A Study of Garbage Collector Scalability on Multicores

Lokesh Gidra, Gaël Thomas, Julien Sopena and Marc Shapiro

INRIA/University of Paris 6

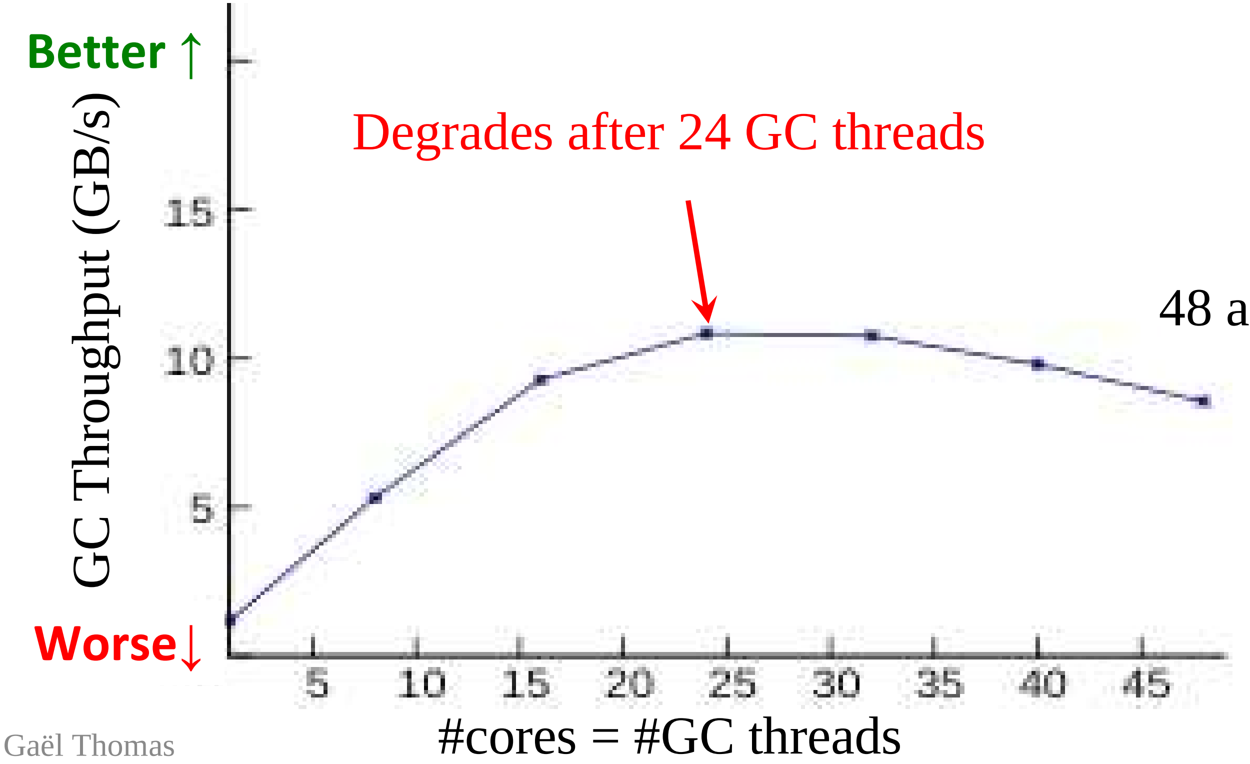
ASPLOS – March 19th 2013

Garbage collection on multicore hardware

14/20 most popular languages have GC

but they don't scale on multicore hardware

Parallel Scavenge/HotSpot scalability on a 48-core machines



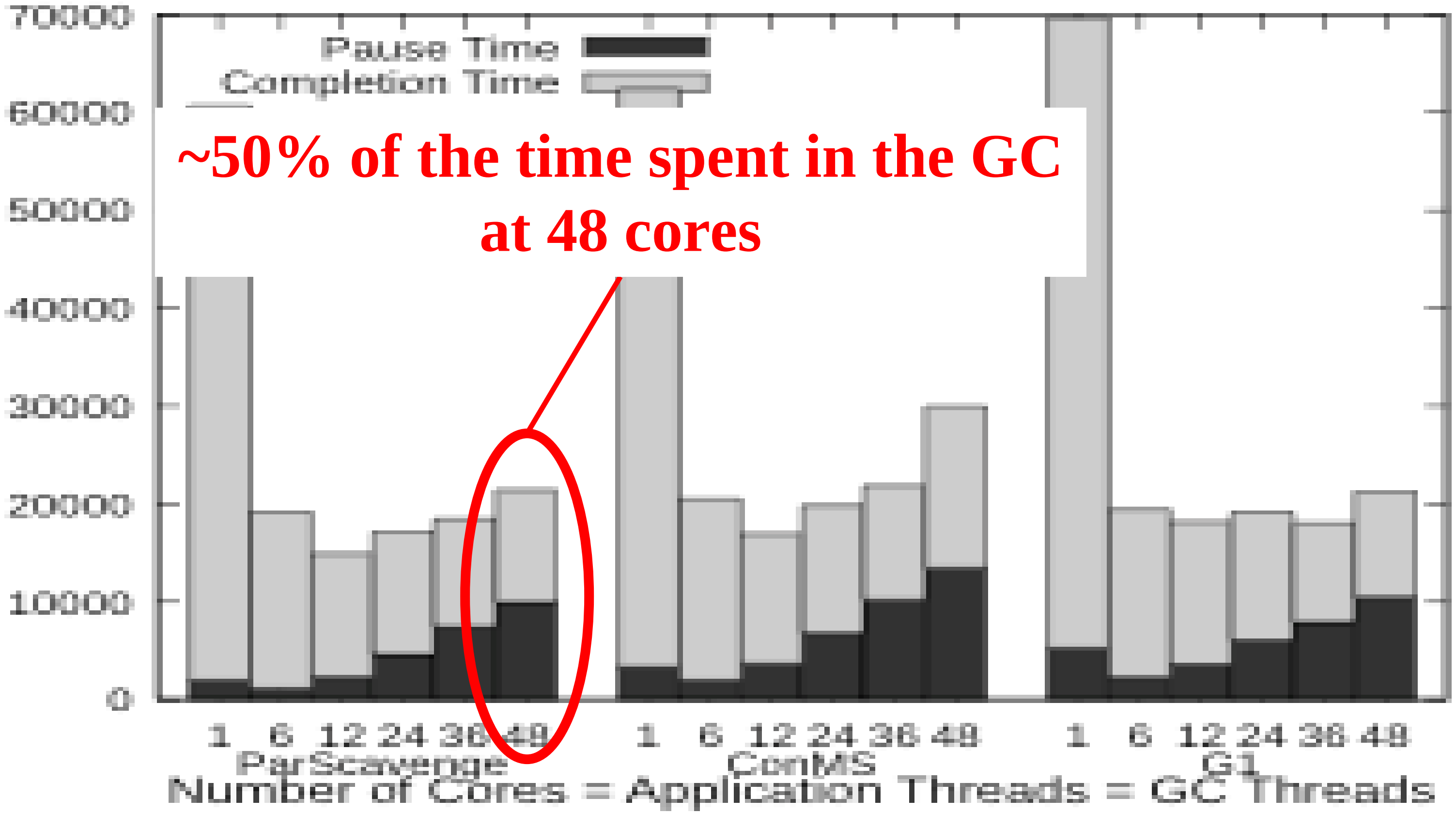
Degrades after 24 GC threads

SpecJBB2005 with
48 application threads/3.5GB

Scalability of GC is a bottleneck

By adding new cores, application creates more garbage per time unit

And without GC scalability, the time spent in GC increases



~50% of the time spent in the GC at 48 cores

Lusearch
[PLOS'11]

Where is the problem?

Probably not related to GC design:

the problem exists in ALL the GCs of HotSpot 7

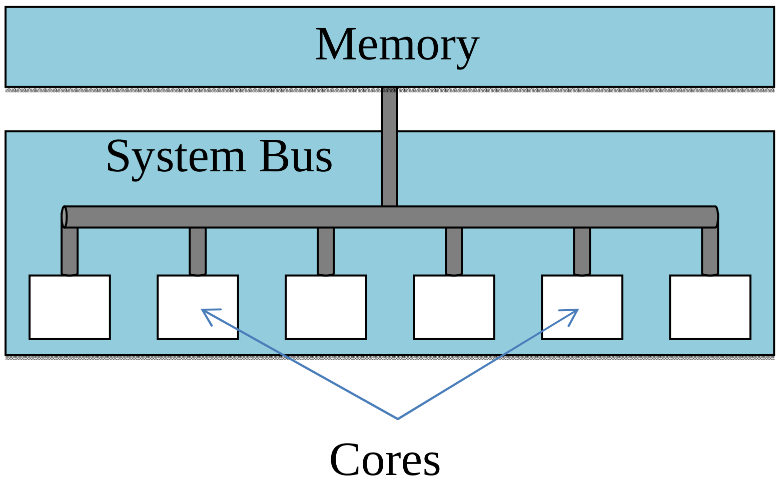
(both stop-the-world and concurrent GCs)

What has really changed:

Multicores are distributed architectures, not centralized architectures

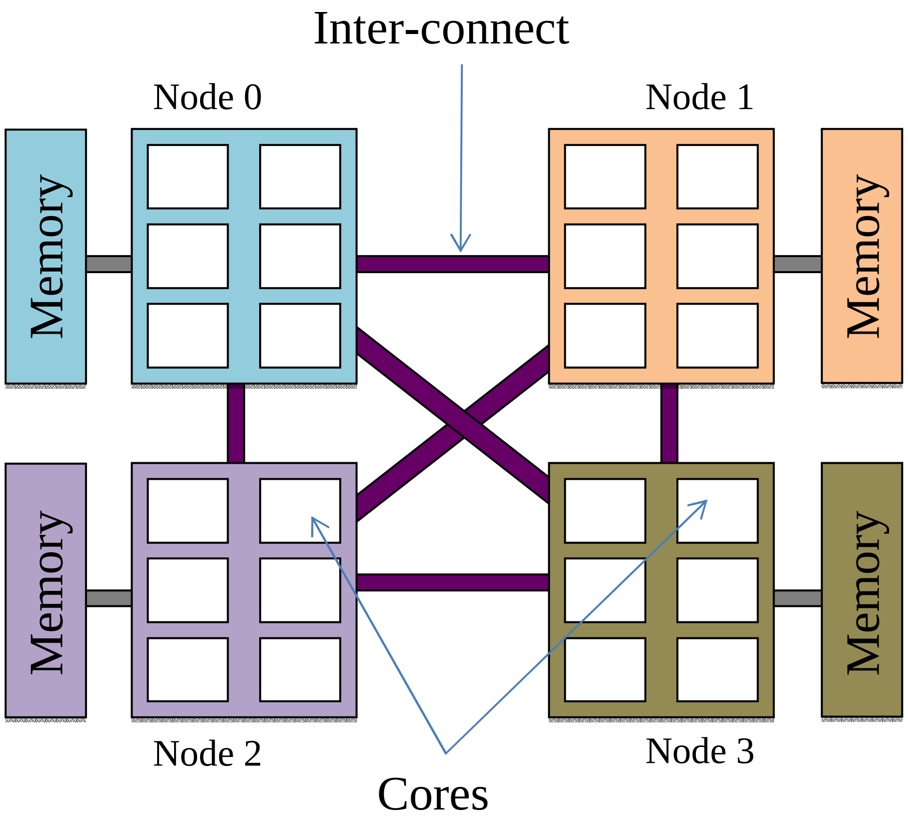
From centralized architectures to distributed ones

