CPU 访问内存方式简介

2016/4/27
renyl

1 介绍

1) 首先，本文将对 3 种体系结构的内存访问方式进行介绍，分别为：
   a) 对称多处理器结构 (SMP：Symmetric Multi-Processor)
   b) 海量并行处理结构 (MPP：Massive Parallel Processing)
   c) 非一致存储访问结构 (NUMA：Non-Uniform Memory Access)

2) 然后，通过 3 种不同体系结构的内存访问实现原理，来对比它们在性能、扩展以及应用方面的优缺点。

3) 最后，对 NUMA 结构的内存分配策略以及如何使用 numactl 工具优化程序进行介绍。

在详细介绍不同体系架构的内存访问方式之前，先对 CPU 的相关概念、各种总线技术以及内存带宽等概念进行介绍。

2 技术背景

2.1 CPU 相关概念

1) 主频（CPU Clock Speed)：CPU 工作的时钟频率（1 秒内发生的同步脉冲数)，其速度由外频乘以倍频决定。

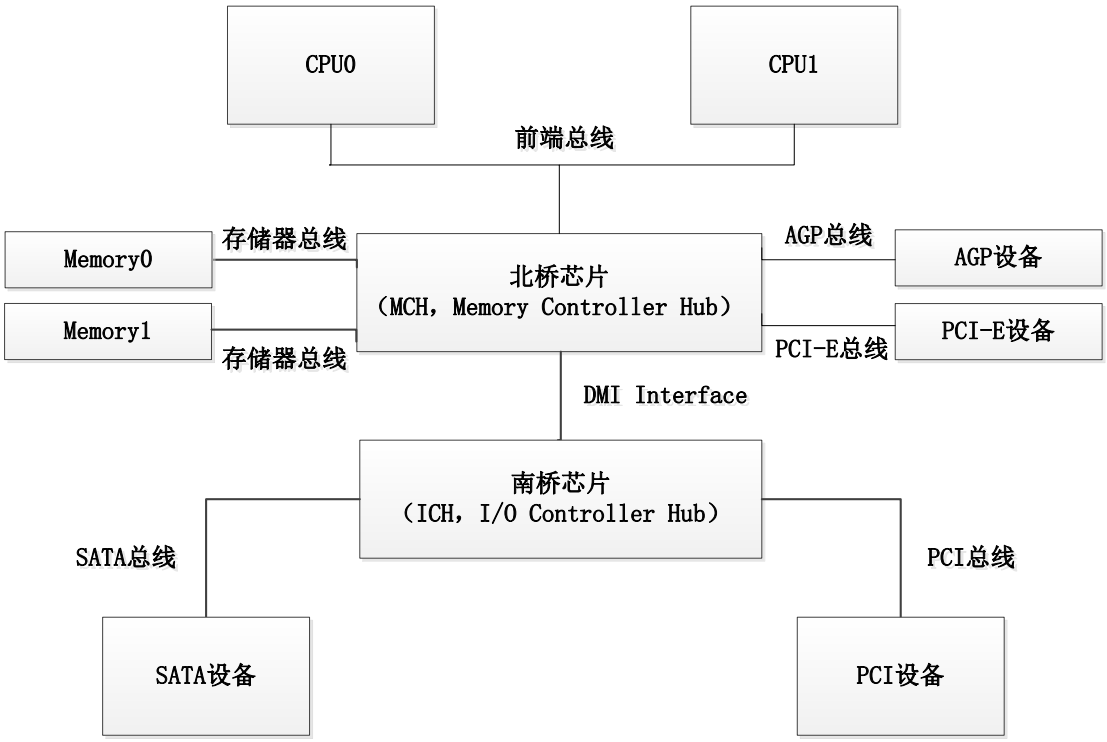
2) 外频：系统总线的工作频率（系统时钟频率)，是 CPU 与周边设备传输数据的频率。具体是指 CPU 到芯片组之间的总线速度（如 CPU 与主板之间同步运行的速度)。

3) 倍频：主频与外频之比的倍数（主频=外频×倍频)，理论上从 1.5 到无限大，以 0.5 为一个间隔单位进行增加。

注：CPU 的性能并不是完全由主频决定，是由主频、管线架构或长度、功能单元数目、缓存设计四个方面共同决定，通常将后面三个要素统称为 CPU 架构。

2.2 FSB（Front Side Bus）

在 Nehalem 微架构之前，Intel 的 CPU 和北桥芯片之间的通信都一直使用 FSB，其体系架构如下所示：



说明：

1) 由于 Intel 使用了“四倍传输”技术，可以使系统总线在一个时钟周期内传送 4 次数据，也就是传输效率是原来的 4 倍。如：在外频为 N 时，FSB 的速度增加 4 倍变成了 4N。

