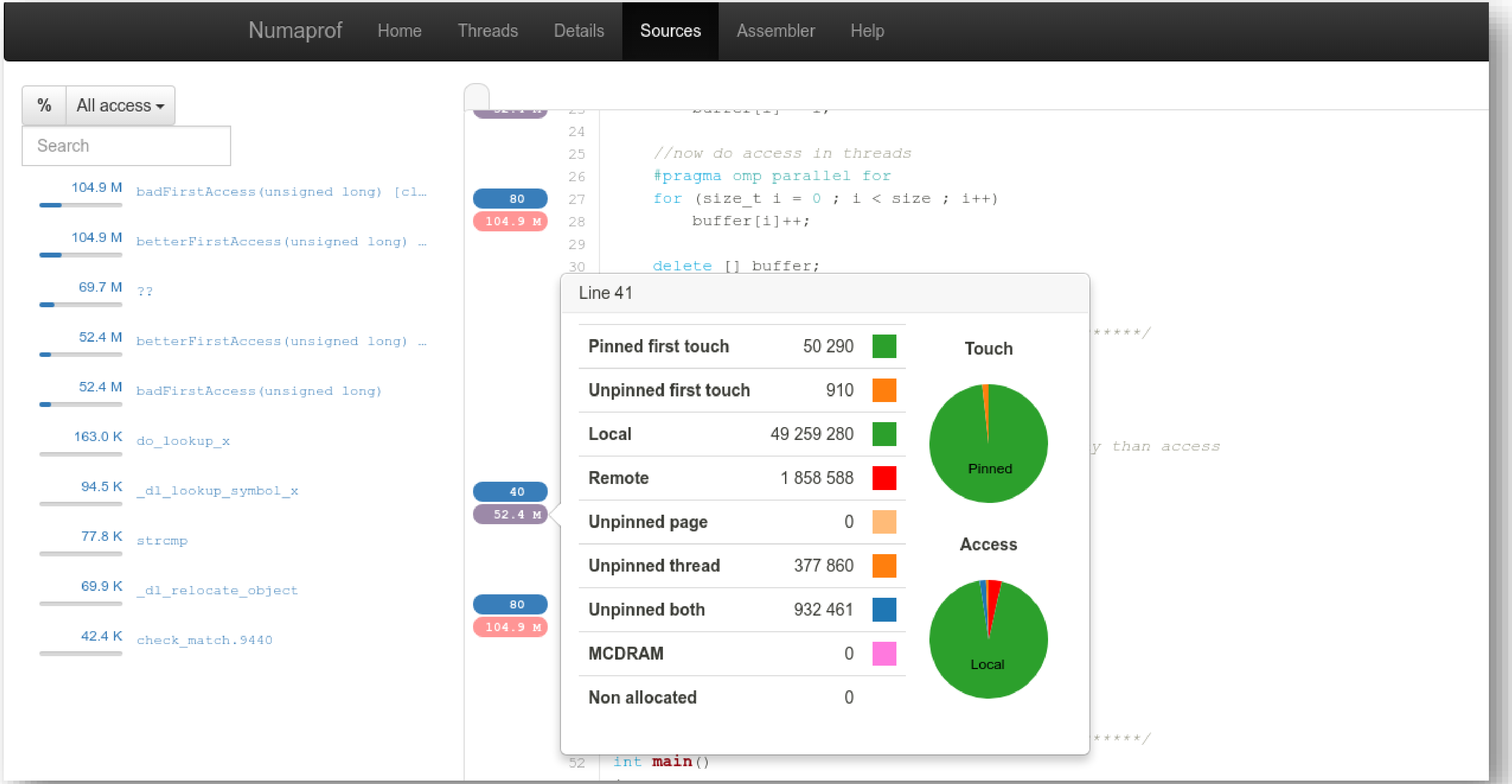
Wish list for a profiling tool…

- We want to know if we make **remote accesses**
- Ideally we need to know **where**…
- We can dream, we want to know **which allocation contain issues**
- We want to know **where** the **first touch** has been done
- On KNL we want to check **MCRAM accesses**

