MPI + OpenMP 混合并行编程模型应用研究

Research on development of mixed mode MPI + OpenMP applications

冯　云　周淑秋　（首都师范大学信息工程学院 北京 100037）

摘要：多处理器结点集群在高性能计算市场上日趋流行，如何在多处理器上编写出高效的并行代码成为研究的热点。MPI + OpenMP 为多处理器结点集群提供了一种有效的并行策略，结点内部共享内存空间编程模式适合 OpenMP 并行，消息传递模型 MPI 被用在集群的结点与结点之间，这样就实现了并行的层次结构化。

关键词：MPI　OpenMP　MPI + OpenMP

1　引言

MPI 是集群计算中广为流行的编程平台。但是在很多情况下，采用纯的 MPI 消息传递编程模式并不能在这种多处理器构成的集群上取得理想的性能。为了结合分布式内存结构和共享式内存结构两者的优势，人们提出了分布式/共享内存层次结构。OpenMP 是共享存储编程的实际工业标准，分布式/共享内存层次结构用 OpenMP + MPI 实现应用更为广泛。OpenMP + MPI 这种混合编程模式提供结点内和结点间的两级并行，能充分利用共享存储模型和消息传递模型的优点，有效地改善系统的性能。

2　OpenMP + MPI 混合编程模式

使用混合编程模式的模型结构图如图 1 在每个 MPI 进程中可以在#pragma omp parallel 编译制导所标示的区域内产生线程级的并行而在区域之外仍然是单线程。混合编程模型可以充分利用两种编程模式的优点 MPI 可以解决多处理器间的粗粒度通信而 OpenMP 提供轻量级线程可以和好地解决每个多处理器计算机内部各处理器间的交互。大多数混合模式应用是一种层次模型 MPI 并行位于顶层 OpenMP 位于底层。比如处理一个二维数组可以先把它分割成结点个子数组每个进程处理其中一个子数组而子数组可以进一步被划分给若干个线程。这种模型很好的映射了多处理器计算机组成的集群体系结构 MPI 并行在结点间 OpenMP 并行在结点内部。也有部分应用是不符合这种层次模型的比如说消息传递模型用于相对易实现的代码中而共享内存并行用于消息传递模型难以实现的代码中还有计算和通信的重叠问题等。

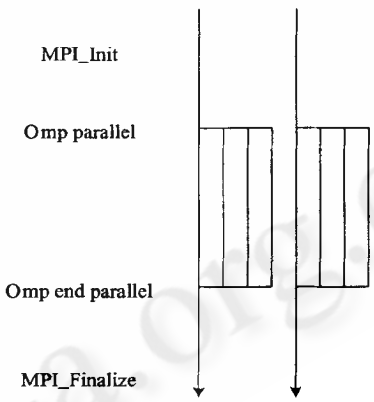

图 1　MPI + OpenMP 模型

2.1　混合编程模式的实现

MPI + OpenMP 混合编程模型易于实现。MPI 的初始化和结束还是调用 MPI_Init 和 MPI_Finalize 其间会有一些 OpenMP 并行区域。虽然很多 MPI 实现都是线程安全的为了保证代码的可移植性 MPI 调用应该在代码的线程串行区域内。但是这通常会导致一些问题因为大多数包含 OpenMP 的代码并行位于 MPI 并行之下因此大多数 MPI 调用都是在 OpenMP 并行区域之外。如果 MPI 调用发生在 OpenMP 并行区域内则根据代码的性质存在于 Critical，Master 或者 Single 相应的区域内。特别应当注意 Single 区域，因为不同的线程都可能成功的执行区域内代码。理想情况下，每个 MPI 进程内

线程的个数由 omp_set_num_threads(n)来决定而不是采用 OMP_NUM_THREADS 环境变量,因为这样系统的可移植性更好。

在 MPI2 标准中,MPI_Init_thread 代替了 MPI_Init 为多线程的进程提供初始化功能。它的 C 语言函数 API 如下:

int MPI_Init_threads(int * argc, char *((*argv)[]), int required, int* provided)

