X86_64 Architecture: NUMA Considerations
Outline:

- X86_64 basic Architecture
- NUMA and ccNUMA Architectures
- NUMA performance issues
- Memory allocation mechanisms
- NUMA Policy
- NUMA API
- Thread placement API
- Tools: likwid, numactl
- Memory access patterns
- C++ Containers
X86 Architecture

- On Each Clock Cycle:
  - One Float-pt Add
  - One Float-pt Multiply
  - Store address generation
  - Load address generation
  - Branch

- If operands are available!
So what is NUMA?

Numa, Numa, Numa...
No, not this guy!
- NUMA: Non-uniform Memory Access
- ccNUMA: Cache coherent NUMA
8 MB L3 cache

For all applications to share

Inclusive cache policy to minimize traffic from snoops
Non-Uniform Memory Access (NUMA)

2005: “Fake” dual-core
- Chipset
- Memory

2006: True dual-core
- Chipset
- Memory

Woodcrest
- “Core2 Duo” 65nm
- Chipset
- Memory

Harpertown
- “Core2 Quad” 45nm
- Chipset
- Memory

2008: Simultaneous Multi Threading (SMT)
- Chipset
- Memory

Nehalem EP
- “Core i7” 45nm

2010: 6-core chip
- Chipset
- Memory

Westmere EP
- “Core i7” 32nm

2012: Wider SIMD units
- Chipset
- Memory

Sandy Bridge EP
- “Core i7” 32nm
- AVX: 256 Bit
NUMA Policy

- `p = malloc(4*4096); // 4096 is pagesize`

Before memory is written, virtual pages map to a single zero page.
When memory is written, a page fault occurs and the system remaps to physical RAM. But where?
NUMA Policy

- Default: Physical location 'nearest' to faulting core (i.e. first touch)
- Implies initialization should be done in same core pattern as processing
NUMA Policy

- Interleaved: Pages are distributed among NUMA domains
- Attempts to amortize access costs
- Use case: shared memory with random write access patterns
NUMA API – System calls

- **set_mempolicy()**
  - Sets the general NUMA policy (process)
  - Does not include mbind'ed memory
  - Does not include current memory
  - Default, bind, preferred, interleave
  - Takes a list of core id's

- **mbind(ptr, len, policy, coremask, mode)**
  - More specific than set_mempolicy()
  - Can move existing memory NUMA domain
**NUMA API**

- `move_pages(pid, count, pages, nodes, status, flags)`
  - Can move existing memory
- In all cases, granularity is `pagesize` (4096)
- NUMA policy inherited by child processes
Thread Placement API

- `sched_setaffinity(pid, cpusetsize, cpuset)`
- `pthread_setaffinity_np(pthread_id, ...)`
- Affinity setting inherited by child process
- Child can only further narrow available cores – Unless it has `CAP_SYS_NICE`
Higher level library for NUMA control query functions, memory policies and process binding

See: man numa(3)
NUMA API

- **numa_num_possible_nodes()**
  - Number of NUMA nodes

- **numa_bind()**
  - Sets task affinity and memory domain preference

- **numa_set_localalloc()**
  - Sets a policy which attempts to bind physical memory to the allocating node (as opposed to first touch node)

- **numa_alloc*()**
  - Like *alloc(), but is slower and always rounds up to multiple of pagesize.

- **numa_distance()**
  - Returns 'distance' or 'cost' of accessing remote node memory. Returns cost in multiples of 10.
Command Line Tools

- numactl – included in most distributions
- likwid – open source project

Examples:
  - likwid -topology
  - likwid-pin
C++ Containers -- Footprints

- Array, Vector – contiguous
- Map – RB tree
- List – list
- Deque – tree of blocks?
- Hashmap(C++11) – Hash table/vector
- Sets – RB trees
C++ Containers -- Attributes

Some thoughts about each data structure:

- `std::list` is very slow to iterate through the collection due to its poor spatial locality.
- `std::vector` and `std::deque` perform better than `std::list` with small sized elements.
- `std::list` efficiently handles very large elements.
- `std::deque` performs better than a `std::vector` for inserting at random positions (especially at the front, which is constant time).
- `std::deque` and `std::vector` are not the best choice for data types with high cost of copy/assignment.
C++ Containers - Which one to use?

This draw simple conclusions on usage of each data structure:

- **Number crunching**: use `std::vector` or `std::deque`
- **Linear search**: use `std::vector` or `std::deque`
- **Random Insert/Remove**: use `std::deque` (if data size is very small [typically < 64B], otherwise use `std::list`)
- **Big data size**: use `std::list` (not if intended for searching)
- **Push to front**: use `std::deque` or `std::vector`
The End
Optimizations

- Multi-threading
- Vectorization
  - FP precision (i.e: double vs. float)
  - Integer/Fixed point
- Const; Arrays instead of pointers
- Loop unrolling
- Remove loop conditionals
- Ordering of conditionals
Optimizations

- Use static function decorator (C/C++)
- Use anonymous namespace (C++)
Tony's Program

- Single threaded ~2+ hours
- Added OpenMP parallel for
  - Required atomic sections
  - Using all cores (16) ~90min
- Replaced log10() function
  - Using 16 cores, ~55min
Tony's Program

- Changed conditional:
  - \( \log_{10}(x) < \text{constant} \)
  - \( X < \exp_{\text{constant}} \)

- Refactored to remove atomic sections
  - With 16 cores, \( \sim 12 \) minutes (10:1)