2) FSB 的速度指的是 CPU 和北桥芯片间总线的速度，更实质性的表示 CPU 和外界数据传输的速度。

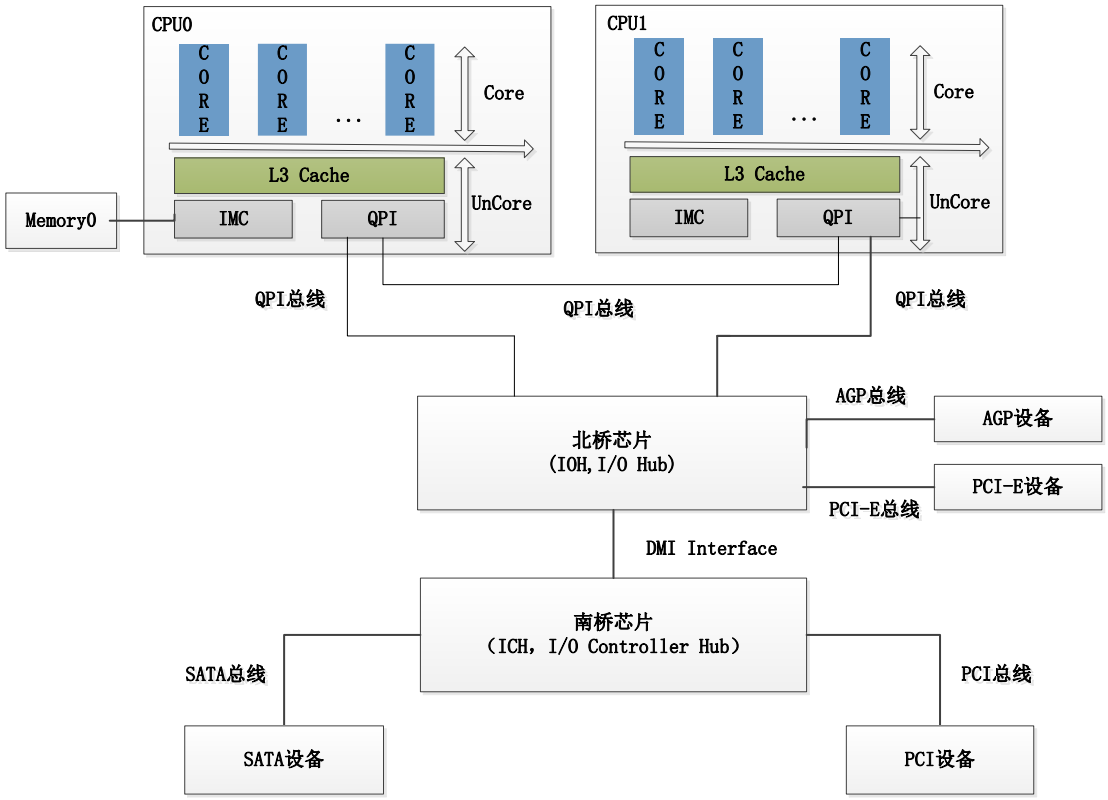
3) 外频的概念是建立在数字脉冲信号震荡的基础上的，它更多的影响了其他总线的频率（如 PCI 总线与 SATA 总线）。如：主板可以通过“二分频”技术将外频降一半，使得 PCI 设备保持在标准工作频率。

优点：
整体成本低。

缺点：
由于只采用一条 FSB 总线，多处理器访问内存时会对 FSB 总线进行抢占，使得多处理器系统间互联和可扩展性差。

2.3 QPI（Quick Path Interconnect）

Intel 在 Nehalem 微架构的 CPU 中首次集成了内存控制器（IMC，Integrated Memory Controller）和引入了 QPI 连接方式，这样 CPU 和北桥芯片之间的通信使用 QPI 取代了前端总线，其体系架构如下所示：



说明：

1) Nehalem 微架构的 CPU 被设计为核心（Core）与非核心（UnCore）两部分。CPU 的执行流水线、L1、L2 Cache 都集成在核心中，而 L3 Cache、IMC、QPI 以及功耗与时钟控制单元都被集成在非核心中。

2) QPI 是在 CPU 中集成内存控制器的体系架构，主要用于多 CPU 间互联以及 CPU 与芯片组间互联的通信，使用 QPI 后 CPU 可直接通过内存控制器访问内存资源，而不是以前繁杂的“前端总线—北桥—内存控制器”模式。