对于其中 required 有以下几种取值,它们级别依次递增。在实际的应用中,调用此函数返回的 provided 可能会低于 required 的级别:

(1) MPI_THREAD_SINGLE 在 MPI 应用中只能是由主线程进行 MPI 调用并且是在其它进程休眠的状态下。它是最简单的 MPI + OpenMP 的模式,也是唯一不在 MPI2 中定义的,它的特点是易于编程,没有负载均衡问题。它的编程模式如下:

```
#pragma omp parallel
{
#pragma omp for
for( )
……
}
MPI_XXX( );
#pragma omp parallel
{
#pragma omp for
for( )
}
```

(2) MPI_THREAD_FUNNELED 进程可以是多线程的,但是仅有主线程可以进行 MPI 调用。

MPI 调用要在 Parallel 以外的区域;如果 MPI 调用需要在 Parallel 区域中,#pragma omp master 来指定主线程,而#pragma omp master 没有任何同步,因此还需要用#pragma omp barrier 在#pragma omp master 前后来进行同步。其它线程的数据或内存空间在 MPI 调用前后是可以使用的。

```
#pragma omp barrier
#pragma omp master
  MPI_XXX( );
#pragma omp barrier
```

(3) MPI_THREAD_SERIALIZED 进程可以是多线程的,可以有多个线程进行 MPI 调用。但是在同一时刻,只能有一个线程进行 MPI 调用(它可以是主线程也可以不是)也就是说线程进行 MPI 调用必须是串行的。#pragma omp single 仅在结束的时候同步,因此在#pragma omp single 开始之前需要用#pragma omp barrier 进行同步。

```
#pragma omp barrier
#pragma omp single
  MPI_XXX( );
```

(4) MPI_THREAD_MULTIPLE 多个线程可以进行 MPI 调用,没有严格的限制。

当编写混合模式应用程序时在实际应用中具体选择哪一种模式,要根据具体情况而定。MPI_THREAD_SINGLE 易于编程,没有负载均衡问题。MPI_THREAD_FUNNELED 和 MPI_THREAD_SERIALIZED 比较适用于动态任务分布编程,而对于问题域的分解编程并不适合,并且存在负载均衡的问题,MPI_THREAD_MULTIPLE 对于具体线程情况没有严加界定,可以根据需要灵活应用。

2.2 OpenMP + MPI 混合编程模式的优缺点分析

2.2.1 优点分析

(1) 有效的改善 MPI 代码可扩展性。MPI 代码不易进行扩展的一个重要原因就是负载均衡。它的一些不规则的应用都会存在负载不均的问题。采用混合编程模式,能够实现更好的并行粒度。MPI 仅仅负责结点间的通信,实行粗粒度并行;OpenMP 实现结点内部的并行,因为 OpenMP 不存在负载均衡问题,从而提高了性能。

(2) 数据拷贝问题。数据拷贝常常受到内存的限制,而且由于全局通信其可扩展性也较差。在纯的 MPI 应用中,每个结点的内存被分成处理器个数大小。而混合模型可以对整个结点的内存进行处理,可以实现更加理想的问题域。

(3) MPI 实现的不易扩展。在某些情况下,MPI 应用实现的性能并不随处理器数量的增加而提高,而是有一个最优值。这个时候,使用混合编程模式会比较有益,因为可以用 OpenMP 线程来替代进程这样就可以减少所需进程数量,从而运行理想数目的 MPI 进程,再用 OpenMP 进一步分解任务,使得所有处理器高效运行。

(4) 带宽和延迟限制问题。减少结点间的消息但是却增加了消息的长度。在简单的通信中,比如,在某时仅允许一个结点发送/接收一条消息,消息带宽是没有影响的,整个延时会降低因为此时消息的数量少了。在更复杂的情况下,允许消息并发的发送/接收,长消息数量的减少会产生不良影响。