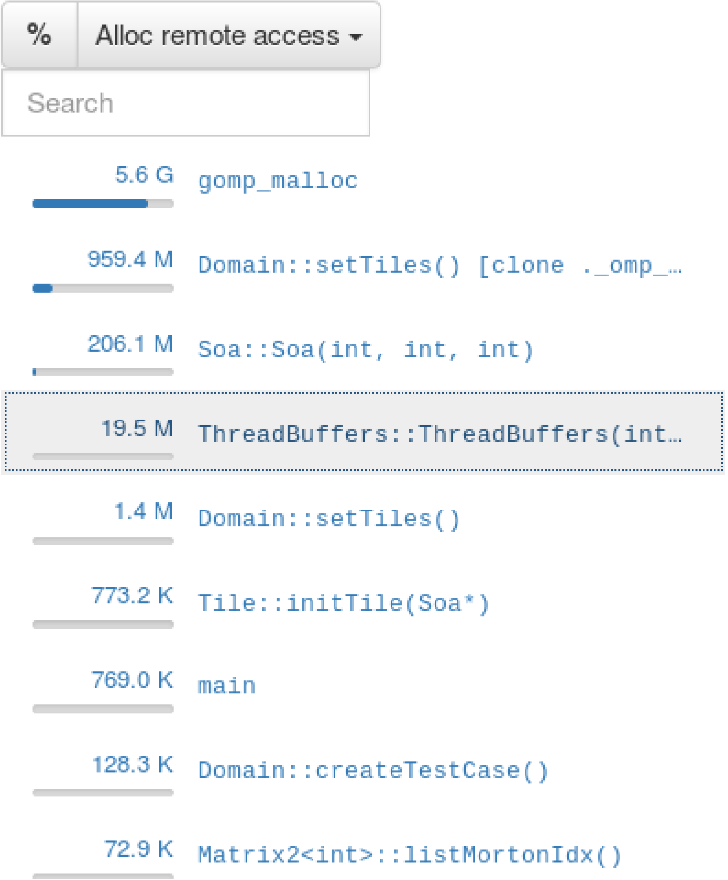
Global summary

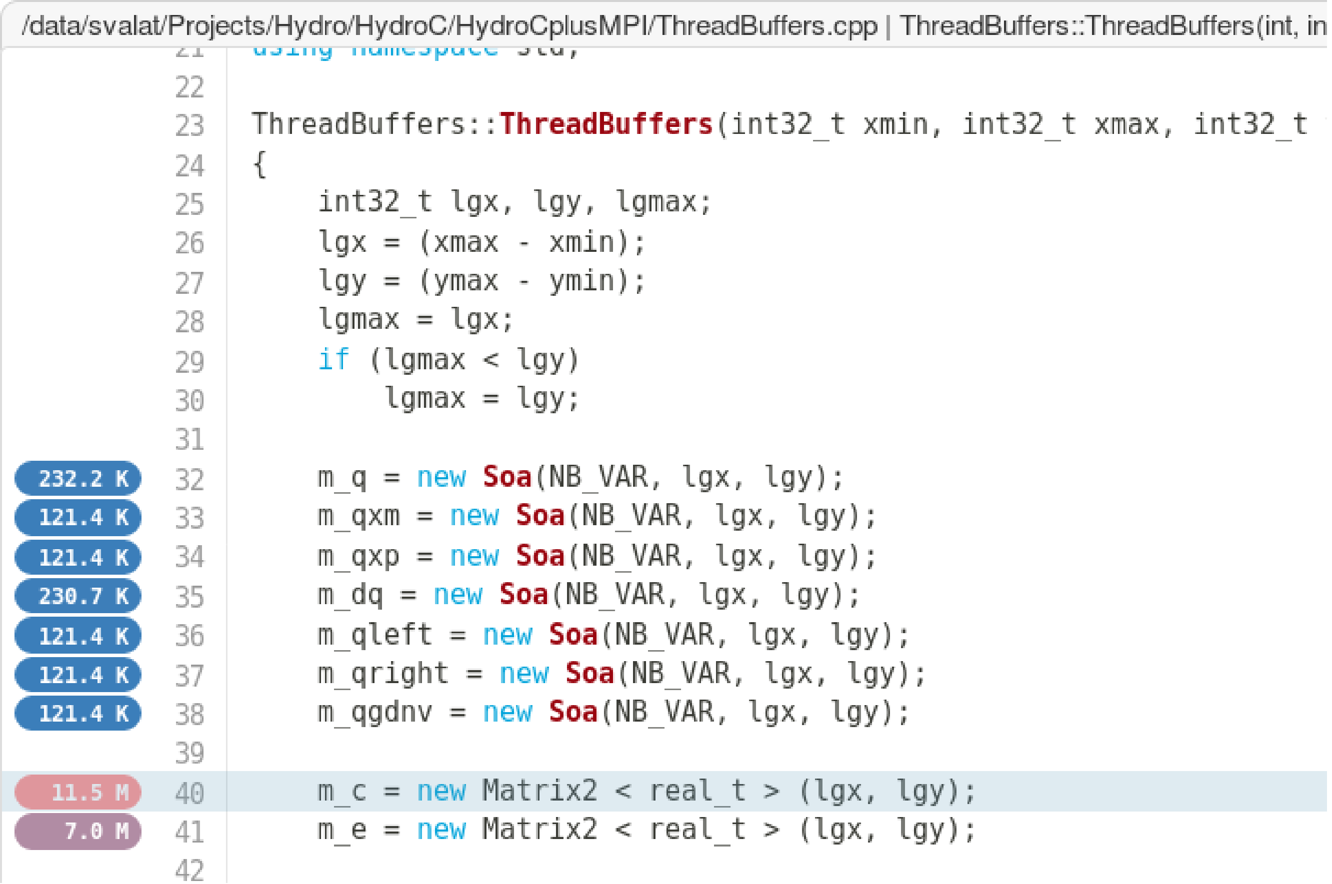


Source & asm annotations

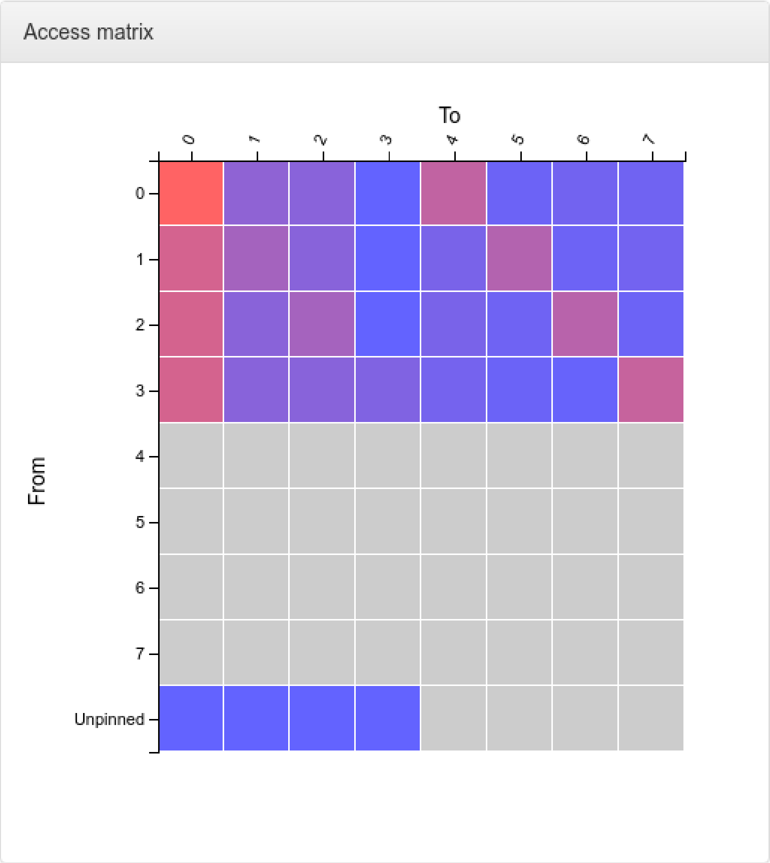


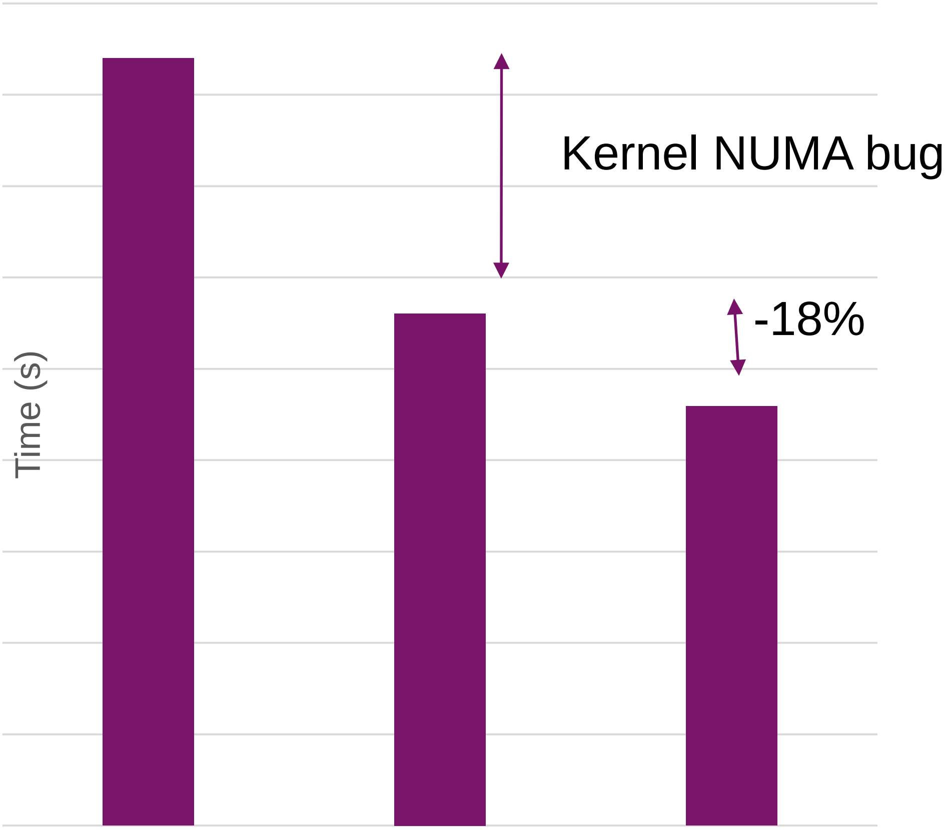
Non parallel allocations





40 minutes optimization on HydroC





CONCLUSION

Conclusion

- Memory is one of the **key for performance**
- **Old management** need to be carefully **looked again**
- Performance **gaps** can be **integer factors**
- Dedicated **tools** can **help a lot**
- It requires **flexible software** to reach **global optimization**
  - Unit tests ?

QUESTIONS ?

On access we need…

- **Intercept** the memory **accessse (Intel PIN)**
- **Track thread location**
- **Intercepts malloc**
- We can **skip** accesses to **local stack**
  - overhead 80x -> 40x
- **Overhead** on 256 KNL threads : **60x**

Ideal view of HPC memory management stack

Apply MAMA allocator approch

↓

Jemalloc

Select arena with NUMA	~~No 4K / 2M alignements~~

↓

Memory sources

NUMA + recycling	Calloc move_pages optim.	Dynamic adaptation	Free pages with madvise

