DE LA RECHERCHE À L'INDUSTRIE

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

CONTRIBUTION TO THE DEVELOPMENT OPTIMIZATION METHODS FOR MEMORY MANAGEMENT IN HIGH−PERFORMANCE COMPUTING.

PhD. thesis defense

Sébastien Valat

17 july 2014

Thesis work done at :
CEA,DAM,DIF F-91297 Arpajon

www.cea.fr

Plan

INTRODUCTION

Context : HPC

- **Supercomputers** for numerical simulations

- **Massively parallel** machines (**3 million cores**)

- At CEA, **Tera 100 :**
  - **6^{e}** from TOP 500 in 2010
  - **140 000** cores, **1.05** Pflops.

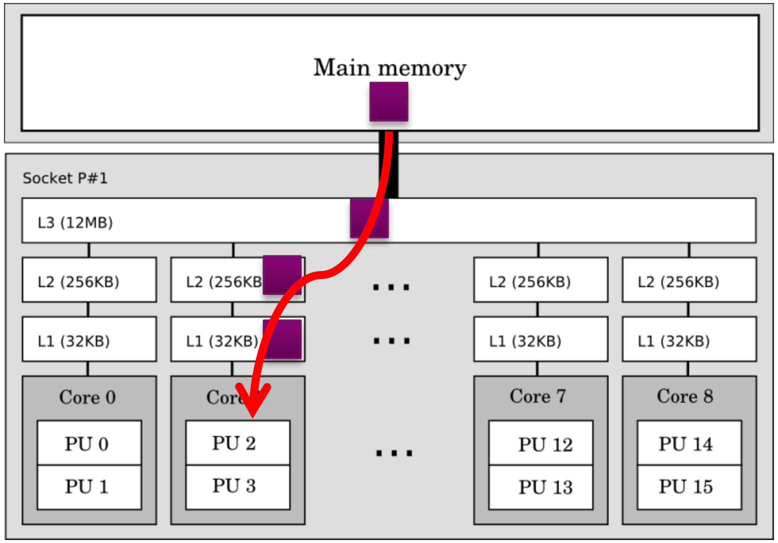
- **Growing parallelism** inside nodes :
  - Tera 100, **large nodes** :**128 cores** (16 processors)
  - Now : **Intel Xeon Phi**, **60 cores** (1 processor)

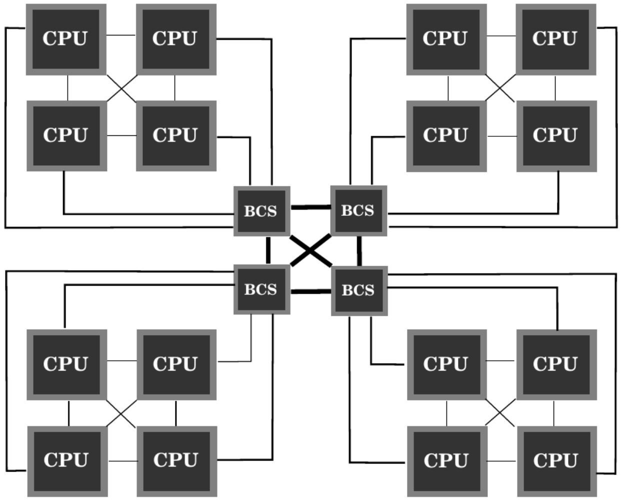
- **Memory** becomes a **critical resource** :
  - Growing impact on **performance** (**data movements / management**)
  - Decreasing **memory per core**

Architecture

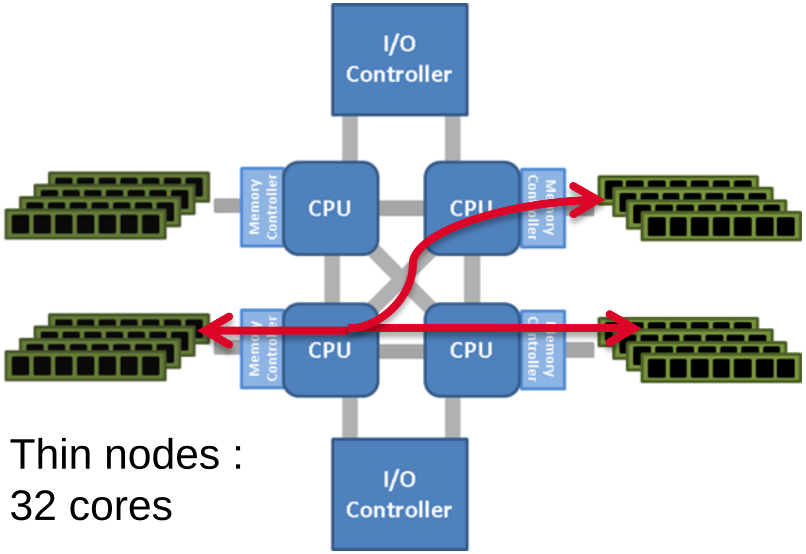
- Computer science : **operations** & **<u>datas</u>**
- Multiple **memory levels**
- Hierarchical **caches**
- **Remote** / **local** memories **(NUMA)**

Processor : 8 cores





Large nodes : 128 cores (BCS)

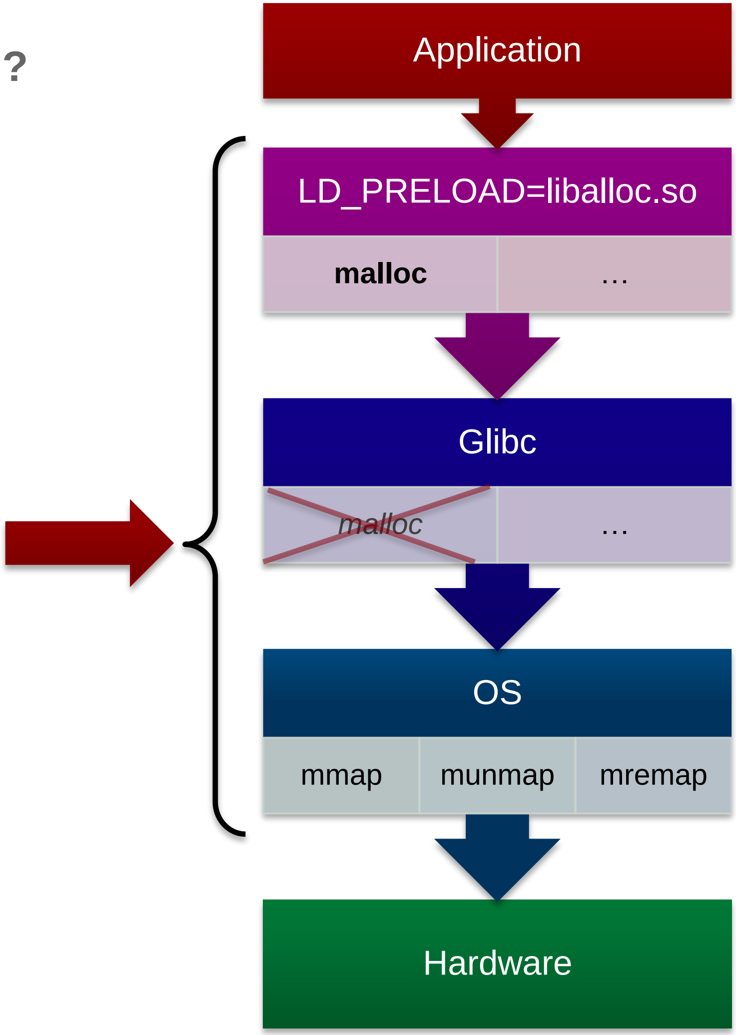


Thin nodes : 32 cores

User space allocator : malloc

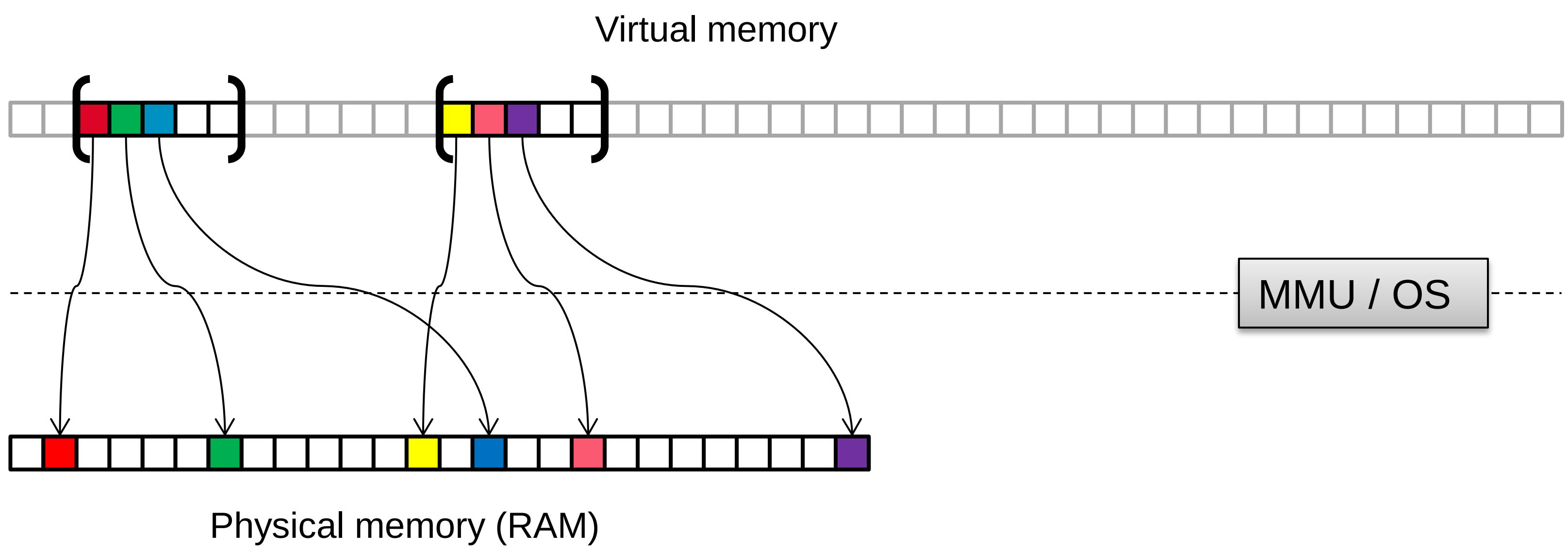
- **Impact of memory management mechanisms ?**
- Focus on :
  - Impact on **allocation time** :
  - Impact on **access efficiency** (placement)
  - Memory consumption
- Involving **two components** :
  - **Operating System** (OS)
  - User space **memory allocator** (malloc)
- Malloc C interface :

```
float * ptr = malloc(SIZE);
…
ptr = realloc(ptr,NEW_SIZE);
…
free(ptr);
```



OS virtual / physical address spaces

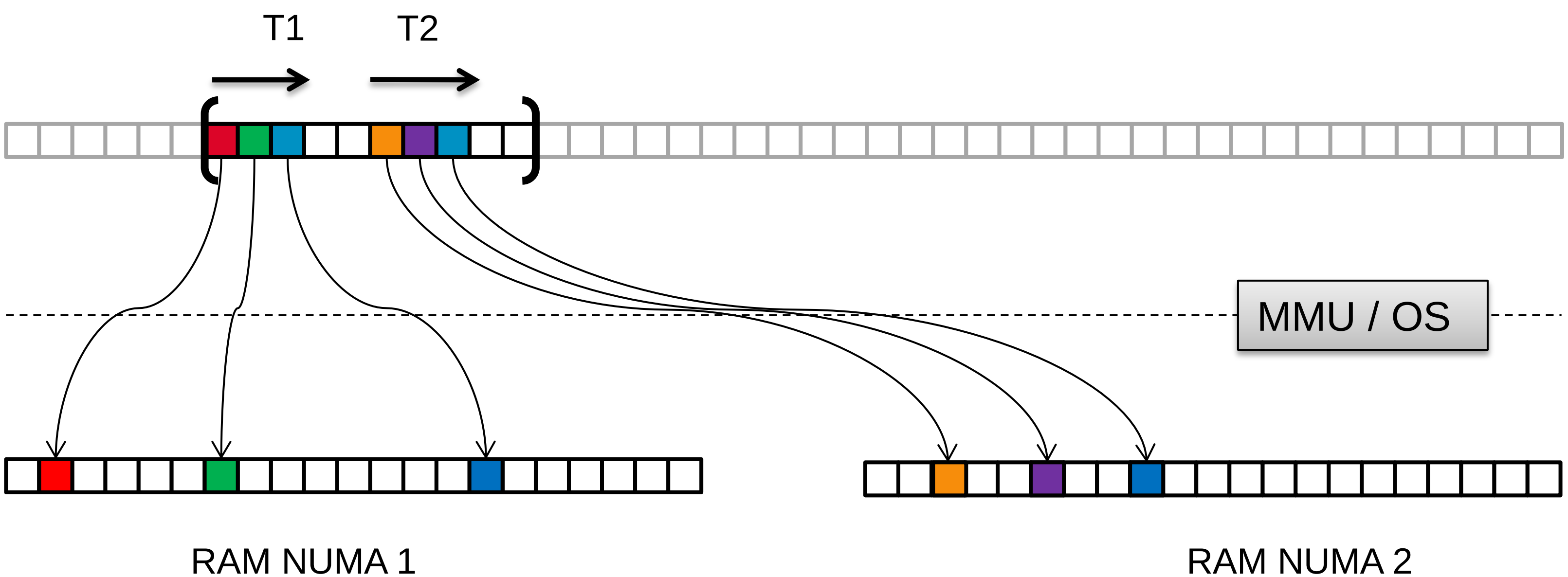
- Two address spaces : **physical** + **virtual**
- Description of the **memory mapping** in blocks of **4 KB (pages)**
- **Segments** creation with syscalls : **mmap / munmap / mremap**
- **Malloc** has the responsibility to **hide the pages to developers**



Lazy page allocation

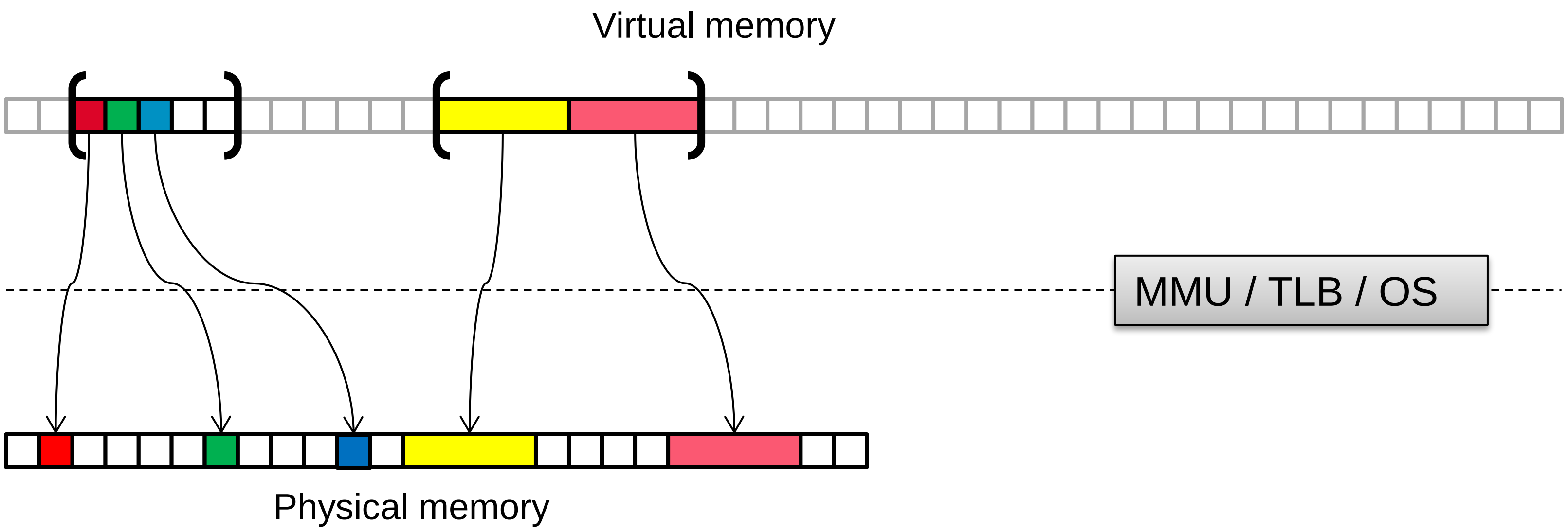
- **mmap** creates **pure virtual** segments
- First touch creates a **page fault** for each virtual page
- OS provides **physical pages** on **first touch**
- **First touch <u>implicitly</u>** determines **NUMA placement** of the page

