OS IMPACT ON PERFORMANCE

OPENING THE ROOF

LAPP - Annecy

Sébastien Valat - INRIA

Gray Scott Reloaded – Summer School - 11 july 2024

Plan

I. Introduction

II. Analysis of OS paging policy

III. NUMA allocator for HPC applications

IV. Page zeroing in Linux first touch handler

V. Conclusion

ADVERTISING

Understanding the memory management

- **From the transistor to the application**

- **What every programmer should know about memory** (Urlich Drepper)
  https://people.freebsd.org/~lstewart/articles/cpumemory.pdf

SCAN ME

What Every Programmer Should Know About Memory

Ulrich Drepper
Red Hat, Inc.
drepper@redhat.com

November 21, 2007

Abstract

As CPU cores become both faster and more numerous, the limiting factor for most programs is now, and will be for some time, memory access. Hardware designers have come up with ever more sophisticated memory handling and acceleration techniques–such as CPU caches–but these cannot work optimally without some help from the programmer. Unfortunately, neither the structure nor the cost of using the memory subsystem of a computer or the caches on CPUs is well understood by most programmers. This paper explains the structure of memory subsystems in use on modern commodity hardware, illustrating why CPU caches were developed, how they work, and what programs should do to achieve optimal performance by utilizing them.

1 Introduction

In the early days computers were much simpler. The various components of a system, such as the CPU, memory, mass storage, and network interfaces, were developed together and, as a result, were quite balanced in their performance. For example, the memory and network interfaces were not (much) faster than the CPU at providing data.

This situation changed once the basic structure of computers stabilized and hardware developers concentrated on optimizing individual subsystems. Suddenly the performance of some components of the computer fell significantly behind and bottlenecks developed. This was especially true for mass storage and memory subsystems which, for cost reasons, improved more slowly relative to other components.

The slowness of mass storage has mostly been dealt with using software techniques: operating systems keep most often used (and most likely to be used) data in main memory, which can be accessed at a rate orders of magnitude faster than the hard disk. Cache storage was added to the storage devices themselves, which requires no changes in the operating system to increase performance.[1] For the purposes of this paper, we will not go into more details of software optimizations for the mass storage access.

Unlike storage subsystems, removing the main memory as a bottleneck has proven much more difficult and almost all solutions require changes to the hardware. Today these changes mainly come in the following forms:

- RAM hardware design (speed and parallelism).
- Memory controller designs.
- CPU caches.
- Direct memory access (DMA) for devices.

For the most part, this document will deal with CPU caches and some effects of memory controller design. In the process of exploring these topics, we will explore DMA and bring it into the larger picture. However, we will start with an overview of the design for today's commodity hardware. This is a prerequisite to understanding the problems and the limitations of efficiently using memory subsystems. We will also learn about, in some detail, the different types of RAM and illustrate why these differences still exist.

This document is in no way all inclusive and final. It is limited to commodity hardware and further limited to a subset of that hardware. Also, many topics will be discussed in just enough detail for the goals of this paper. For such topics, readers are recommended to find more detailed documentation.

When it comes to operating-system-specific details and solutions, the text exclusively describes Linux. At no time will it contain any information about other OSes. The author has no interest in discussing the implications for other OSes. If the reader thinks s/he has to use a different OS they have to go to their vendors and demand they write documents similar to this one.

One last comment before the start. The text contains a number of occurrences of the term "usually" and other, similar qualifiers. The technology discussed here exists

[1] Changes are needed, however, to guarantee data integrity when using storage device caches.

https://people.freebsd.org/~lstewart/articles/cpumemory.pdf

A look for HPC / supercomputers

- **My PhD. work**

- Contribution à l'amélioration des méthodes d'optimisation de la gestion de la mémoire dans le cadre du Calcul Haute Performance

https://hal.science/tel-01253537

THÈSE DE DOCTORAT DE
L'UNIVERSITÉ DE VERSAILLES SAINT-QUENTIN EN YVELINES

Spécialité

Informatique

à l'École doctorale des Science et Technologie de Versailles (STV)

présentée pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ DE VERSAILLES

et intitulée

Contribution à l'amélioration des méthodes d'optimisation de la gestion de la mémoire dans le cadre du Calcul Haute Performance.

par **Sébastien Valat**

Organisme d'accueil :

CEA, DAM, DIF
F-91297 Arpajon France
Département des Sciences de la Simulation et de l'Information (DSSI)

Soutenue publiquement le 17 juillet 2014
devant le jury composé de :

M. William Jalby	Directeur de thèse	Professeur à l'université de Versailles
M. Marc Pérache	Encadrant	Ingénieur chercheur au CEA.DAM.DIF
M. Allen D. Malony	Président de jury	Professeur à l'université de l'Oregon
M. Jean-François Méhaut	Rapporteur	Professeur à l'université Joseph Fourier
M. Alfredo Goldman	Rapporteur	Professeur à l'université de São Paulo
M. Emmanuel Jeannot	Examinateur	Directeur de recherche INRIA

SCAN ME

OPERATING SYSTEM (OS)

BETWEEN APPLICATIONS AND HARDWARE

The keywords of the last two weeks

Loops, loops, loops

Buffer

GPU

CPU

Cores

NUMA

SYCL

Vectorization

SSE / AVX

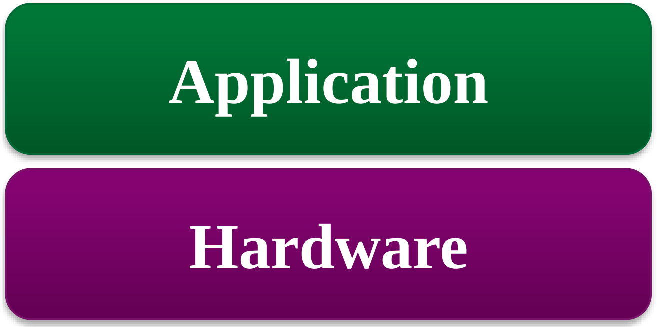
Prefetcher

Registres

wrap

Hardware and software stack

- **To obtain performance**
- We need to **optimize** the **interaction** between **all components**
- **The synthetic view up to now :**



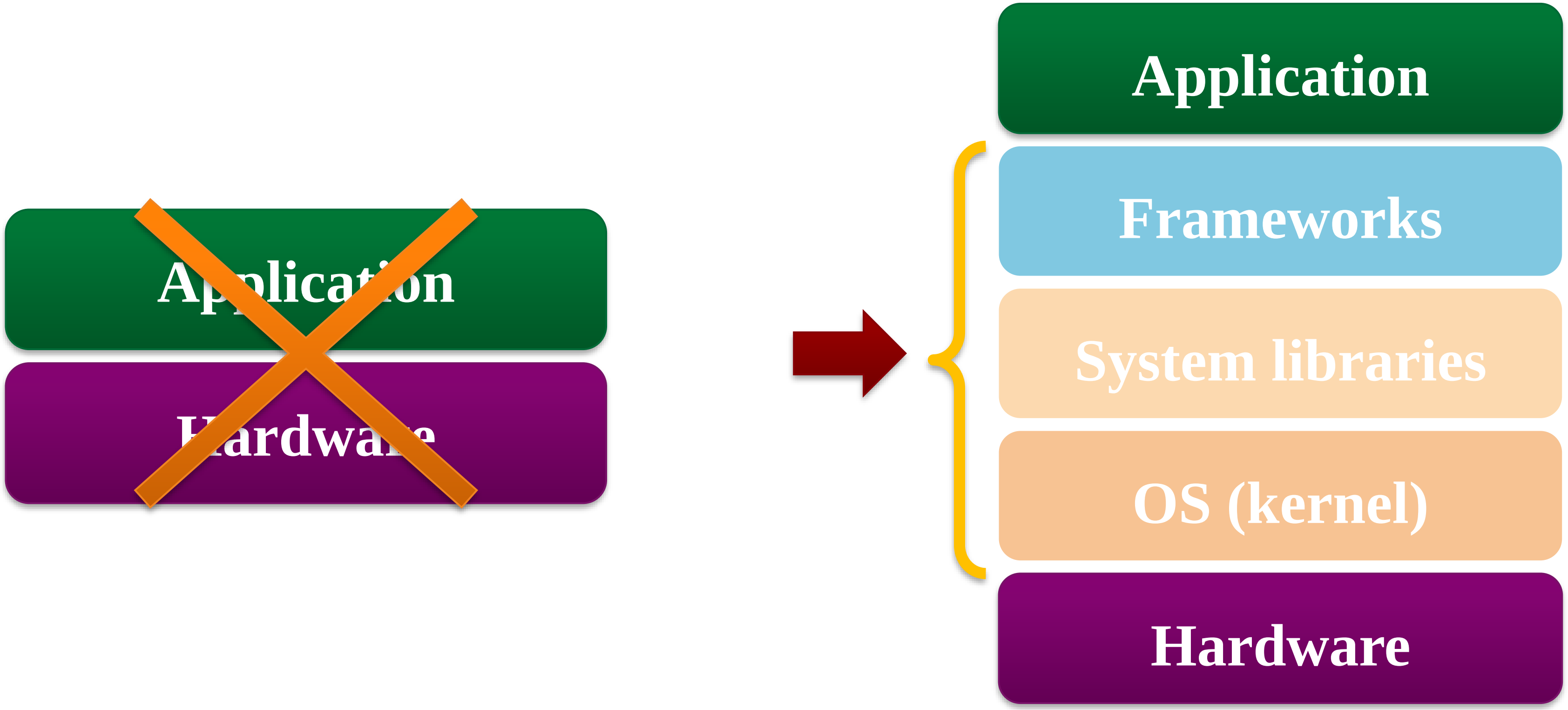
Application

Hardware

https://press.ariane.group/le-nouveau-lanceur-europeen-ariane-6-a-pris-son-envol/?lang=fra

Hardware and software stack

- **Non optimal hardware usage** leads to **slow do**wn,
- We **didn'**t be in **direct contact** to the **hardware**.
- **Bad usage** of **OS too**.

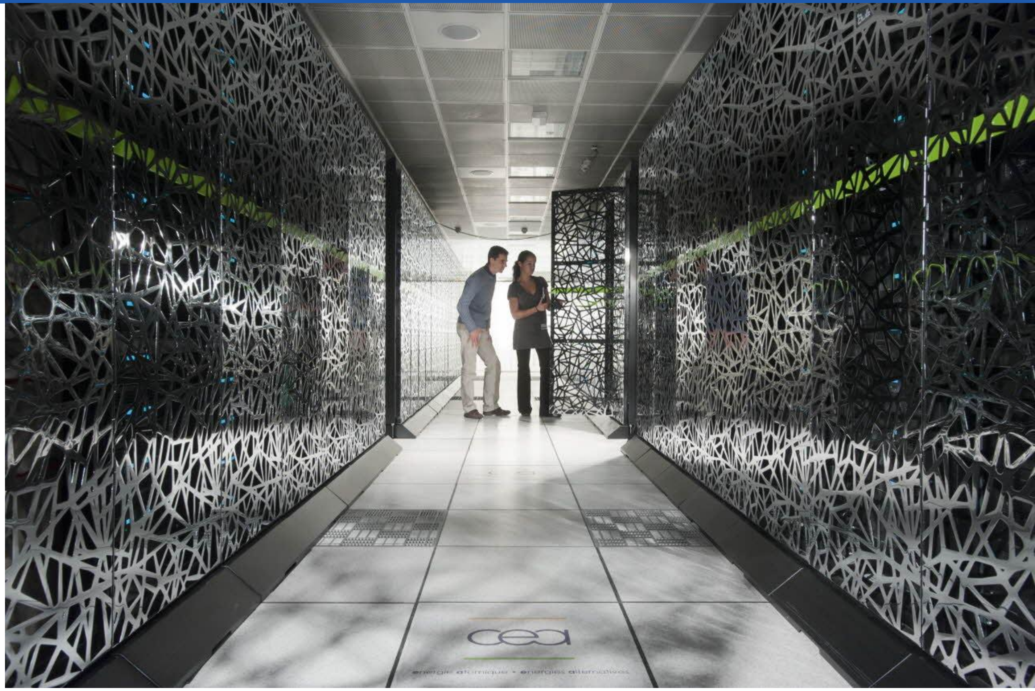


I. Introduction
II. Analysis of OS paging policy
III. NUMA allocator for HPC applications
IV. Page zeroing in Linux first touch handler
V. Conclusion

INTRODUCTION

Context : HPC

- **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 **memory per core**

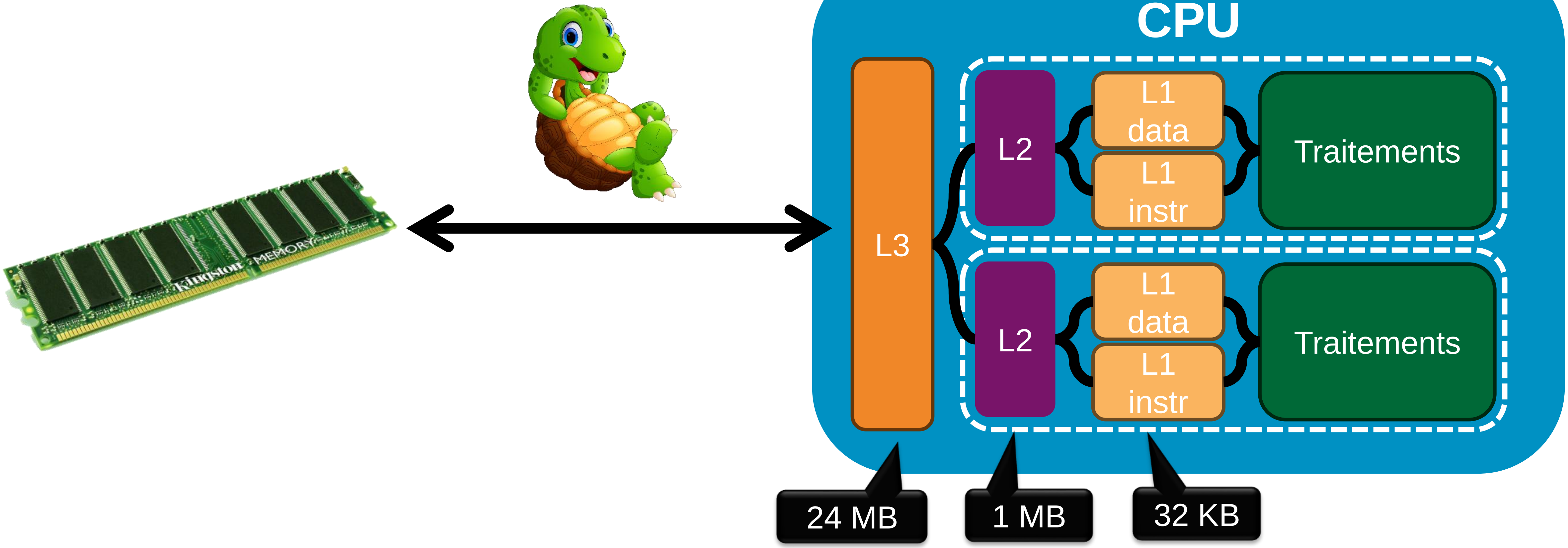


http://www.cea.fr/multimedia/Pages/galeries/defense/Tera-100.aspx

LES CACHES

Damn slow memory

- **A story** of **caches** and **hierarchy**

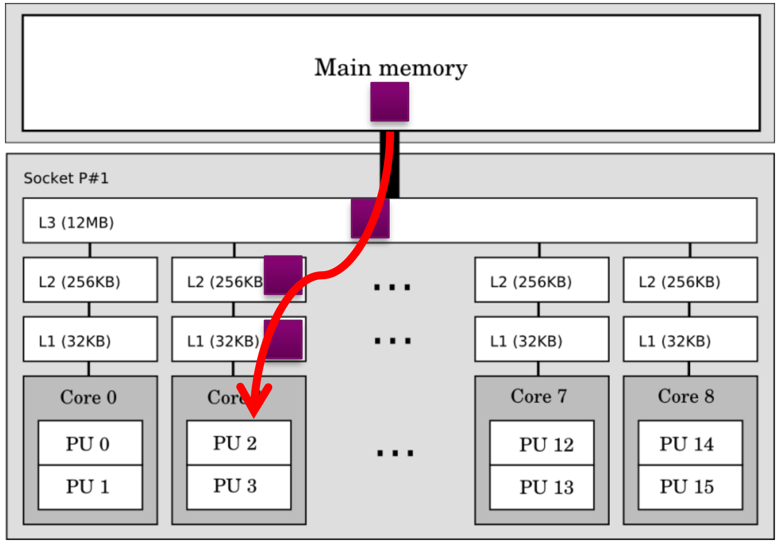


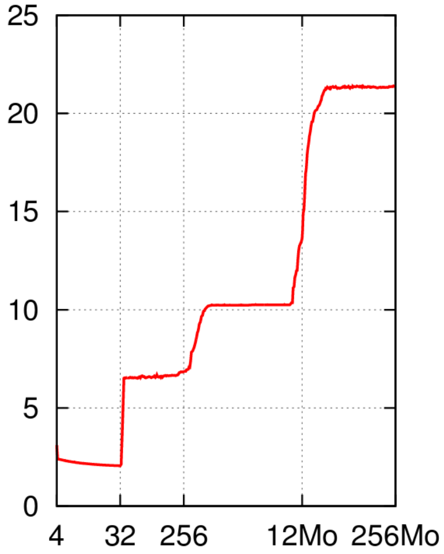
https://similarpng.com/cartoon-funny-turtle-isolated-on-transparent-background-png/#getdownload

Architecture

- Computer science : **operations** & **<u>data</u>**
- Multiple **memory levels**
- Hierarchical **caches**
- *Pre-fetcher*

