NumaGiC: A garbage collector for big-data on big NUMA machines

Lokesh Gidra[‡], *Gaël Thomas*[*], *Julien Sopena*[‡], *Marc Shapiro*[‡], *Nhan Nguyen*[♀]

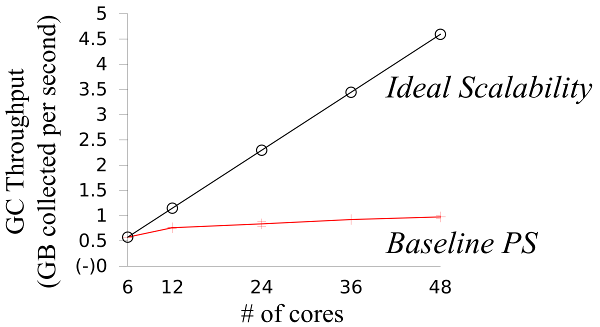
[‡] *LIP6/UPMC-INRIA* [*] *Telecom SudParis* [♀] *Chalmers University*

Motivation

- Data-intensive applications need large machines with plenty of cores and memory

Motivation

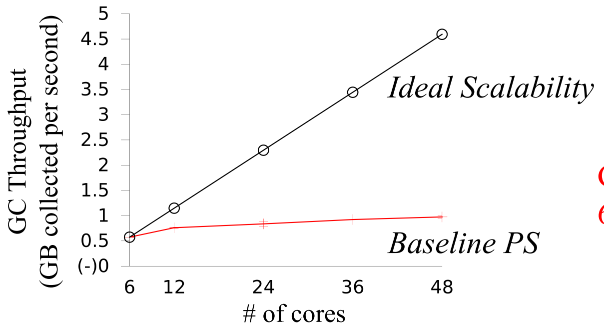
- Data-intensive applications need large machines with plenty of cores and memory
- But, for large heaps, GC is inefficient on such machines



Page rank computation of 100million edge Friendster dataset with Spark on Hotspot/Parallel Scavenge with 40GB on a 48-core machine

Motivation

- Data-intensive applications need large machines with plenty of cores and memory
- But, for large heaps, GC is inefficient on such machines



GC takes roughly 60% of the total time

Page rank computation of 100million edge Friendster dataset with Spark on Hotspot/Parallel Scavenge with 40GB on a 48-core machine