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



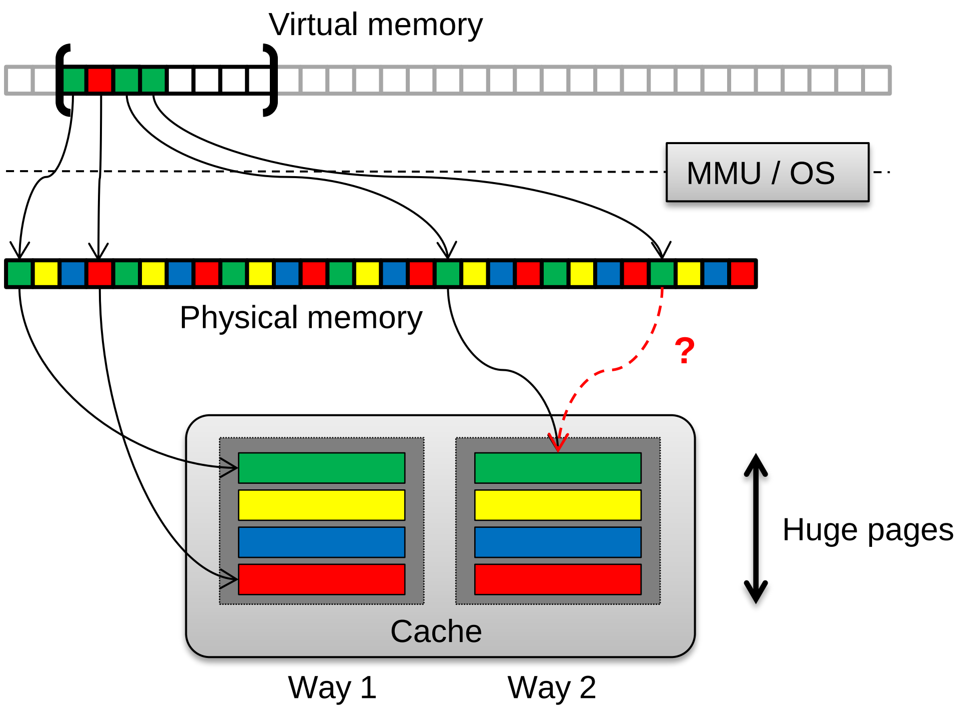
Huge pages

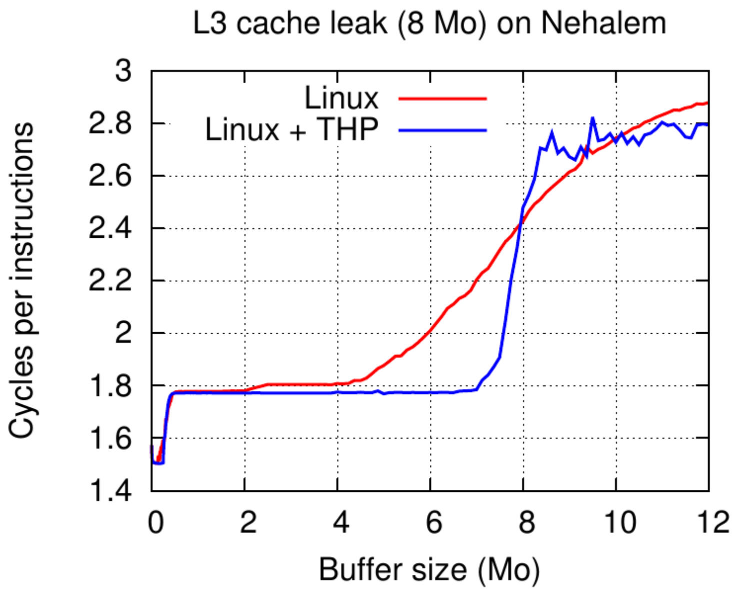
- **x86_64** processors also support **2 MB** or **1 GB** pages **(Huge pages)**
- **Address more** with **less pages**
- **TLB** (*Translation Lookaside Buffer)* **cache** inside the processor MMU
- **Support Linux : Transparent Huge Pages (THP)**



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

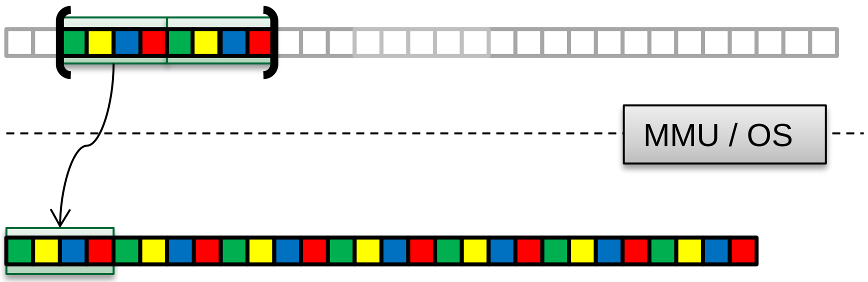




Existing solutions

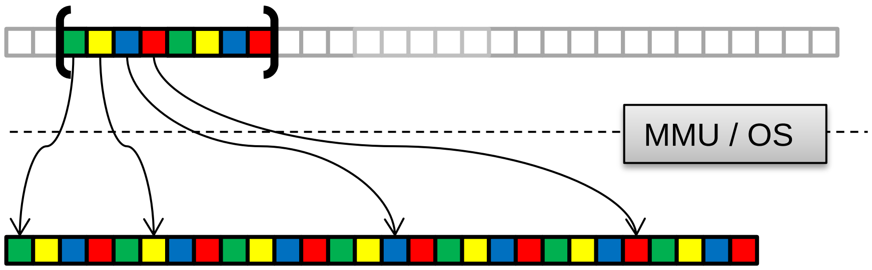
Huge pages

- **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**



ANALYSIS OF OS / ALLOCATOR / CACHES INTERACTIONS

OS strategies comparison

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

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

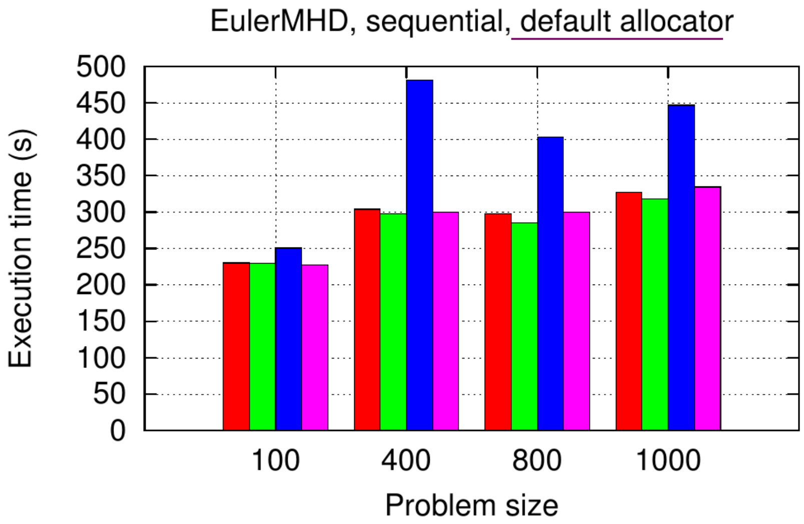
- Is **Linux** slower due to **random paging** ?
- Tested architecture : **Nehalem bi-socket**
- Use a fixed compile chain : **GCC/Binutils/MPI/BLAS**
- **Focus a pathological case**