(5) 通信与计算的重叠。大多数 MPI 实现如:MPICH 和 LAM,都是使用单线程实现。这种单线程的实现可以避免同步和上下文转换的开销,但是它不能将通信和计算分开。因此,即使是在有多个处理器的系统上,单个的 MPI 进程不能同时进行通信和计算。MPI + OpenMP 混合模型可以选择主线程或指定一个线程进行通信,而其它的线程执行计算的部分。

2.2.2 缺点分析

虽然在很多情况下,使用 OpenMP + MPI 混合编程模式的程序效率更高。但是它也存在着一些不足,比如对于纯 MPI 应用,每个参与通信 CPU 可以饱和结点间的带宽而 MPI + OpenMP 若分出一个线程进行通信则难以做到。同时,OpenMP 也要产生系统开销如:线程 fork/join ,为了达到同步清洗 cache,空间局部性会更糟糕。采用混合编程模型的程序能否取得更高的效率取决于以下几种因素:采用混合编程模型的程序往往有着更小的通信开销[3],是否可以用轻量级的线程来代替重量级的 MPI 进程来实现并行化,还包括其它一些因素,比如 MPI 进程数量限制,MPI 进程负载均衡问题,数据拷贝的内存限制因素。在面对实际应用的时候,一定要考虑 MPI 和 OpenMP 这两者的结合是否能够提供一个更加优化的并行平台,怎样利用两者实现并行化。

3 实验分析

3.1 实验基本环境

实验环境为 4 个结点构成的集群,其中每个结点有两个处理器,处理器为 IntelXeon3.06GHZ,256M 内存两条,512K 二级缓存,前端总线 800MHz。采用 Infiniband 网络,PCI - x 的 HCA 卡。MPI 实现采用的是 Ohio 州大学的 MVAPICH[4]。

3.2 Jacobi 迭代实验

Jacobi 迭代是一种比较常见的迭代方法,它得到的新值是原来旧值点的平均。由于它的局部性好,可以取得很高的并行性,是并行计算中常见的一个例子。参加迭代的数据按块分割后,各块之间除了相邻的数据需要通信以外,在每个块的内部都可以独立进行计算。而且,随着计算量的增大,通信的开销相对于计算来说比例会降低。这将有利于提高并行效率。在 Jacobi.c 中,主要的计算时间都耗在那两个二重 for 循环中,因此考虑用 OpenMP 对其进行并行化。

在 4 个双 CPU 机器的集群上对程序进行了测试。对于仅用 MPI 实现的 C 程序,在程序执行的时候,每个进程被分配到 CPU 上,在 2 个进程的情况下,仅被分在一个结点机上,在 8 个进程的情况下,进程均匀的分配到 4 个结点上,每个结点上有两个进程。对于 MPI + OpenMP 实现的 C 程序,每个结点上仅分配一个进程,每个进程内有两个线程。

实验结果如图 2,3 所示。图 2 是两种编程模式的加速比图,图 3 是两种编程模式执行 Jacobi 的时间图。采用 MPI + OpenMP 混合编程模式的性能要明显高于单纯的 MPI 编程模式。

3.3 通信与计算重叠实验

大多数 MPI 实现如 LAM、MPICH、MVAPICH 都是使用单线程实现。这种单线程的实现可以避免同步和上下文转换的开销,但是它不能将通信和计算分开[5]。因此,在多处理器系统上,单个 MPI 进程不能同时通信和计算。在编写程序的时候,可以利用多线程编程实现计算和通信的重叠。计算和通信分别用不同的线程来完成。本实验采用耗 CPU 量大的 FFT 为例,每个进程内有两个线程,一个用于通信,一个用于进行 FFT 计算。实现结果如图 4 所示,采用 OpenMP + MPI 的带宽比纯 MPI 带宽提高了近 30%。(FFT order 进行 FFT 变换数据的大小为 N,N = 2order)

