NUMA in High-Level Languages

Patrick Siegler
Non-Uniform Memory Architectures
Hasso-Plattner-Institut

Agenda

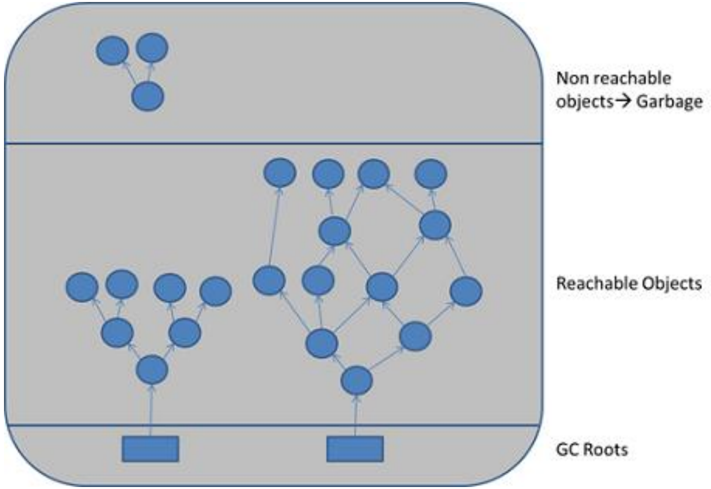
1. Definition of High-Level Language
2. C#
3. Java
4. Summary

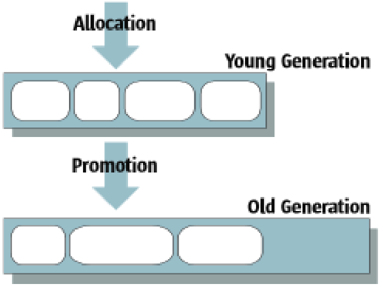
High-Level Language

- Interpreter, no directly machine executable format
- Platform Independence
- Automated Memory Management

GC - Short recap

- Traverse reference trees to find non-referenced objects
  - More than one GC root possible
- Reclaim space by moving referenced objects together
- Generational GC
  - many short-lived objects
  - old objects collected less freqently





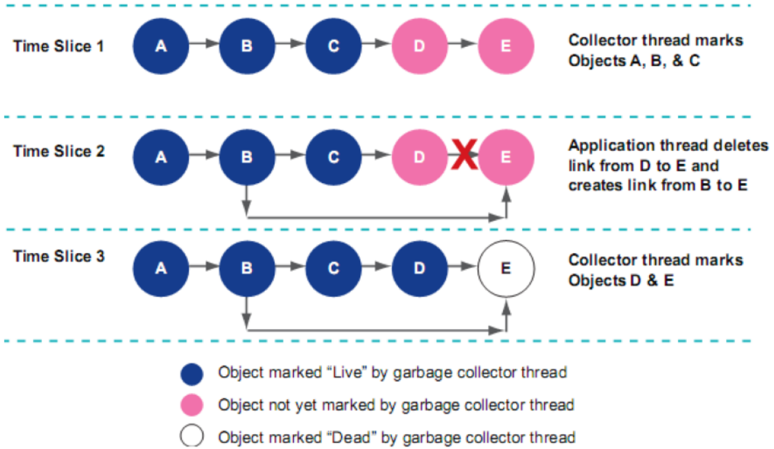
NUMA in high-level languages

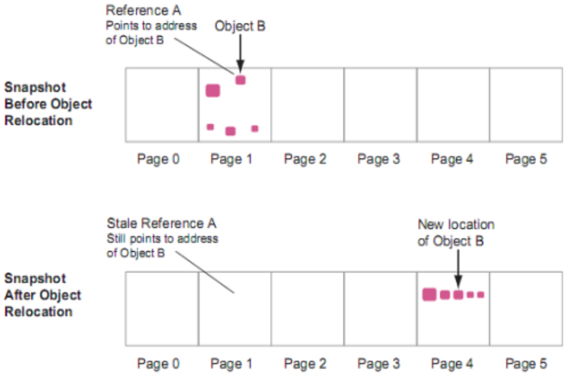
Patrick Siegler
12.12.2014

Concurrent GC

- Difficult on multi-threaded systems
  - Modification of references during scanning
  - Lock Contention around MM data structures
  - References may be outdated

- Stop-the-World at some point


Time Slice 1
A
B
C
D
E
Collector thread marks Objects A, B, & C
Time Slice 2
A
B
C
D
X
E
Application thread deletes link from D to E and creates link from B to E
Time Slice 3
A
B
C
D
E
Collector thread marks Objects D & E
● Object marked "Live" by garbage collector thread
● Object not yet marked by garbage collector thread
○ Object marked "Dead" by garbage collector thread



GC on NUMA Systems

- GC compacting is copying memory
  - Expensive across nodes

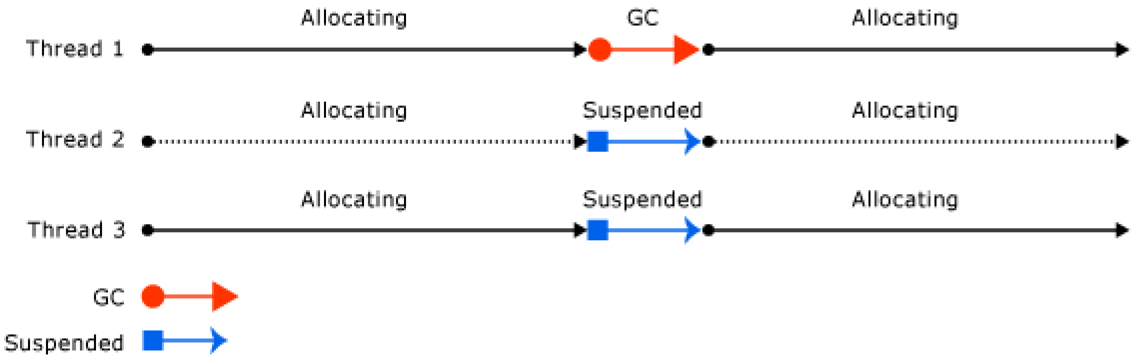
- Runtime faces same problem as OS: Who is going to use which memory?