A few years ago…



Uniform memory access machines

Now…

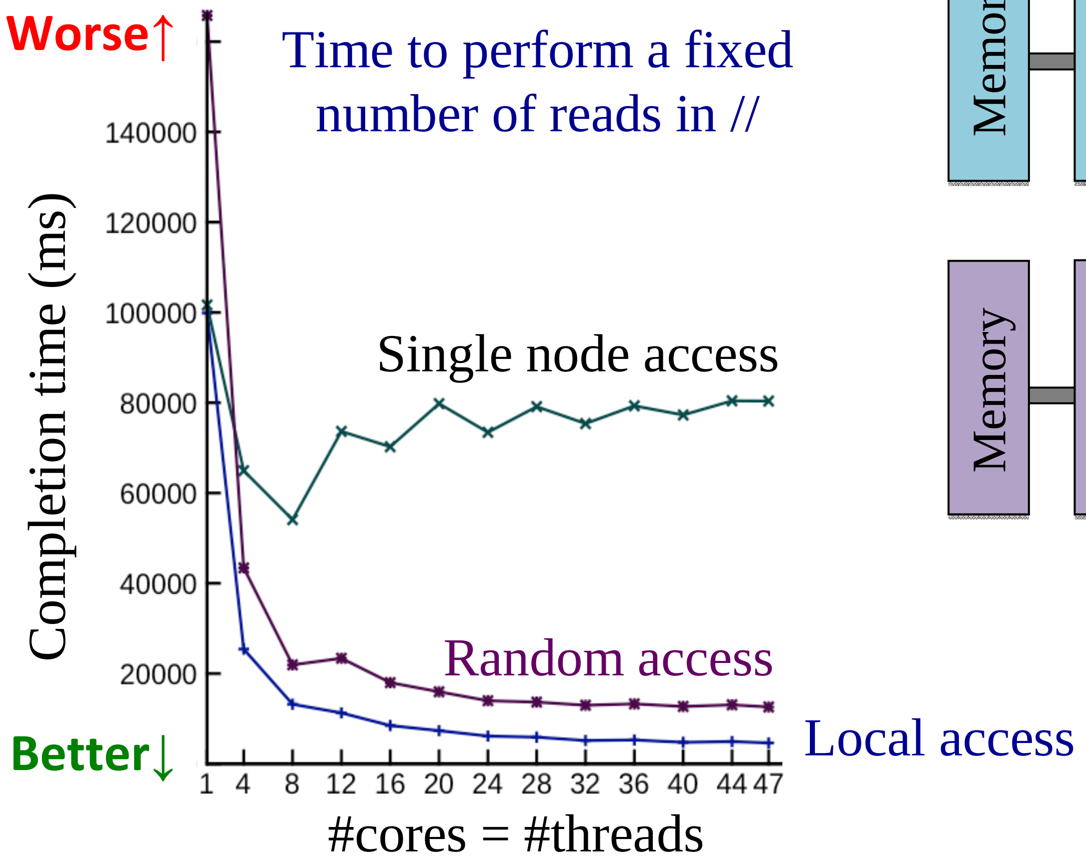


Non-uniform memory access machines

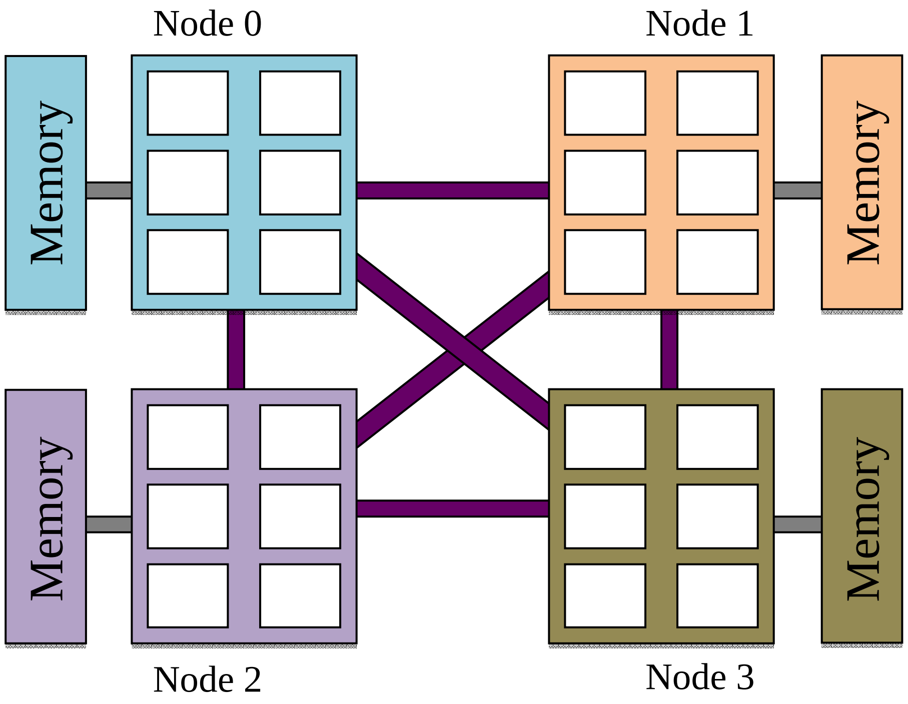
From centralized architectures to distributed ones

Our machine: AMD Magny-Cours with **8 nodes** and **48 cores**

- ✓ 12 GB per node
- ✓ 6 cores per node



Time to perform a fixed number of reads in //

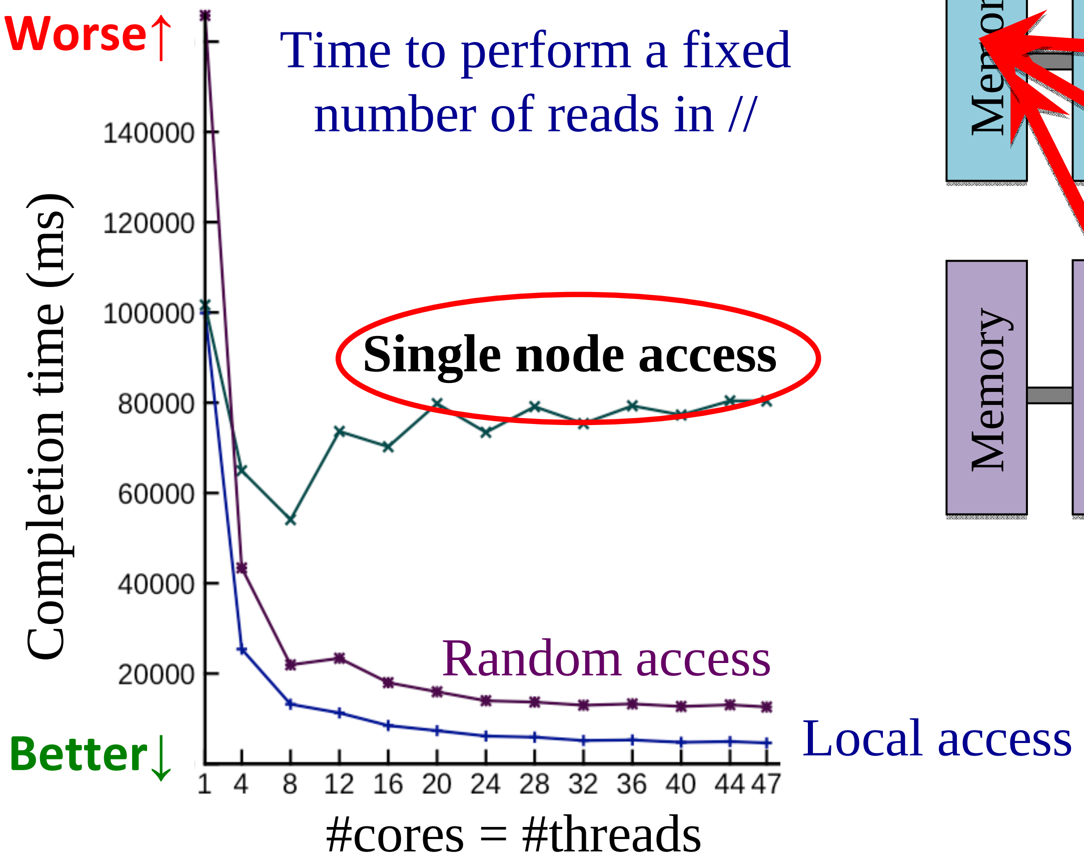


Local access: ~ 200 cycles
Remote access: ~300 cycles

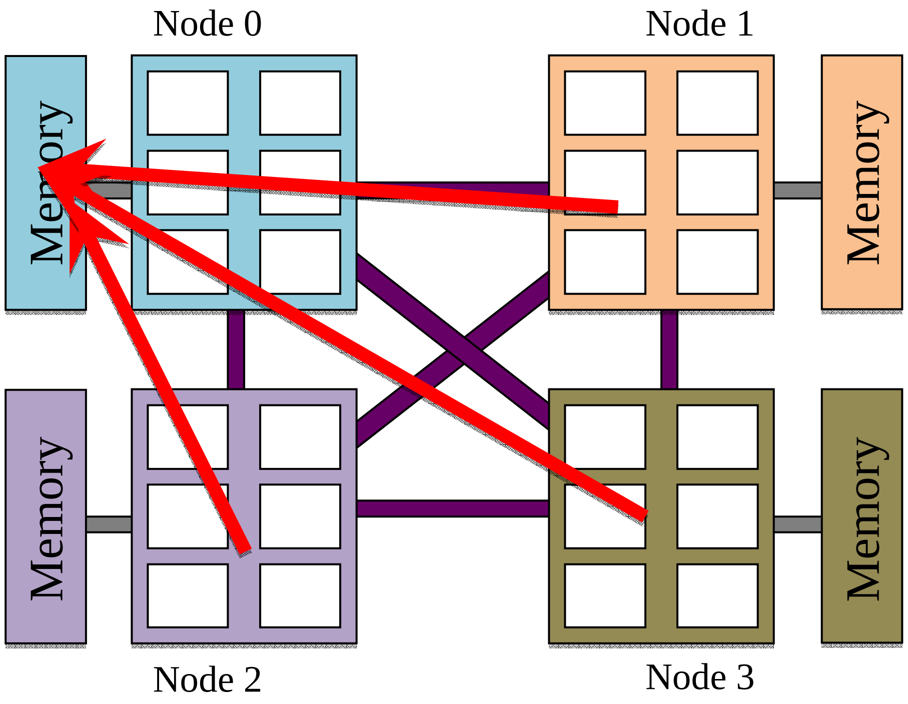
From centralized architectures to distributed ones

