Overview

Non-Uniform Memory Access (NUMA) is a shared memory architecture in which each processor in a multiprocessor system is directly connected to its own memory. Indirect access to memory connected to other processors (also called System-on-Chip or SoCs, and, when on the same SoC, called Processing Elements (PEs), or cores) remains possible, but at the cost of higher access times. NUMA is commonly implemented in multi-socket platforms in which each socket contains an SoC that has its own memory local to the socket.

Over time, disparities in speed between processors and memory have increased dramatically. Multi-socket platforms and SoCs with numerous PEs, such as Ampere® Altra® and Ampere® Altra® Max (referred to as the "Altra Family"), exacerbate this problem. These innovations have led to high-performance processors that can be frequently "starved for data."

NUMA attempts to solve this by providing separate pools of memory to each processor (SoC or group of PEs), thus providing fast access to local memory. However, this increases the performance penalty that occurs when processors attempt to access remote memory, that is, memory that is not directly connected to the processor. Using NUMA helps to ensure that a given processor, whenever possible, accesses memory that is physically closest and has the shortest access times.

Note: Because the term "CPU" sometimes refers to SoCs and sometimes to PEs, this guide avoids the term. However, some Linux utilities produce output that uses the term CPU, while some Linux utilities use the term core. In almost all such cases, CPU and core refer to one of the PEs on an Altra or Altra Max SoC. Where the meaning is ambiguous, clarification is provided.

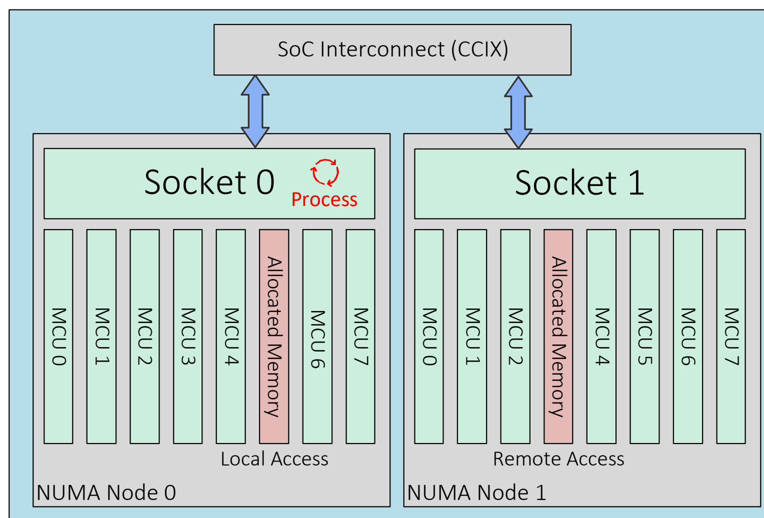
NUMA Use Cases

Dual-Socket (2P) Systems

A 2P system containing an Altra Family SoC in each socket has at least two NUMA nodes. The number of NUMA nodes is configured at boot time using Unified Extensible Firmware Interface (UEFI).

In the sample 2P system shown in *Figure 1*, the identical Altra Family SoCs in Socket 0 and Socket 1 each contain eight Memory Controller Units (MCUs). By default, memory connected to the Socket 0 MCUs is in NUMA Node 0 and the memory connected to the Socket 1 MCUs is in NUMA Node 1. In the example, the process running in a PE in Socket 0 has memory allocated in NUMA Node 0 and NUMA Node 1. For optimal performance, the process should allocate all of its memory in NUMA Node 0; this figure merely points out the difference between local and remote memory accesses.

Figure 1: Local and Remote Memory Accesses



Comparing Local and Remote Memory Accesses

NUMA, by definition, provides access to fast local memory. The obvious downside is that access to remote memory is slower. How much slower depends upon factors such as the frequency of the SoC interconnect, memory speed, and the frequency and patterns of memory access in the workload. Sometimes, differences in performance are hidden by other costs or inefficiencies at higher levels of the software stack and are therefore workload dependent.

The difference in accessing local memory compared to remote memory for stress-ng (a microbenchmark), shown in *Table 1*, is a 5x difference in performance. Note that this is the worst-case performance for one benchmark and is shown only to illustrate that it is important to consider NUMA and optimize for it.