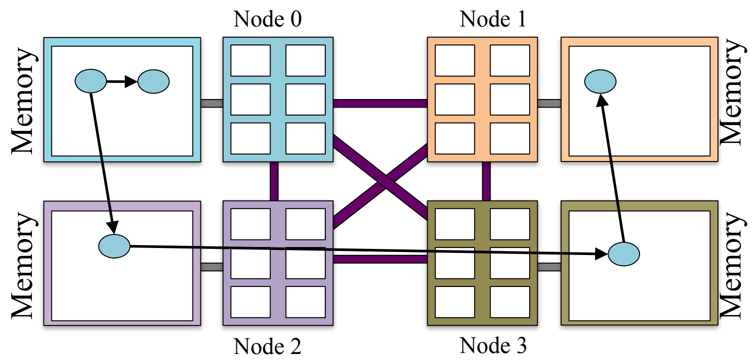
Outline

- Why GC doesn't scale?
- Our Solution: NumaGiC
- Evaluation

GCs don't scale because machines are NUMA

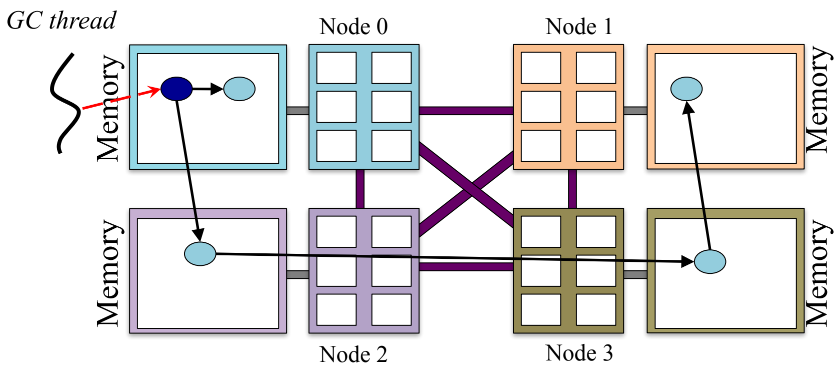
Hardware hides the distributed memory

⇒ application silently creates inter-node references



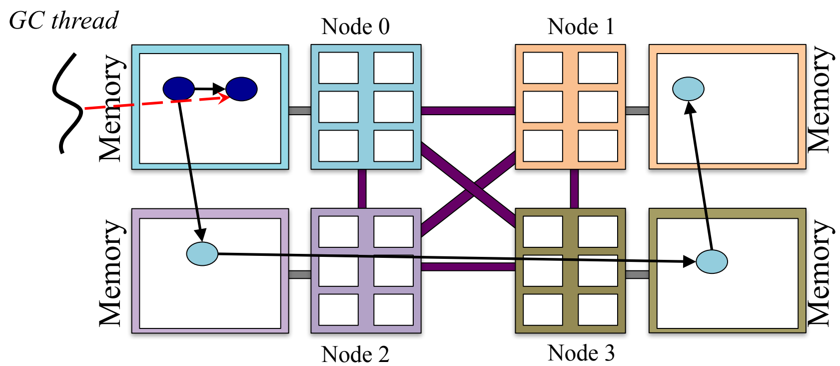
GCs don't scale because machines are NUMA

But memory distribution is also hidden to the GC threads when they traverse the object graph



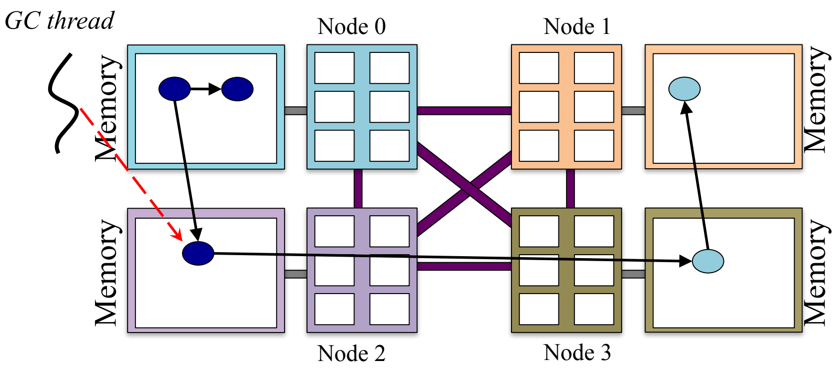
GCs don't scale because machines are NUMA

But memory distribution is also hidden to the GC threads when they traverse the object graph



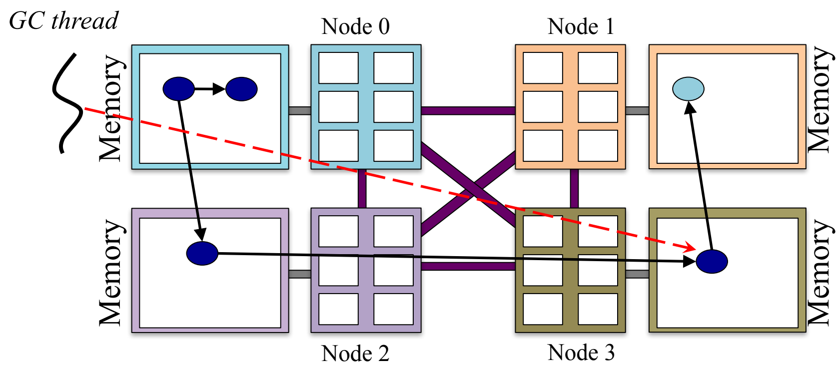
GCs don’t scale because machines are NUMA

A GC thread thus silently traverses remote references



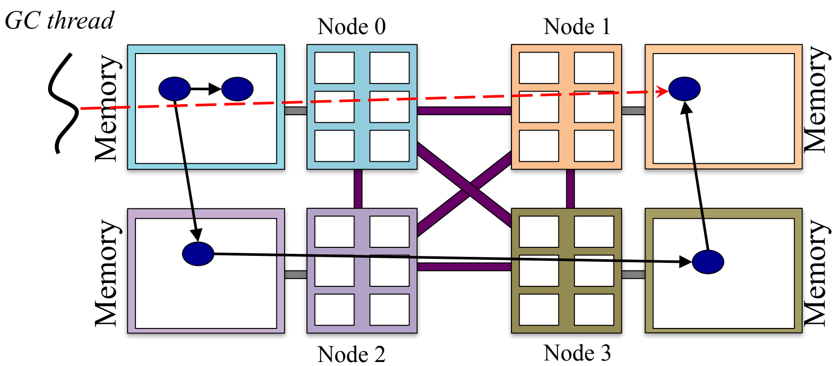
GCs don’t scale because machines are NUMA