Our machine: AMD Magny-Cours with **8 nodes** and **48 cores**

- ✓ 12 GB per node
- ✓ 6 cores per node



Time to perform a fixed number of reads in //

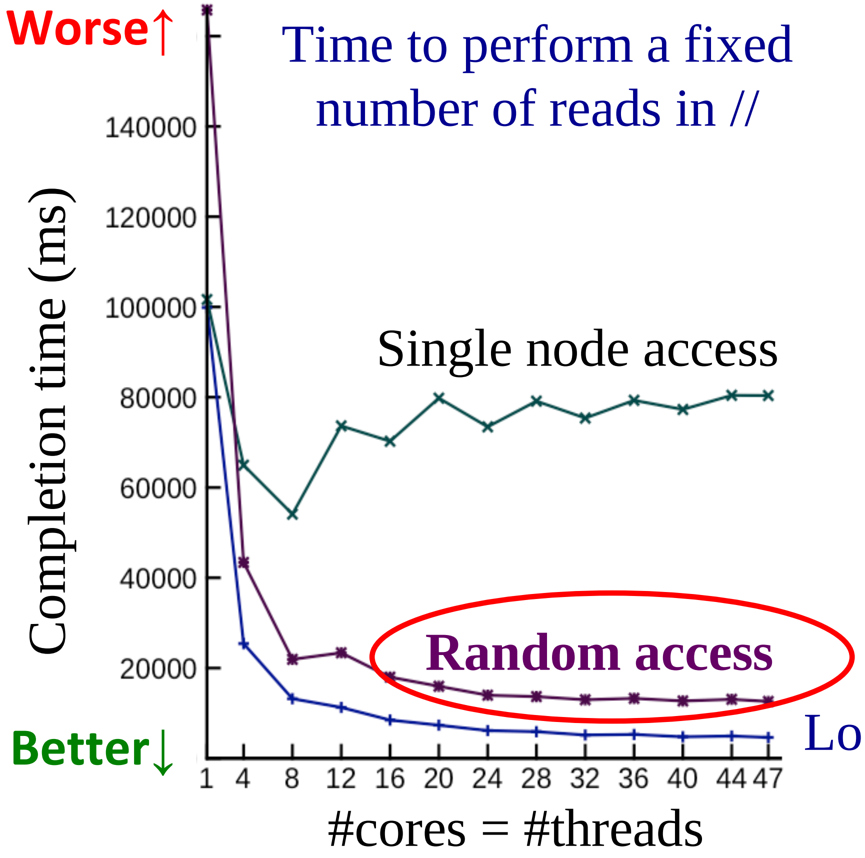


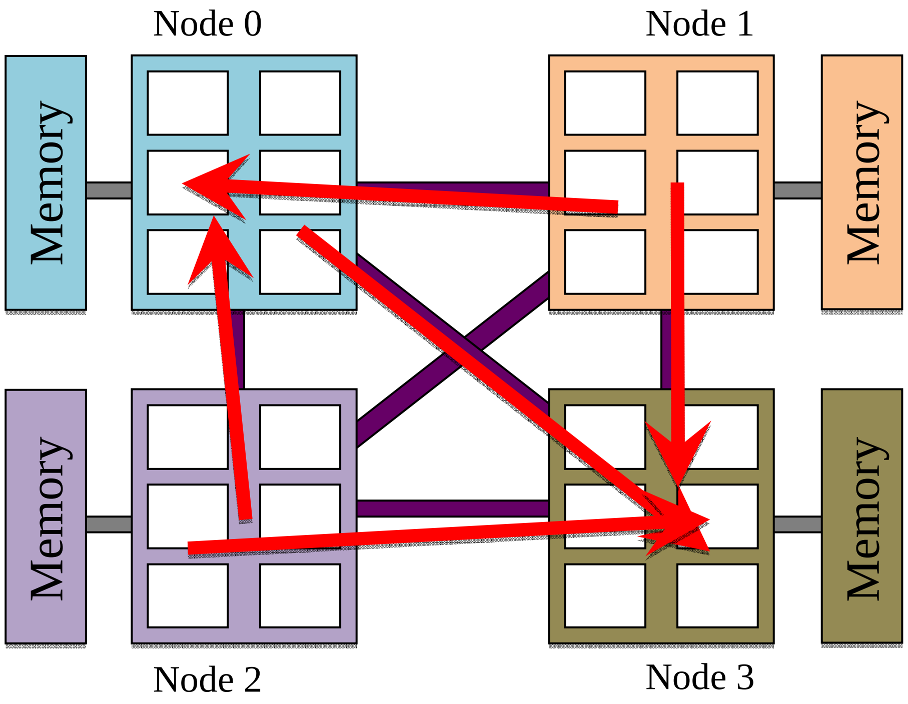
Local access: ~ 200 cycles
Remote access: ~300 cycles

From centralized architectures to distributed ones

Our machine: AMD Magny-Cours with **8 nodes** and **48 cores**

- ✓ 12 GB per node
- ✓ 6 cores per node



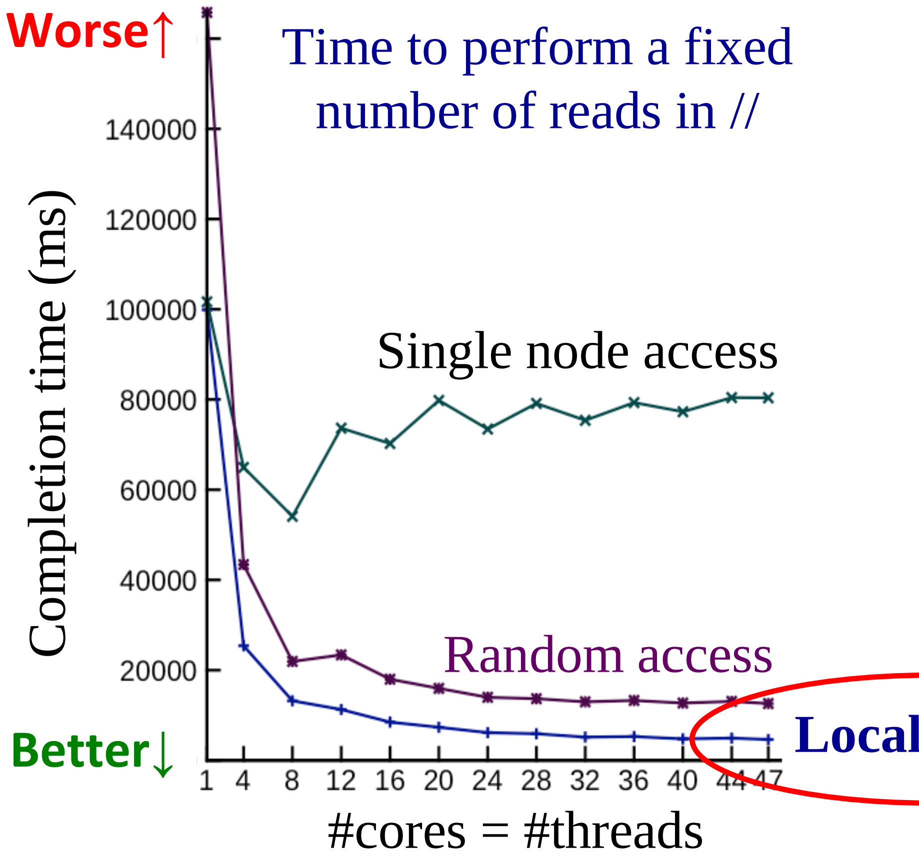


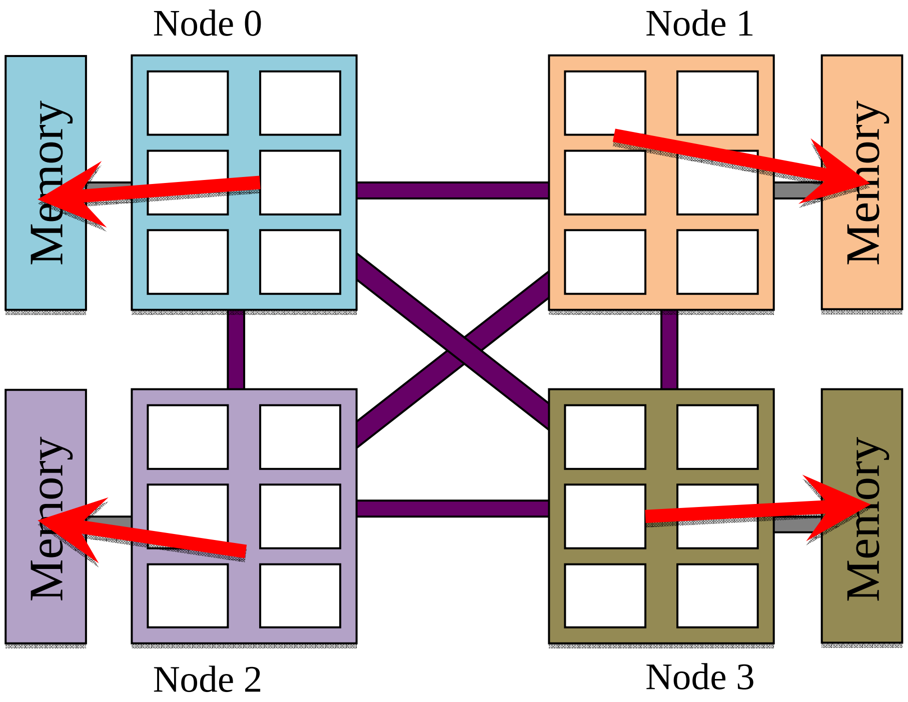
Local access: ~ 200 cycles
Remote access: ~300 cycles

