基于CMP多核集群的混合并行编程技术研究

王文义 王春霞 王 杰
(中原工学院并行处理技术研究所 郑州 450007)

摘 要 高性能科学计算(High Performance Science Computing,简称HPC)是验证某些理论和测试计算机系统处理能力的一种有效的实验手段。鉴于目前CMP(Chip Multi-processor)多核集群已变得越来越普及,尝试对由MPI和OpenMP两种不同并行编程技术构成的混合编程模式做一些实验性的研究。通过对程序执行时间和加速比的实验数据分析,可以看出在多核和多节点集群上采用细粒度的混合并行编程方法较单一使用MPI并行编程方法更加合理和高效,也更能体现出系统硬软件的特性与优势。

关键词 高性能计算,CMP多核集群,墙钟时间,MPI+OpenMP混合并行编程
中图法分类号 TP39 **文献标识码** A

Research on Hybrid Parallel Programming Technique Based on CMP Multi-cure Cluster

WANG Wen-yi WANG Chun-xia WANG Jie
(Institute of Parallel Processing Technology, Zhongyuan Institute of Technology, Zhengzhou 450007, China)

Abstract When validating some theory and testing computer systems processing capacity, high-performance scientific computing is an effective experimental means. Currently, as CMP multi-core clusters become increasingly common, this article attempted to do some experimental studies to the MPI and OpenMP two different parallel programming technique consisting of hybrid programming model. Through analyzing the experimental data of the program execution time and the speedup, it can be seen that in multi-core and multi-node clusters using fine-grained Hybrid parallel programming method than the single using MPI parallel programming method will be more rational and efficient, so it can also better reflect the features and advantages of system hardware and software.

Keywords High performance computing, CMP multi-core cluster, Wall clock time, Hybrid parallel programming with MPICH and OpenMP

1 前言

传统集群通常采用基于消息传递的编程模式MPICH(Message Passing Interface Chameleon,简称MPI),而随着如图1所示的片上多处理器CMP(Chip Multi-processor)即多核处理器(Multi-core Processor)的问世,多核集群以其低廉成本和高效的并行处理能力迅速普及,于是过去单一的MPI编程模式,或者是耗费较大成本研发成功但却远未完成生命周期的那些高性能计算(HPC)并行应用程序,显然已不能有效发挥作用,甚至不能适应现有硬件资源的变化。众所周知,多核只是实现了硬件技术的飞跃,它只能在获得了相应软件技术的跟进与支持之后,才会真正发挥出优势,因为当集群节点变为多核结构时,若仍使用单一的MPI编程模式,就忽视了硬件的优势,相当于仍把多核作为单核使用,大马拉小车,势必造成很大的资源浪费。

由于现代多核集群系统涉及到分布式存储和共享存储两种并行体系结构,因此我们可称其为混合并行计算系统。并行编程难[1]本来就是阻碍并行计算普及的主要原因之一,更不用说在基于多核集群的混合并行环境中,还存在着多级并行化问题,即节点间并行、芯片间并行、芯片内多个核心并行,同时涉及到消息传递和共享变量两种并行编程模式,其编程难度更大。因此,如何为混合并行计算环境选择合适的并行编程模式和程序设计方法,成为HPC领域研究的热点和难点,也是本文探讨的一个主要问题。

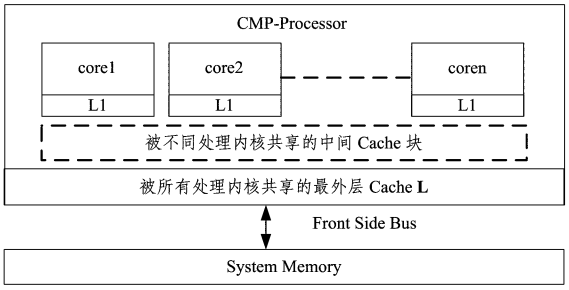

图1 同构多核处理器结构

到稿日期:2013-04-25 返修日期:2013-06-29 本文受国家863计划项目(2008AA01A315),河南省基础与前沿技术研究项目(122300410314)资助。

