DE LA RECHERCHE À L'INDUSTRIE

INTRODUCING KERNEL-LEVEL PAGE REUSE FOR HIGH PERFORMANCE COMPUTING

MSPC 2013

Sébastien Valat[1], Marc Pérache[1,2], William Jalby[2]

[1] CEA,DAM,DIF, F-91297 Arpajon France

[2] University of Versailles Saint-Quentin en Yveline
45 Avenue des États-Unis, Versailles

22 JUNE 2013, SEATTLE

CONTEXT : HPC AND MULTI-CORE

- HPC today, **massively parallel**
- Tera 100 : **140K cores** (2010, 1,05 PFlops)
- Nodes are now **multi-core**
- Tera100 / Curie large nodes : **128 cores**
- Nodes have **NUMA** hierarchy
- To exploit such computer we need **MPI** + **threads**.
- MPC, a unified runtime for **manycore** and **NUMA** architectures.
  (MPI 1.3 / OpenMP 2.5 / Pthreads)
- MPC provides a parallel + NUMA memory allocator.

HOW TO MEASURE MALLOC TIME

- How to measure malloc performance :

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

- Ok for **small blocks**, but not for **large** one (> ~128K) :

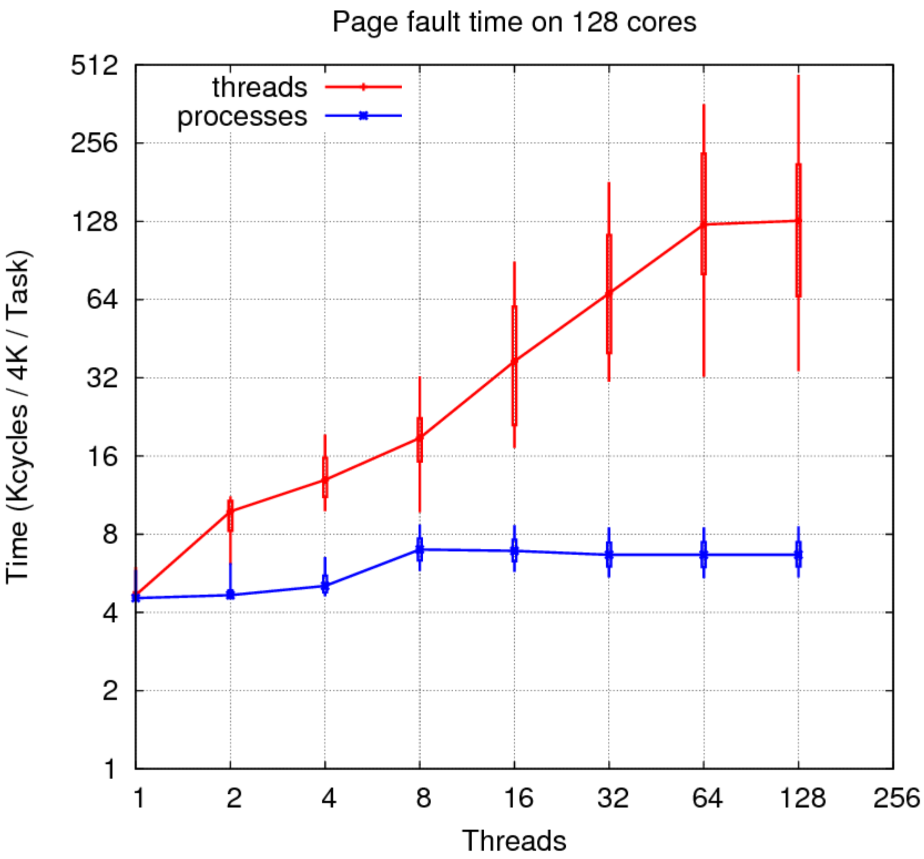
```
T0 = clock_start();
ptr = malloc(SIZE);
for ( i = 0 ; i < SIZE ; i+= PAGE_SIZE)
        ptr[i] = 0;
T1 = clock_end();
```

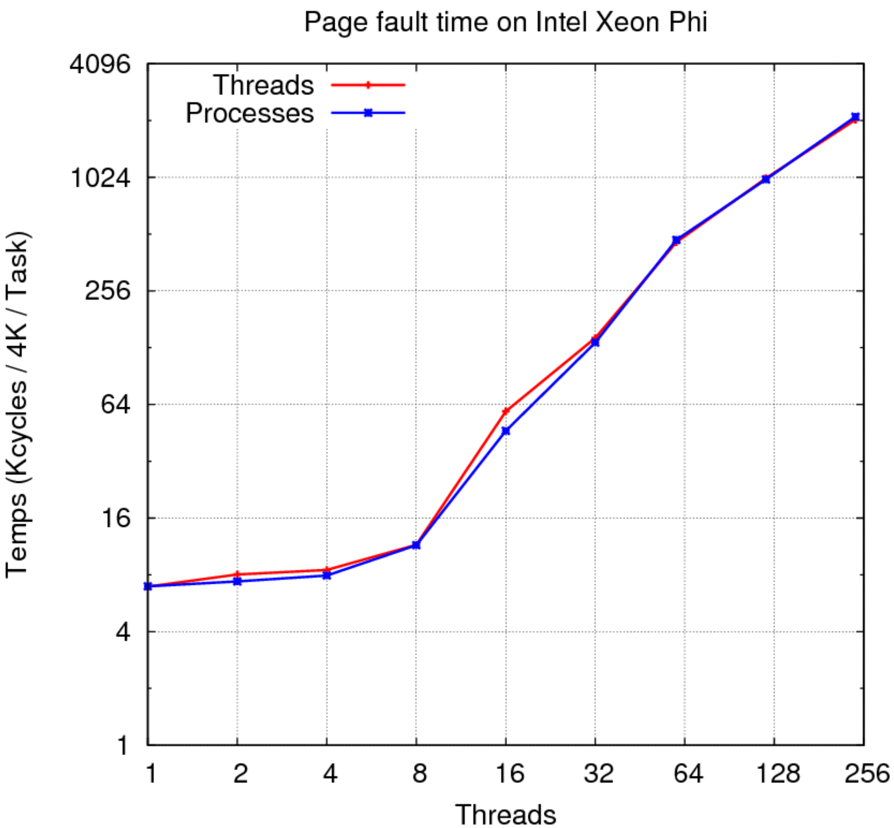
- **Lazy page allocation**.

For 4GB	Malloc	First access	Second access
Time (M cycles)	0,008	1 217	5.4

PAGE FAULT SCALABILITY (LINUX 2.6.3X)

- Are page faults scalable over **threads** and **processes** ?
- **Ideally** fault time must be **constant**.
- Measurement on **4*4 Nehalem-EP** (128 cores) and on **Intel Xeon Phi** :





SCALABILITY ISSUE

- Page faults are **not scalable** over **threads**

- Some **applications** are **memory intensive** : Hera
  - Large MPI C++ hydrodynamic platform
  - 3D **AMR** meshes
  - Multi-physic / multi-material

- Solutions for applications :
  - Improve applications (not trivial for large one)
  - User-space memory pools (increase memory consumption)
  - **Improve the OS**