From centralized architectures to distributed ones

Our machine: AMD Magny-Cours with **8 nodes** and **48 cores**

- ✓ 12 GB per node
- ✓ 6 cores per node


Time to perform a fixed number of reads in //

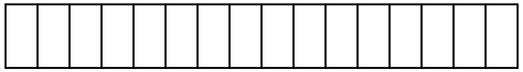


Local access: ~ 200 cycles
Remote access: ~300 cycles

Parallel Scavenge Heap Space

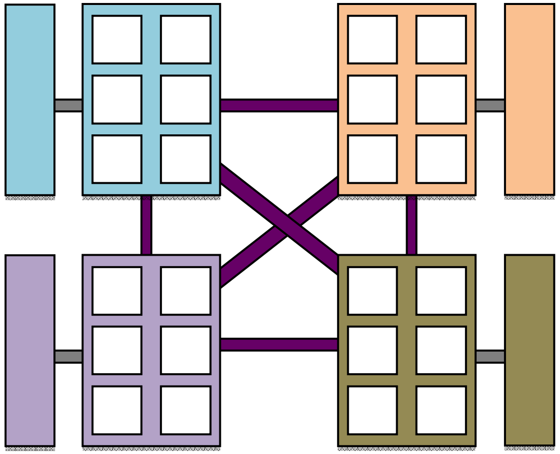
Parallel Scavenge

First-touch allocation policy



Virtual address space

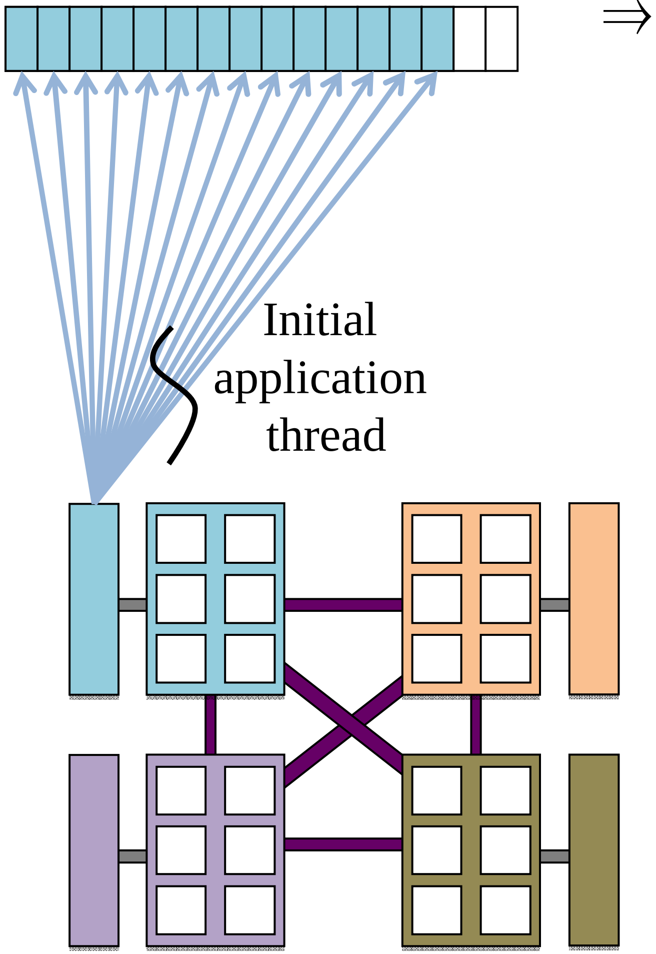
Kernel’s lazy first-touch page allocation policy



Parallel Scavenge Heap Space

Parallel Scavenge

First-touch allocation policy

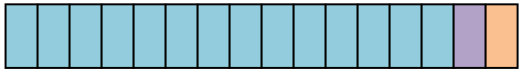


Kernel’s lazy first-touch page allocation policy
⇒ initial sequential phase maps most pages on first node

Parallel Scavenge Heap Space

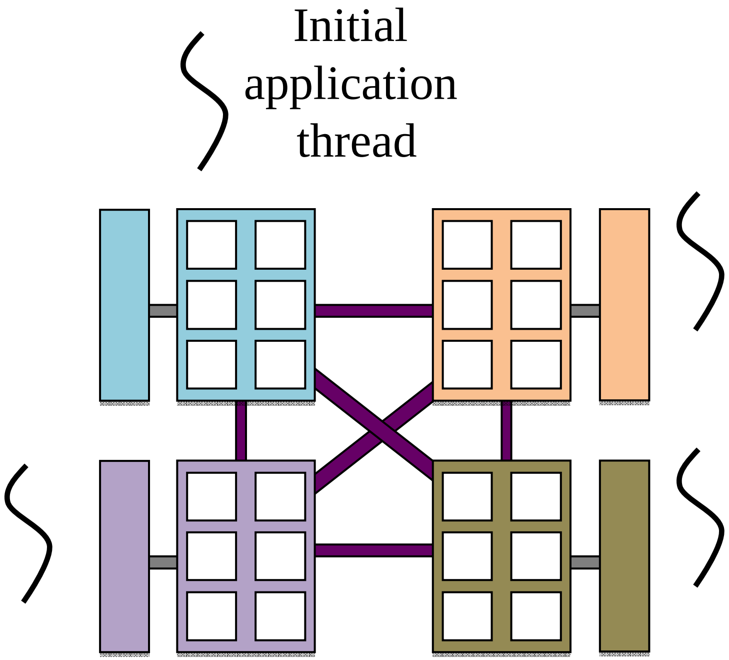
Parallel Scavenge

First-touch allocation policy



Kernel’s lazy first-touch page allocation policy
⇒ initial sequential phase maps most pages on its node

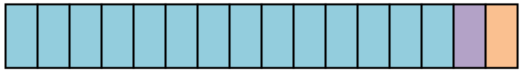
But during the whole execution,
the mapping remains on a single node
(virtual space reused by the GC)