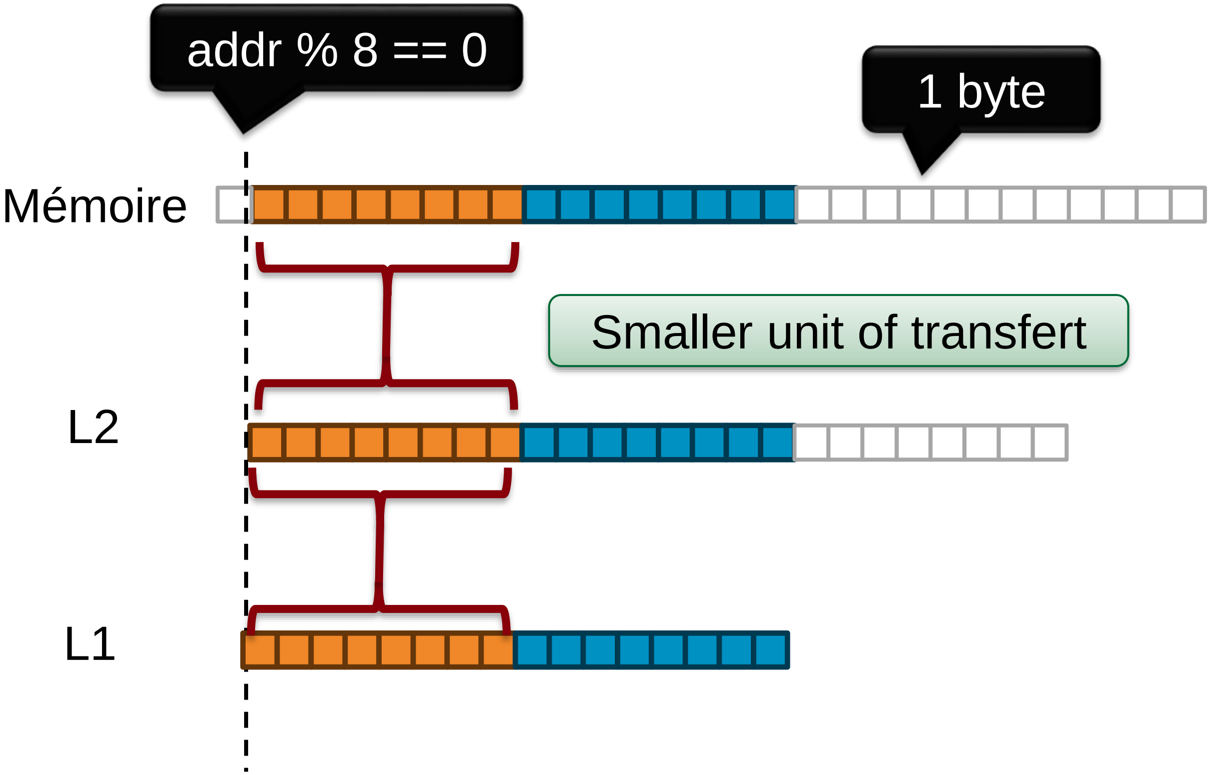
Processor : 8 cores



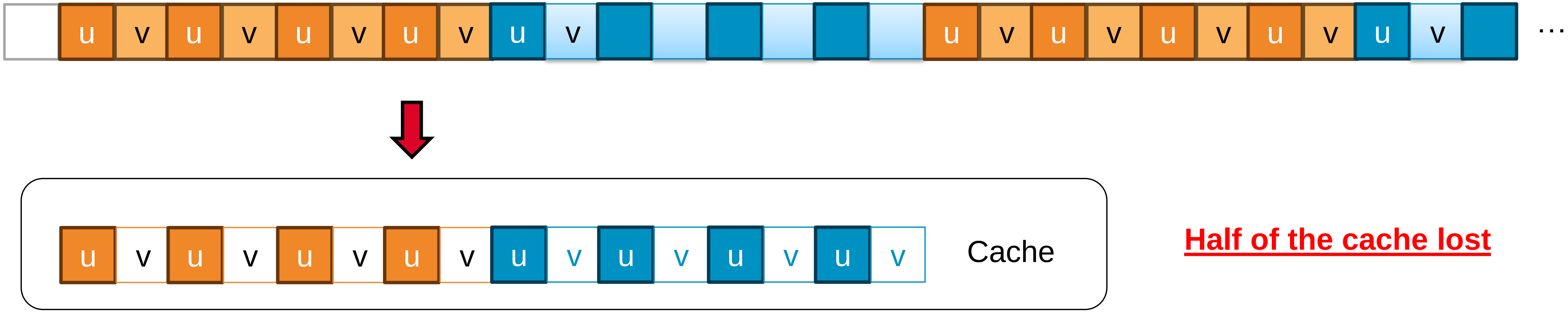


Cache lines

- Data are fetced by : **cache lines**
- Typically : **8 bytes** (CPU)
- Common mistake : **AOS** (**Array** Of **Structure**)
- Better to use **SOA** (**Structure** of **Array**)



Array of Struct in memory



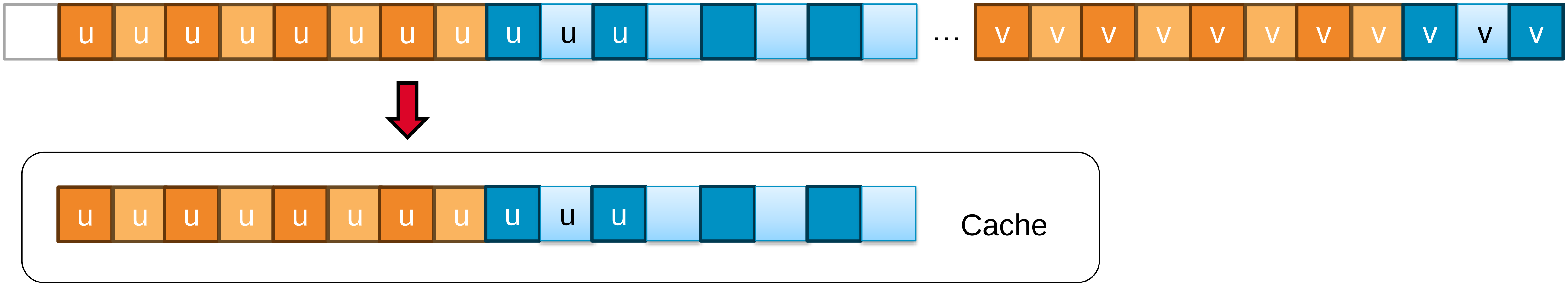
Half of the cache lost

```
# AOS (Array of Struct)
struct Cell
{
        float u;
        float v;
};
Cell mesh[HEIGHT][WIDTH];
```

```
# Loop using one of the members
# pragma omp parallel for
for ( size_t y = 0 ; y < HEIGHT ; y++)
        for ( size_t x = 0 ; x < WIDTH ; x++)
                mesh[y][x].u += 5.0;
```

Struct of Array in memory

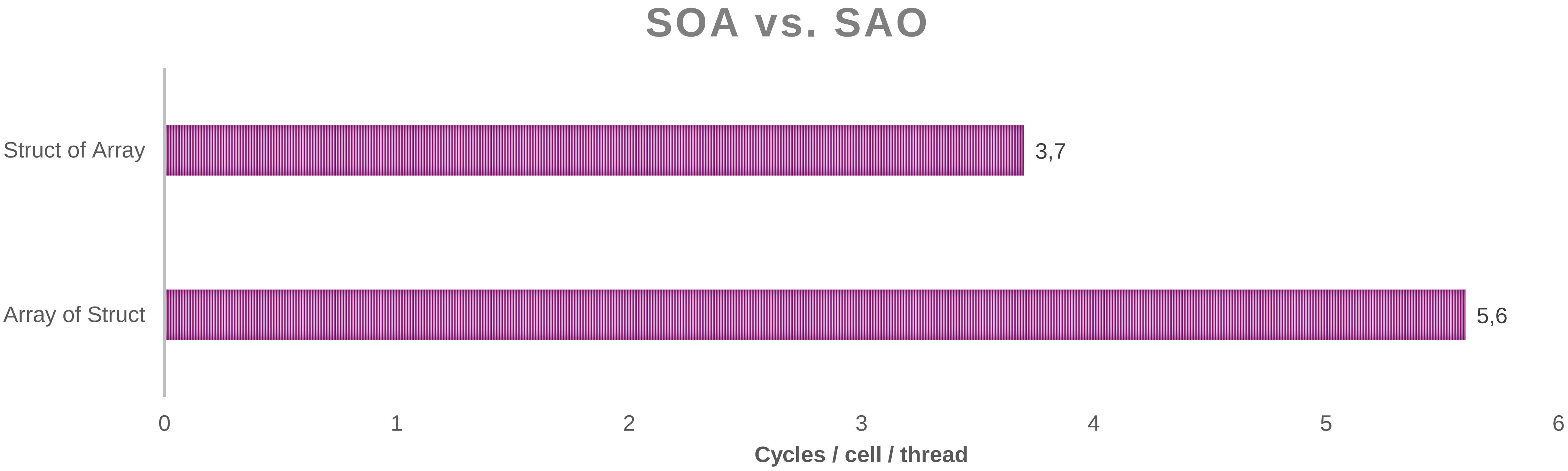
Struct of Array



```
# SOA (Struct of Array)
struct Mesh
{
        float u[HEIGHT][WIDTH];
        float v[HEIGHT][WIDTH];
};
Mesh mesh;
```

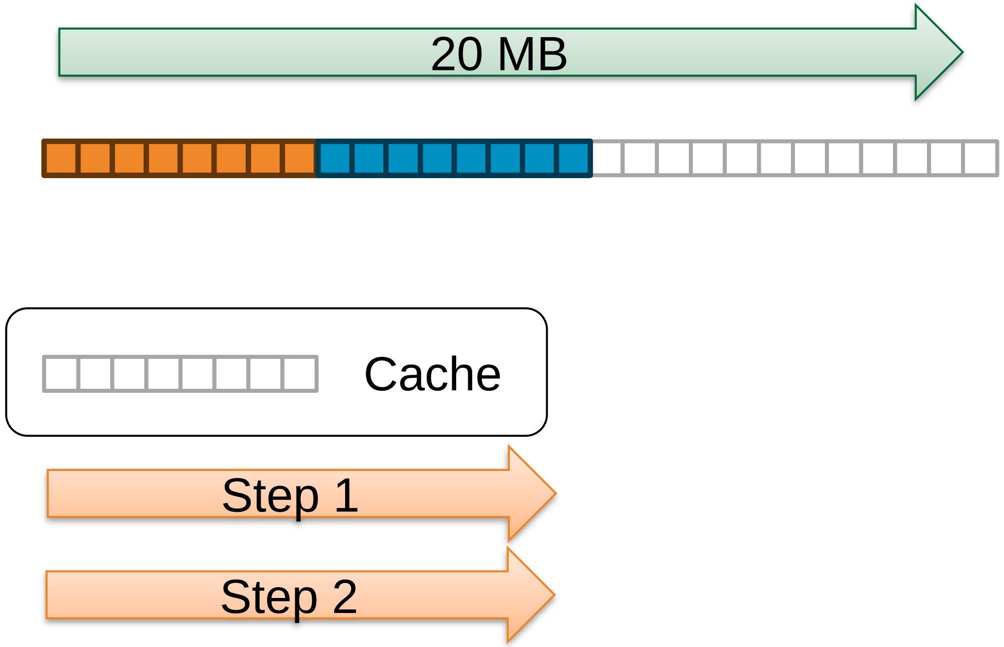
```
# Loop using one of the members
# pragma omp parallel for
for ( size_t y = 0 ; y < HEIGHT ; y++)
        for ( size_t x = 0 ; x < WIDTH ; x++)
                mesh[y][x].u += 5.0;
```

```
OMP_PROC_BIND=spread OMP_NUM_THREADS=6 ./soa
```

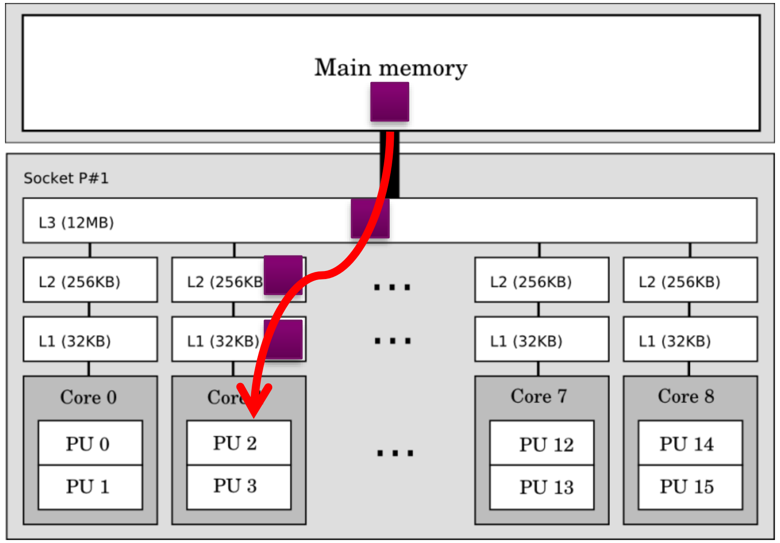


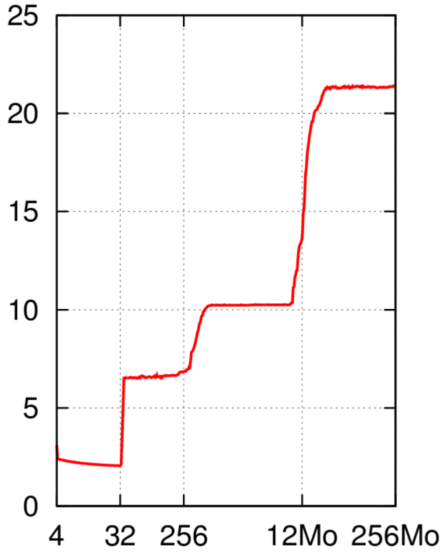
Cache blocking

- The classic **mistake** :
- **Walk over** data (time 1)
- **Walk again over** data (time 2)
- …



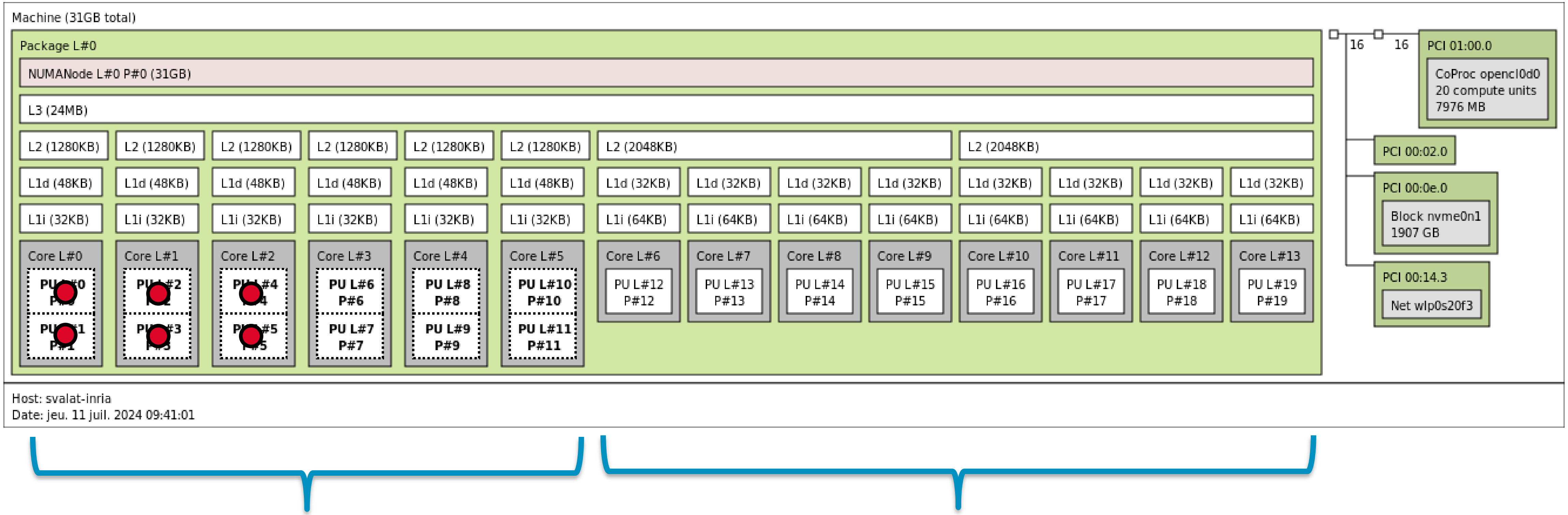
Processor : 8 cores





Question of binding

```
OMP_PROC_BIND=close OMP_NUM_THREADS=6 hwloc-bin core:0-5 ./soa
```

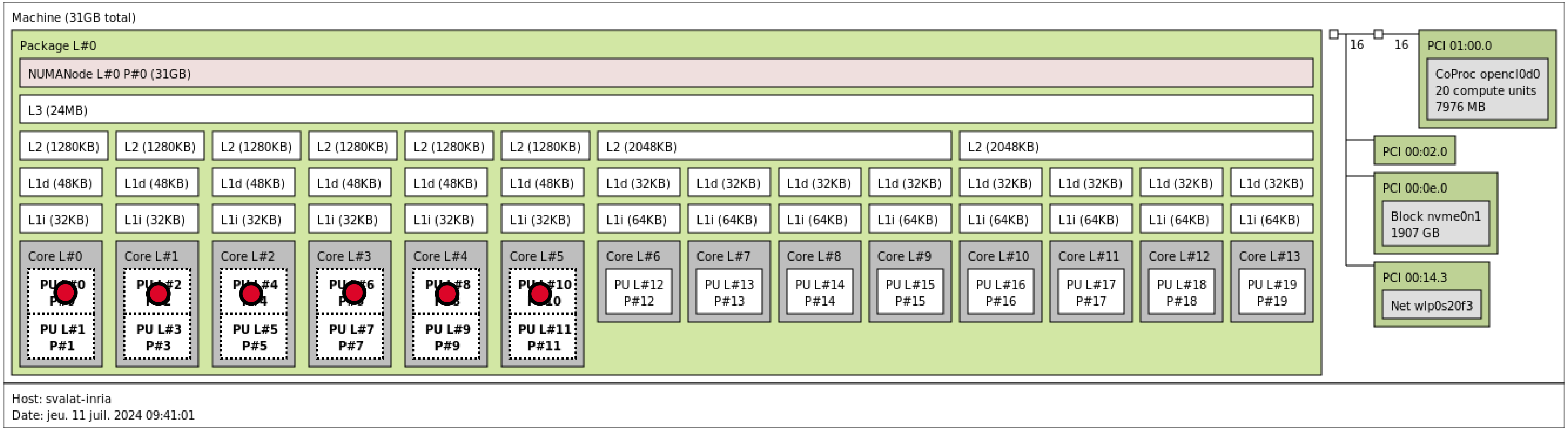


Fast cores

Energy efficient cores

Question of binding

```
OMP_PROC_BIND=spread OMP_NUM_THREADS=6 hwloc-bin core:0-5 ./soa
```