A GC thread thus silently traverses remote references
and continues its graph traversal on any node



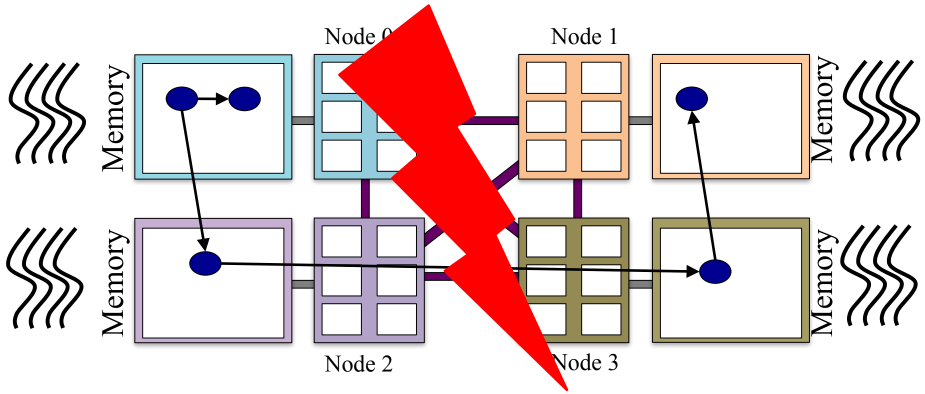
GCs don’t scale because machines are NUMA

A GC thread thus silently traverses remote references
and continues its graph traversal on any node



GCs don’t scale because machines are NUMA

When all GC threads access any memory nodes,
the inter-connect potentially saturates
=> high memory access latency



Outline

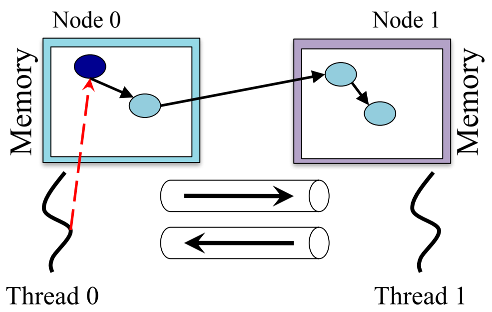
- Why GC doesn't scale?
- Our Solution: NumaGiC
- Evaluation

How can we fix the memory locality issue?

Simply by preventing any remote memory access

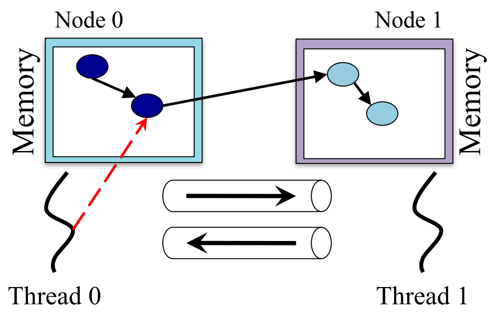
Prevent remote access using messages

Enforces memory access locality
by trading remote memory accesses by messages



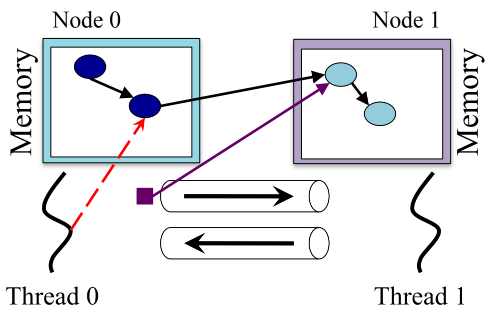
Prevent remote access using messages

Enforces memory access locality
by trading remote memory accesses by messages



Prevent remote access using messages

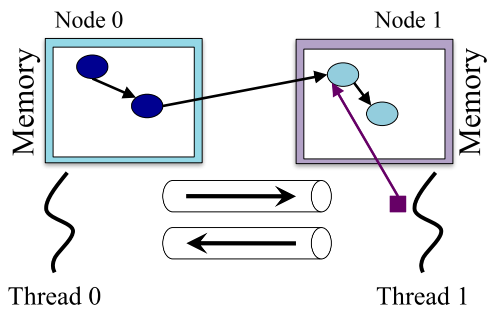
Enforces memory access locality
by trading remote memory accesses by messages