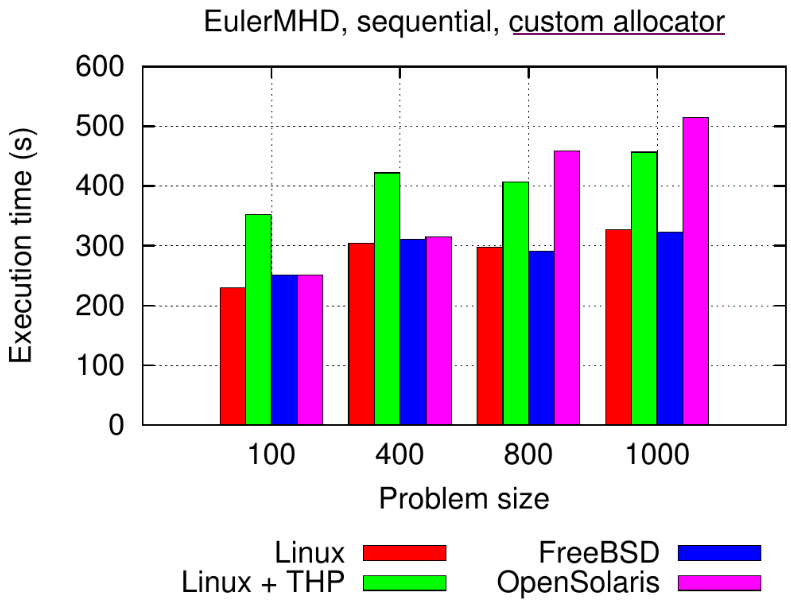
EulerMHD issue

- **EulerMHD :**
  - **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]
```

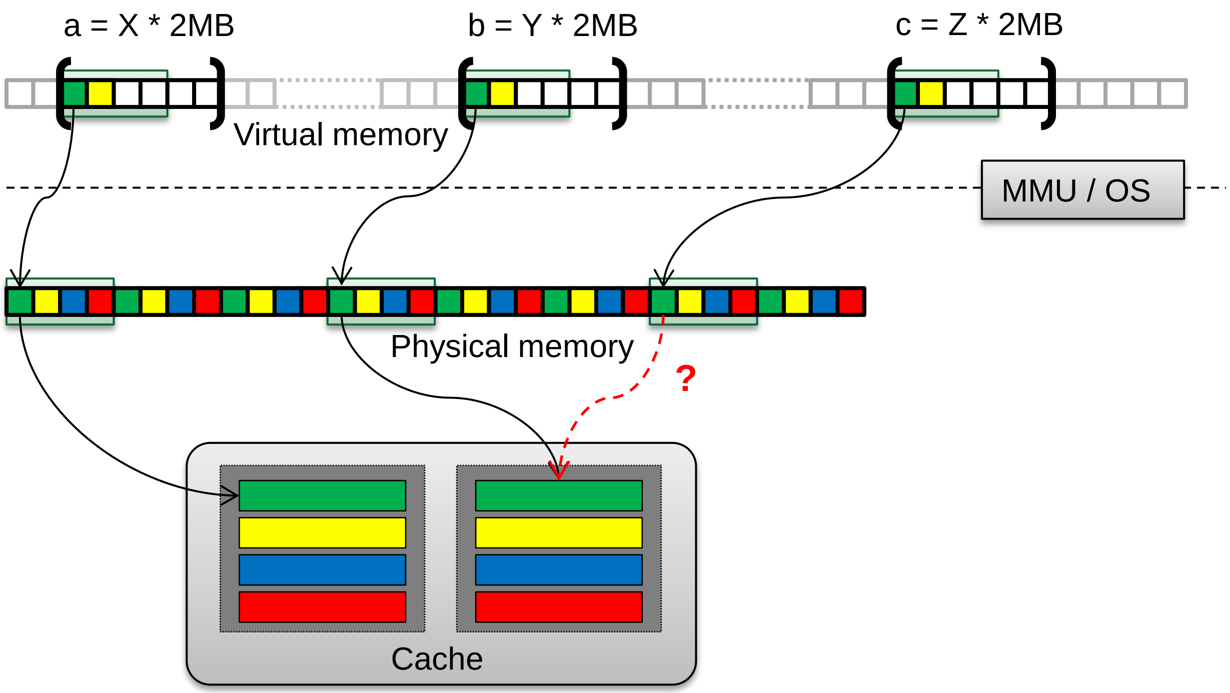
- Change between OS's :
  - **User space memory allocator** (malloc).
  - **OS paging policy**
  - *(Scheduler)*
- Effect can be controlled by **changing the allocator.**





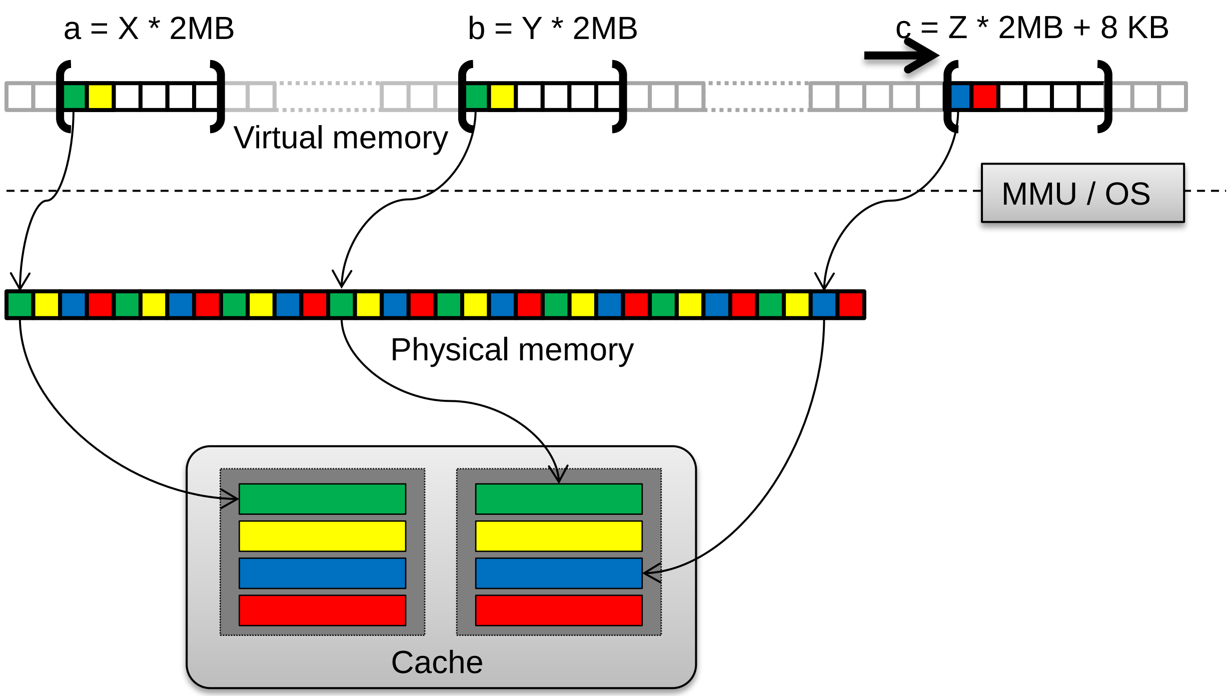
Alignment effect on regular coloring

- 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



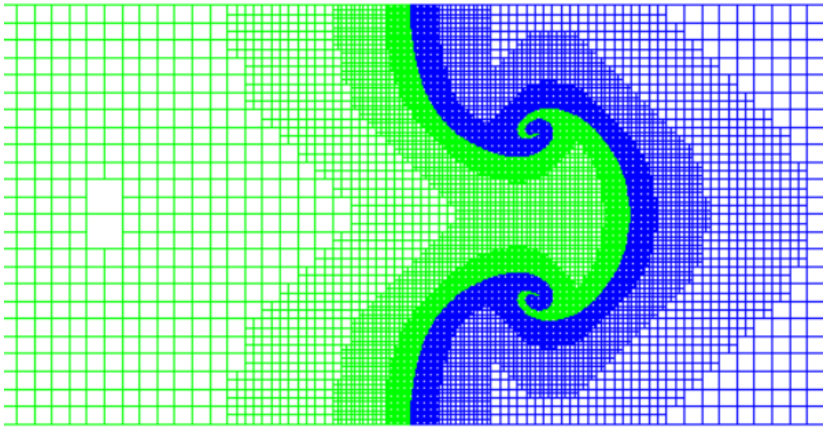
Solution

- Avoid segment **alignments** on **cache way size** (mmap / malloc)**.**
- The **Linux random** approach **prevents pathological cases**
- Do not use **regular patterns** for **page coloring** (eg. **single modulo**)
- **Huge pages** are **regular** by **hardware definition**

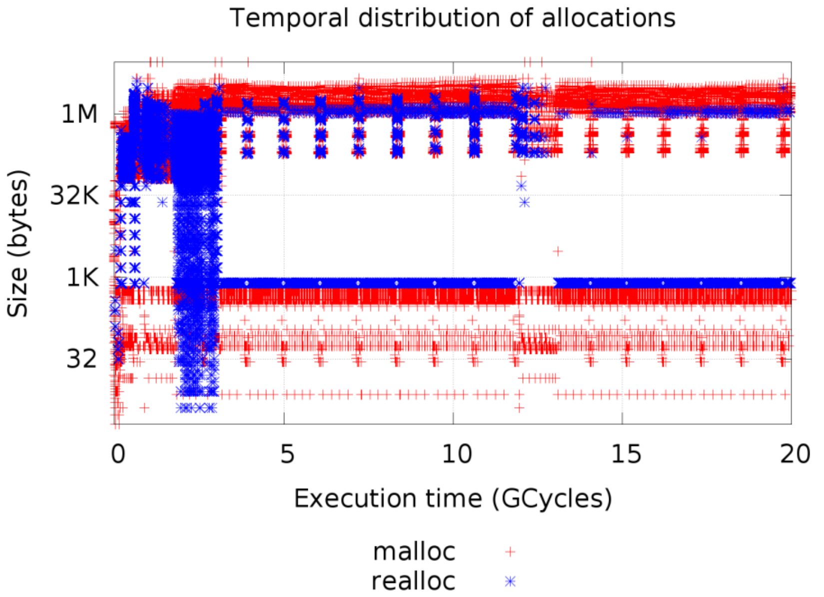


ALLOCATOR FOR HPC APPLICATIONS

Allocator performance on HPC applications

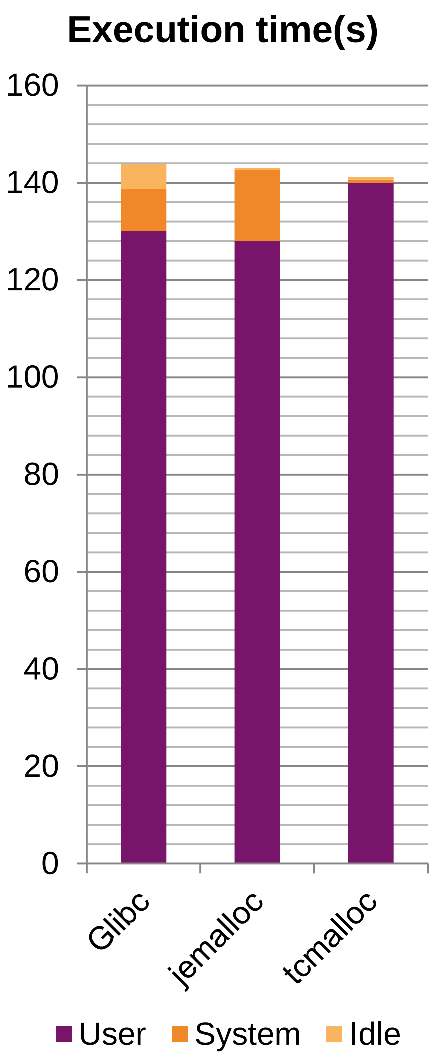


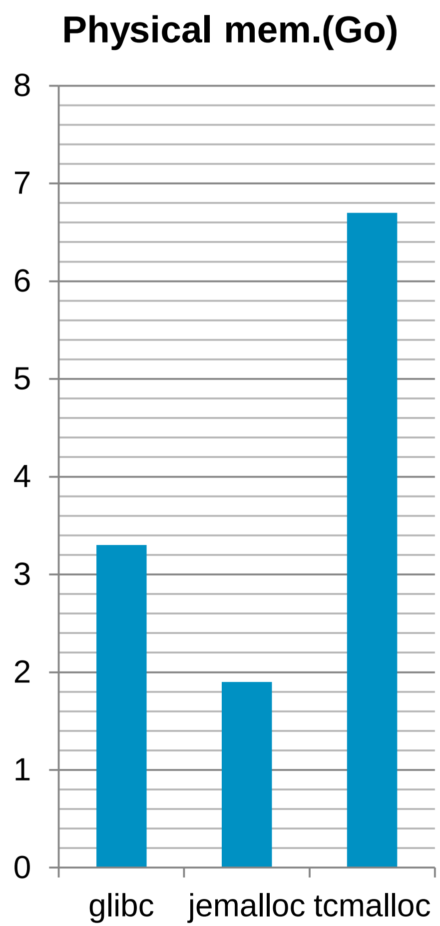
- Main interest : **malloc time cost**
- Test case : **Hera**
  - **Adaptive Mesh Refinement** (AMR)
  - **Massive C++/MPI code** (~1 million lines).
- **Large number** of **memory allocations** (~75 millions / 5 minutes on 12 cores)
- **Large number** of **alloc/realloc** around **~20 MB**
- **Available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



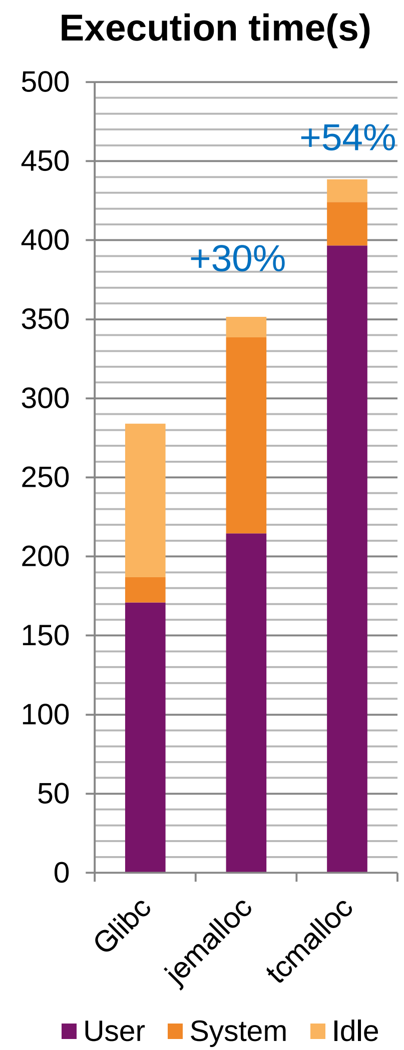
Hera preliminary results

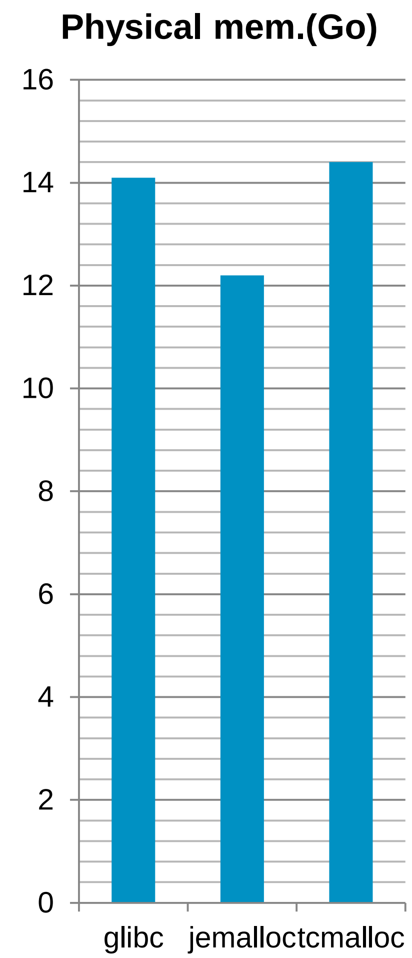
12 cores





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

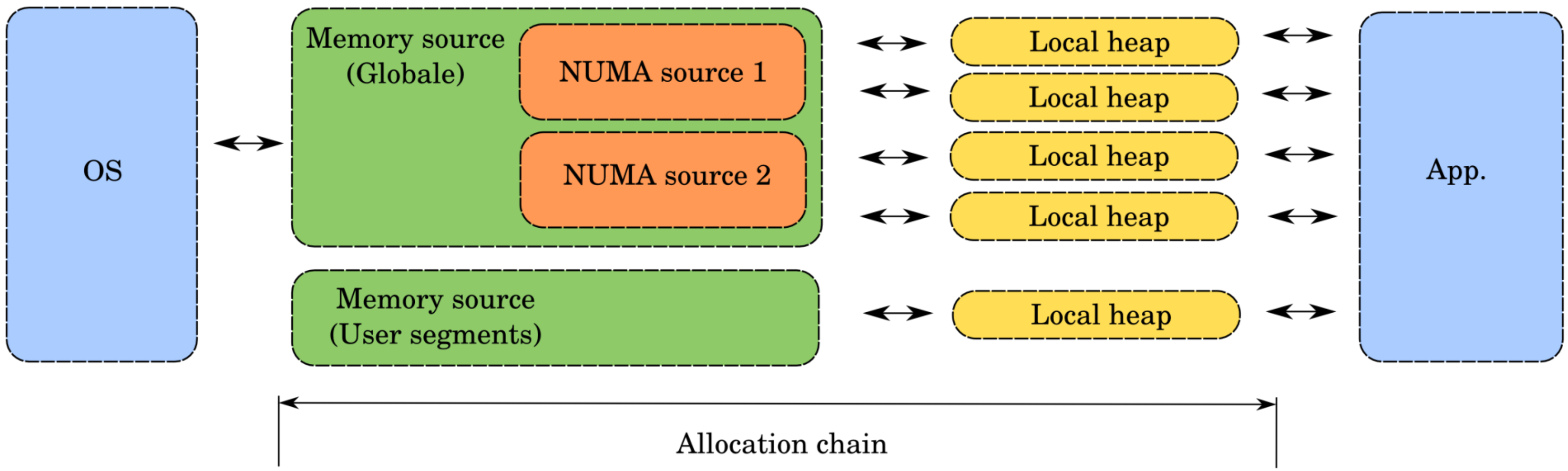
Large allocations

- 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**
- **Goals :**
  - **Recycle** large arrays
  - Avoid **fragmentation** on large segments
  - Take care of **NUMA**
  - Limit **locks**

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

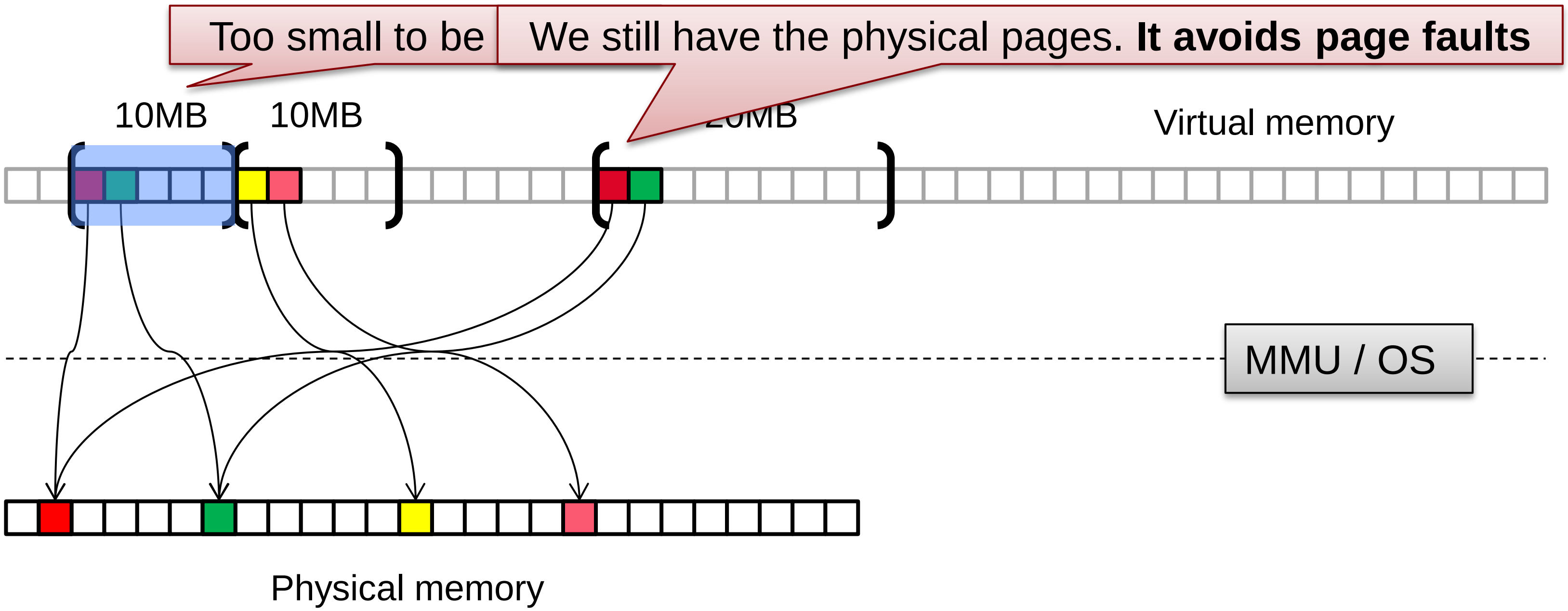


No fragmentation for large segments

- **Reuse of large segments** can induce **fragmentation**
- Example :

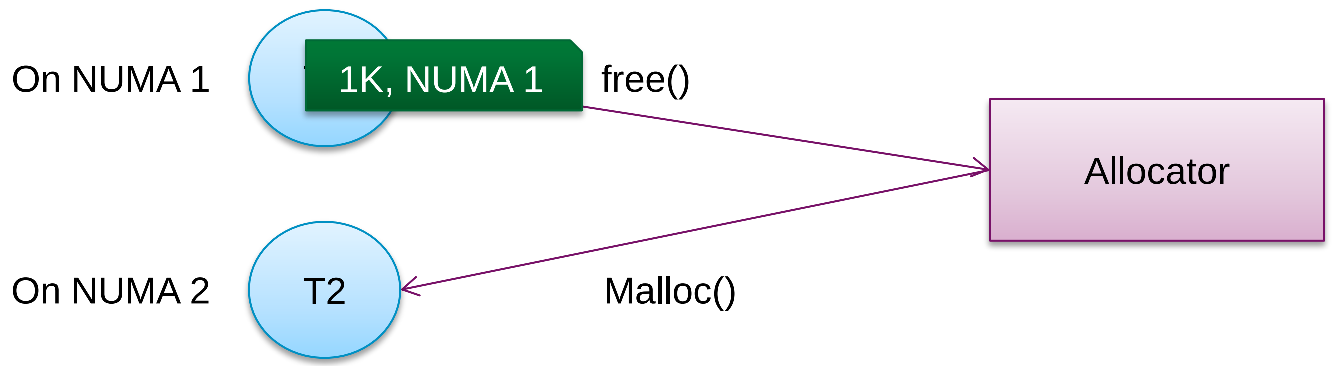
```
a = malloc(10MB);
b = malloc(10MB);
free(a);
a = malloc(20MB);
```

- **Can be avoided** by use of **mremap**



Malloc NUMA issue