- Young objects likely to stay on node

- Abstraction conflict
  - Programs do not want to care about hardware layout
  - Association of Threads / Tasks to nodes relevant for performance

C# - First Mutli-Processing Approach

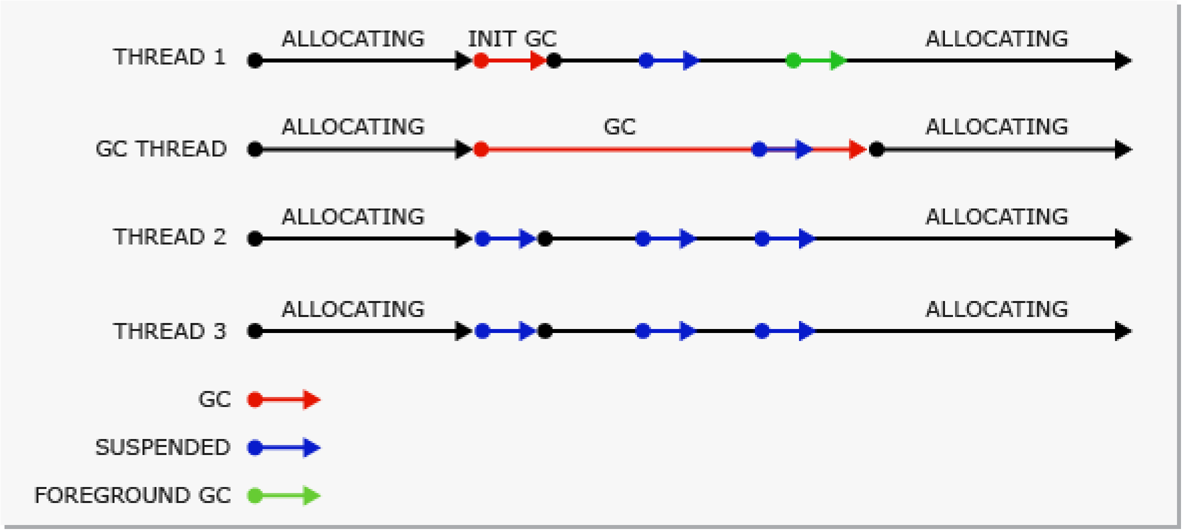
- Stop-the-World when needed



Program, badGC

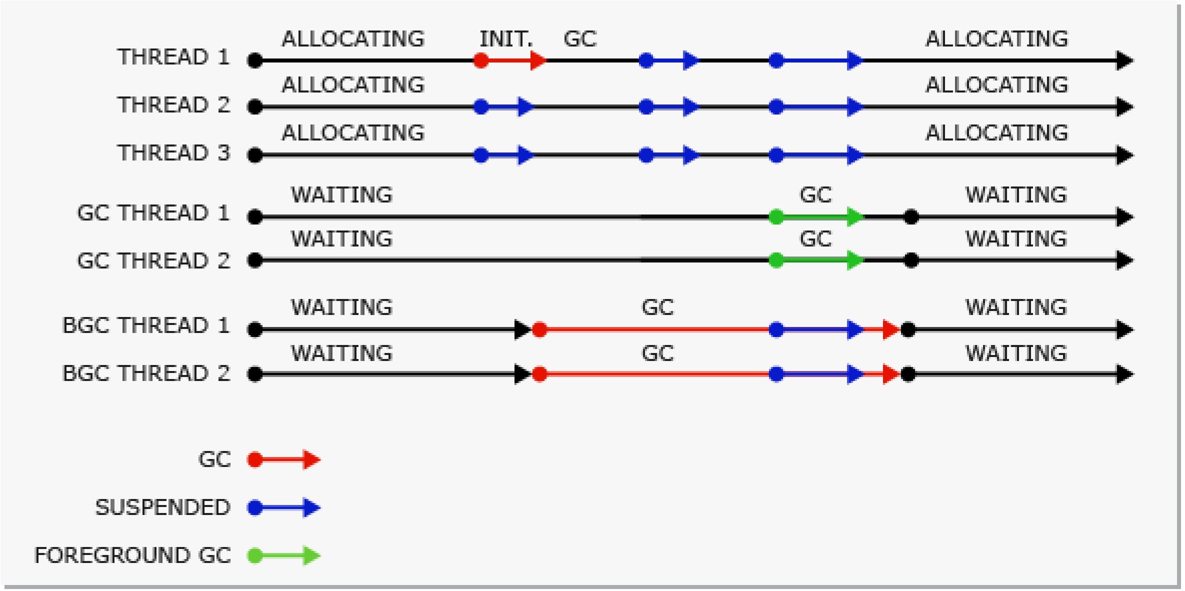
C# - Multi-Processing Enhancements

- Young generation collected per-thread “foreground”
- Old generation collected concurrently “background”



