Improving the Design and the Performance of Managed Runtime Environments

Gaël Thomas

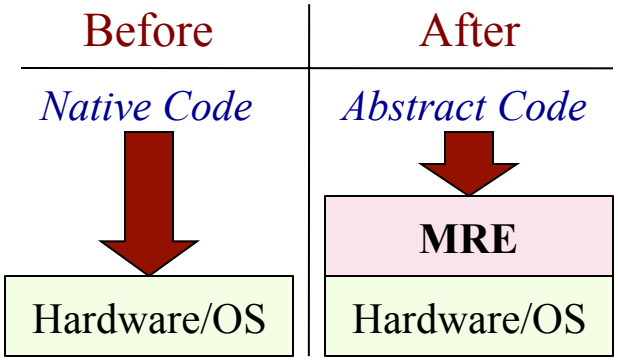
INRIA/LIP6

Exploiting Multicores in Managed Runtime Environments

Lokesh Gidra, Gaël Thomas, Julien Sopena Marc Shapiro

INRIA/LIP6

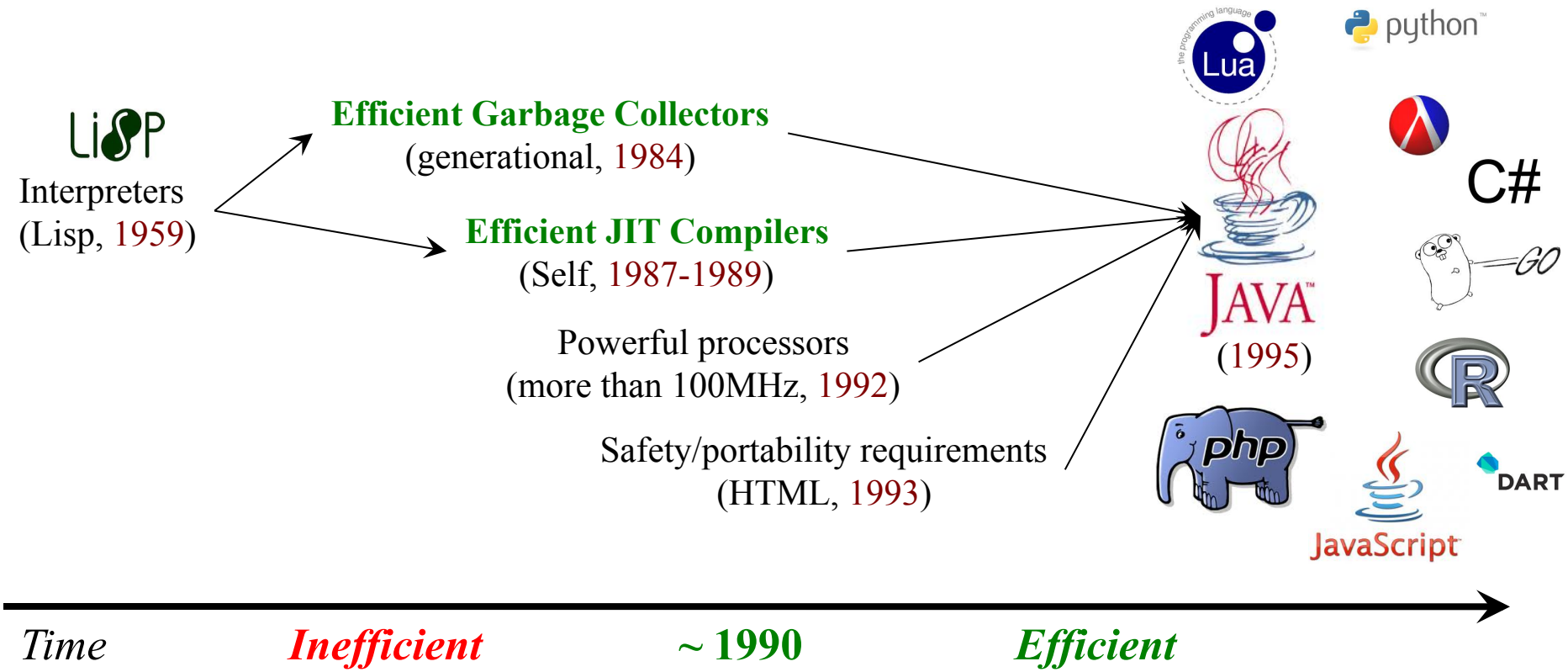
Managed Runtime Environments (MREs)



MRE: simulates an abstract hardware/OS

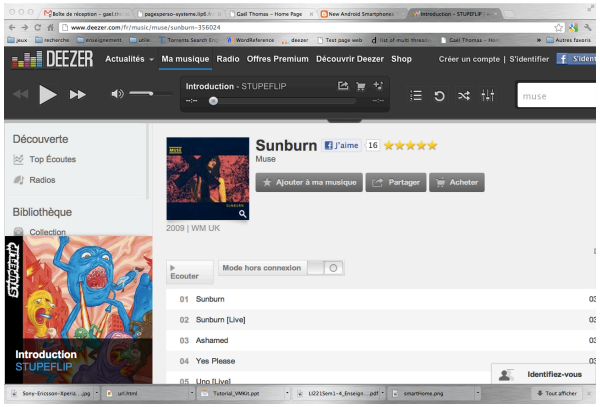
- ✓ **Safety**: isolate code from the rest of the system
- ✓ **Portability**: write once, run anywhere

MREs are efficient



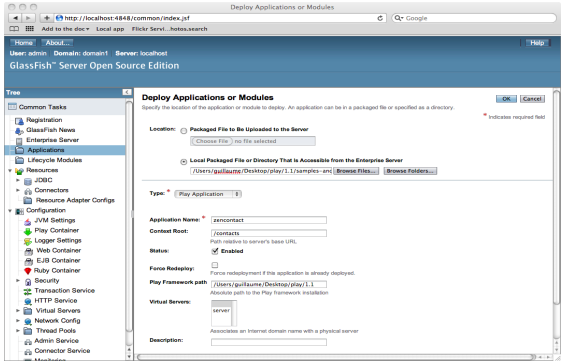
MREs are everywhere

Smartphones



Web browsers

Desktop



Web servers

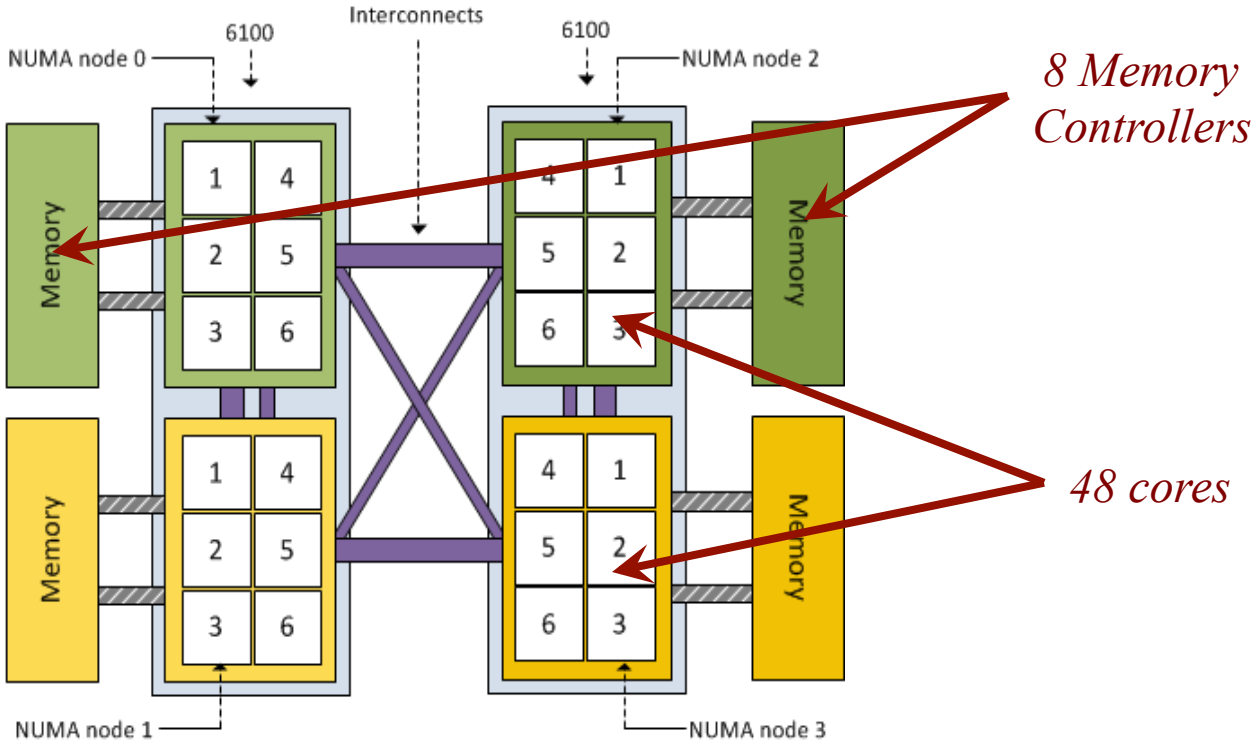
But they were not prepared to multicore

Most MREs were designed for a monocore architecture

⇒ Necessary to study their bottlenecks on a multicore architecture

Dell Power Edge r815
48-core
(Magny-Cours)

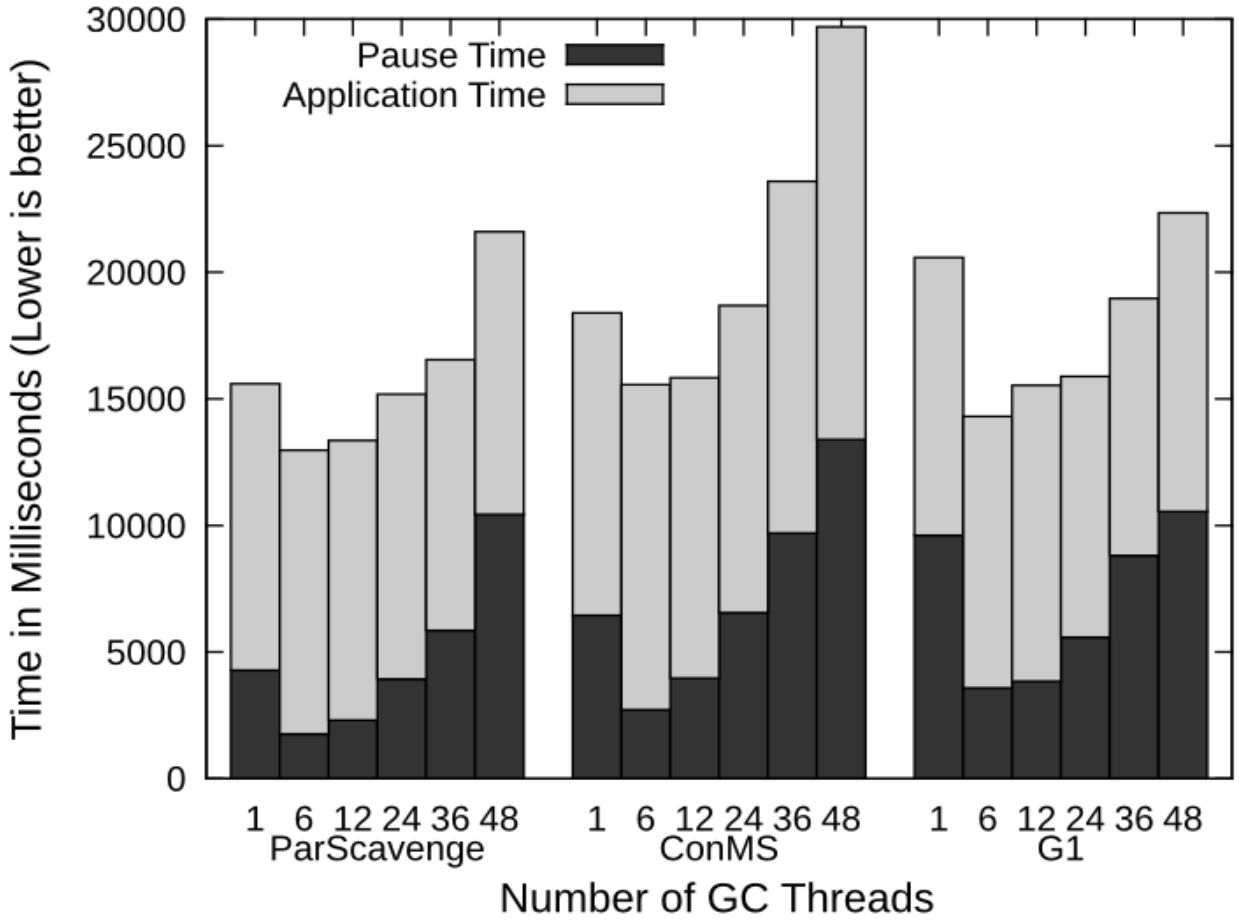
Samsung Galaxy S3
4-core



AMD Opteron 6172 Magny-Cours

GC Scalability (Lusearch) [PLOS'11]

HotSpot JVM's Garbage Collectors

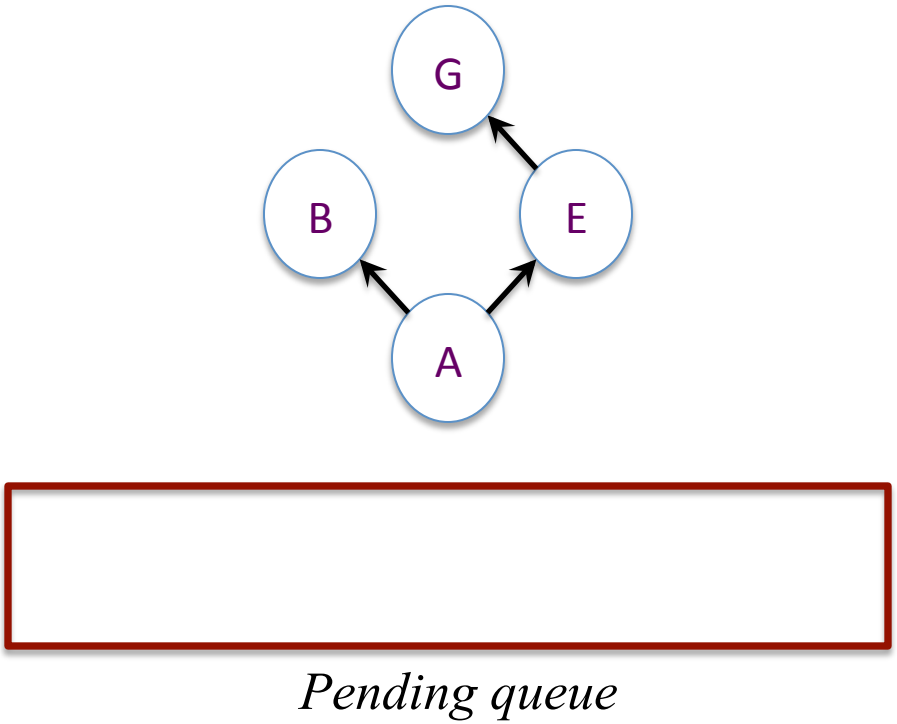


Pause time of the GC increases with GC threads

⇒ Negative scalability!