3) 由于 QPI 应用于多 CPU 间互联以及 CPU 与芯片组间互联，因此可以灵活的修改 CPU 中集成的 QPI 数量。如：在针对双路 CPU 的系统中，将集成两组 QPI。

4) 需要注意的是：QPI 并非一种 I/O 接口， CPU 仍然采用 PCI-Express 来处理 I/O 通讯问题。

优点：

CPU 集成内存控制器，由于 CPU 和内存之间的数据传输不再需要经过北桥芯片，因此可以缩短 CPU 与内存之间的数据交换周期。

缺点：
由于内存控制器是集成在 CPU 内部，因此内存的工作频率与 CPU 相同，而且不能进行频率异步设置，这样的话在 CPU 超频的时候会导致内存的频率同 CPU 的外频一起升高，一旦超过内存的承受能力，就会导致内存无法工作，这会大大限制 CPU 的超频能力。

针对使用 QPI 的体系架构，对如下几个时钟概念进行介绍：

1) Base Clock（BCLK）：也叫 Bus Clock，即外频。

2) Core Clock：即主频，Core Clock= Base Clock ＊ 倍频。

3) UnCore Clock（UCLK）：由 BCLK 乘以 UnCore 倍频得到。UCLK 既不是内存频率也不是 QPI 频率，UCLK 不能低于内存频率的 4 倍，这也就是其内存规格限制所在。

4) QPI Clock：由 BCLK 乘以 QPI 倍频得到。由于内存控制器的集成，它的频率大大超出了需要，所以在超频时如果会碰到瓶颈，直接降低它的倍频即可。（在 BIOS 里设置）

5) Memory Clock：由 BCLK 乘以内存倍频得到，内存性能直接受此频率影响。

2.4 内存带宽

数据传输最大带宽取决于所有同时传输数据的带宽和传输频率，公式如下所示：
数据带宽=频率 ＊ 数据位宽 / 8。

使用 FSB 时：

1) 由于内存与 CPU 进行通信需要通过 FSB，因此内存带宽会受到内存频率与 FSB 频率的共同影响。

2) 在内存采用多通道情况下，内存带宽往往会因为 FSB 频率而受限。如：外频为 200HZ，FSB 频率为 800HZ，那么当内存为 DDR2-800（表示工作频率为 400MHZ，等效工作频率为 1600MHZ，采用双通道），FSB 频率就成了内存带宽的瓶颈。此时，内存带宽并不是 12.5GB/s 而仅为 6.25GB/s （800MHZ ＊ 64bit / 8 ）。

使用 QPI 时：

1) QPI 是一种基于包传输的串行式高速点对点连接协议，采用差分信号与专门的时钟进行传输。在延迟方面，QPI 与 FSB 几乎相同，却可以提升更高的访问带宽。

2) 一组 QPI 具有 20 条数据传输线，以及发送（TX）和接收方（RX）的时钟信号。在每次传输的 20bit 数据中，有 16bit 是真实有效的数据，其余四位用于循环冗余校验，以提高系统的可靠性。

3) QPI 频率以 GT/s（QPI Data Rate），明确地表明了 QPI 总线实际的数据传输速率而不是时钟频率。（每个时钟周期上下沿各传输一次数据，实际的数据传输速率两倍于 QPI 总线时钟速率，如：QPI Data Rate 为 6.4GT/s，那么它的 QPI 频率是 3200MHZ）

4) 由于每个 QPI 总线是双向的，在发送的同时也可以接收另一端传输来的数据,因此一组 QPI 带宽为：25.6GB/s（6.4GT/s * 16bit * 2 / 8，每秒传输次数 * 有效数据位）。