Remote reference ⇒ sends it to its home-node

Prevent remote access using messages

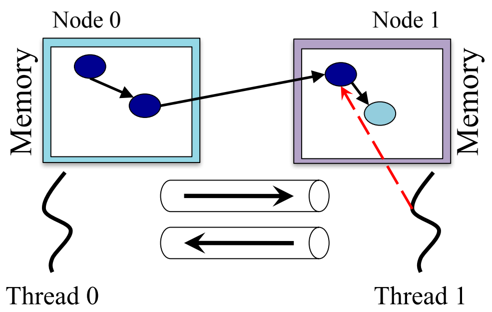
Enforces memory access locality
by trading remote memory accesses by messages



Remote reference ⇒ sends it to its home-node

Prevent remote access using messages

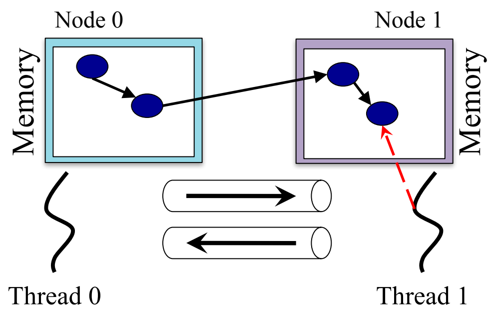
Enforces memory access locality
by trading remote memory accesses by messages



And continue the graph traversal locally

Prevent remote access using messages

Enforces memory access locality
by trading remote memory accesses by messages

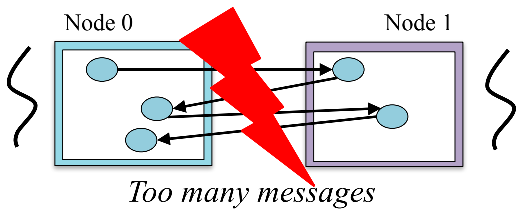


And continue the graph traversal locally

Using messages enforces local access…

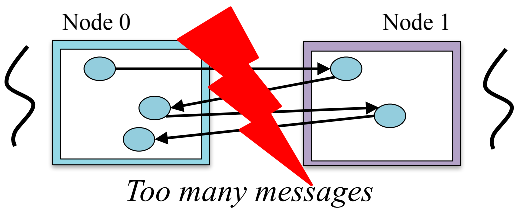
…but opens up other performance challenges

Problem1: a msg is costlier than a remote access



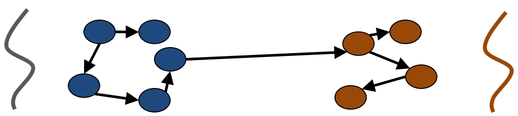
Inter-node messages must be minimized

Problem1: a msg is costlier than a remote access

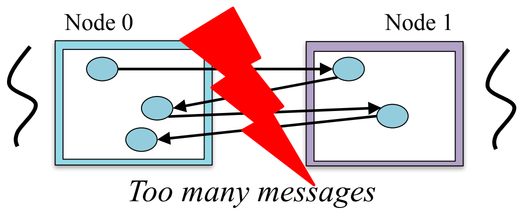


Inter-node messages must be minimized

- Observation: app threads naturally create clusters of new allocated objs
- 99% of recently allocated objects are clustered

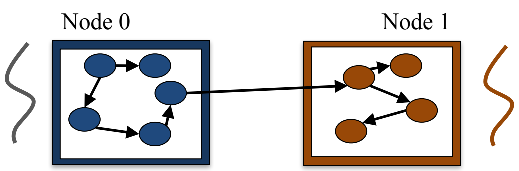


Problem1: a msg is costlier than a remote access



Inter-node messages must be minimized

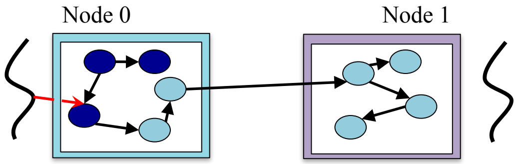
- Observation: app threads naturally create clusters of new allocated objs
- 99% of recently allocated objects are clustered