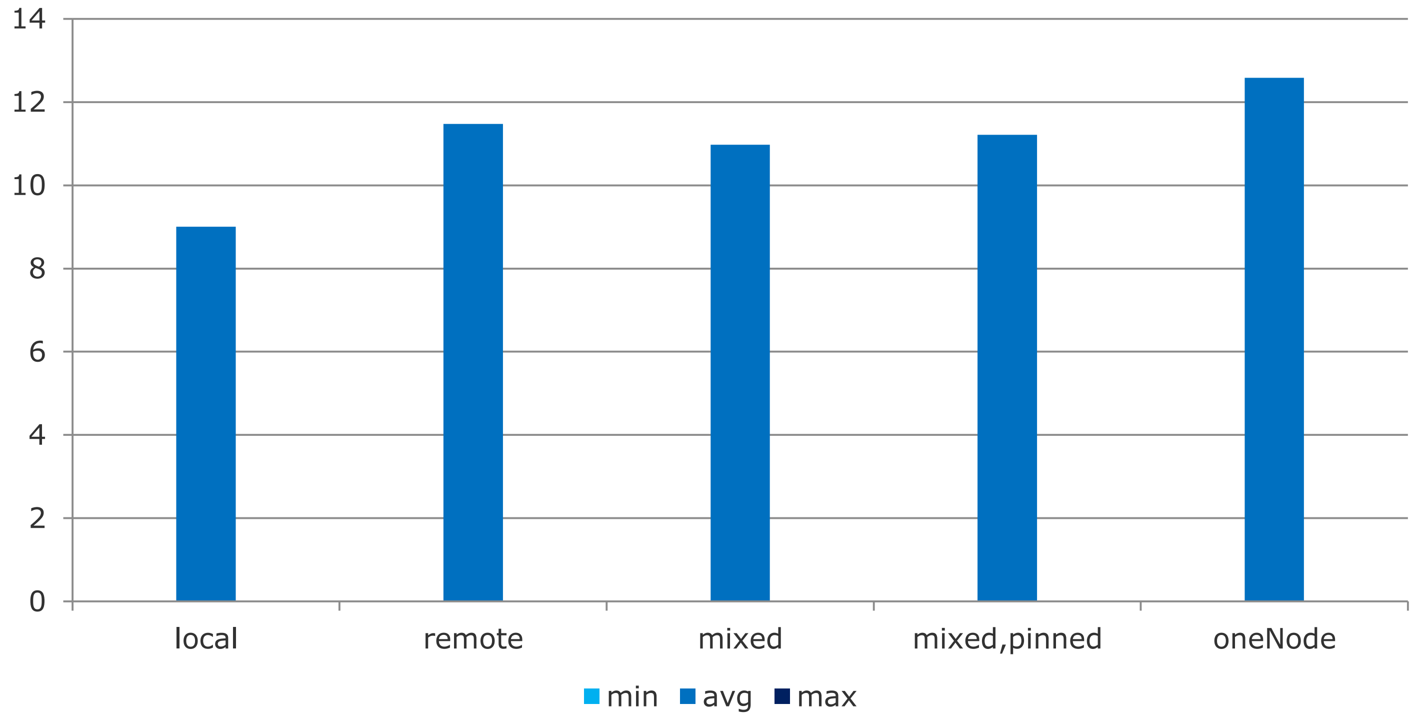
C# - Multi-Processing Enhancements

- “Server” GC uses dedicated high-priority threads
- 2 GC threads and a dedicated heap space per logical processor

