# 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：
2. 对称多处理器结构 (SMP：Symmetric Multi-Processor)
3. 海量并行处理结构 (MPP：Massive Parallel Processing)
4. 非一致存储访问结构 (NUMA：Non-Uniform Memory Access)
5. 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。
6. 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](http://detail.zol.com.cn/cpu_index/subcate28_125_list_1.html)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。
5. 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。
6. 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。
5. node：资源组，每个资源组内的CPU和内存是几乎相等。
6. 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。
7. 节点访问远程内存的过程如下：（CPU0访问MEM1为例）
8. CPU0通过QPI先向CPU1发起访问请求
9. 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
10. 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。

1. 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。

1. 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。

## 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。

1. 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。

1. 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。

## 应用对比

1. 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。

1. 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。

1. 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 |

1. 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。

1. 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：

1. 由sar采集的信息可以看出，test2始终运行在node0上。
2. 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。
3. 由于程序test2跨node访问了内存，性能表现较差。
4. 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：

1. 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。
2. 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===" |