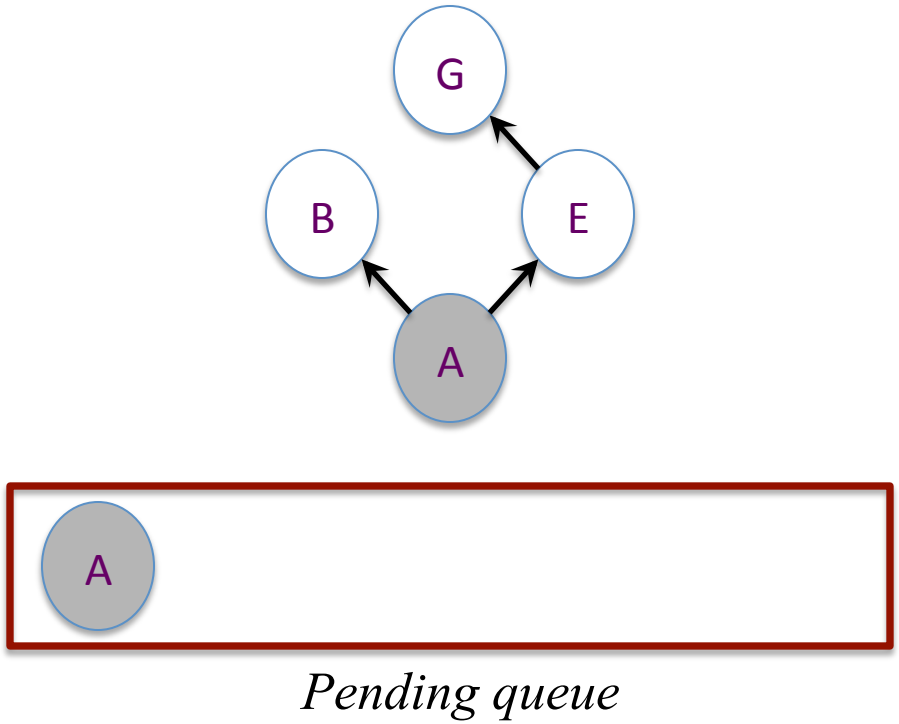
Why suspending the application

The concurrency issue



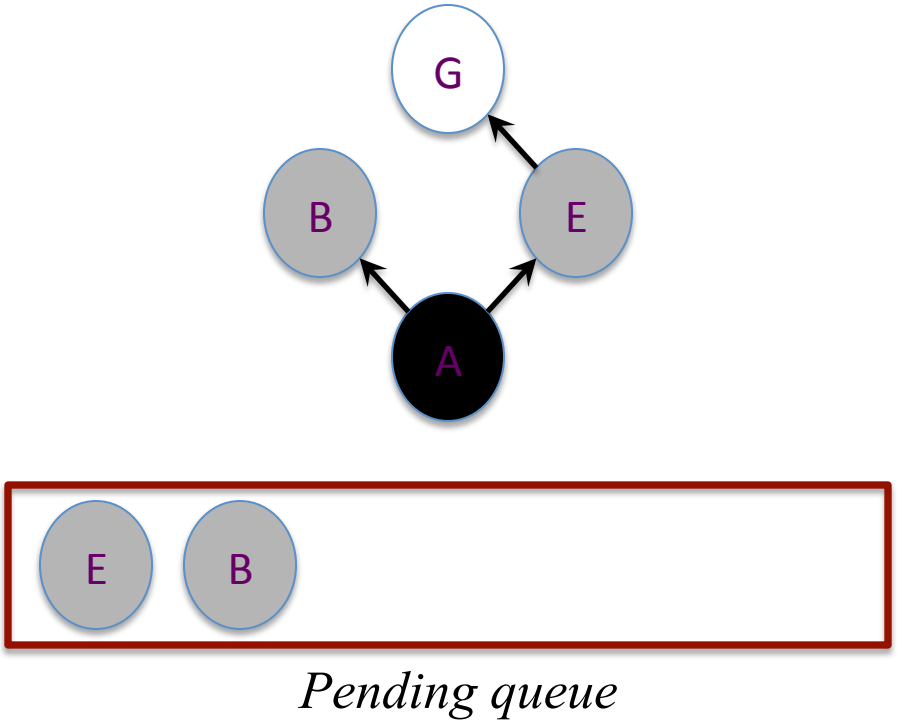
Why suspending the application

The concurrency issue



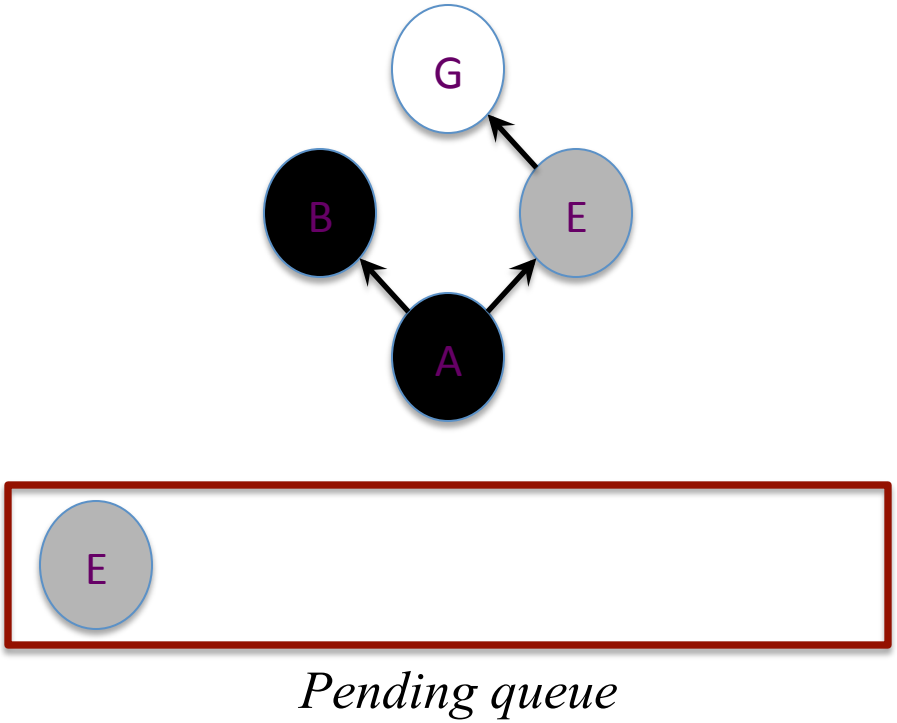
Why suspending the application

The concurrency issue



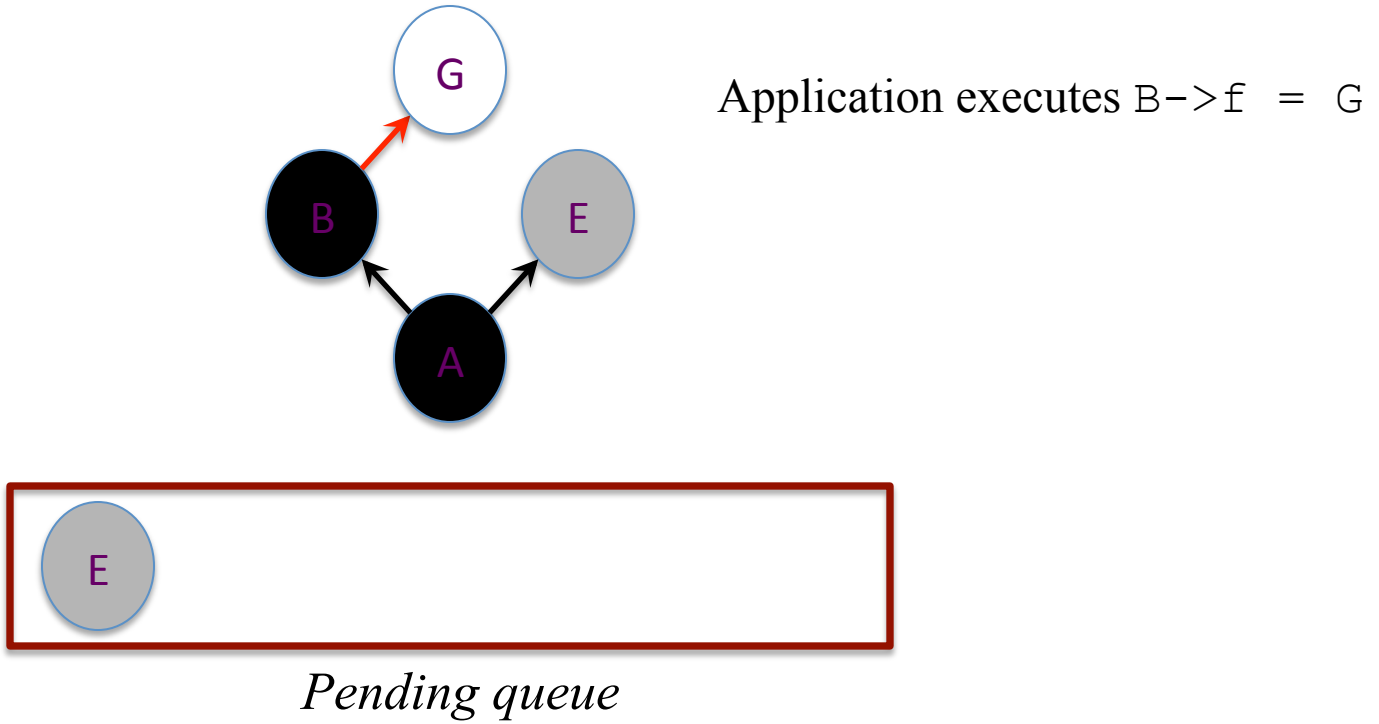
Why suspending the application

The concurrency issue



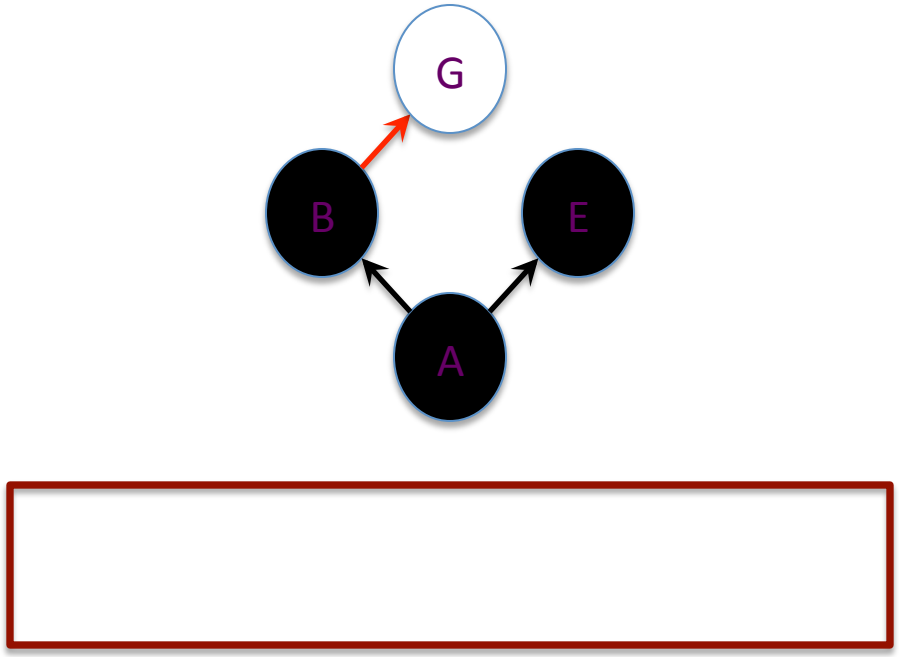
Why suspending the application

The concurrency issue



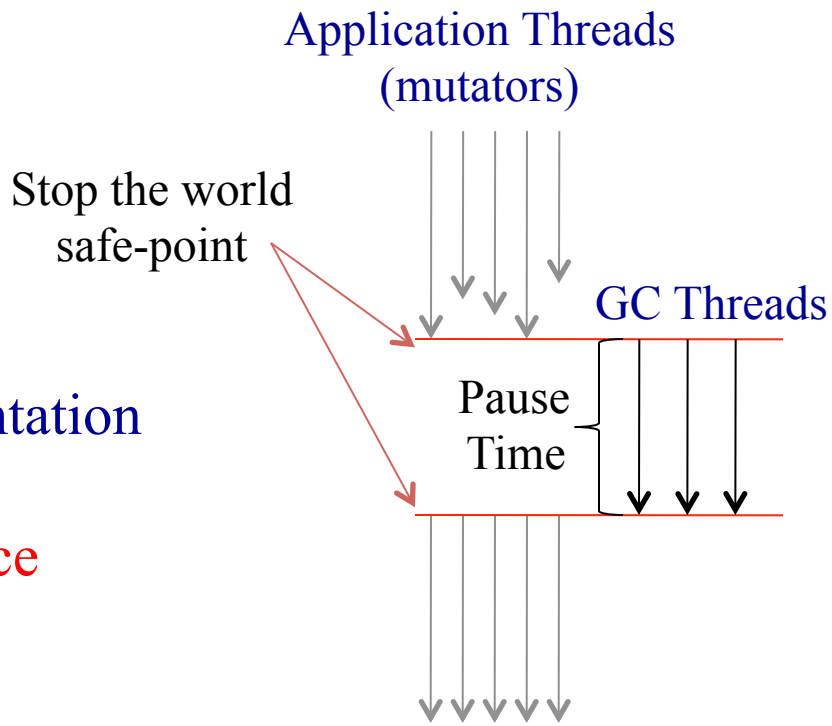
Why suspending the application

The concurrency issue ⇒ G freed while still used



Common believe

1. Stop-the-world:
   - Suspend the application
   + Simple to implement



2. Concurrent:
   fine-grain locking, code instrumentation
   - Hard to implement
   - Degrades application performance
   + Do not pause the application

Current believe:

STW are unacceptable for server apps [Iyengar, ISMM 2012]

Long pauses due to larger heaps

Our hypothesis

Increase in transistor count is for both memory and CPU

- ✓ Large heaps come with large core count
- ✓ STW GC should be still useful, provided they scale

Can we make a GC scales with the number of cores
to avoid the price of concurrent collectors?

Contribution

Identify the bottlenecks of Parallel Scavenge

(the most scalable GC of OpenJDK – used by default)

- ✓ Heavy contended locks
- ✓ Lack of NUMA-awareness

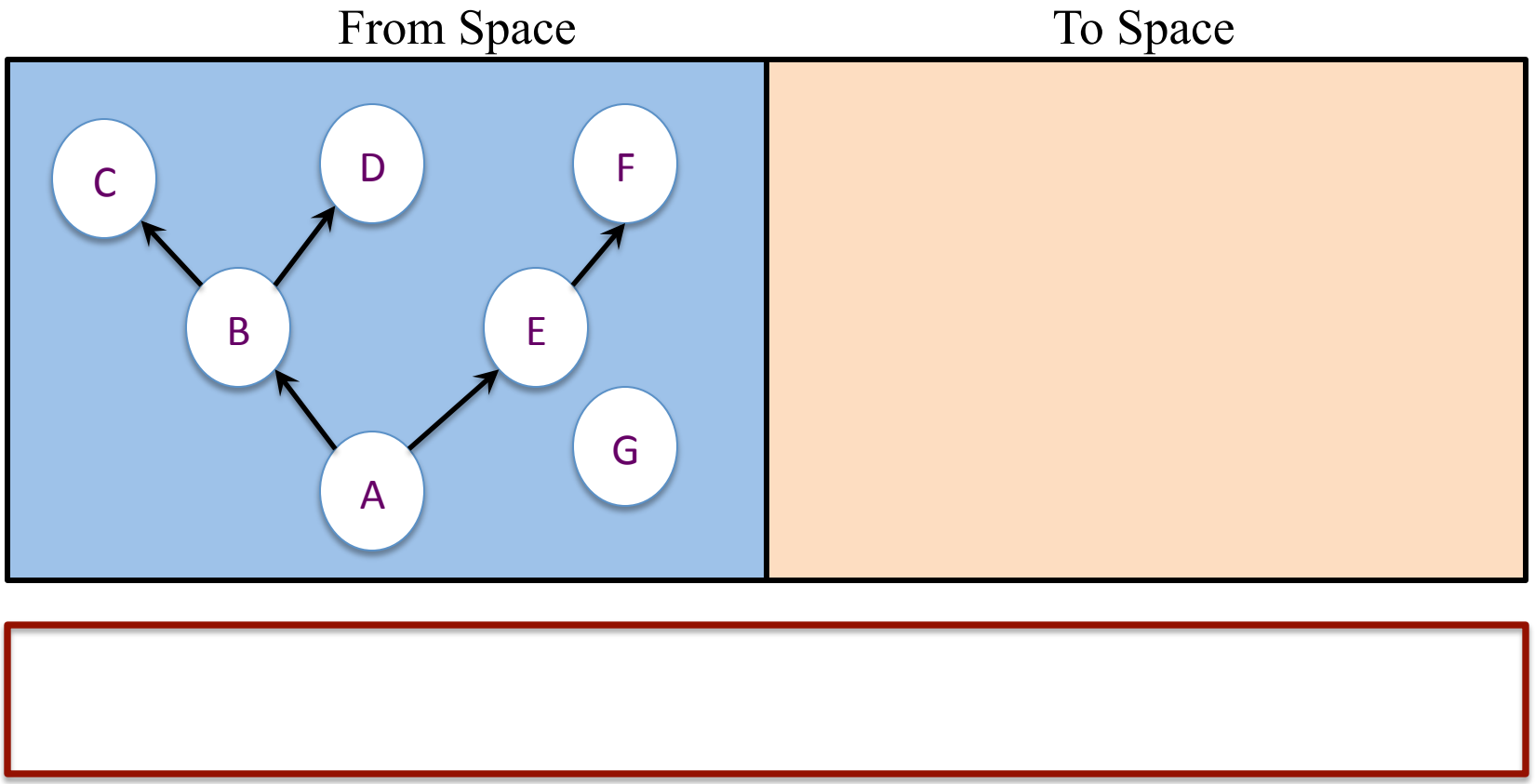
Solve the bottlenecks

- ✓ Remove all the locks during the collection
- ✓ Propose 3 NUMA-aware heap layouts
  - ❖ Interleaved: balance memory accesses across the nodes
  - ❖ Fragmented: balance + increase memory locality
  - ❖ Segregated: balance + perfect memory locality

1. Background
2. The lock bottleneck
3. The NUMA bottleneck
4. Evaluation
5. Conclusion

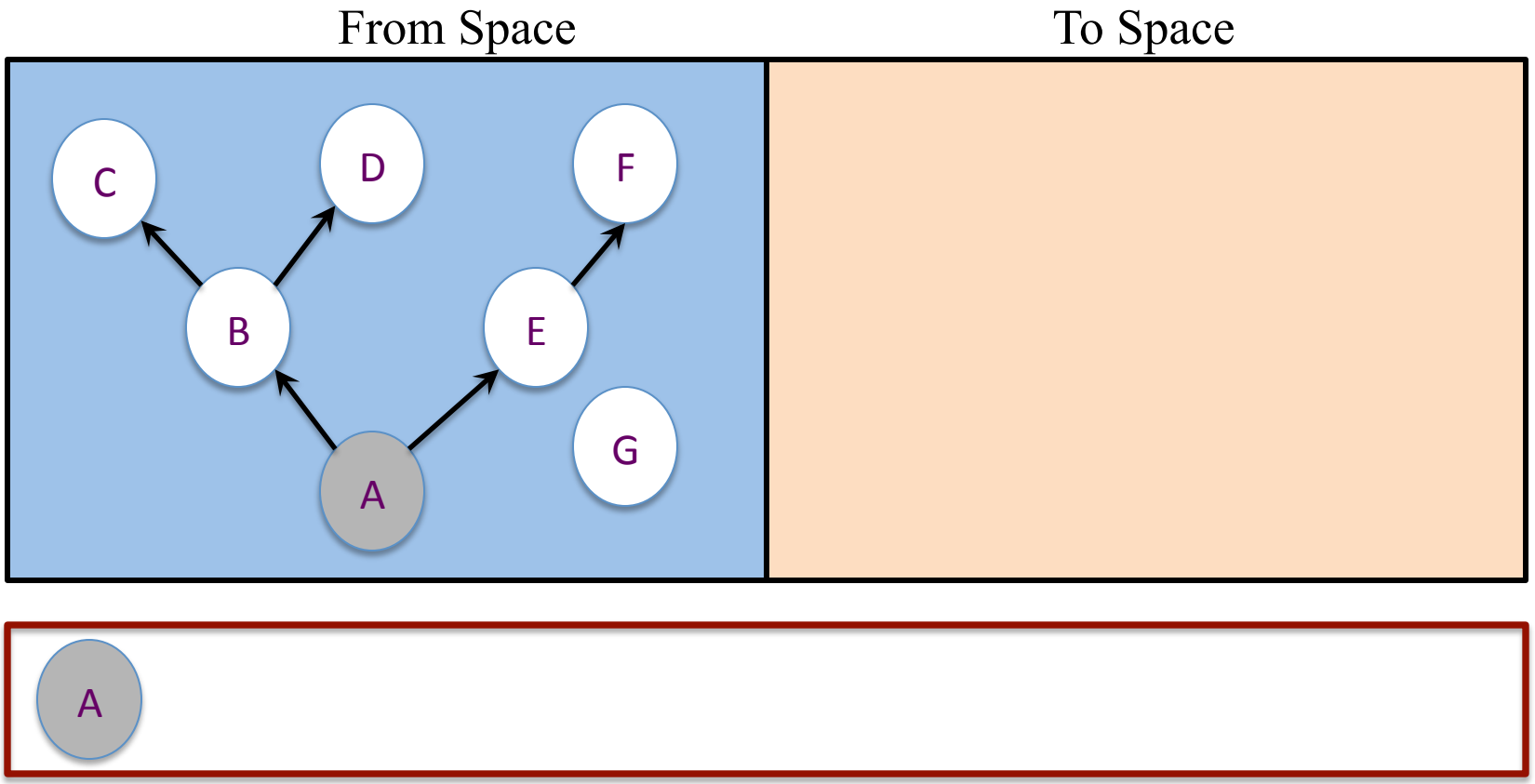
1. Background
2. The lock bottleneck
3. The NUMA bottleneck
4. Evaluation
5. Conclusion