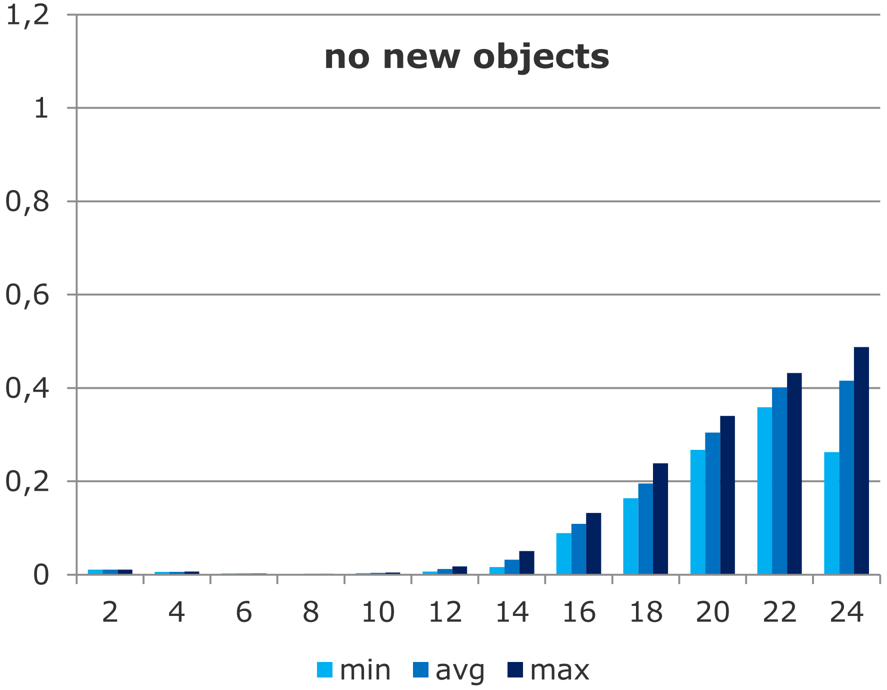


Jump, AllHelp

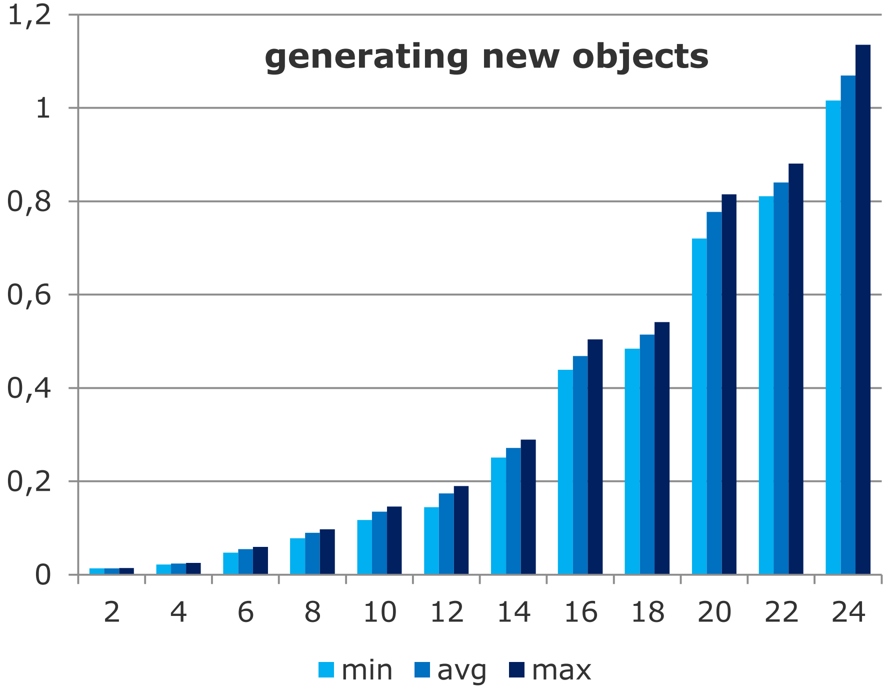
C# - Shared data access



C# - Cost of GC

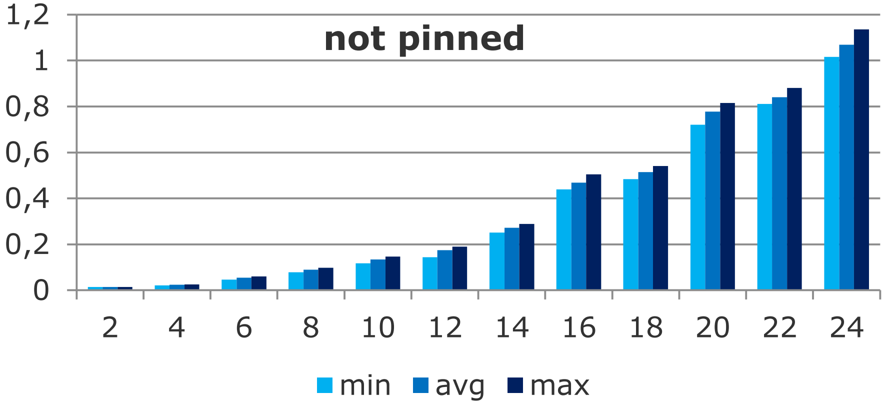


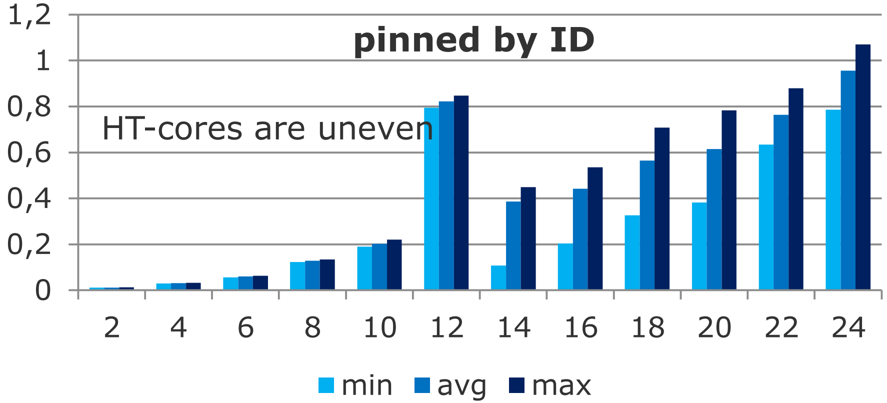
no new objects



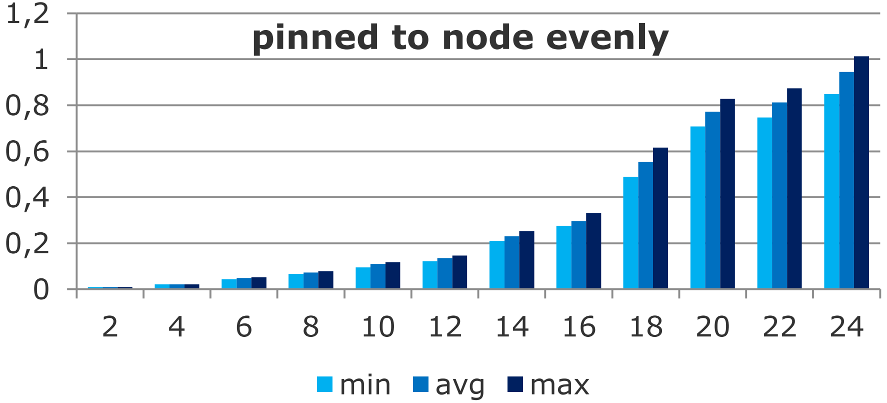
generating new objects

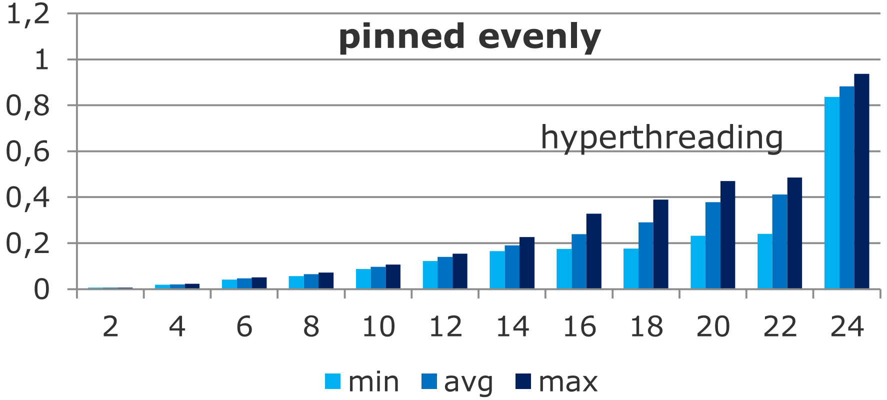
C# - Manually pinning





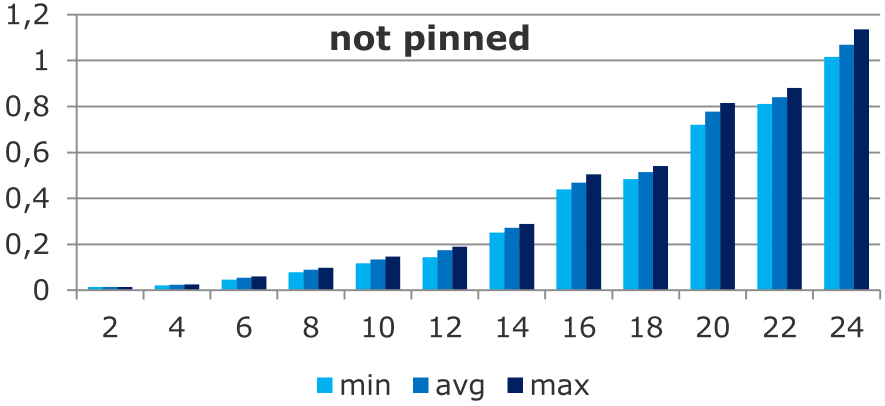
