Supporting Document

NUMA-Aware Memory Allocator

Devesh-Kartik-Padmnabh

August 30, 2025

Introduction

This document supports the presentation of our project: a **NUMA-Aware Memory Allocator**. The primary motivation behind this work is to solve a critical performance bottleneck that occurs in modern multi-socket servers due to the non-uniform distribution of memory access times.

The following sections expand upon the problem, our proposed solution, the testing methodology, as well as the tools and technologies that supported our implementation.

Problem Statement

The Hardware Context

Modern high-performance servers are often built as **multi-socket systems**, meaning they contain multiple physical CPUs. These CPUs follow the **NUMA (Non-Uniform Memory Access)** architecture.

In NUMA systems, each CPU owns a bank of local memory that it can access quickly. Accessing memory belonging to another CPU introduces higher latency due to slower interconnects between nodes.

Analogy

Consider an office building with two towers:

- Fetching coffee from the kitchen in your own tower is fast (*local memory access*).
- Walking to the other tower for coffee takes longer (*remote memory access*).

The delay experienced when accessing remote memory is what we call **non-uniform latency**.

The OS Allocation Policy

Operating systems such as Linux follow a simple rule known as the **"first-touch" policy**. Memory is not physically allocated until a CPU core first writes to it. At that moment, the memory is allocated to the NUMA node of the accessing CPU.

This creates a problem:

- Suppose a program initializes a large dataset on a CPU in Node 0.
- The data is placed on Node 0's memory because of the first-touch rule.
- Later, a worker thread on Node 1 accesses the data. It now incurs constant slow remote memory accesses.

Thus, traditional memory allocation is **blind to NUMA topology**, leading to sub-optimal performance in multi-threaded, multi-node workloads.

Proposed Solution: The Smart Allocator

Design Philosophy

We developed a user-space library that acts as a **NUMA-aware allocator**. By intercepting allocation requests, our solution explicitly controls where memory is physically allocated, thereby overcoming the limitations of the first-touch policy.

Implementation Methodology

The allocator is injected into programs using **LD_PRELOAD**, which allows overriding of the system `malloc` without modifying application code.

The allocation process follows a **Controlled First-Touch Strategy**:

1. **Reserve Virtual Memory:** Use `mmap` to reserve address space.
2. **Spawn a Helper Thread:** Create a temporary worker thread.
3. **Pin the Thread:** Lock the helper thread to a CPU core on the target NUMA node (via CPU affinity).
4. **Perform the First-Touch:** The thread touches each memory page, forcing the OS to allocate physical memory from the target node's local bank.

Once the process is complete, the helper thread exits and the memory pointer is returned to the requesting program.

Testing and Verification

Benchmark Setup

To validate our approach, we designed a benchmark program with two threads, each pinned to different NUMA nodes. Both threads perform compute-intensive tasks on large memory blocks.

Testing Scenarios

1. **Baseline (System malloc):** Memory placement follows the first-touch policy. Both allocations may end up on one node, forcing remote accesses for one thread.
2. **Our Allocator:** Each thread's memory is explicitly placed on its local NUMA node.

Metrics and Tools

- **Execution Time:** Primary metric to measure performance improvement.
- **numastat:** Used to verify actual memory distribution across NUMA nodes.

The allocator demonstrated measurable improvements in execution time, while `numastat` confirmed correct memory placement.

Tools and Technologies

The implementation of our allocator relied on the following tools and technologies:

- **Linux OS:** Provides NUMA system calls and LD_PRELOAD support.
- **C Programming Language:** For implementing low-level memory management logic.
- **pthread Library:** To create and manage helper threads.
- **mmap System Call:** For reserving virtual memory regions.
- **CPU Affinity APIs:** To bind helper threads to specific NUMA nodes.
- **numastat Utility:** For memory allocation verification.

Development & Testing Plan

The project will be developed in distinct phases:

- **Phase 1: Interception & Passthrough:** Implement the `LD_PRELOAD` mechanism to successfully intercept `malloc` calls and simply pass them through to the real `malloc` function. This verifies the core interception works.

- **Phase 2: NUMA Detection Policy:** Integrate `libnuma` to detect the number of nodes and implement the round-robin logic. Log the chosen node for each allocation.
- **Phase 3: Kernel-Level Allocation:** Replace the passthrough to the real `malloc` with direct `mmap`/`mbind`/`munmap` calls to implement the full allocation and deallocation logic.
- **Phase 4: Verification Debugging:** Create a benchmark program. Use the `/proc/PID/numa_maps` filesystem to manually verify that memory is being placed on the correct nodes as intended.
- **Phase 5: Performance Analysis (Stretch Goal):** If access to a true multi-node system is available, use `taskset` to benchmark the performance difference between local and remote memory access scenarios.

Conclusion

This project tackles the inefficiency of the first-touch policy in NUMA systems with a practical software-based solution. By combining low-level system calls with controlled first-touch allocation, we provide explicit programmer control over memory placement.

Our NUMA-aware allocator shows that careful awareness of hardware topology can lead to significant performance improvements in real-world applications.