Background: the copying collector



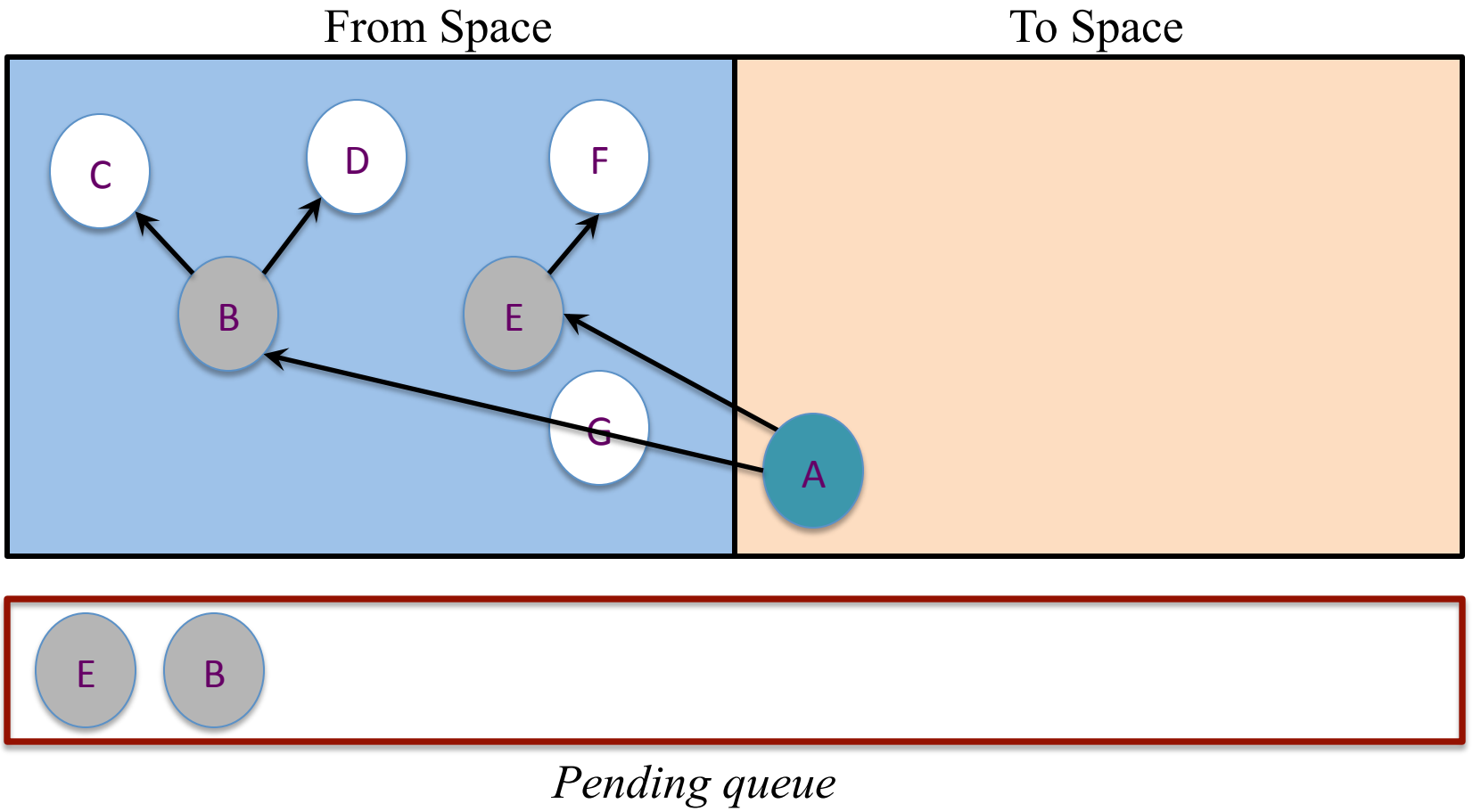
Background: the copying collector

Step 1: identify the root objects (globals, stack)