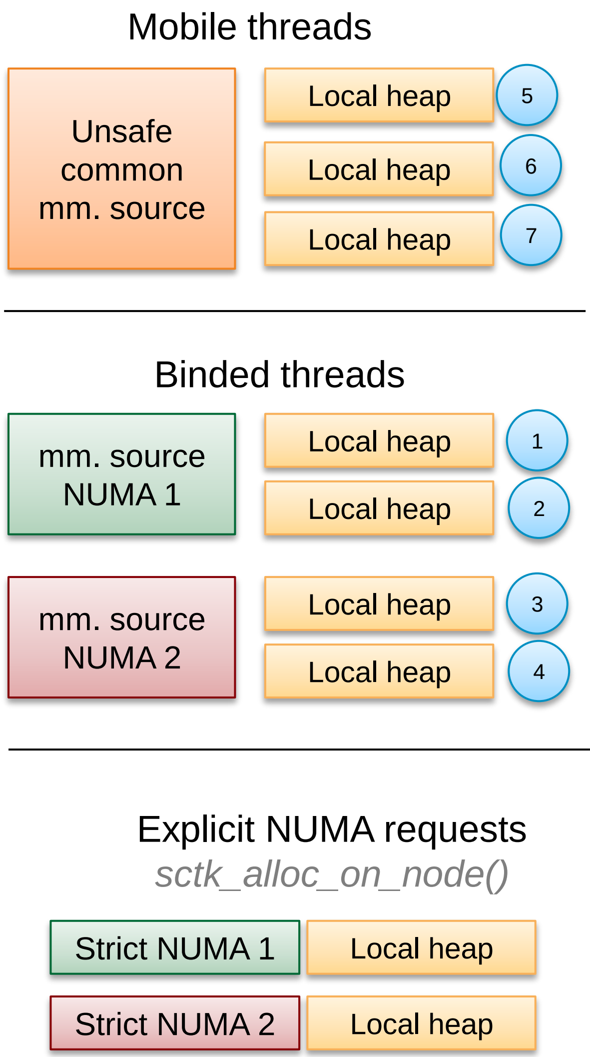
- **Exchanges** between **NUMA nodes :**



- Most **current allocators** are affected by this issue
- **Malloc** has **no information** about the **use** of allocated segments
- **Implicit binding** on **first touch**
- User space **allocator** do **not control physical binding** of **multi-page** segments

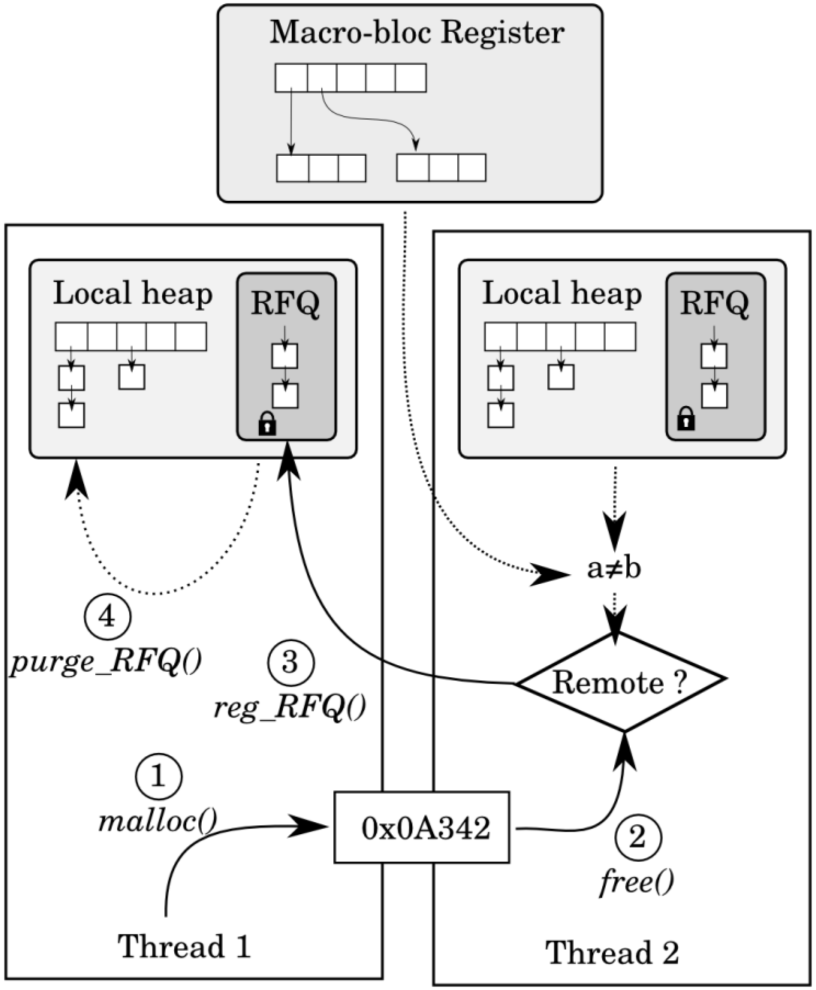
NUMA strategy

- 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)



Remote free without locks

- **Remote Free** :
  - Chunk allocated by a thread.
  - Freed by another thread.
- Commonly **implies locks** on **all local heaps**
- We use a **dedicated atomic queue (RFQ)**
- **RFQ flush** on **next** memory **operation**
- Tracking **ownership** with a **lockfree register**



Allocator Profiles

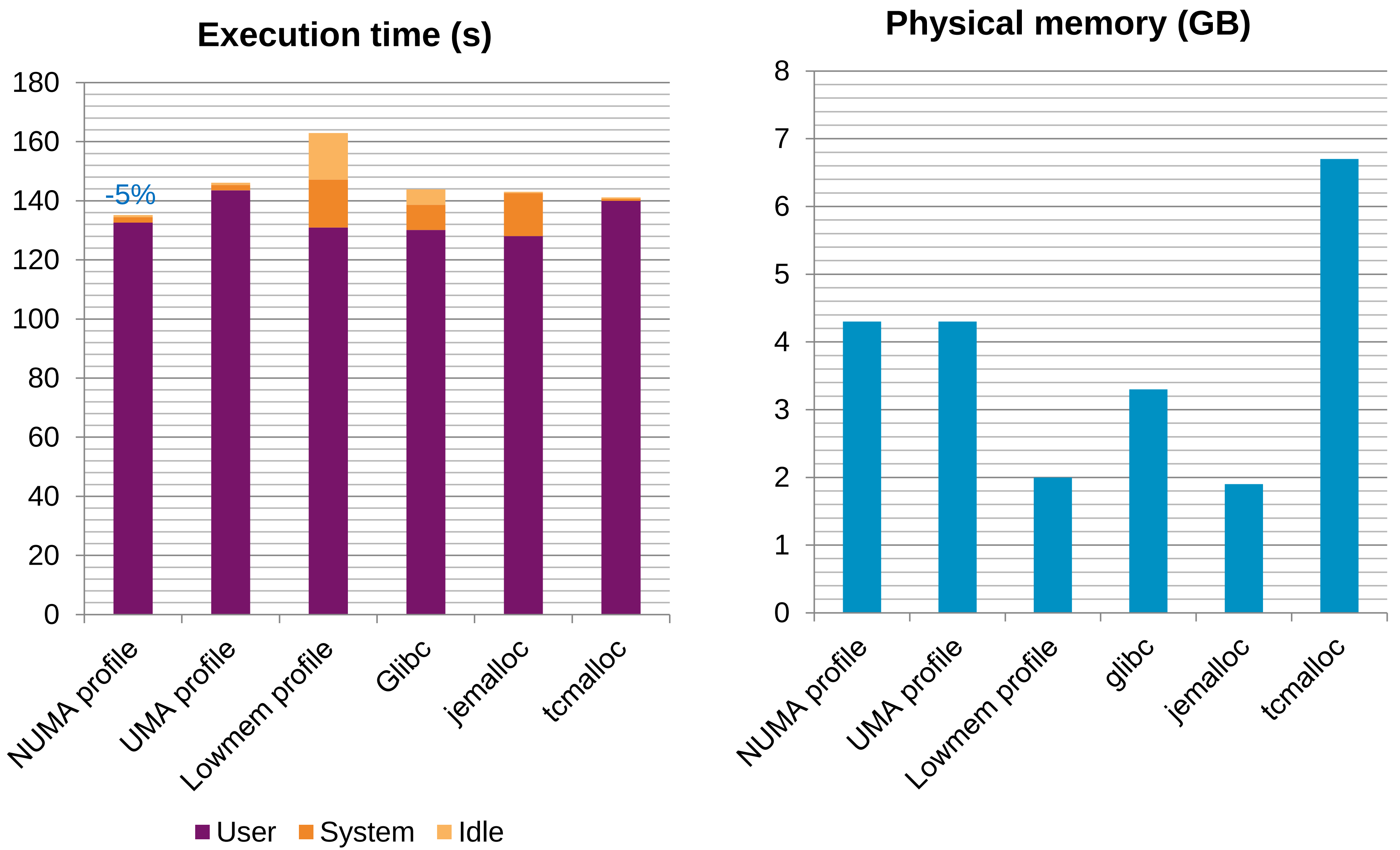
- Test allocator with **multiple profiles**

- **Lowmem** profile
  - Return memory to the OS as soon as possible

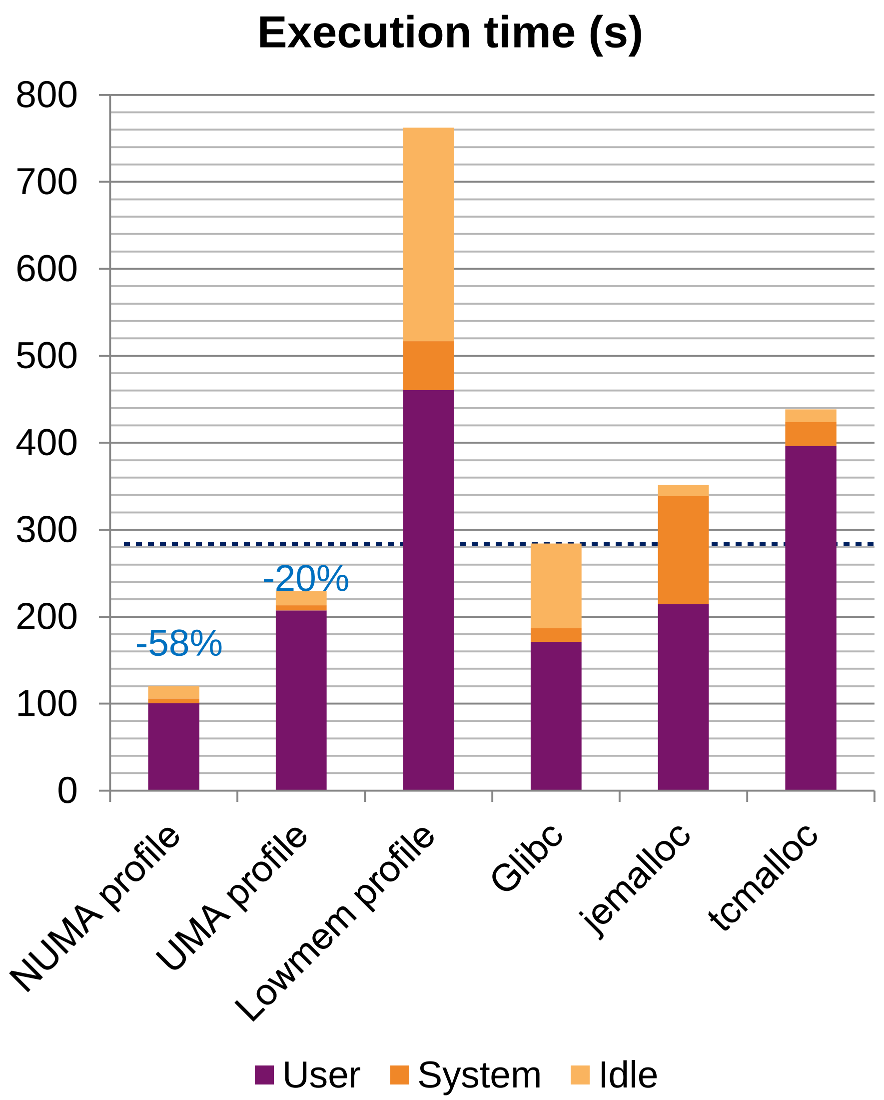
- **UMA** Profile
  - Recycle large segments
  - Disable NUMA
  - Use only one common memory source

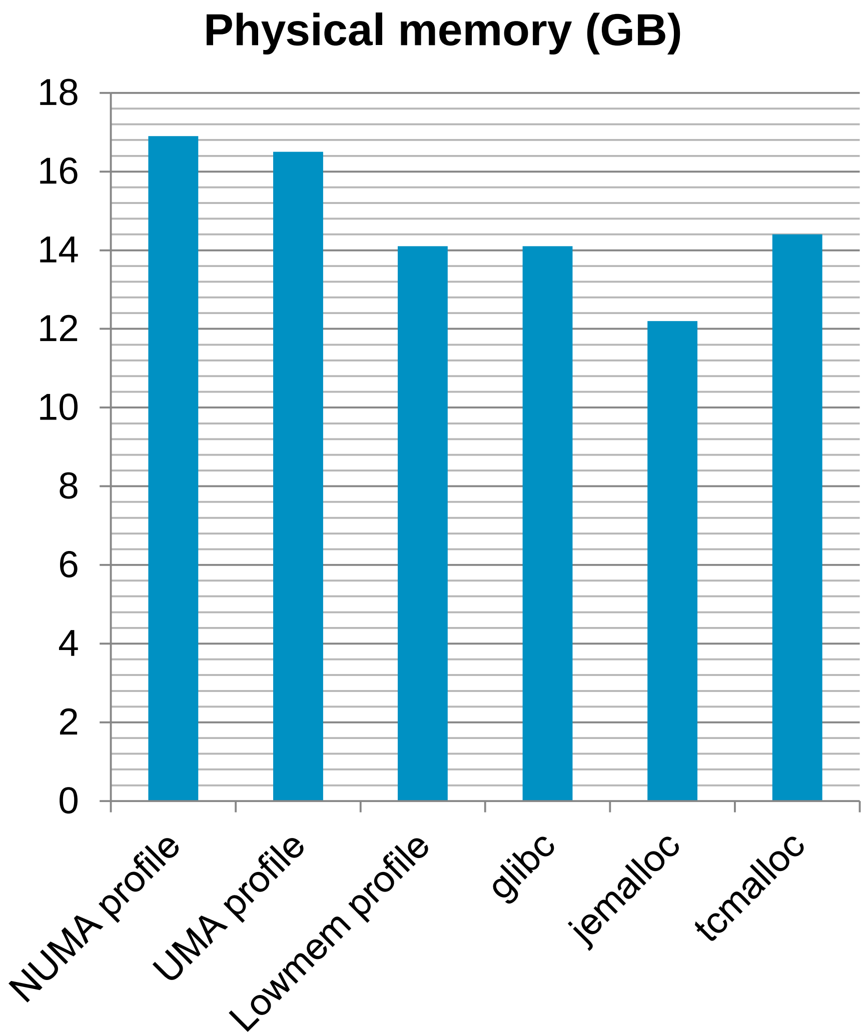
- **NUMA** profile :
  - Recycle large segments
  - Enable NUMA structures

Hera on bi-Westmere (12 : 2 * 6 cores)



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





OPTIMIZING LINUX PAGE FAULT HANDLER

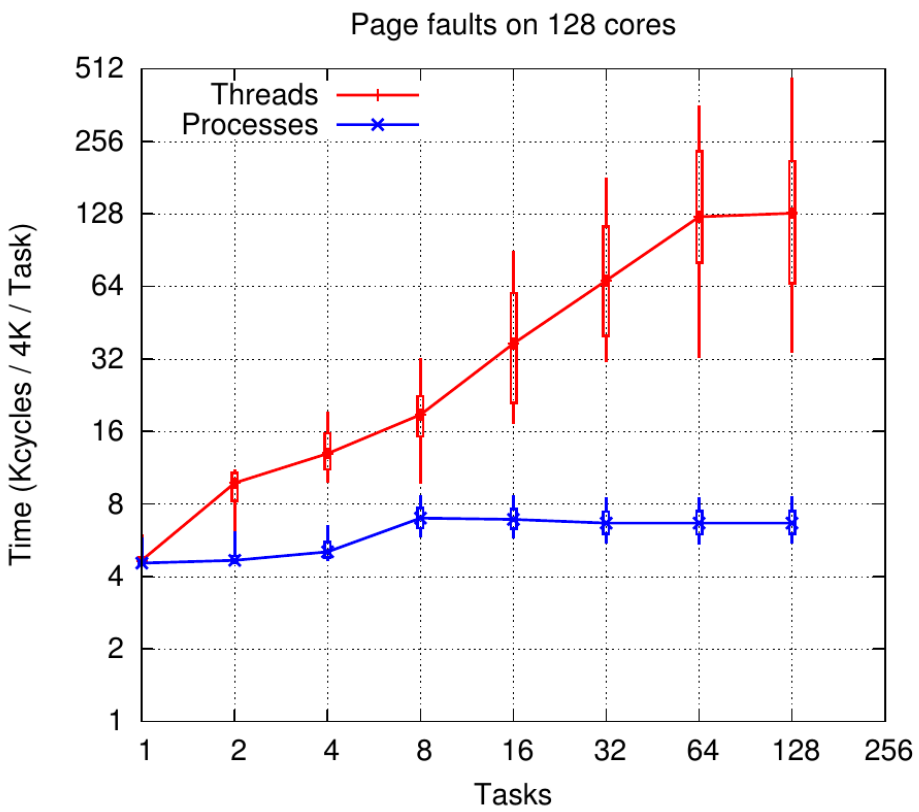
Benchmarking page faults

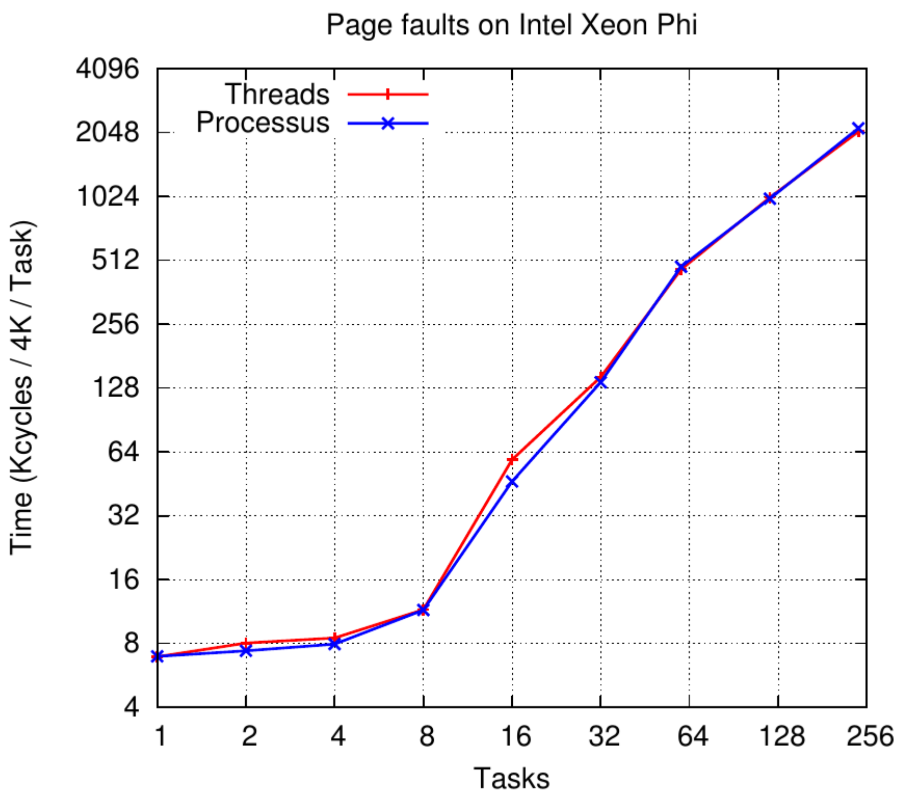
- **Page faults** are an issue for **allocation performance**
- We **previously limit them** with **large segment recycling**
- Can we **improve fault performance**?
- **Micro-benchmark** :

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

Page fault scalability

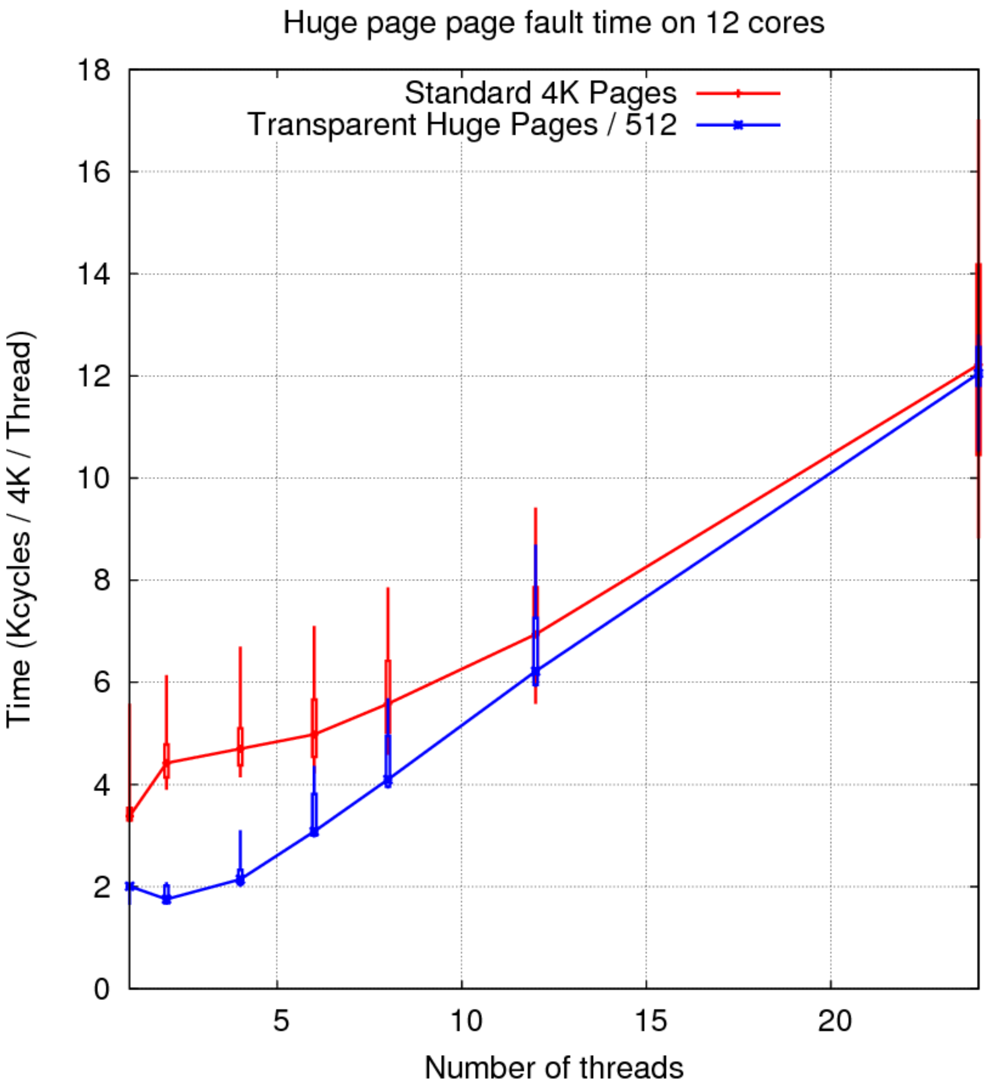
- **Are page faults scalable** ? Over threads or processes.
- Mesurement on **4*4 Nehalem-EP** (128 cores) and on **Xeon Phi** (60 cores)
- **Get scalability issue !**





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 Xeon Phi

~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 ?

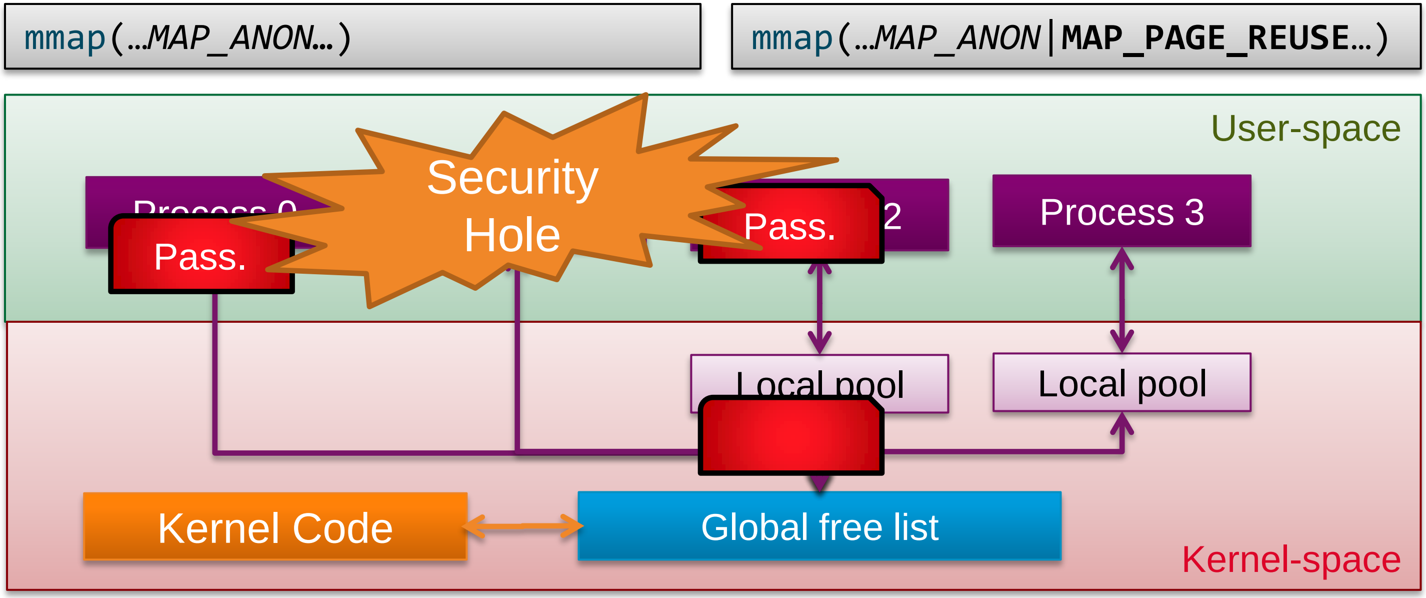
- Microsoft approach **:**
  - **Windows** uses a **system thread** to clear the memory
  - So its done **out** of **critical path**