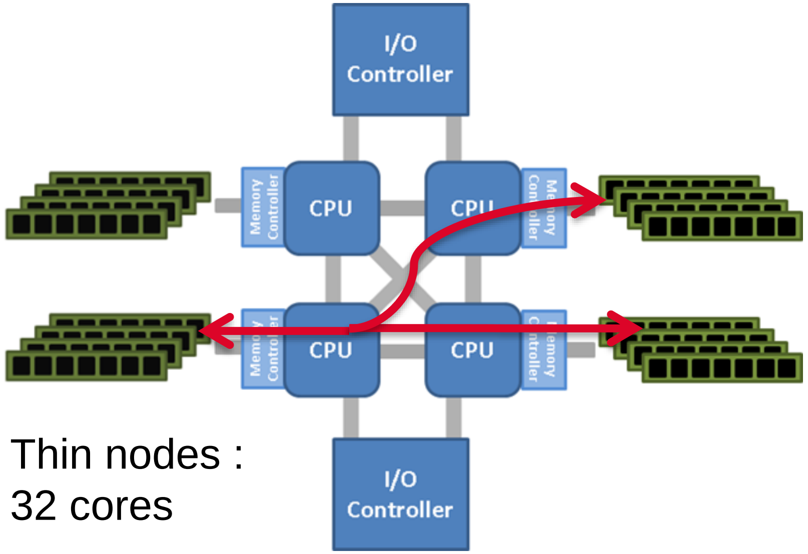
Fast cores

Energy efficient cores

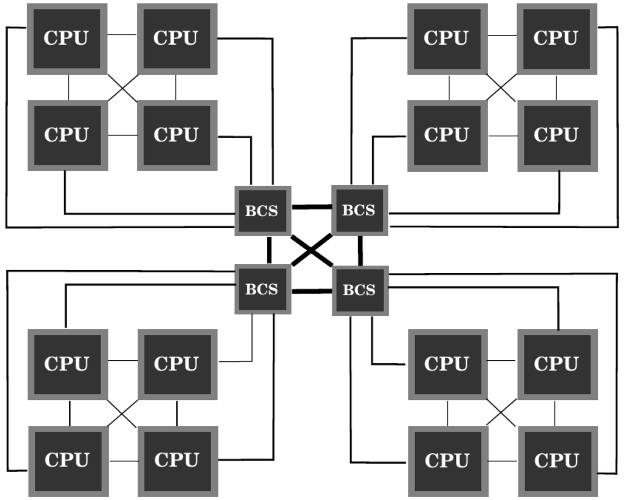
LE NUMA

Architecture

- Hierarchical **memory**
- **Remote** / **local** memories **(NUMA : Non Uniform Memory Access)**



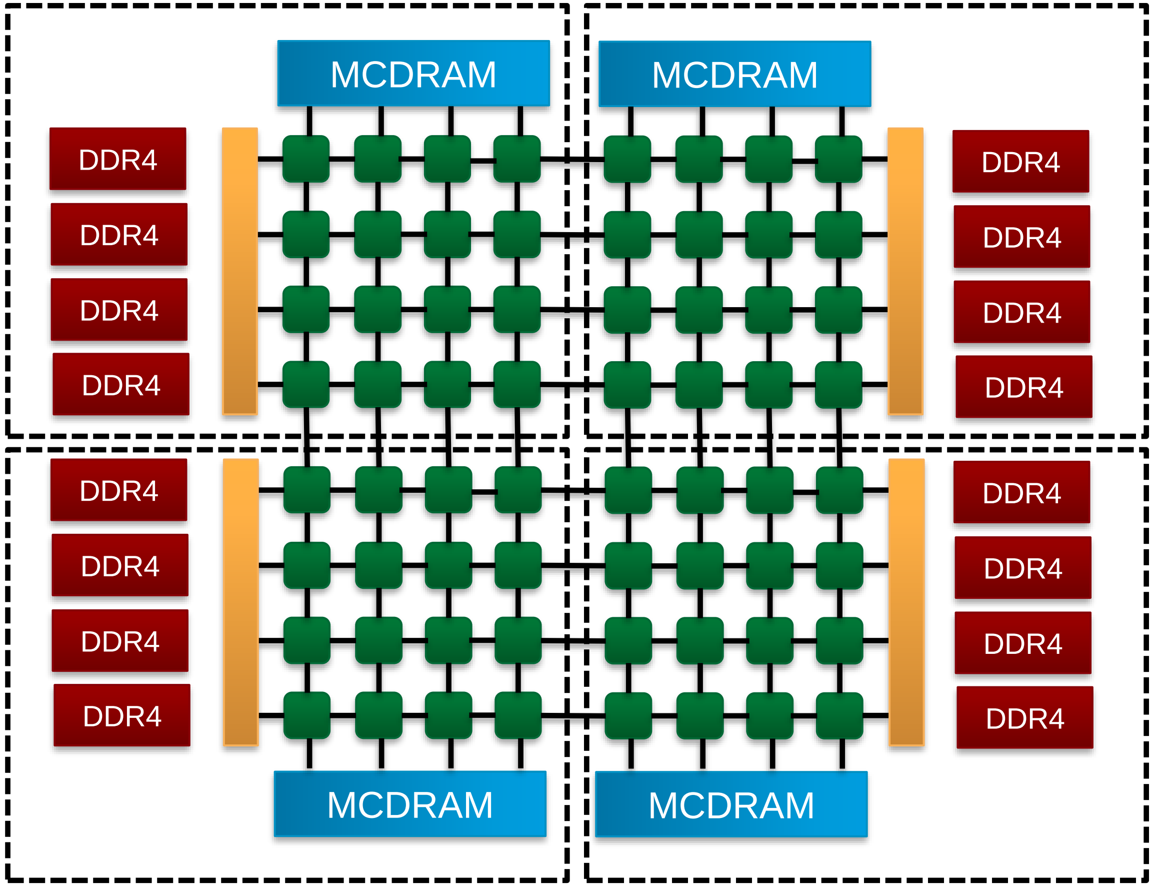
Thin nodes :
32 cores



Large nodes : 128 cores (BCS)

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

PAGINATION

Software memory management layer

- **Impact of memory management mechanisms ?**

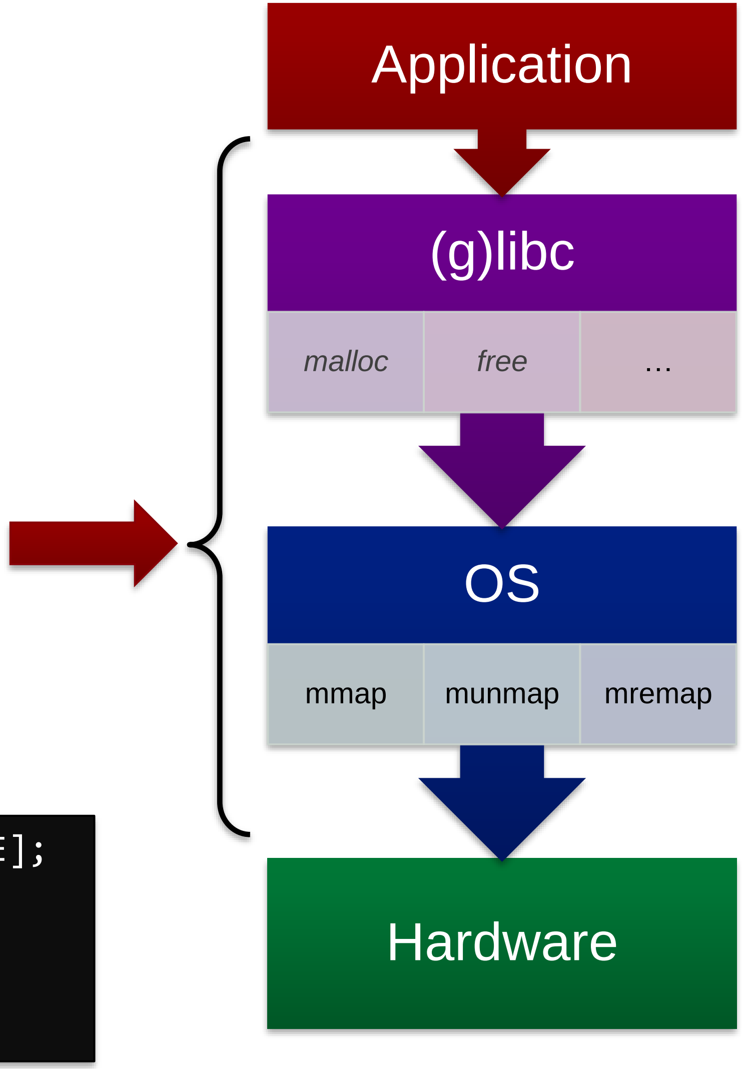
- Involving **two components** :
  - User space ***memory allocator*** (malloc)
  - **Operating System** (OS)

- Focus on :
  - Impact on **allocation time**
  - Impact on **access efficiency** (placement)

- Malloc C or C++ interface :

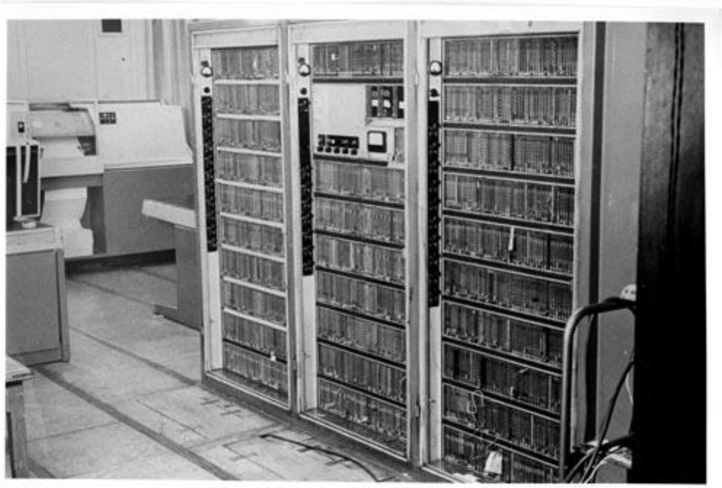
```
float * ptr = malloc(SIZE);
…
ptr = realloc(ptr,NEW_SIZE);
…
free(ptr);
```

```
float * ptr = new float[SIZE];
…
…
…
delete [] ptr;
```

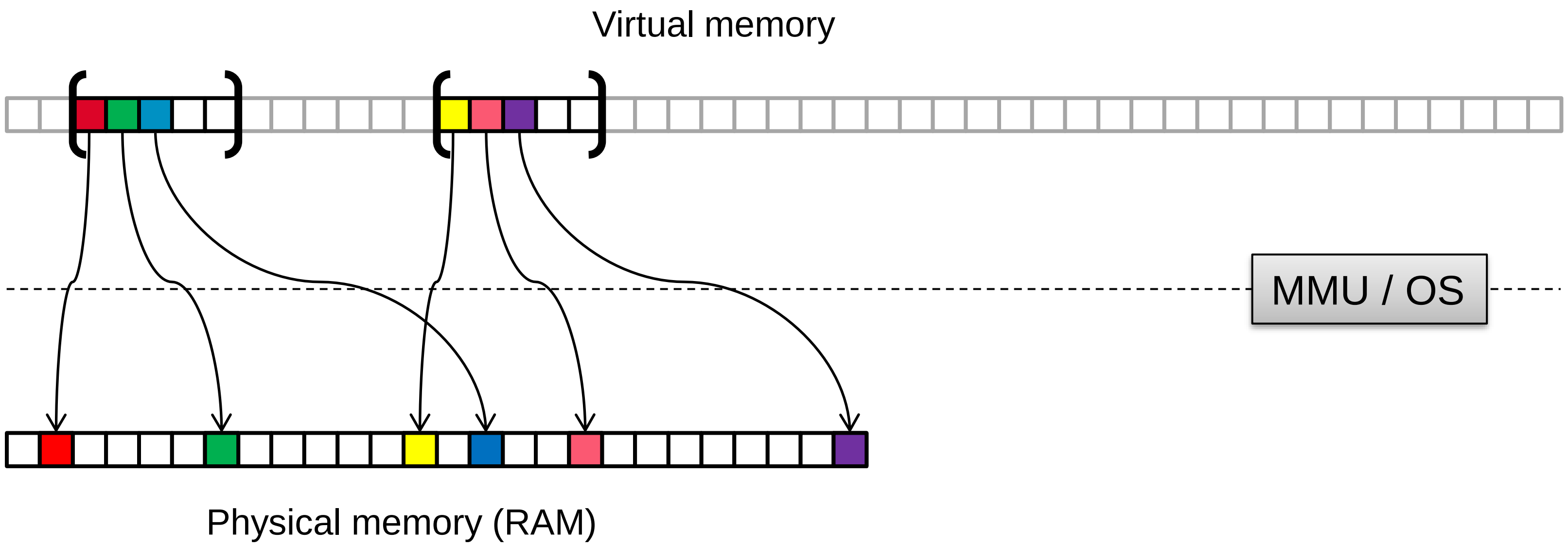


OS virtual / physical address spaces

- Two address spaces : **physical** + **virtual**
- Description of the **memory mapping** in blocks of **4 KB (pages)**
- **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**



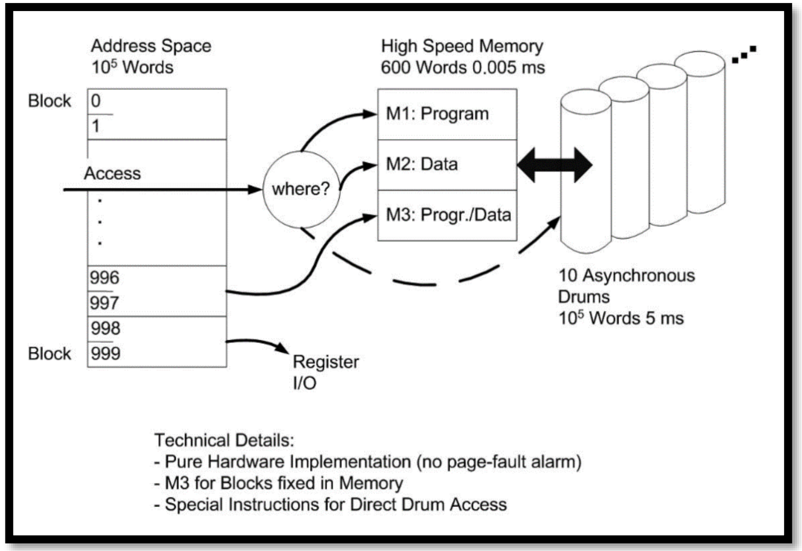
http://www.computerhistory.org/collections/catalog/102698470



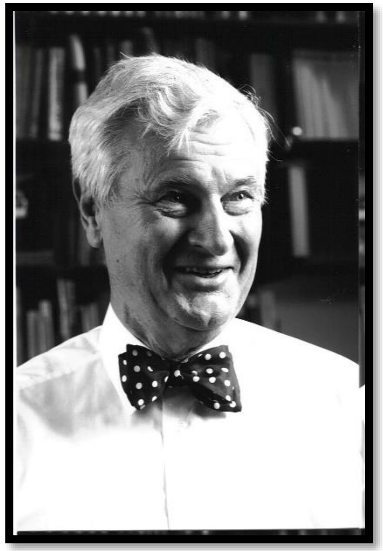
32 bits = 4 GB
48 bits = 256 TB
57 bits = 128 PB
~~64 bits = 16 EB~~

Origin of the concept

- May **1956** - Fritz Rudolf Güntsch’s
- **Logical Design of a Digital Computer with Multiple Asynchronous Rotating Drums and Automatic High Speed Memory Operation**
- **Origin of the virtual memory concept**
  [IEEE Annals of the History of Computing – Anecdotes](https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1369143)
  https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1369143

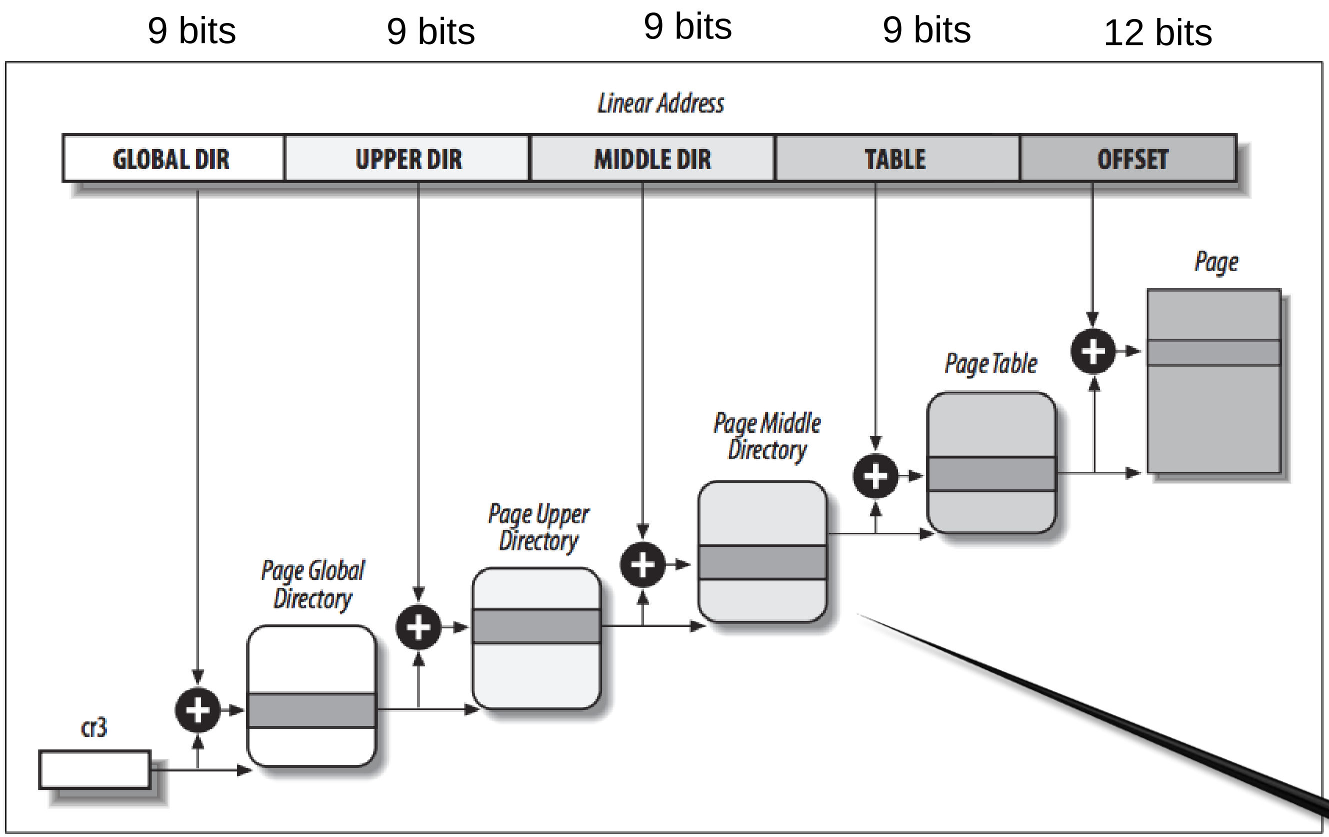


Technical Details:
- Pure Hardware Implementation (no page-fault alarm)
- M3 for Blocks fixed in Memory
- Special Instructions for Direct Drum Access



SCAN ME

Page table and TLB


9 bits 9 bits 9 bits 9 bits 12 bits

* From : Linux Kernel Driver

4 KB

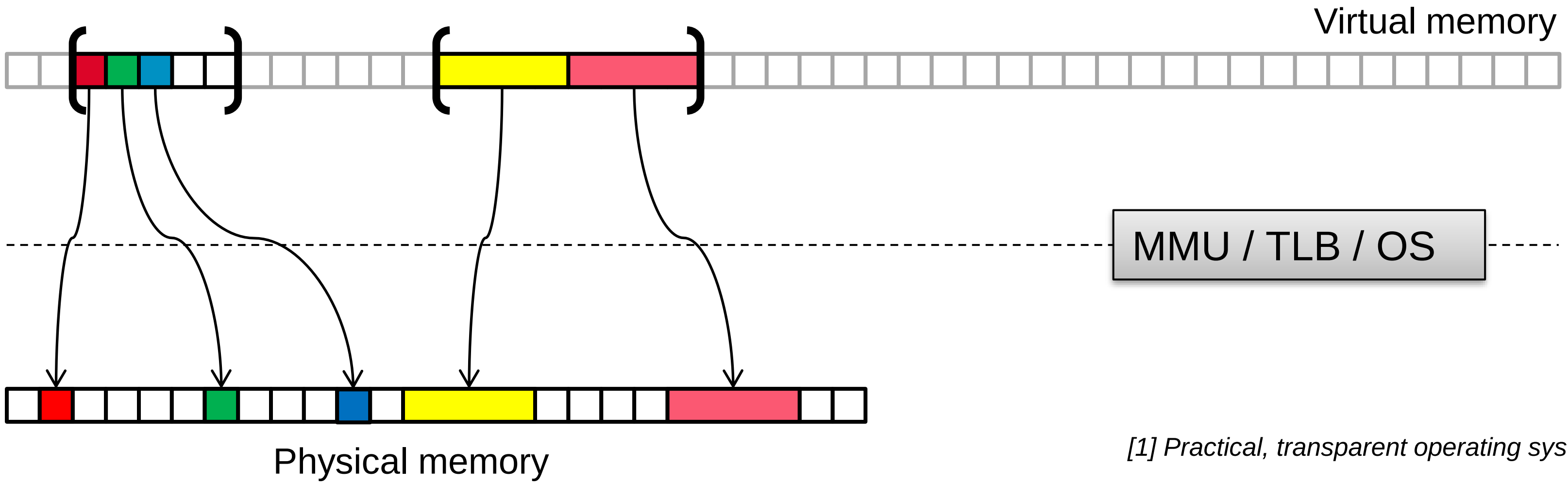
If we would had a flat page table ?