HT-cores are uneven

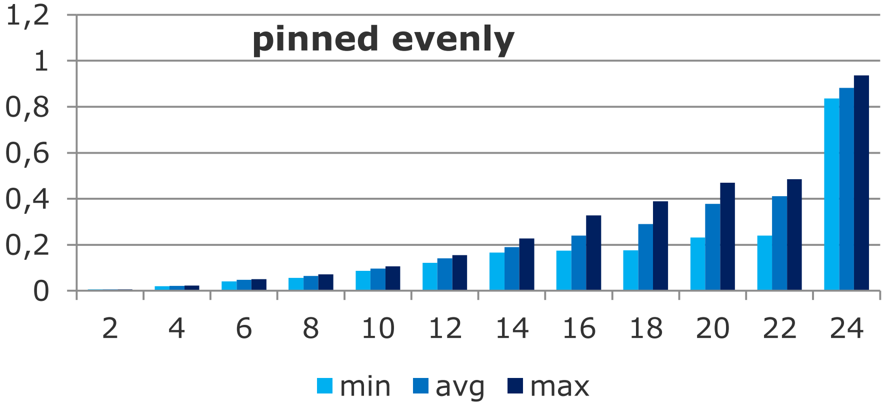


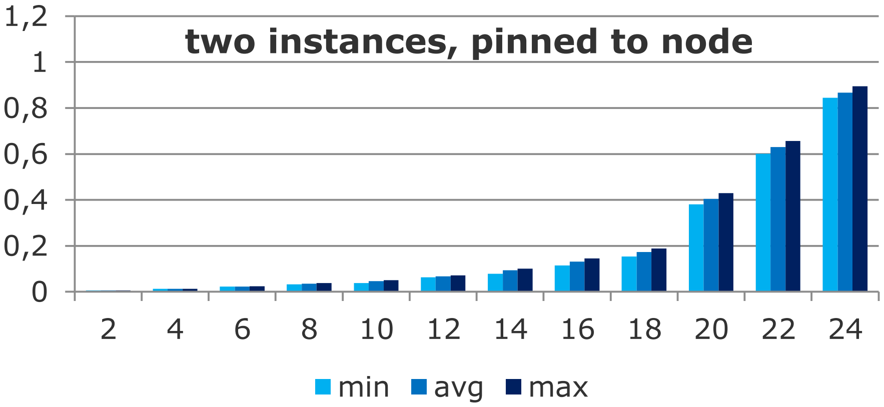


hyperthreading

C# - Single instance vs Two instances


not pinned


pinned evenly


two instances, pinned to node

C# - Summary

- Windows
  - Unpinned threads jump (away from their memory)
  - Natively pinned threads increase performance by >50%
  - Interconnect usage n/a on test system