Table 1: Stress ng Comparison of Local and Remote NUMA Accesses

	LOCAL NUMA NODE	REMOTE NUMA NODE
Stress-ng Bogo operations/sec (higher is better)	356	64

Single-Socket (1P) Systems

A 1P system containing an Altra Family SoC has at least one NUMA node. Ampere Altra Family SoCs, which are built on monolithic dies, support these NUMA modes, which are configured at boot time using UEFI.

- Monolithic

  The processor is configured to have one NUMA domain; all eight MCUs are connected in an interleaved manner.
- Hemisphere

  The processor is partitioned into two NUMA domains. Half the PEs are grouped with four of the eight MCUs.
- Quadrant

  The processor is partitioned into four NUMA domains. Memory is interleaved across two MCUs in each quadrant.

Determining the NUMA Configuration of a Platform

lscpu and numactl are two popular Linux utilities that can display the current NUMA configuration.

lscpu

This utility lists the number of PEs and NUMA nodes, along with the PEs allocated to each NUMA node. This sample output shows that there are two sockets and two NUMA nodes on the platform (one NUMA node per socket). PEs 0-79 are part of NUMA node 0, while PEs 80-159 are part of NUMA node 1.

```
$ lscpu
Architecture:         aarch64
Byte Order:           Little Endian
CPU(s):               160
On-line CPU(s) list:  0-159
Thread(s) per core:   1
Core(s) per socket:   80
Socket(s):            2
NUMA node(s):         2
Vendor ID:            ARM
Model:                1
Stepping:             r3p1
CPU max MHz:        3300.0000
CPU min MHz:          1000.0000
BogoMIPS:             50.00
L1d cache:            64K
L1i cache:            64K
L2 cache:             1024K
NUMA node0 CPU(s):    0-79
NUMA node1 CPU(s):    80-159
Flags:                fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp
cpuid asimdrdm lrcpc dcpop asimddp ssbs
```

numactl

This utility displays the NUMA configuration and an approximation of relative node-to-node latency as shown:

```
$ numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66
67 68 69 70 71 72 73 74 75 76 77 78 79
node 0 size: 261458 MB
node 0 free: 256117 MB
node 1 cpus: 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106
107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131
132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156
157 158 159
node 1 size: 260754 MB
node 1 free: 251522 MB
node distances:
node   0   1
  0:  10  20
  1:  20  10
```

Detecting NUMA Problems

NUMA is used optimally when memory is allocated on the same socket as the process (called localalloc). Several utilities can detect when access to memory is remote.

numastat

The numastat utility, part of the numactl Linux package, can report NUMA statistics. The utility displays aggregate memory statistics for processes and the OS for each NUMA node.

This sample numastat output is for a Java process (pid 169423) running on a platform containing two NUMA nodes. The output shows that the allocated memory of the process is on both NUMA nodes, which can indicate a performance problem.

```
$ numastat -p 169423
Per-node process memory usage (in MBs) for PID 169423 (java)
                           Node 0          Node 1           Total
                  --------------- --------------- ---------------
Huge                         0.00            0.00            0.00
Heap                         0.06            0.12            0.19
Stack                        0.12            0.00            0.12
Private                   1101.38         2175.00         3276.38
----------------  --------------- --------------- ---------------
Total                     1101.56         2175.12         3276.69
```

Numa_Hit indicates that the process wanted to allocate memory on that node and succeeded. Local_Node indicates that the process ran on the node for which it was allocated memory. Other_Node shows the number of times the process ran on a node for which memory was not allocated.

In this numastat output, the Other_Node and Numa_Miss values should ideally be zero or small for optimal performance.

```
$ numastat -n
Per-node numastat info (in MBs):
                          Node 0          Node 1           Total
                 --------------- --------------- ---------------
Numa_Hit               125446.12       145069.00       270515.12
Numa_Miss                   0.00            0.00            0.00
Numa_Foreign                0.00            0.00            0.00
Interleave_Hit            983.62          997.88         1981.50
Local_Node             124877.94       143901.38       268779.31
```