- But **zeroing**:
  - Implies **useless work**
  - Consumes CPU **cycles** so **energy**
  - Consumes **memory bandwidth**
- **Allocation pattern** follow:

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

- Why not **avoid them** ?

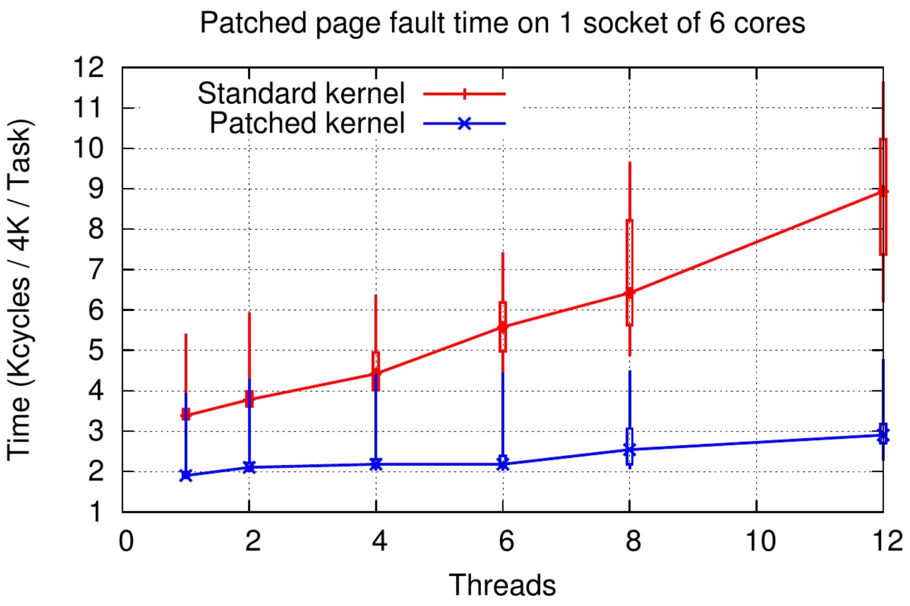
Reusing local pages to avoid zeroing

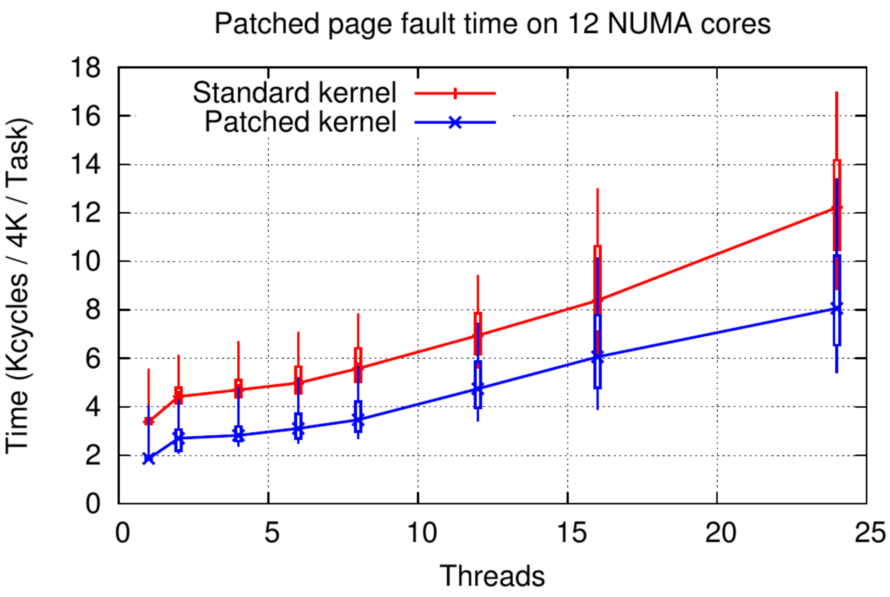
- Page zeroing is **required** for **security reason**
- It prevents information **leaks** from **another processes** or from the **kernel**.
- **But we can reuse pages locally !**
- Need to **extend** the **mmap semantic** :
- Usable by **malloc / realloc**.



Performance impact

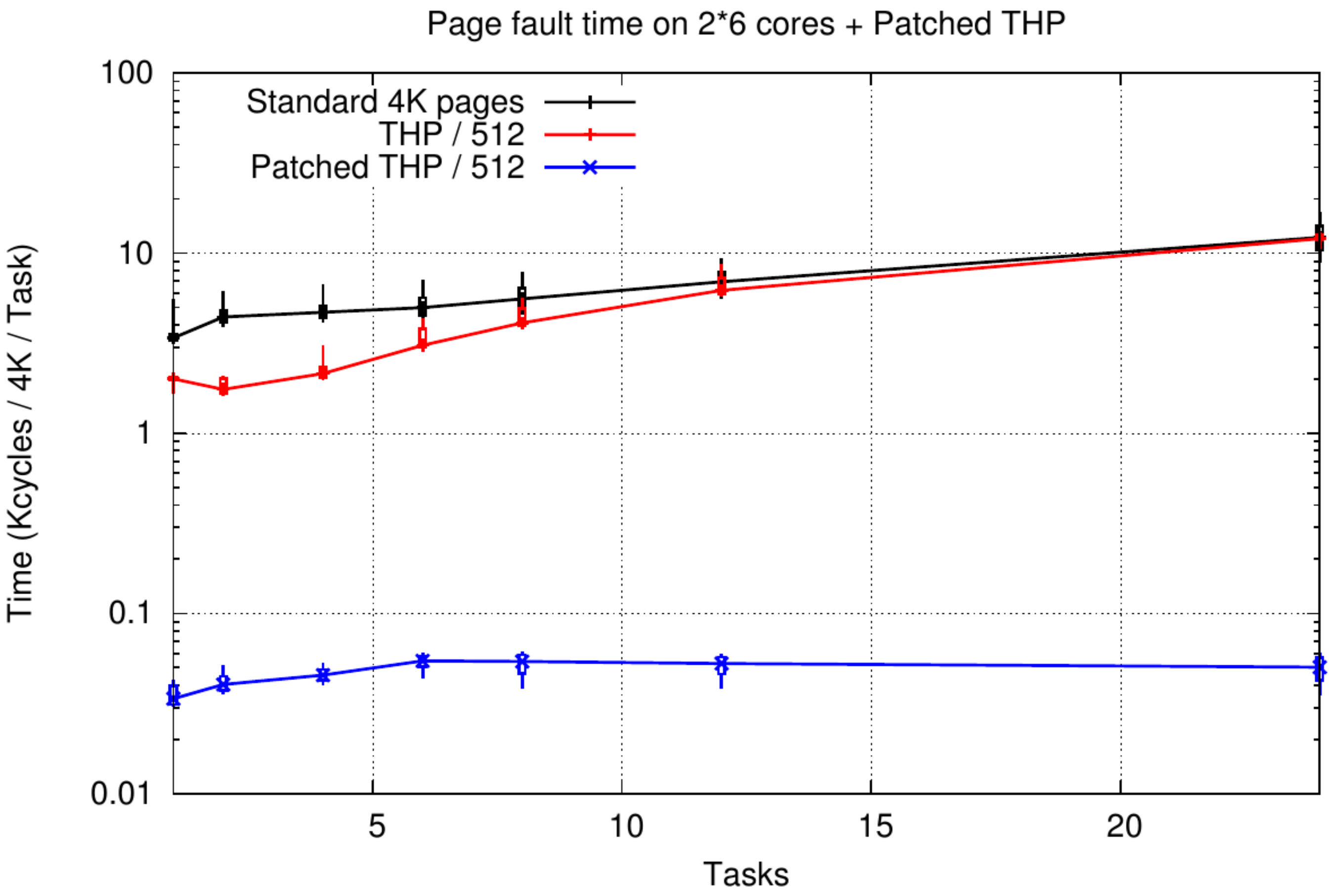
- Get the **expected improvement** on **4K pages** (40% for sequential).
- Also improve **scalability** on 1 socket
- On NUMA **locking effets become dominant for scalability**
- Get the constant improvement related to page zeroing.





Performance impact on huge pages

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



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

CONCLUSION AND FUTURE WORK

DE LA RECHERCHE À L'INDUSTRIE

Conclusion

Paging / alignment policies :

- **Avoid large alignments** in malloc.
- Need to avoid **regular coloring**.
- **Random paging** is **more robust** !
- **Huge pages** are regular by **hardware definition**.
- Need to **co-design malloc** and **OS** paging **policies**.

Malloc :

- Interest of **large allocation recycling**.
- **NUMA** support is required on large nodes.
- **Speed-up** of **2x** on Hera 128 cores.

Page faults (OS) :

- Observe a **scalability issue**.
- **40%** of fault time : **zeroing memory** !
- Proposal for a **semantic extension**.
- **New interest** for **huge pages** : **47x** !

Published articles :

[1] A Decremental Analysis Tool for Fine-Grained Bottleneck Detection (Partool 2010)
Souad Koliaï, Sébastien Valat, Tipp Moseley, Jean-Thomas Acquaviva, William Jalby

[2] Introducing Kernel-Level Page Reuse for High Performance Computing (MSPC 2013)
Sébastien Valat, Marc Pérache, William Jalby

Future work

Paging / coloring / alignments

- Implement **controlled non regular coloring**
- **Hardware mixing** inside **huge pages** ?