王文义(1947—),男,教授,硕士生导师,主要研究方向为高性能计算与并行处理技术,E-mail:wwy@zzu.edu.cn;王春霞 讲师,主要研究方向为高性能计算;王 杰 硕士生,主要研究方向为多核程序设计技术。

2 MPI 与 OpenMP 性能简介

MPI 并行编程技术[2,3]已被广为接受和熟悉。它主要采用粗粒度级别并行，将任务分配给集群系统的所有计算机，以完成并行计算。

OpenMP 并行编程技术[4-6]采用的是 fork-join 并行执行模式，OpenMP 程序在执行时首先从一个独立的主进程开始，当遇到大计算量时会生成一个并行域(Parallel Region)用以完成并行计算任务，这时将会派生出多个线程并行执行，线程执行完成后它们被同步或者中断，于是只剩下主进程在执行，如此类推。OpenMP 主要针对细粒度的循环级并行，即在循环中将每次循环分配给各个线程执行，主要针对于一台独立的计算机。

MPI 与 OpenMP 的性能比较[7]见表 1。

表 1 MPI 与 OpenMP 性能比较

种 属	OpenMP	MPI
并行粒度	线程级并行	进程级并行
存储方式	共享存储	分布式存储
数据分配	隐式分配	显式分配
编程复杂度	相对简单。可充分利用共享存储结构的特点，避免了消息传递的开销。但数据的放置策略不当可能会引发其它问题，并行化的循环粒度过小会增加系统开销等	模型复杂。需要分析与划分应用程序问题，并将它们映射到分布式进程集合，细粒度的并行要求大量的通信，存在通信延迟大和负载不平衡两个主要问题，调试 MPI 程序麻烦，MPI 程序可靠性差，一个进程出问题，将引起整个程序出错
可扩展性	共享存储意味着只适应于 SMP、DSM 机器，不适合于集群，可扩展性差	可扩展性好，适合于各种机器
并行化	串行程序转换为并行程序时无须对代码作大的改动。支持细粒度并行和粗粒度并行，但尤其适合针对细粒度的循环级并行	并行化改进需要大量修改原有的串行代码

3 MPI＋OpenMP 混合编程模式

针对多核集群存储结构的特点，选择 MPI＋OpenMP 混合编程方法，即将节点间的分布式存储和节点内的共享存储两种并行编程模式结合起来使用，由 MPI 接口实现节点间的并行任务分解与分配，而由 OpenMP 编程模式实现节点内部的大量并行计算任务。

人们真正对 MPI＋OpenMP 混合编程模式的研究历史尚较短，还需要通过大量的实验来探索。MPI＋OpenMP 混合模式架构如图 2 所示。

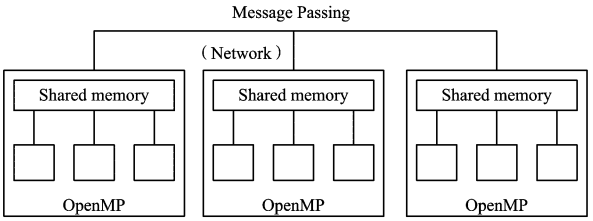

图 2 MPI＋OpenMP 混合模型架构

根据 OpenMP 制导指令和给定目标任务的不同，可使生成的并行域的任务粒度大小也不同，因此可以把混合模式分为以下两种。

3.1 MPI＋OpenMP 混合编程细粒度模式

在图 3 所示的细粒度混合模式(Fine-grain hybrid model)中，只需要在现有的 MPI 程序代码中进行增量并行，也就是在 MPI 进程内的主循环部分采用 OpenMP 模式，其意义就在于可以完成 MPI 的进程和 OpenMP 的线程[4,5]。