↓

Huge pages

Zeroing patch

↓

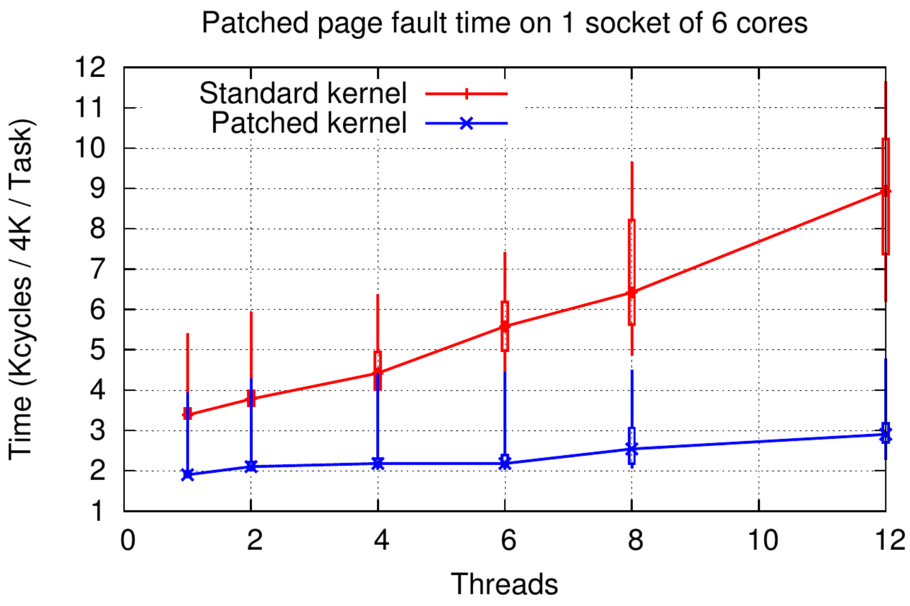
Hardware

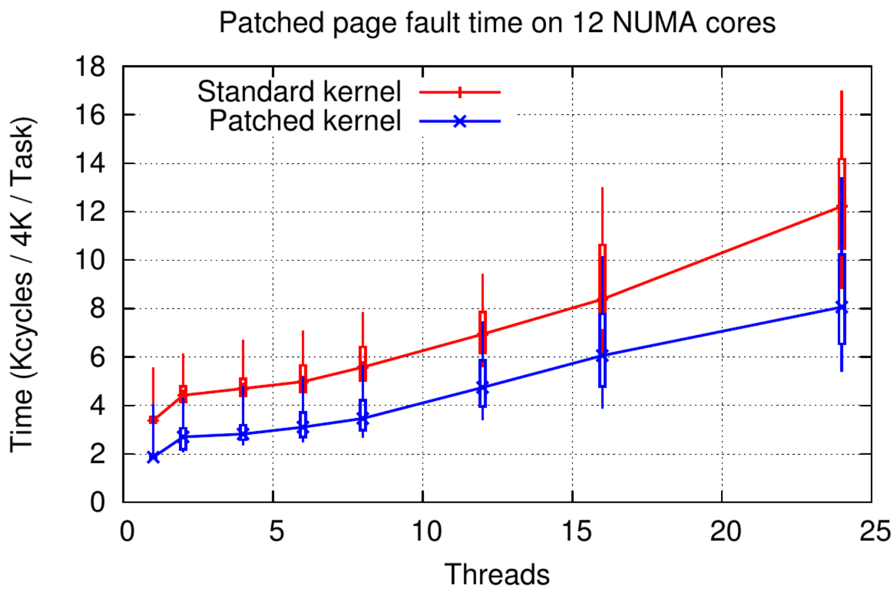
Zeroing by RAM	~~Mixing inside huge pages~~	Smaller huge pages (256K ?)

BACKUPS

Performance impact

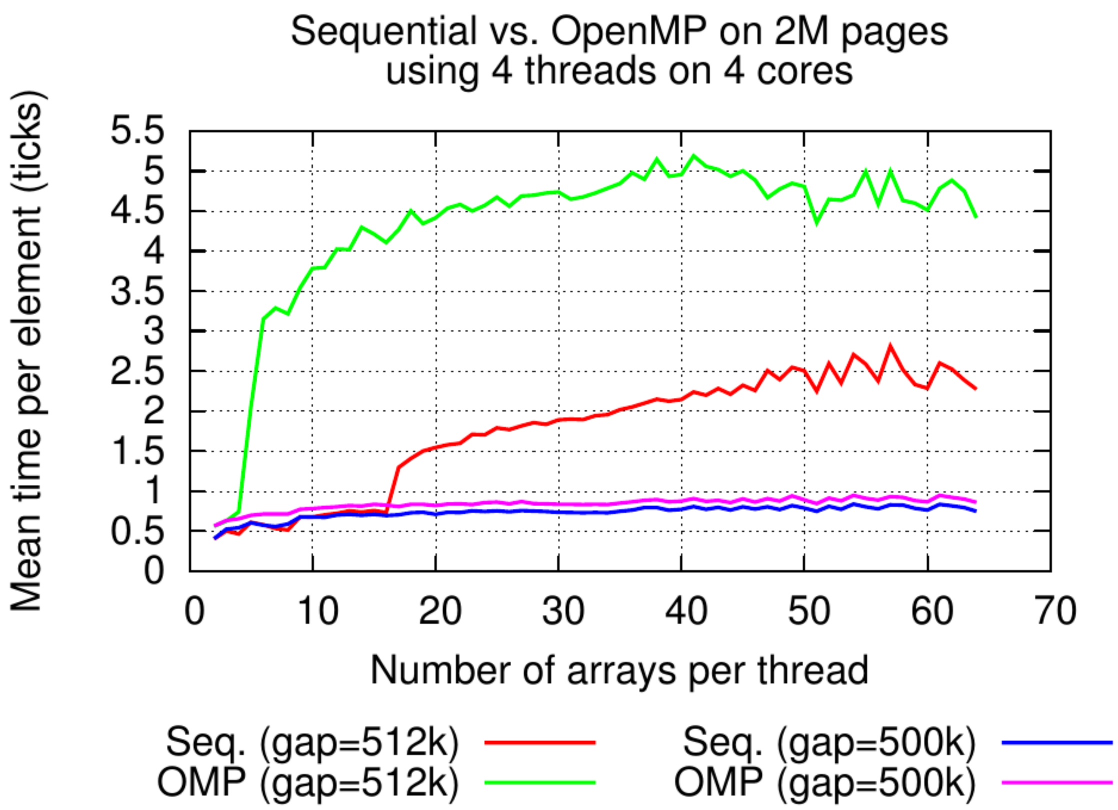
- Get the **expected improvement** on **4K pages** (40% for sequential).
- Also improve **scalability** on 1 socket
- On NUMA **locking effets become dominant for scalability**
- Get the constant improvement related to page zeroing.