- **Linux huge pages**: be aware of **alignments** (**allocator / mmap**)
- Smaller huge page size ?

Page zeroing :

- **Cleanup** the patch (swap) and **discuss with community**
- **Hardware zeroing** done by **RAM** ?

Malloc :

- Using our **memory sources** and **NUMA strategy** inside **Jemalloc** ?
- Mix with **TCMalloc method** (madvise(DONT_NEED)) ?
- **Dynamic control of consumption / performance ratio**

QUESTIONS ?

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

↓

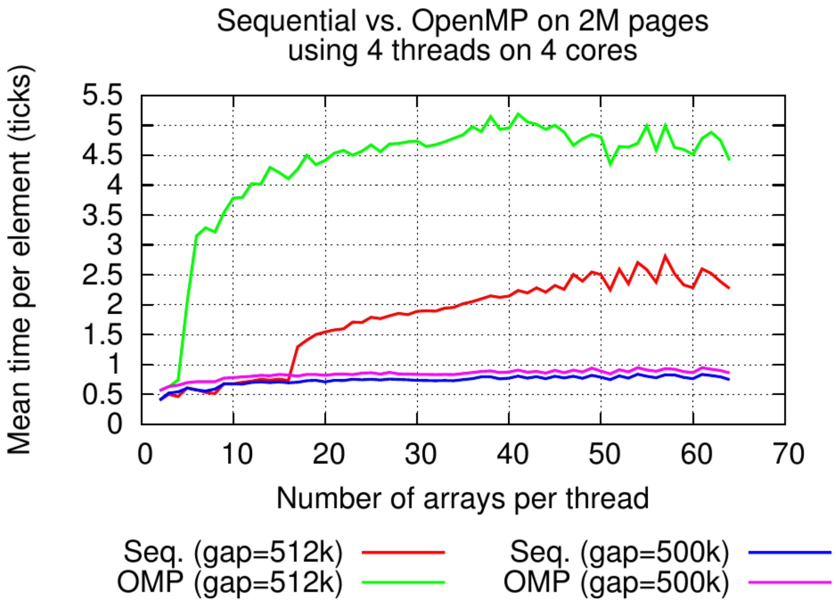
Hardware

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

BACKUPS

Solution

- The **Linux random** approach **prevents pathological cases**
- Do not use r**egular patterns** for **page coloring** (eg. **single modulo**)
- **Huge pages** are **regular** by **hardware definition**
- **Malloc** must **take care** of OS **paging strategy**
- **Malloc** must avoid **too large alignments**
- Existing **similar cases** for **4K alignments** (eg. L1 caches, 4K aliasing)



Kernel-space VS. user-space memory pools

Kernel-space advantages:

- Control the **physical memory**, not virtual one
- Follow the **real access pattern**
- **NUMA support** at page level, not segment
- Buffered memory **can be reclaimed** by kernel.

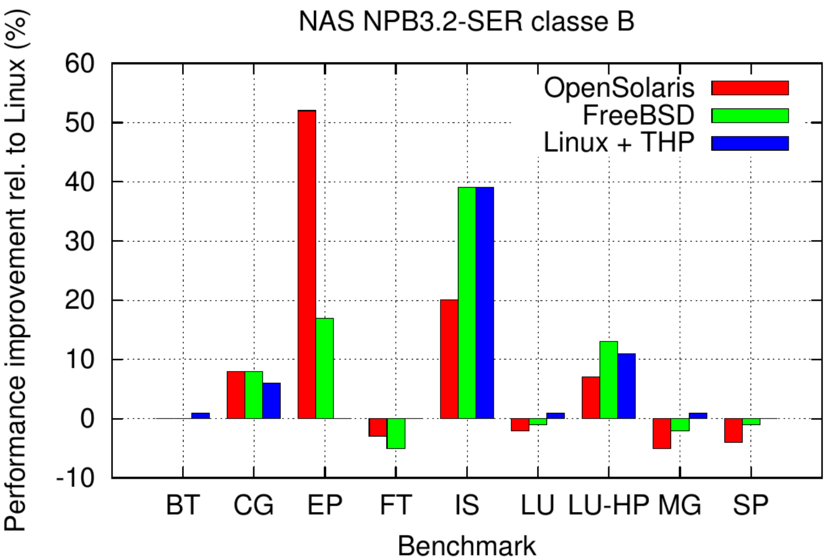
Limitations:

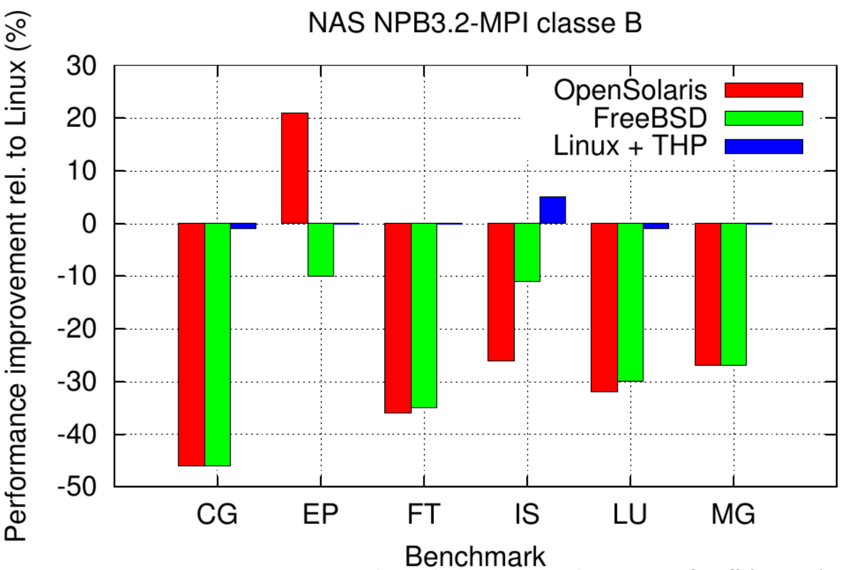
- **More efforts** to implement.
- Do not remove the **interruption** and **locking costs**

OS strategies comparison

- Each **system** has its default paging **strategy**:
- Is **Linux** slower due to **random paging** ?
- Tested architecture : **Nehalem bi-socket**
- Use a fixed compile chain : **GCC/Binutils/MPI/BLAS**

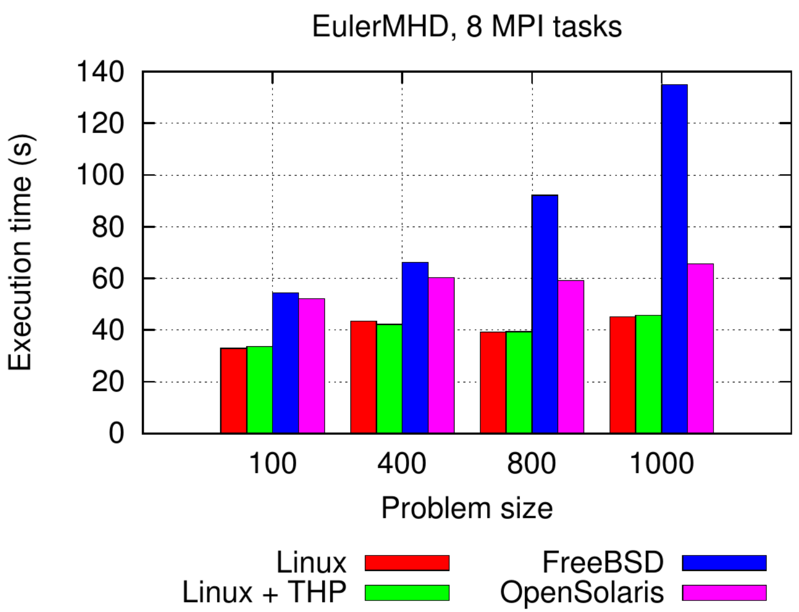
OS	Strategy
Linux	4K random
OpenSolaris	Page coloring
FreeBSD	Superpages

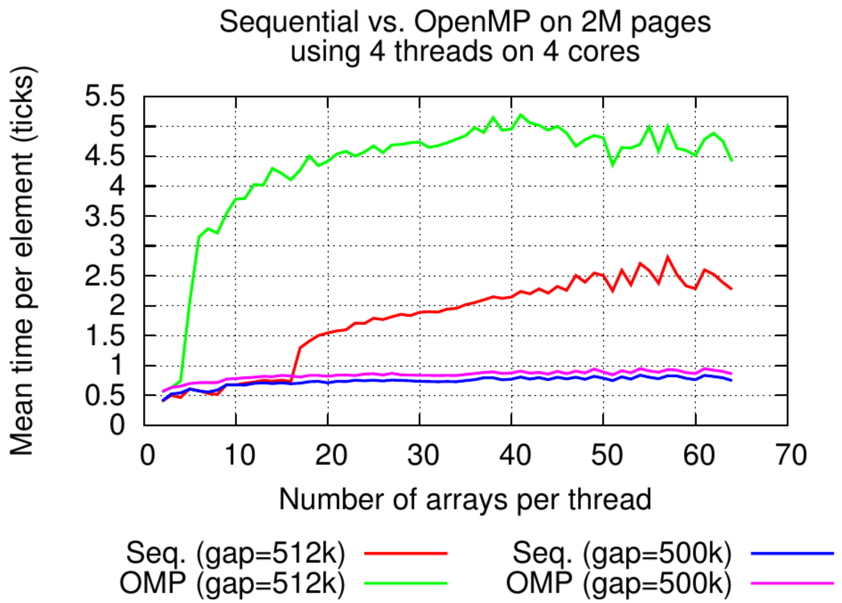




Impact on threads

- **Larger effects** on **shared caches** with threads/processes (Nehalem)
- EulerMHD : **Slowdown** up to **3x** on **FreeBSD**
- **16** ways L3 cache implies a maximum of **4 aligned arrays** per core
- **No limit** on concurrent arrays for **unaligned allocations**





4K aliasing

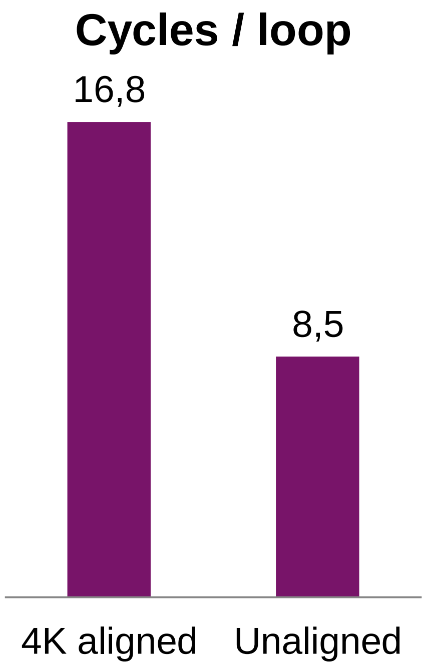
- Consider the simple loop :