Approach: let objects allocated by a thread stay on its node

Problem2: Limited parallelism

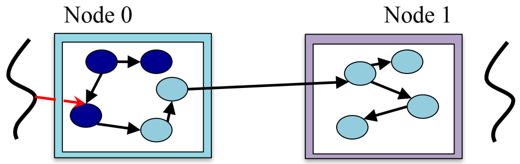
- Due to serialized traversal of object clusters across nodes



Node 1 idles while node 0 collects its memory

Problem2: Limited parallelism

- Due to serialized traversal of object clusters across nodes



Node 1 idles while node 0 collects its memory

- Solution: adaptive algorithm

  Trade-off between locality and parallelism

  1. Prevent remote access by using messages when not idling
  2. Steal and access remote objects otherwise

Outline

- Why GC doesn't scale?
- Our Solution: NumaGiC
- Evaluation

Evaluation

- Comparison of NumaGiC with –
  1. ParallelScavenge (PS): baseline stop-the-world GC of Hotspot
  2. Improved PS: PS with lock-free data structures and interleaved heap space
  3. NAPS: Improved PS + slightly better locality, but no messages
- Metrics
  - GC throughput –
    - amount of live data collected per second (GB/s)
    - Higher is better
  - Application performance –
    - Relative to improved PS
    - Higher is better

Experiments

Name	Description	Heap Size	
		Amd48	Intel80
Spark	In-memory data analytics (page rank computation)	110 to 160GB	250 to 350GB
Neo4j	Object graph database (Single Source Shortest Path)	110 to 160GB	250 to 350GB
SPECjbb2013	Business-logic server	24 to 40GB	24 to 40GB
SPECjbb2005	Business-logic server	4 to 8GB	8 to 12GB

