Motivation

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

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
Outline

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

GCs don’t scale because machines are NUMA

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

Hardware hides the distributed memory
⇒ application silently creates inter-node references

GC thread

Node 0
Node 1
Node 2
Node 3

Memory

GC thread

Node 0
Node 1
Node 2
Node 3

Memory
GCs don’t scale because machines are NUMA

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

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

And continue the graph traversal locally

Lokesh Gidra
Using messages enforces local access…

…but opens up other performance challenges

Problem 1: a msg is costlier than a remote access

• 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
Problem 2: Limited parallelism

- Due to serialized traversal of object clusters across nodes

Node 0

Node 1

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Heap Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spark</td>
<td>In-memory data analytics (page rank computation)</td>
<td>Amd48: 110 to 160GB, Intel80: 250 to 350GB</td>
</tr>
<tr>
<td>Neo4j</td>
<td>Object graph database (Single Source Shortest Path)</td>
<td>Amd48: 110 to 160GB, Intel80: 250 to 350GB</td>
</tr>
<tr>
<td>SPECjbb2013</td>
<td>Business-logic server</td>
<td>Amd48: 24 to 40GB, Intel80: 24 to 40GB</td>
</tr>
<tr>
<td>SPECjbb2005</td>
<td>Business-logic server</td>
<td>Amd48: 4 to 8GB, Intel80: 8 to 12GB</td>
</tr>
</tbody>
</table>

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

Experiments

- Improved PS multiplies GC performance up to 5.4X
- NAPS multiplies GC performance up to 2.9X
- NumaGiC multiplies GC performance up to 5.4X

GC Throughput (GB collected per second)

- Improved PS
- NAPS
- NumaGiC

Heap Sizes

1 billion edge Friendster dataset
The 1.8 billion edge Friendster dataset
GC Throughput Scalability

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

<table>
<thead>
<tr>
<th>Application</th>
<th>Middleware</th>
<th>Language runtime</th>
<th>Operating system</th>
<th>Hypervisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data analytic</td>
<td>Hadoop, Spark, Neo4j, Cassandra...</td>
<td>JVM, CLI, Python, R...</td>
<td>Linux, Windows...</td>
<td>Xen, VMWare...</td>
</tr>
</tbody>
</table>

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