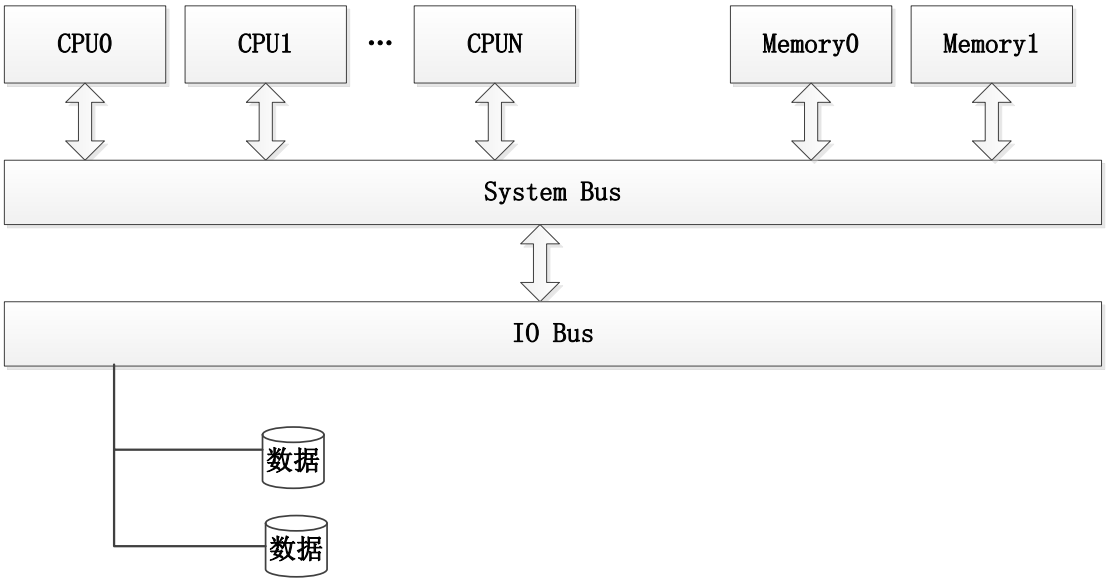
5) Nehalem 微架构 CPU 中的内存控制器具有三个通道，支持三通道 DDR3 1333 内存，对于三条通道全部启用的情况下，内存带宽将高达 32GB/s（1333*3*64bit/8=32GB/s）。

6) 在 Intel 高端的安腾处理器系统中，QPI 高速互联方式使得 CPU 与 CPU 之间的峰值带宽可达 96GB/s，峰值内存带宽可达 34GB/s。

3 体系结构原理

3.1 SMP（Symmetric Multi-Processor）

SMP 体系结构如下所示：



说明：

1) SMP 系统最大的特点就是所有 CPU 共享全部资源（如，系统总线、内存和 I/O 系统）。

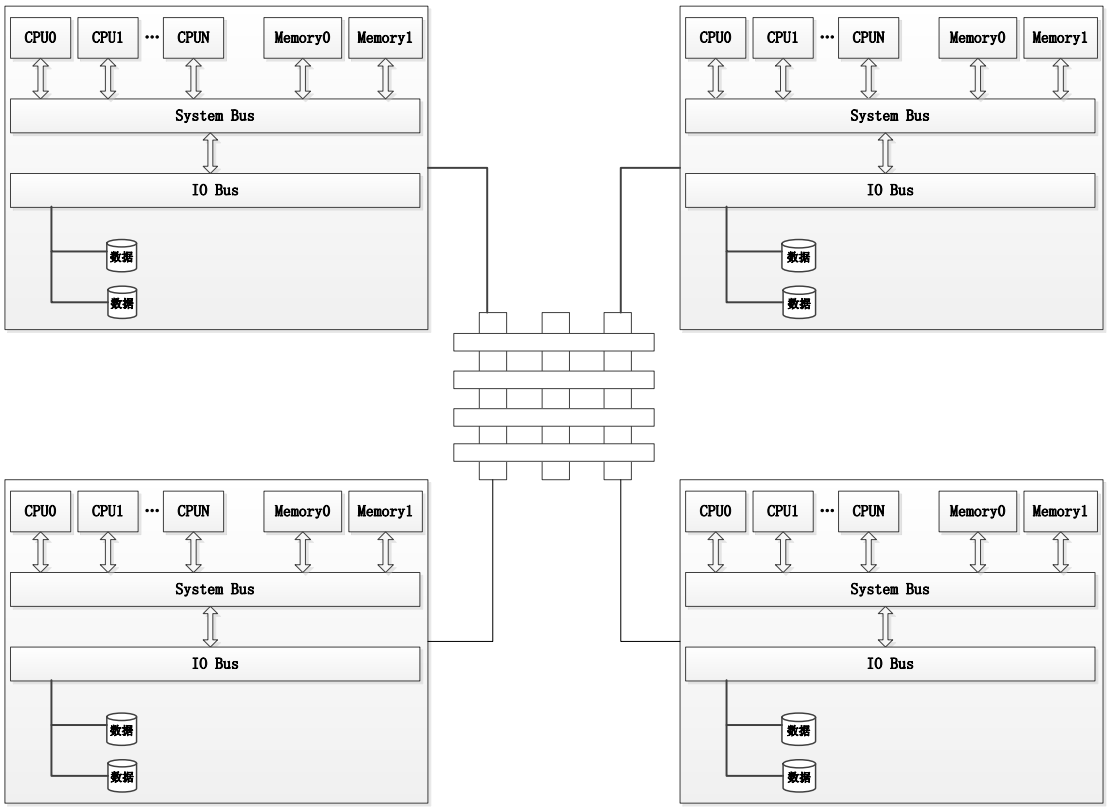
2) SMP 中的多个 CPU 之间没有区别，每个 CPU 访问内存中的任何地址所需时间是相同的。

