Lecture 11: Non Uniform Memory Access Architecture

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in
Lecture 11: Non Uniform Memory Access Architecture

Last Lecture (Recap)

Thread 1

std::atomic<bool> A(false), B(false);
int non_atomic=0;

// Memory Operations MO1
A.store(true, memory_order_release);

Thread 2

if(B.load(memory_order_acquire) == true) {
    // Memory Operations MO2
    assert(non_atomic == 10)
}

std::atomic<int> X, Y, Z;

// Some memory operations
X.store(1, memory_order_seq_cst);
// Some memory operations
Y.load(memory_order_relaxed);
// Some memory operations
Z.load(memory_order_acquire);
// Some memory operations

Thread 1

std::atomic<bool> X(false), Y(false);

// Memory Operations-1 (MO1)
X.store(true, memory_order_relaxed);
Y.store(true, memory_order_relaxed);

Thread 2

// Memory Operations-2 (MO2)
if(Y.load(memory_order_relaxed) == true) {
    assert(X.load(memory_order_relaxed))
}

std::atomic<bool> Flag(false);

Thread 1

Memory Operations-1 (MO1)
PushTask();
Flag.store(true);

Thread 2

MO1 happens-before MO2
if(Flag.load() == true) {
    PopTask();
}

MO1 visible to
Thread-2 before it performs MO2
Memory Operations-2 (MO2)
Today’s Lecture

- Non uniform memory accesses
- Page allocation policies
- NUMA aware parallel runtime
Recall: Virtual VS. Physical Memory

- Programs refers to virtual memory addresses
  - Memory can be thought of large array of bytes
  - System provides private address space to each of the processes

- Problems
  - How does the memory for all the processes fit?
    - Virtual address can address exabytes (64-bit), whereas physical memory ranges between some gigabytes
  - Processes shouldn’t access/change other processes address space

- Solution
  - OS manages the mapping of the virtual memory to physical memory
Recall: Virtual VS. Physical Memory

- **Granularity of address spaces (virtual or physical)**
  - Typically 4KB
    - Linux OS makes it possible to support larger page size
- **Virtual memory helps in efficient use of DRAM**
  - Uses DRAM as a cache
  - Non-cached data on disk
- **Simplifies memory management**

**Diagram:**
- **PA virtual address space**
- **PB virtual address space**
- **Physical memory**
- **Disk**
- **Swap space**

**Legend:**
- Blue arrows represent virtual to physical memory mapping.
- Orange arrows represent physical to swap space mapping.
Uniform Memory Access (UMA)

- All memory is equidistant from all the cores
  - Equal access times to memory units
- Also known as CC-UMA
  - Cache Coherent UMA
- Disadvantage?
  - Hard to scale with increasing number of cores
Virtual to Physical Mapping in UMA

- As memory is equidistant, same latency to access each physical pages of $P_B$’s virtual address space
  - Equal number of hops
Non Uniform Memory Access (NUMA)

- Generally made by physically linking two or more multicore processors (i.e., socket) on the same motherboard
  - Single processor can also have a NUMA architecture (e.g., AMD EPYC processors)
- One socket can directly access memory of another socket
  - Both DRAM and caches (e.g., LLC)
- Cache coherent NUMA architecture called as CC-NUMA
  - Cache Coherent NUMA
- Cache coherent interconnect (e.g., QPI on Intel processors) used for low latency and high bandwidth memory accesses across the NUMA domains
Virtual to Physical Mapping in NUMA

- Each NUMA domain has its own physical pool of memory.
- Local memory offers higher bandwidth and lower latency than remote memory.
- Even if it’s a cc-NUMA system where memory is addressed with a global address space, accessing different parts of memory can result in different latency and bandwidth.
  - Imagine LLC on socket-1 is sharing a cache line residing on the LLC of socket-2.
- High performance can only be achieved by placing the computation and its data inside the same NUMA domain.
  - What about the energy usage?
### Getting CPU & NUMA Information on Linux

```bash
vivek@hippo:~$ numactl --hardware
available: 4 nodes (0-3)
node 0 cpus:  0  1  2  3  4  5  6  7  32  33  34  35  36  37  38  39
node 0 size:   15943 MB
node 0 free:      14753 MB
node 1 cpus:  8  9 10 11  12  13  14  15  40  41  42  43  44  45  46  47
node 1 size:   16121 MB
node 1 free:      10956 MB
node 2 cpus: 16 17 18 19  20  21  22  23  48  49  50  51  52  53  54  55
node 2 size:   16121 MB
node 2 free:      13932 MB
node 3 cpus: 24 25 26 27  28  29  30  31  56  57  58  59  60  61  62  63
node 3 size:   16120 MB
node 3 free:      15058 MB
node distances:
  node  0  1  2  3
  0:   10  16  16  16
  1:   16  10  16  16
  2:   16  16  10  16
  3:   16  16  16  10
```