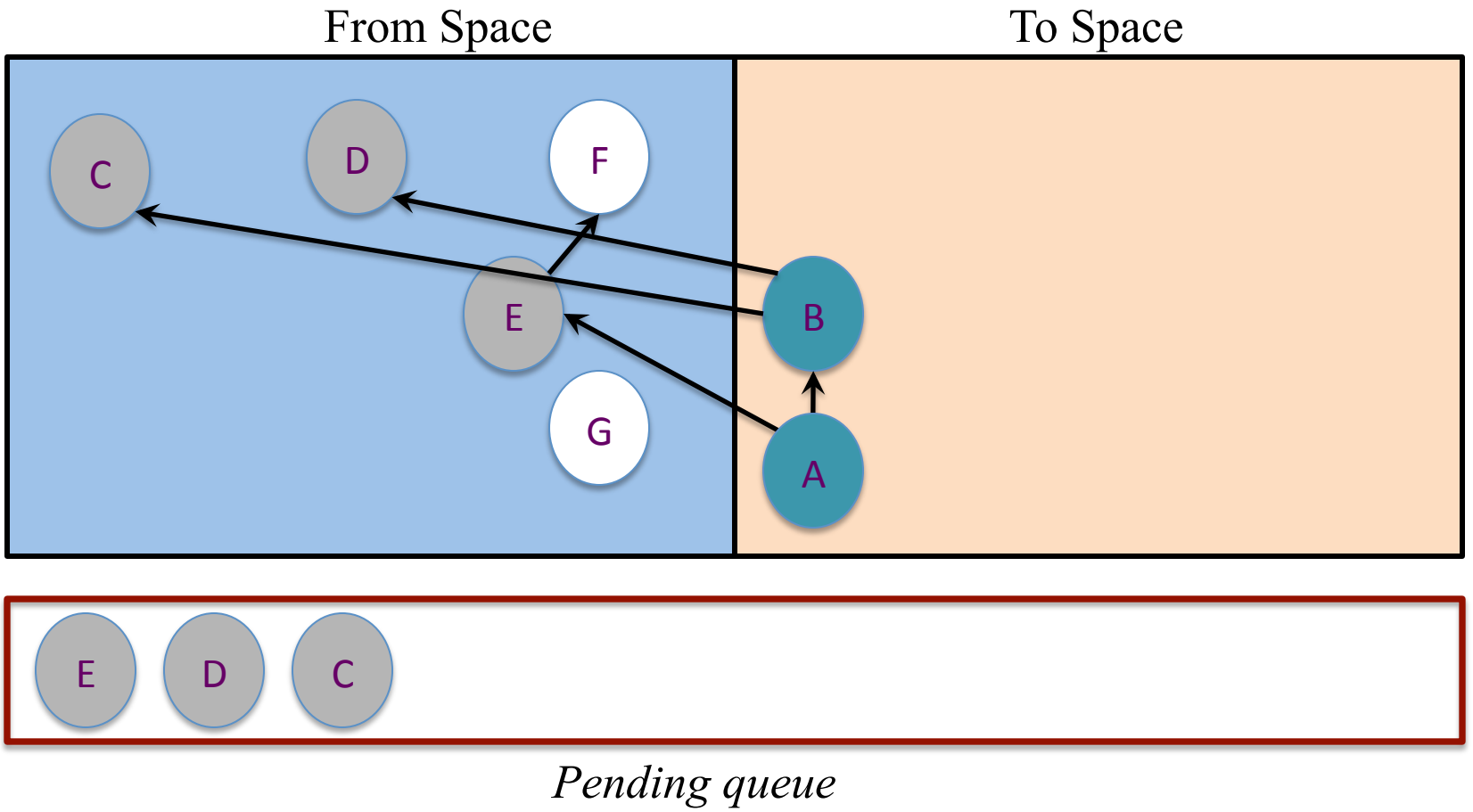
Background: the copying collector

Step 2: copy an object from the pending queue + update pending queue



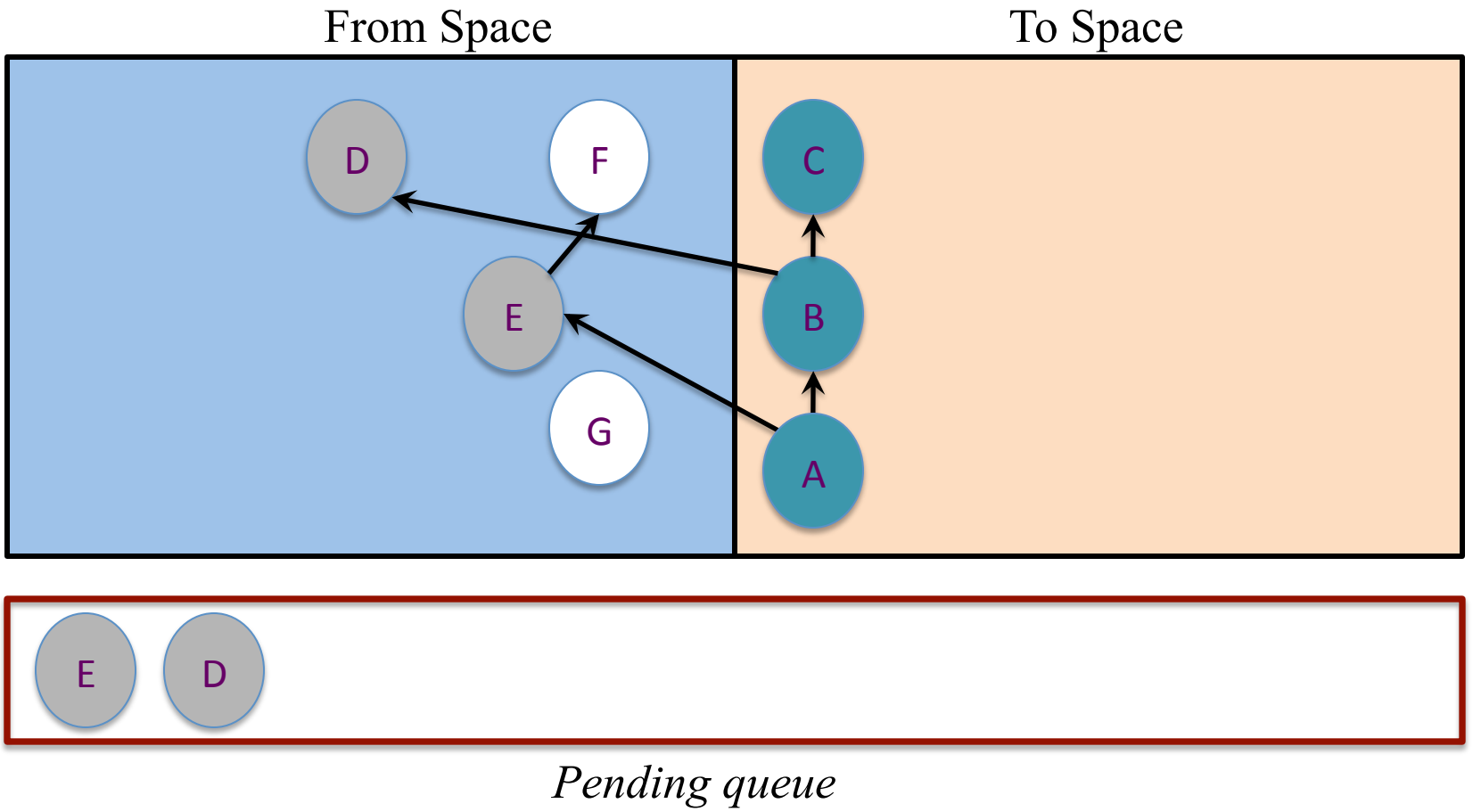
Background: the copying collector

Step 2: copy an object from the pending queue + update pending queue



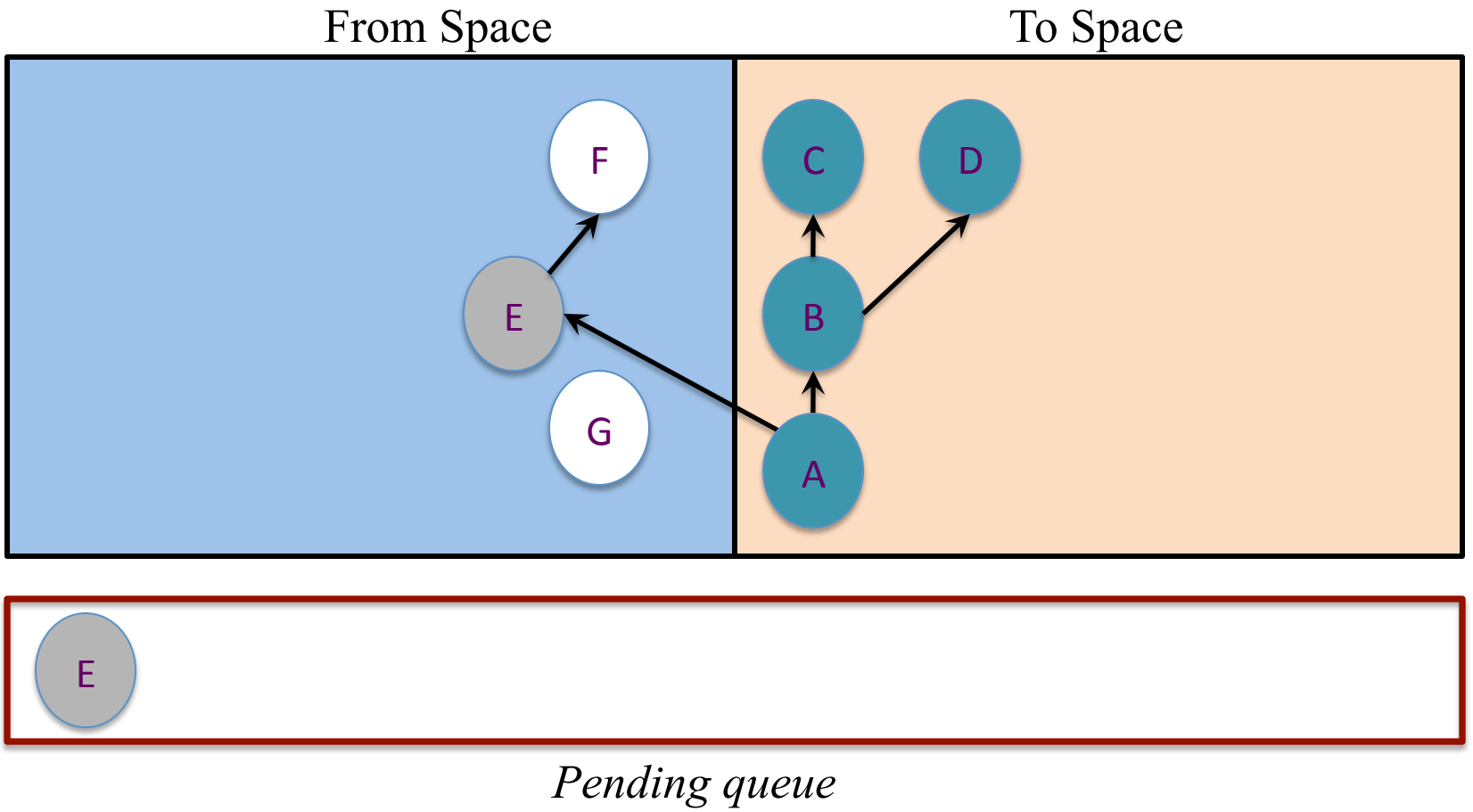
Background: the copying collector

Step 2: copy an object from the pending queue + update pending queue



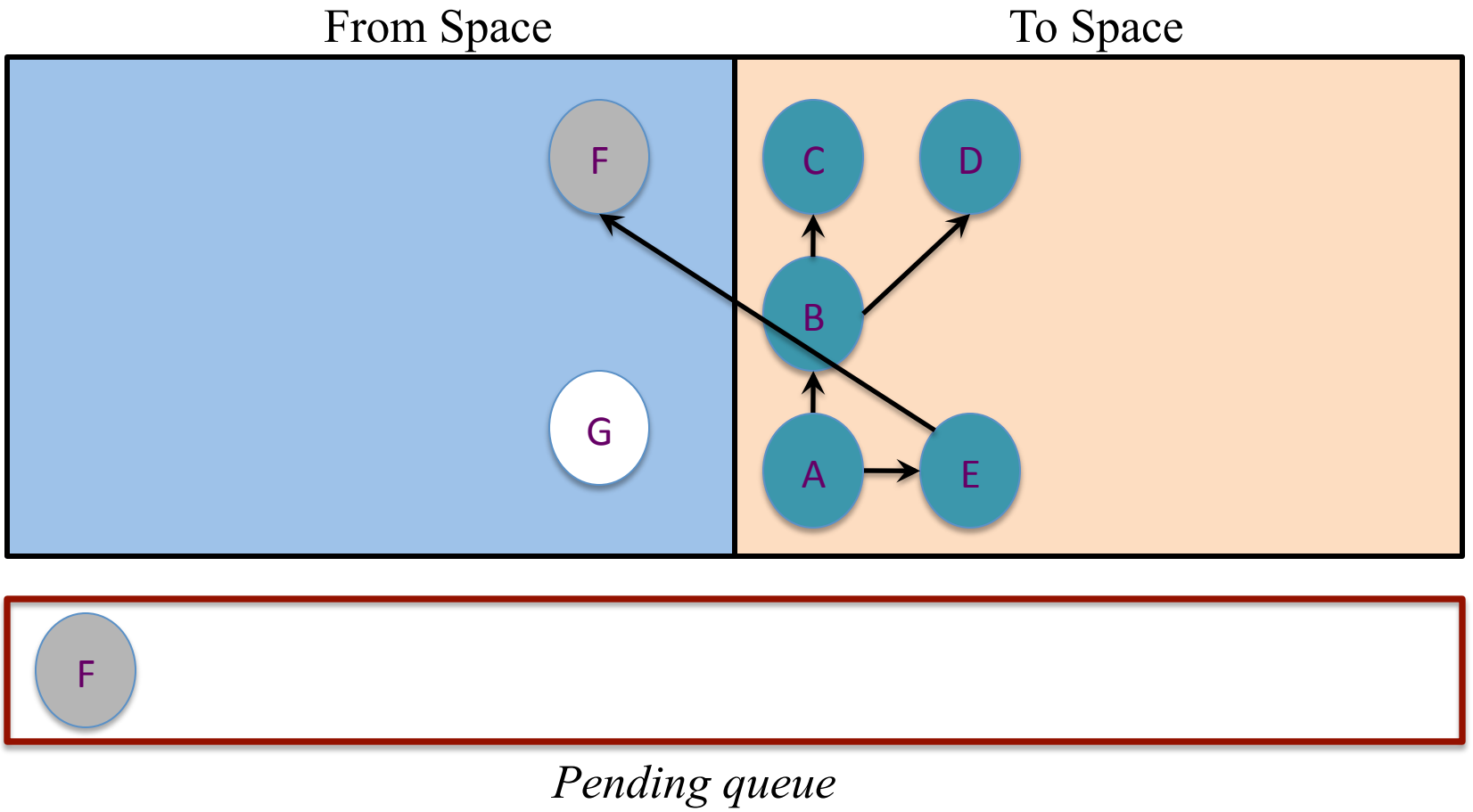
Background: the copying collector

Step 2: copy an object from the pending queue + update pending queue



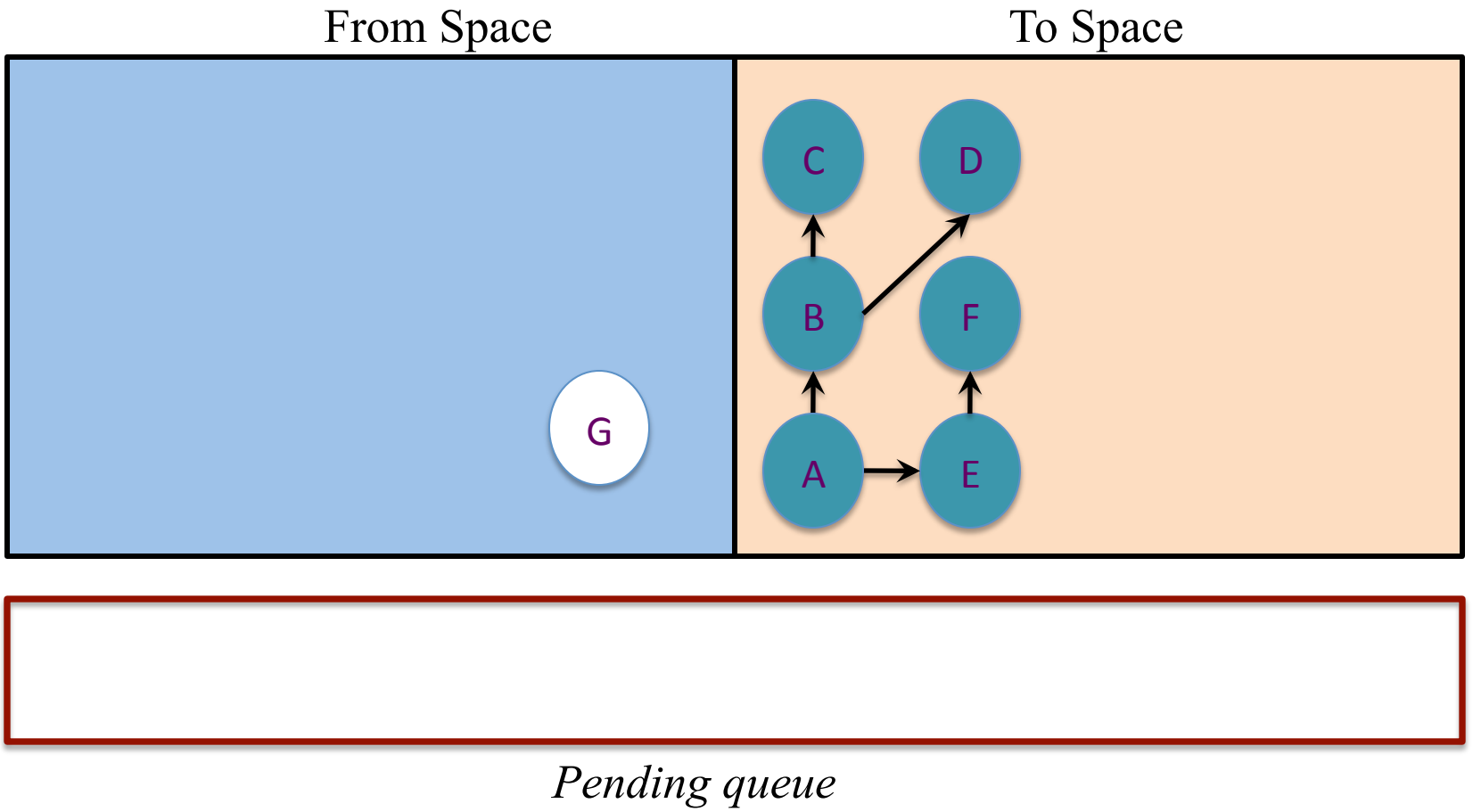
Background: the copying collector

Step 2: copy an object from the pending queue + update pending queue



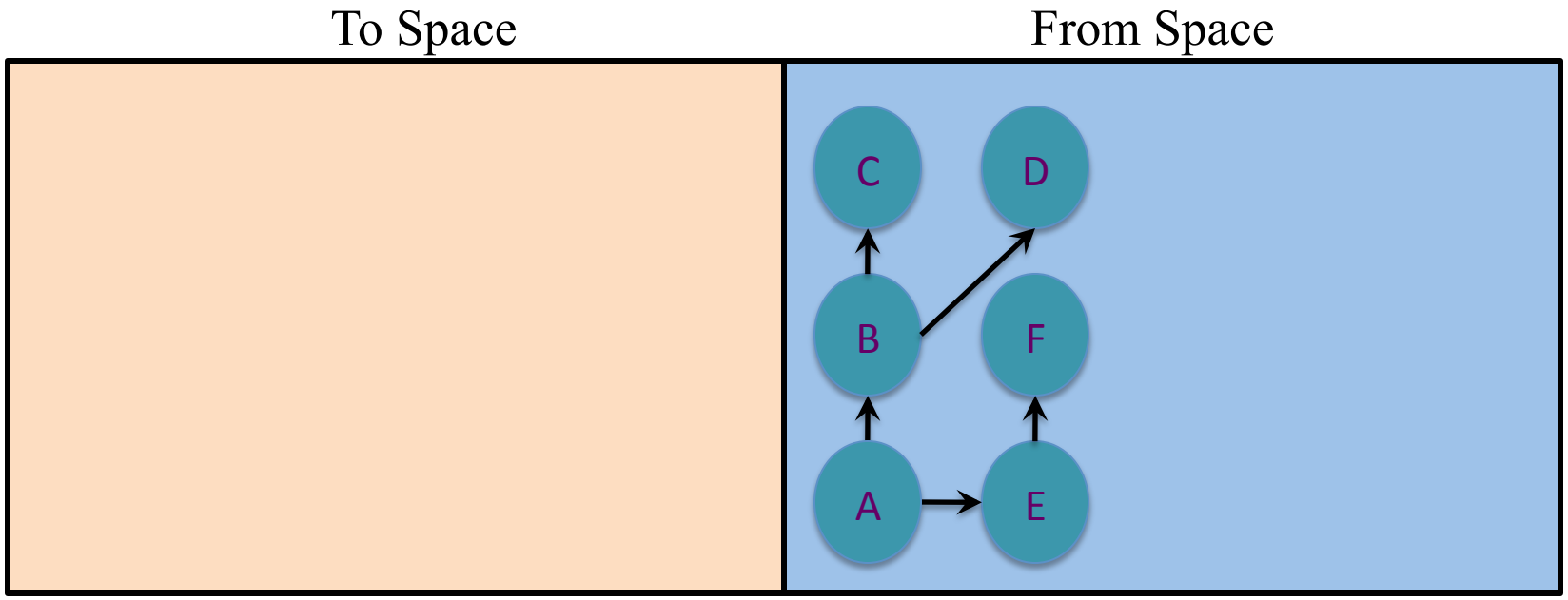
Background: the copying collector

Step 2: copy an object from the pending queue + update pending queue



Background: the copying collector

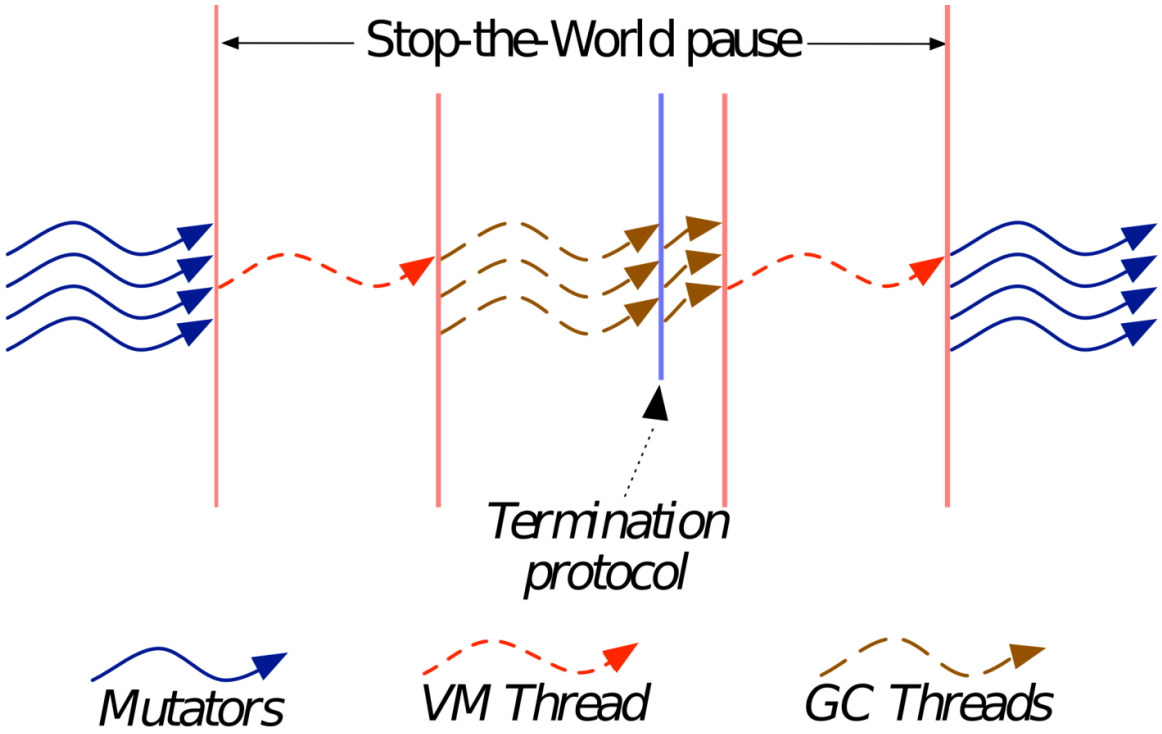
Step 3: invert the spaces + consider to space empty



Advantage: spaces are never fragmented

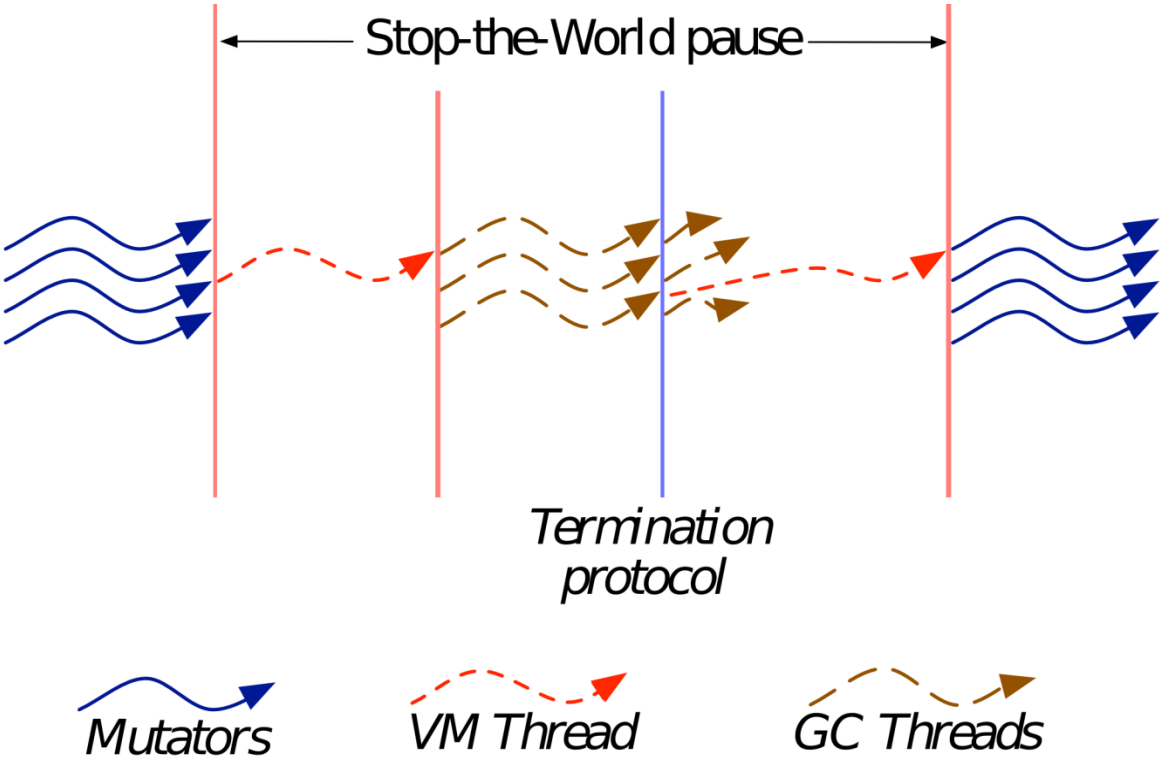
1. Background
2. The lock bottleneck
3. The NUMA bottleneck
4. Evaluation
5. Conclusion

Poor Synchronization in Parallel Scavenge



Coarse grained synchronization + use of monitors

Simplify synchronizations

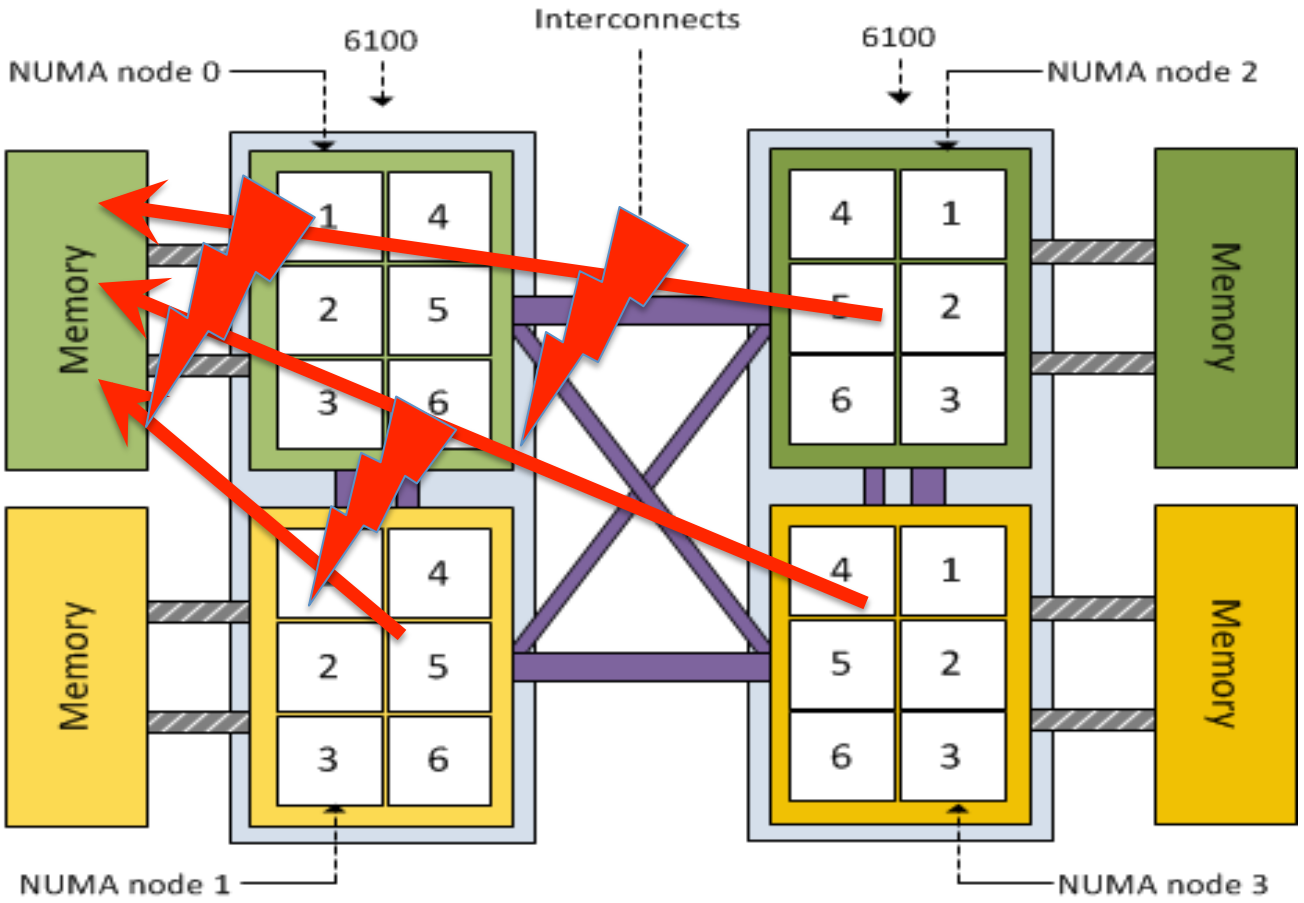


1. Background
2. The lock bottleneck
3. The NUMA bottleneck
4. Evaluation
5. Conclusion

Impact of a NUMA architecture

Problem 1: unbalanced memory accesses

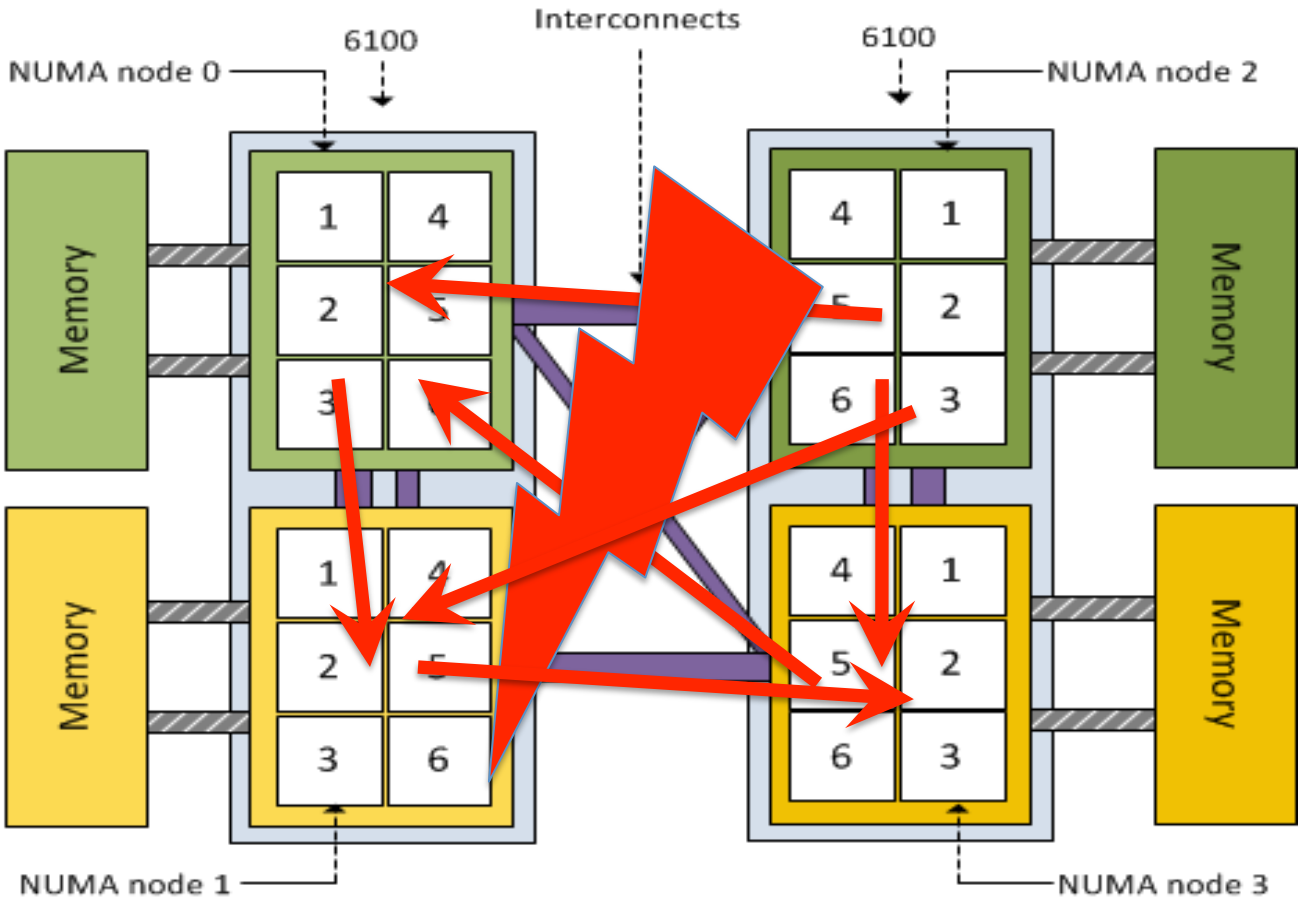
Interconnect or memory controllers saturate