```
for (i = 1 ; i < SIZE ; i++)
            a[i] = b[i-1]
```

- If addresses verify :

```
a % 4Ko = b % 4Ko
```

- It produces **false inter-iterations conflicts** between :
  - **store a[(i-1)]** from **i-1**
  - **load b[(i) - 1]** from **i**



- **Processor thinks** (fast check with 12 lower bits) **addresses are equals** (alias)

- Processor do **not execute** them in **parallel** (**out of order**)

- In malloc, **direct call to mmap** generate **4K alignment by default** !

Default fallback to mmap

- **Allocators commonly** use **mmap** for **large arrays**
- Call to mmap imply **alignment on page start** (4K)
- **It exposes them** to the issue for **large arrays** !
- **4K aliasing** was **fixed** on Sandy Bridge
- But **4K alignments** also create issue on **L1 associativity**
- **Allocator must avoid to force large alignments**

Report a list of similar issue

- Need to take care of **large alignments** on **regular page coloring**
- **Huge pages** are regular by **hardware definition**
- **Malloc** and **OS** politics **interact.**
- Studies **must consider the two.**
- We **reported other similar issues** (see the manuscript) :
  4K aliasing, L1 and TLB associativity

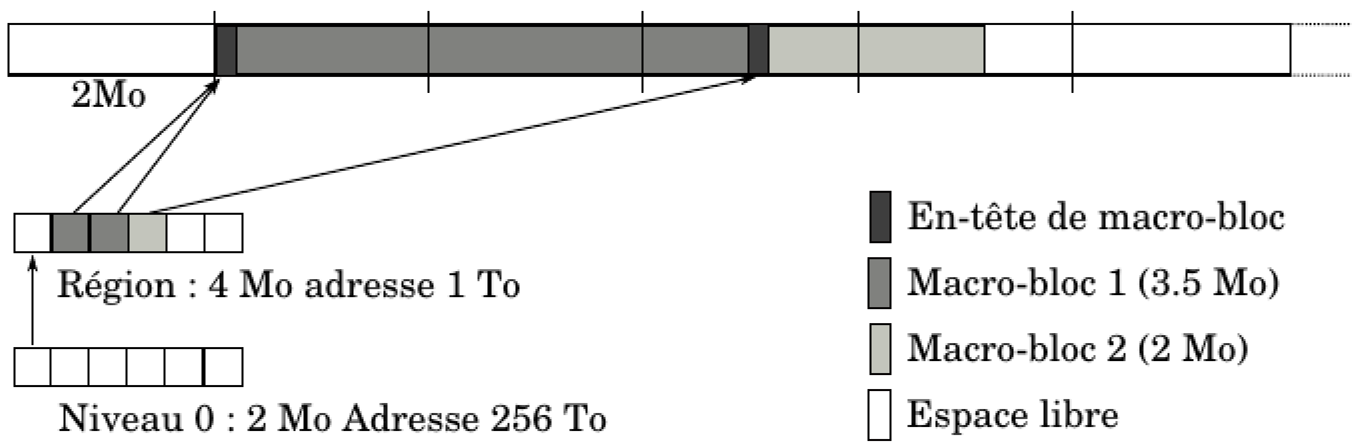
Impacte	Nom	Alignement	OMP	OS	Pages	Condition	Solutions	Probabilité
LL	Fuite dernier niveau de cache	-	-	Oui	4kB	– Utilisation de l'ensemble du dernier cache.	color, nrcolor, huge ou smcache	Elevé : Linux, Faible : SunOS
	OpenMP sur coloration régulière	$LLSS$	Oui	Oui	4 Ko	– SBA aligné relativement à $LLSS$ – $NBS > LLASSO$ – $NBTH <= CPUTH$	16bp, 4kp, nrcolor, nrsplit ou chnbs	Élevé : SunOS, Null : Linux
				Non	$>= LLSS$		16bp, 4kp, nrsplit ou chnbs	Moyen
L1 ? ,LL	Pagination régulière	$LLSS$, $L1SS$?	Non	Oui	4 Ko	– $NBS > LLASSO$ – SBA aligné sur $LLSS$ (ou à $L1SS$?)	16bp, 4kp, nrcolor ou chnbs	Élevé : SunOS, Null : Linux
				Non	$>= LLSS$		16bp, 4kp ou chnbs	Moyen
L1	Conflits Load/Store	4 Ko	Non	Non	? ? ?	– Utilisation d'accès de type a[i] = b[i-1]. – Tableaux alignés sur 4 Ko.	16bp ou chacc	Élevé
TLB, L1	Limite des PDE	PDEASIZE	Non	Non	4 Ko	– $NBS > TLBASSO$ – BSA aligné sur $TLBSASIZE$ – BSA distants de plus que $PDEASIZE/NBS$	16bp, 4kp ou chnbs	Faible
hline TLB	Limit d'associativité du DTLB	$TLBSASIZE$	Non	Non	4 Ko	– BSA aligné sur $TLBSASIZE$ – $NBS > TLBASSO$	16bp, 4kp ou chnbs	Moyen

Small / large allocations

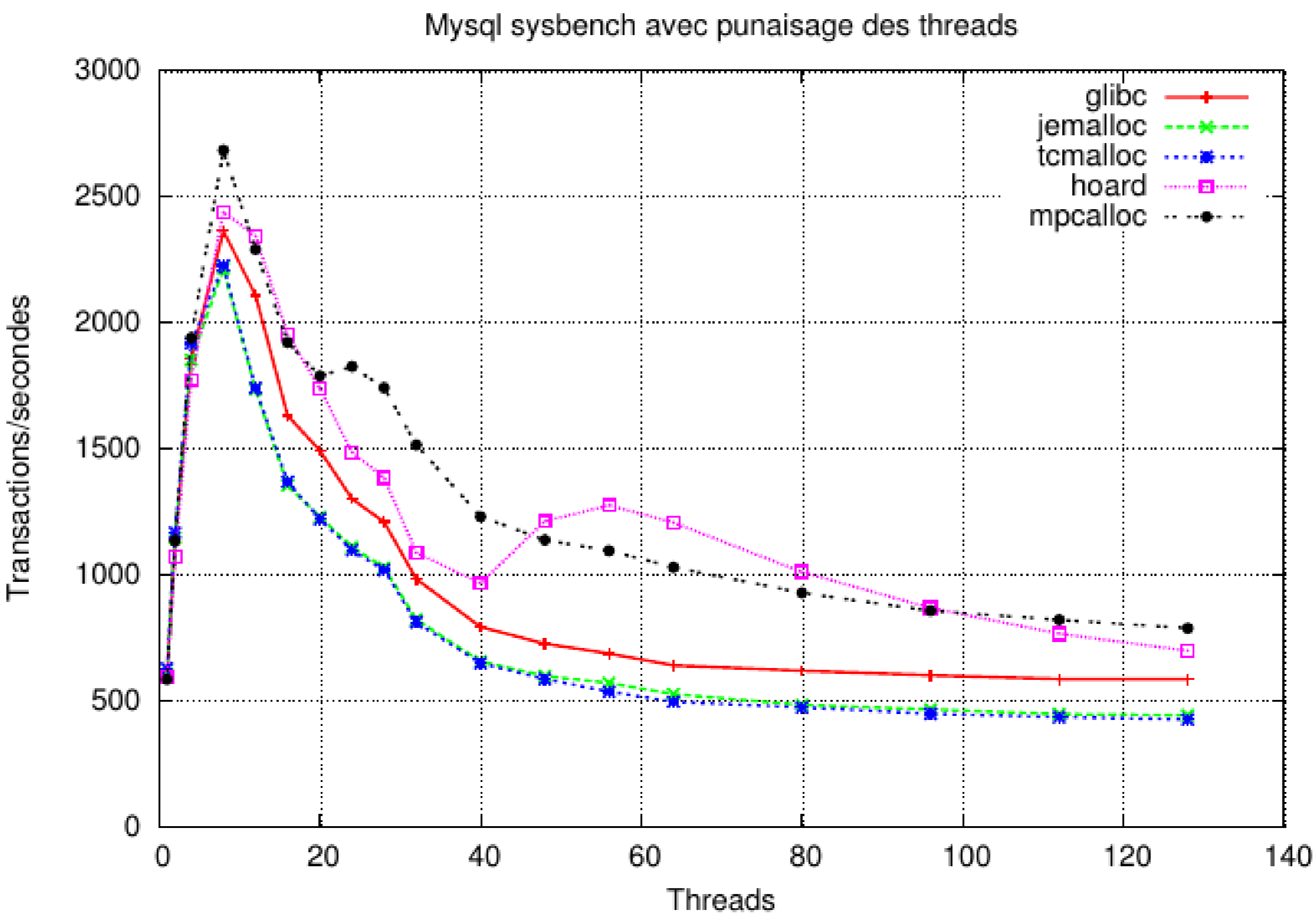
- Cost for **large allocation : page faults.**
- **Commonly neglected**, literature mainly discuss small allocations
- Direct call to **mmap/munmap**
- **HPC applications** (expected to) use **large arrays**
- **Goals :**
  - **Recycle** large arrays
  - Avoid **fragmentation** on large segments
  - Take care of **NUMA**