Impact on threads

- **No limit** on concurrent arrays for **unaligned allocations**

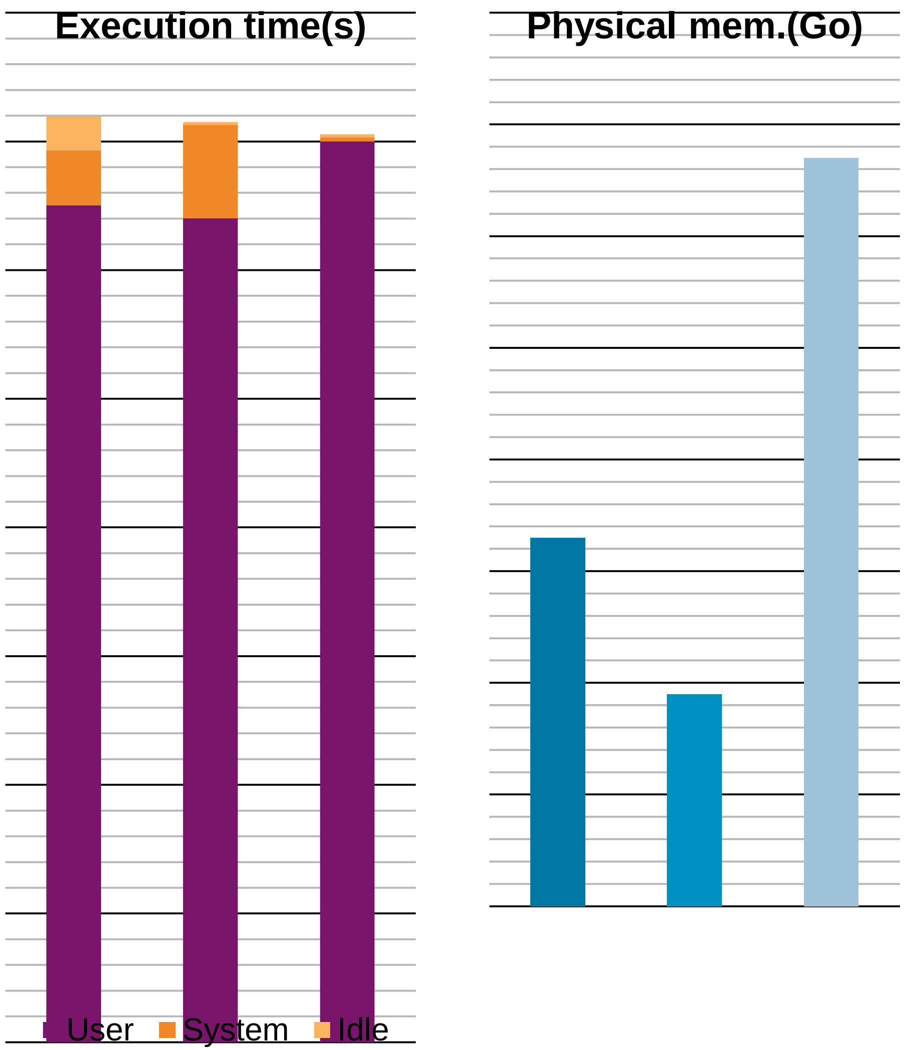


A little bit of bibliography

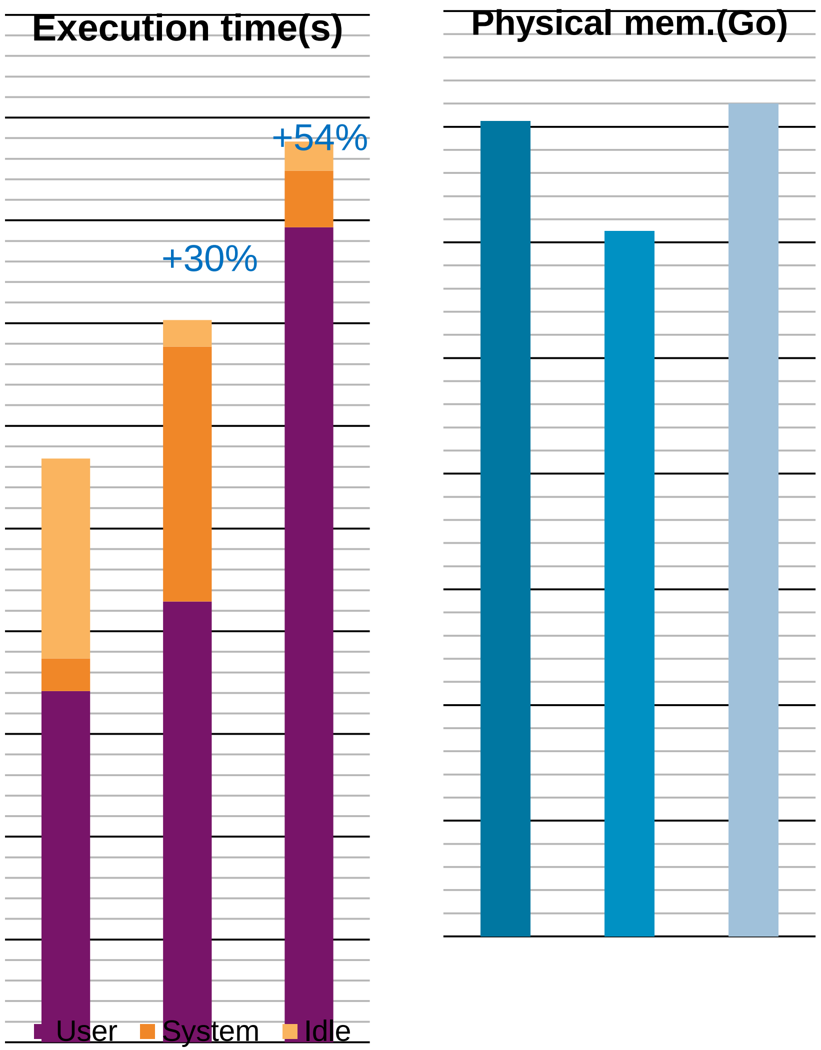
- **Interesting to read :**
- What every programmer should know about memory (Urlich Drepper)
  https://people.freebsd.org/~lstewart/articles/cpumemory.pdf
- For all details from this presentation : look on my PhD. thesis **:**
- Contribution à l'amélioration des méthodes d'optimisation de la gestion de la mémoire dans le cadre du Calcul Haute Performance
  https://hal.archives-ouvertes.fr/tel-01253537

Hera preliminary results

12 cores



128 cores



How to avoid page zeroing cost ?

- Microsoft approach **:**
  - **Windows** uses a **system thread** to clear the memory
  - So its done **out** of **critical path**
- But **zeroing**:
  - Implies **useless work**
  - Consumes CPU **cycles** so **energy**
  - Consumes **memory bandwidth**
- **Allocation pattern** follow:

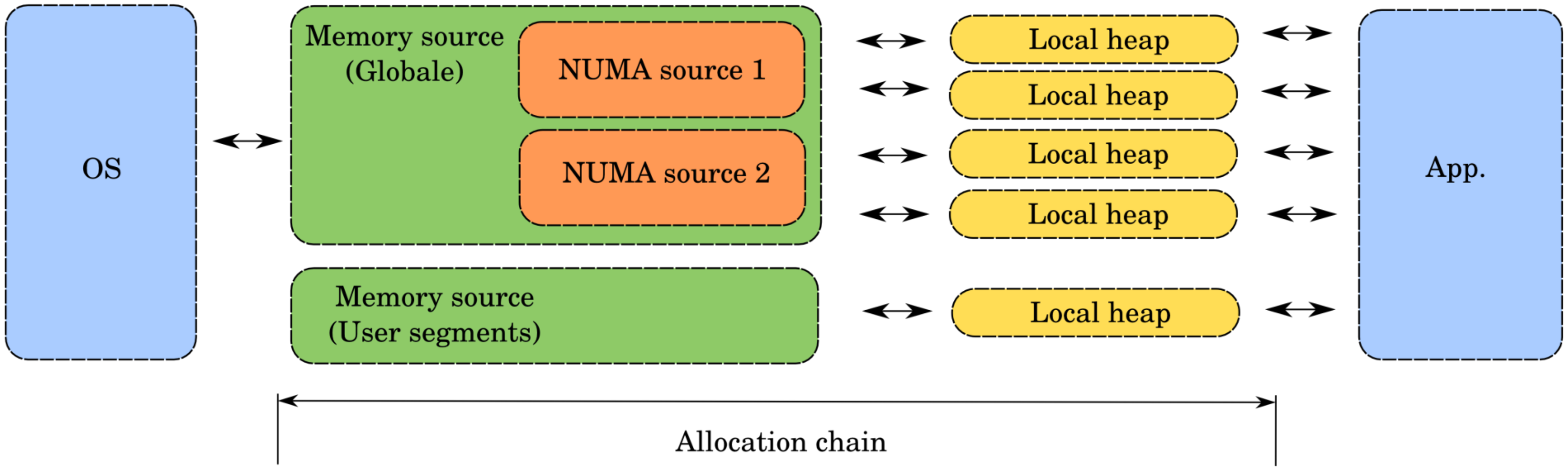
```
double * ptr = malloc(SIZE * sizeof(double));
for ( i = 0 ; i < SIZE ; i++)
        ptr[i] = default_value(i);
```

- Why not **avoid them** ?