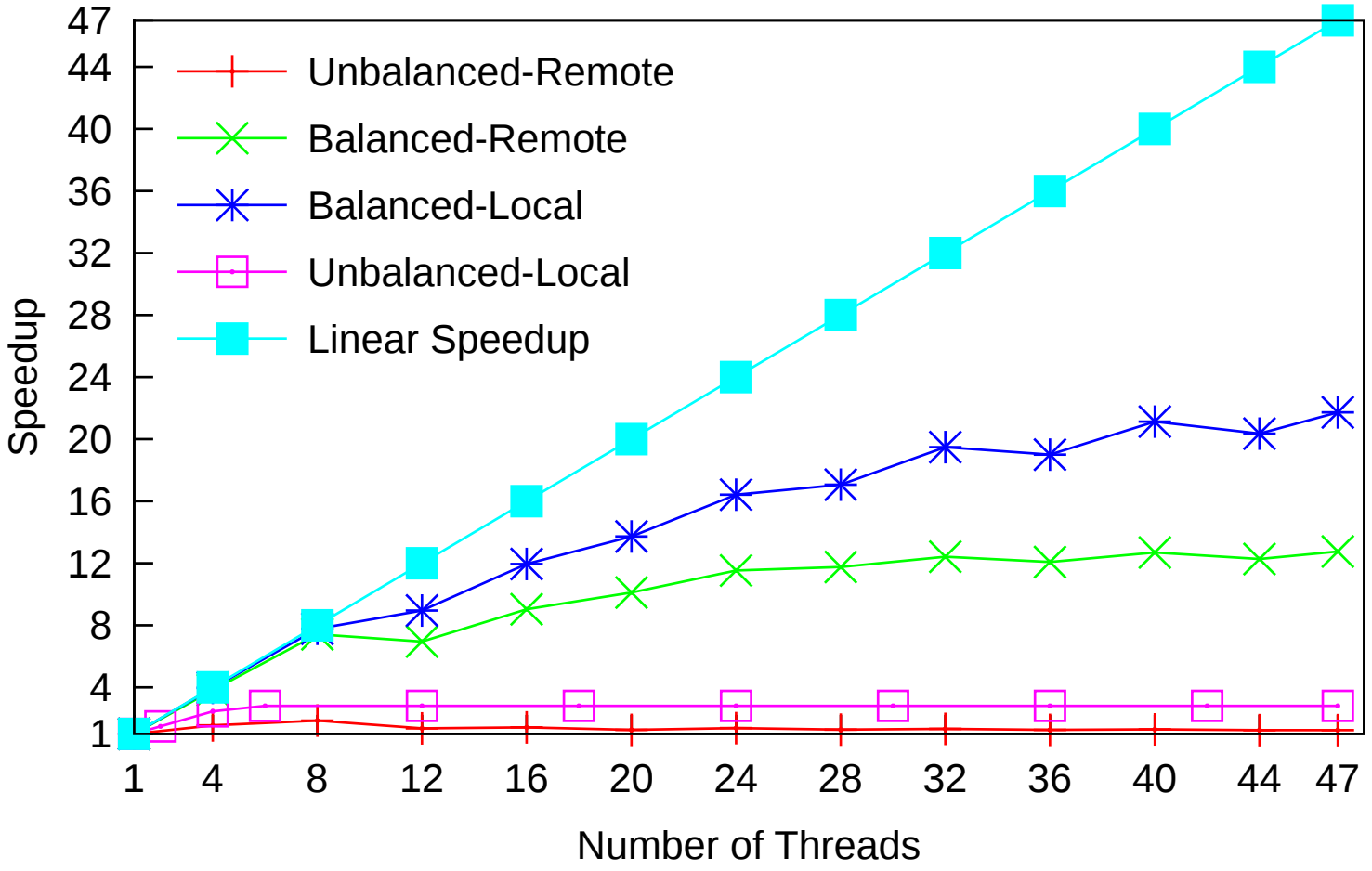
Impact of a NUMA architecture

Problem 2: remote memory accesses

Interconnect saturates

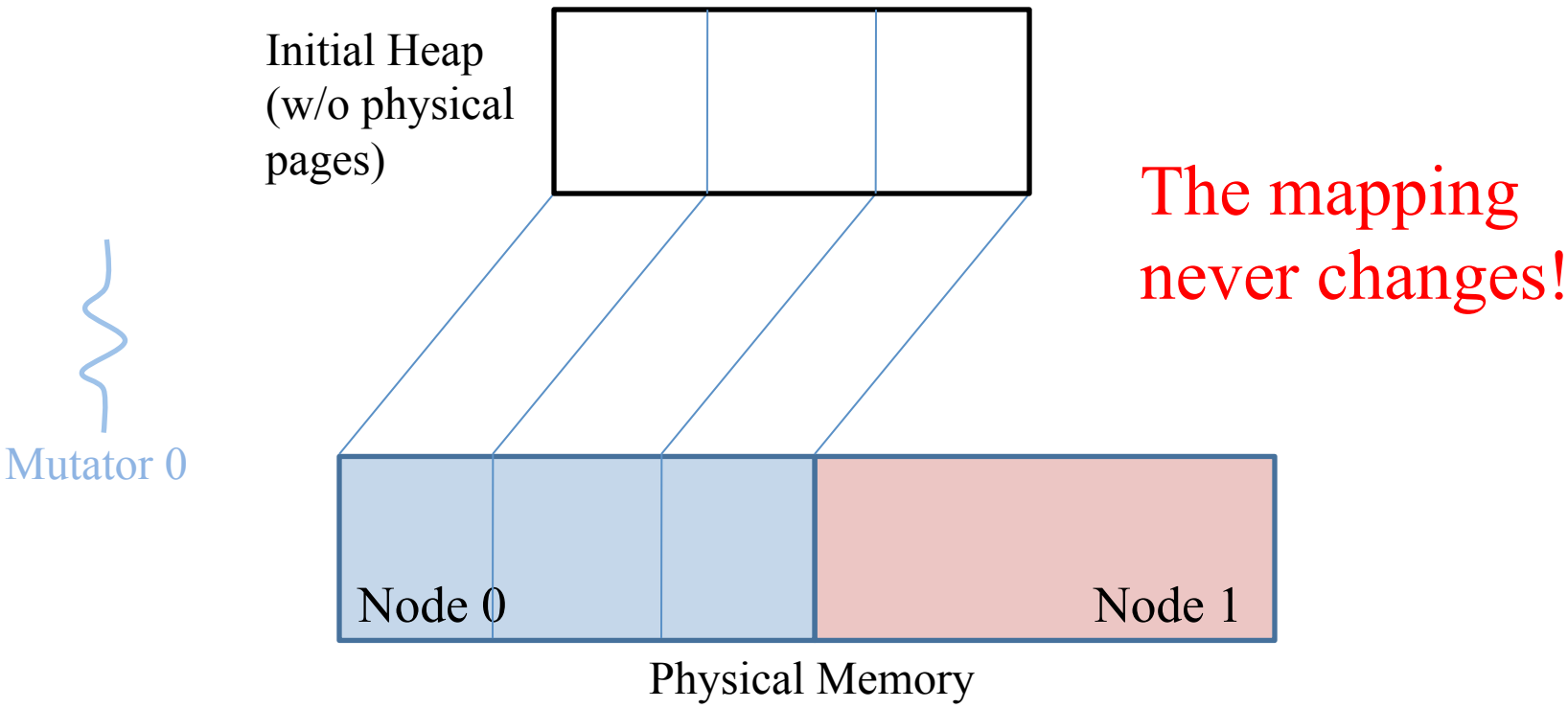


Impact of a NUMA architecture



Inefficient Memory Layout in ParallelScavenge (PS)

The initial thread fixes the mapping of physical pages

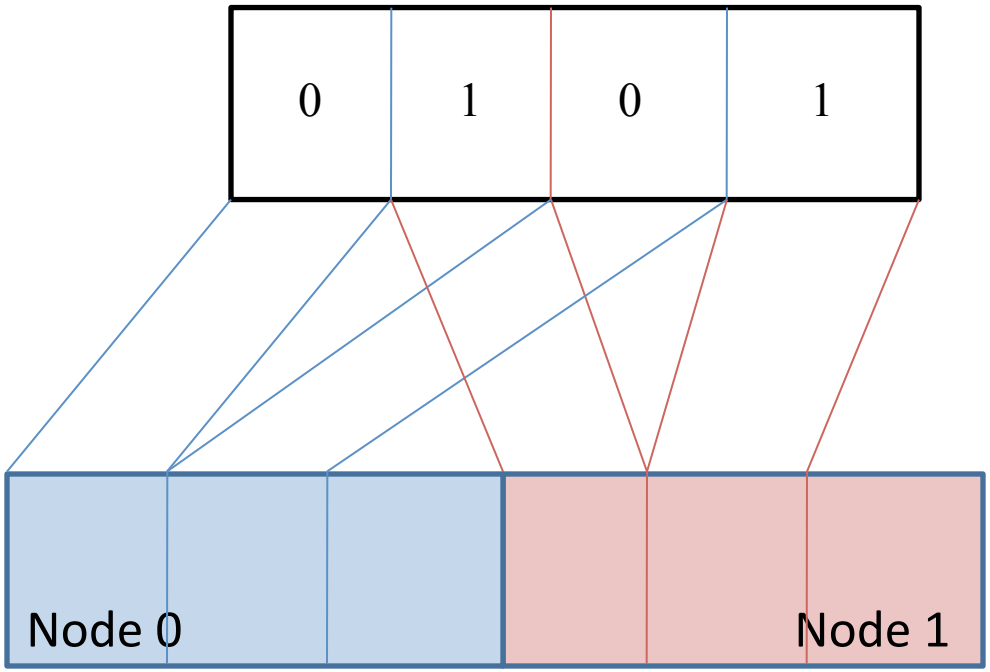


The mapping never changes!

SPECjbb2005 allocates ~95% of memory from a single node

Solution 1: Interleaved Space

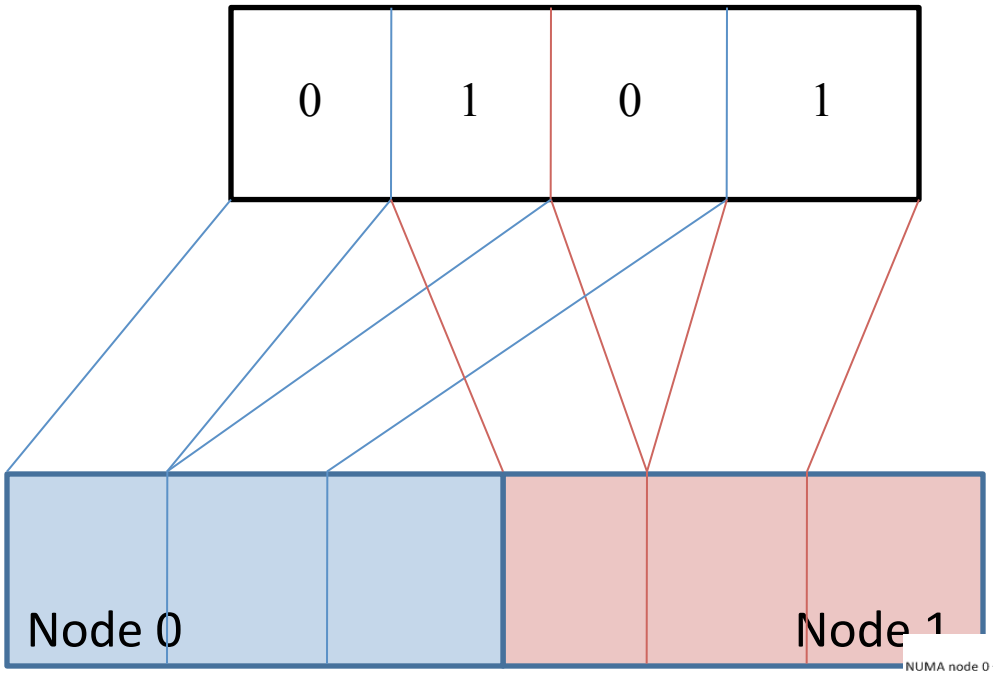
Map the pages on the node in round-robin



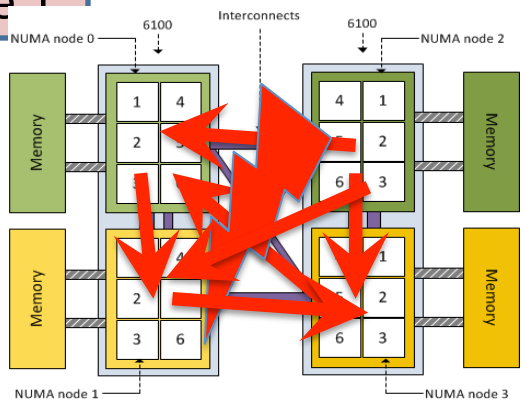
⇒ perfect memory balance

Solution 1: Interleaved Space

Idea: map the pages on the node in round-robin



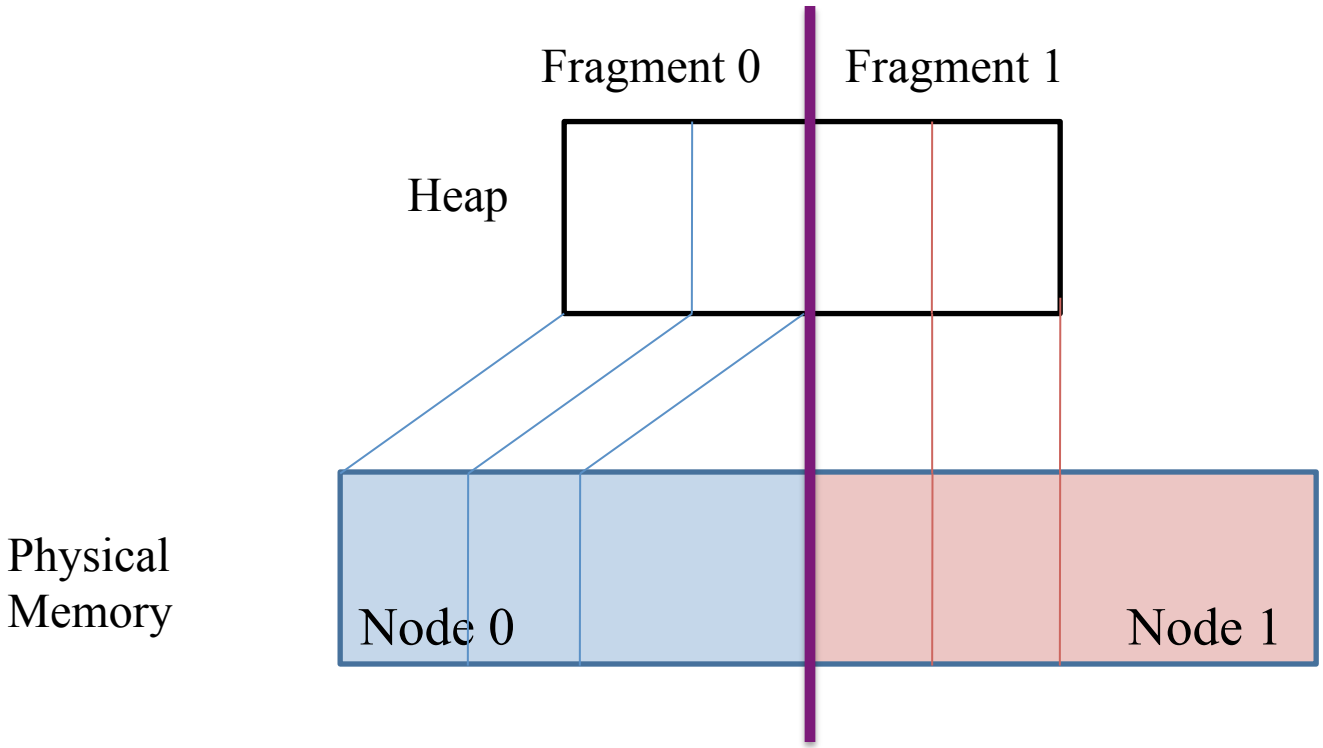
⇒ perfect memory balance but bad memory locality