- **NUMA node memory**
- **Logical core IDs**
- **Node distance** measures the relative cost to access the memory of another NUMA-node. A NUMA-node has always a distance 10 to itself (lowest possible value).
Recursive Array Sum on NUMA Processor

```c
int array_sum(int low, int high) {
    if(high - low > THRESHOLD) {
        int mid = (low + high)/2;
        future<int> left = async([=]() { return array_sum(low, mid); });
        right = array_sum(mid, high);
        return left.get() + right;
    } else {
        int sum = 0;
        for(int i=low; i<high; i++) {
            sum += array[i];
        }
        return sum;
    }
}
```

- Async would be executing on cores of both the sockets
- Overheads?
  - Remote DRAM accesses
  - Cache lines shared across caches on both socket. Can we fix it?
    - Easily resolvable by setting THRESHOLD as multiple of cache line size (64Bytes on x86)
- Physical page allocations
  - Only on socket-1
    - Local DRAM accesses at socket-1, but remote DRAM accesses at socket-2
  - Only on socket-2
    - Local DRAM accesses at socket-2, but remote DRAM accesses at socket-2
  - Shared between socket-1 and socket-2
    - Random work-stealing will lead to remote DRAM accesses

Why we are not using Fib as an example here?
Recursive Array Sum on NUMA Processor

```cpp
int array_sum(int low, int high) {
    if(high - low > THRESHOLD) {
        int mid = (low + high)/2;
        future<int> left = async([=]() { return array_sum(low, mid); });
        right = array_sum(mid, high);
        return left.get() + right;
    } else {
        int sum = 0;
        for(int i=low; i<high; i++) {
            sum += array[i];
        }
        return sum;
    }
}
```

Two issues to resolve:
- a) Ensuring pages allocations at right place
- b) Partitioning computation at each socket

- Async would be executing on cores of both the sockets
- Overheads?
  - Remote DRAM accesses
  - Cache lines shared across caches on both socket
    - Easily resolvable by setting THRESHOLD as multiple of cache line size (64Bytes on x86)

Physical page allocations
- Only on socket-1
  - Local DRAM accesses at socket-1, but remote DRAM accesses at socket-2
- Only on socket-2
  - Local DRAM accesses at socket-2, but remote DRAM accesses at socket-2
- Shared between socket-1 and socket-2
  - Random work-stealing will lead to remote DRAM accesses

High performance only if
- Physical pages for left half of the array on socket-1’s DRAM and left subtree executes on socket-1
- Physical pages for right half of the array on socket-2’s DRAM and right subtree executes on socket-2
Today’s Lecture

- Non uniform memory accesses
- Page allocation policies
- NUMA aware parallel runtime
Page Allocation Policy on Linux

- First Touch Policy
  - Calling malloc/new doesn’t allocate the physical pages on a NUMA node

```c
int * array = new int[2048]; // Two pages
```
Page Allocation Policy on Linux

- **First Touch Policy**
  - Calling `malloc/new` doesn’t allocate the physical pages on a NUMA node
    - It is allocated only when those addresses are first “touched”
    - The physical page is allocated during page-fault handling
      - A hardware fault will be generated when a process touches an address (page fault) that has not been used yet
  - Allocates the physical page in the memory closest to the thread/process accessing this page for the first time
    - Default policy on Linux

```c
int * array = new int[2048]; // Two pages
for(int i=0; i<size; i++) {
    array[i] = 0;
}
```
Page Allocation Policy on Linux

- **First Touch Policy**
  - Where both the pages will be allotted in the shown program?
    - Not guaranteed, may even be on a single socket if both the threads are going to run on a single socket

```cpp
int * array = new int[2048]; // Two pages
int mid = 2048/2;
std::thread T1([=]() {
    for(int i=0; i<1024; i++) {
        array[i] = 0;
    }
});
std::thread T2([=]() {
    for(int i=1024; i<2048; i++) {
        array[i] = 0;
    }
});
```

How to ensure each thread runs on different socket?
Page Allocation Policy on Linux

- First Touch Policy
  - Where both the pages will be allotted in the shown program?
    - Not guaranteed, may even be on a single socket if both the threads are going to run on a single socket
    - To ensure they are allotted on different sockets, they must be mapped to two different sockets
      - Set the CPU affinity of a thread using `sched_setaffinity` before letting them “touch” the memory
      - Even if there are only two threads, and two dual core sockets, why should we still set their affinities on two different sockets?
        - Lesser contention over on-chip memory controller

```cpp
int * array = new int[2048]; // Two pages
int mid = 2048/2;
std::thread T1([]() { // set affinity(C1)
    for(int i=0; i<1024; i++) {
        array[i] = 0;
    }
};
std::thread T2([]() { // set affinity(C3)
    for(int i=1024; i<2048; i++) {
        array[i] = 0;
    }
});
```
Lecture 11: Non Uniform Memory Access Architecture

Page Allocation Policy on Linux

- **First Touch Policy**
  - Where both the pages will be allotted in the shown program?
    - Not guaranteed, may even be on a single socket if both the threads are going to run on a single socket
  - To ensure they are allotted on different sockets, they must be mapped to two different sockets
    - Set the CPU affinity of a thread using `sched_setaffinity` before letting them “touch” the memory
    - Even if there are only two threads, and two dual core sockets, why should we still set their affinities on two different sockets?
      - Lesser contention over on-chip memory controller

```cpp
int * array = new int[2048]; // Two pages
int mid = 2048/2;
std::thread T1([=]() { //set affinity(C1)
    for(int i=0; i<1024; i++) {
        array[i] = 0;
    }
});
std::thread T2([=]() { //set affinity(C3)
    for(int i=1024; i<2048; i++) {
        array[i] = 0;
    }
});
```