ps

It is useful to determine which PEs a process most recently ran to see whether finer control of memory allocation and PE resources is necessary. The PSR column in the ps output lists the PEs on which the corresponding PID ran most recently. The NUMA topology can be determined using numactl-H or the lscpu utility as previously shown.

```
ps -T -o pid,psr,comm <pid>
PID        PSR COMMAND
  169423  32 java
  169423 120 java
  169423  81 GC Thread#0
  169423  81 VM Thread
  169423  30 Reference Handl
  169423 159 Finalizer
  169423 123 Signal Dispatch
  169423 101 Service Thread
  169423  59 C2 CompilerThre
  169423   1 C1 CompilerThre
  169423 125 Sweeper thread
  169423   4 Notification Th
  169423  24 VM Periodic Tas
  169423  90 Common-Cleaner
  169423  72 Grizzly-worker(
  169423  13 Grizzly-worker(
  169423  65 grizzly-nio-ker
  169423 115 grizzly-nio-ker
```

numa_maps

The file /proc/<pid>/numa_maps describes the NUMA policy and page allocation for a given process. Each line of this file contains information about a memory range used by the process and on which NUMA nodes the pages are allocated.

numa_maps can be used to determine where pages for applications and libraries reside. numa_maps uses the sleep command to illustrate this.

```
$ sleep 10 & sleep 2 ; cat /proc/$(pgrep --newest sleep)/numa_maps
[1] 2563

00400000 default file=/usr/bin/sleep mapped=4 mapmax=2 N0=4 kernelpagesize_kB=4
00606000 default file=/usr/bin/sleep anon=1 dirty=1 N1=1 kernelpagesize_kB=4
00607000 default file=/usr/bin/sleep anon=1 dirty=1 N1=1 kernelpagesize_kB=4
00608000 default heap anon=2 dirty=2 N1=2 kernelpagesize_kB=4
7ffff14e4000 default file=/usr/lib/locale/locale-archive mapped=11 mapmax=77 N1=11
kernelpagesize_kB=4
```

```
7ffff7a0e000 default file=/usr/lib64/libc-2.17.so mapped=87 mapmax=243 N0=17 N1=70
kernelpagesize_kB=4
7ffff7bd0000 default file=/usr/lib64/libc-2.17.so
7ffff7dd0000 default file=/usr/lib64/libc-2.17.so anon=4 dirty=4 N1=4 kernelpagesize_kB=4
7ffff7dd4000 default file=/usr/lib64/libc-2.17.so anon=2 dirty=2 N1=2 kernelpagesize_kB=4
7ffff7dd6000 default anon=3 dirty=3 N1=3 kernelpagesize_kB=4
7ffff7ddb000 default file=/usr/lib64/ld-2.17.so mapped=28 mapmax=56 N0=2 N1=26 kernelpagesize_kB=4
7ffff7fdd000 default anon=3 dirty=3 N1=3 kernelpagesize_kB=4
7ffff7ff9000 default anon=1 dirty=1 N1=1 kernelpagesize_kB=4
7ffff7ffa000 default
7ffff7ffc000 default file=/usr/lib64/ld-2.17.so anon=1 dirty=1 N1=1 kernelpagesize_kB=4
```

The numa_maps output shows:

1. The file /usr/bin/sleep is mapped onto four pages of memory on Node 0 (N0=4), where each page contains 4 KB (kernelpagesize_kB=4).
2. The anonymous mappings from /usr/bin/sleep reside on
3. Node 1 (N1=1).
4. Some libraries are mapped
5. partially on Node 0 and partially on Node 1. For example, libc.so occupies 17 pages on node 0 and 70 pages on node 1.

Note: If the process and the shared libraries used by this process have pages allocated on different NUMA nodes, lower performance can result.