- **64 GB** / 4 KB = 16 M pages
- Page table : 16 B per page
- Per process : 16 B * 16 M = **256 MB** !

- We **don't** want to **walk** over the page table for **every access**
- CPU has **a cache** (**TLB** : Translation Lookaside Buffer)

Huge pages

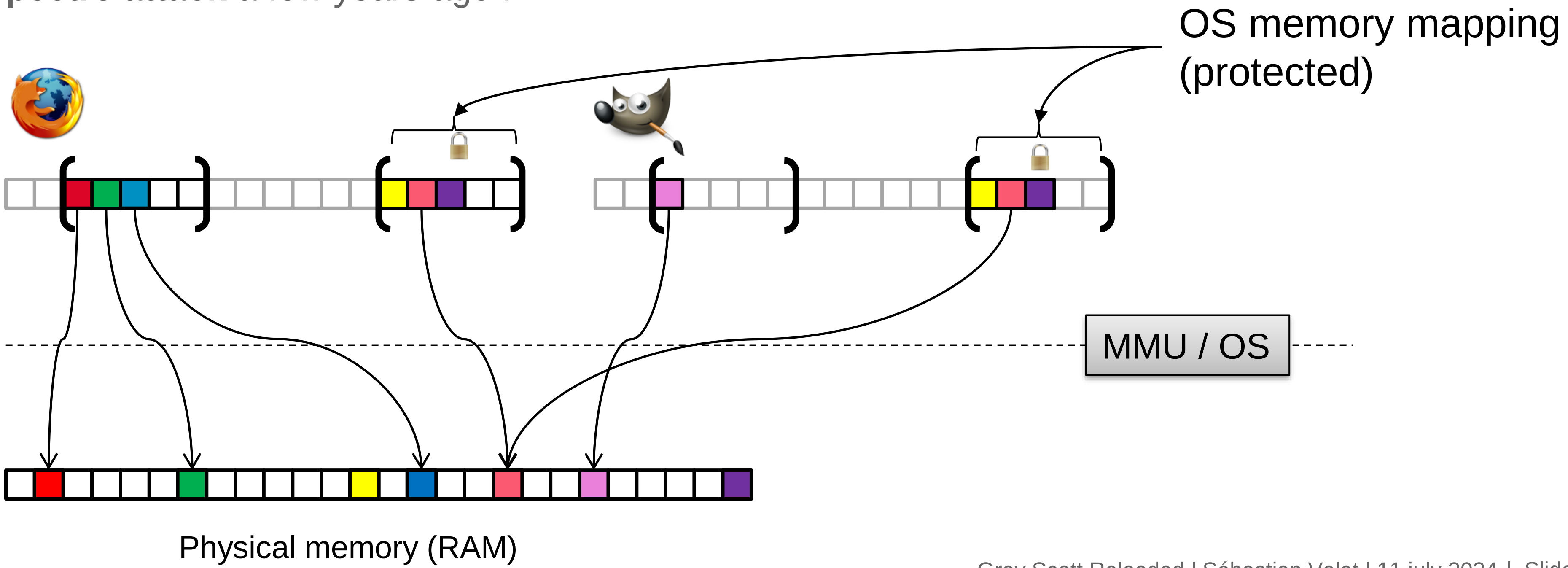
- With **4K pages**, intel CPU TLB has **1024 entries** => address **4 MB**
- **x86_64** processors also support **2 MB** or **1 GB** pages **(Huge pages)**
- With **2M pages**, TLB address **2 GB**
- **First real support : FreeBSD (superpages, 2002)** [1]
- Support **Linux : ***old HugeTLBfs* then now **Transparent Huge Pages (THP), 2011**



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

Play with memory mapping

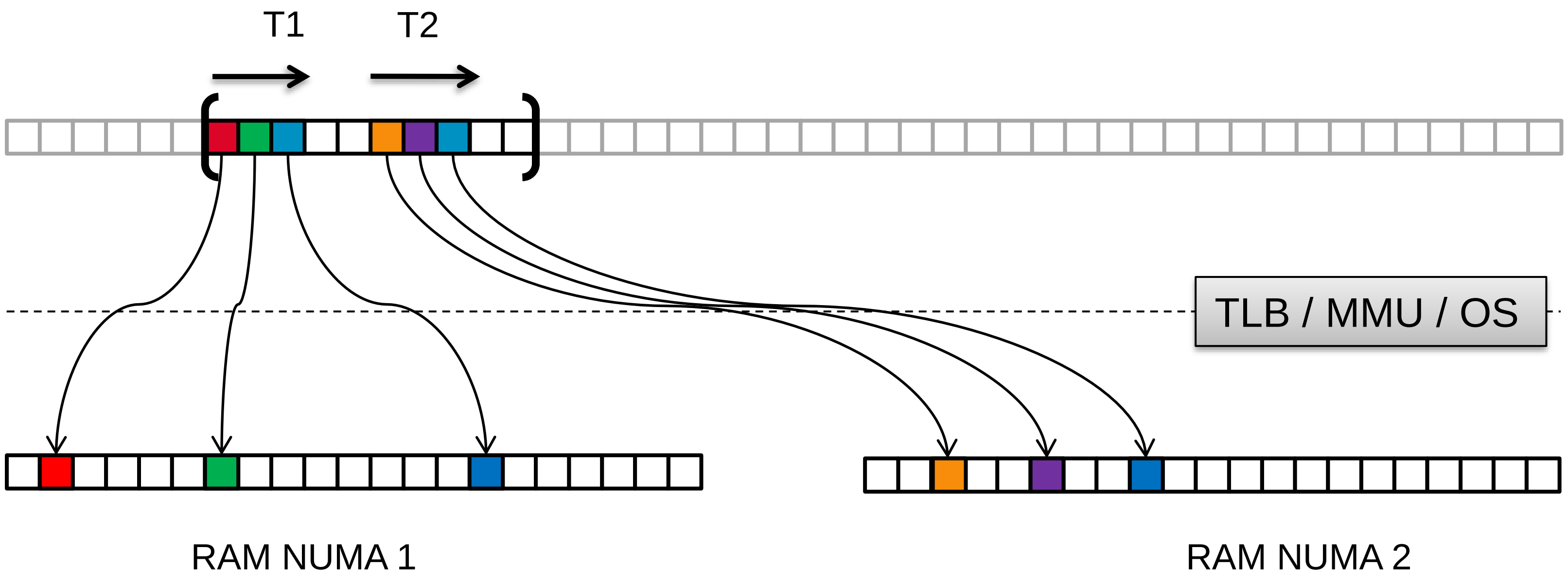
- Virtual memory **isolate** each process
- We can do **shared memory,** mapping **the same memory twice**
- **Most OS** also use a **trick** by **mapping** the **OS memory in each process**
  - At the **end of the address space**
  - **Protected**
- Issue with **Spectre attack** a few years ago !



Lazy page allocation

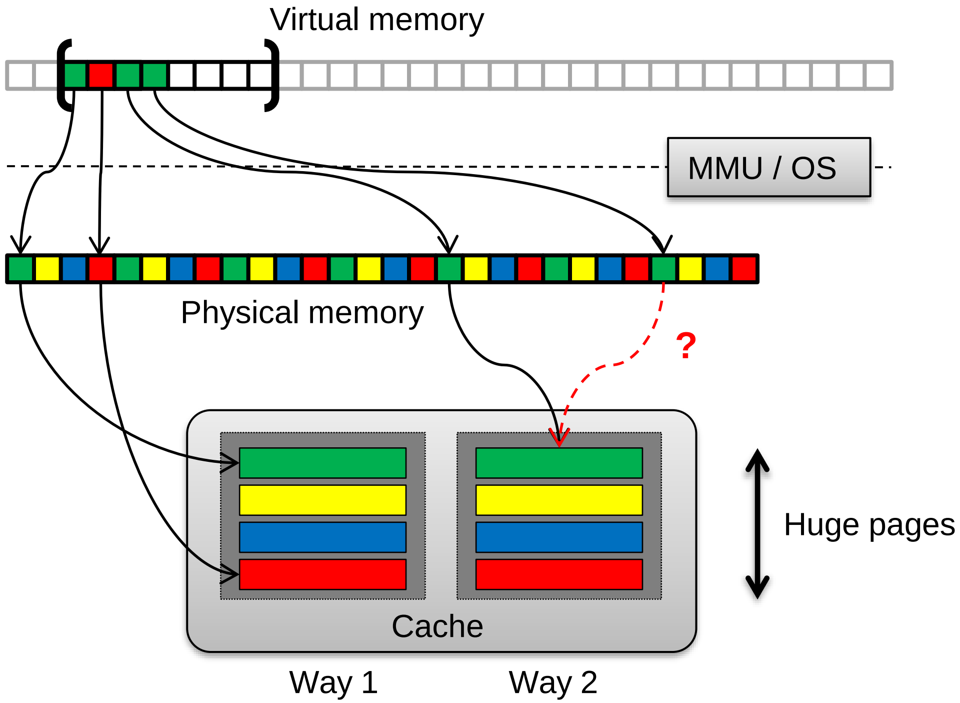
- **mmap** creates **pure virtual** area
- First touch creates a **page fault** for each virtual page
- OS provides **physical pages** on **first touch**
- **First touch implicitly** determines **NUMA placement** of the page

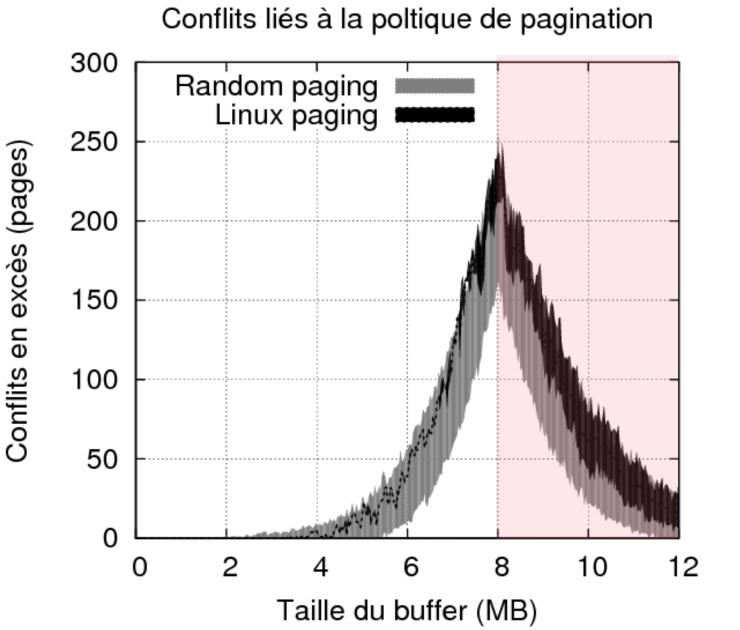
```
ptr = mmap(…,SIZE,…);
#pragma omp parallel for
for (i = 0 ; i < SIZE ; i++)
            ptr[i] = 0;
```

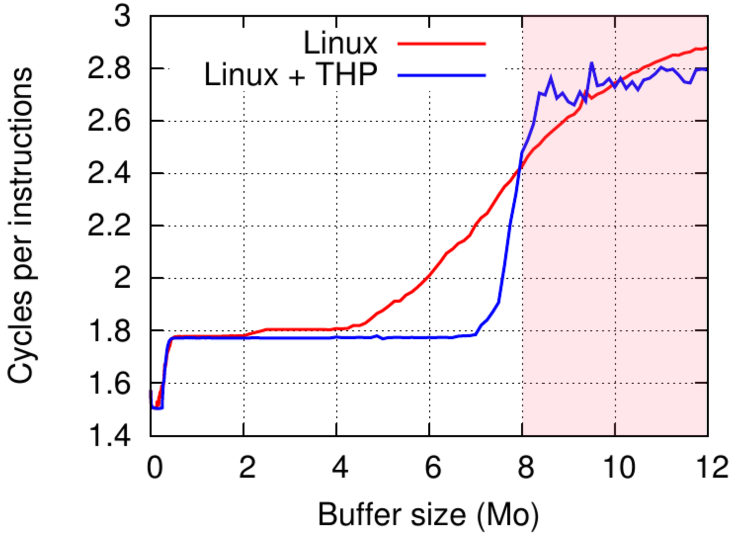


Cache associativity

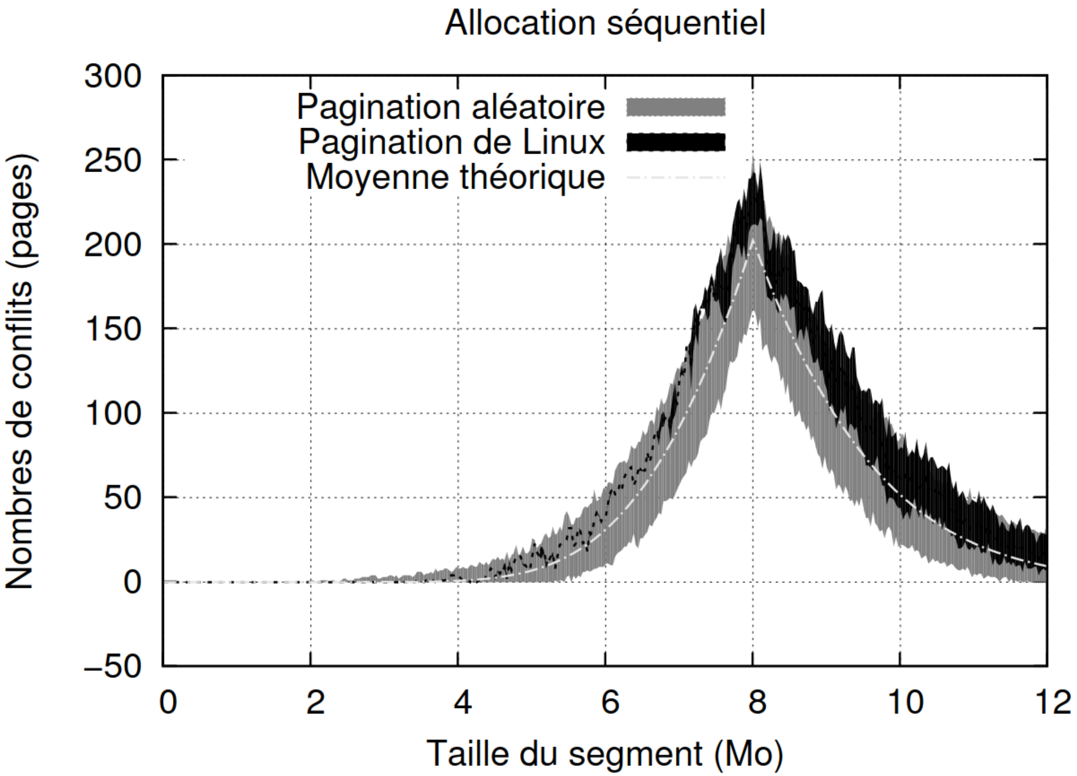
- Data can only be placed in one of the **N lines associated to the address**
- Can create **conflicts** depending on the OS
- Linux “**randomly” chooses** the pages

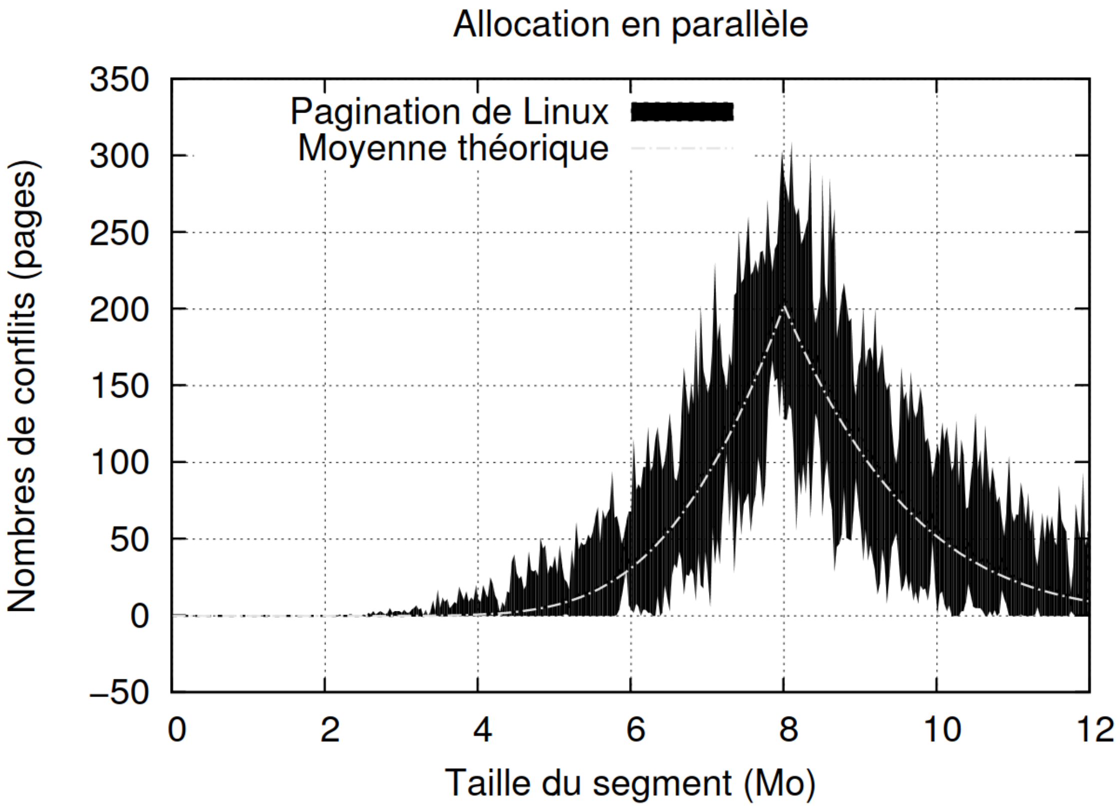






Same equation than electron in boltzman statistic !