COMPARE ALLOCATORS

- **Compare** to production grade **allocators**

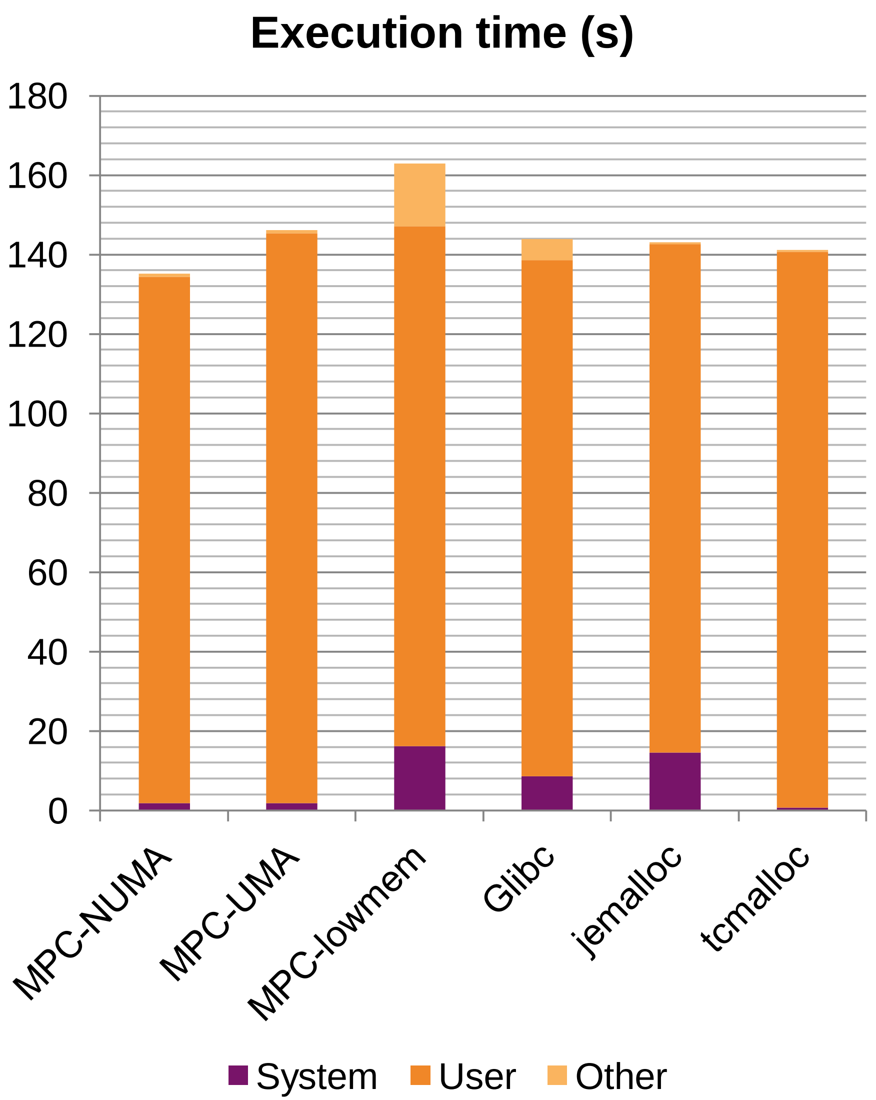
- The default one from **glibc**

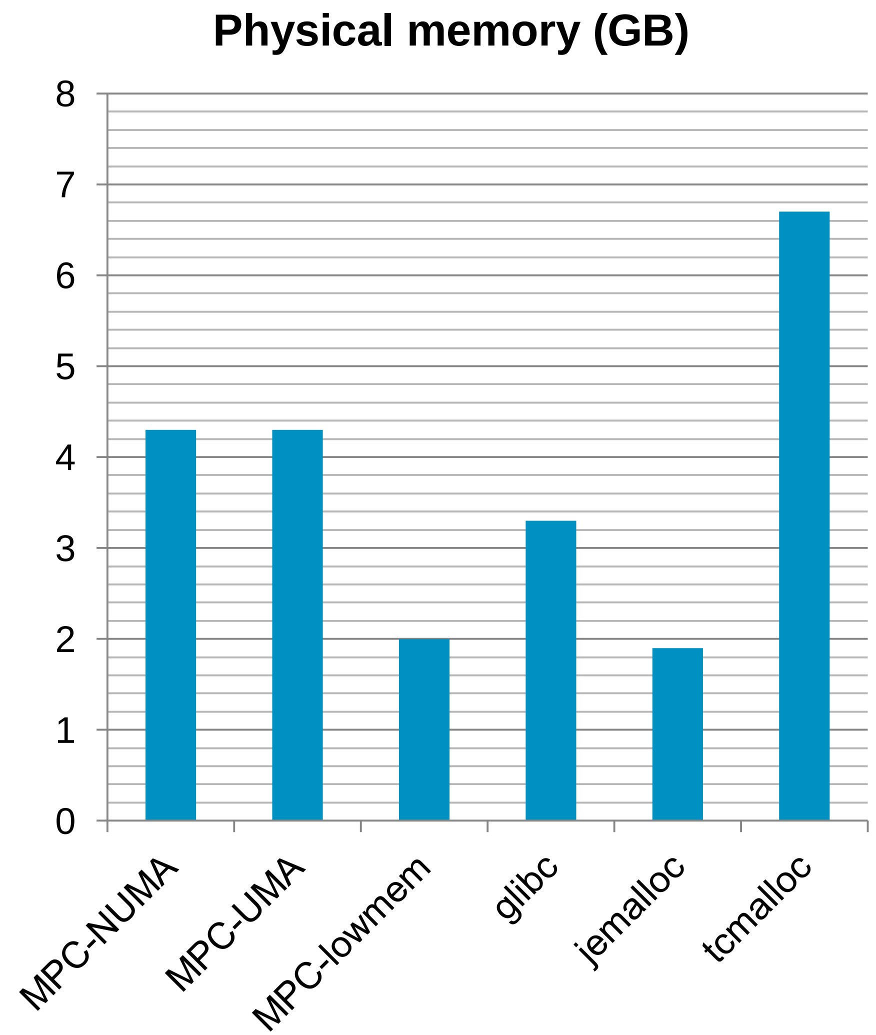
- Jemalloc (FreeBSD) :
  - Parallel
  - **Lower memory footprint**
  - May generate **too much call to the OS**

- TCMalloc (Google) :
  - Parallel
  - **Keep memory** for **fast reuse**
  - Get **larger memory consumption**

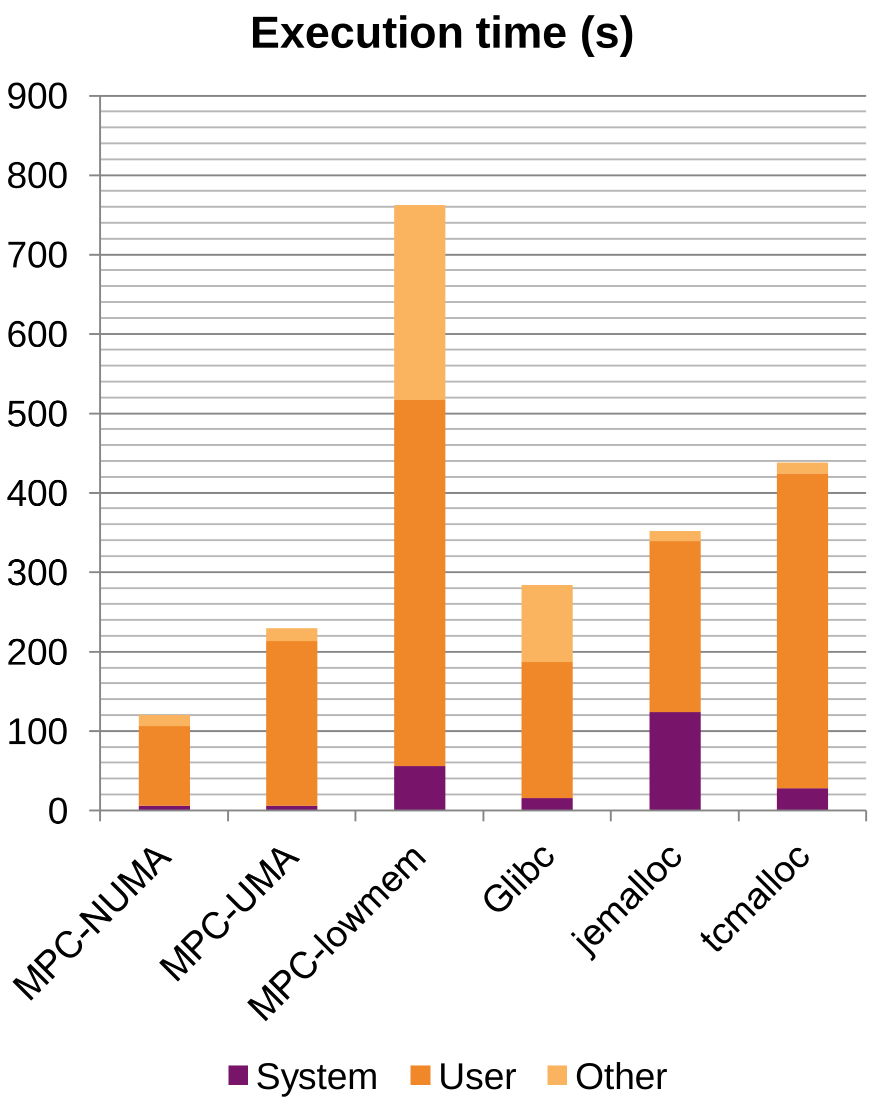
- MPC :
  - Parallel
  - **Reuse** large memory segments (>1MB)
  - Explicit **NUMA** support
  - Two memory **profiles** (resp. comparable to Jemalloc/TCMalloc)

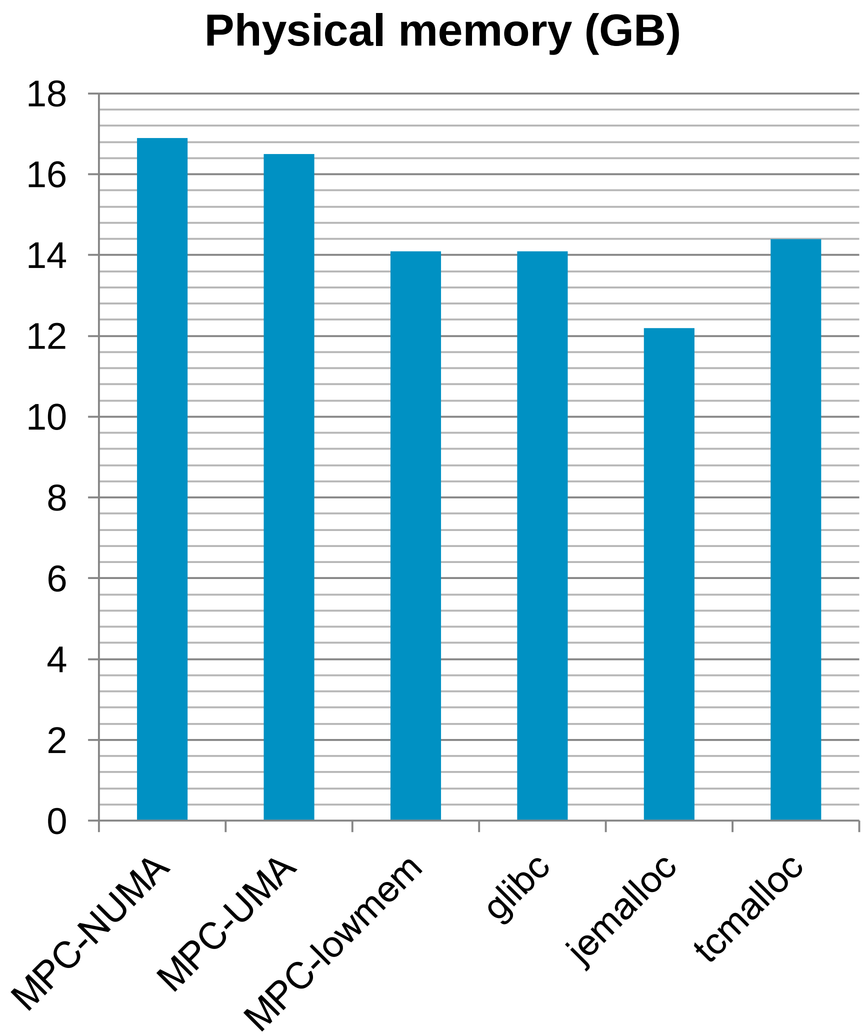
HERA + MPC ON A BI-WESTMERE (12 : 2 * 6 CORES)





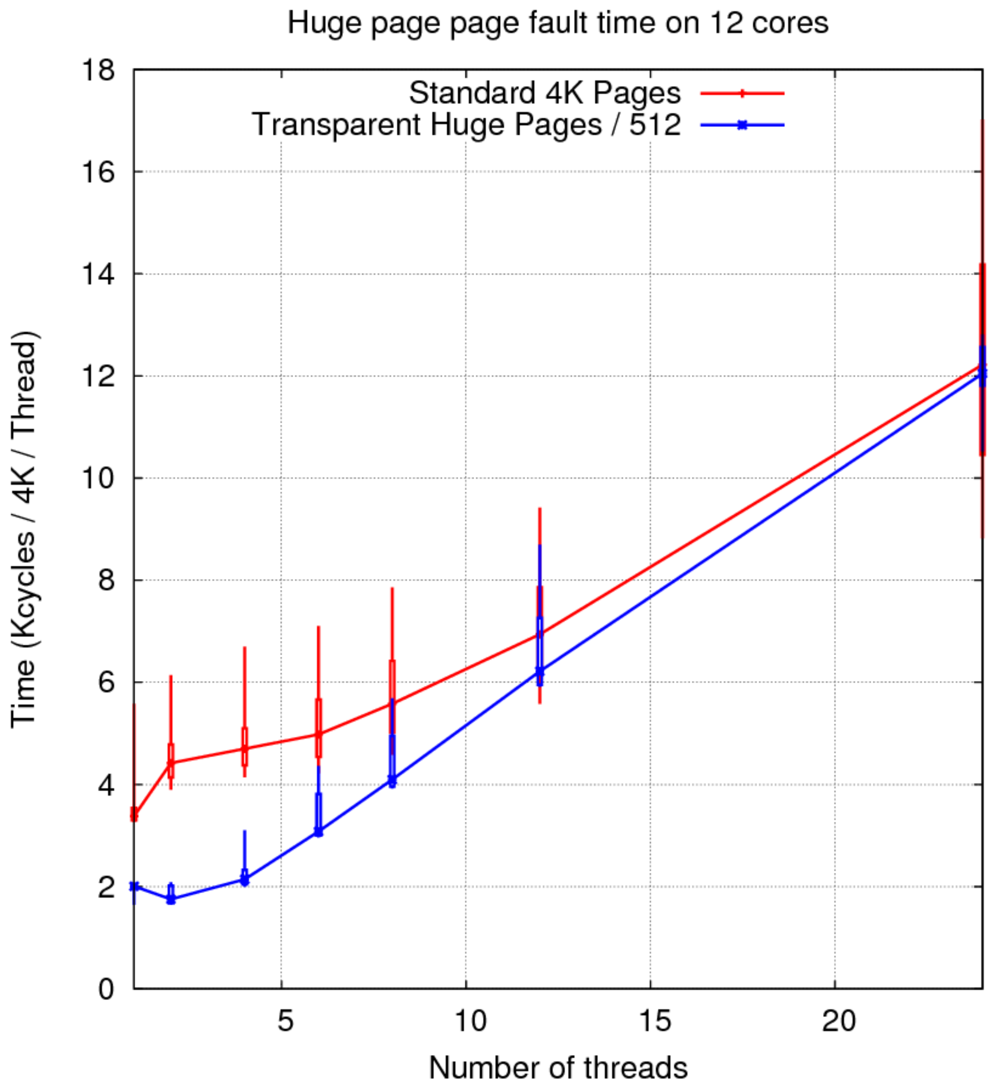
HERA + MPC ON A NEHALEM-EP (128 : 4*4*8 CORES)





CAN HUGE PAGES SOLVE THE ISSUE ?