Determining the NUMA Affinity of PCIe Devices and Interrupts

In general, PCIe devices should have affinity with the NUMA node of the local socket for optimal performance. To determine the NUMA node affinity of a PCIe device, use lstopo-no-graphics to get the device addresses as shown:

```
$ lstopo-no-graphics
Machine (510GB total)
  Package L#0
    NUMANode L#0 (P#0 255GB)
    L2 L#0 (1024KB) + L1d L#0 (64KB) + L1i L#0 (64KB) + Core L#0 + PU L#0 (P#0)
    L2 L#1 (1024KB) + L1d L#1 (64KB) + L1i L#1 (64KB) + Core L#1 + PU L#1 (P#1)
    …

    …
    HostBridge
      PCIBridge
        PCI 0000:01:00.0 (Ethernet)
          Net "enp1s0f0"
          OpenFabrics "mlx5_0"
        PCI 0000:01:00.1 (Ethernet)
          Net "enp1s0f1"
          OpenFabrics "mlx5_1"
    HostBridge
      PCIBridge
        PCI 0003:01:00.0 (NVMExp)
          Block(Disk) "nvme0n1"
    HostBridge
      PCIBridge
```

```
      PCIBridge
        PCI 0004:02:00.0 (VGA)
  Package L#1
    NUMANode L#1 (P#1 255GB)
    L2 L#80 (1024KB) + L1d L#80 (64KB) + L1i L#80 (64KB) + Core L#80 + PU L#80 (P#80)
    …
    …
    L2 L#159 (1024KB) + L1d L#159 (64KB) + L1i L#159 (64KB) + Core L#159 + PU L#159 (P#159)
```

Message Signaled Interrupts (MSIs) also have PE affinity based on the location of the PCIe devices. The Interrupt Request (IRQ) numbers and the affinity of an Ethernet interface can be listed as shown:

```
$ ls /sys/class/net/enp1s0f0/device/msi_irqs/
219  221  223  225  227  229  231  233  235  237  239  241  243  245  247  249  251  253
255  257  259  261  263  265  267  269  271  273  275  277  279  281
220  222  224  226  228  230  232  234  236  238  240  242  244  246  248  250  252  254
256  258  260  262  264  266  268  270  272  274  276  278  280  282
$ cat /proc/irq/219/smp_affinity_list
```

If the reported IRQ affinity is unexpected, write to smp_affinity_list as shown to change the IRQ affinity:

```
$ # The following example sets the irq 227 affinity to cpu core 1
$ echo 1 > /proc/irq/219/smp_affinity_list
```

Mellanox provides useful scripts to display and set the IRQ affinity. Refer to *https://github.com/Mellanox/mlnx-tools*.

NUMA Considerations for Virtualized Linux Environments

NUMA tuning guidelines for libvirt (Linux Virtualization) resemble those for bare metal environments. To achieve optimal performance, limit guest memory to one NUMA node. When provisioning new guests using libvirt, the guest follows the default affinity policy of the hypervisor. Typically, the default affinity policy is to use available PEs on the host. On a NUMA system, explicit placement is desired to guarantee local memory accesses. If sufficient memory for a VM is unavailable on one NUMA node, it helps to make the guest aware of NUMA to take advantage of the underlying NUMA architecture of the host.

libvirt Considerations

libvirt uses libnuma to set memory binding policies for processes in the NUMA domain. Specific NUMA settings can be described in a domain XML file or specified using the virsh virtualization management tool. As in bare metal environments, the combination of vCPU and memory node pinning increases cache locality for optimal performance and to avoid cross-node communication. Set vcpupin and numatune attributes in the VM XML file as shown: guest vCPUs 0-7 are pinned to PEs 20-27 and use memory on NUMA node 0:

```
<cputune>
    <vcpupin vcpu='0' cpuset='20'/>
    <vcpupin vcpu='1' cpuset='21'/>
    <vcpupin vcpu='2' cpuset='22'/>
    <vcpupin vcpu='3' cpuset='23'/>
    <vcpupin vcpu='4' cpuset='24'/>
    <vcpupin vcpu='5' cpuset='25'/>
    <vcpupin vcpu='6' cpuset='26'/>
    <vcpupin vcpu='7' cpuset='27'/>
  </cputune>

  <numatune>
```

This can also be accomplished using virsh:

```
$ virsh vpcupin instancename  0 20
$ virsh vpcupin instancename  1 21
$ virsh vpcupin instancename  2 22
$ virsh vpcupin instancename  3 23
$ virsh vpcupin instancename  4 24
$ virsh vpcupin instancename  5 25
$ virsh vpcupin instancename  6 26
$ virsh vpcupin instancename  7 27
$ virsh numatune instancename --nodeset 0
```