Parallel Scavenge Heap Space

Parallel Scavenge

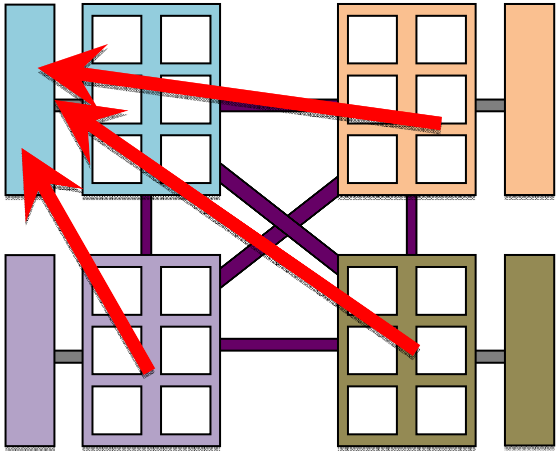
First-touch allocation policy

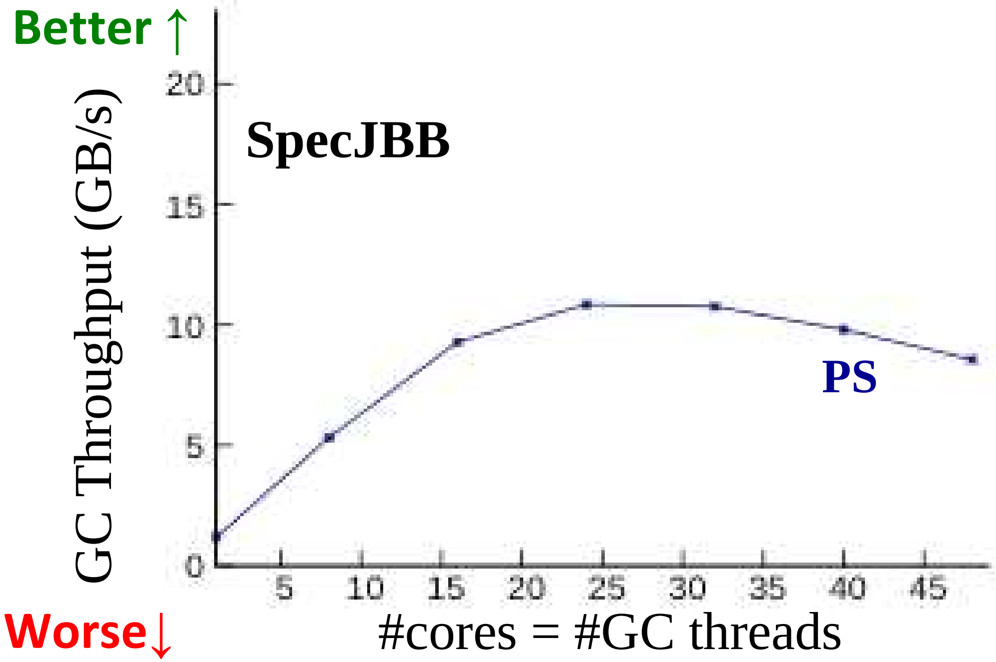


Bad balance

Bad locality

95% on a single node

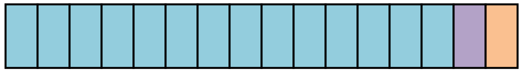




NUMA-aware heap layouts

Parallel Scavenge

First-touch allocation policy



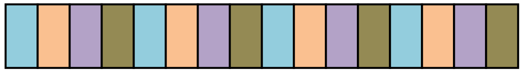
Bad balance

Bad locality

95% on a single node

Interleaved

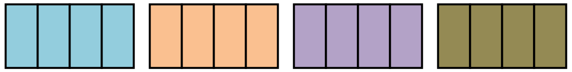
Round-robin allocation policy



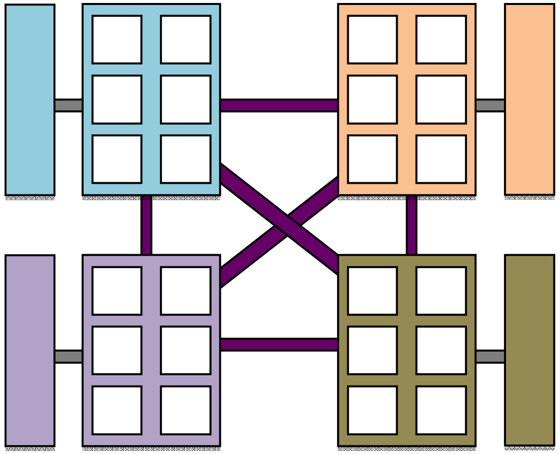
Targets balance

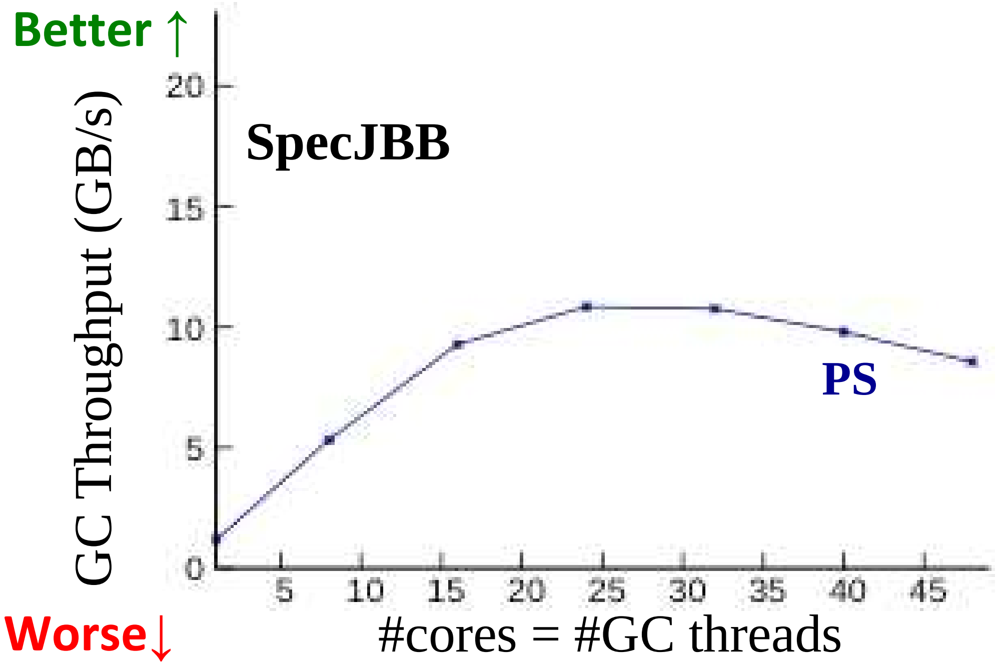
Fragmented

Node local object allocation and copy



Targets locality

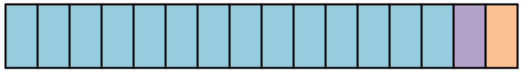




Interleaved heap layout analysis

Parallel Scavenge

First-touch allocation policy



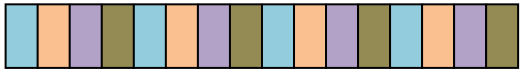
Bad balance

Bad locality

95% on a single node

Interleaved

Round-robin allocation policy



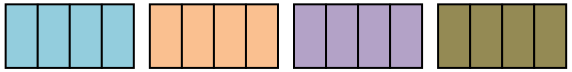
Perfect balance

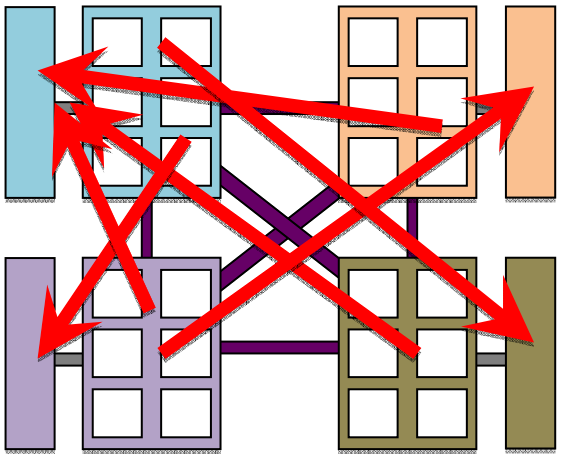
Bad locality

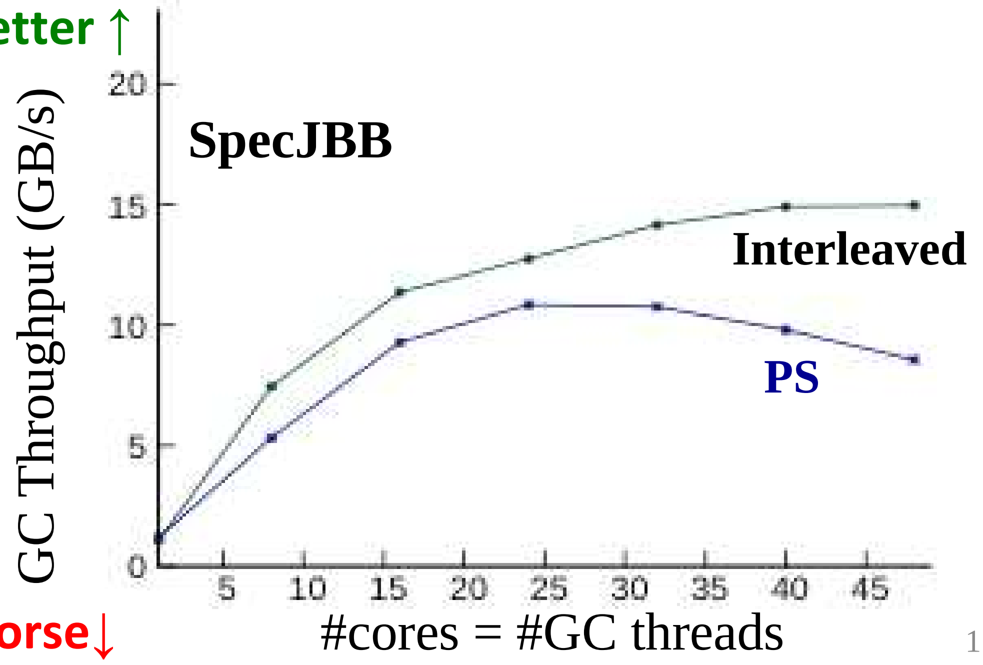
7/8 remote accesses

Fragmented