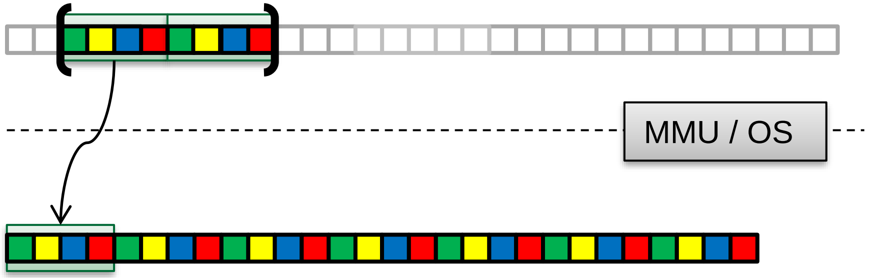


$$f_{poli.}(N) = C_{max}(Np - \sum_{k=0}^{A} kP(N,k) - A(1 - \sum_{k=0}^{A} P(N,k))) - f_{capa.}(N)$$

Existing solutions

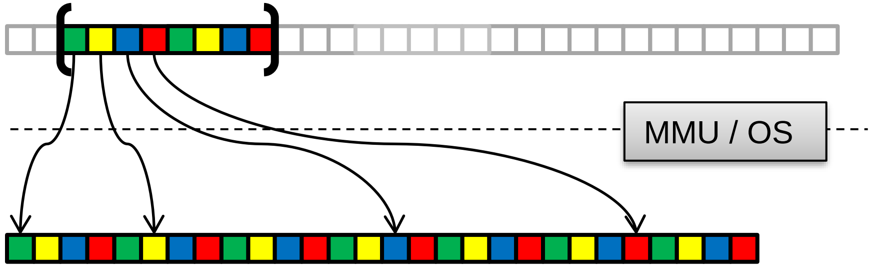
Huge pages

- **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**



I. Introduction
II. Analysis of OS pagin policy
III. NUMA allocator performances for HPC applications
IV. Page zeroing in Linux first touch handler
V. Conclusion

ANALYSIS OF OS PAGING POLICY

OS strategies comparison

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

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

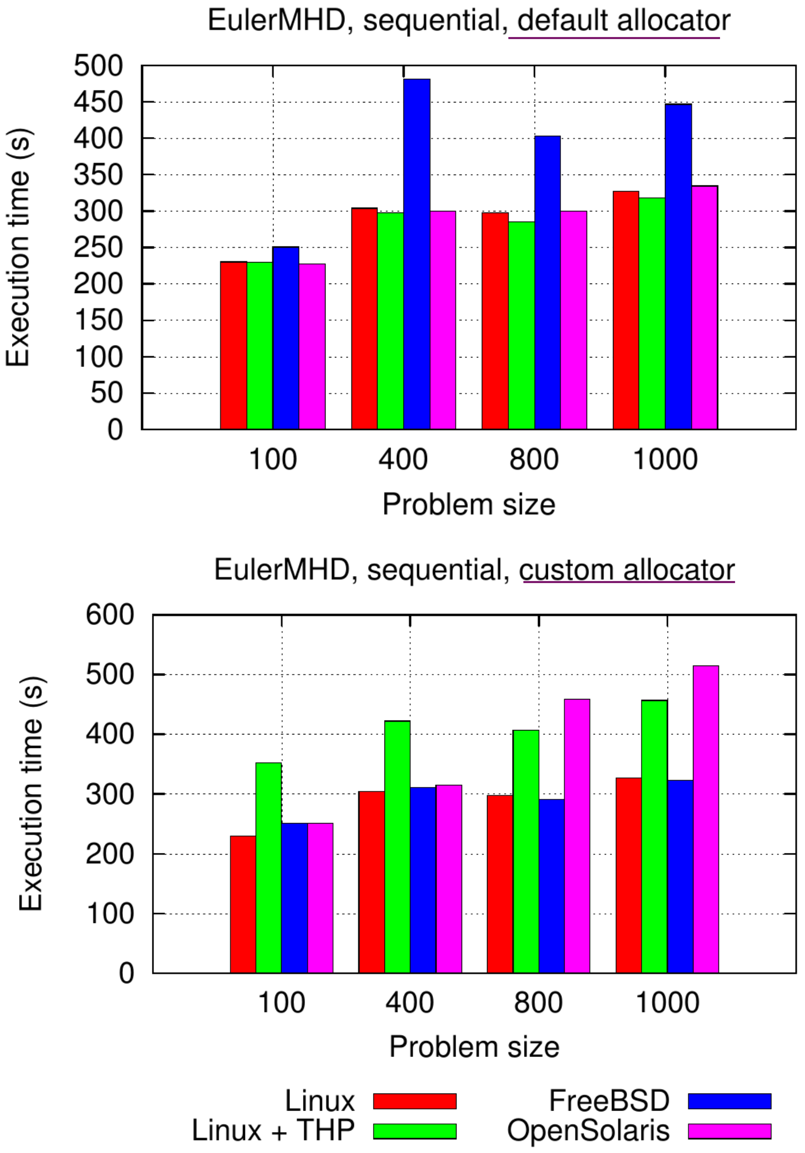
- Is **Linux** slower due to **random paging** ?
- Tested architecture : Intel **Nehalem bi-socket**
- Use a fixed compile chain : **GCC/Binutils/MPI/BLAS**
- **Focus a pathological case**