Kernel Samepage Merging (KSM) Considerations

KSM is a Linux kernel feature used by Kernel-based Virtual Machine (KVM) hypervisors. KSM tries share common libraries and other frequently used data to deduplicate and save memory space. By default, KSM is enabled by default and pages across NUMA nodes can be merged by default. VMs accessing merged pages can suffer significant performance penalties, so only pages from the same NUMA node should be enabled to merge. The Sysfs interface merge_across_nodes can be written 0 to disable merges across nodes:

```
$ echo 0 > /sys/kernel/mm/ksm/merge_across_nodes
```

NUMA Considerations for Containerized Environments Such as Docker

Docker runs processes in isolated containers. A container is a process which runs on a host. The operator can add or adjust the constraints on container resources when executing “docker run”.

`--cpuset-cpus=””` : specify the PEs on which to enable execution.

`--cpuset-mems=””` : specify the NUMA nodes in which to enable execution.

This example sets the affinity of a Redis container to PEs 0 and 1 while using memory on NUMA node 0:

```
$ docker run --cpuset-cpus=”0-1” –cpuset-mems=0 –name redis-server -d redis
```

NUMA Recommendations for Optimal Data Placement

The task of assigning processes to NUMA nodes is called NUMA placement. NUMA placement policies can greatly influence workload performance. Several techniques control NUMA placement on Linux. Linux has been NUMA-aware since kernel 2.5 and tries to optimize data placement without user intervention, as described in:

https://www.kernel.org/doc/html/latest/admin-guide/mm/numa_memory_policy.html

However, special NUMA configurations and placement is available using tools or kernel parameters when the OS scheduler does not do a good job. Working set and data access patterns are two factors to consider while choosing the best technique for NUMA optimizations.

The following are techniques (automated and manual) that can be taken to optimize data placement in general (that is, without tuning for specific workloads).

Kernel-Level NUMA Balancing in Linux

Automatic NUMA balancing is a kernel scheduler feature that migrates tasks (processes and threads) “closer” to the memory they access. This is the simplest method of NUMA placement because it is automatic.

sysctl or the /proc interface can read the current numa_balancing value. The default value is enabled on kernels on most Linux distros.

```
$ cat /proc/sys/kernel/numa_balancing
1
$ sysctl kernel.numa_balancing
kernel.numa_balancing = 1
```

When automatic NUMA balancing is enabled by writing a value of 1 as shown, the kernel periodically scans portions of the address space of a task and unmaps the memory, resulting in page faults when that portion of the address space is next accessed. When a page fault occurs, it is determined whether data should migrate to a local memory node. Keep in mind that page unmapping and page migration incurs some overhead. However, the overhead is justified if, over time, data migrates closer to the process.

Other kernel tunables in /proc/sys/kernel/numa*, along with numa_balancing, support fine-tuning this feature.

It can be useful to observe the activity caused by the automatic NUMA balancer using numastat and the metrics provided in the /proc/vmstat interface. Measure performance before and after balancer activity to ensure that only recent changes are monitored.

```
$ numastat -n
Per-node numastat info (in MBs):
                          Node 0          Node 1           Total
                 --------------- --------------- ---------------
Numa_Hit               125446.12       145069.00       270515.12
Numa_Miss                   0.00            0.00            0.00
Numa_Foreign                0.00            0.00            0.00
Interleave_Hit            983.62          997.88         1981.50
Local_Node             124877.94       143901.38       268779.31
Other_Node                568.19         1167.62         1735.81
```

```
$ grep -i numa /proc/vmstat
numa_pte_updates 1445376649
numa_huge_pte_updates 172586
numa_hint_faults 30661524
numa_hint_faults_local 28176991
numa_pages_migrated 41643072
```

numa_pte_updates and numa_huge_pte_updates indicate the number of base and huge pages unmapped for NUMA hints.

numa_hint_faults, (numa_hint_fault_local- numa_hit_faults) record the NUMA hinting page faults trapped, and numa_hint_fault_local reports how many of those were local to the node.

A high percentage of local to total hint faults is a good indication that the data is moving closer to the process accessing the data. numa_pages_migrated records the pages migrated due to memory misplacement. This statistic is an indication of the overhead of the automatic NUMA balancing feature.