Can we improve the performance?
Page Allocation Policy on Linux

- **First Touch Policy**
  - Another way to achieve the same result, but with only a few CPU cycles
    - Using `posix_memalign`

```cpp
posix_memalign(&array, ...); // Two pages

std::thread T1([=](){ // set affinity(C1)
    array[0] = 0;    // array is int type
});
std::thread T1([=](){ // set affinity(C3)
    array[1024] = 0;
});
```
Alternative Allocation Policies

**numactl**
- Linux tool to control NUMA policy for launching processes or allocating shared memory
- Controls the policy for the entire program, but not to individual memory areas

**Libnuma**
- Shared library that can be linked to programs and offers several policies for NUMA allocations that could be used differently for different memory areas

### Example Usage of numactl

```bash
numactl [-a] [--interleave=] [-r <nodes>] [--preferred=] [-p <node>]
        [--physcpubind=] [-C <cpus>] [--cpunodebind=] [-N <nodes>]
        [--membind=] [-m <nodes>] [--localalloc | -l] command args ...
```

### Memory Policy Description
- `--interleave` specifies the nodes for interleaving.
- `--preferred` sets the preferred nodes.
- `--physcpubind` binds the worker threads to the specified CPU nodes.
- `--cpunodebind` binds the worker threads to the specified node.
- `--membind` binds the memory to the specified nodes.
- `--localalloc` binds the entire program to a single node.
Few Routines from libnuma

- `numa_max_node()`: How many nodes are there?
- `numa_alloc_onnode()`: Alloc memory on a particular node
- `numa_alloc_local()`: Alloc memory on the current “local” node
- `numa_alloc_interleaved()`: Places memory pages across all the available NUMA nodes in round robin
- `numa_free()`: Free the memory
- `numa_run_on_node()`: Run thread and its children on this node
- `numa_node_of_cpu()`: My current NUMA node
Frequently used Page Allocation Policies

- Allocating an integer array of size 8192
  - Assuming page aligned memory allocation, total pages are 8 (4KB each)
    - Assume physical page ids P1-P8
  - Total 4 NUMA nodes
    - N₁-N₃

- Block cyclic
  - Block size is ratio of total pages and number of NUMA nodes

- Interleaved

Question: which of these two policies you would choose, if you have to run recursive task parallel implementation of vector addition using the traditional random work-stealing runtime?
Today’s Lecture

- Non uniform memory accesses
- Page allocation policies
- NUMA aware parallel runtime
NUMA Aware Parallel Runtime: Naïve Approach

Recursive task parallel vector addition

Question: What should be our first concern, and how to resolve it?
NUMA Aware Parallel Runtime: Naïve Approach

- Spread the memory evenly across all the NUMA nodes
  - Reduce bottleneck at individual node’s memory controller
  - Improving locality

Question: How to improve locality in random work-stealing?
NUMA Aware Parallel Runtime: Naïve Approach

- Spread the memory evenly across all the NUMA nodes
  - Reduce bottleneck at individual node’s memory controller
  - Improving locality
- Create teams of worker threads at each NUMA node where they can steal tasks from their local team members

Question: How to start the execution?
NUMA Aware Parallel Runtime: Naïve Approach

- Spread the memory evenly across all the NUMA nodes
  - Reduce bottleneck at individual node’s memory controller
  - Improving locality
- Create teams of worker threads at each NUMA node where they can steal tasks from their local team members
- Give each team a seed task as per the locality of the data associated with that task
NUMA Aware Parallel Runtime: Naïve Approach

- Spread the memory evenly across all the NUMA nodes
  - Reduce bottleneck at individual node’s memory controller
  - Improving locality
- Create teams of worker threads at each NUMA node where they can steal tasks from their local team members
- Give each team a seed task as per the locality of the data associated with that task
- Dealing with load imbalance (if any)?
  - Allow stealing from a remote team
  - Migrate the memory pages associated with that task from remote node to local node
    - `numa_move_pages()` in `libnuma`
### NUMA Aware Parallel Runtime: Naïve Approach

- **Spread the memory evenly across all the NUMA nodes**
  - Reduce bottleneck at individual node’s memory controller
  - Improving locality
- **Create teams of worker threads at each NUMA node where they can steal tasks from their local team members**
- **Give each team a seed task as per the locality of the data associated with that task**
- **Dealing with load imbalance (if any)?**
  - Allow stealing from a remote team
  - Migrate the memory pages associated with that task from remote node to local node
    - `numa_move_pages()` in `libnuma`
- **What if it’s not so easy to map seed tasks?**
  - We will see in next lecture
Reading Materials

- NUMA APIs for Linux

- Page migration
Next Lecture (L #12)

- Recursive task parallelism on NUMA architecture