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.**



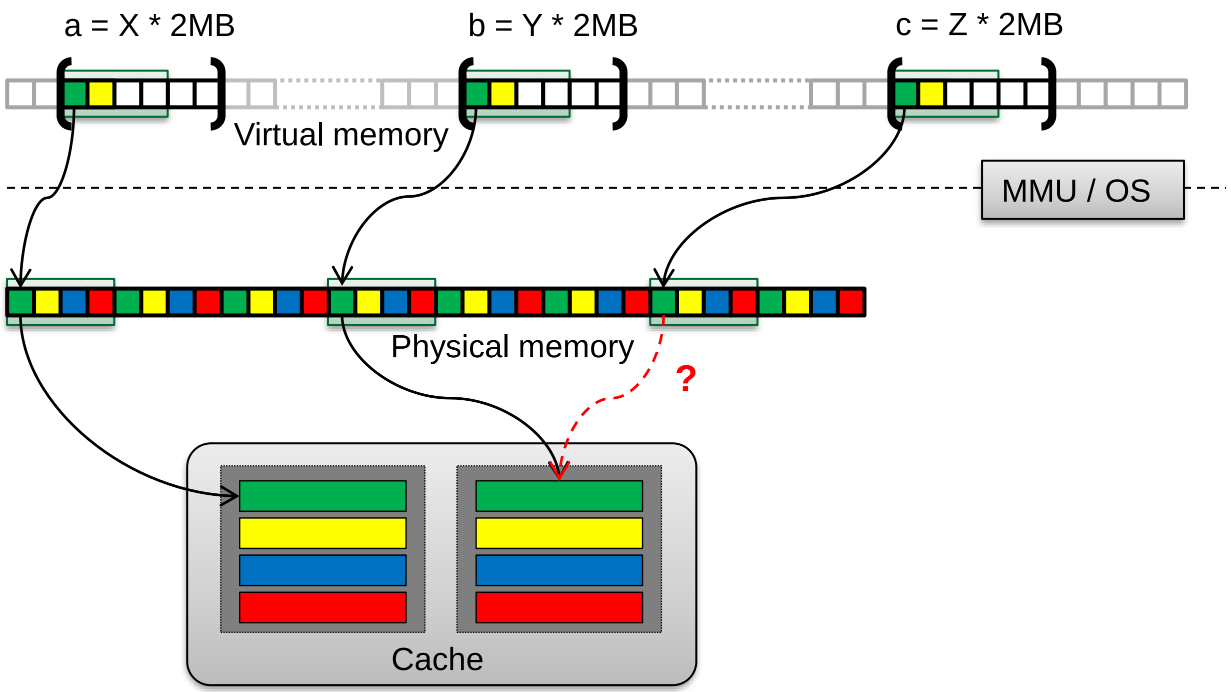
EulerMHD, sequential, default allocator

EulerMHD, sequential, custom allocator

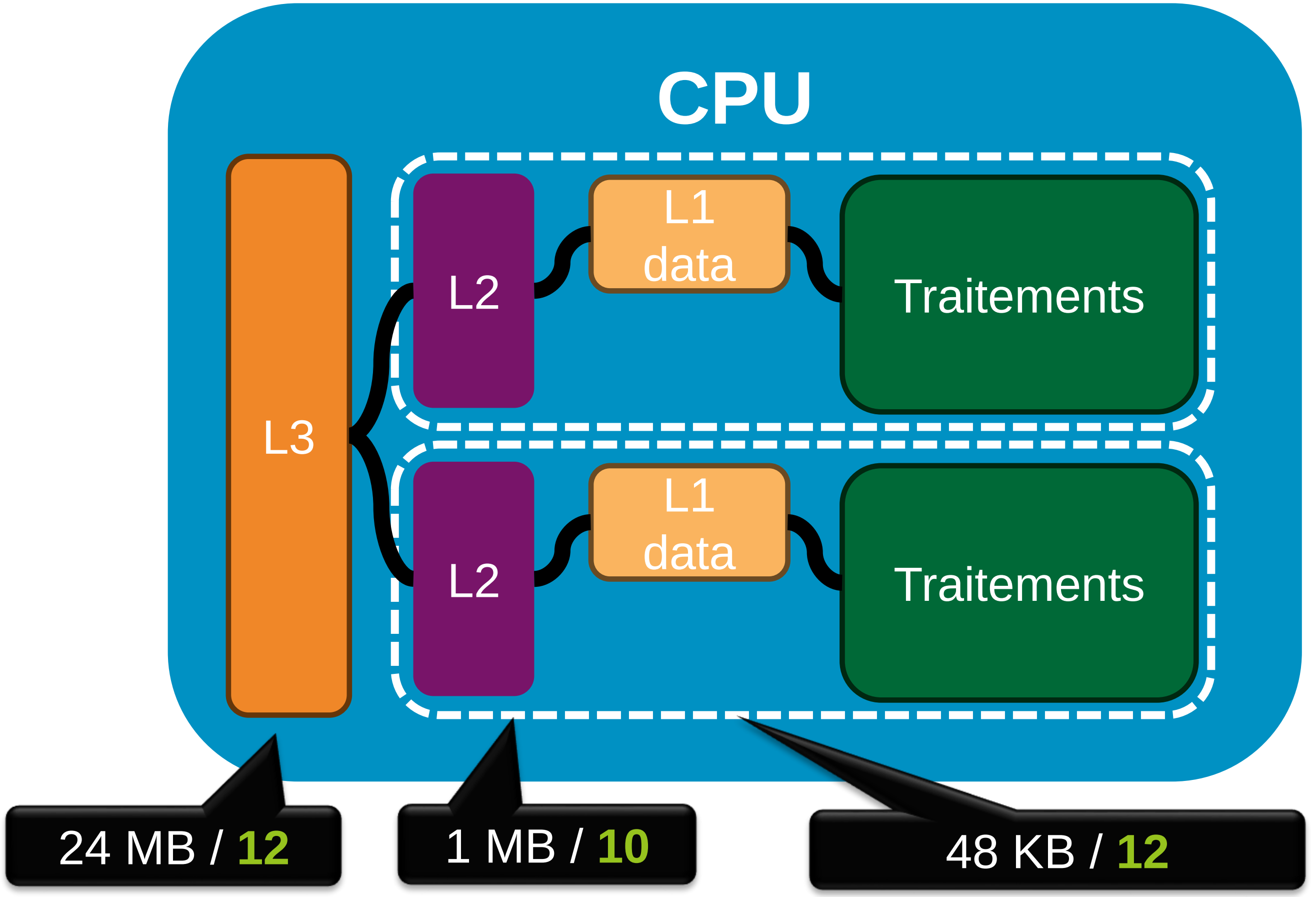
Alignment effect on regular coloring

- 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];
```

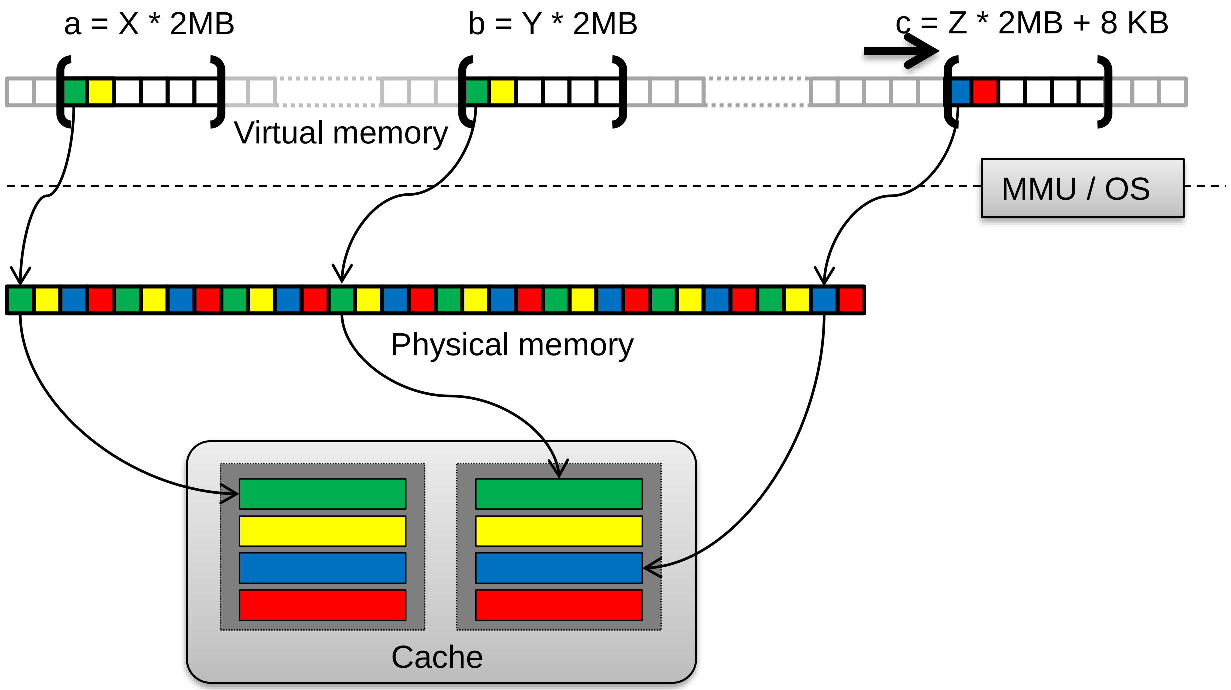


Typical sizes now



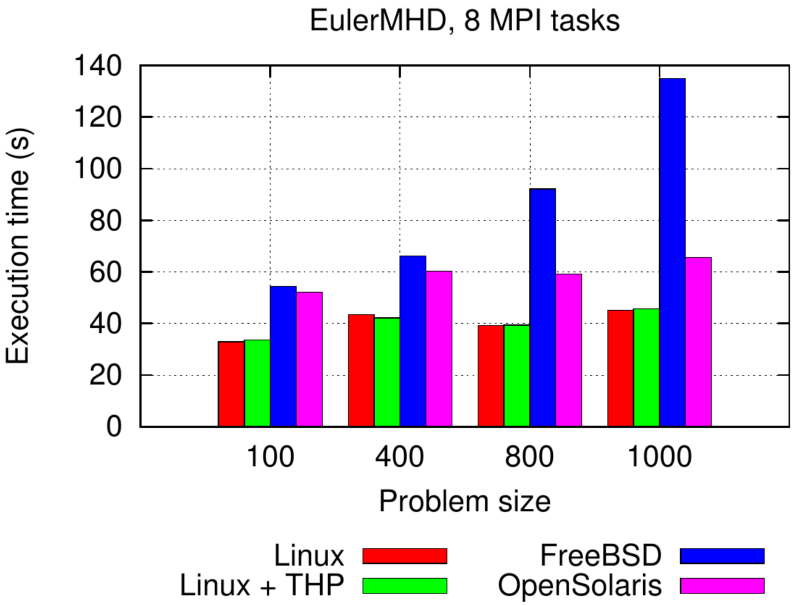
Solution

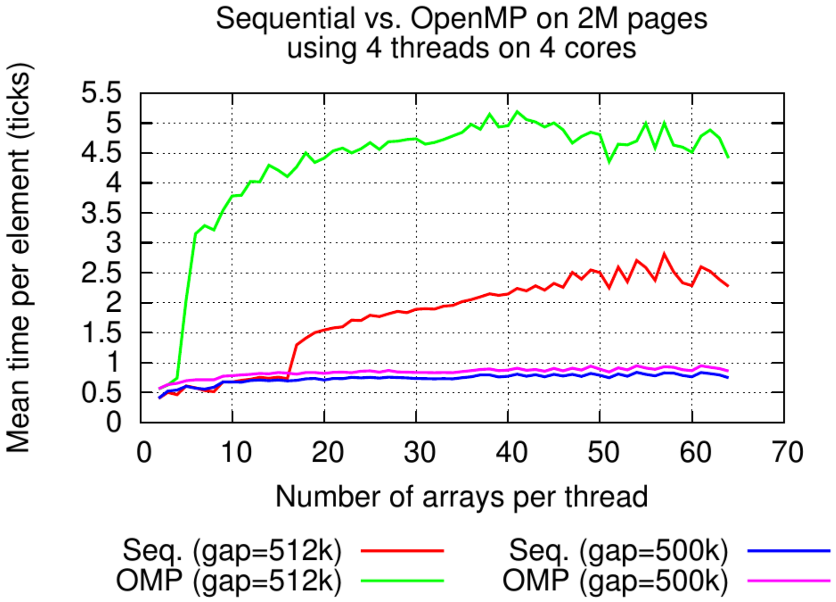
- Avoid segment **alignments** on **cache way size** (mmap / malloc)**.**
- The **Linux random** approach **prevents pathological cases**
- Do not use **regular patterns** for **page coloring** (eg. **single modulo**)
- **Huge pages** are **regular** by **hardware definition**



Impact on threads

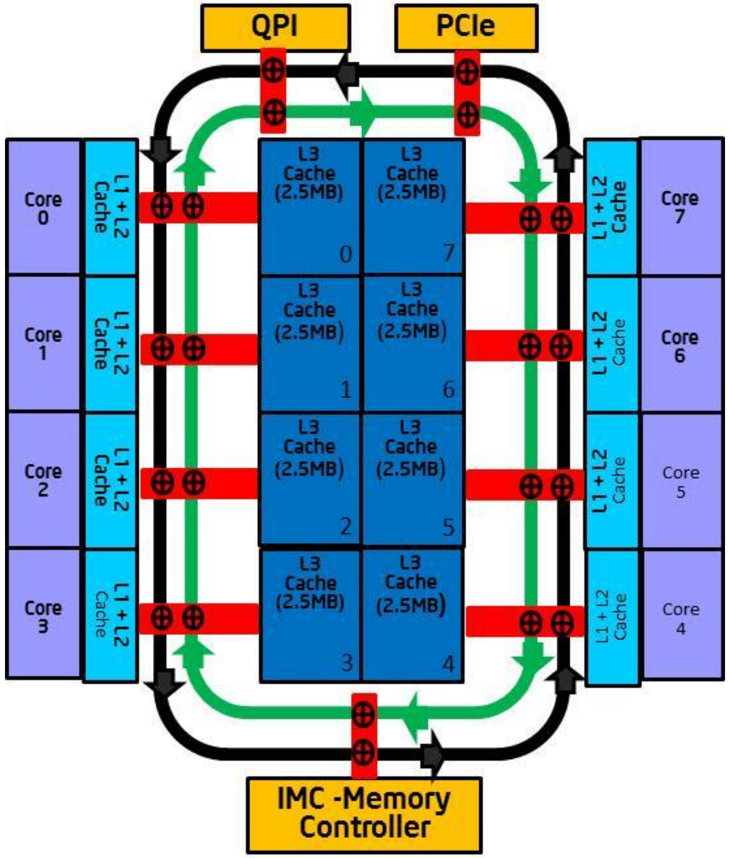
- **Larger effects** on **shared caches** with threads/processes (Nehalem)
- EulerMHD : **Slowdown** up to **3x** on **FreeBSD**
- **16** ways L3 cache implies a maximum of **4 aligned arrays** per core
- **No limit** on concurrent arrays for **unaligned allocations**





New intel L3 cache slices

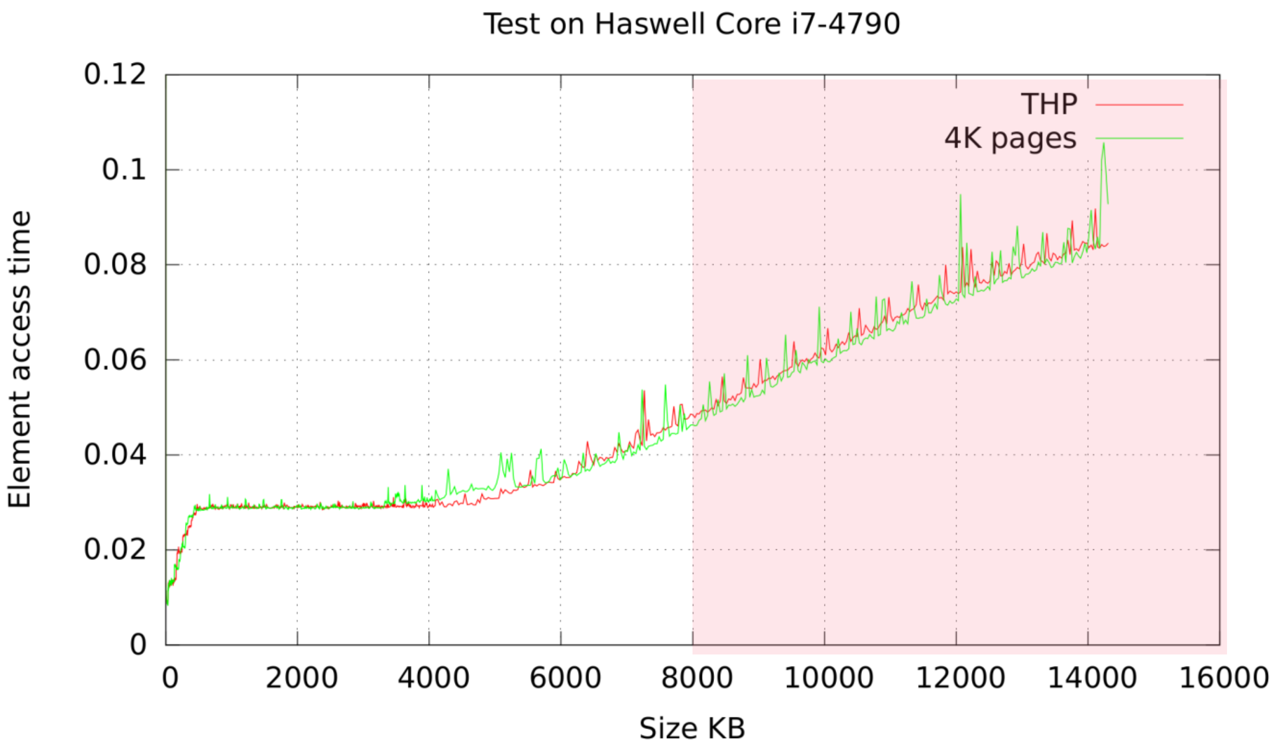
- Since **Sandy Bridge**
- L3 splits in **slices**
- **Slice** is selected by **hashing the address**
- **Each slice** has associativity with **16 ways**
- This **fix** the **coloring/alignment issue**



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**
- **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 !



4K aliasing (old issue but fun !)

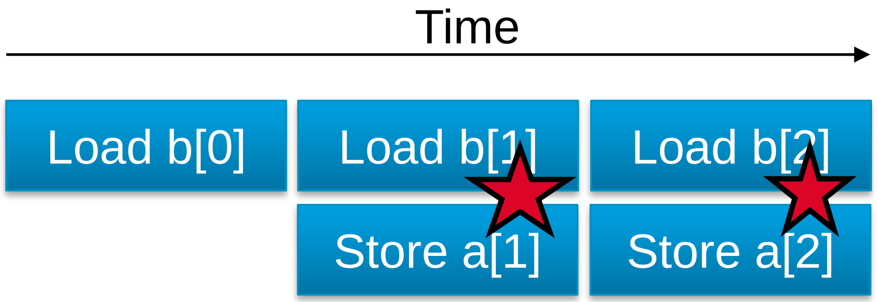
- Consider the simple loop :

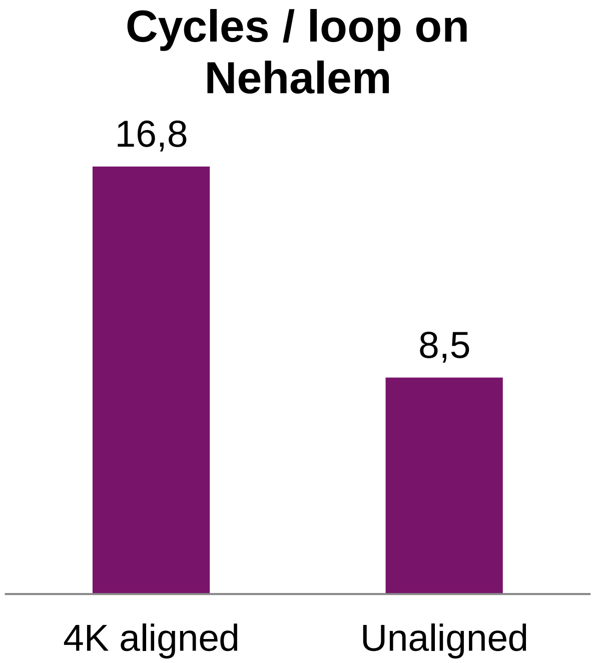
```
for (i = 1 ; i < SIZE ; i++)
            a[i] = b[i-1]
```

- If addresses verify :

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

- **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** !
- Mainly **fixed since sandy bridge**





I. Introduction

II. Analysis of OS paging poliocy

III. NUMA allocator for HPC applications

IV. Page zeroing in Linux first touch handler

V. Conclusion

NUMA ALLOCATOR FOR HPC APPLICATIONS