- Standard pages : **4K**
- Huge pages (x86_64) : **2M**
- **Divide number of faults by 512**
- Can we improve performances ?
  - Sequential : **only 40%**
  - Parallel **: No**
- **Why ?**



WHAT APPENS ON FIRST TOUCH PAGE FAULT ?

- Hardware generates an interruption
- Jump to the OS
- Check reason of the fault
- Request a free page to NUMA **free lists**
- **Reset the page content**
- Map the page, update the **page table**
- It was done for all 4K pages (262 144 times for 1GB)

Possible issue on Xeon Phi

~1400/3400 cycles
40%

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

HOW TO AVOID PAGE ZEROING COST

- Windows use a system thread
- So at fault time, pages are already cleaned
- But **zeroing** :
  - Is **unproductive**
  - Consume CPU **cycles** so **energy**
  - Consume **memory bandwidth**
- Why not to **avoid them** ?
- Most allocations pattern follow :

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

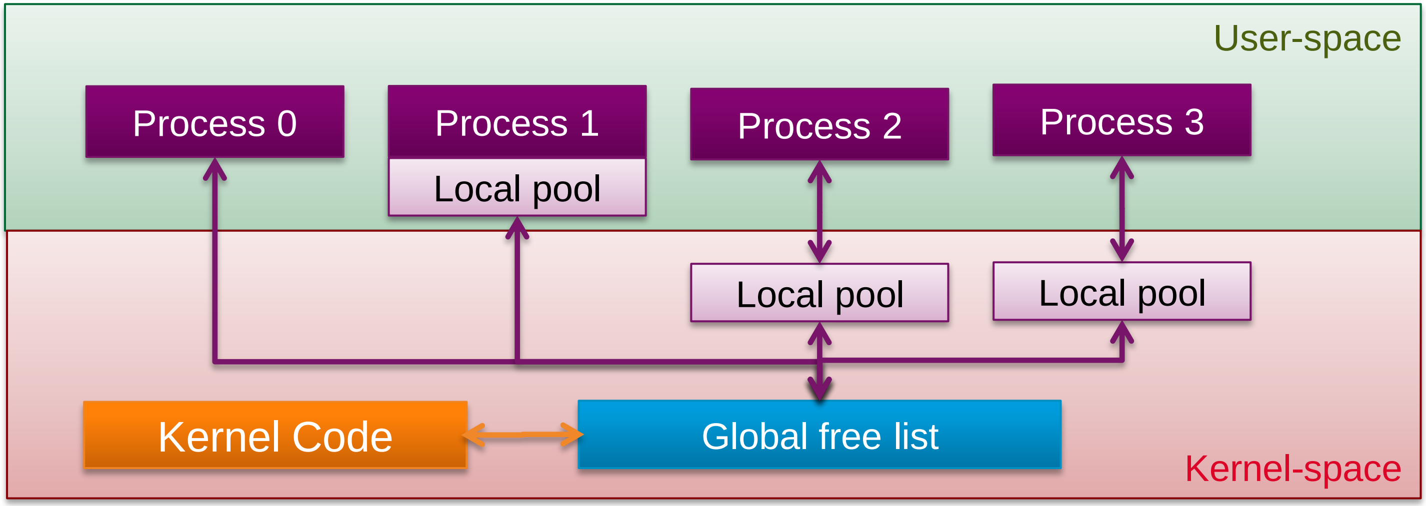
- Why not **inform the kernel** that we do not need zeros ?

REUSING LOCAL PAGES TO AVOID ZEROING

- We can **extend** the **mmap semantic** :
- Page zeroing is **required** for **security reason**
- It prevent information **leaks** from **another processes** or from the **kernel**.
- **But we can reuse pages locally !**