Automatic NUMA balancing can help improve performance on long running tasks but can hurt performance for short-lived ones. There can also be cases where applications starting up gradually on both sockets can result in triggering back-and-forth migrations that can hurt performance. Using the tools outlined in preceding sections should help identify when such migrations occur.

User-Level NUMA Balancing Using numad

numad is a user-level PE-to-memory affinity manager that can improve NUMA performance automatically. numad tries to allocate PE and memory resources to localize usage while adjusting to dynamic system conditions. numad can also provide resource preplacement information that various job management systems can query to help with initially bind PE and memory resources. When enabled, numad overrides kernel automatic NUMA balancing behavior.

```
$ numad # start numad
$ numad -i 0 # stop numad
$ numad -S 0 -p <pid> # limit numad to manage pid only. -S 0 includes explicitly included pids only
```

numad accesses information in the /proc file system to monitor available system resources on a per-NUMA node basis. numad activities are logged in /var/log/numad.log. numad attempts to localize and isolate significant processes on a NUMA node or subset of NUMA nodes while maintaining specified target utilization. numad tends to improve performance when required resources for a process are contained in a NUMA node. numad management can be limited to a set of processes or can exclude certain processes, unlike kernel level automatic NUMA balancing.

Controlling PE and Memory Affinity Using numactl

numactl is a command line utility that specifies NUMA scheduling policies to the OS when starting an application. This binds applications to specific PEs and NUMA nodes by. The policies are inherited by the child tasks and processes. numactl uses the libnuma APIs to set the scheduling policies.

```
$ # run and allocate memory on numa node 0 for the program
$ numactl --cpunodebind=0 --membind=0 <program>
$
$ #run the program on node0 and preferably allocate memory on node 0
$ numactl --cpunodebind=0 --preferred=0 <program>
```

numactl supports more command line options for scheduling jobs at the PE-level and the node level.

numactl controls management policies at the program level and not at thread level. APIs in the libnuma shared library can be used if finer-grained policy control is needed. It provides higher-level APIs than using NUMA system calls directly in the application program. However, managing the allocation policies within the program requires recompilation and may not be a feasible option in production environments.

Migrating Memory Pages Using migratepages

Using the migratepages utility, one can move the physical location of process pages between NUMA nodes without changes to the virtual address space of the process, reducing the distance between a process and memory, thus improving performance.

For information about page migration, refer to *https://www.kernel.org/doc/html/latest/vm/page_migration.html*.

```
$ sleep 600 &
$ head -n 3 /proc/$(pgrep --newest sleep)/numa_maps
aaaad2420000 default file=/usr/bin/sleep mapped=1 N1=1 kernelpagesize_kB=64
aaaad2430000 default file=/usr/bin/sleep anon=1 dirty=1 N0=1 kernelpagesize_kB=64
aaaad2440000 default file=/usr/bin/sleep anon=1 dirty=1 N0=1 kernelpagesize_kB=64

$ migratepages `pgrep --newest sleep` 1 0

$ head -n 3 /proc/$(pgrep --newest sleep)/numa_maps
aaaad2420000 default file=/usr/bin/sleep mapped=1 N0=1 kernelpagesize_kB=64
aaaad2430000 default file=/usr/bin/sleep anon=1 dirty=1 N0=1 kernelpagesize_kB=64
```

Clearing the File System Cache (Also Called Dropping Caches)

Linux uses the file cache or page cache to reduce frequent access to disk I/O. Over time, this can result in pages being on a different NUMA node than the one on which the process runs. Writing to the /proc/sys/vm/drop_caches file asks the kernel to immediately drop as much clean cached data as possible. Subsequent accesses by the application process to memory bring the data or instructions into local memory.

```
$ sync; echo 1 > /proc/sys/vm/drop_caches
```

Note: Given the implications to processes on the other socket, this option must be evaluated before using in a production environment.