Node local object allocation and copy







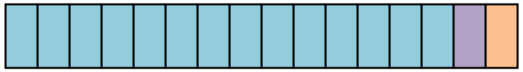
Better ↑

Worse↓

Fragmented heap layout analysis

Parallel Scavenge

First-touch allocation policy



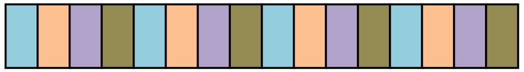
Bad balance

Bad locality

95% on a single node

Interleaved

Round-robin allocation policy



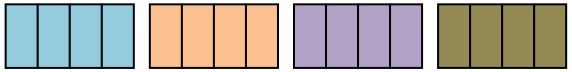
Perfect balance

Bad locality

7/8 remote accesses

Fragmented

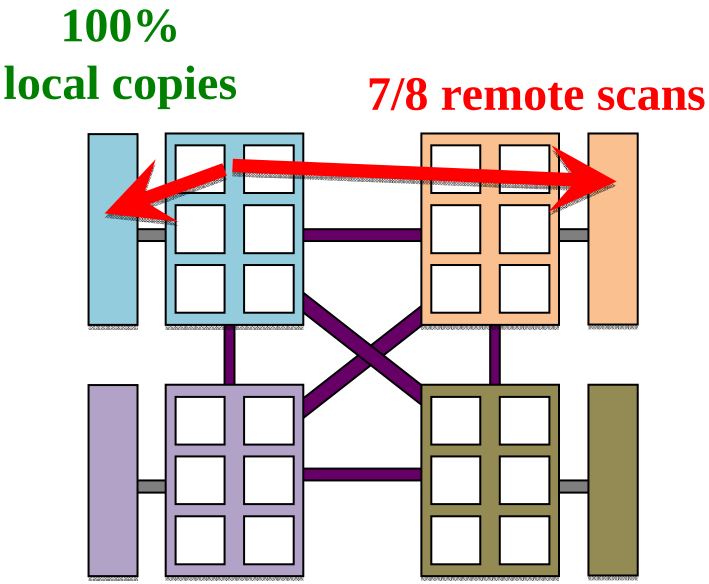
Node local object allocation and copy

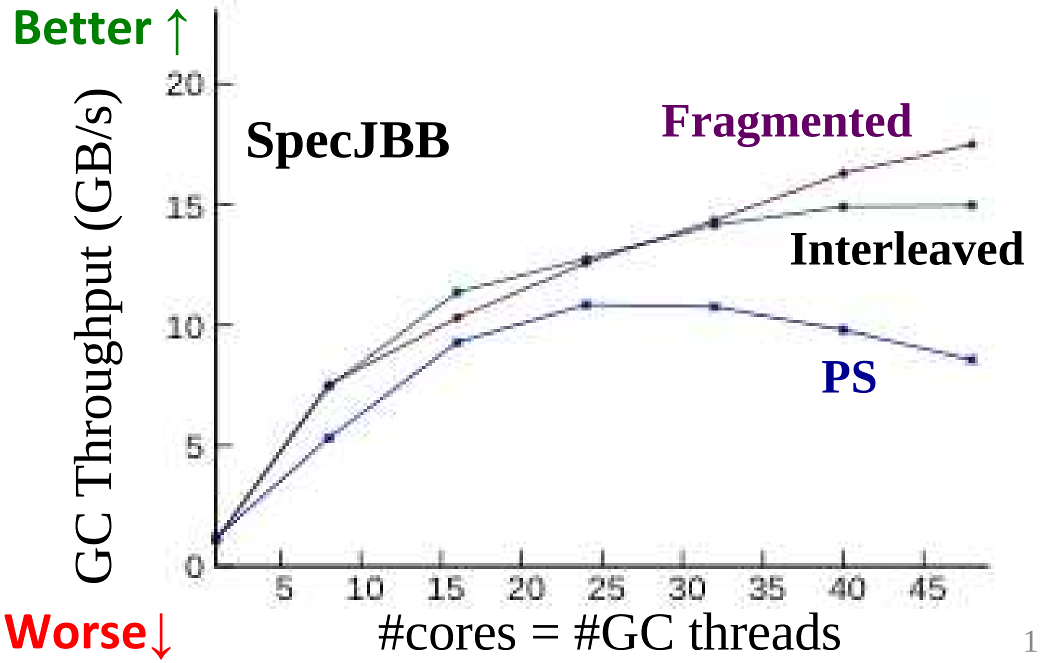


Good balance

Average locality

Bad balance if a single thread allocates for the others

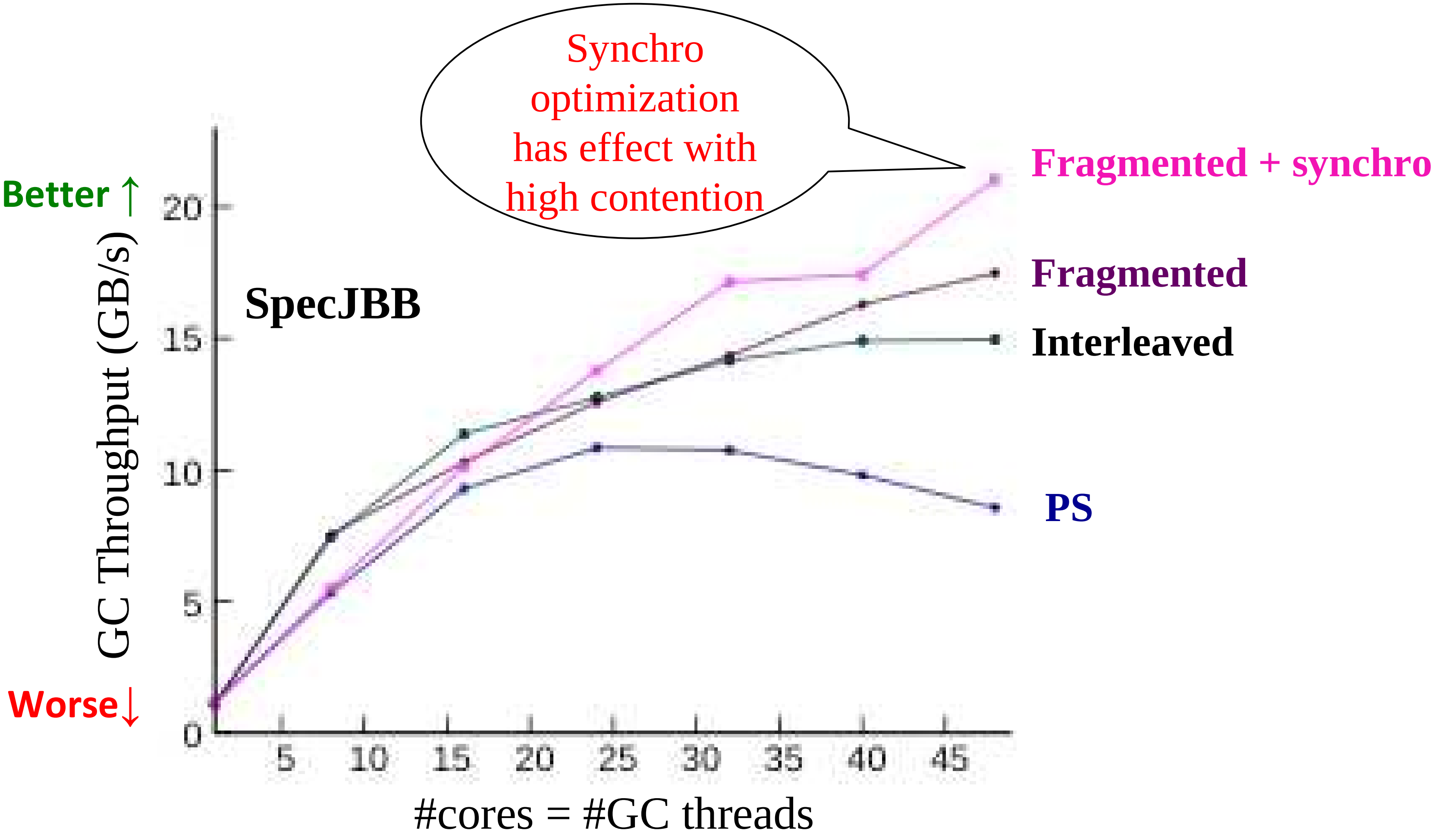




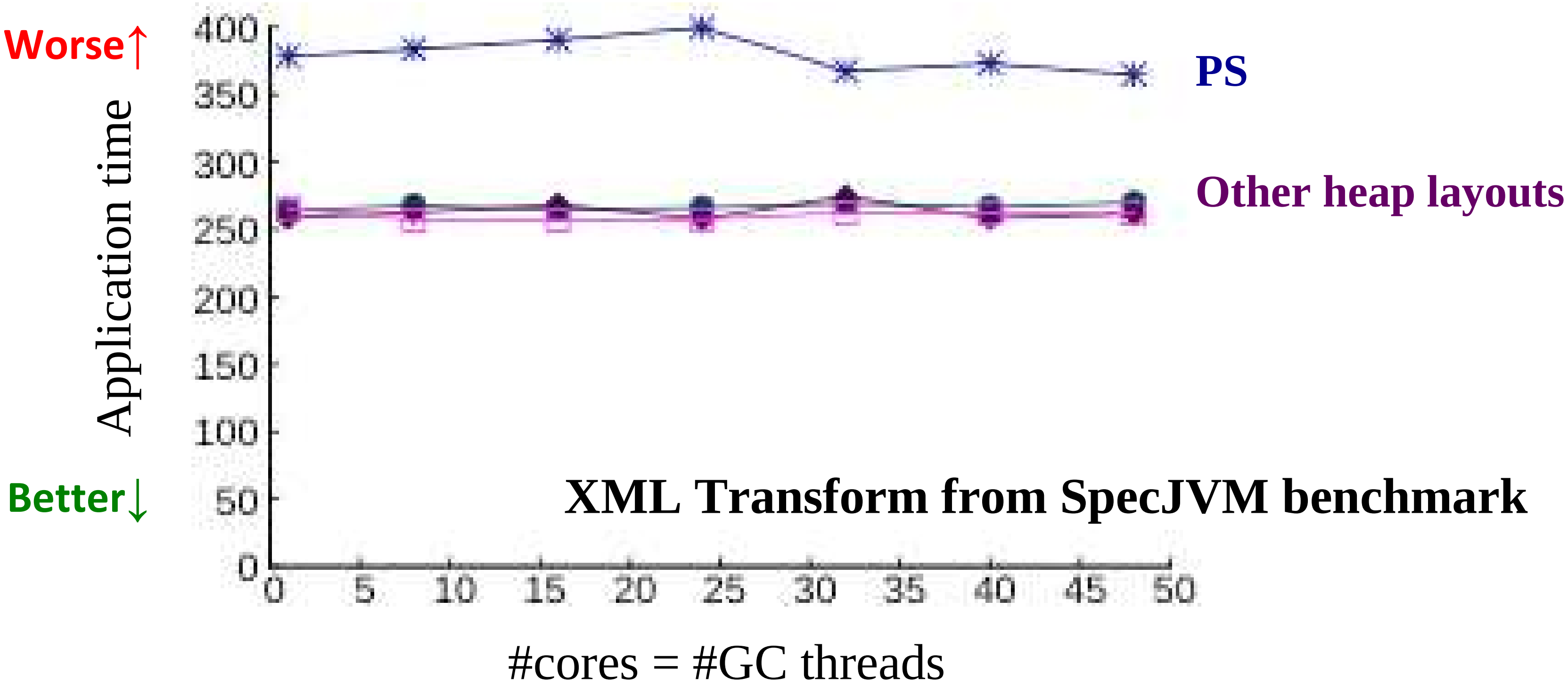
Synchronization optimizations

Remove a barrier between the GC phases

Replace the queue of GC tasks with a lock-free one



Effect of Optimizations on the App (GC excluded)

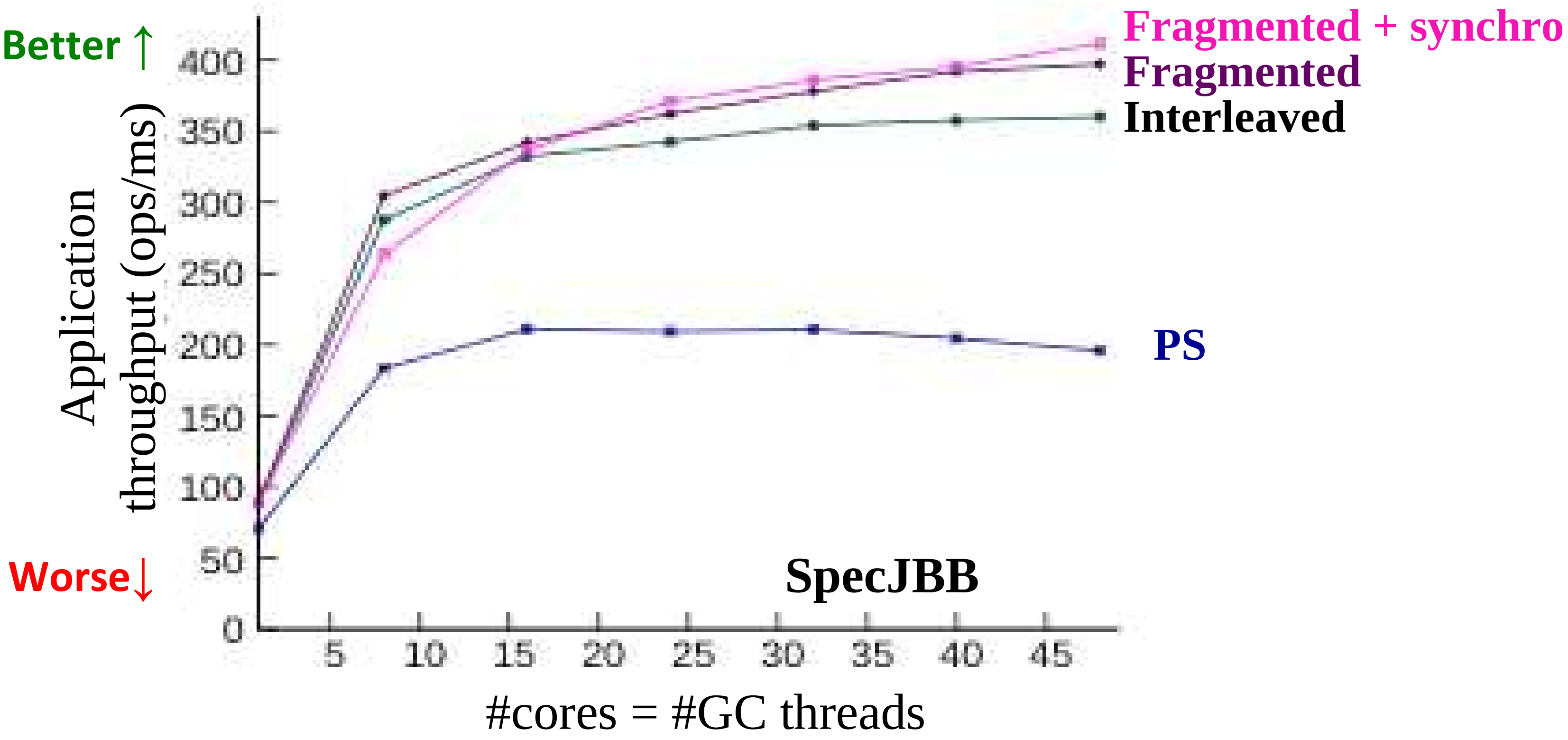


A good balance improves a lot application time

Locality has only a marginal effect on application

While fragmented space increases locality for application over interleaved space (recently allocated objects are the most accessed)