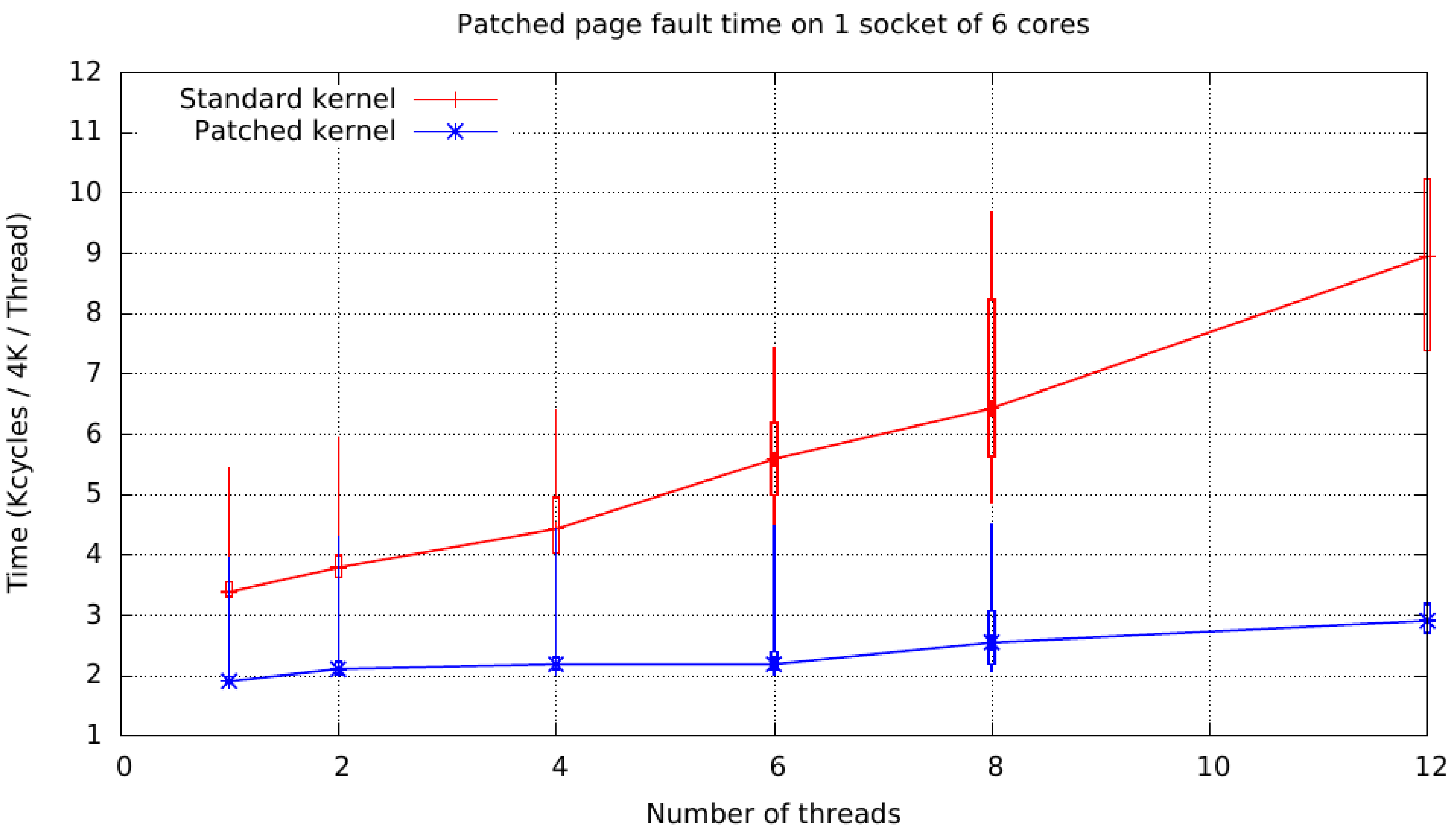
`mmap(…MAP_ANON…)`

`mmap(…MAP_ANON|`**`MAP_PAGE_REUSE`**`…)`



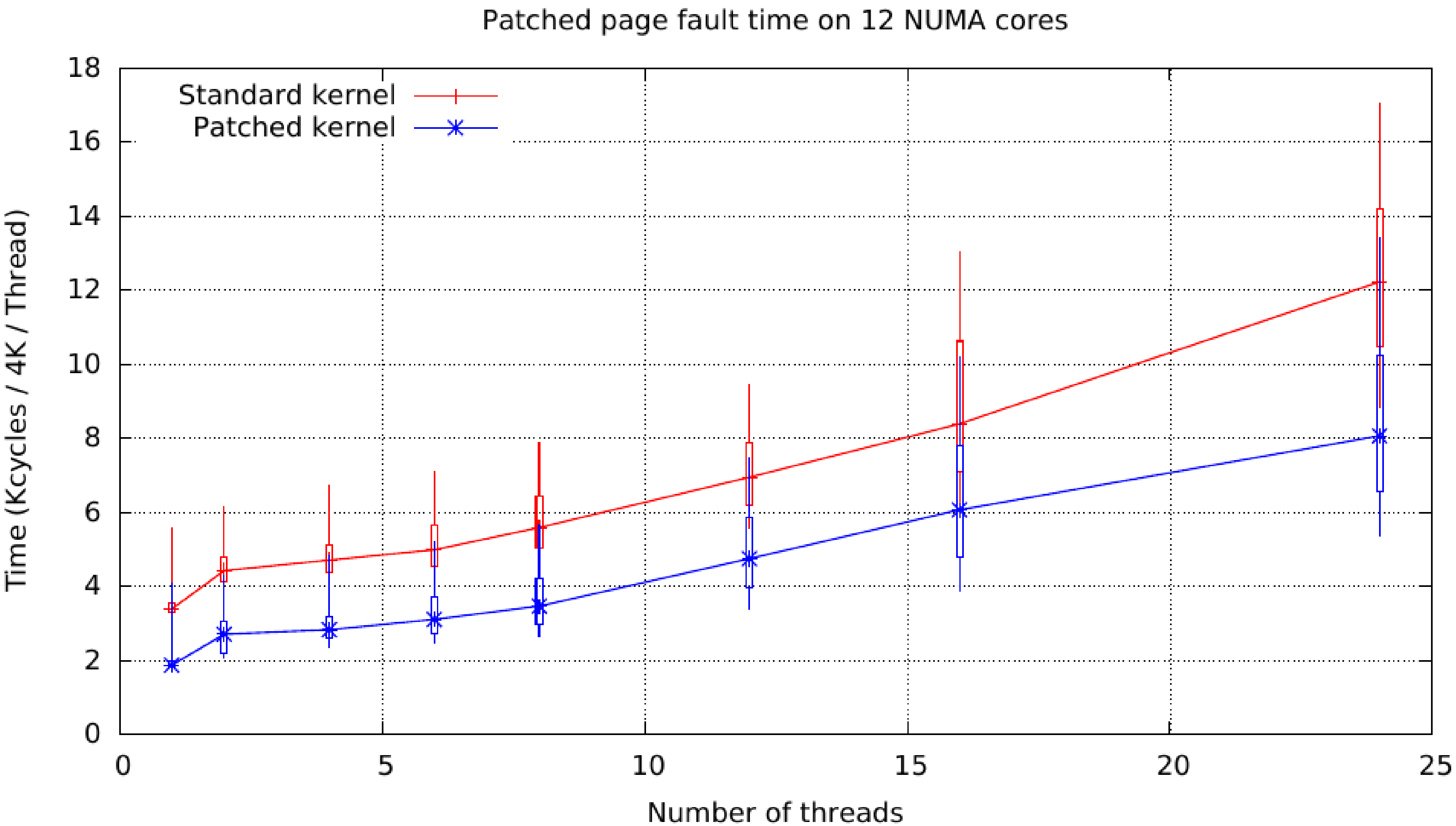
IMPROVEMENT OF FAULTS ON 6 CORE WESTMERE

- Without NUMA, get good results.



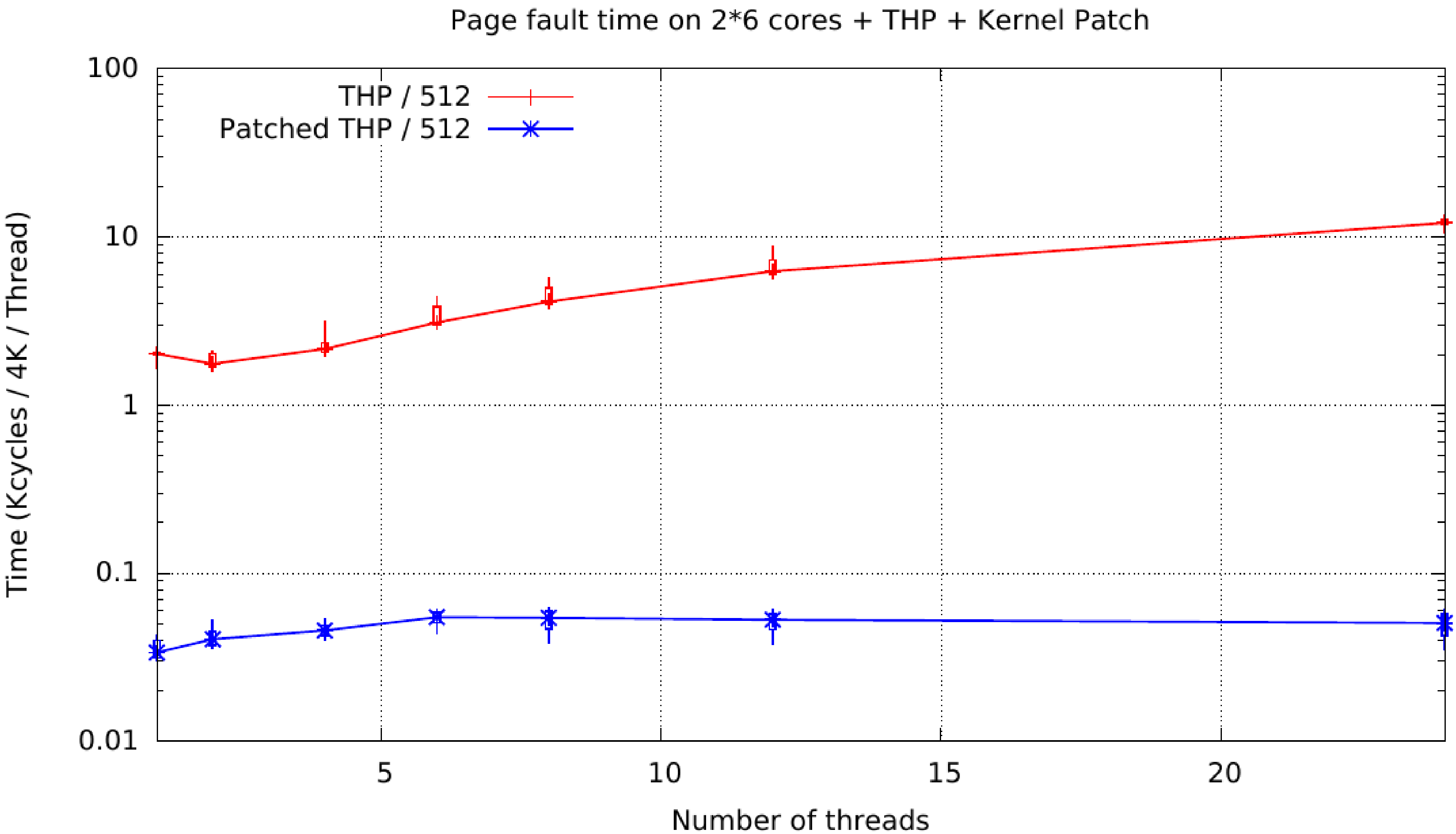
USING TWO SOCKETS (NUMA)

- Our patch improve performance, but **NUMA effects** due to locks became dominant.



EFFECT ON HUGE PAGES

- Get huge improvement (x60), **new interest for huge pages**.



RESULTATS HERA SUR BI-WESTMERE (2*6 COEURS)

Standad pages (4K) :

Allocator	Kernel	Total (s)	Sys. (s)	Mem. (GB)
Glibc	Std.	144	9	3,3
MPC-NUMA	Std.	**135**	**2**	**4,3**
MPC-Lowmem	Std.	162	16	**2,0**
MPC-Lowmem	**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
MPC-NUMA	Std.	**138**	**2**	**6,2**
MPC-Lowmem	Std.	196	28	3,9
MPC-Lowmem	**Patched**	138	3	3,8
Jemalloc	Std.	145	15	2,5
Jemalloc	Patched	138	6	3,2

WHERE TO PLACE MEMORY POOLS ?

	User-space	Kernel-space	
Sizes	*	4KB	2MB
Controlling memory	Virtual	Physical	
Limit mono process consumption	~	+	
Limit multi-processes consumption	-	~ / + ?	
Adaptation to real access pattern	-	+	
Ease of implementation	+	-	
Support of NUMA	~	+	
Performance gain	+	~	+

CONCLUSION AND FUTURE WORK

Conclusion

- **Page zeroing** account for **~40%** in sequential !
- **Extend** `mmap/madvise` **semantic** to remove need of page zeroing.
- Get the expected 40% sequential improvement with 4K pages.
- **New interest for huge pages** (reduction of x60)**.**

Future work and open question

- Integration in page reclaim algorithm.
- Still limited by **lock scalability** on **NUMA !**
- What is a **good huge page size ?** 2M **too large** ?

QUESTIONS ?

BACKUPS

OBSERVATIONS

- Need to find **balance** between **consumption** vs. **performances**.
- On **128 cores**, improvement of **20%** for **2GB**.
- With **NUMA** support, improvement of **48%** compared to the best one.
- Can we **improve** the **consumption / performance ratio** ?

INTEGRATION WITH PAGE RECLAIM

- **Consumption** currently limited to the **maximum working set** of the application.
- Need more work to support “**page reclaim**” in case of memory famine.
- Page reclaim functions need to loops overs local pools before swaping.
- In case of repetitive reclaims, disable usage of local pools until lower memory pressure.
- General aspects of swap integration was looked, but not implemented.

SEMATIC COMPATIBILITY

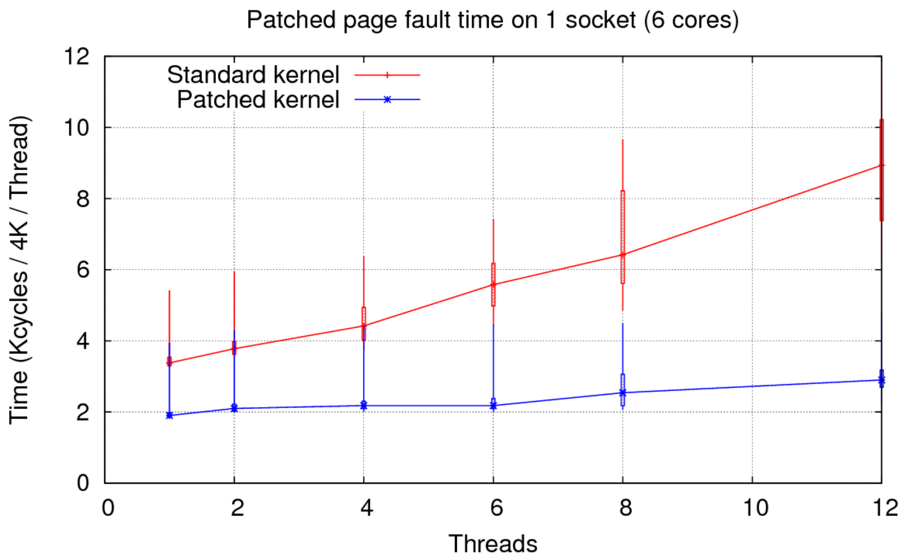
- Codes can rely on lazy page zeroing!
- Cannot enable it by default.
- Need explicit demand with `mmap / madvise` flags :