3.4 试验结果分析

以上的两个实验都是利用 MPI + OpenMP 编程模型典型应用,从实验的结果来看,MPI + OpenMP 相比较于纯的 MPI 来说性能有所改善。在实验一中,采用混合编程模式可以减少通信的进程个数,这样计算量/通信量的比值更高,更低的通信开销导致更高的加速比。在实验二中,区别于纯 MPI 编程模式,对于 FFT 计算和通信分别采用两个线程并行处理,使得计算与通信能够真正重叠。但有以下问题应当注意:当线程数大于物理处理器的数量时,会因为线程竞争总线带宽

而导致性能下降。当采用线程通信时，应注意将发送和接收操作都是同一线程号线程来进行处理。当处理通信/计算重叠问题时候，要注意负载均衡问题。

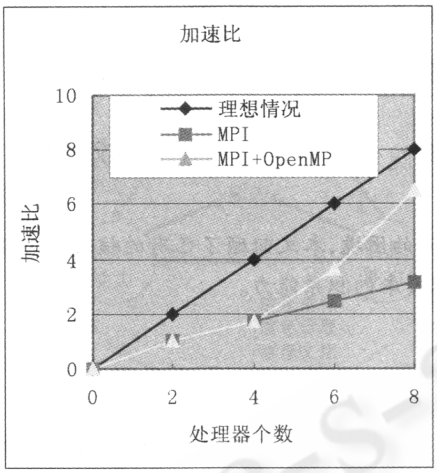

图 2　Jacobi 加速比

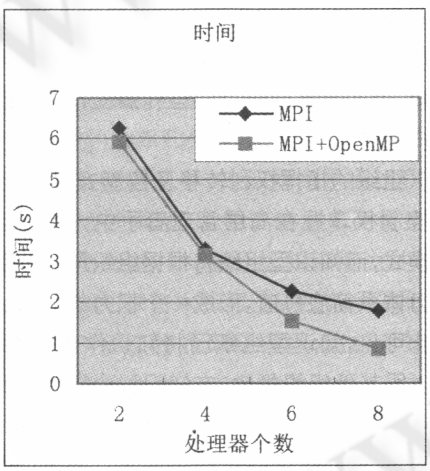

图 3　Jacobi 时间

4　总结

OpenMP + MPI 这种混合编程模型相比于单纯的 MPI 消息传递编程模型更能充分利用多处理器计算机集群的体系结构特点，在某些情况下可以有效的改善集群的性能，它为多处理器构成的计算机集群提供了一种不错的并行策略，但是这种编程模式并不能适应所有的代码。因此，在实际的应用中，是否选择混合编程模式还需要针对实际情况而定。

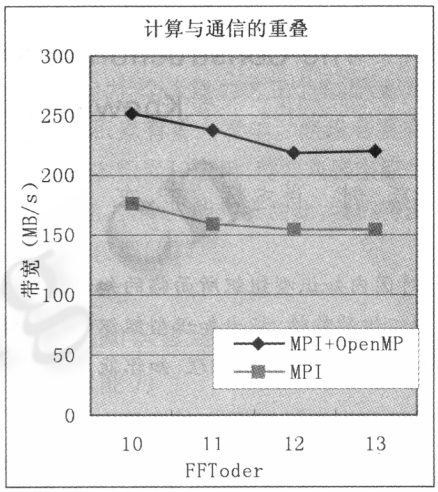

图 4

参考文献

1　Ananth Grama, George Karypis, Vipin Kumar, Anshul Gupta Introduction to Parallel Computing 2nd edition, [M] Addison Wesley

2　Hybrid Programming: Combining MPI with OpenMP Brown CCV Parallel Programming Workshop Spring 2005 www. ccv. brown. edu/Events/Hybrid_mar05. pdf

3　Michael J. Quinn parallel programming in c with MPI and OpenMP[M] McGraw – Hill higer education

4　http://nowlab. cse. ohio – state. edu/projects/mpi – iba/

5　USFMPI: A Multi – threaded Implementation of MPI for Linux Clusters[C]In Proceedings of the 15th International Conference on Parallel and Distributed Computing and Systems, Marina del Rey, CA 2003. [J]Scientific Programming, Vol. 9, No 2 – 3, 2001, 83 – 98.