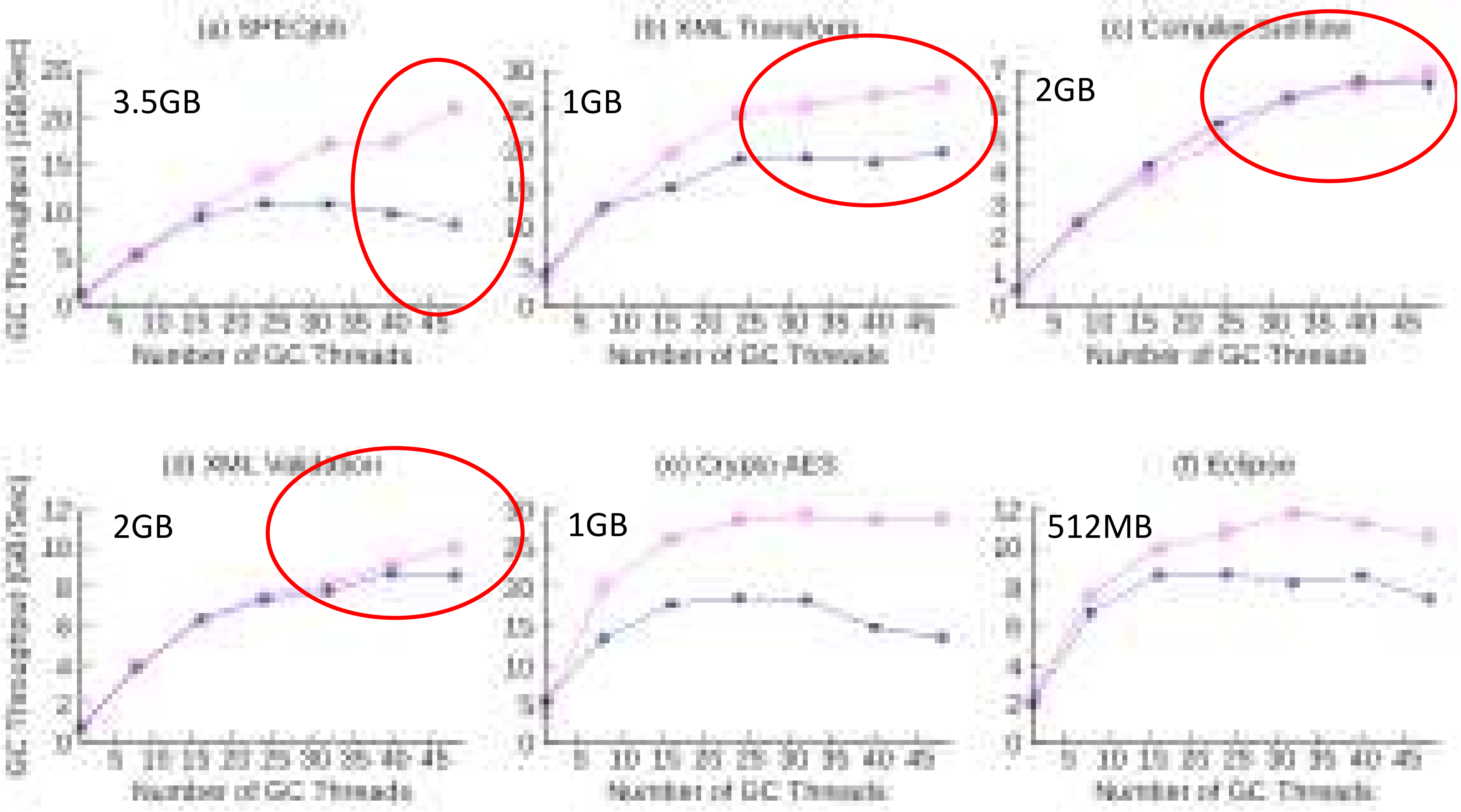
Overall effect (both GC and application)



Optimizations double the app throughput of SPECjbb

Pause time divided in half (105ms to 49ms)

GC scales well with memory-intensive applications



PS —✳—　**Fragmented + synchro** —⊡—

Conclusion

Previous GCs do not scale because they are not NUMA-aware

- Existing mature GCs can scale with standard // programming techniques
- Using NUMA-aware memory layouts should be useful for all GCs (concurrent GCs included)

Most important NUMA effects

1. Balancing memory access
2. Memory locality only helps at high core count

Conclusion

Previous GCs do not scale because they are not NUMA-aware

- Existing mature GCs can scale with standard // programming techniques
- Using NUMA-aware memory layouts should be useful for all GCs (concurrent GCs included)

Most important NUMA effects

1. Balancing memory access
2. Memory locality only helps at high core count

Thank You ☺

Issues in the original fragmented space of hotspot

Fragmented space of hotspot was degrading performance

- ✓ 98.4 GB/s with baseline Parallel Scavenge
- ✓ Hotspot’s fragmented space performs degrades GC performance by 33% (63.5 GB/s)

Issues in the original fragmented space

- ✓ Collection triggered when a single fragment is full
  ⇒ 325 collections instead of 177
- ✓ Resizing of spaces implies a lot of system calls
  ⇒ 20% of the GC time spent in resizing

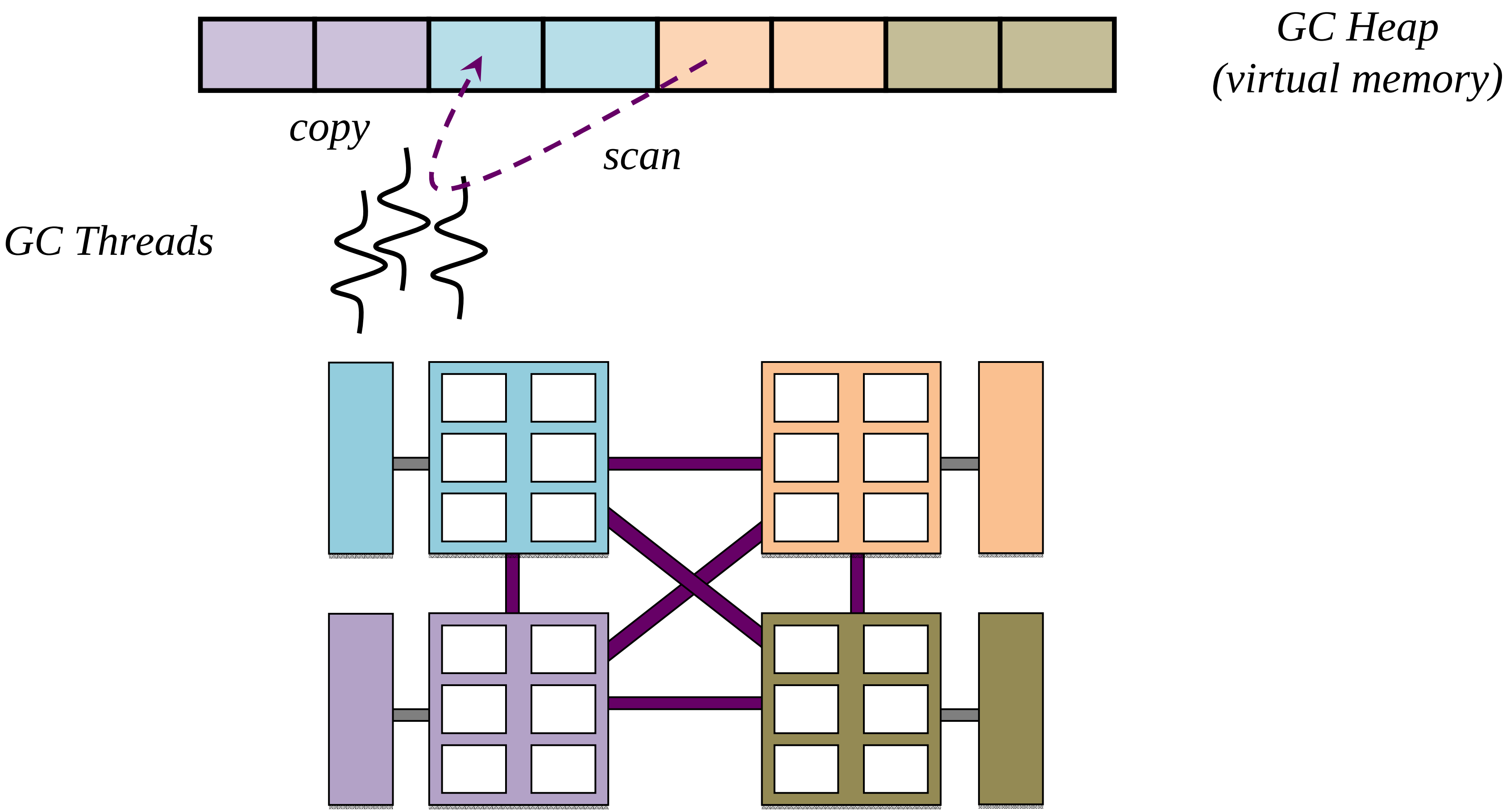
Solutions proposed in our work

- ✓ Virtually, each fragment is an entire space to avoid early collection
- ✓ Pre-allocate and pre-map the maximal heap size to avoid system calls