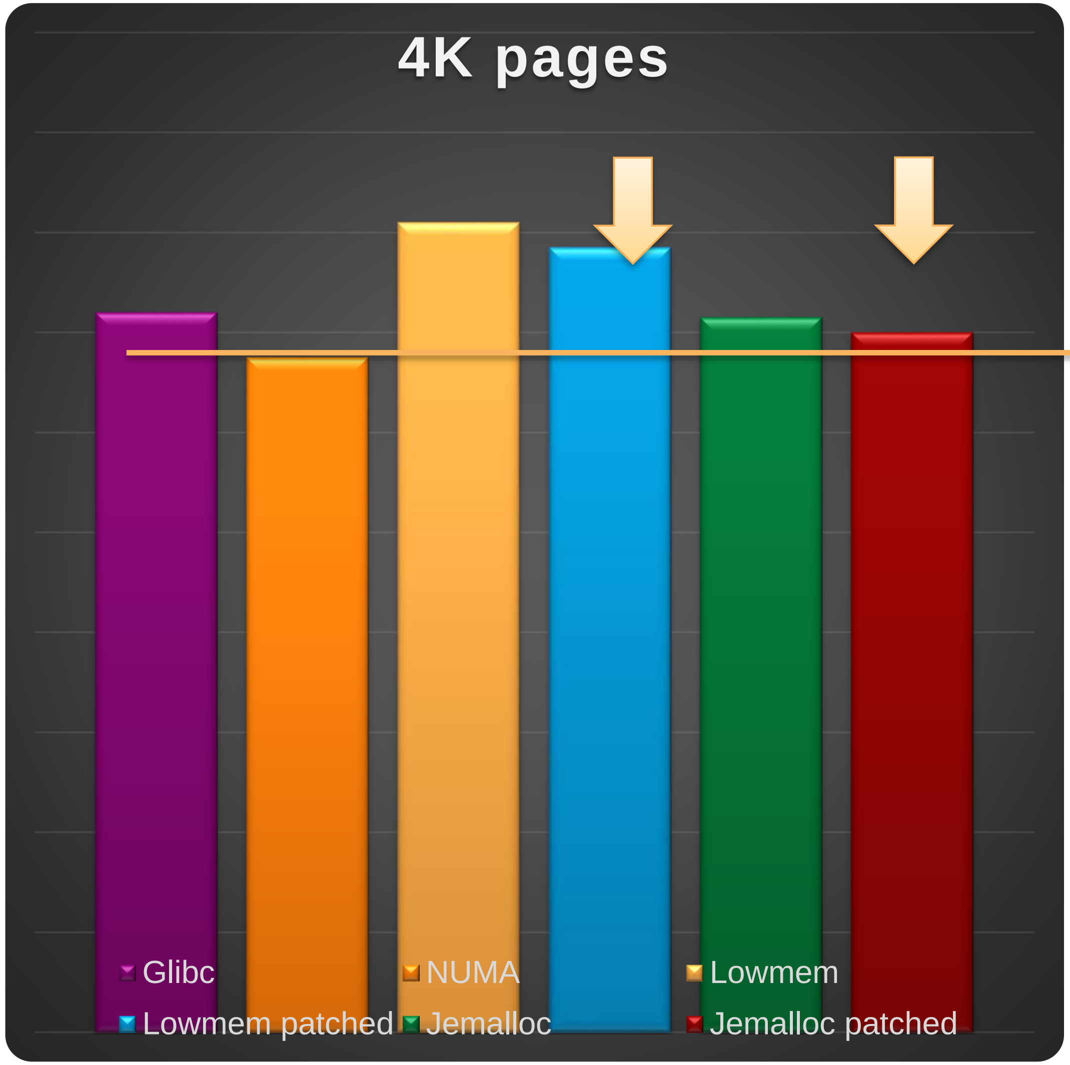
Global structure

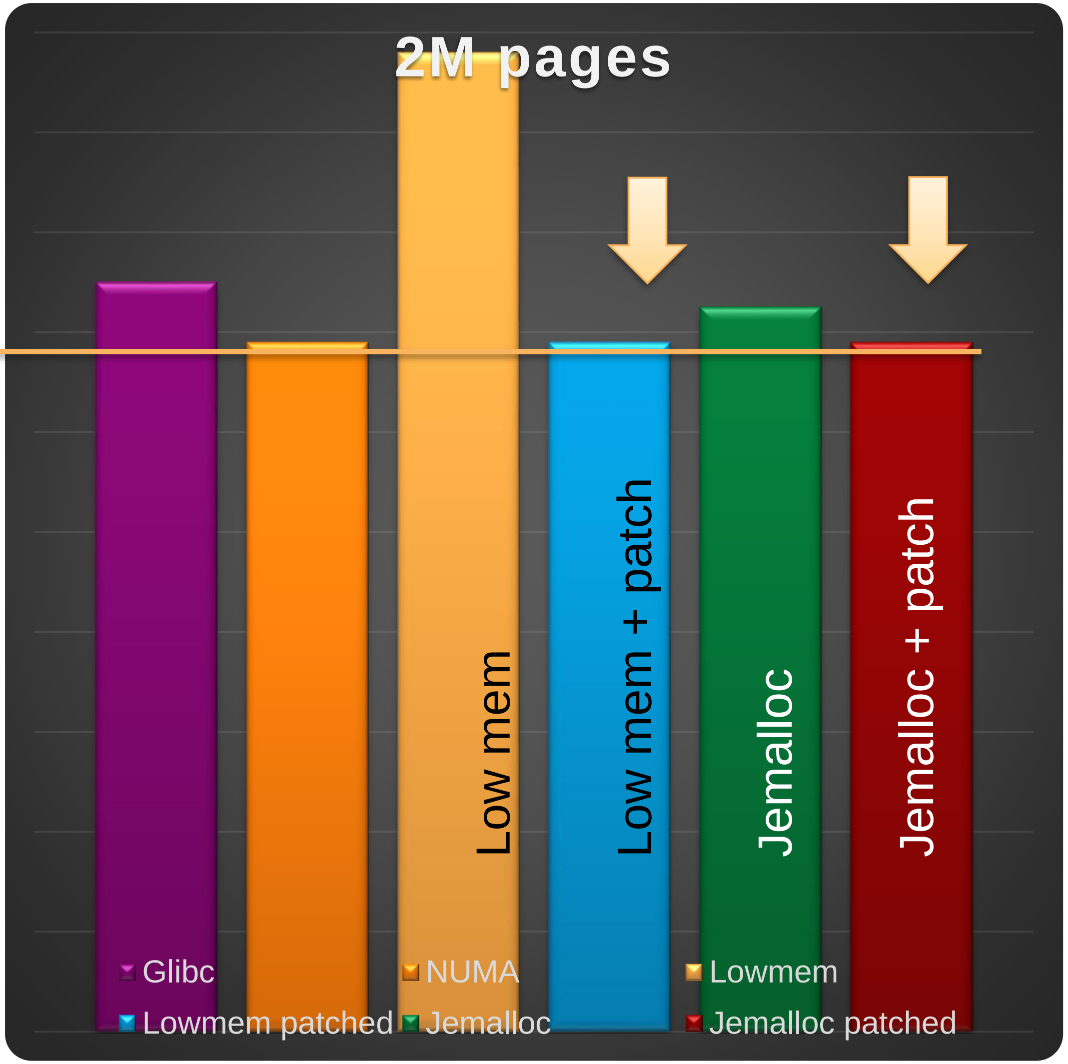
- **Memory source** :
  - Manages **requests to the OS**
  - Exchanges per **macro-blocs** larger than **2 MB**
  - Acts as a **cache** by keeping macro-blocks
  - Manages balance **performance / consumption**

- Per thread **local heap :**
  - **Lock free**
  - Manages **small chunks**
  - **Split** macro-blocs



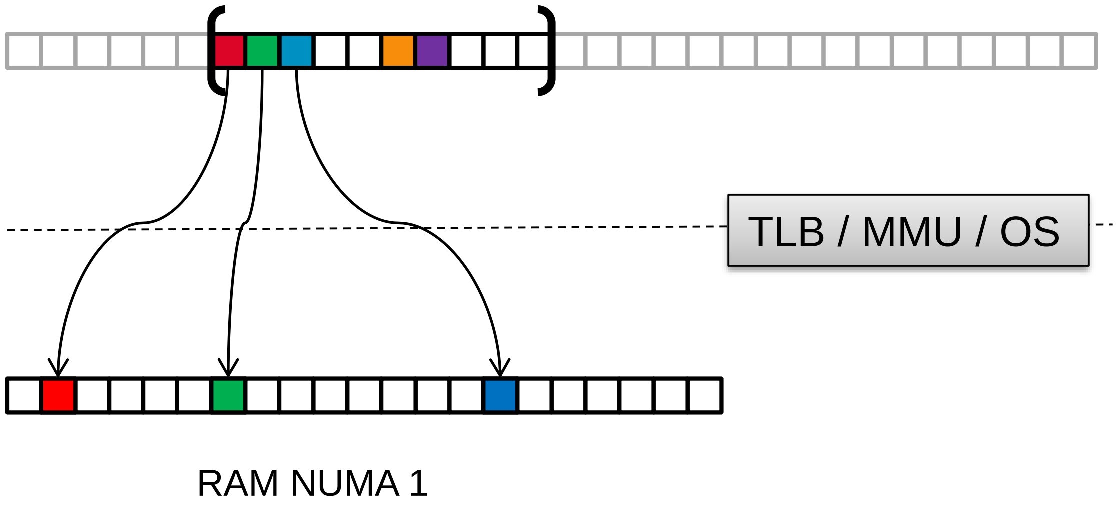
Hera results on bi-westmere (2*6 cores)