3) 由于在 SMP 系统中所有 CPU 都共享系统总线，因此当 CPU 的数目增多时，系统总线的竞争冲突迅速加大，系统总线成为了性能瓶颈，可扩展性受到很大限制。实验证明， SMP 服务器 CPU 利用率最好的情况是 2 至 4 个 CPU。

4) 在 SMP 系统扩展中，性能瓶颈除了系统总线外，抢占内存以及内存同步也是难点。
   a) 抢占内存是指：当多个 CPU 共同访问内存中的数据时，它们并不能同时去读写数据。
   b) 内存同步是指：各 CPU 通过 Cache 访问内存数据时，要求系统必须经常保持内存中的数据与 Cache 中的数据一致，若 Cache 的内容更新了，内存中的内容也应该相应更新，否则就会影响系统数据的一致性。

3.2 MPP (Massive Parallel Processing)

MPP 体系结构如下所示：



说明：

1) MPP 系统由多个 SMP 服务器通过一定的节点互联网络进行连接，协同工作完成任务，从用户的角度来看是一个服务器系统。

2) 在 MPP 系统中，每个 SMP 节点可以运行自己的操作系统、数据库等。

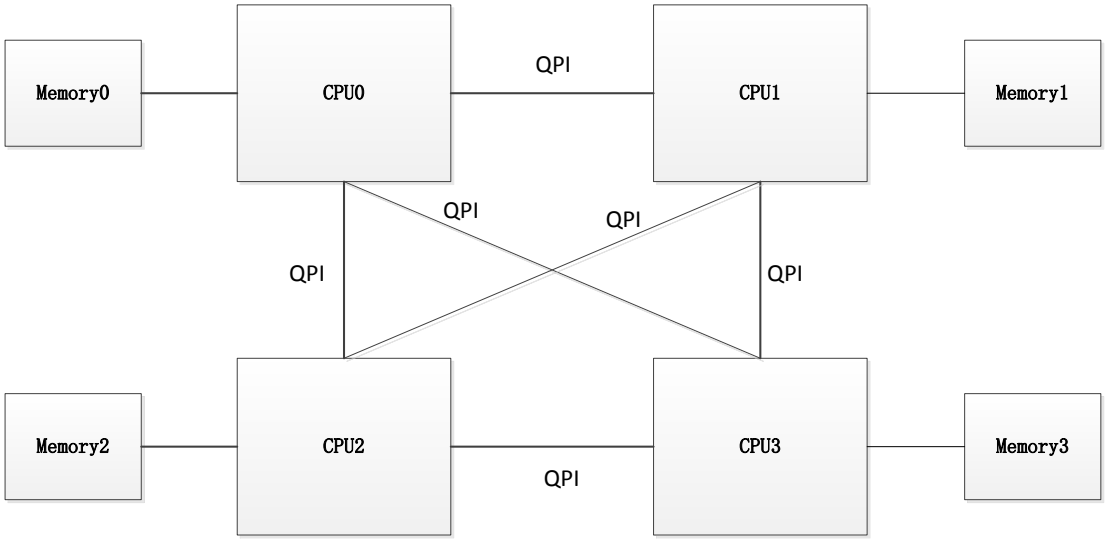
3) MPP 系统的基本特征是每个 SMP 节点只能访问自己的本地资源（如，内存、存储），是一种完全无共享结构。SMP 节点之间的信息交互是通过节点互联网络实现的，这个过程一般称为数据重分配(Data Redistribution) 。

4) 由于 MPP 的完全无共享结构，其扩展能力理论上无限制。目前的技术可实现 512 个节点互联数千个 CPU。目前业界对节点互联网络暂无标准，不同公司采用了不同的内部实现机制（如，IBM 采用了 SP_Switch 机制）。