  - Limit **locks**

Parenté des blocs

- A l'appel à free**, à quel tas appartient le bloc** ?
- Ajout d'un **registre** pour retrouver **l'appartenance des blocs**
- Approche type table des pages.
- **Pas de verrous** contrairement aux arbres.
  - **Unicité** des adresses renvoyées par **mmap**.
  - **Un seul macro-bloc** peu couvrir **une entrée**.
  - Pas de **suppression des niveaux intermédiaires**.



Mysql results



kernel-space VS. user-space memory pools

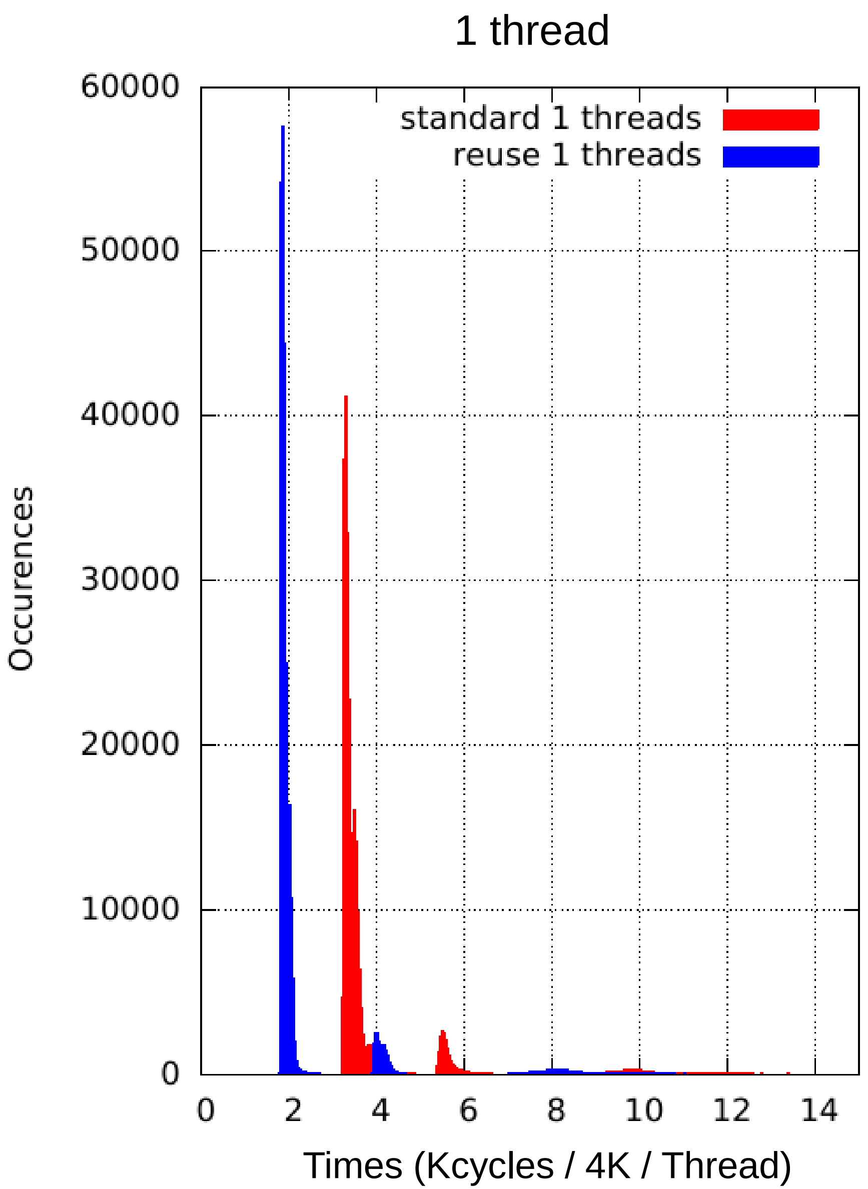
Kernel-space advantages:

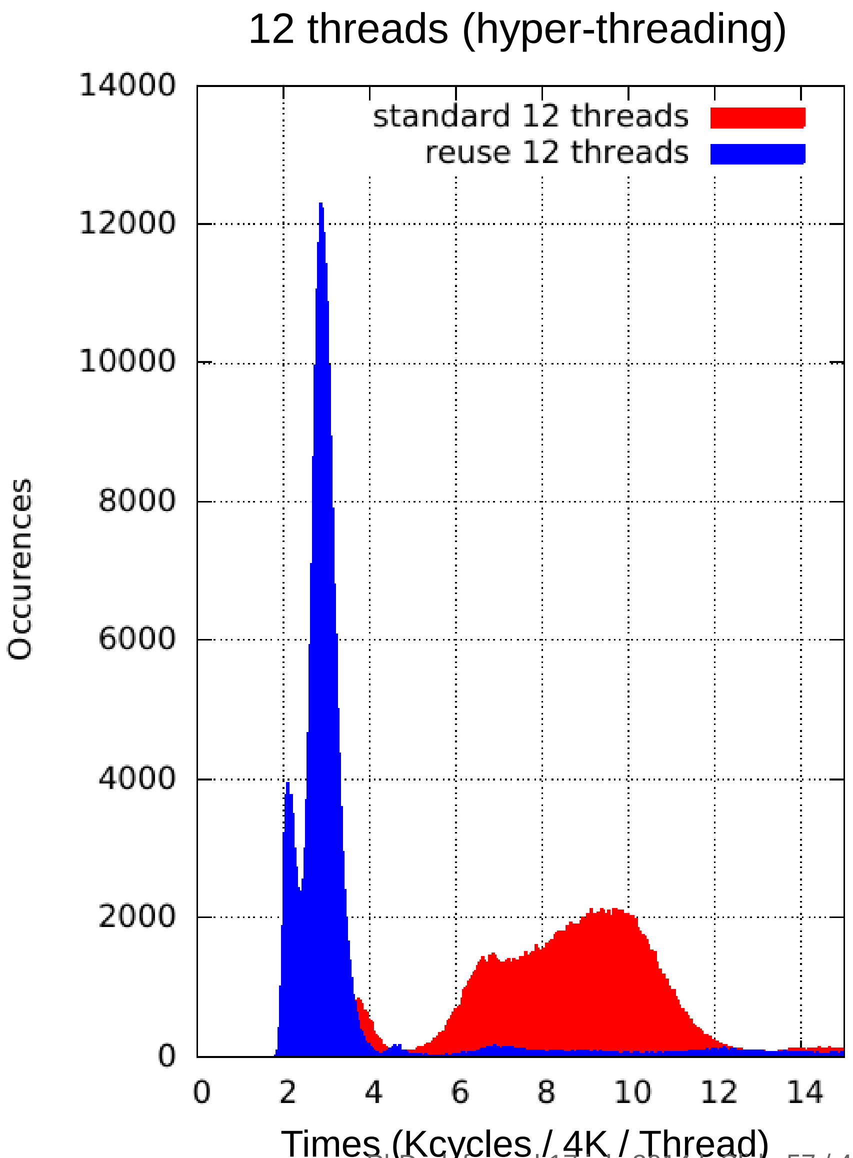
- Control the **physical memory**, not virtual one
- Follow the **real access pattern**
- **NUMA support** at page level, not segment
- Buffered memory **can be reclaimed** by kernel.

Limitations:

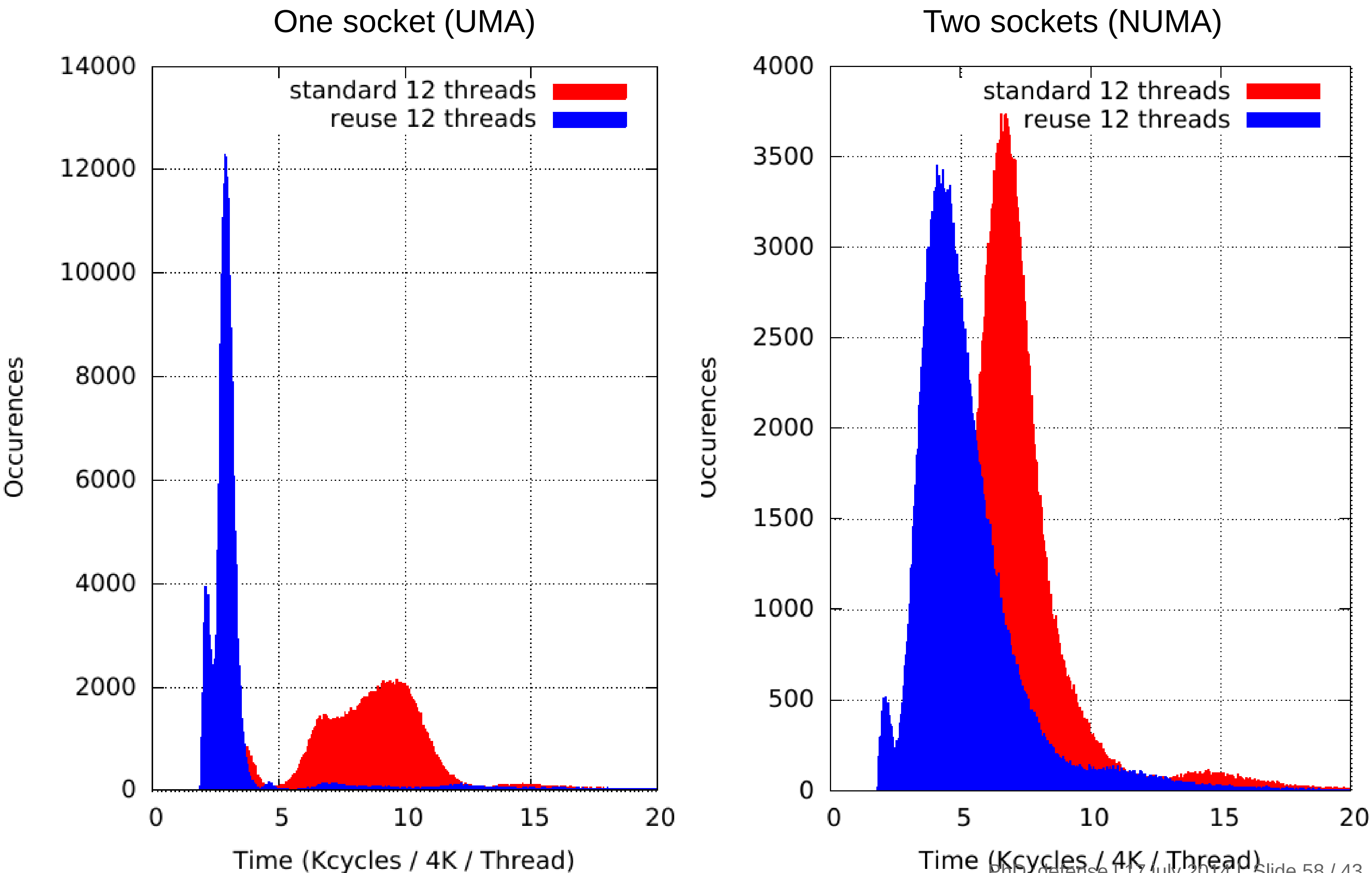
- **More efforts** to implement.
- Do not remove the **interruption** and **locking costs**

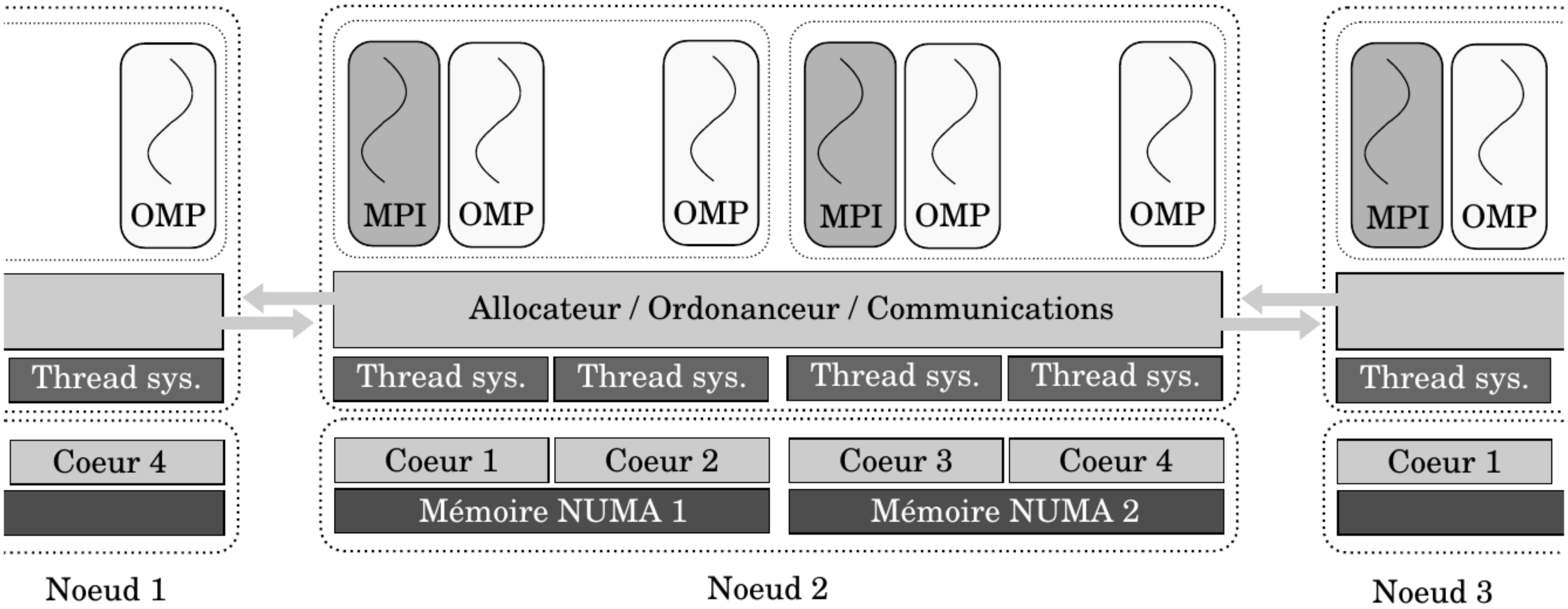
Improvement of faults on 6 core westmere

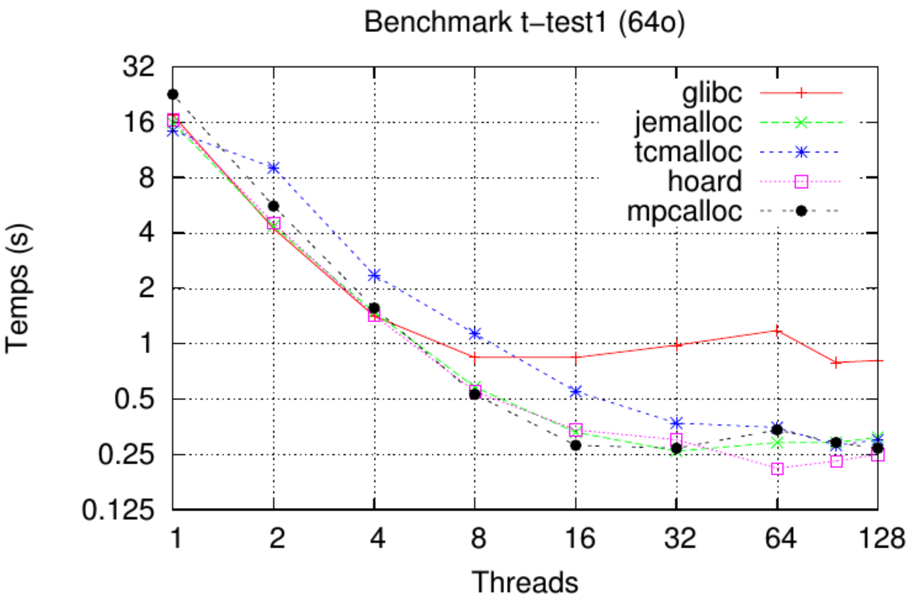


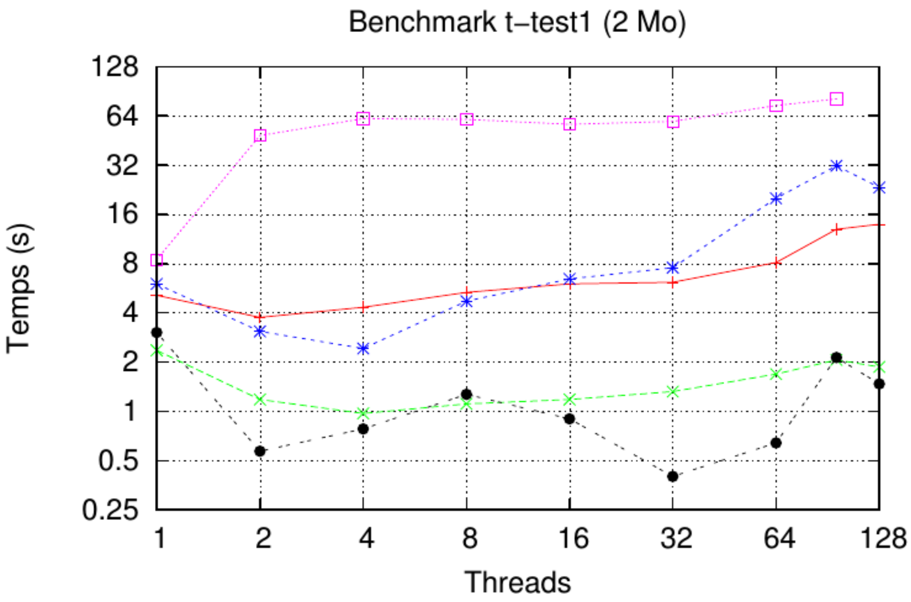


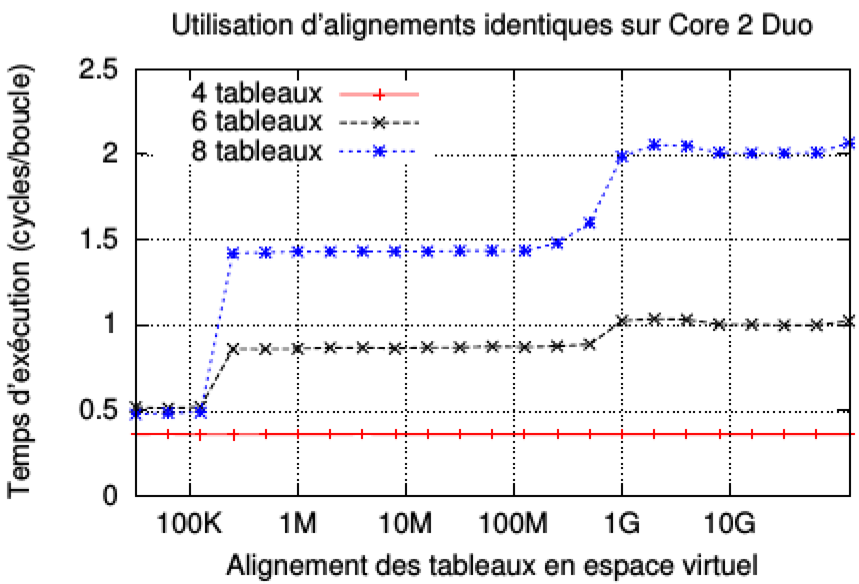
Using two sockets (NUMA)



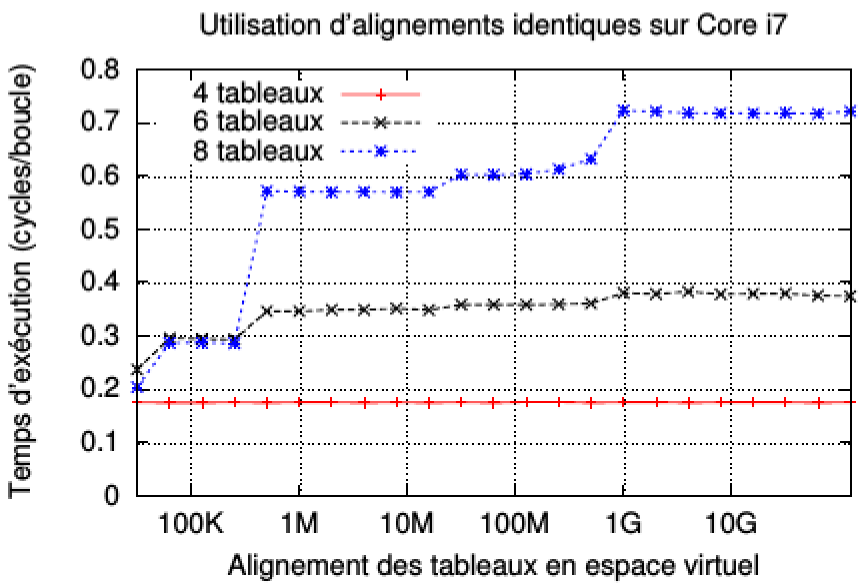

OMP
MPI
OMP
OMP
MPI
OMP
OMP
MPI
OMP
Allocateur / Ordonanceur / Communications
Thread sys.
Thread sys.
Thread sys.
Thread sys.
Thread sys.
Thread sys.
Coeur 4
Coeur 1
Coeur 2
Coeur 3
Coeur 4
Coeur 1
Mémoire NUMA 1
Mémoire NUMA 2
Noeud 1
Noeud 2
Noeud 3


Benchmark t−test1 (64o)
Temps (s)
Threads
glibc
jemalloc
tcmalloc
hoard
mpcalloc
32
16
8
4
2
1
0.5
0.25
0.125
1
2
4
8
16
32
64
128


Benchmark t−test1 (2 Mo)
Temps (s)
Threads
128
64
32
16
8
4
2
1
0.5
0.25
1
2
4
8
16
32
64
128


Utilisation d'alignements identiques sur Core 2 Duo
Temps d'exécution (cycles/boucle)
2.5
2
1.5
1
0.5
0
4 tableaux
6 tableaux
8 tableaux
100K
1M
10M
100M
1G
10G
Alignement des tableaux en espace virtuel

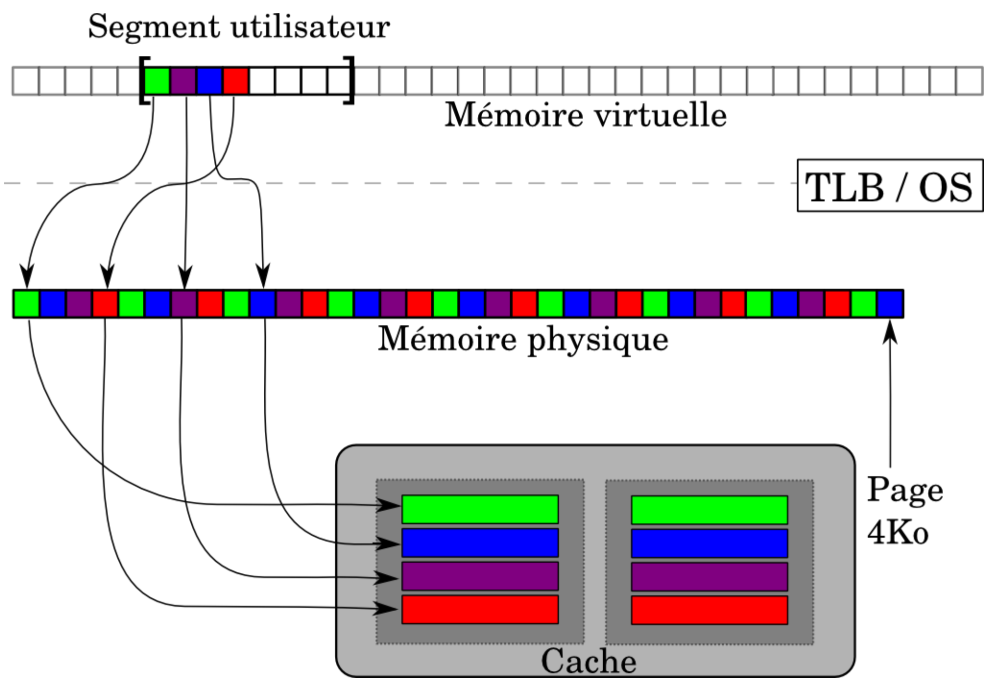
(a)

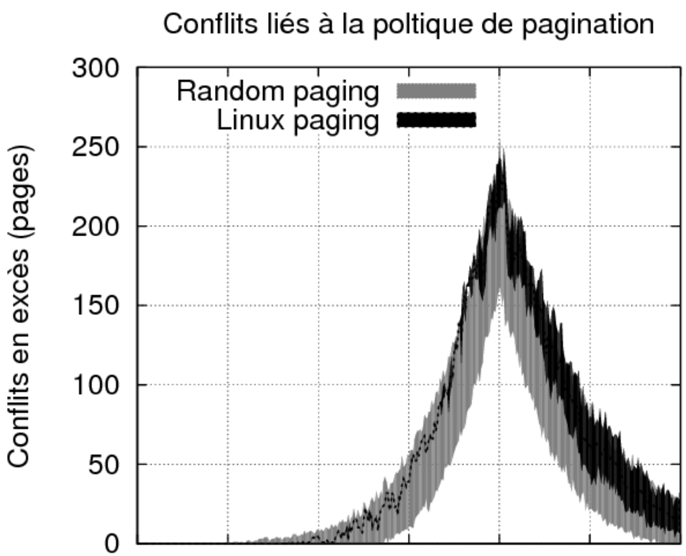

Utilisation d'alignements identiques sur Core i7
Temps d'exécution (cycles/boucle)
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
4 tableaux
6 tableaux
8 tableaux
100K
1M
10M
100M
1G
10G
Alignement des tableaux en espace virtuel

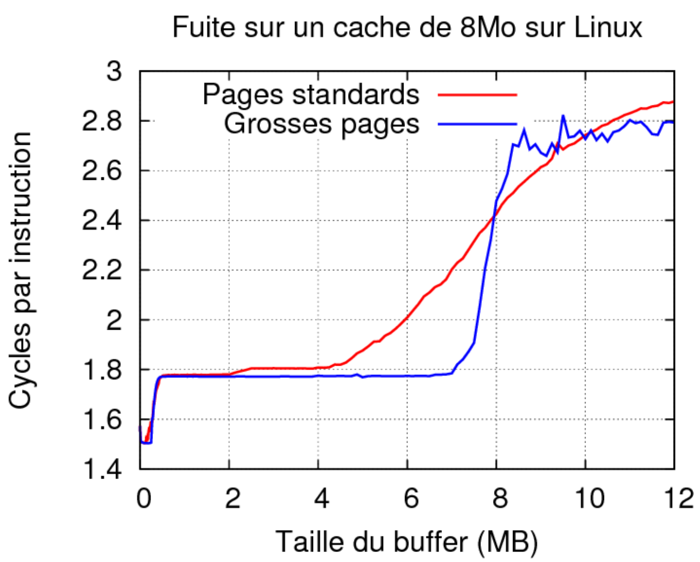
(b)

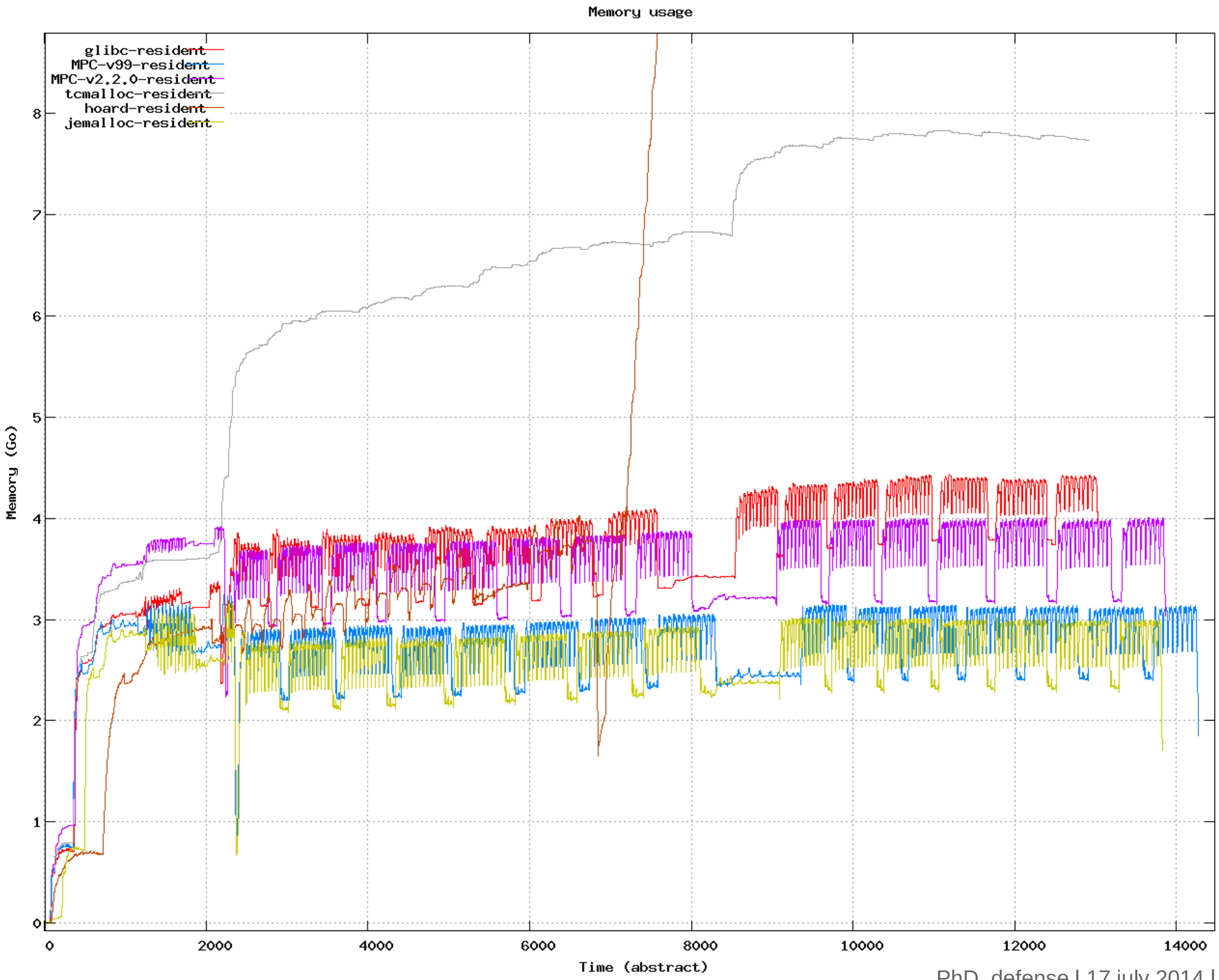
Associativité et coloration de pages

- Les cache sont **associatifs**
- Les données sont **placées** suivant leur **adresse**.
- Des **conflits** possibles générés par l'OS
- Coloration de page, habituellement, modulo :








Memory usage
glibc-resident
MPC-v99-resident
MPC-v2.2.0-resident
tcmalloc-resident
hoard-resident
jemalloc-resident
Memory (Go)
0
1
2
3
4
5
6
7
8
0
2000
4000
6000
8000
10000
12000
14000
Time (abstract)