Allocator performance on HPC applications

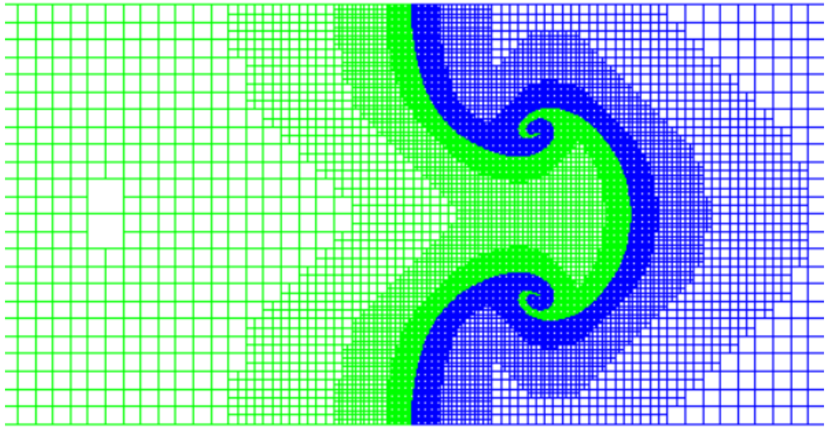
- Main interest : **malloc time cost**

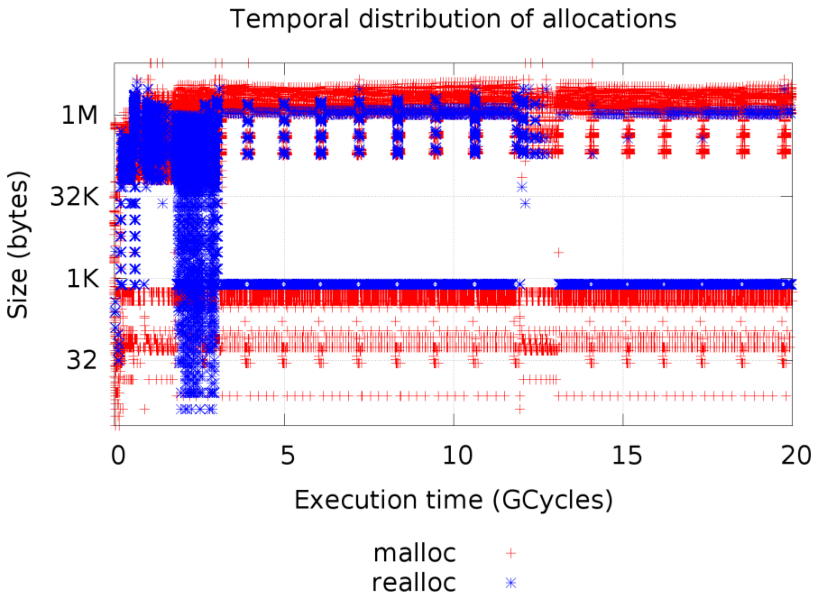
- Test case : **Hera (CEA)**
  - **Adaptive Mesh Refinement** (AMR)
  - **Massive C++/MPI code** (~1 million lines).

- **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*

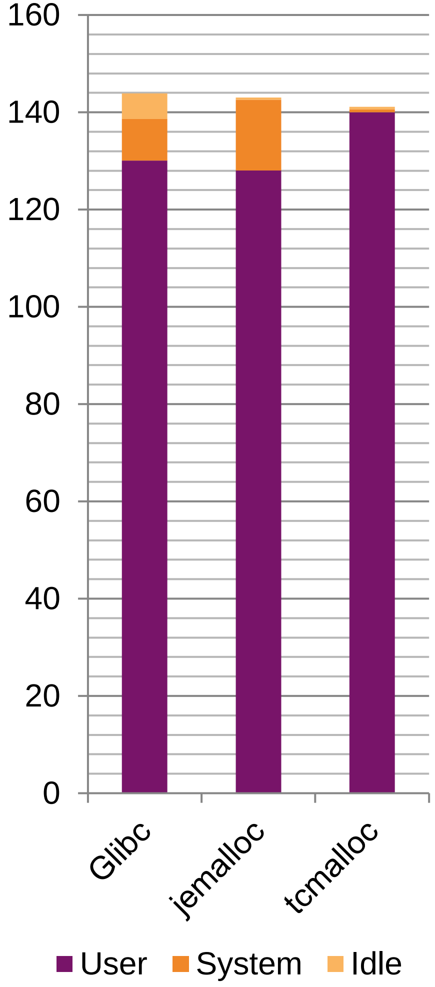




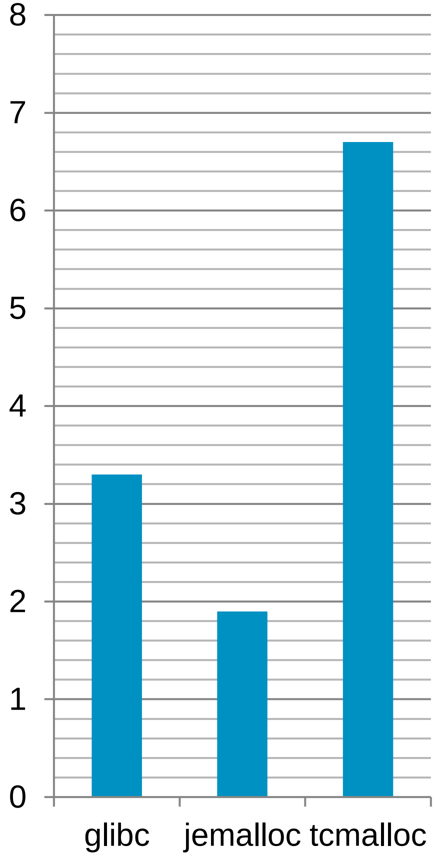
Hera preliminary results

12 cores

Execution time(s)

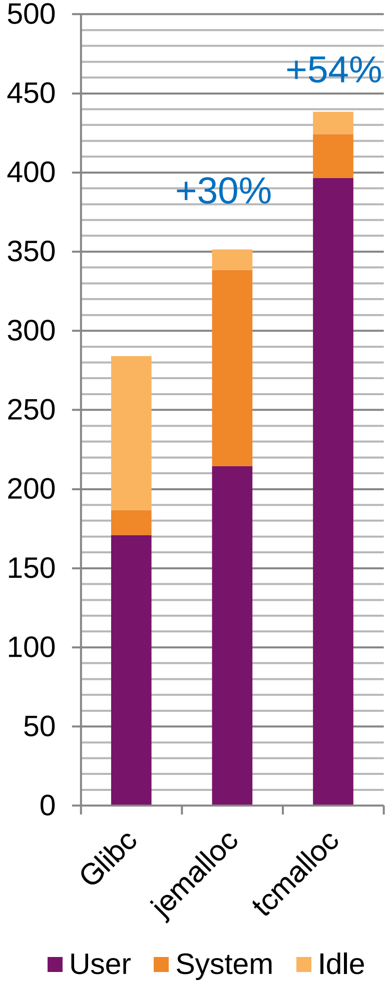


Physical mem.(Go)

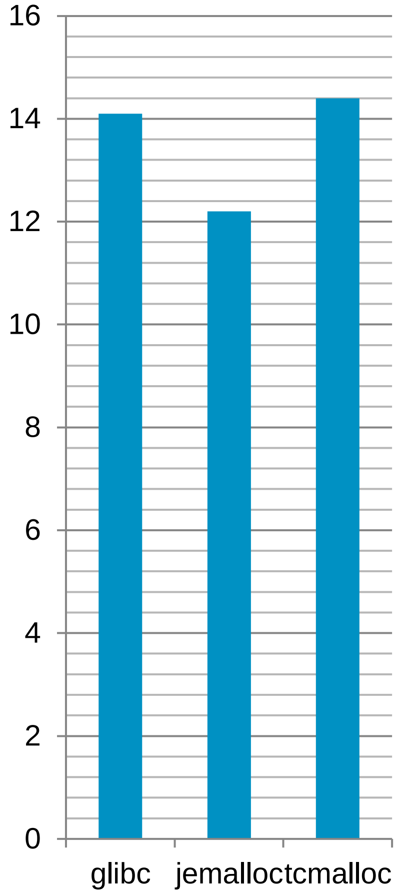


128 cores

Execution time(s)



Physical mem.(Go)



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

- 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**

Large allocations

My goals :

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

Allocator Profiles

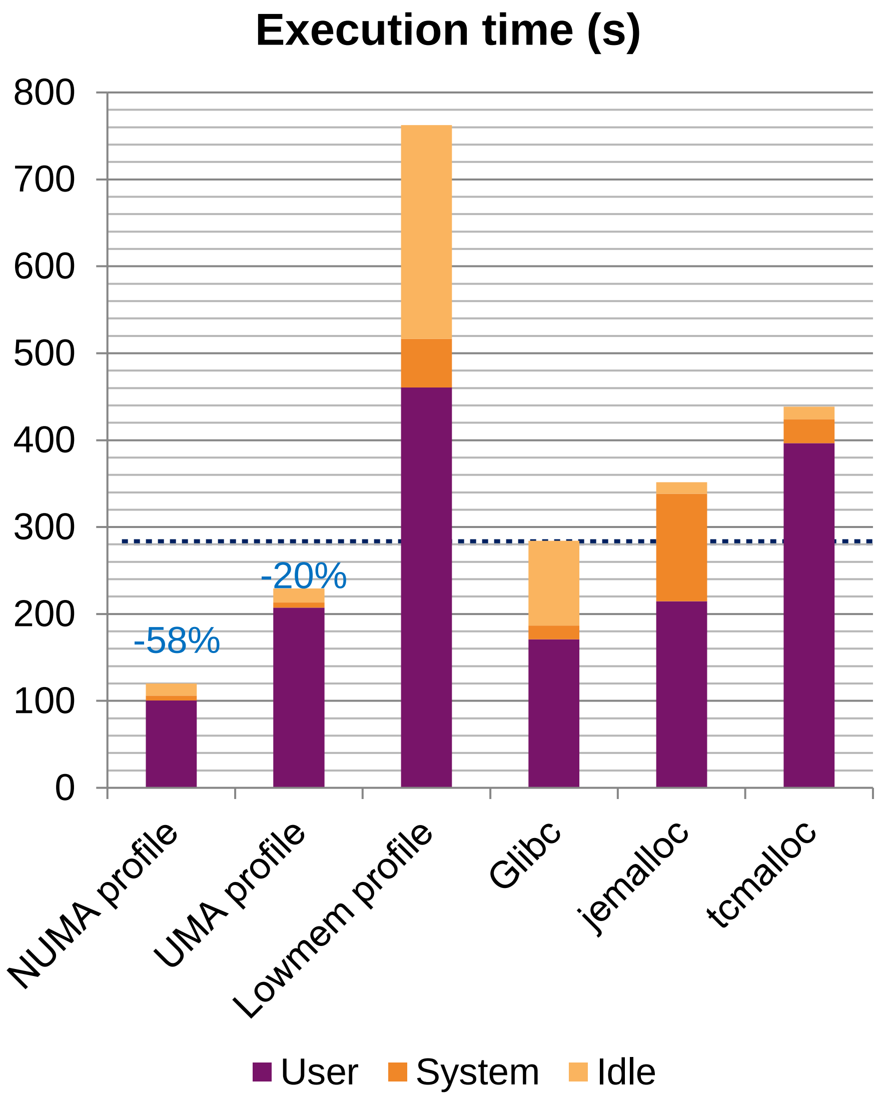
- Test allocator with **multiple profiles**

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

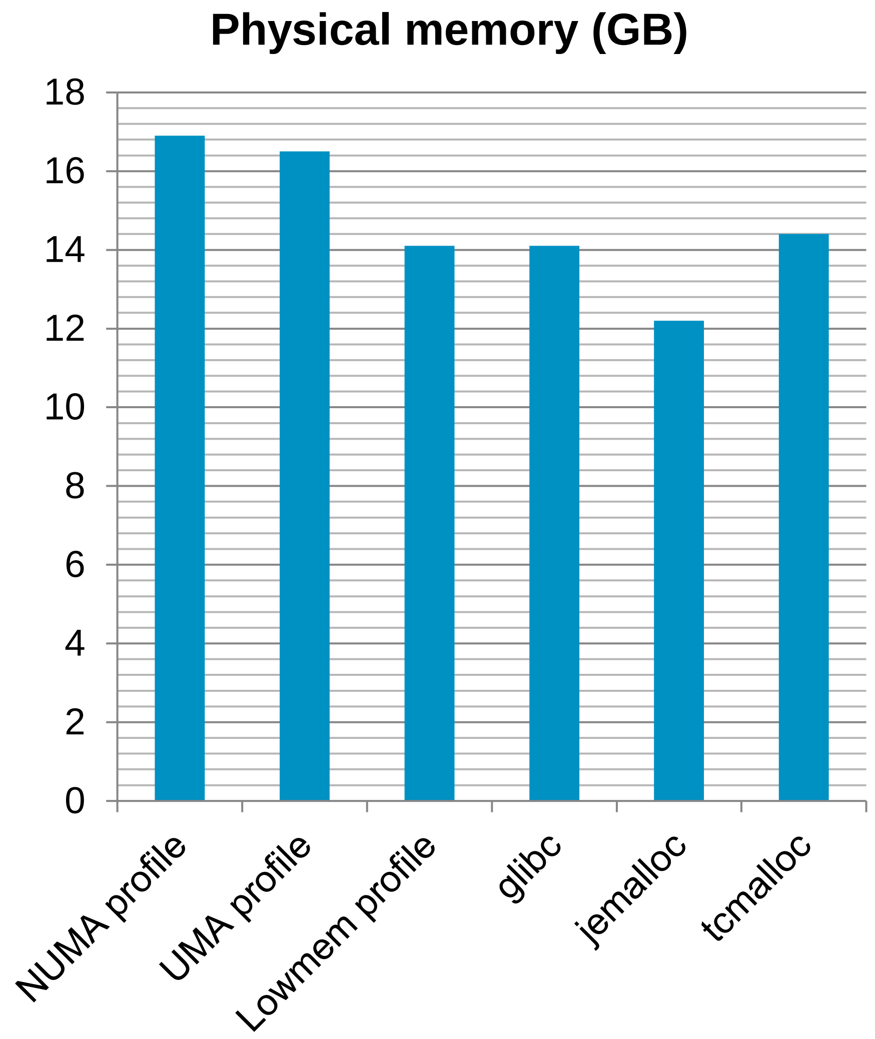
- **UMA** Profile
  - Recycle large segments
  - Disable NUMA
  - Use only one common memory source

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

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

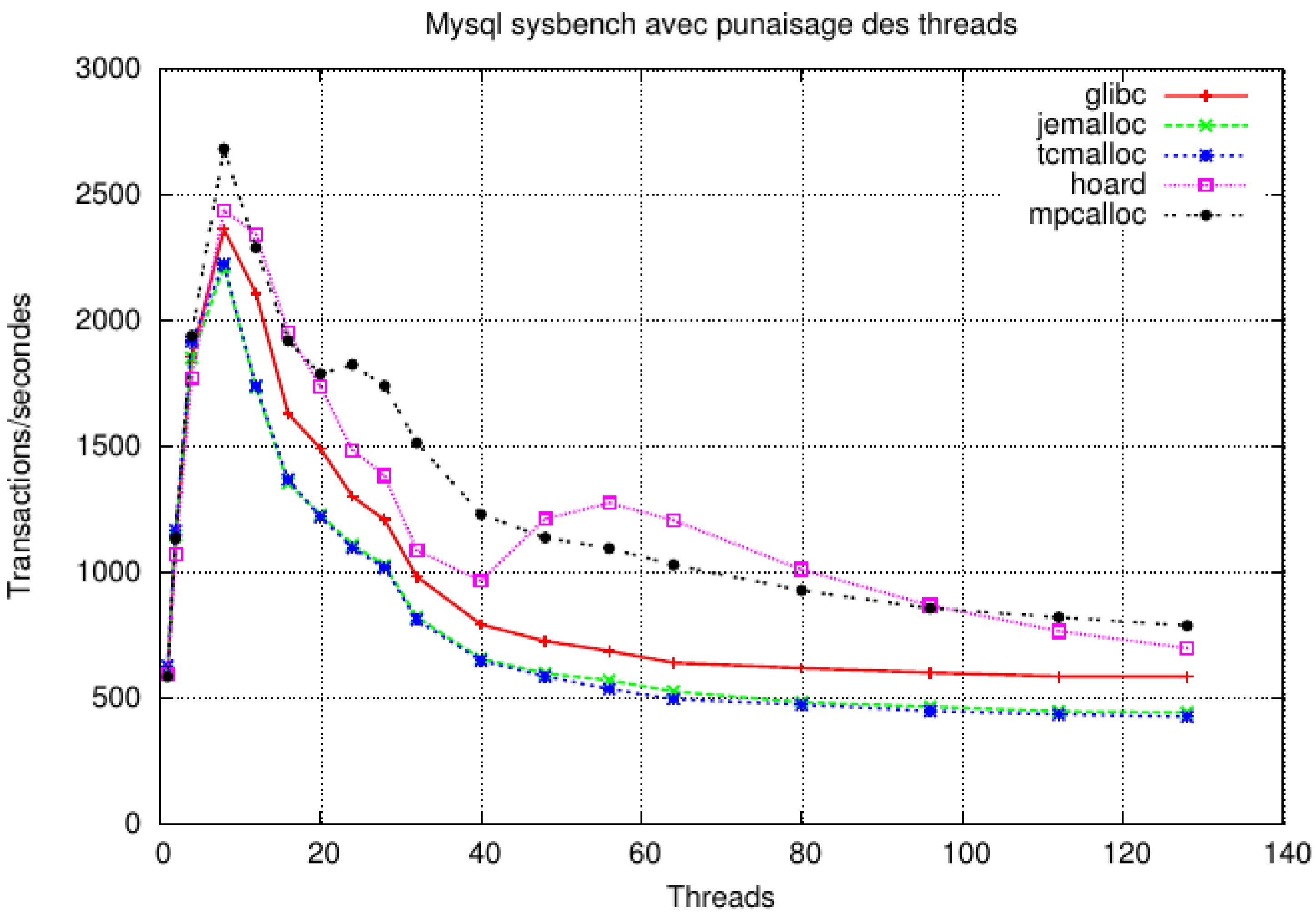


Execution time (s)



Physical memory (GB)

Mysql results



I. Introduction
II. Analysis of OS paging policy
III. Allocator for HPC applications
IV. Page zeroing in Linux first touch handler
V. Conclusion and future work

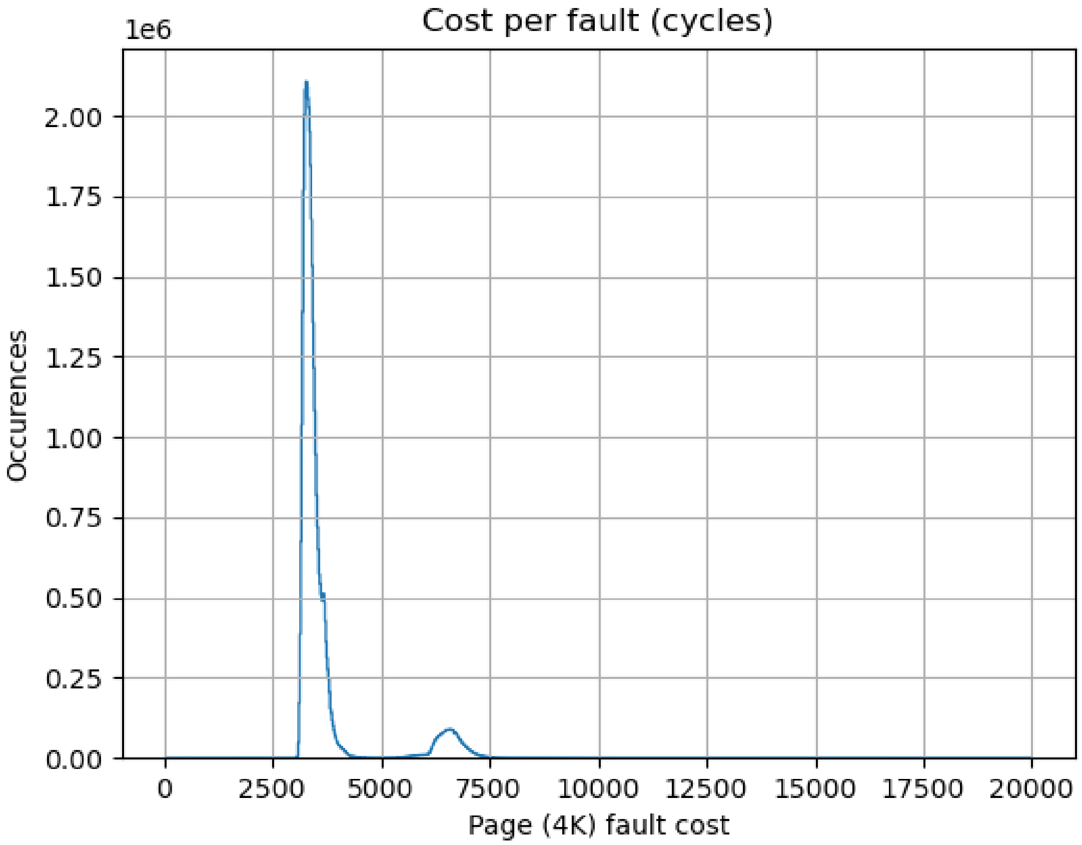
PAGE ZEROING IN LINUX FIRST TOUCH HANDLER

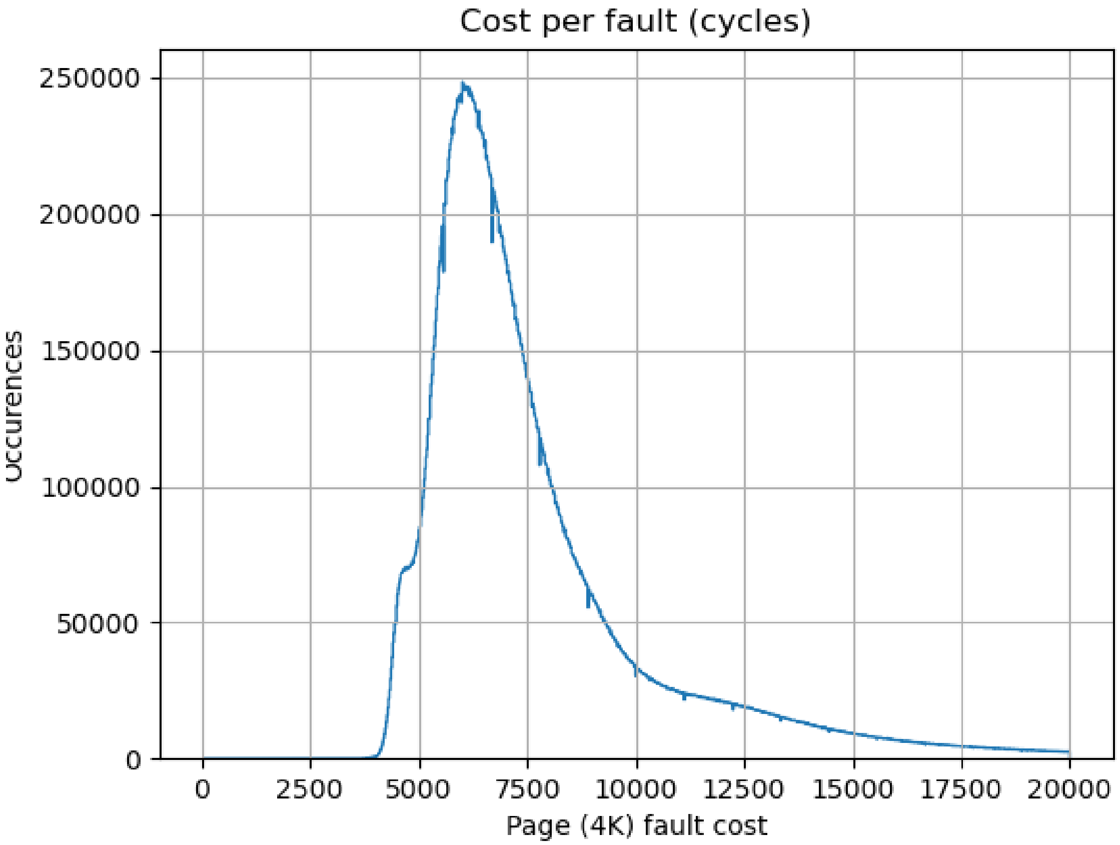
Benchmarking page faults

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

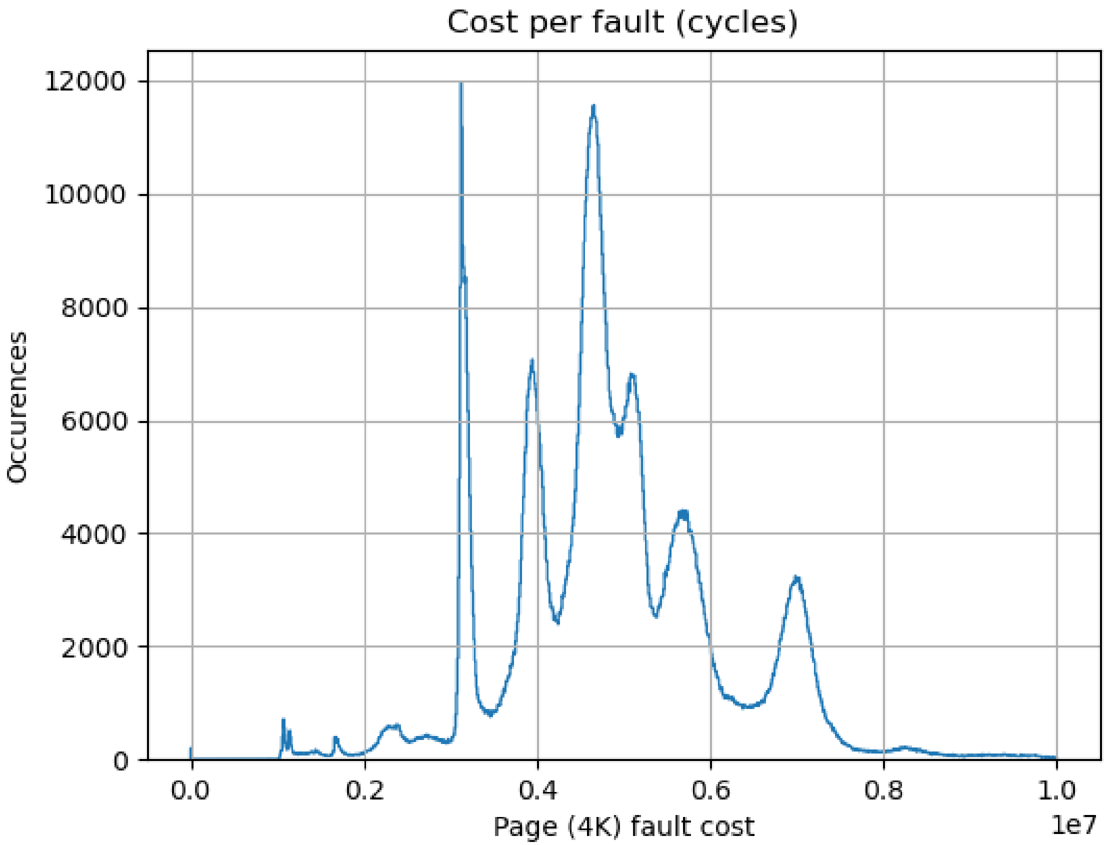
```
ptr = mmap(SIZE);
#pragma omp parallel for
for ( i = 0 ; i < SIZE ; i += PAGE_SIZE)
{
         TIME_DISTRIBUTION(ptr[i] = 0);
}
```

On my adctual latop – page fault costs (4K pages)





Using also the energy efficient cores

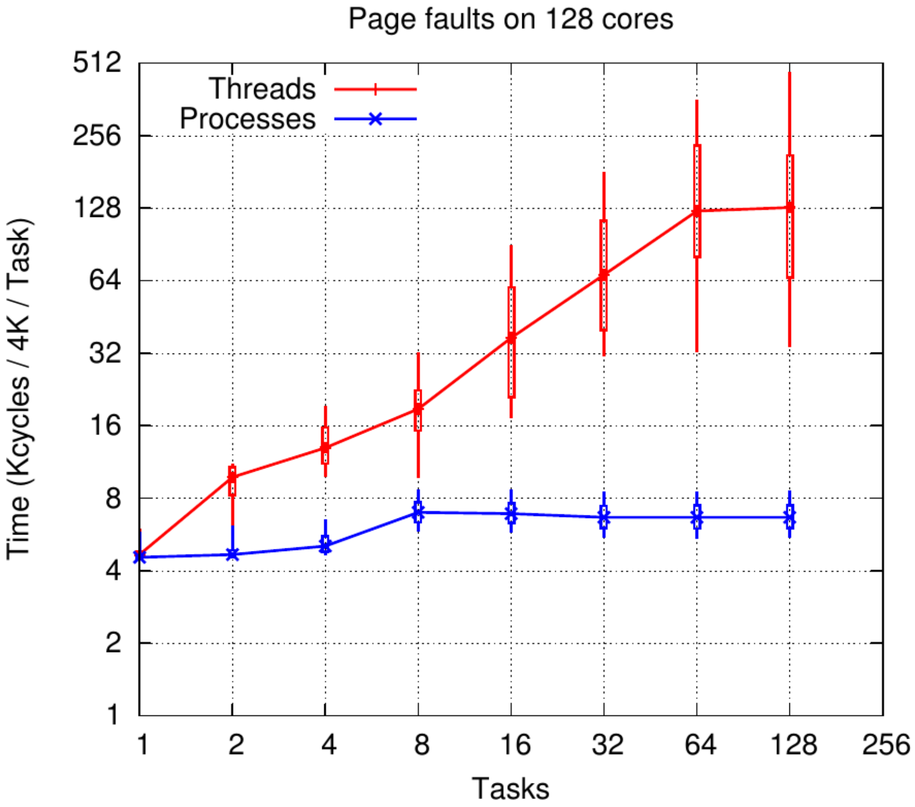


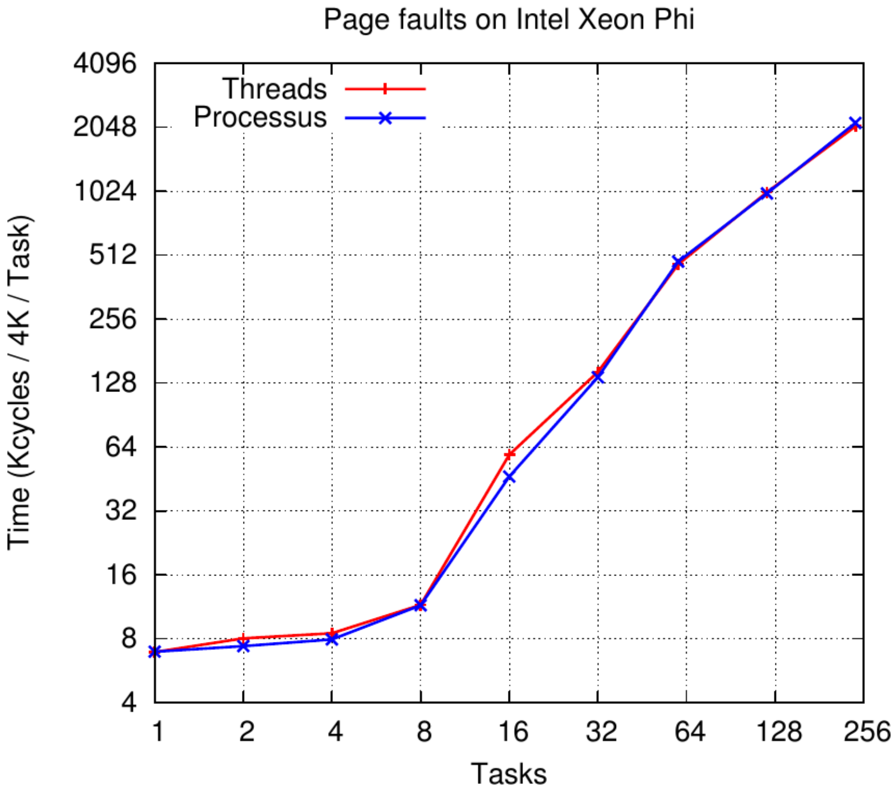