5) 在 MPP 系统中，由于各 SMP 节点之间的通信需要通过节点互联网络实现，因此需要一种复杂的机制来调度和平衡各个节点的负载和并行处理过程。

3.3 NUMA（Non-Uniform Memory Access）

NUMA 体系结构如下所示：



说明：

1) NUMA 结构综合了 SMP 结构和 MPP 结构的特点：逻辑上整个系统也是分为多个节点，但每个节点可以访问本地内存资源，也可以访问远程内存资源。

2) NUMA 结构的每个 CPU 都可以访问整个系统的内存，但访问本地内存资源远远快于远程内存资源。因此，为了更好地发挥系统性能，开发应用程序时需要尽量减少 CPU 跨节点访问内存。

3) 利用 NUMA 技术，可以较好地解决原来 SMP 系统的扩展问题，在一个物理服务器内可以支持上百个 CPU，但由于访问远地内存的延时远远超过本地内存，因此当 CPU 数量增加时，系统性能无法线性增加。

4) NUMA 最大的特点是引入了 node 和 distance 的概念。
   a) node：资源组，每个资源组内的 CPU 和内存是几乎相等。
   b) distance：用来定义各个 node 之间调用资源的开销，为资源调度优化算法提供支持。

5) 节点访问远程内存的过程如下：（CPU0 访问 MEM1 为例）
   a) CPU0 通过 QPI 先向 CPU1 发起访问请求
   b) CPU1 的内置内存控制器发起 MEM1 访问，MEM1 的数据返回到 CPU1 的三级缓存中
   c) CPU1 再通过 QPI 将数据返回给 CPU0

   由于 QPI 延迟不高，内置的内存控制器延迟也很小，因此即便对远程内存访问，整体的访问延迟仍然会保持在一个较好的水平。

4 体系结构对比

4.1 性能对比

1) SMP

SMP 系统内的所有 CPU 共享系统总线。因此，在 CPU 增加时总线竞争冲突迅速加大，性能无法线性增加。

2) MPP

MPP 系统是由多个 SMP 系统通过节点互联机制实现的，每个节点只访问本地内存和存储，节点之间的信息交互与节点本身的处理是并行进行的。因此，MPP 系统在增加节点时性能基本上可以实现线性增加。

3) NUMA

NUMA 系统内的各节点通过互联机制连接，当某个节点需要访问远程内存时，需要等待。因此，NUMA 系统在增加节点时性能无法线性增加。

4.2 扩展对比

1) SMP

SMP 扩展能力很差，目前 2 个到 4 个 CPU 的利用率最好。但是 IBM 的 BOOK 技术，能够将 CPU 扩展到 8 个。

2) MPP

MPP 理论上可以实现无限扩展，目前技术比较成熟的能够支持 512 个节点，数千个 CPU 进行扩展。

3) NUMA

NUMA 理论上可以无限扩展，目前技术比较成熟的能够支持上百个 CPU 进行扩展。

4.3 应用对比

a) SMP

在联机事务处理系统（OLTP，On-Line Transaction Processing OTLP）中，如果用户访问一个中心数据库，那么采用 SMP 系统结构的效率要比采用 MPP 结构要好。

b) MPP

由于 MPP 系统不共享资源，相对而言，资源要比 SMP 多。当需要处理的事务达到一定规模时，MPP 的效率要比 SMP 好。由于 MPP 系统要在不同处理单元之间传送信息，在通讯时间少的时候，那么 MPP 系统可以充分发挥资源的优势，达到高效率。

c） NUMA

从 NUMA 系统内部集成了多个 CPU，使系统具有较高的事务处理能力。由于远地内存访问时延远长于本地内存访问，因此需要尽量减少不同 CPU 模块之间的数据交互。

5 NUMA 策略及应用

5.1 NUMA 策略

NUMA 策略可以基于 CPU 和内存设定，如下所示：

1） CPU 分配策略：

序号	策略	说明
1	cpunodebind	规定进程运行在某个或某几个 node 上
2	physcpubind	规定进程运行在某个或某几个 core 上

2） 内存分配策略：

序号	策略	说明
1	localalloc（默认模式）	总是在本地节点分配，失败则在其它节点上分配
2	preferred	在指定节点上分配，失败则在其它节点上分配
3	membind	指定的节点上分配，失败也不能从其它节点上分配
4	interleave	在所有节点或者指定的节点上交织分配

注：每个进程（或线程）都会从父进程继承 NUMA 策略，并分配有一个优先 node。

NUMA 结构下的系统会提供两个工具：numastat 与 numactl。接下来，分别对其详细介绍：

1） numastat

numastat 用于显示每个 numa 节点的内存统计信息，如下所示：

```
[root@localhost ~]# numastat
                           node0           node1
numa_hit                215856030       218396123
numa_miss                   1438           24043
numa_foreign               24043            1438
interleave_hit             35970           36105
local_node              215851724       218301862
other_node                   574          118304
```