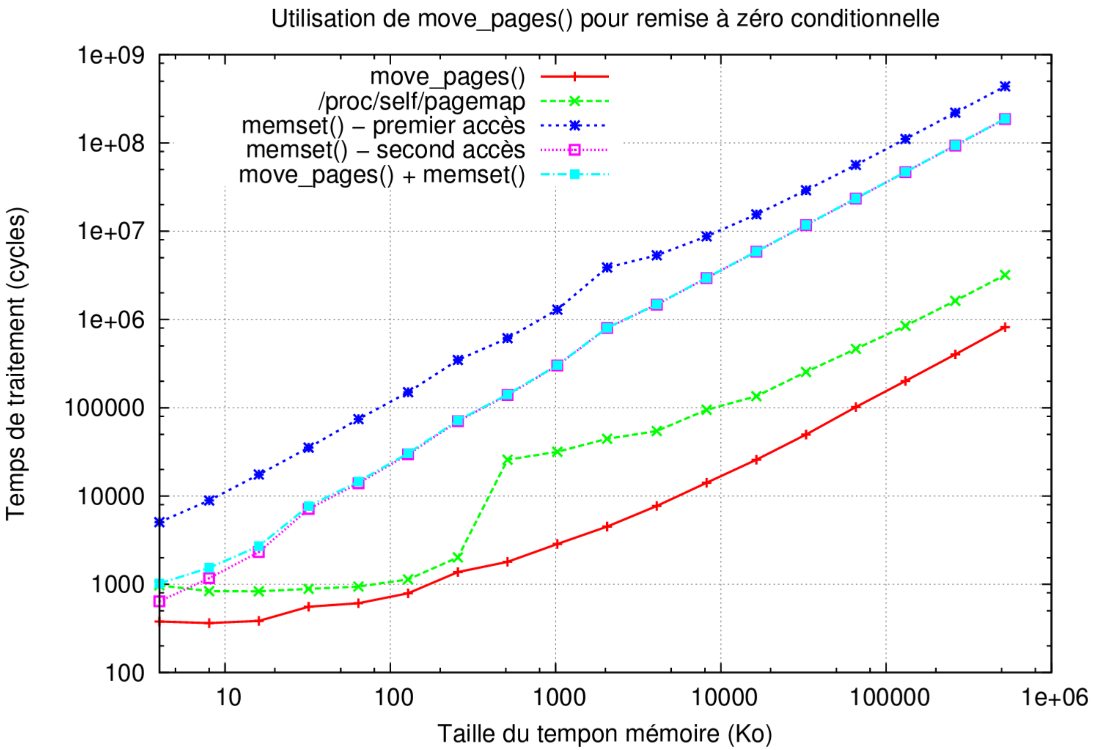




Calloc case

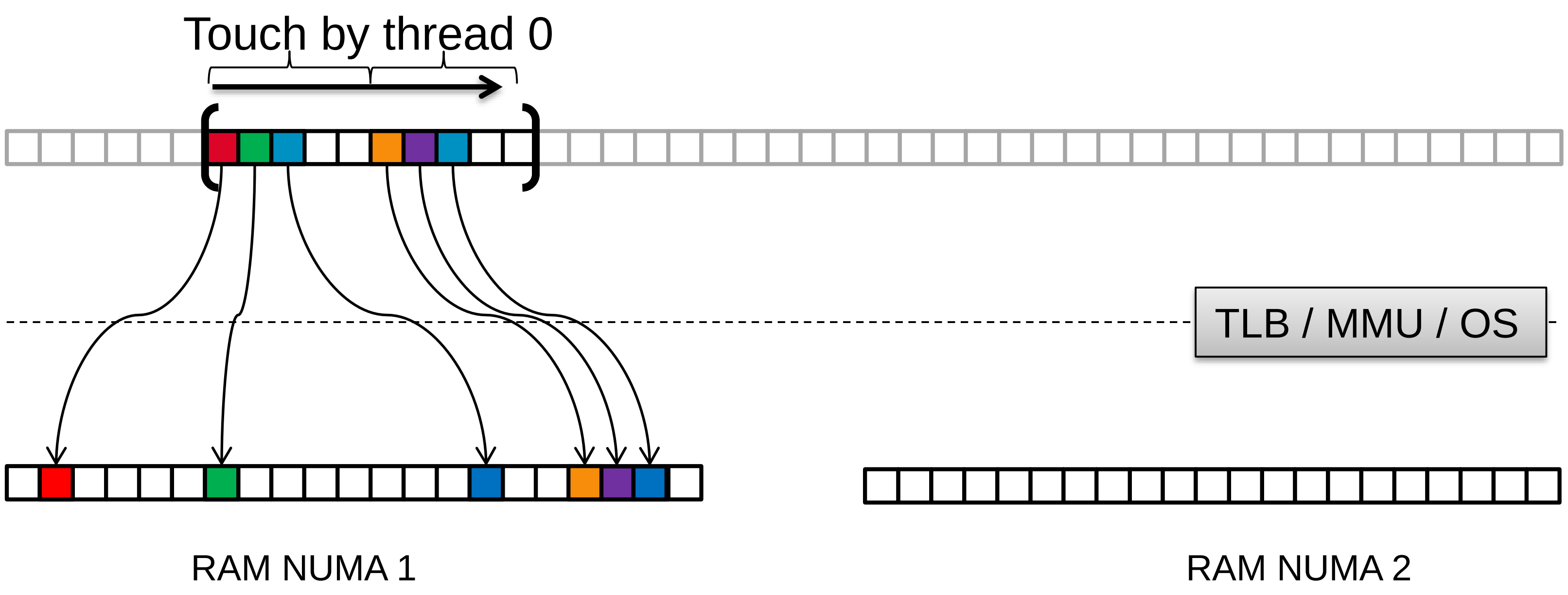
- **Calloc** need to clear all the memory to **ensure zeroing**
- One can remark that **untouched memory** will **already be cleared** by OS
- Can we **avoid** to **clear untouched pages** ?
  - **Yes**
  - We can detect with **move_pages()**
  - Or **/proc/PID/pagemap** [not anymore]





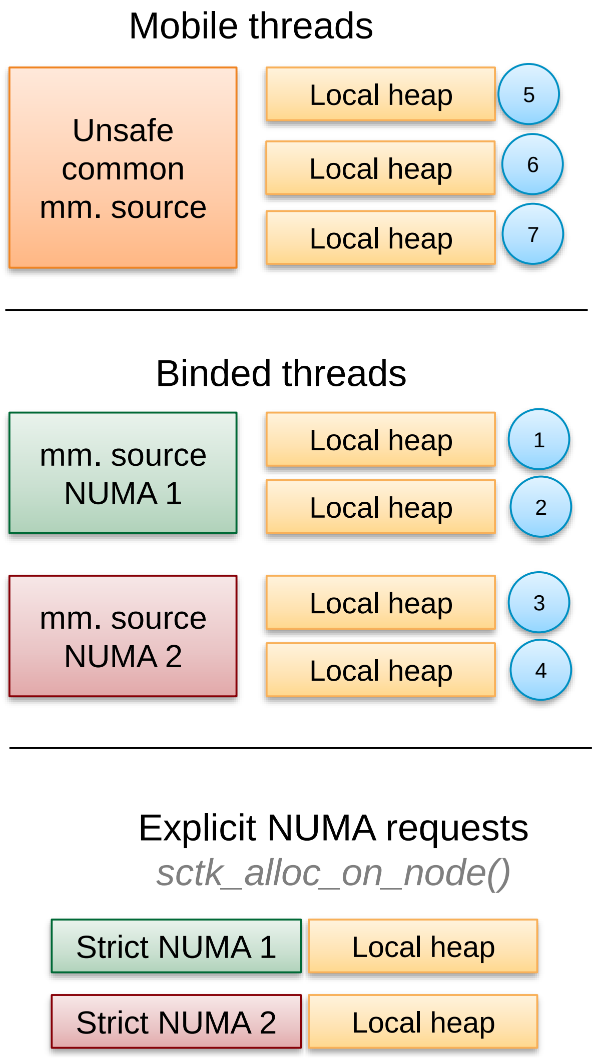
Example of NUMA allocation issue

- Thread 0 call malloc
- Then is call memset and touch all the memory
- Then we access with multiple threads……
- But all the memory have been mapped on the NUMA node 0 !