Is **average** + **standard deviation** a right **observable** ?

Is **median + 10% quartiles better**

Page fault scalability

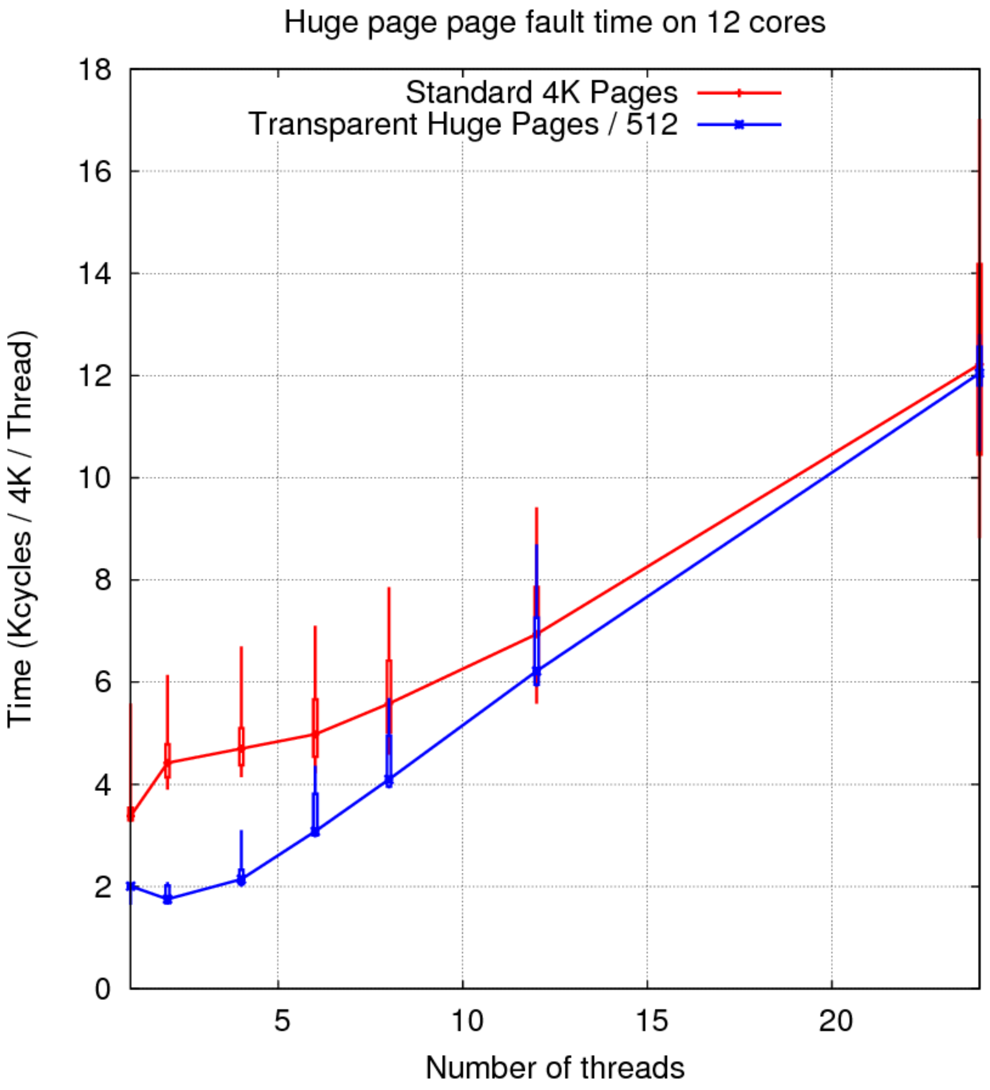
- **Are page faults scalable** ? Over threads or processes.
- Mesurement on **4*4 Nehalem-EP** (128 cores) and on **Xeon Phi** (60 cores)
- **Get scalability issue !**





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**

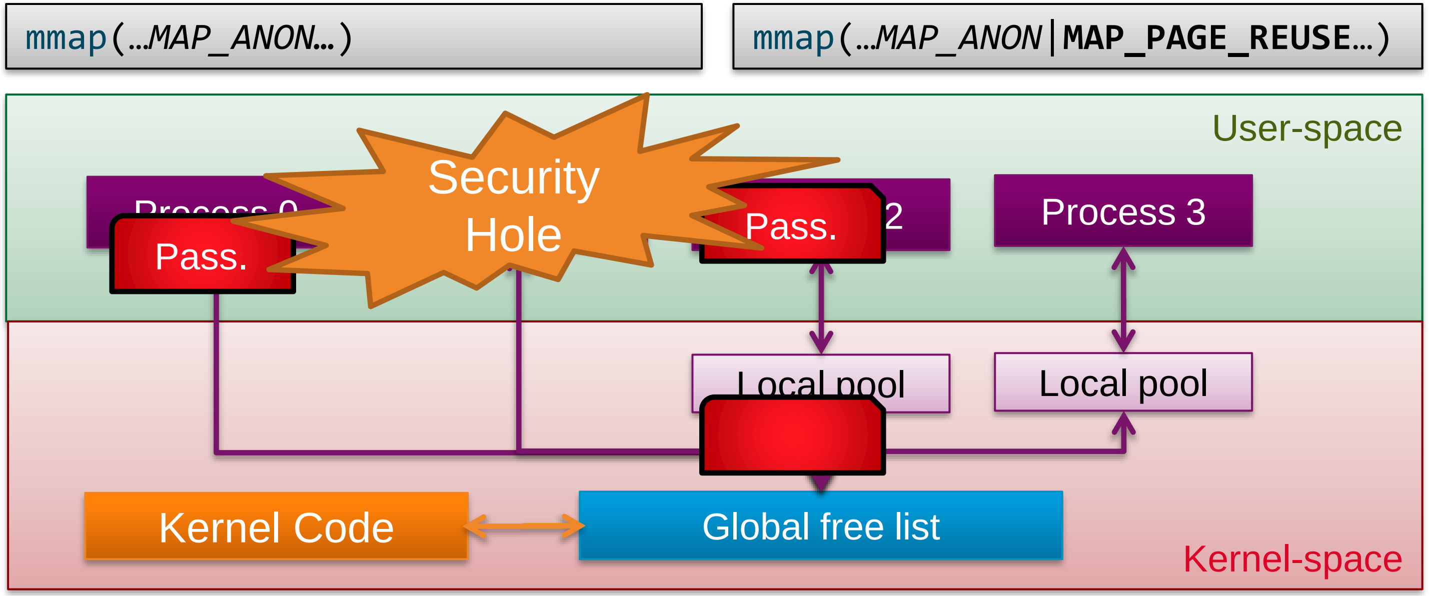
- **Allocation pattern** follow:

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

- Why not **avoid them** ?

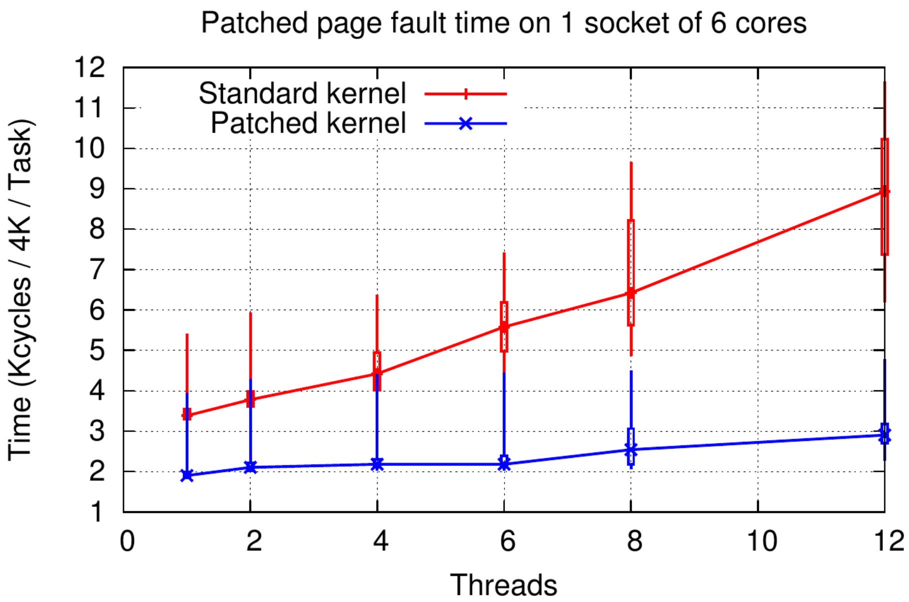
Reusing local pages to avoid zeroing

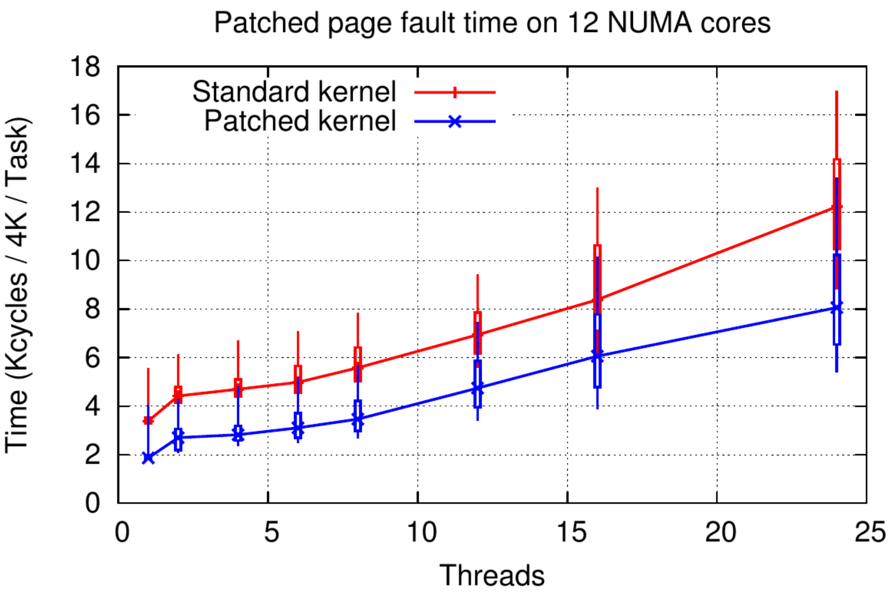
- Page zeroing is **required** for **security reason**
- It prevents information **leaks** from **another processes** or from the **kernel**.
- **But we can reuse pages locally !**
- Need to **extend** the **mmap semantic** :
- Usable by **malloc / realloc**.



Performance impact

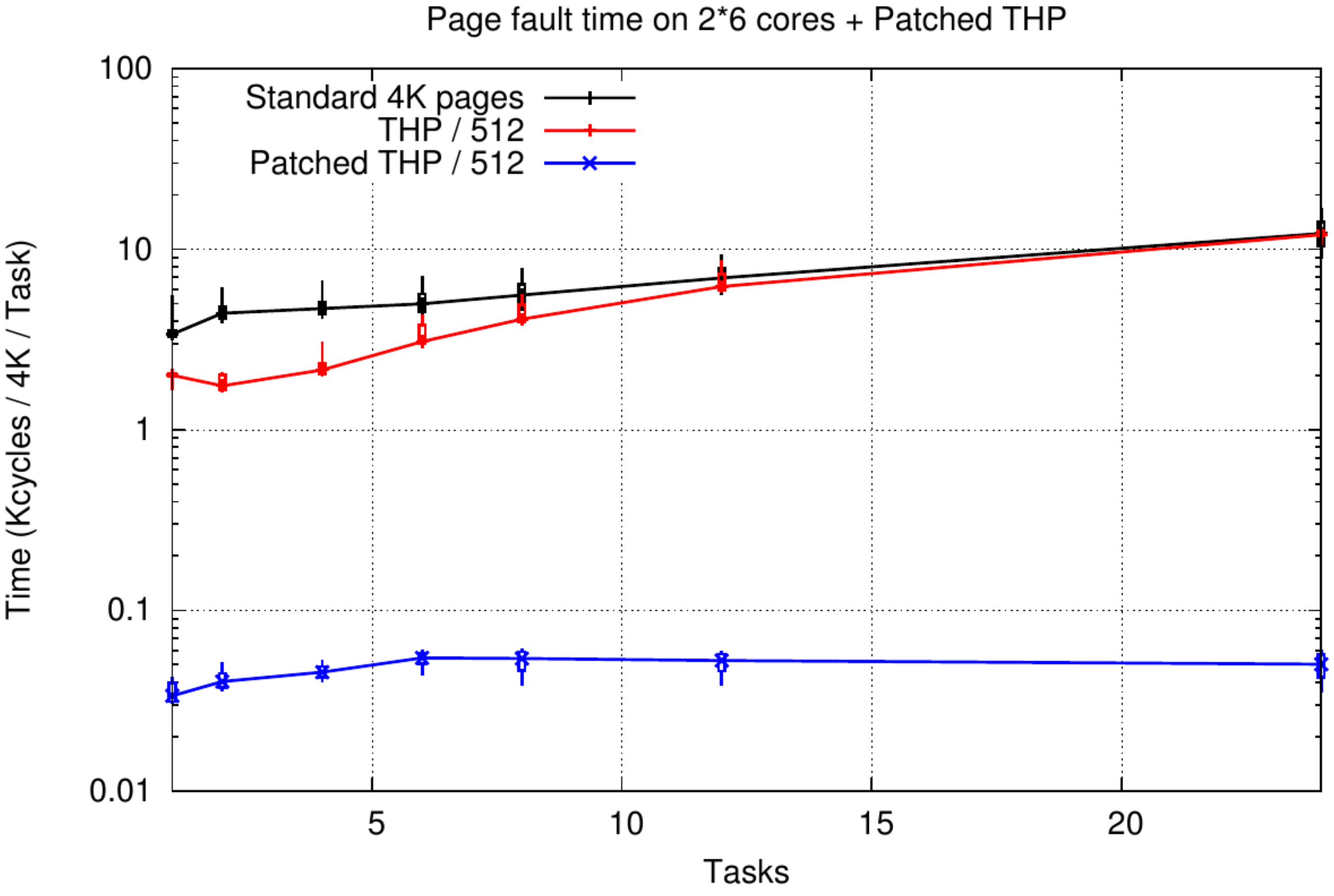
- Get the **expected improvement** on **4K pages** (40% for sequential).
- Also improve **scalability** on 1 socket
- On NUMA **locking effets become dominant for scalability**
- Get the constant improvement related to page zeroing.





Performance impact on huge pages

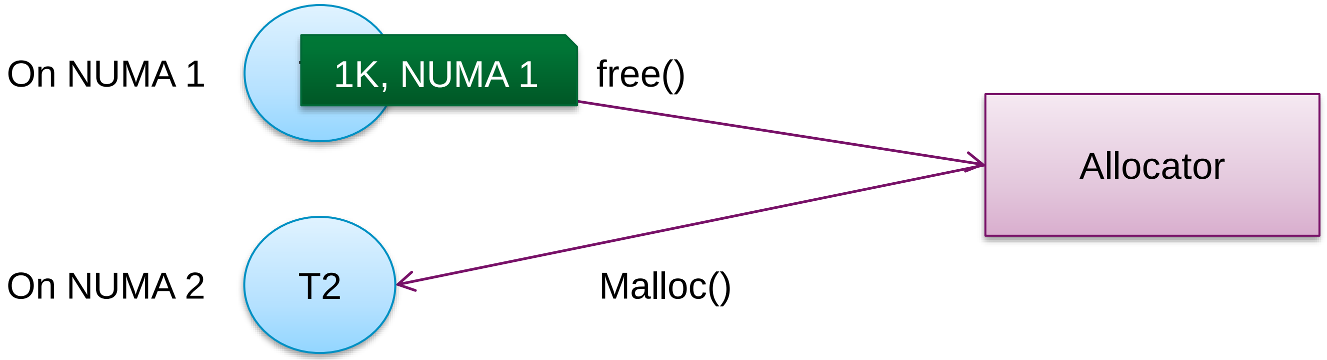
- **Huge pages** (2 MB) faults become **47** times faster, **60** in parallel.
- **New interest** for huge pages.



A SHORT PROBLEM WITH NUMA

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

- C++ tend to have more allocs, so more exposed on NUMA

I. Introduction
II. Analysis of OS / allocator / caches interactions
III. Allocator for HPC applications
IV. Optimization of Linux page fault handler
V. Conclusion and future work

CONCLUSION

Conclusion

- Consider the **genius** of **peoples who invented** the **pagination** !
- Event after **60 years** of memory management we **can still do a lot !**
- Current operating systems **still have to digest** side effects of **multi-core** and **NUMA**
- Impact **can be huge** !
- Hope you know better **what is behind malloc** now !

QUESTIONS ?