NUMA Recommendations for Specific Workloads and Uses

One consequence of partitioning memory into NUMA nodes is that access to data that is “local” to a specific processor is faster than data that is “remote.” For modern massively parallel and distributed software, it is common for multiple processes to need access to the same data, so remote access is becoming increasingly common. This can adversely affect performance for common use cases:

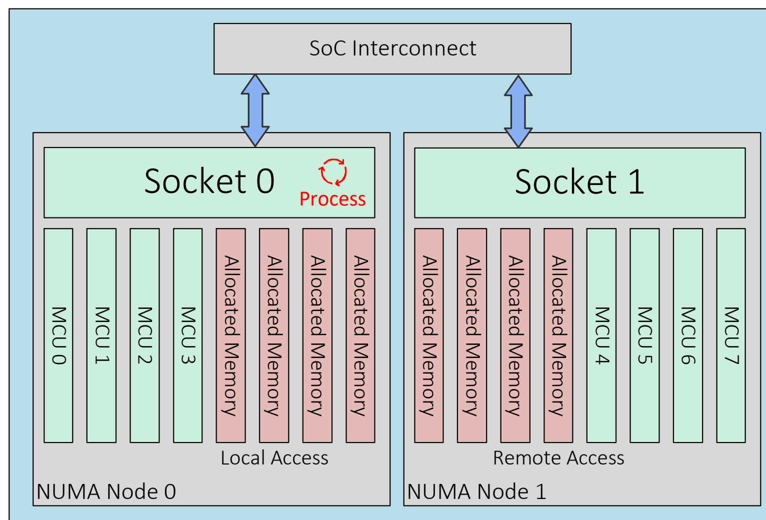
- Workloads with working sets that can fit in local memory (a large set of applications)

  Where possible, workloads PEs and memory should have affinity with the local NUMA node using utilities like numactl, libraries like OpenMP, or directly from the application. numa_balancing and numad are two other options worth studying for a specific application.

- Workloads with large in-memory data structures that cross NUMA node boundaries (for example, databases)

  Monolithic applications, such as databases, allocate large in-memory data structures that are capacity-constrained and must cross NUMA boundaries as shown:

Figure 2: Allocated Memory Across NUMA Boundaries



If all allocated memory is not used uniformly, one option is to use the--interleave memory policy when starting the application. This policy ensures that memory is allocated using all NUMA nodes in a round robin manner at a page-level.

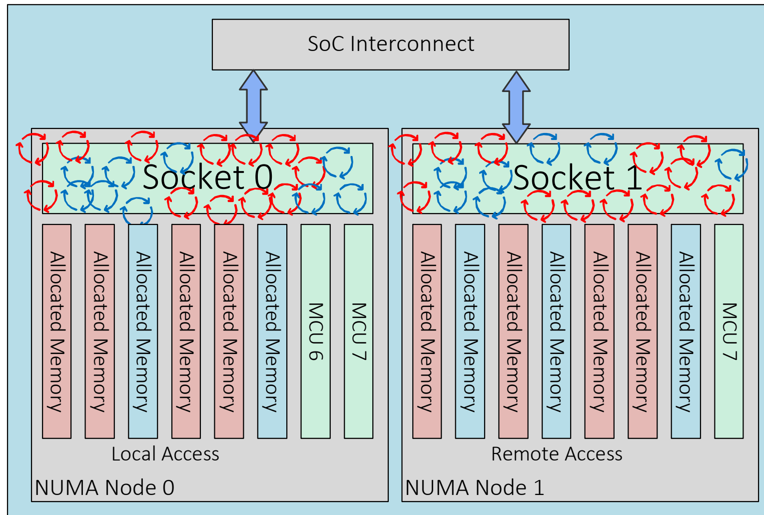
```
$ numactl –interleave=all ./stream
```

This command invokes the stream benchmark so that memory is allocated with alternating pages residing on both sockets in a round robin manner.

- Workloads that oversubscribe PEs (for example, Microservices)

  When PEs are oversubscribed (that is, the number of processes and threads exceeds the number of PEs), the kernel cannot always ensure that scheduled processes and threads wake up on the same socket for which they are initially allocated memory. Over time, this can result in processes, threads, and allocated memory becoming distributed across both sockets as shown:

Figure 3: Workloads That Oversubscribe PEs

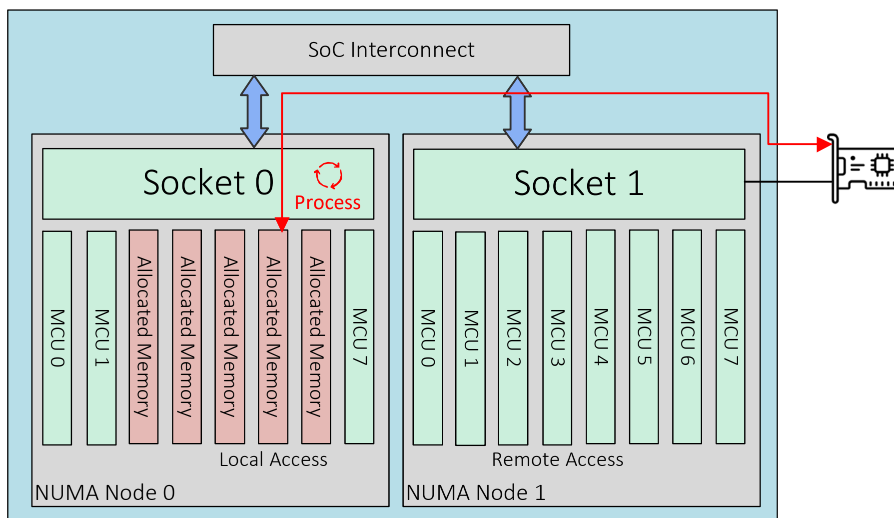


  For such workloads, we recommend using the--interleave memory policy with numactl when starting the applications. This policy ensures that memory is allocated using all NUMA nodes in a round robin manner at a page-level to ensure predictable performance. numa_balancing and numad are also options worth studying for your application.

- I/O Device NUMA Affinity

  It is common for I/O devices, such as NICs and NVMe SSDs, to connect to PCIe slots that are attached to a specific socket. Processes or threads running on the remote socket must then access data across the SoC interconnect as shown:

Figure 4: PCIe Device Connected to a Remote Socket



For I/O intensive workloads, allocating application memory on the socket that is connected to the PCIe device can improve performance. Utilities such as lstopo can be used to study the relationship between I/O devices and NUMA nodes as previously shown.

Most PCIe devices support Direct Memory Access (DMA). This enables the devices to write data directly to a buffer in memory without PE involvement. Allocating the DMA buffer and the applications buffer on the NUMA node to which the PCIe device is connected reduces trips across the socket interconnect for the DMA operation and reduce latency for reading and writing the DMA buffer. This is also the case for hardware interrupts (hard IRQs).

Additional Resources

Additional resources for Ampere Altra Family processors, including datasheets, user's manuals, and firmware images, are available on the Ampere Computing secure website at *https://connect.amperecomputing.com*.

If You Need Help

For help using the system, contact Technical Support at *https://connect.amperecomputing.com/help*.

Document Revision History

ISSUE	DATE	DESCRIPTION
0.50	October 20, 2021	Initial release.

October 20, 2021

Ampere Computing reserves the right to change or discontinue this product without notice.

While the information contained herein is believed to be accurate, such information is preliminary, and should not be relied upon for accuracy or completeness, and no representations or warranties of accuracy or completeness are made.

The information contained in this document is subject to change or withdrawal at any time without notice and is being provided on an "AS IS" basis without warranty or indemnity of any kind, whether express or implied, including without limitation, the implied warranties of non-infringement, merchantability, or fitness for a particular purpose.

Any products, services, or programs discussed in this document are sold or licensed under Ampere Computing's standard terms and conditions, copies of which may be obtained from your local Ampere Computing representative. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of Ampere Computing or third parties.

Without limiting the generality of the foregoing, any performance data contained in this document was determined in a specific or controlled environment and not submitted to any formal Ampere Computing test. Therefore, the results obtained in other operating environments may vary significantly. Under no circumstances will Ampere Computing be liable for any damages whatsoever arising out of or resulting from any use of the document or the information contained herein.

Ampere Computing
4655 Great America Parkway, Santa Clara, CA 95054
Phone: (669) 770-3700
https://www.amperecomputing.com

Ampere Computing reserves the right to make changes to its products, its datasheets, or related documentation, without notice and warrants its products solely pursuant to its terms and conditions of sale, only to substantially comply with the latest available datasheet.

Ampere, Ampere Computing, the Ampere Computing and 'A' logos, Altra, and eMAG are registered trademarks of Ampere Computing.

Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All other trademarks are the property of their respective holders.

Copyright © 2021 Ampere Computing. All rights reserved.