NUMA strategy

- With **standard API**, we can only **suppose local use**
- **Local heap** guarantees **NUMA isolation**
- **No exchanges** between **NUMA sources**
- **MM. sources** are **selected** with **hwloc** at **thread init.**
- **Threads** are **not binded by default**, so they **move** !
- Create memory sources with **confidence levels** :
  - A **common one** for **mobile threads**
  - **Per NUMA** for **binded threads**
  - **Per NUMA** for **explicit requests** (binded with hwloc)



Hera results on bi-westmere (2*6 cores)

- Standard pages (4K):

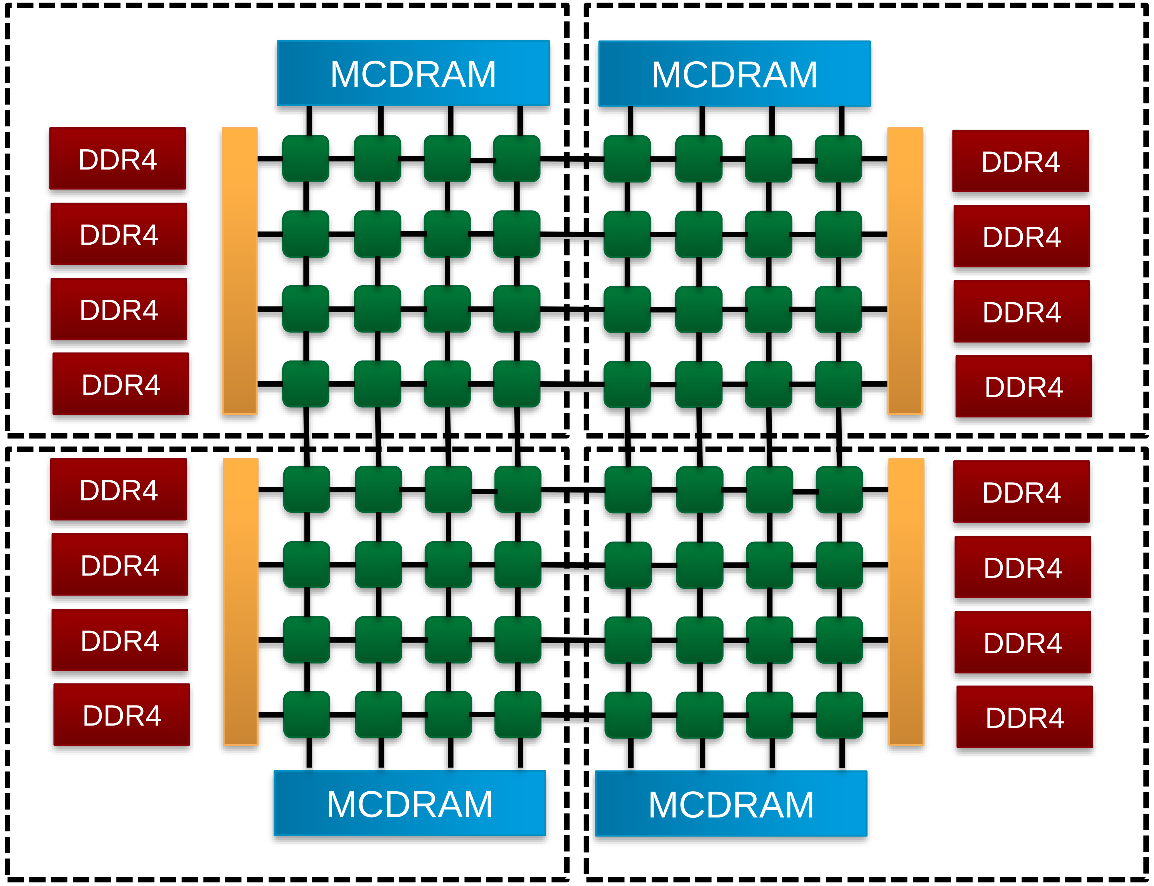
Allocator	Kernel	Total (s)	Sys. (s)	Mem. (GB)
Glibc	Std.	144	9	3,3
NUMA profile	Std.	**135**	**2**	**4,3**
Lowmem profile	Std.	162	**16**	**2,0**
Lowmem profile	**Patched**	157	**11**	2,0
Jemalloc	Std.	143	15	1,9
Jemalloc	Patched	140	9	3,2

- Transparent Huge Pages (2M):

Allocator	Kernel	Total (s)	Sys. (s)	Mem. (GB)
Glibc	Std.	150	13	4,5
NUMA profile	Std.	**138**	**2**	**6,2**
Lowmem profile	Std.	196	28	3,9
Lowmem profile	**Patched**	138	3	3,8
Jemalloc	Std.	145	15	2,5
Jemalloc	Patched	138	6	3,2

Now also inside the CPU – Intel KNL

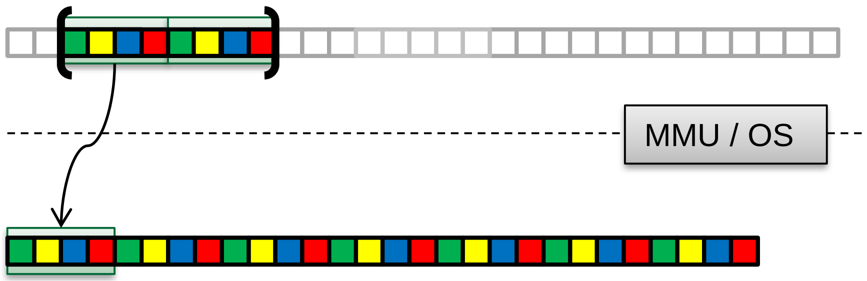
- Intel KNL (64 cores) can be configured in **2 or 4 NUMA domains**
- Also add **MCDRAM** (similar idea than GPU GDDR5) **viewed** as a **NUMA** node



- Or on **AMD** CPUs

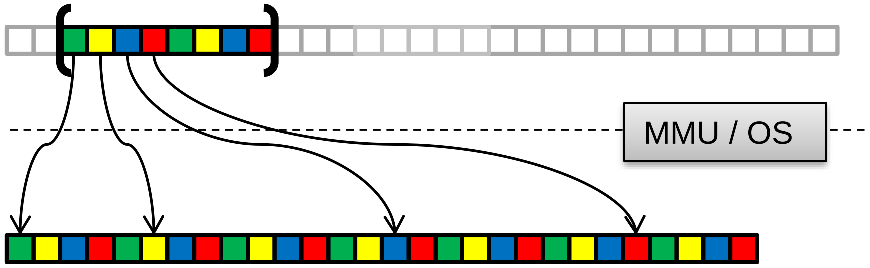
Existing solutions

Huge pages



- **Larger than cache ways**
- **Native** support on **FreeBSD**
- **Extended** support on **Linux** / **OpenSolaris**