- Linux
  - Mono’s GC seems to suffer from lock contention

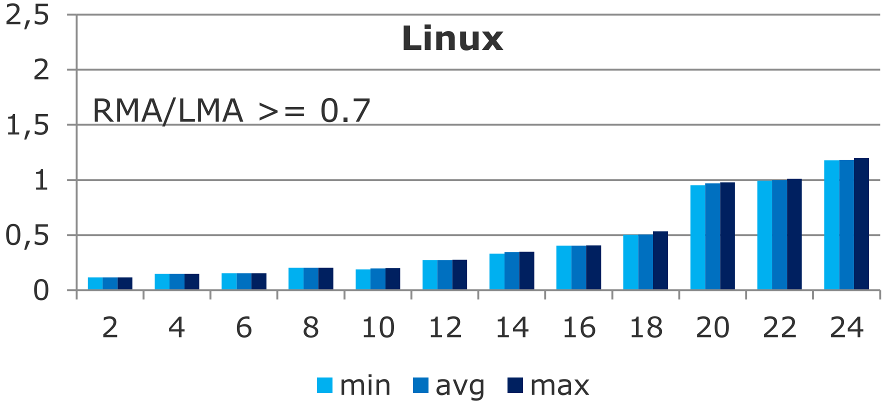
Java

- Various virtual machines offer many GCs with varying levels of concurrency

- Thread-Local Allocation Buffers (TLABs)
  - synchronization-free allocation
  - no NUMA-awareness

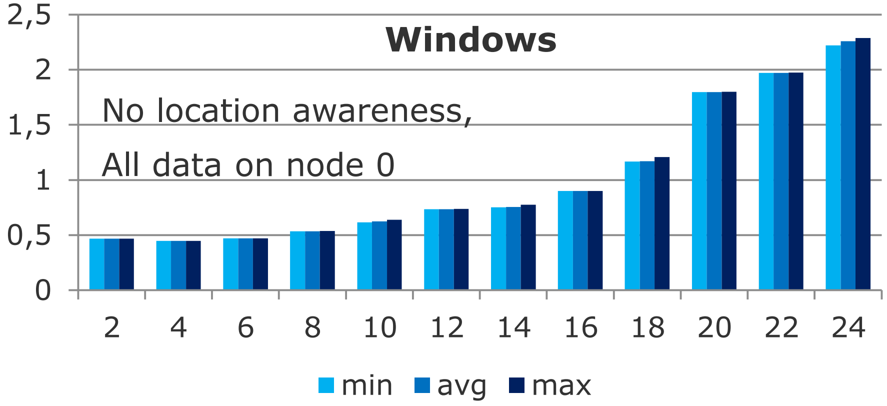
- Parallel Scanvenger GC (not concurrent, `-XX:+UseParallelGC`)
  - `-XX:+UseNUMA` since Java 6u2 (+40% in SPEC JBB 2005)
    - per-node regions
    - page interleaving for old and permanent generation
  - `-XX:+UseNUMAInterleaving` on Windows

Java - OS differences (per-thread data)