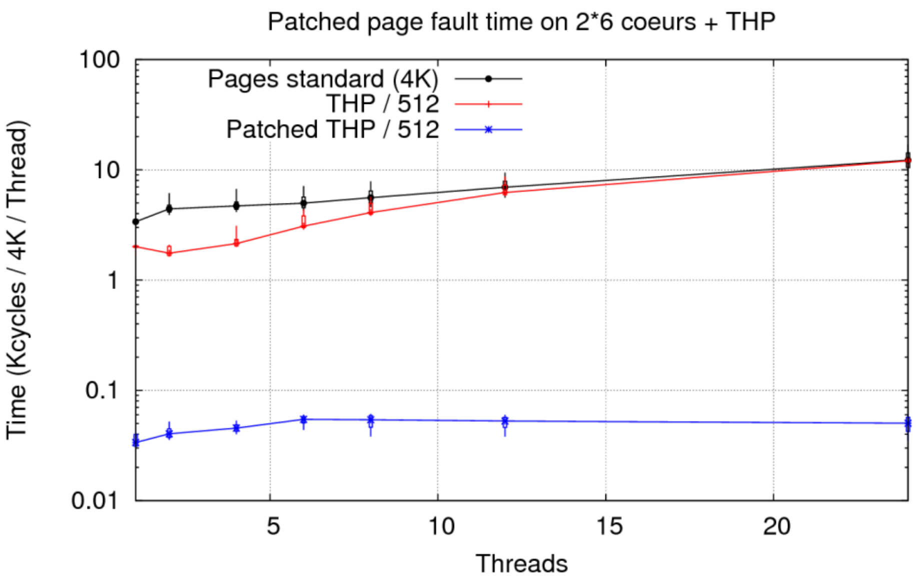
```
void * ptr = mmap(... MAP_ANON | MAP_PAGE_REUSE …);
munmap(ptr);

ptr = mmap(... MAP_ANON …);
madvise(…MADV_PAGE_REUSE);
munmap(ptr);
```

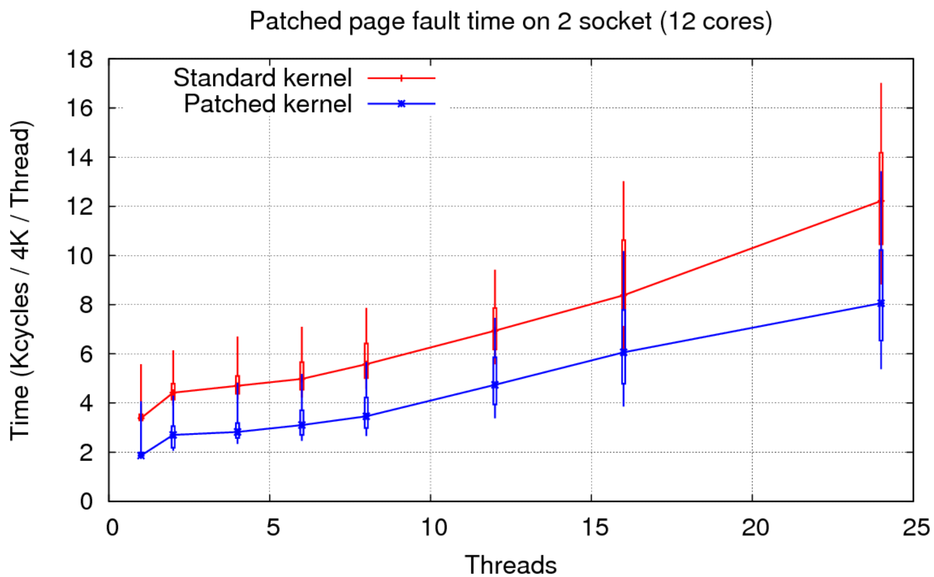
- Need to patch malloc/realloc, tested support in
  - MPC_Allocator
  - Jemalloc

IMPROVEMENT OF FAULTS ON BI-WESTMERE

- **Improvement for 4K pages**
- But dominated by **NUMA effects** on kernel locks
- Large impact on **huge pages** (2M)
- Observe **new interest** for **huge pages**







INTEGRATION WITH PAGE RECLAIM

- Current implementation implicitly limits the memory consumption to the maximum working set of the application.
- Need more work to support “page reclaim” in case of memory famine.
- Page reclaim functions need to loops overs those local caches before swaping.
- In case of repetitive reclaims, disable usage of local pools until lower memory pressure.
- General aspects of swap integration was looked, but not implemented.

HYDRO RESULTS ON BI-WESTMERE (2*6 CORES)

- Kernel patch and standard 4K pages

App.	Allocator	Kernel	Total (s)	Sys. (s)	MFlops
Std.	Glibc	Std.	**1:29**	30,7	**1770**
Std.	MPC	Std.	**1:28**	31,5	**1775**
Std.	MPC	Patched	**1:19**	19,7	**2004**
Std.	MPC-KeepMem	Std.	**0:59**	0,5	**2649**
Patch.	Glibc	Std.	**0:43**	0,4	**3606**

- Kernel patch and Transparent Huge Pages

App.	Allocator	Kernel	Total (s)	Sys. (s)	MFlops
Std.	Glibc	Std.	**1:13**	18,8	**2140**
Std.	MPC	Std.	**1:18**	17,8	**2007**
Std.	MPC	Patched	**1:11**	7,0	**2224**
Std.	MPC-KeepMem	Std.	**1:05**	1,0	**2412**
Patch.	Glibc	Std.	**0:50**	0,4	**3554**

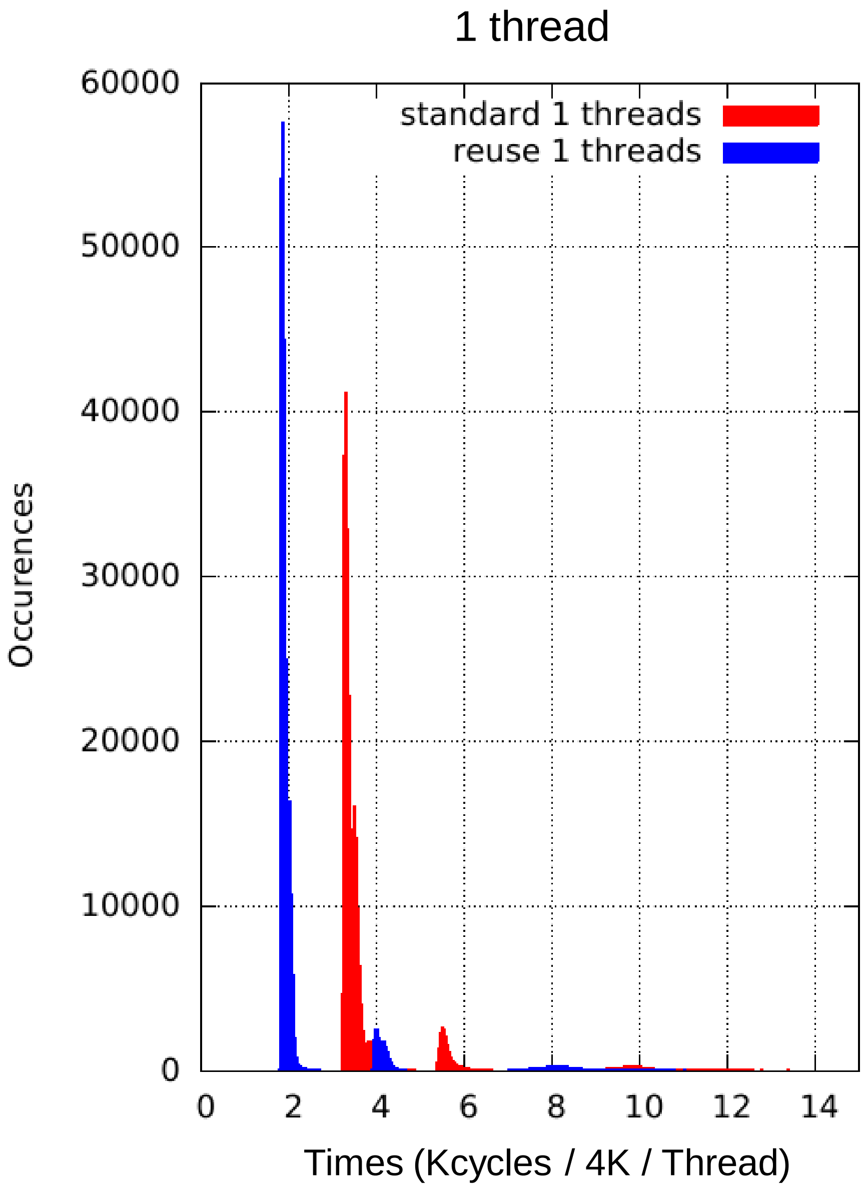
APPLICATION ISSUE

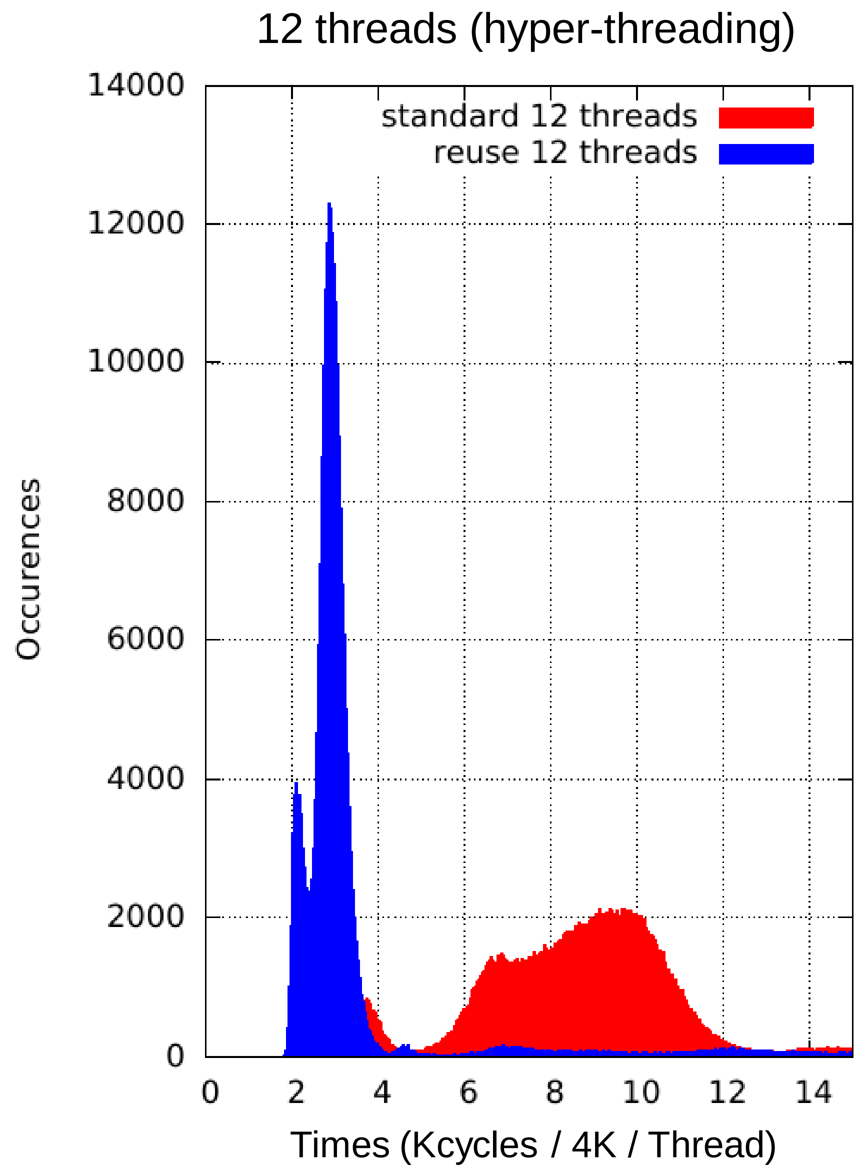
- Some applications are **intensive** in **memory allocations**
- Application **Hera**:
  - Large MPI C++ hydrodynamic platform
  - 3D **AMR** meshes
  - Multi-physic / multi-material
  - We used it with MPC thread-based MPI.
- Application HydroBench:
  - A smaller hydrodynamic MPI / OpenMP benchmark
  - An older version generate large number of memory allocations.
- Memory management can become a **bottleneck**
- **OS (Linux) memory management scalability** ?