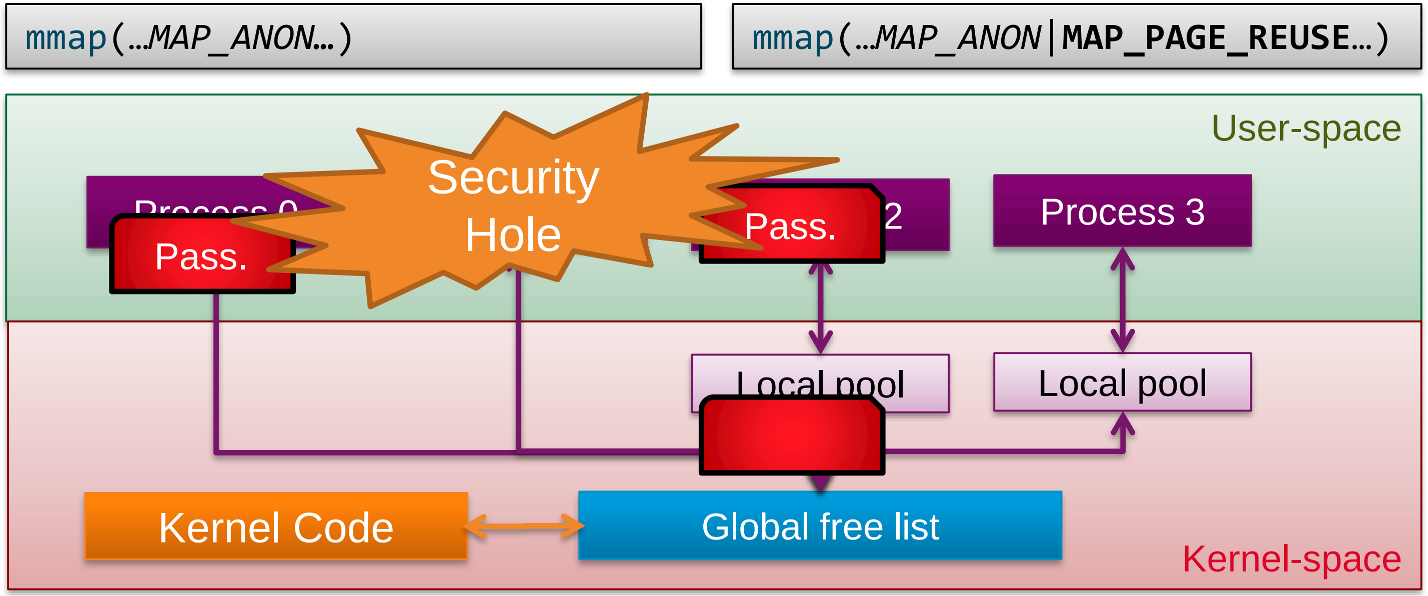
Page coloring



- 4K pages by **taking care of associativity**
- Available on **OpenSolaris**
- **Color** based on **virtual address** (modulo)
- **Regular coloring** : coloration with **repeated patterns**

Reusing local pages to avoid zeroing

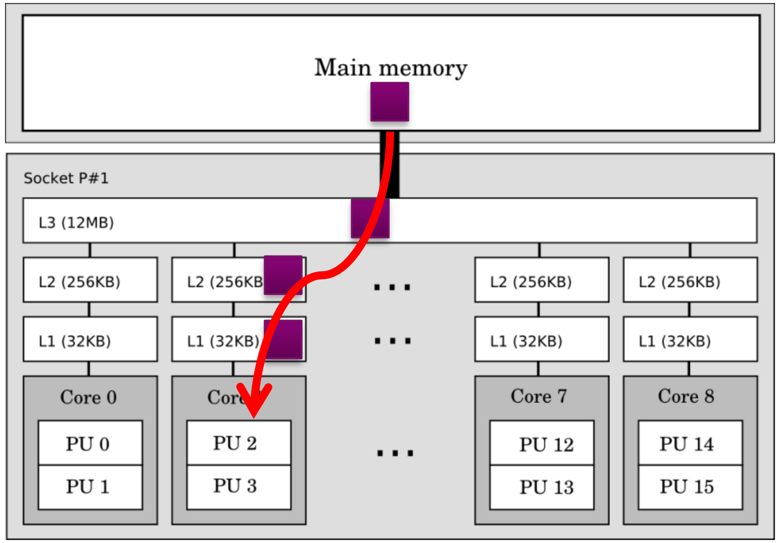
- Page zeroing is **required** for **security reason**
- It prevents information **leaks** from **another processes** or from the **kernel**.
- **But we can reuse pages locally !**
- Need to **extend** the **mmap semantic** :
- Usable by **malloc / realloc**.

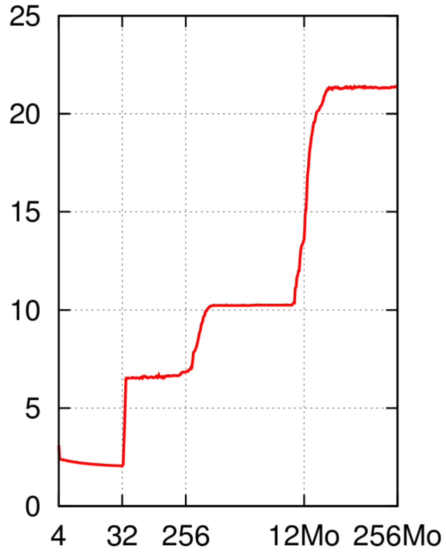


Architecture

- Computer science : **operations** & **data**
- Multiple **memory levels**
- Hierarchical **caches**
- *Pre-fetcher*

Processor : 8 cores

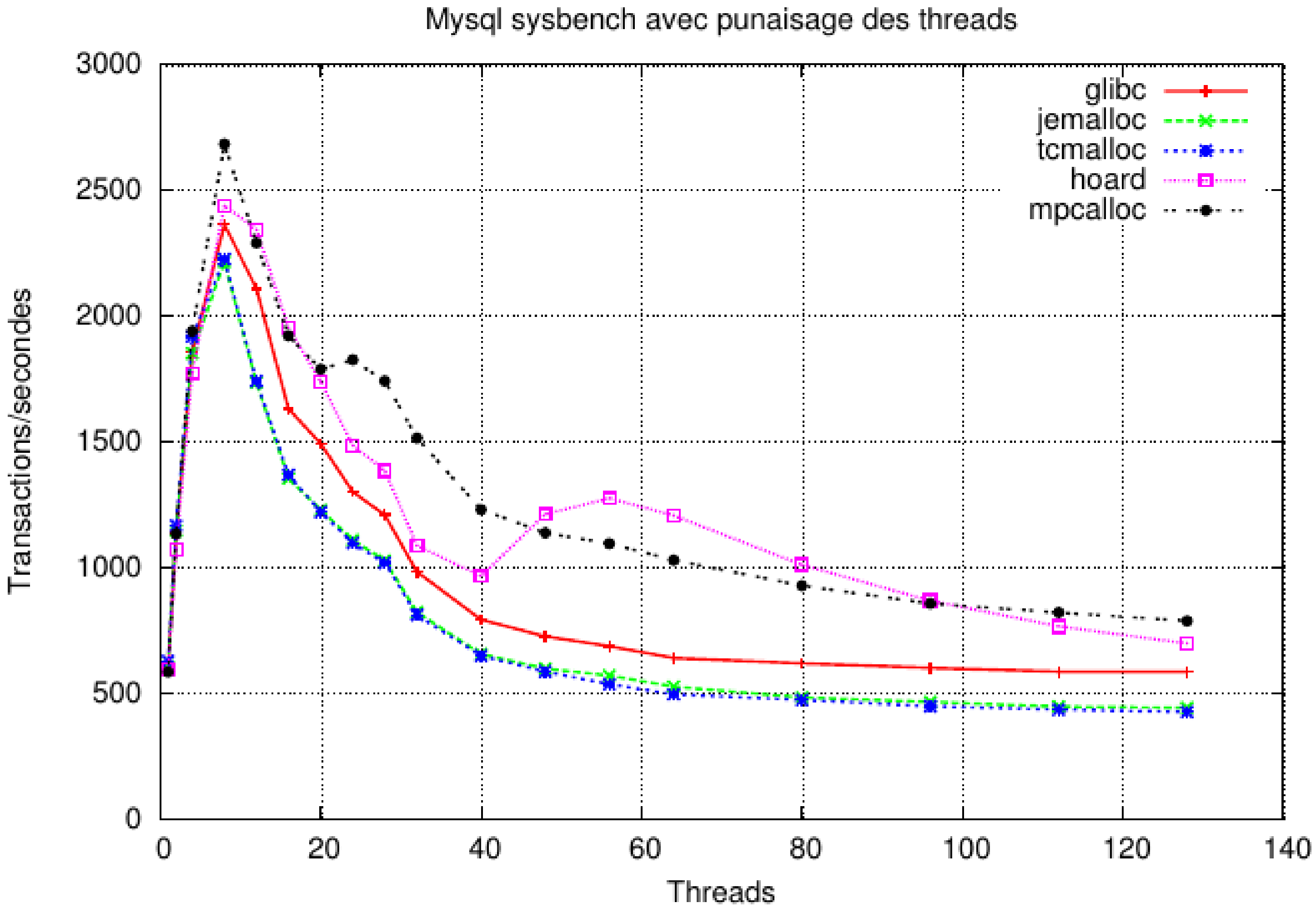




Large allocations

- Small allocation **well handled** by most allocators, **best is jemalloc / tcmalloc**.
- Cost for **large allocation : page faults.**
- **Commonly neglected**, literature mainly discuss small allocations
- Direct call to **mmap/munmap**
- **HPC applications** (expected to) use **large arrays**

Mysql results



Impact on threads

- **Larger effects** on **<u>shared caches</u>** with threads/processes (Nehalem)
- EulerMHD : **Slowdown** up to **3x** on **FreeBSD**
- **16** ways L3 cache implies a maximum of **4 aligned arrays** per core