对 numastat 命令输出的各个参数进行如下说明：

序号	参数	说明
1	numa_hit	内存原计划分配在该 node 上，最终成功分配在该 node 上
2	numa_miss	内存原计划分配在别的 node 上，最终却被分配在该 node 上
3	numa_foreign	内存原计划分配在该 node 上，最终却被分配在别的 node 上

4	interleave_hit	内存原计划交织分配在某 node 上，最终内存成功分配在某 node 上
5	local_node	内存分配在该 node 上，程序运行在该 node 上（这里指 CPU）
6	other_node	内存分配在 node 上，程序运行在别的 node 上（这里指 CPU）

需要特别注意的是：
如果绑定一个进程在 node0 上运行，内存在 node1 分配（如：numactl -N 0 -m 1 ./a.out），并不会产生 numa_miss。因为 numa_miss 并不是指运行的 CPU 和分配的内存不在一个 node 上就产生 numa_miss，而是指原计划分配的内存与最终分配的内存不在同一个 node 上时产生的。

2) numactl
numactl 用于查看与控制进程的 CPU 和内存分配策略，如下所示：

序号	参数	说明
1	numactl -s	显示 NUMA 当前的策略
2	numactl -H	显示每个 numa 节点的详细信息
3	numactl -C cpus	设置进程的 CPU 分配策略（cpunodebind 策略）
4	numactl -N nodes	设置进程的 node 分配策略（physcpubind 策略）
5	numactl -l	设置进程的内存分配策略（localalloc 策略）
6	numactl -preferred=nodes	设置进程的内存分配策略（preferred 策略）
7	numactl -m nodes	设置进程的内存分配策略（membind 策略）
8	numactl -i nodes	设置进程的内存分配策略（interleave 策略）

5.2 NUMA 应用

通过一个实例来说明 NUMA 的应用，该实例有两个程序组成：（详见附录）

序号	程序	说明
1	test1	实现的功能是向操作系统申请 14GB 内存，然后进行 sleep。
2	test2	实现的功能是向操作系统申请 2GB 内存，然后访问内存进行写操作。

注：测定的操作系统一个 node 上只有 16GB 内存，故 Case 设计成 14GB+2GB。

程序的执行通过一个脚本 start_run.sh 来控制，流程如下所示：
1） 后台启动一个 sar 进程采集 CPU 信息
2） 后台启动 test1 进程（使用 numactl 绑定 CPU 和内存到 node0 上）
3） 后台启动 test2 进程
4） 循环采集 10 次 numastat 信息
5） 等待 test2 进程测定结束，杀掉 sar 进程和 test1 进程，程序运行结束。

为了体现 NUMA 的应用，对程序 test2 分两种情况来运行：
1） 不绑定
采集的 numa_miss 信息如下：

numa_miss 数	
程序运行前	程序运行后
30,917	43,858

采集的 numa 内存分配信息如下：（某一时刻）

```
[root@localhost testdir]# cat numastat.log | less
…
Per-node process memory usage (in MBs) for PID 9489 (test2)
                           Node 0          Node 1           Total
                  --------------- --------------- ---------------
Huge                         0.00            0.00            0.00
Heap                         0.00            0.00            0.00
Stack                        0.01            0.00            0.01
Private                   1072.08          976.36         2048.44
---------------- --------------- --------------- ---------------
Total                     1072.09          976.36         2048.45
…
```

采集的 sar 信息如下：

```
[root@localhost testdir]# sar -f sar.log -P ALL | less
…
Average:        all      5.25      0.00      3.34      0.00      0.00     91.41
Average:          0      0.00      0.00      0.06      0.00      0.00     99.94
Average:          1      0.06      0.00     19.24      0.00      0.00     80.71
Average:          2     62.67      0.00     19.69      0.00      0.00     17.64
Average:          3      0.00      0.00      0.00      0.00      0.00    100.00
Average:          4      0.06      0.00      0.06      0.00      0.00     99.88
Average:          5      0.06      0.00      0.00      0.00      0.00     99.94
Average:          6      0.00      0.00      0.00      0.00      0.00    100.00
Average:          7      0.00      0.00      0.06      0.00      0.00     99.94
Average:          8      0.00      0.00      0.00      0.00      0.00    100.00
Average:          9      0.00      0.00      0.06      0.00      0.00     99.94
Average:         10      0.24      0.00      0.77      0.00      0.00     99.00
Average:         11      0.00      0.00      0.00      0.00      0.00    100.00
```