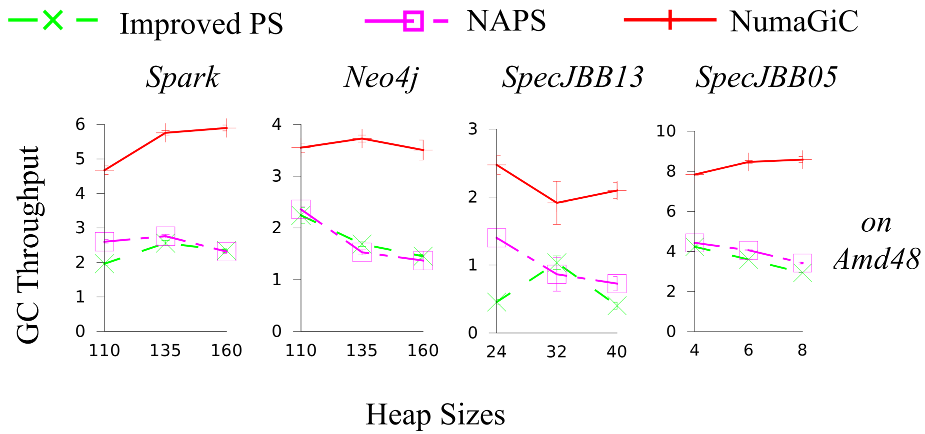
1 billion edge Friendster dataset

The 1.8 billion edge Friendster dataset

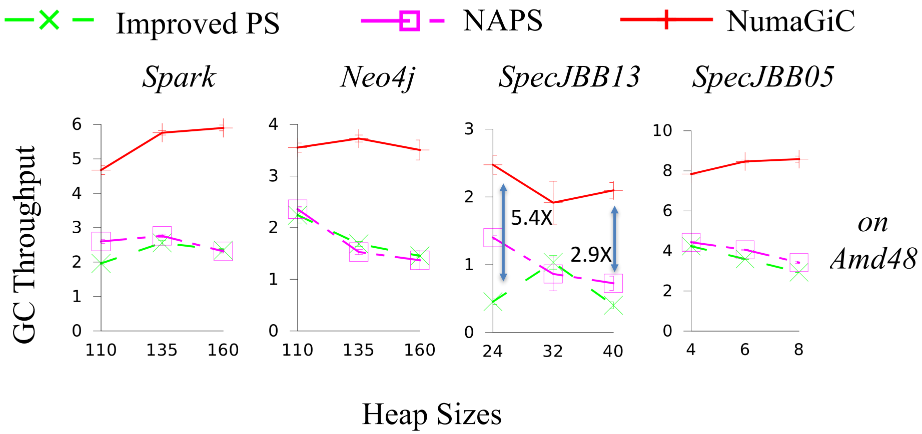
Hardware settings –

1. AMD Magny Cours with 8 nodes, 48 threads, 256 GB of RAM
2. Xeon E7-2860 with 4 nodes, 80 threads, 512 GB of RAM

GC Throughput (GB collected per second)

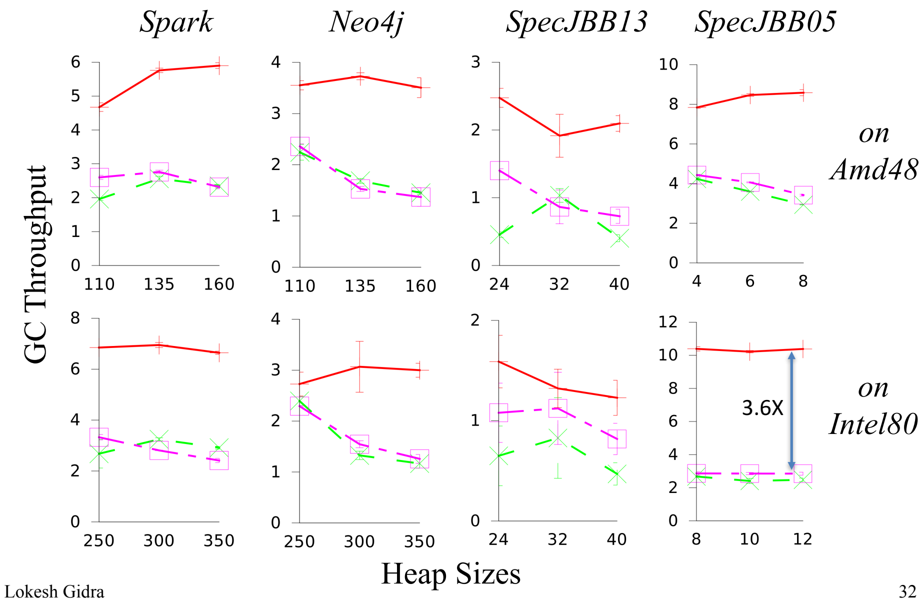


GC Throughput (GB collected per second)

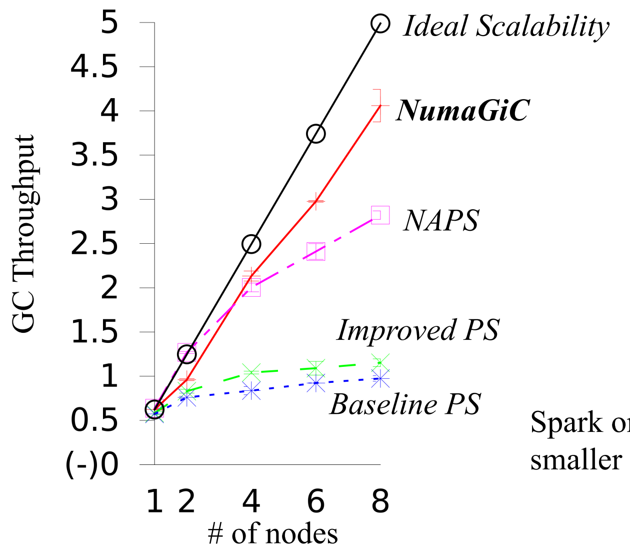


NumaGiC multiplies GC performance up to 5.4X

GC Throughput (GB collected per second)