Solution 2: Fragmented Space

Associate fragments to memory nodes

⇒ node-local allocation

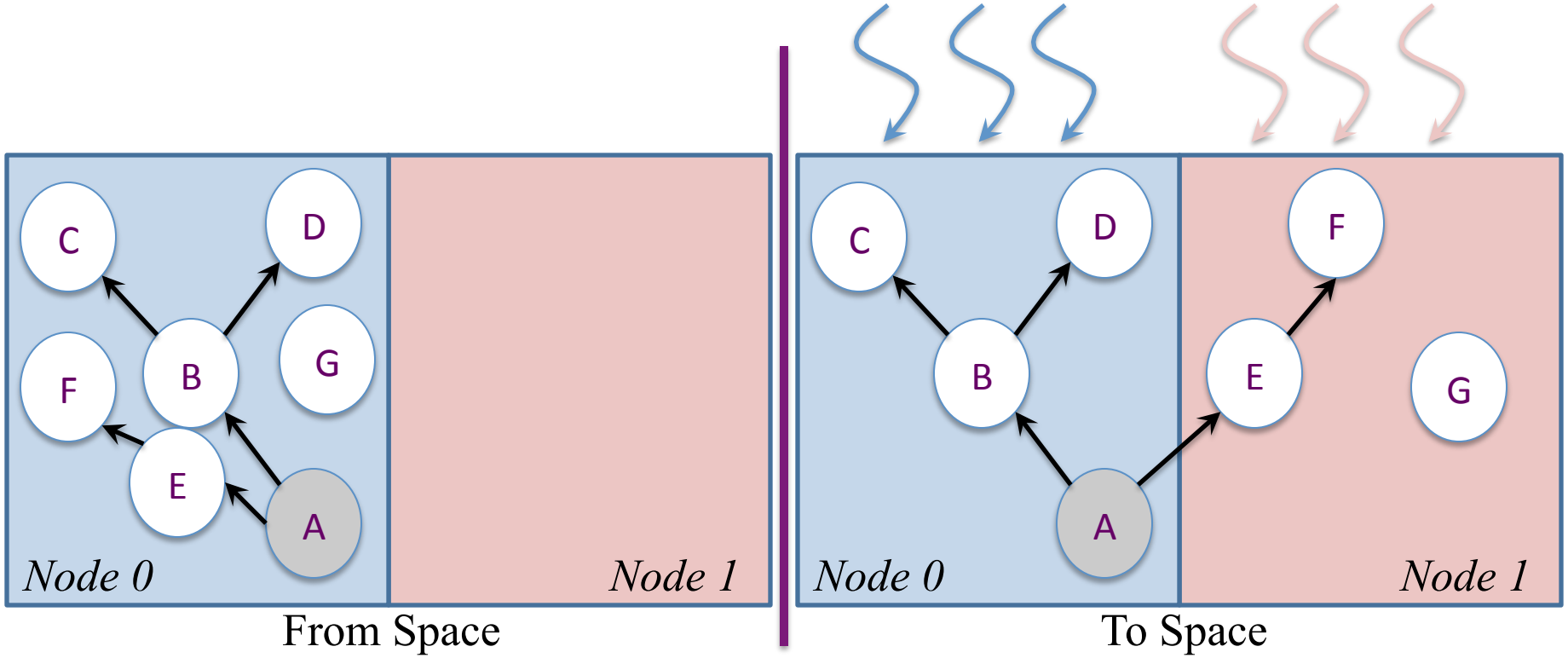


Solution 2: Fragmented Space

Naturally balance the load

- ✓ GC threads uniformly distributed on the nodes
- ✓ Efficient work stealing between GC threads

⇒ balance the copies on all the nodes



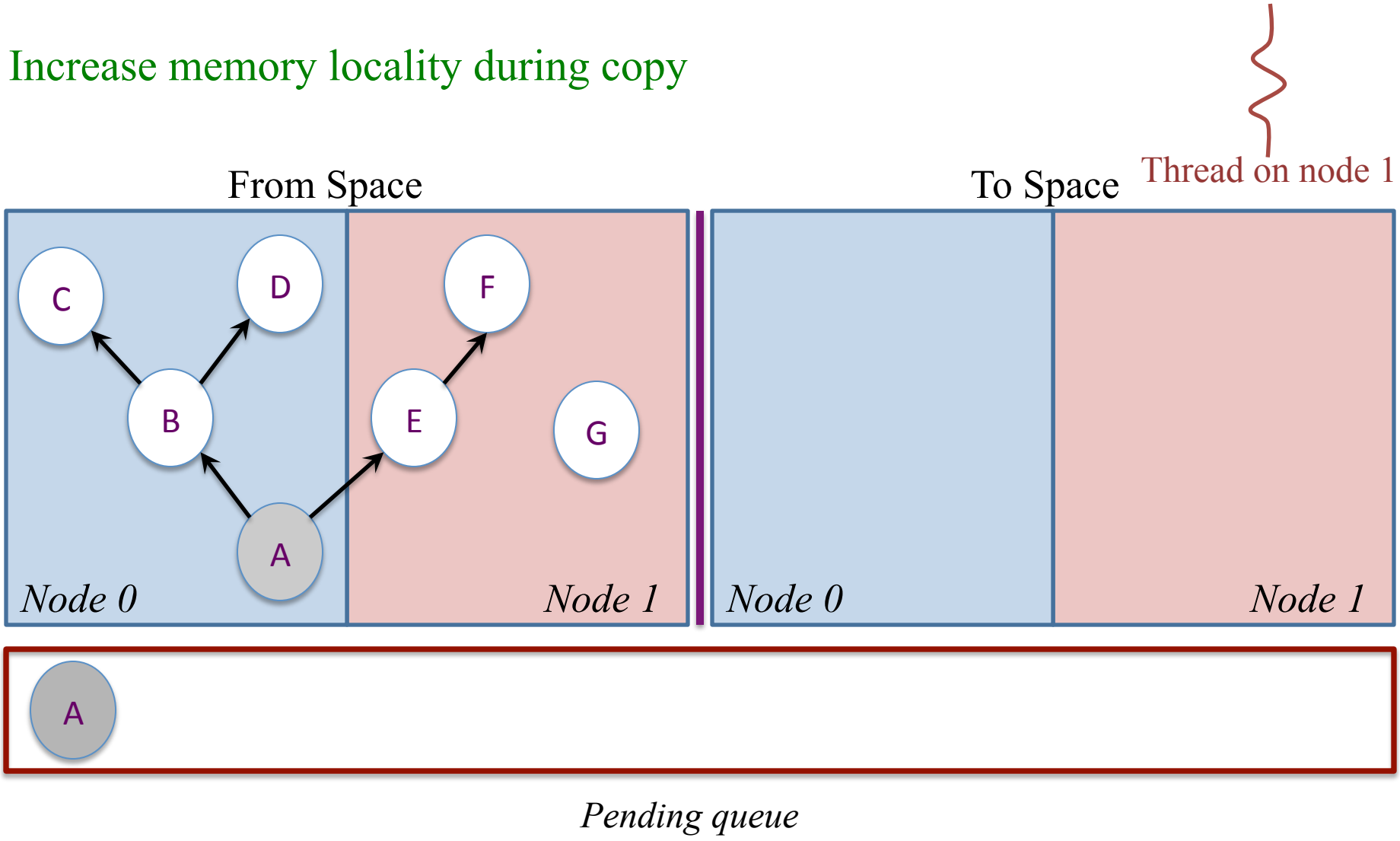
Solution 2: Fragmented Space

Increase locality for the application

- ✓ Mutator mostly accesses recently allocated objects
- ✓ Recently allocated object is on the mutator's node

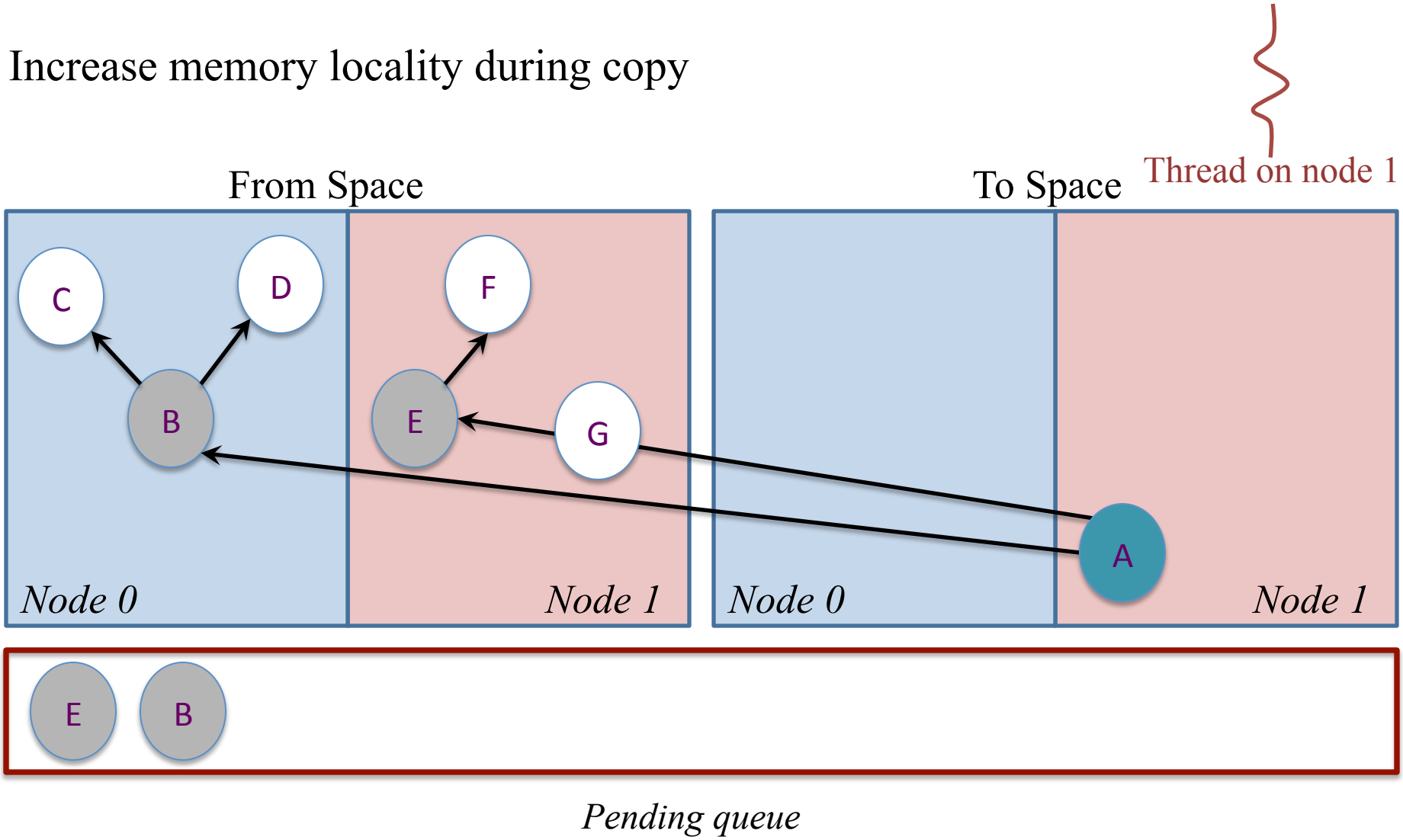
Solution 2: Fragmented Space

Increase memory locality during copy



Solution 2: Fragmented Space

Increase memory locality during copy



Solution 3: Segregated Space

Fragmented space + node-local scanning

(Send remote references to the owner of the object)

Perfect locality for the GC

But have to pay the price of inter-node message exchanges

Good locality for the mutators

Mutator mostly accesses recently allocated objects

Natural balance of the load if allocation rates of the mutators are similar

Summary of the spaces

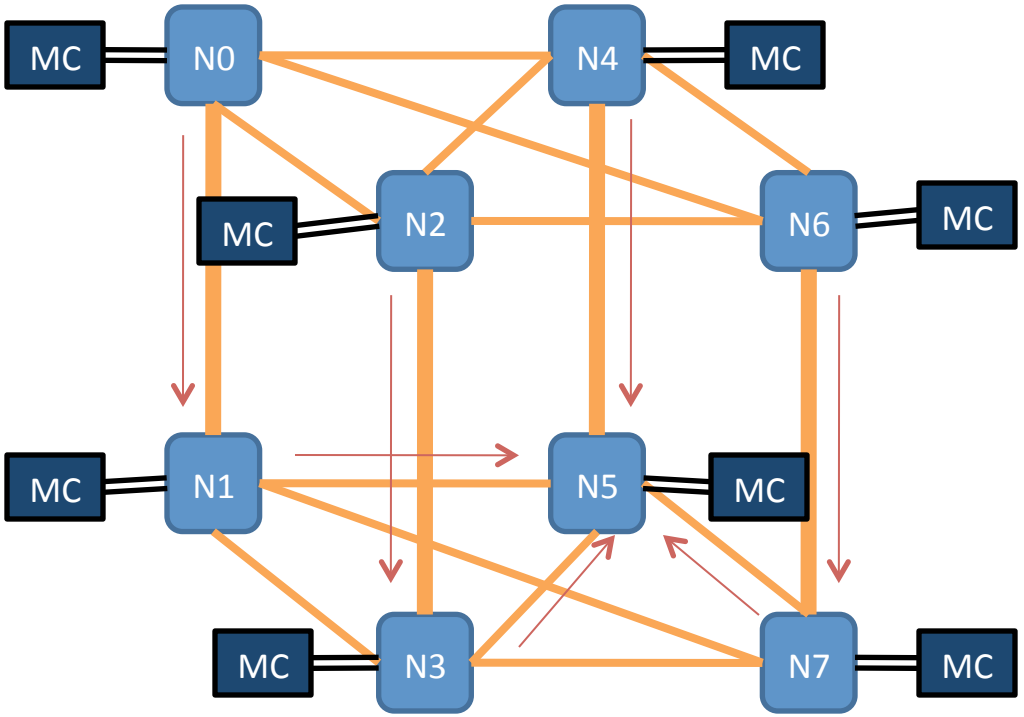
	Interleaved	Fragmented	Segregated
Memory Balance	Perfect	Good	Good
Mutator Locality	Bad	Good	Good
GC Locality	Bad	Good	Perfect
Comment			Inter-node messages

1. Background
2. The lock bottleneck
3. The NUMA bottleneck
4. Evaluation
5. Conclusion

Evaluation

Hardware:

- ✓ AMD Opteron 6172 sockets
- ✓ 8 nodes and 48 cores

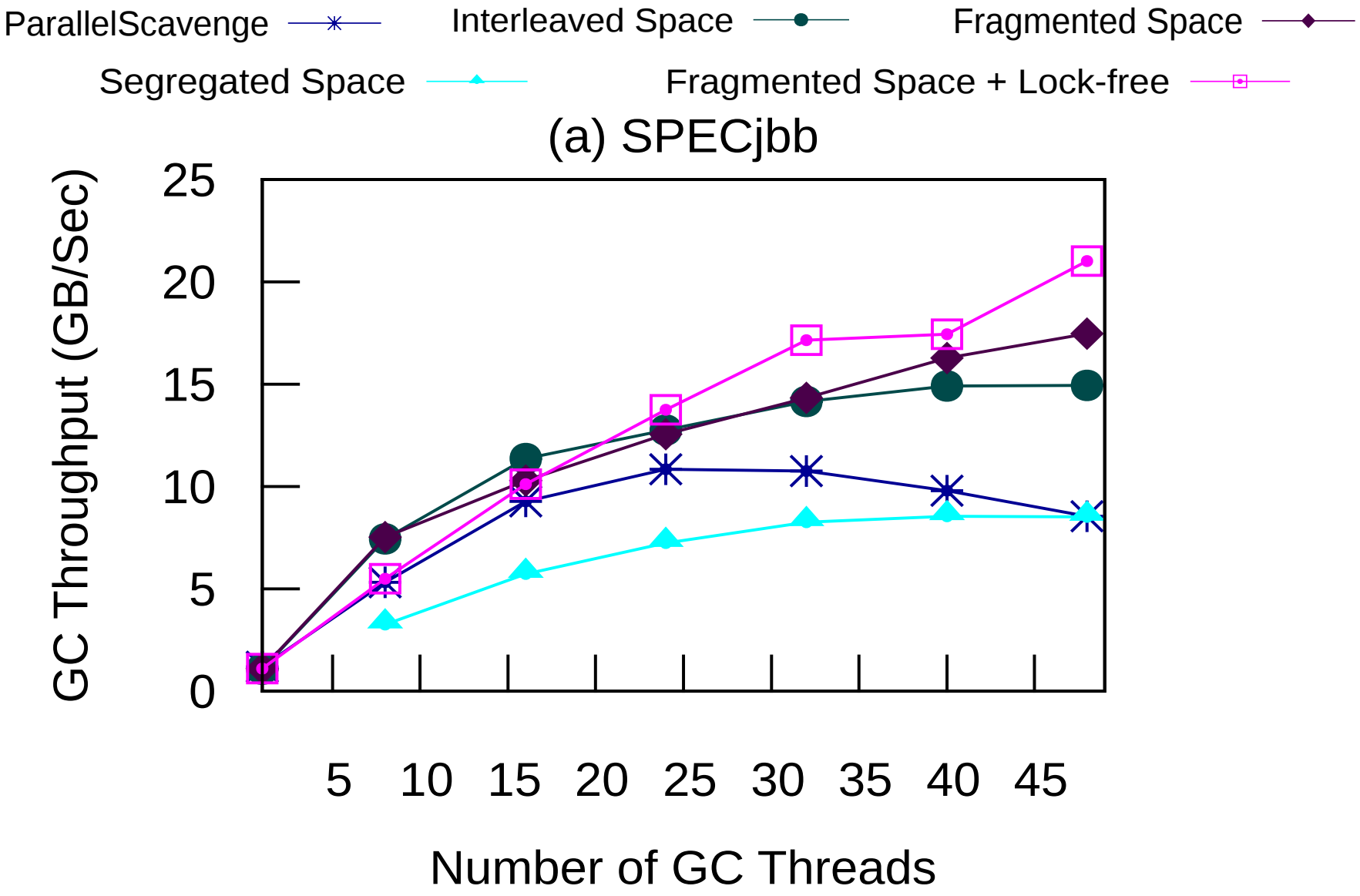


25 evaluated applications:

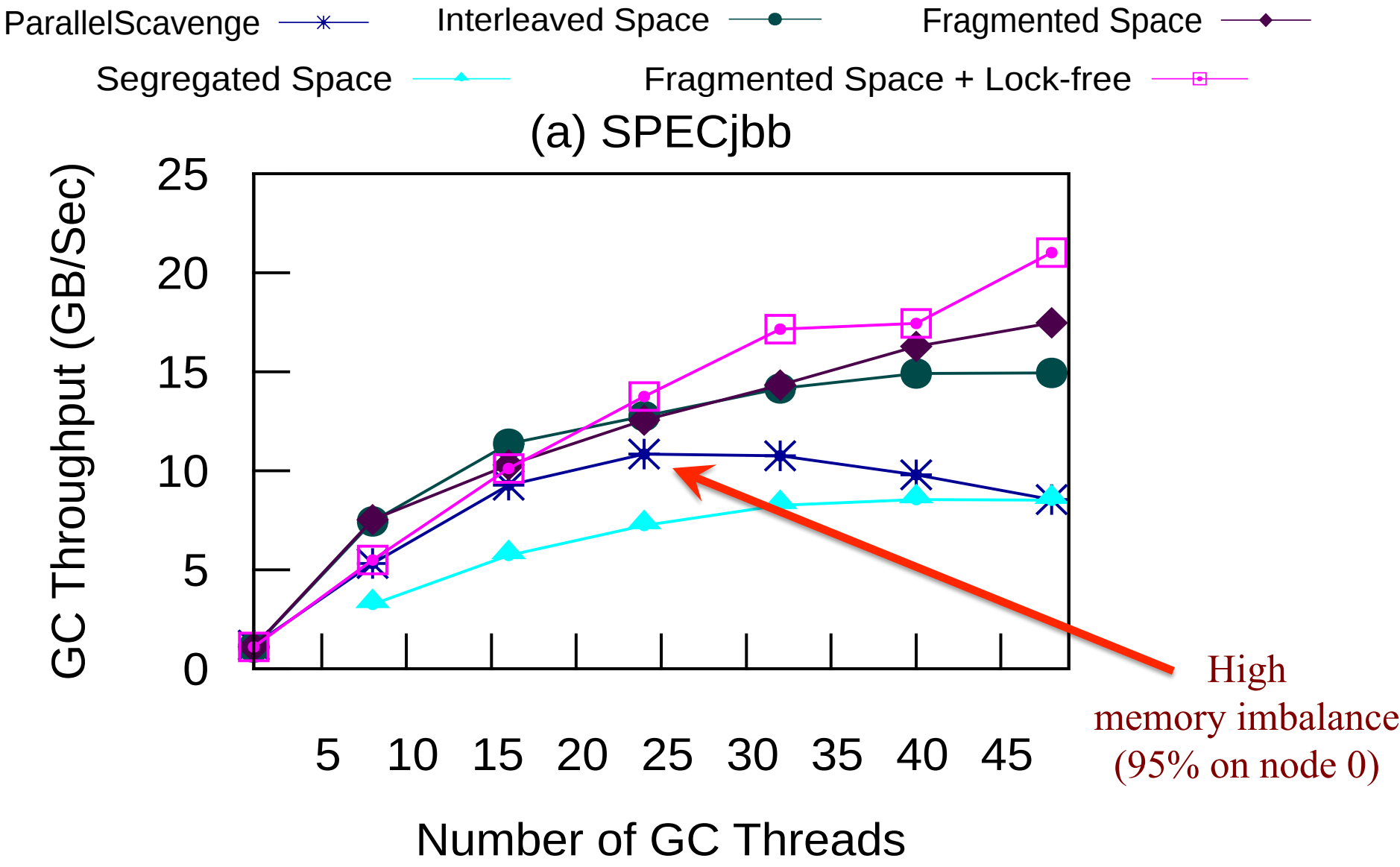
- ✓ SPECjbb2005: ~ 3.5 GB
- ✓ SPECjvm2008: ~ 1 to 2 GB
- ✓ DaCapo 9.12: ~ 500