OBSERVATIONS

- Need to find **balance** between **consumption** vs. **performances**.
- On **128 cores**, improvement of **20%** for **2GB**.
- With **NUMA** support, improvement of **48%** compared to the best one.
- Can we **improve** the **consumption / performance ratio** ?

IMPROVEMENT OF FAULTS ON 6 CORE WESTMERE



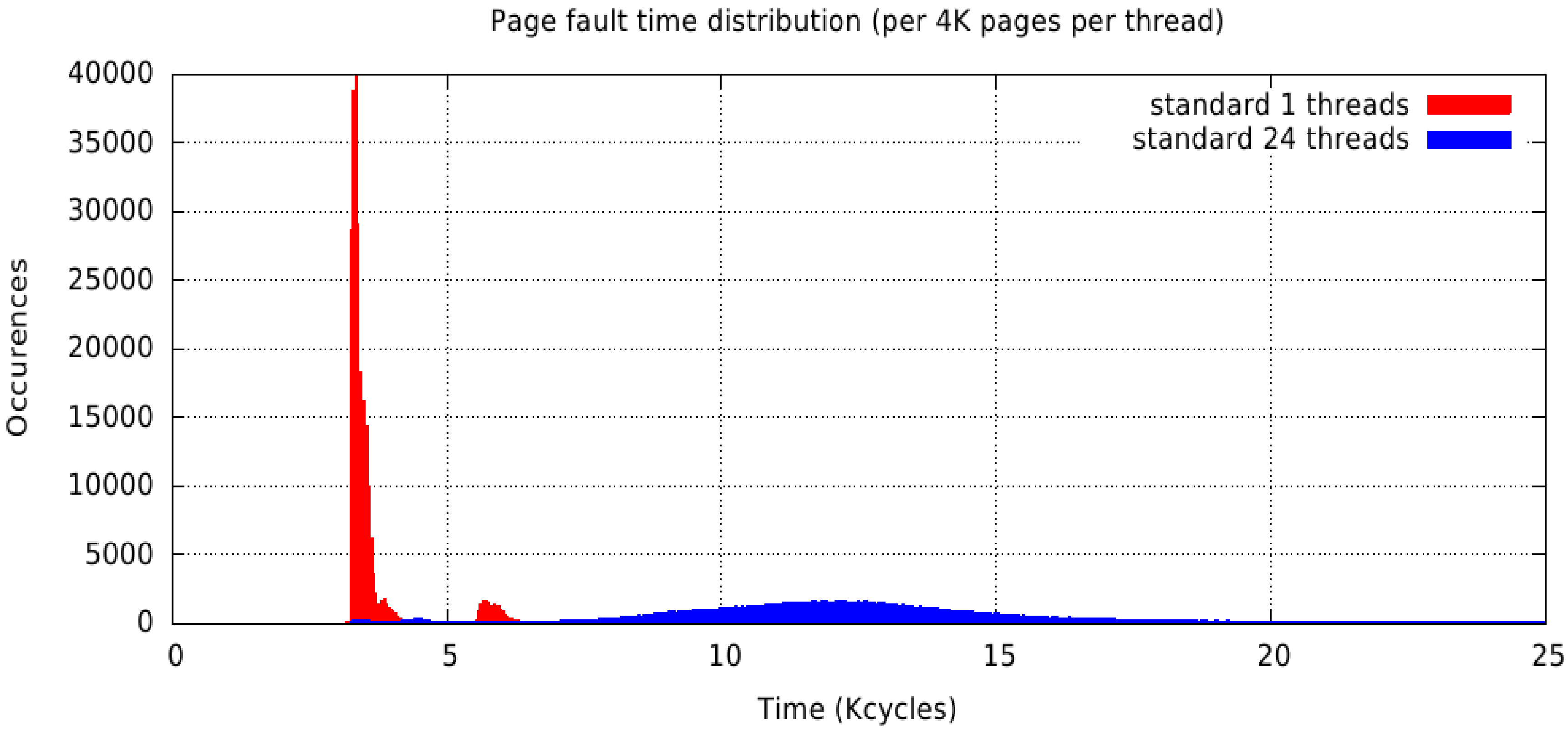


MPC FRAMEWORK

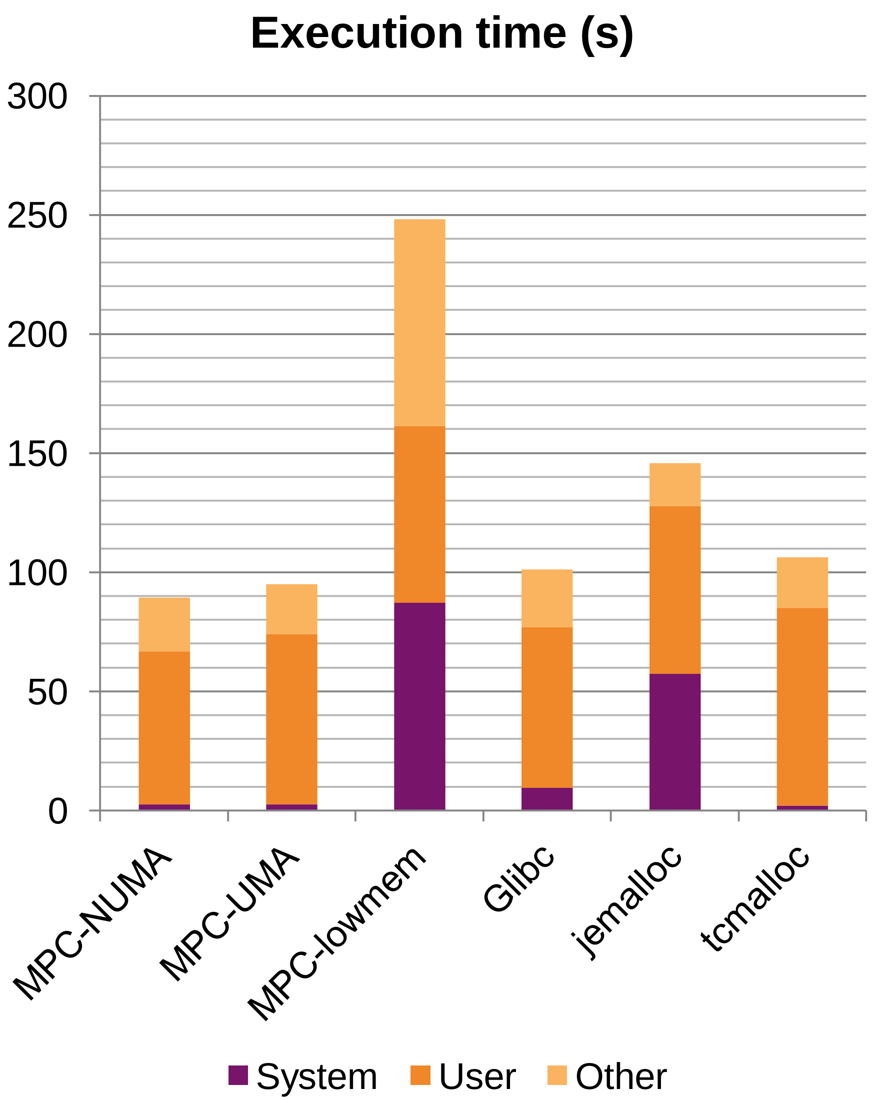
- A **unified runtime** to support **MPI + X** applications.
- Implement **standards** :
  - **MPI** 1.3
  - **OpenMP** 2.5
  - **Pthread**
- Optimized for **manycore** and **NUMA** architectures.
- Provide a **thread-base MPI** mode (tasks are threads, not processes).
- **Be interested in thread performances.**
- **Need parallel and NUMA aware memory allocator.**

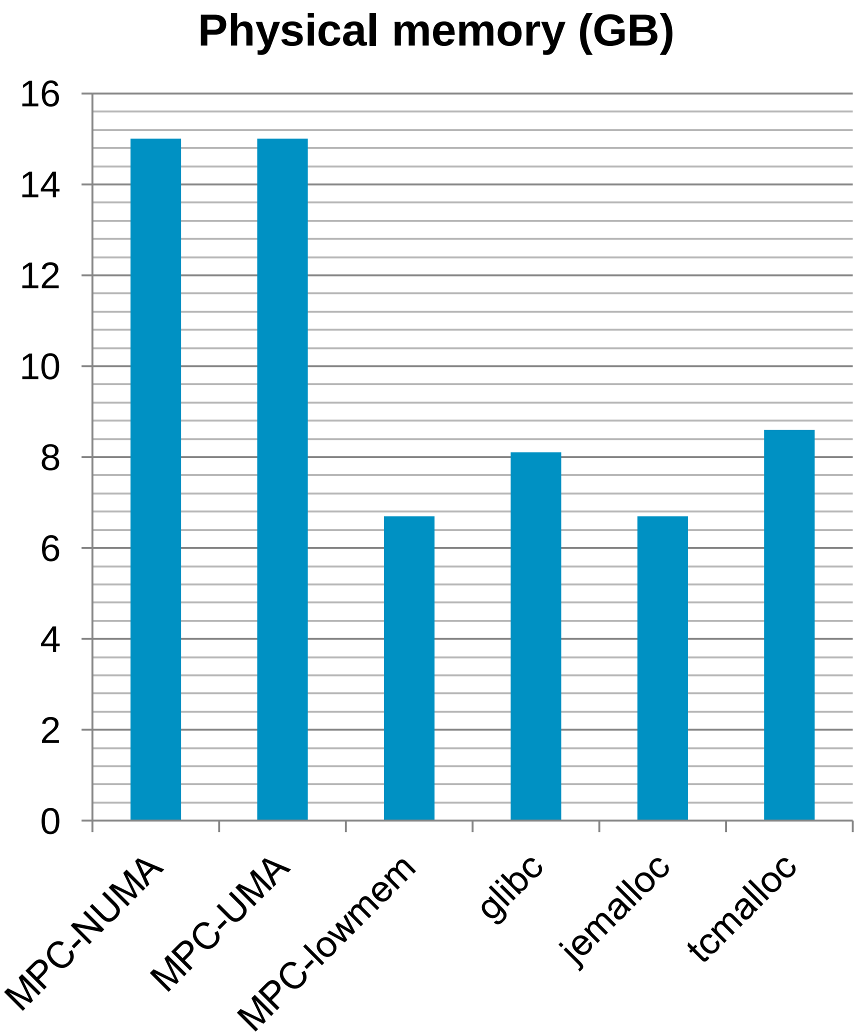
MEASURE PAGE FAULT TIME DISTRIBUTION

- Measure each fault with RDTSC (first access to fresh memory segments).
- Obtain time distribution by repeating many times to observe variability.
- In sequential or parallel.