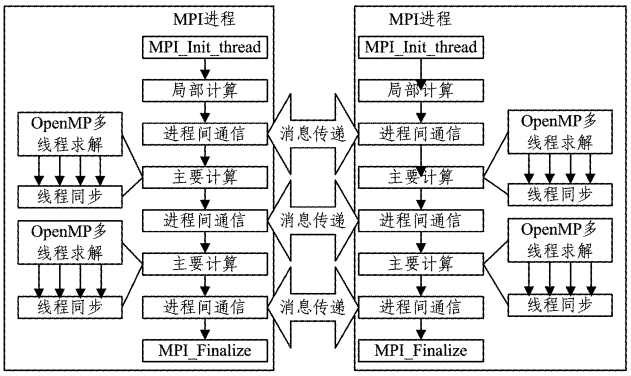

图 3 混合编程细粒度模式

在细粒度混合模式中，把对 MPI 消息传递接口的节点间和 OpenMP 多线程节点内的任务处理相结合，从而实现节点间和节点内的两级并行。可以看出多核集群的每个节点上只有一个 MPI 进程，而且还独立地承担初始化和做一些局部少量的计算以及节点间的通信的功能。当 MPI 的进程遇到程序中需要大量计算的任务时，再使用 penMP 制导[8,9]指令对任务进行分解，这时通过该 MPI 进程分配给各个节点的主线程将派生出多个从线程来执行，并由多线程在并行区域中并行求解。

3.2 MPI＋OpenMP 粗粒度混合模式

实际上，图 4 描述的粗粒度混合编程模式(Coarsegrain hybrid model)并不符合增量式的并行思想。因为，在这种方式下考虑到整个程序的并行方案，线程的创建是在 MPI 进程生成和初始化之后，而在 MPI 节点间通信时需要设置线程同步[2]，还需要考虑 MPI 通信与线程的交互作用和保证程序语义正确的 barrier 制导等问题。因此所增加的线程间同步的开销可能会抵消掉并行化所带来的好处。另外，程序中 Master 制导和 do 制导之外的其它共享结构的引入，也将加重负载不平衡的程度，这会导致增加程序本身的开销，引起性能的降低。

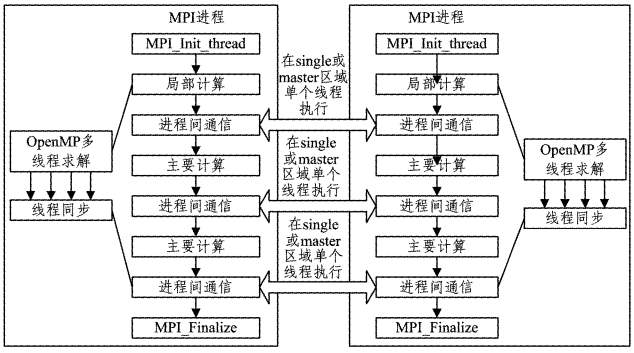

图 4 混合编程粗粒度模式

由图 4 可以看出 MPI 的调用发生在多线程并行区域内，进程间的通信由并行区域内某个或某些线程来完成。在这种模式中，线程的通信与计算是并行执行的，它和细粒度混合编

程模式一样，每个节点的顶层都是 MPI 的进程，不同的则是 MPI 进程在启动 OpenMP 并行区域的多线程中，当节点间进行通信时，需要在 Critical、Single 或 Master 相应的区域内进行消息传递，以保证通信安全，因此这种通信就需要设置线程同步，但这会使同步点增加，增加的同步开销可能会抵消并行化所带来的优势。计算任务全部完成后，先结束 OpenMP 并行域的多线程，然后再结束 MPI 进程。

比较上述两种混合编程模式，笔者认为在多核集群中还是比较适合采用细粒度的方式来实现节点内并行，同时细粒度混合模式的代码编写也相对比较容易些。

4 混合编程模式的多核集群测试

测试平台使用多核 InfiniBand 集群中的 2 个节点，每节点由 2 个 Intel Core i7 四核八线程的处理器组成[10]。共采用 7 种不同矩阵规模分别对 MPI 并行程序和 MPI+OpenMP 混合并行程序[11-13]做矩阵乘测试，每种规模连续测试 3 次，以取得的平均 Wall Clock 时间为准。