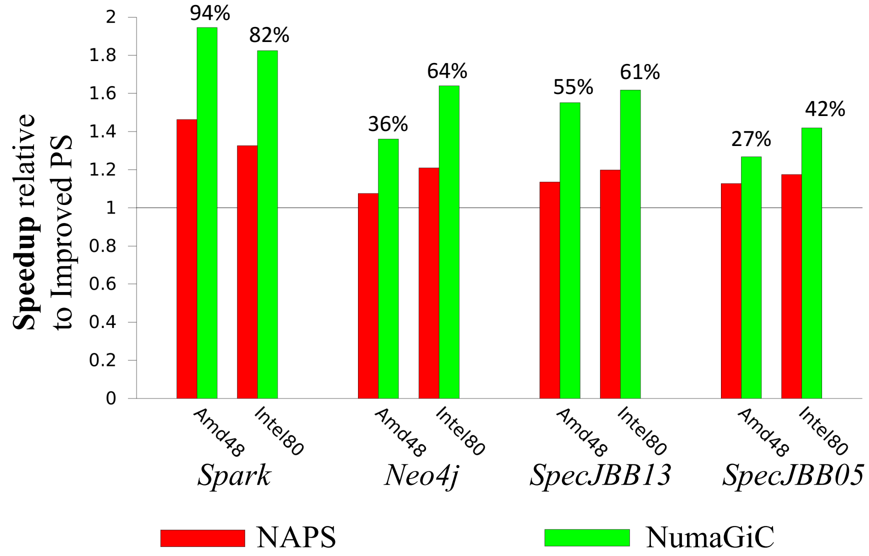


GC Throughput Scalability

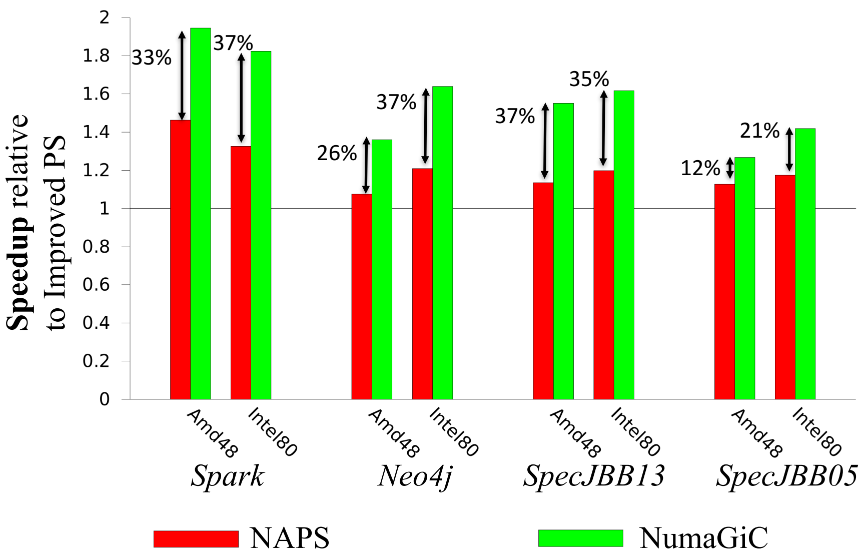


Spark on Amd48 with a smaller dataset of 40GB

Application speedup



Application speedup



Conclusion

- Performance of data-intensive apps relies on GC performance
- Memory access locality has huge effect on GC performance
- Enforcing locality can be detrimental for parallelism in GCs
- Future work: NUMA-aware concurrent GCs

Conclusion

- Performance of data-intensive apps relies on GC performance
- Memory access locality has huge effect on GC performance
- Enforcing locality can be detrimental for parallelism in GCs
- Future work: NUMA-aware concurrent GCs

Thank You ☺

Large multicores provide this power

But scalability is hard to achieve
because software stack was not designed for

Application	*Data analytic*
Middleware	*Hadoop, Spark, Neo4j, Cassandra…*
Language runtime	*JVM, CLI, Python, R…*
Operating system	*Linux, Windows…*
Hypervisor	*Xen, VMWare…*
I/O controllers / Cores / Memory Banks	

Large multicores provide this power

But scalability is hard to achieve
because software stack was not designed for

Application	*Data analytic*
Middleware	*Hadoop, Spark, Neo4j, Cassandra…*
Language runtime	*JVM, CLI, Python, R…*
Operating system	*Linux, Windows…*

Do not consider hypervisors in this talk:
Software stack is already complex
and hard to analyze!

controllers / Cores / Banks