HERA + MPC ON A NEHALEM-EP (32 : 4 * 8 CORES)





SOME DETAILS

Reuse policy :

- Search the best fitting segment.
- Rely on `mremap` to reuse segments which do not fit with the request.

Limit the consumption :

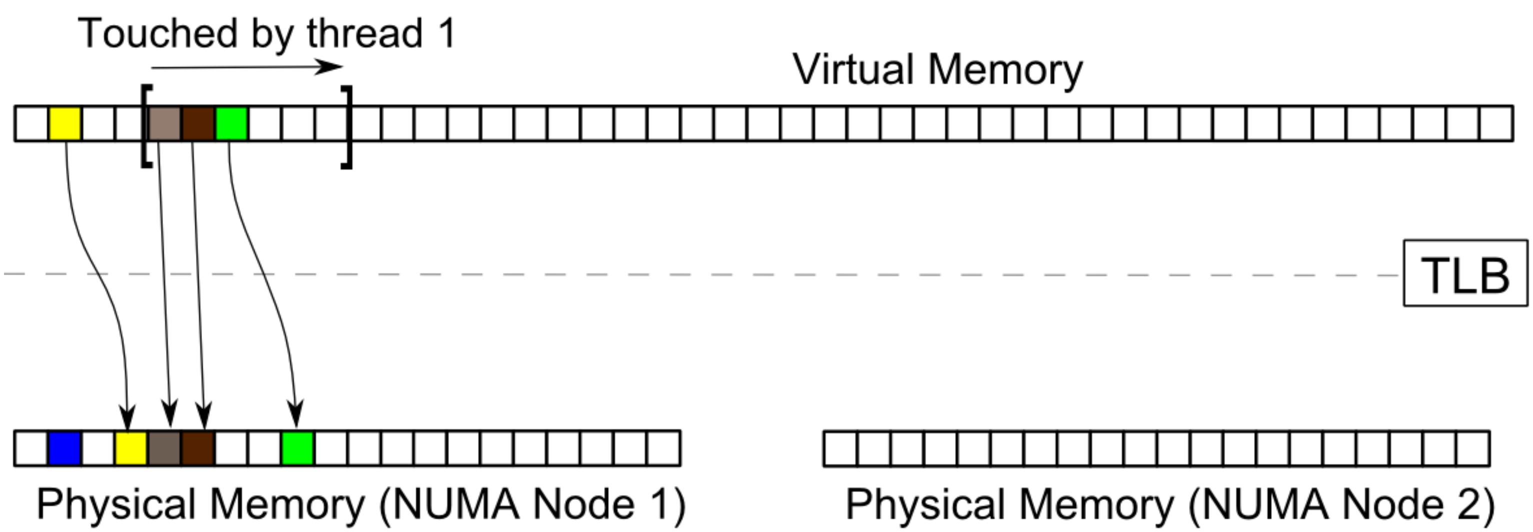
- Keep all segments smaller than configured size (~20 MB).
- Limit the total amount of unused memory (~4GB per NUMA node).

Limitations

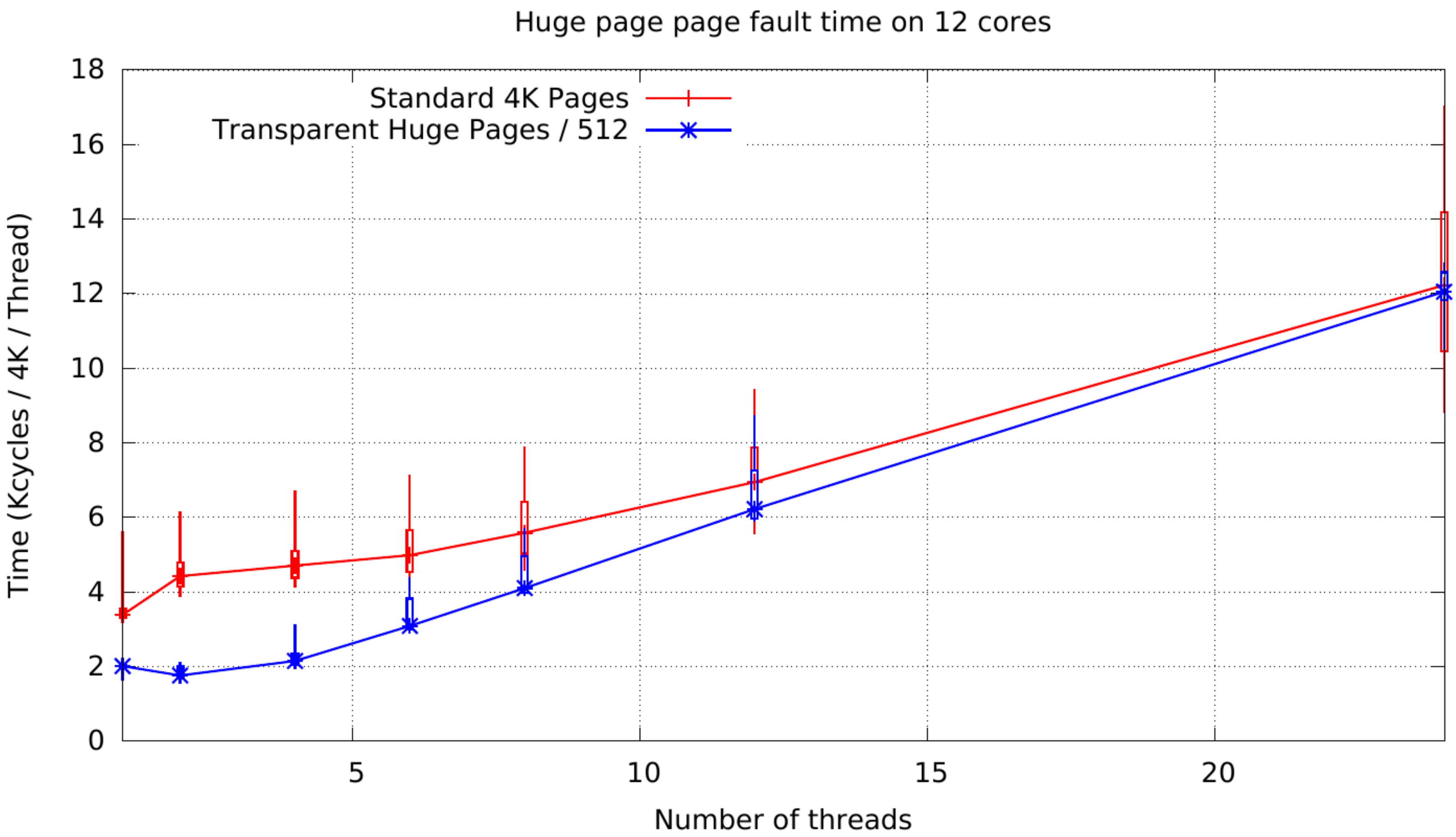
- Larger memory consumption.
- Side effects with application which allocate more memory than required.
- Application dependent parameters, need to automate.

ALLOCATION AND PAGE FAULT SEMANTIC

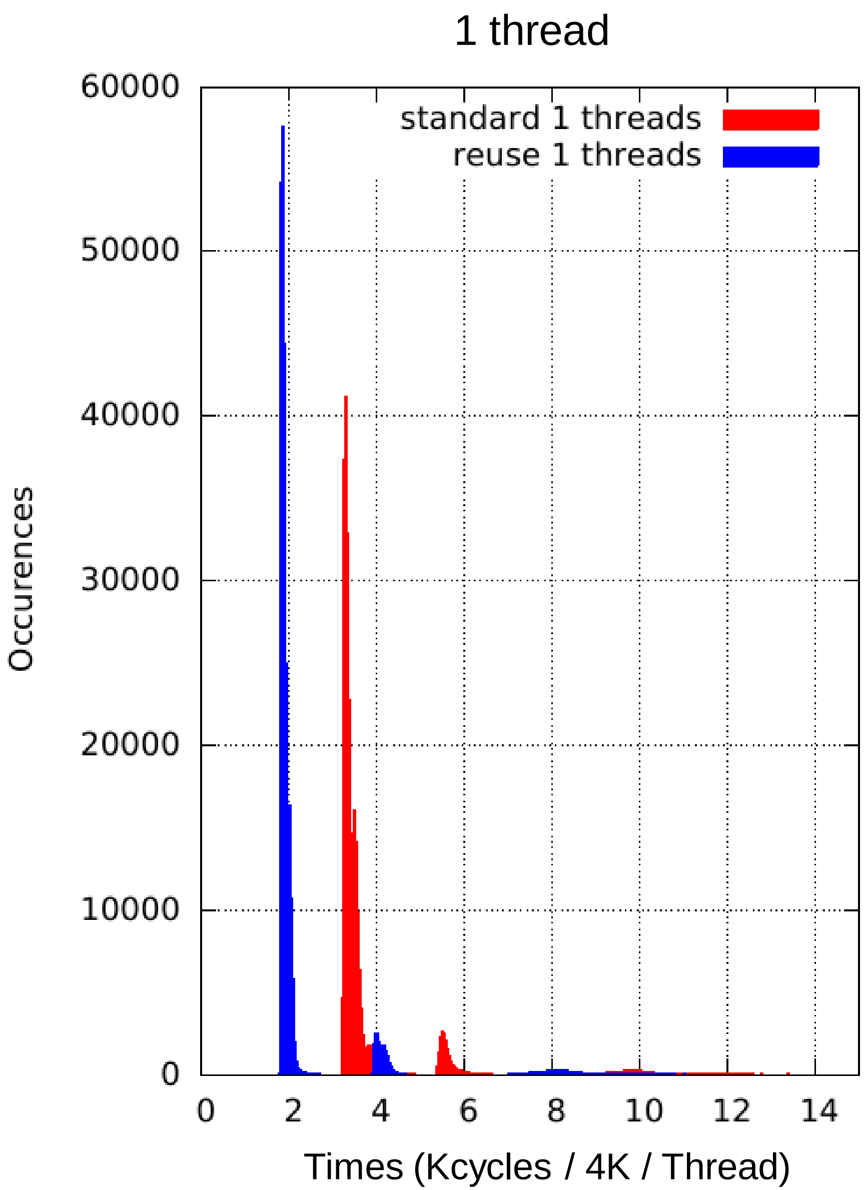
- `mmap` reserves memory regions (in Linux kernel, VMA : Virtual Memory Area).
- Regions are initially not provisioned in pages.
- Pages are really mapped on first access (first touch).