Fragmented Space: node-local allocation

Good **locality** for both the **GC** *(copies are node-local)*

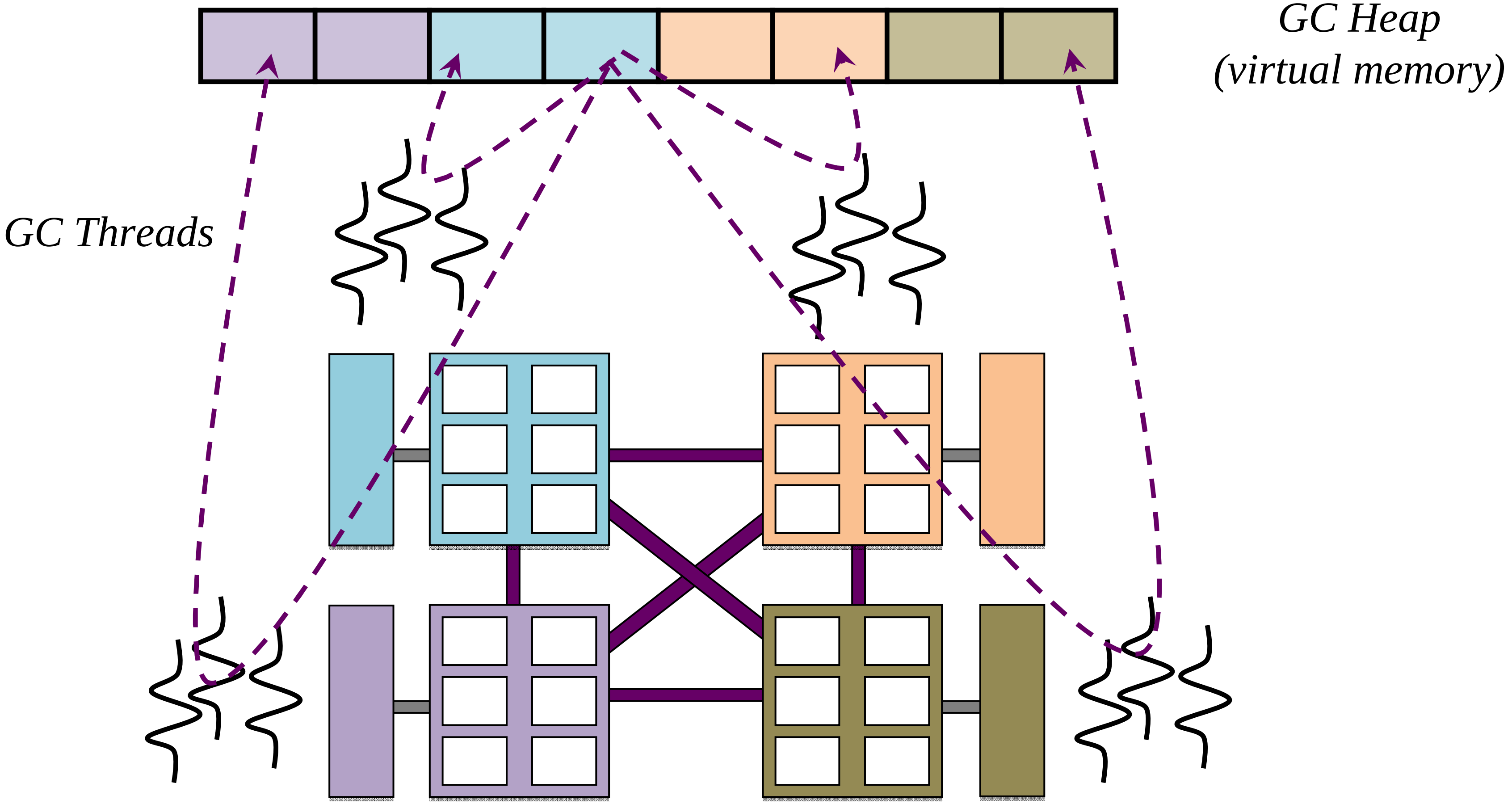
and the **application** *(recently allocated objects are the most used)*



Fragmented Space: node-local allocation

Good **balance** for both the **GC** *(copies are balanced among the nodes)*

and the **application** *(objects are spread among the nodes after the first collection)*



Evaluated applications

SpecJBB 2005: the most memory-intensive application

- ✓ Simulate an application server
- ✓ Working set: 3.5GB

5 applications from SpecJVM 2008

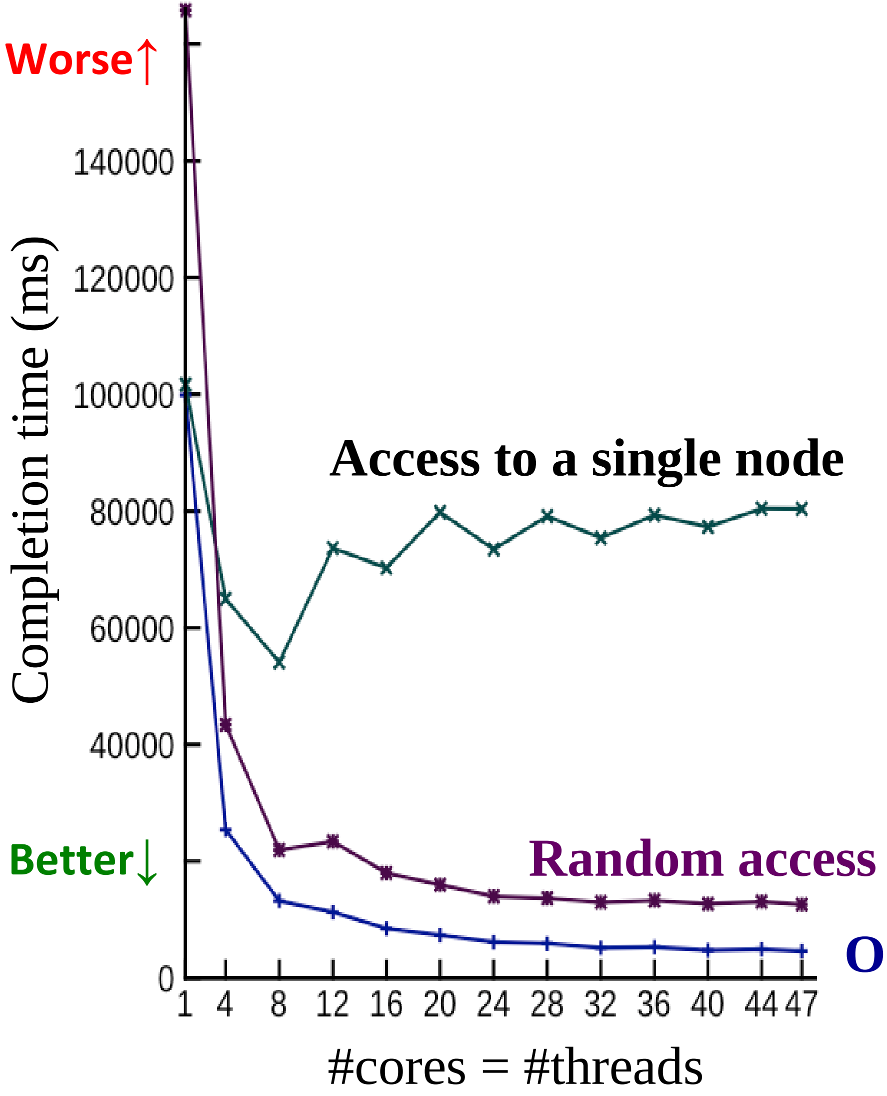
- ✓ Discard applications that do not use memory
- ✓ Working set: between 1 and 2 GB

2 applications from Dacapo 9.12

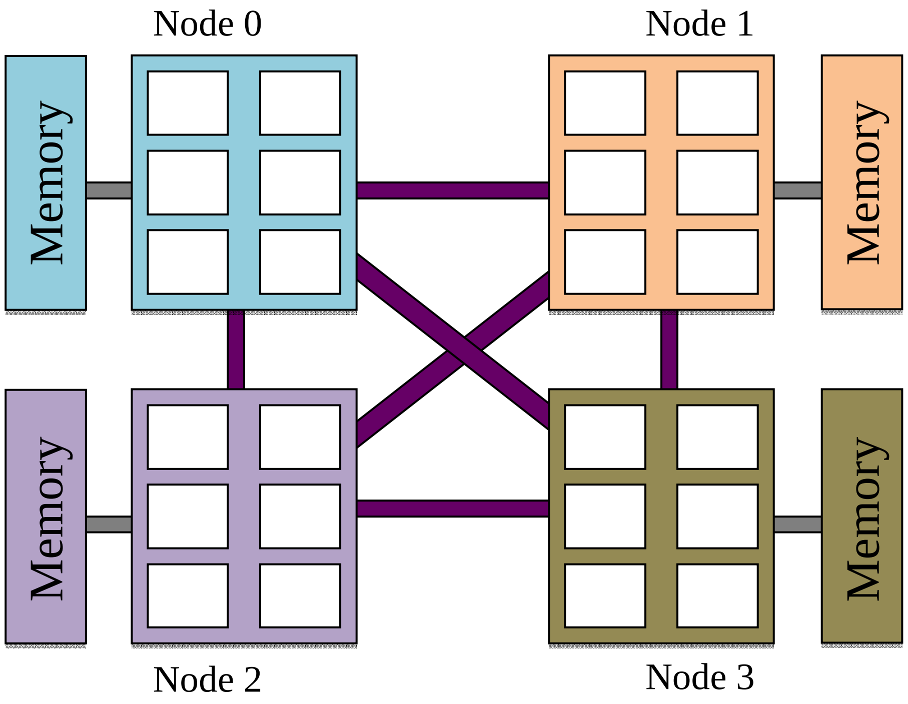
- ✓ Illustrates the effect on non-memory intensive applications
- ✓ Working set: 500MB
- ✓ A GC Thread has only few KB to collect

Memory access micro-benchmark

Measure the time to access a fixed number of memory locations



Only local access



Scalability of GC is a bottleneck

Processor frequency is stagnant since a decade

but not memory size

By adding new cores, application creates more garbage

and without scalability, time spent in GC increases

⇒ Prevents the use of GC for data-intensive applications

(application servers, data-intensive applications, scientific applications…

Where is the problem?

Lack of parallelism?

- ✓ Parallel GCs exist since 30 years
- ✓ Parallel graph traversal is a well-studied problem

Design of parallel Scavenge ill-suited for many cores?

- ✓ The problem exists with ALL the GCs of HotSpot 7
  (both stop-the-world and concurrent)

What has really changed:

Multicores are distributed architectures, not centralized architectures