Linux

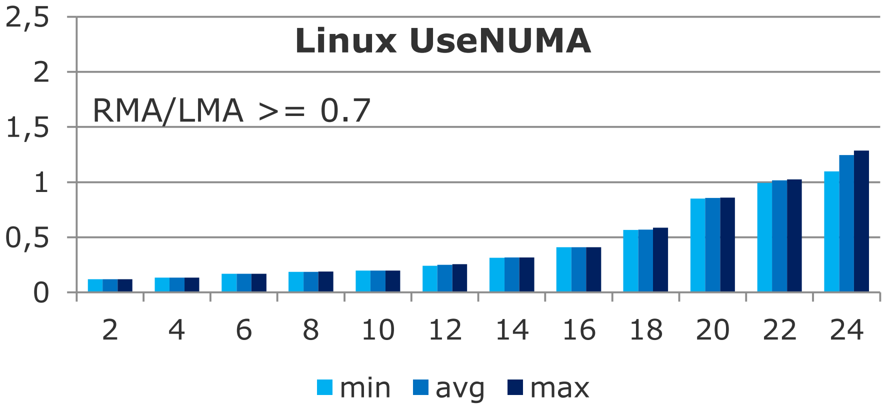
RMA/LMA >= 0.7



Windows

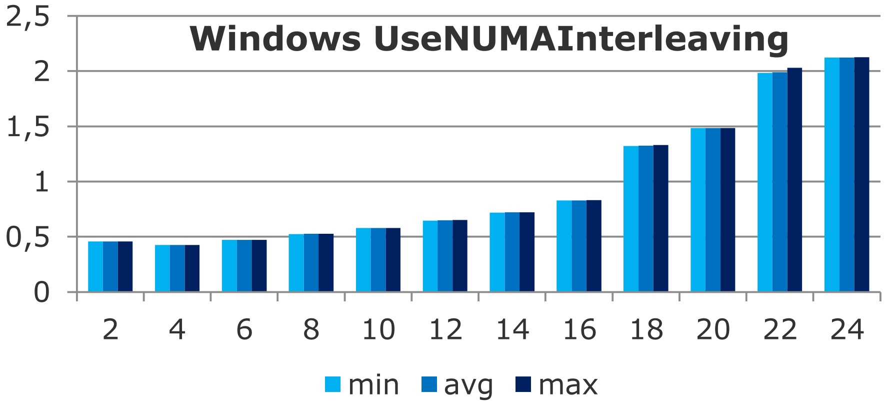
No location awareness,

All data on node 0



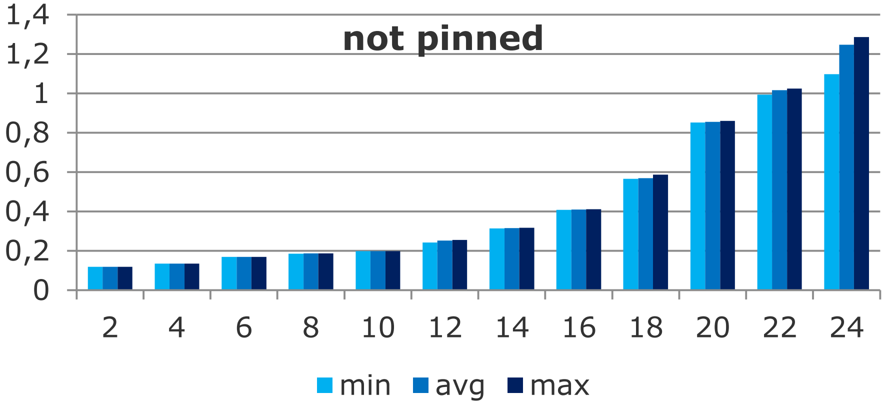
Linux UseNUMA

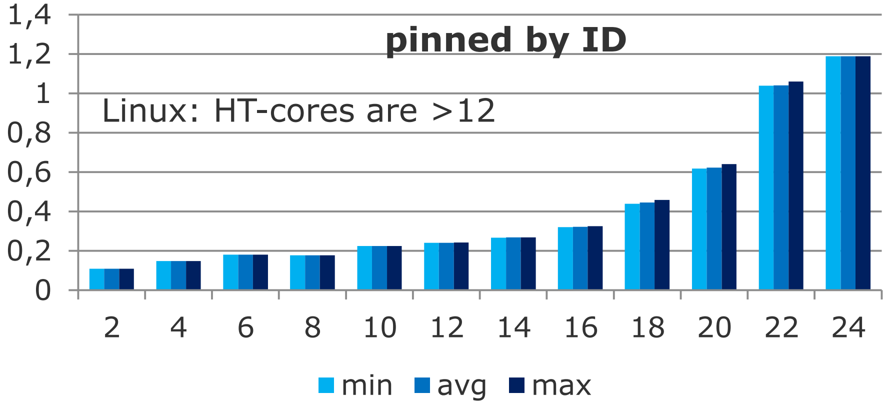
RMA/LMA >= 0.7



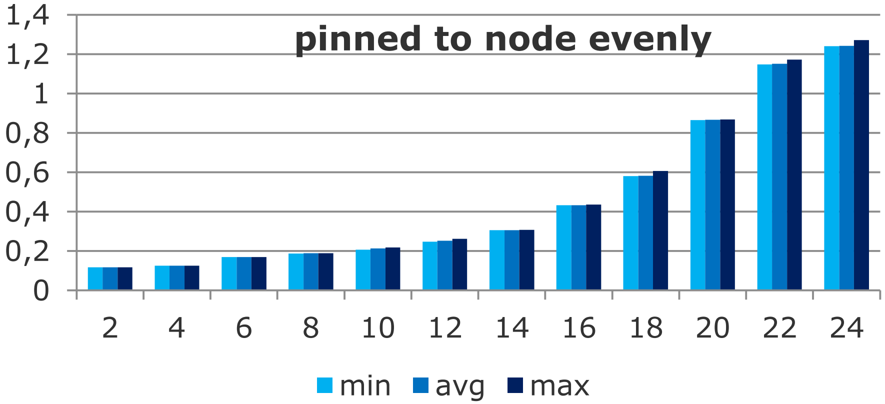
Windows UseNUMAInterleaving

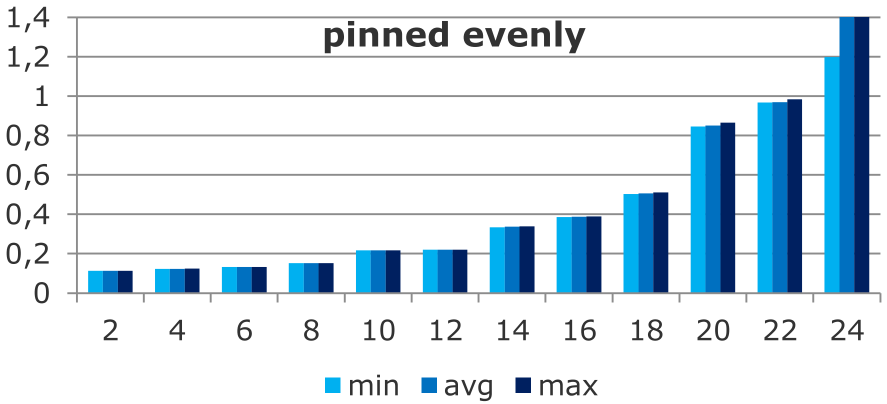
Java - manually pinning



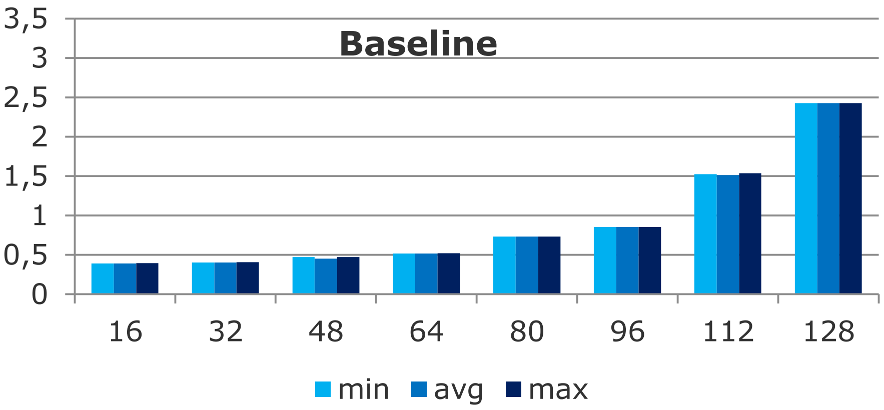


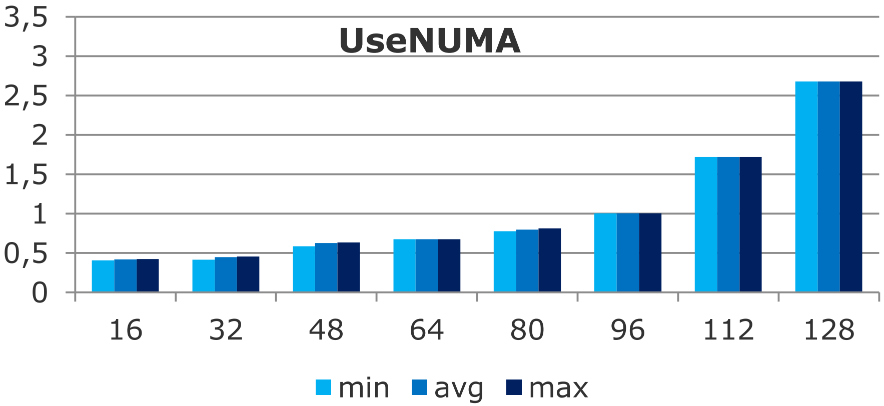
Linux: HT-cores are >12

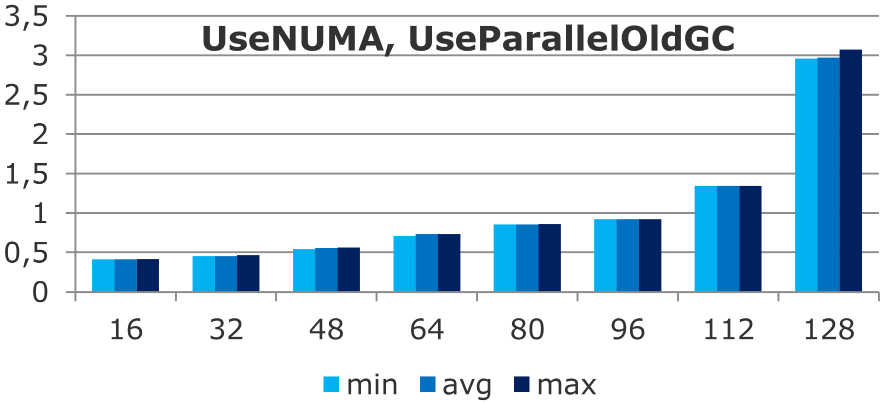




Java - 8 nodes


Baseline
3,5
3
2,5
2
1,5
1
0,5
0
16
32
48
64
80
96
112
128
min
avg
max


UseNUMA
3,5
3
2,5
2
1,5
1
0,5
0
16
32
48
64
80
96
112
128
min
avg
max


UseNUMA, UseParallelOldGC
3,5
3
2,5
2
1,5
1
0,5
0
16
32
48
64
80
96
112
128
min
avg
max

Java - Summary

- Windows
  - Limited support, only interleaving old generation
  - Unpinned threads jump

- Linux
  - Threads jump less often
  - numatop: >0.7 with UseNUMA and each thread has own data

Summary

- No explicit API in high-level languages

- Potentially catastrophic consequences of choice of GC

- Newer GC mechanisms are “NUMA-aware”
  - New allocations happen in a node-local buffer
  - Old generations are interleaved between all nodes
  - Only GC is optimized

- Varying support on different OSes

- Under-commiting allows GC to operate concurrently

Future and unexplored issues

- Improving OS schedulers will also apply to high-level languages
- Tracing and performance counters to determine which memory is used
- Actual low-level instruction flow and hyperthreading
- Actual layout of objects in memory esp. after compactation
  - Sufficient size should reduce caching effects
- Different JVM implementations

Sources

- http://blogs.msdn.com/b/dotnet/archive/2012/07/20/the-net-framework-4-5-includes-new-garbage-collector-enhancements-for-client-and-server-apps.aspx
- http://msdn.microsoft.com/en-gb/library/ee787088.aspx#workstation_and_server_garbage_collection
- http://msdn.microsoft.com/en-us/magazine/cc163552.aspx

- http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html
- www.azulsystems.com/sites/default/files//images/wp_pgc_zing_v5.pdf
- http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf
- http://developer.amd.com/community/blog/2011/10/27/a-new-numa-option-for-windows-on-the-hotspot-jvm/
- https://blogs.oracle.com/jonthecollector/entry/help_for_the_numa_weary
- http://javabook.compuware.com/content/memory/how-garbage-collection-works.aspx