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 frequently

NUMA in high-level languages
Patrick Siegler
12.12.2014
Chart 4
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  - References may be outdated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

NUMA in high-level languages
Patrick Siegler
12.12.2014
Chart 5
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 Multi-Processing Approach

- Stop-the-World when needed

NUMA in high-level languages
Patrick Siegler
12.12.2014
Chart 7
Program, badGC
C# - Multi-Processing Enhancements

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

NUMA in high-level languages
Patrick Siegler
12.12.2014
Chart 8
C# - Multi-Processing Enhancements

- “Server” GC uses dedicated high-priority threads
- 2 GC threads and a dedicated heap space per logical processor
C# - Shared data access

NUMA in high-level languages

Patrick Siegler
12.12.2014

Chart 10
MemTravels
C# - Cost of GC

no new objects

generating new objects
C# - Manually pinning

Not pinned

Pinned by ID

Pinned to node evenly

Pinned evenly

HT-cores are uneven

Hyperthreading
C# - Single instance vs Two instances

**not pinned**

- Min
- Avg
- Max

**pinned evenly**

- Min
- Avg
- Max

**two instances, pinned to node**

- Min
- Avg
- Max
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
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**

- RMA/LMA >= 0.7
Java - manually pinning

- **not pinned**
- **pinned by ID**
- **pinned to node evenly**
- **pinned evenly**

Linux: HT-cores are >12
Java - 8 nodes

![Bar charts showing performance comparison between Baseline, UseNUMA, and UseNUMA, UseParallelOldGC with varying node counts and performance metrics.](image-url)
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-committing 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 compaction
  - Sufficient size should reduce caching effects

- Different JVM implementations