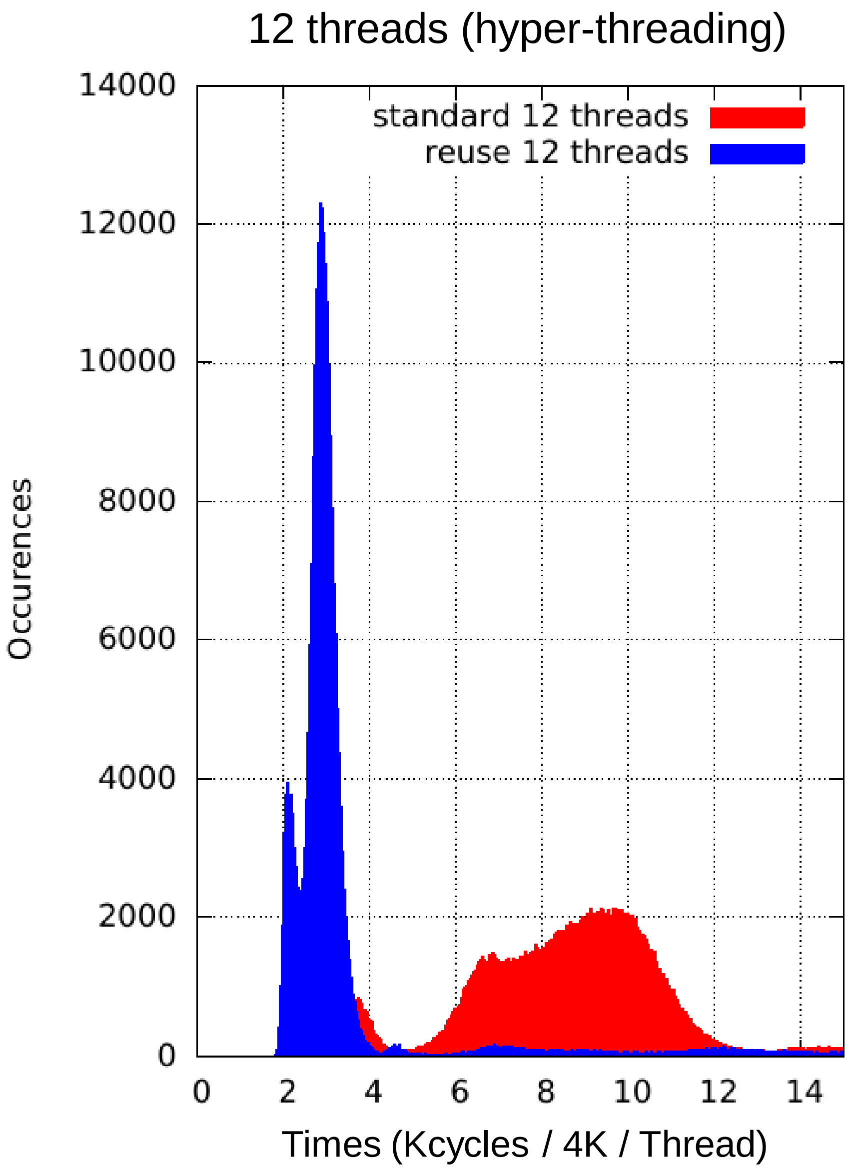


HUGE PAGES SCALABILITY

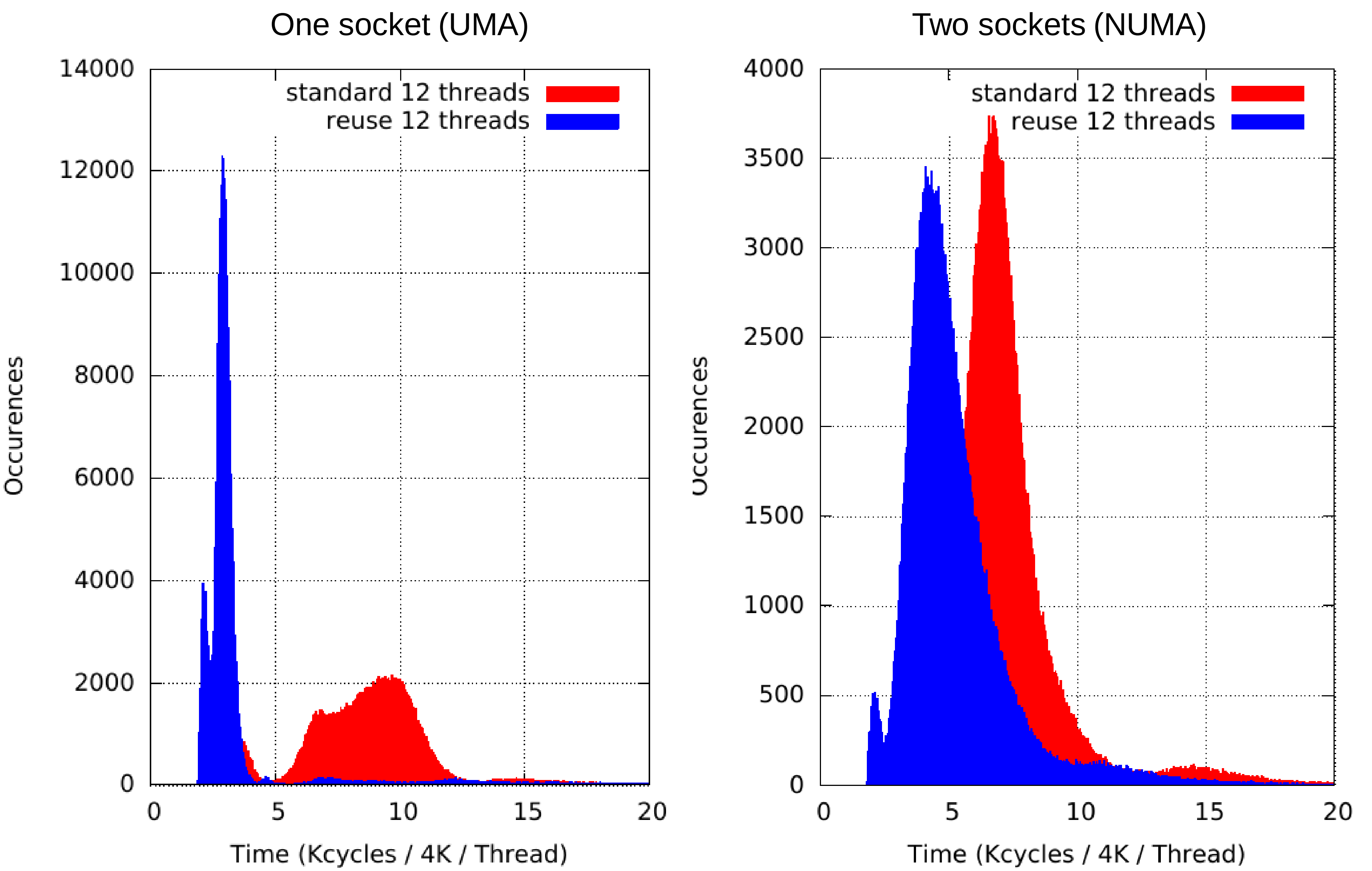


IMPROVEMENT OF FAULTS ON 6 CORE WESTMERE



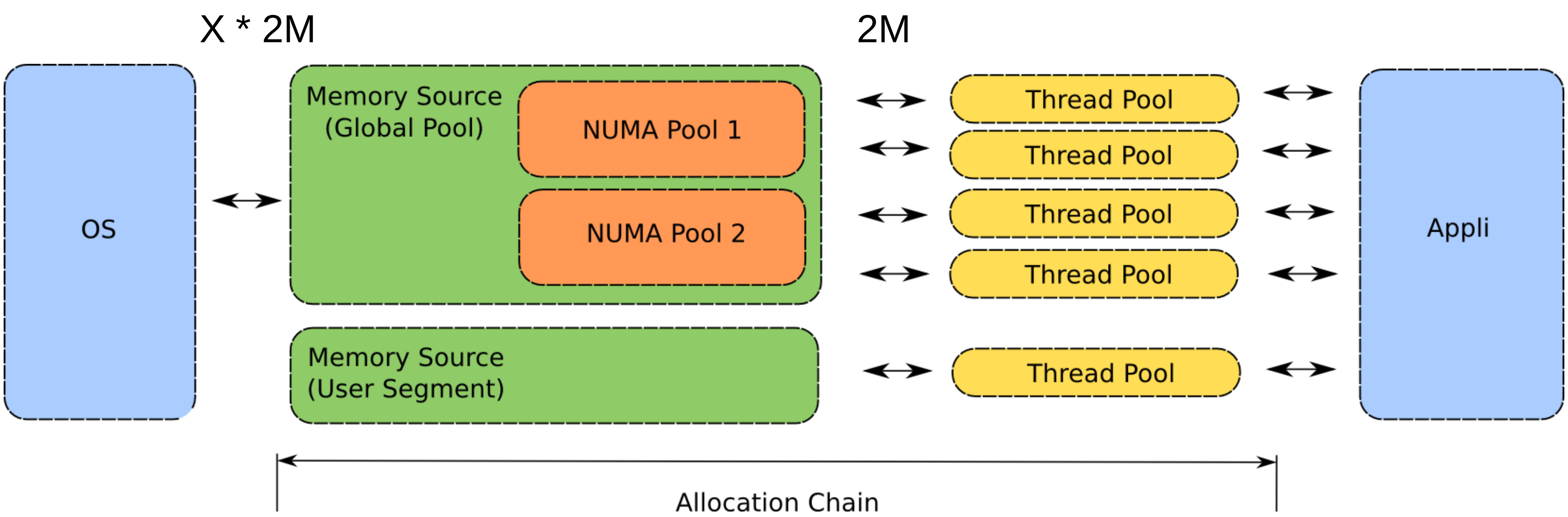


USING TWO SOCKETS (NUMA)



REDUCE MEMORY EXCHANGES WITH THE OS

- Introduce a **pool between application** and **OS**
- **Reuse large segments in memory allocator**.
- Require **explicit NUMA** support in whole allocation chain to be efficient.
- **Current allocators** do reuse for small segments, but **not for large** (> 1MB).



SCALABILITY ISSUE

- Page faults are **not scalable** over **threads**
- Improve applications ? (not trivial for large one)
- User-space memory pools ? (increase memory consumption)
- **Improve the OS ?**

CLEAR PAGE COSTS

- Page fault cost in mean : **~3400** cycles
- Clear page cost in mean : **~1400** cycles
- On page fault, **40%** of the time is due to **zero filling** !
- Clear page function is called **between** two read **locks**
- It prevent parallel usage of `mmap/munmap/brk`.
- On huge pages it has to clear **2MB** instead of **4KB** per page fault.