Focus on SPECjbb2005 in the presentation

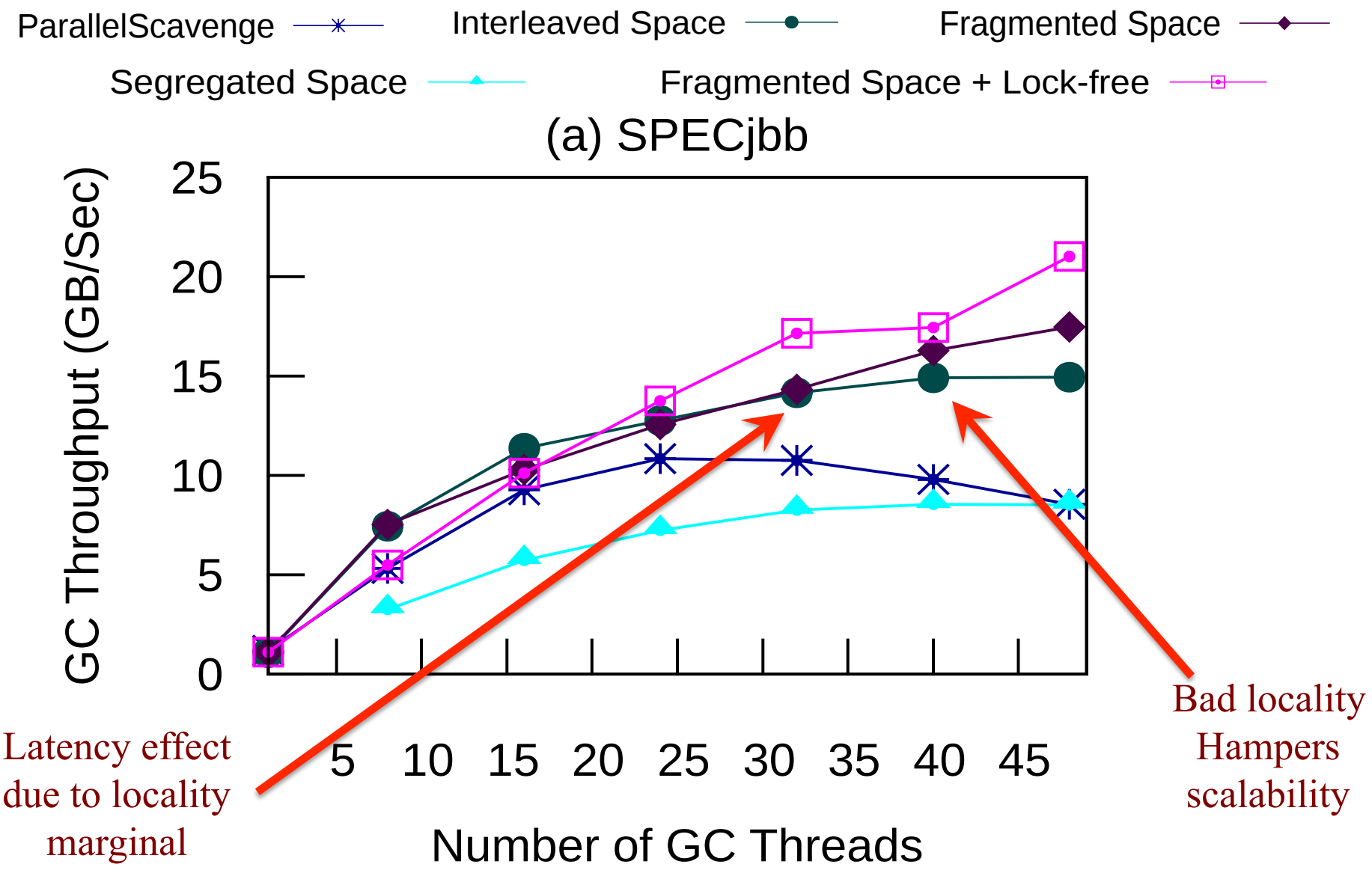
Effect of optimizations on the GC



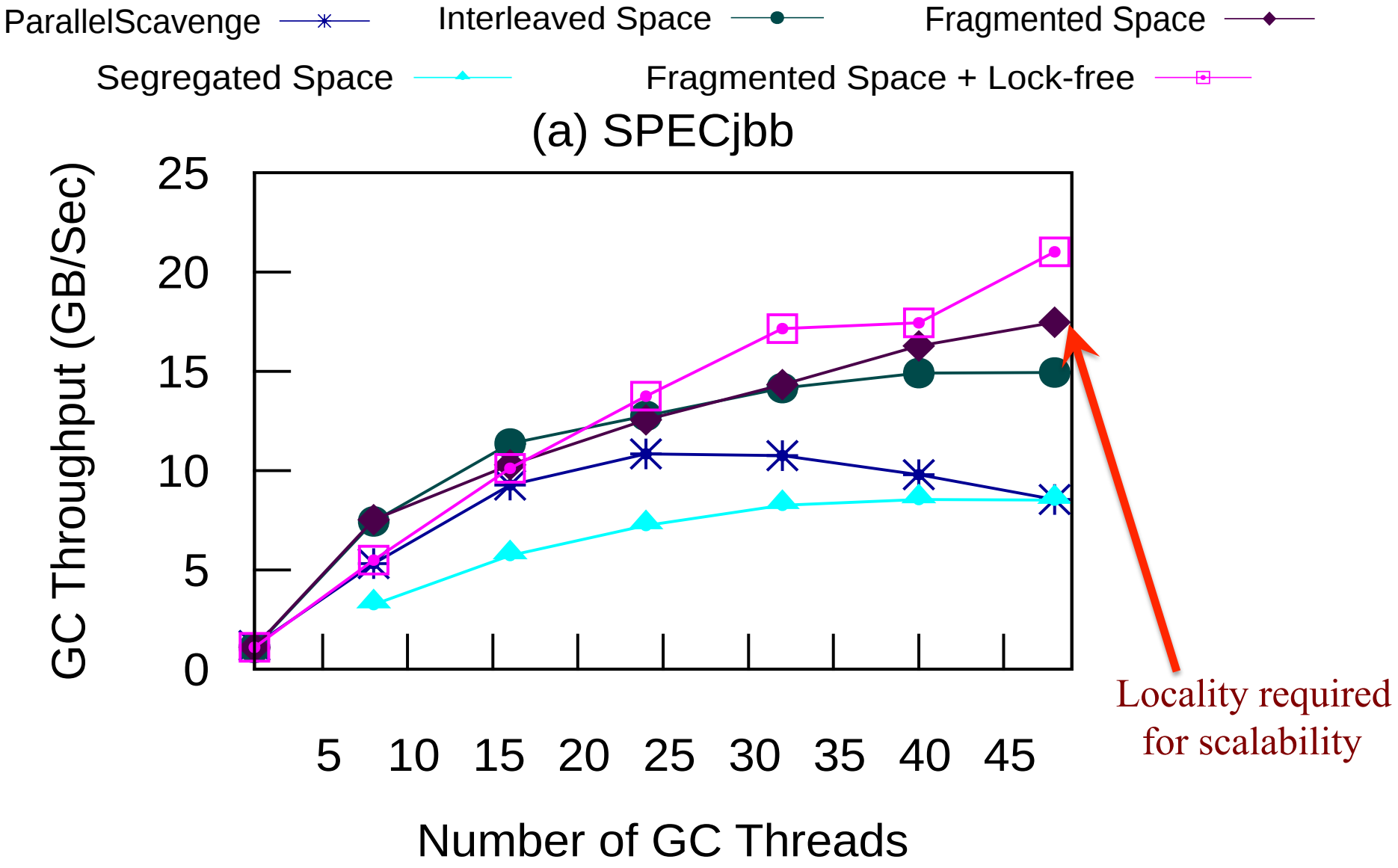
Effect of optimizations on the GC



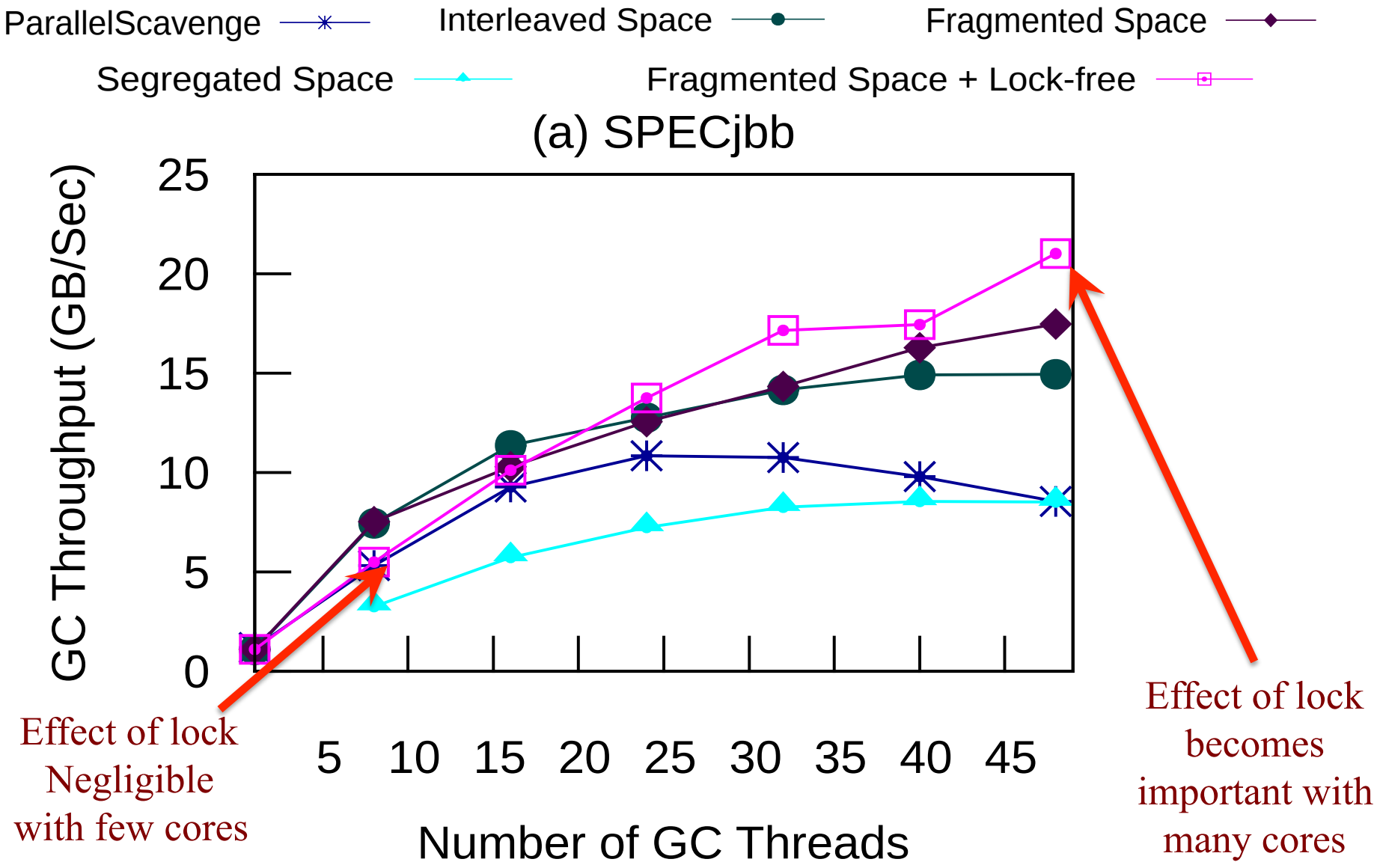
Effect of optimizations on the GC



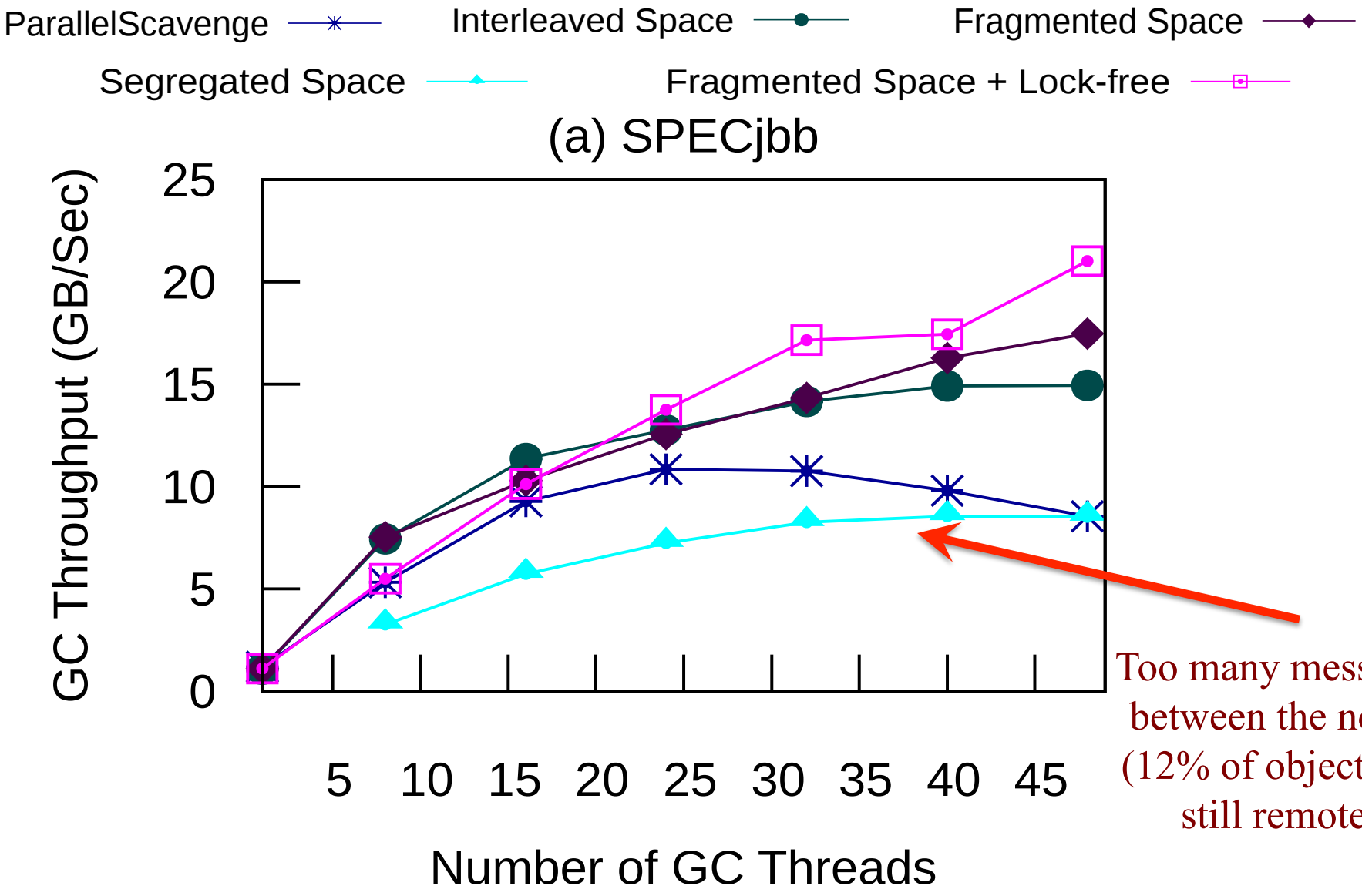
Effect of optimizations on the GC



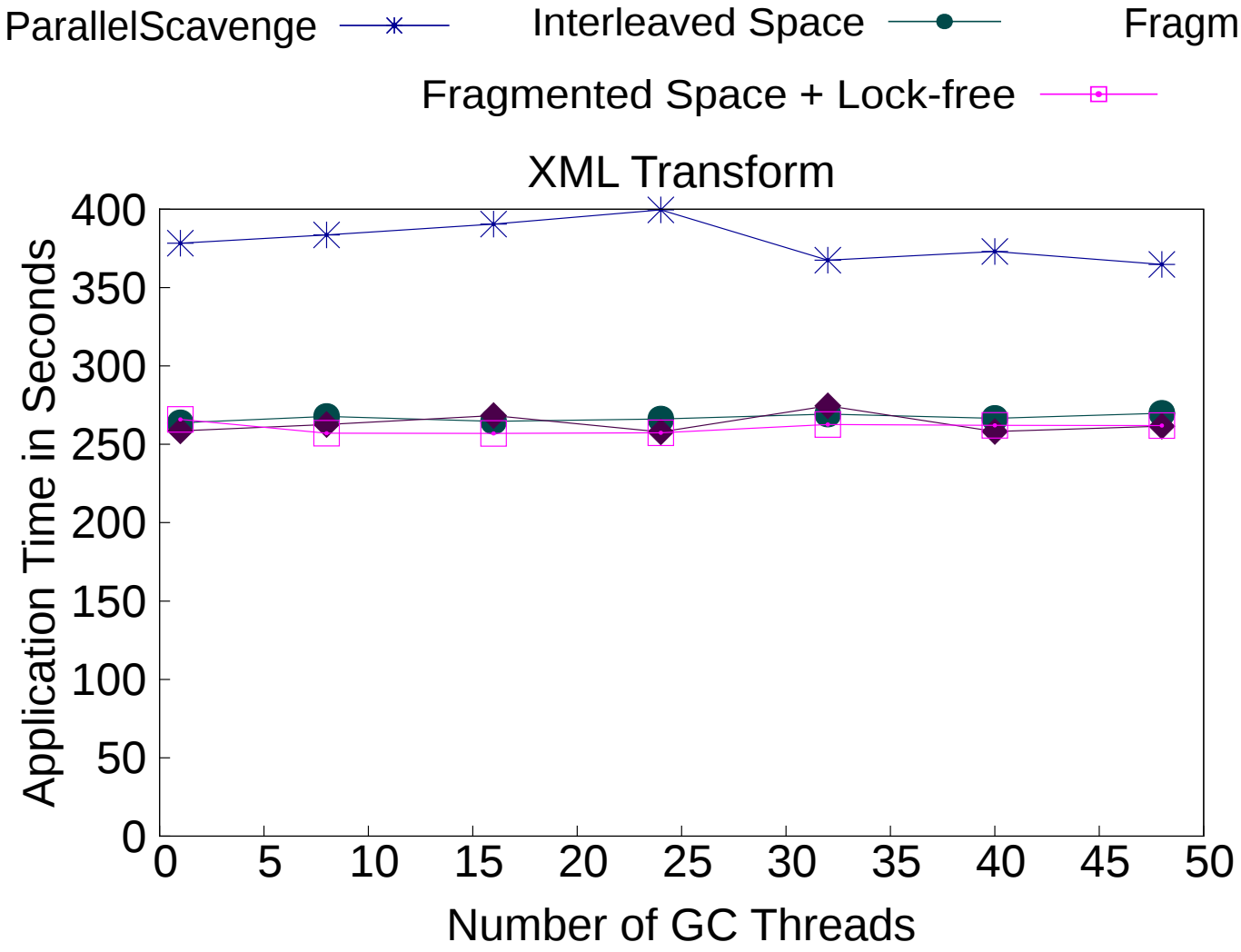
Effect of optimizations on the GC



Effect of optimizations on the GC



Effect of optimizations on the Application

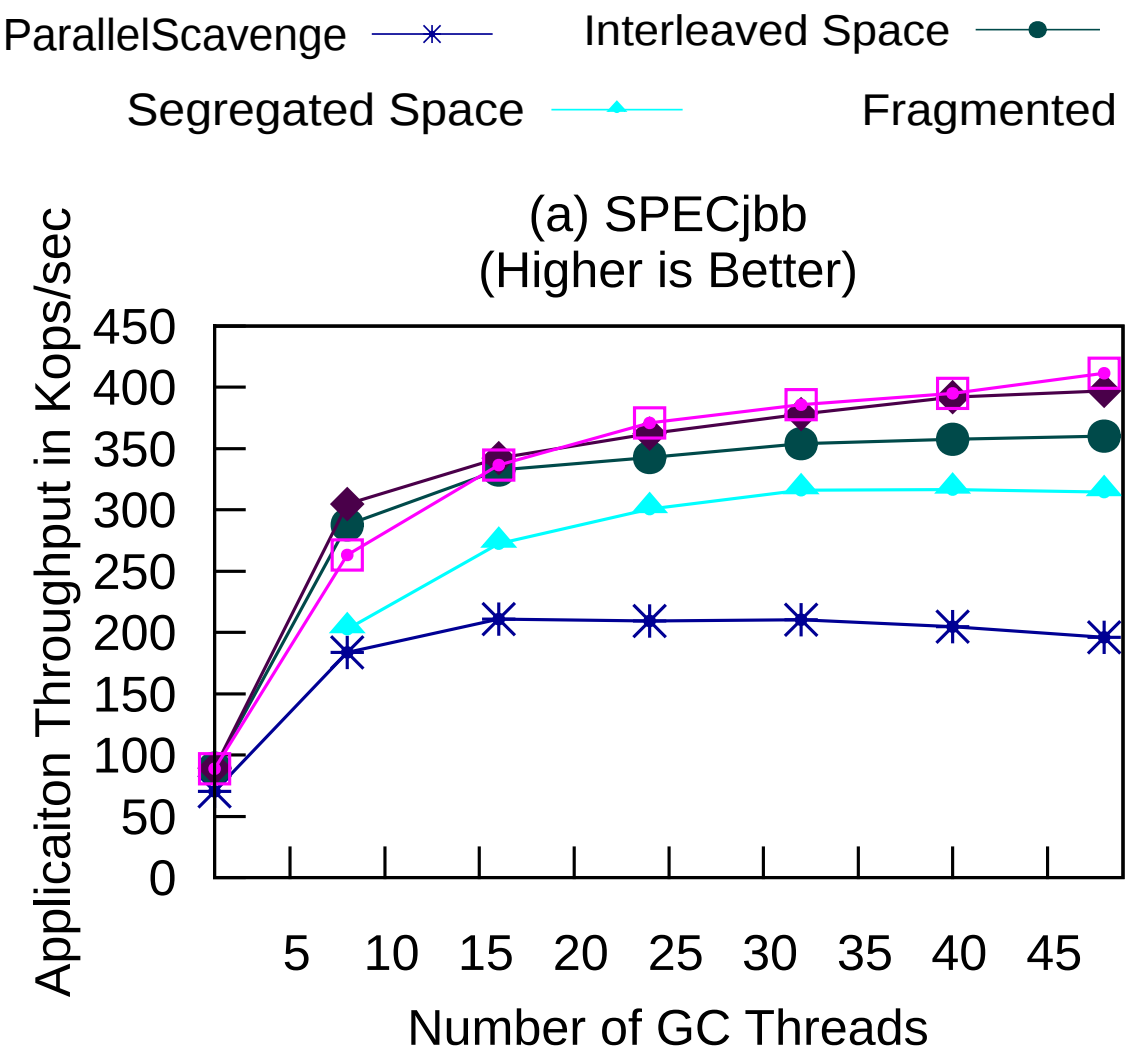


Balance ⇒ increase performance of app.

Locality ⇒ marginal effect

GC time excluded

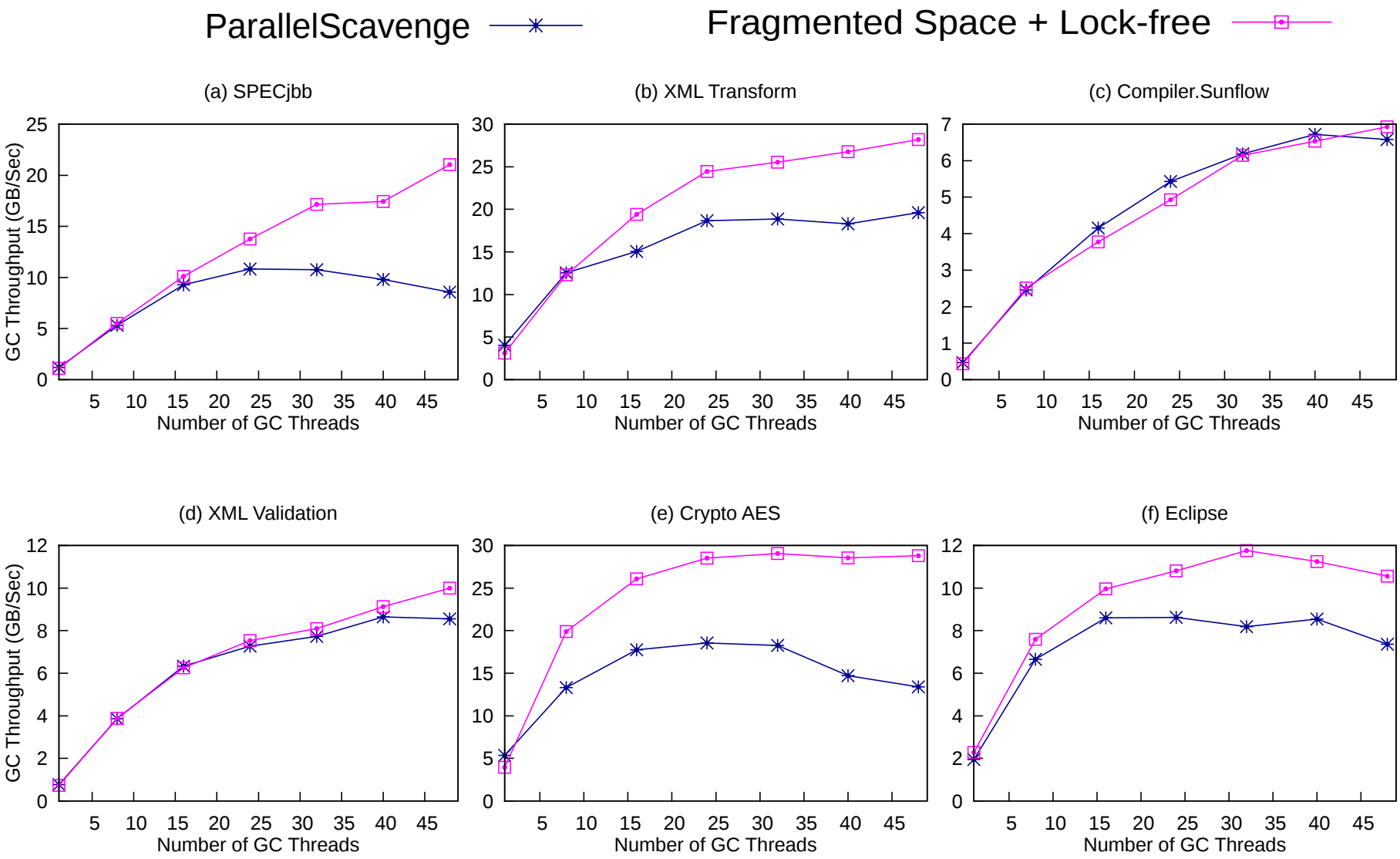
Overall effect



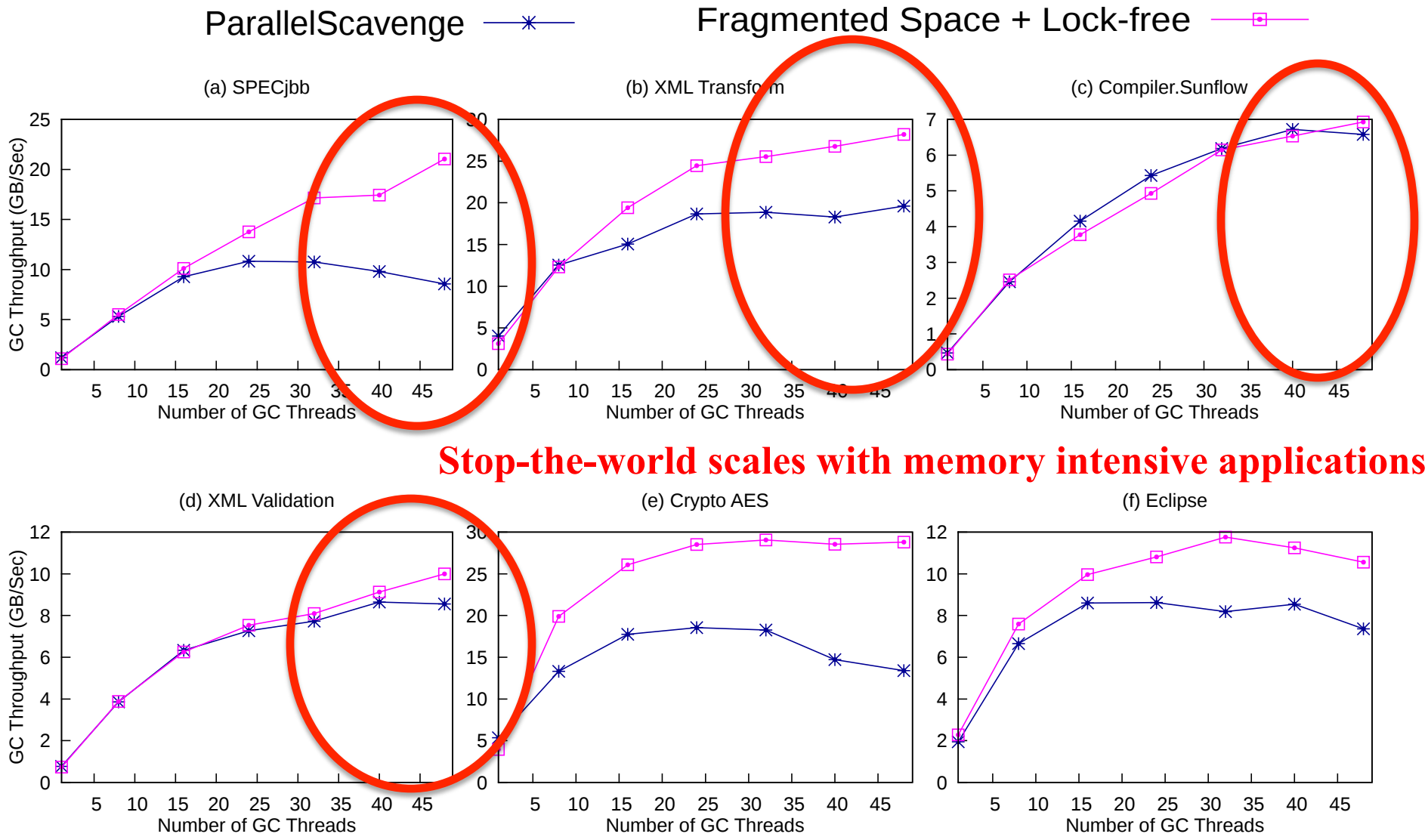
Optimizations translates into a x2 on throughput of SPECjbb

Pause time of 1 collection: from 105ms to 49ms

Scalability with the cores

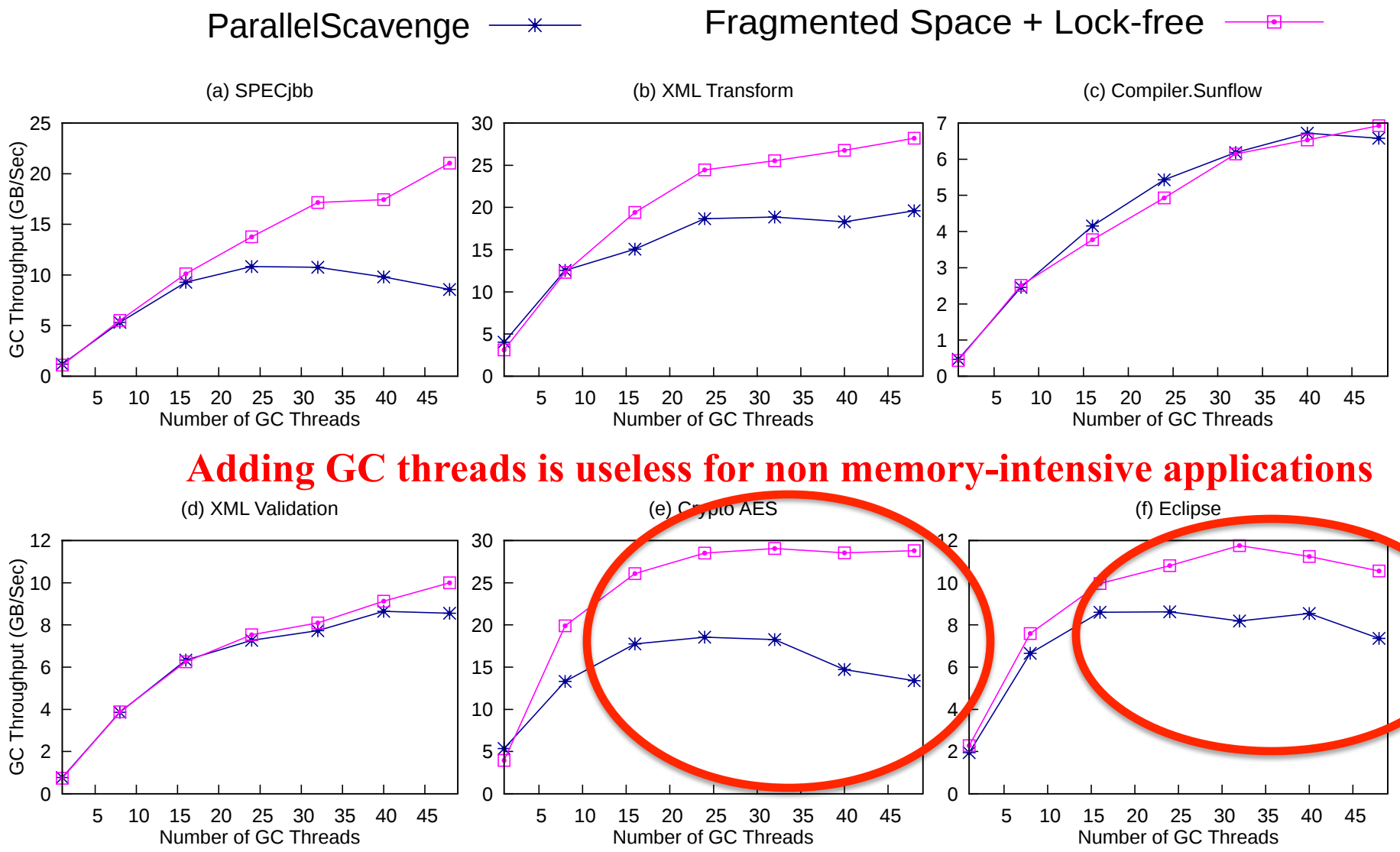


Scalability with the cores



Stop-the-world scales with memory intensive applications

Scalability with the cores



Adding GC threads is useless for non memory-intensive applications

To take away

STW GCs assumed to be inherently non-scalable is probably a mistake

⇒ Stop-the-world GC still well suited for contemporary H/W

Most important NUMA effects

- ✓ Balancing memory accesses has the most important impact
- ✓ Increasing memory locality is required to scale
- ✓ Latency improvement due to locality negligible

Next step

- ✓ Avoiding most of the messages between the nodes

[A Study of the Scalability of Stop-the-world Garbage Collectors on Multicores, ASPLOS 2013]