说明：

a) 由 sar 采集的信息可以看出，test2 始终运行在 node0 上。

b) 由 numastat 采集的信息以及 NUMA 的默认策略，可知 test2 所需的内存原计划全部在 node0 上分配，但是在 node0 上分配 1GB 的内存后，发现内存不够了，被迫在 node1 上分配了 1GB 内存，从而导致产生 numa_miss。

c) 由于程序 test2 跨 node 访问了内存，性能表现较差。

2） 使用 numactl 绑定（numactl -N 1 -m 1 ./test2）

采集的 numa_miss 信息如下：

numa_miss 数	
程序运行前	程序运行后
43,858	43,858

采集的 numa 内存分配信息如下：（某一时刻）

```
[root@localhost testdir]# cat numastat.log | less
…
Per-node process memory usage (in MBs) for PID 6289 (test2)
                           Node 0          Node 1           Total
                  --------------- --------------- ---------------
Huge                         0.00            0.00            0.00
Heap                         0.00            0.00            0.00
Stack                        0.00            0.01            0.01
Private                      0.00         2011.52         2011.52
---------------  --------------- --------------- ---------------
Total                        0.00         2011.53         2011.53
…
```

采集的 sar 信息如下：

```
[root@localhost testdir]# sar -f sar.log -P ALL | less
…
Average:        all      3.90      0.00      4.18      0.00      0.00     91.92
Average:          0      0.00      0.00      0.17      0.00      0.00     99.83
Average:          1      0.00      0.00      0.00      0.00      0.00    100.00
Average:          2      0.08      0.00     26.89      0.00      0.00     73.02
Average:          3      0.08      0.00      0.00      0.00      0.00     99.92
Average:          4      0.00      0.00      0.00      0.00      0.00    100.00
Average:          5      0.08      0.00      0.00      0.00      0.00     99.92
Average:          6      0.00      0.00      0.00      0.00      0.00    100.00
Average:          7     46.63      0.00     21.90      0.00      0.00     31.47
Average:          8      0.00      0.00      0.00      0.00      0.00    100.00
Average:          9      0.00      0.00      0.00      0.00      0.00    100.00
Average:         10      0.08      0.00      1.17      0.00      0.00     98.75
Average:         11      0.00      0.00      0.08      0.00      0.00     99.92
```

说明：

a) 有 numastat 采集的信息和 sar 采集的信息可以看出，由于使用 numactl 绑定 test2 在 node1 上运行与分配内存，因此没有 numa_miss 产生。

b) 由于程序 test2 运行和访问在都在 node1，不存在跨 node 访问，性能表现较好。

需要注意的是：

由于 RHEL7 系提供了一个新特性：AUTOMATIC NUMA BALANCING（默认自动打开，配置文件为 /proc/sys/kernel/numa_balancing），会自动迁移 CPU 或 Memory 使其在同一个 node 上，从而减少跨 node 访问,减少 numa_miss 的产生，提供程序性能。

附录 测试代码

test1.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <errno.h>

int main()
{
    int *data;
    data=malloc(1024*1024*1024*14UL);
    printf("malloc 14GB\n");

    if(mlockall(MCL_CURRENT | MCL_FUTURE) == -1)
    {
        perror("mlockall failed:");
        exit(1);
    }

    printf("mlock 14GB, now sleep 100s\n");

    sleep(100);

    return 0;
}
```

test2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int iLoop = 0;
    int test_count = 10;
    long buffer_size = 1024 * 1024 *1024UL;

    char *testbuff;
    char *mcpdata;

    mcpdata = (char *) malloc(sizeof(char) * buffer_size);  //1GB
```

```
    memset(mcpdata, 0x41, buffer_size);

    iLoop = 0;
    for (iLoop = 0; iLoop < test_count; iLoop++) {

        testbuff = (char *) malloc(sizeof(char) * buffer_size);
        memcpy(testbuff, mcpdata, buffer_size);
        memset(testbuff, 0x42, buffer_size);

        free(testbuff);
    }
}
```

start_run.sh

```
#!/bin/bash

echo "===test start==="

taskset -c 10 sar  1   100  -o sar.log >& /dev/null &
PID_1=$!

numactl -N 0 -m 0 ./test1  &
PID_2=$!
sleep 3

./test2 &
PID_3=$!

for ((i=1;i<10;i=i+1))
do
    taskset -c 10 numastat  -mn -p test2  >> numastat.log
    sleep 1
done

wait ${PID_3}
taskset -c 10 numastat  -mn  >> numastat.log

kill -9 ${PID_1}
kill -9 ${PID_2}

echo "===test over==="
```