4.1 MPI+OpenMP 细粒度模式矩阵相乘的主要代码（注：MPI 代码略）

```
void solve(int num_thread)
{
  omp_set_num_threads(num_thread);
   //矩阵初始化
  ……
}
int _tmain(int argc,char * * argv)
{
  int processors;//处理器数
  int myid;//当前正在运行的进程的标识号
  int local_m_begin,local_m_end;
  int num_thread;//线程数目
  while(scanf("%d",&num_thread)! =EOF)
{
  solve(num_thread);
}
  start=omp_get_wtime();//获取时间
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
MPI_Comm_size(MPI_COMM_WORLD,&processors);
//把矩阵 A 按行分成 P 份，分别计算
local_m_begin=(M/processors) * myid;
local_m_end=(M/processors) * (myid+1)-1;
#pragma omp parallel
{
  #pragma omp for private(i,j,k)
  for(i=local_m_begin;i<=local_m_end;i++)
  {
    for(k=0;k<P;k++)
    {
      for(j=0;j<N;j++)
        {
          C[i][k]+=A[i][j] * B[j][k];
        }
      }
    }
  }
  //收集数据
MPI_Gather(&C[local_m_begin][0],(M/processors) * P,MPI_
INT,C,(M/processors) * P,MPI_INT,0,MPI_COMM_WORLD);
end=omp_get_wtime();
cout≪"MPI+OPENMP 在"≪myid≪"号机器的时间耗时为："≪end
-start≪"\n";
  MPI_Finalize();
  return 0;
}
```

两种测试程序的测试结果列于表 2 中。

表 2 测试结果（时间单位：秒）

计算规模	编程模式	运行测试 Wall Clock 时间			平均 Wall Clock 时间	加速比
		第 1 次	第 2 次	第 3 次		
400 * 400	MPI	0.620416	0.615277	0.606601	0.614098	0.632
	MPI+OpenMP	0.764782	0.417187	0.530137	0.570702	0.68
800 * 800	MPI	2.17472	2.14644	2.14495	2.15537	1.46
	MPI+OpenMP	1.06647	1.36373	1.55666	1.32895	2.37
1600 * 1600	MPI	18.213	17.2659	17.2659	17.2659	2.01
	MPI+OpenMP	9.25432	9.35401	9.30509	9.30447	3.70
2000 * 2000	MPI	30.6864	30.6983	32.9784	31.45437	1.20
	MPI+OpenMP	16.8428	16.7723	16.7665	16.79387	3.73
2400 * 2400	MPI	59.4388	57.9278	59.9441	59.10357	2.00
	MPI+OpenMP	31.7355	31.6845	31.4729	31.63097	3.74
3000 * 3000	MPI	123.422	120.329	119.848	121.1997	2.01
	MPI+OpenMP	64.3627	64.1355	64.3952	64.2978	3.78
4000 * 4000	MPI	298.554	295.537	292.958	295.683	1.88
	MPI+OpenMP	145.517	145.483	145.658	145.5537	3.82

4.2 测试结果分析

从表 2 中数据可以看出，当矩阵规模＜800 * 800 时，MPI 和 MPI+OpenMP 两种模式程序在多核集群上的运行时间上并没有太大差别。而当矩阵规模达到 1600 * 1600 并继续增大时，后一种模式程序的运行时间和加速比就明显优于前者。这是因为矩阵规模在＜1600 * 1600 时，进程之间的通信量尚比较小，也可以说是因为 MPI 的优势尚未被很好地发挥出来，同样这时线程间的通信量也没有充分体现出来。当矩阵规模逐渐加大，任务的划分块也随之变大，将加大数据的广播、读写等操作量，使得 MPI 进程间的通信开销增加，同时随着分解的任务块增加，节点内部 OpenMP 的并行域所承担的计算任务也随之增加，这就有助于充分发挥节点内的多线程

的任务并行化，于是 MPI＋OpenMP 混合编程的优势就逐渐显露出来，无论从运行时间还是从加速比来看，它都要明显优于 MPI 并行程序。

另外，从图 5 中 MPI 与 MPI＋OpenMP 运行加速比曲线可看出，当矩阵规模＞1600＊1600 时 MPI 程序的加速比趋势呈现缓慢下移，而 MPI＋OpenMP 混合程序也趋于平缓的走势，这是因为程序在运行时受到了硬件、通信和节点数目等因素的限制所致，但从图 6 所示的 MPI 和 MPI＋OpenMP 程序运行的平均墙钟时间上可以看出，MPI＋OpenMP 模式程序的运算速度依然要明显高于单一 MPI 模式。

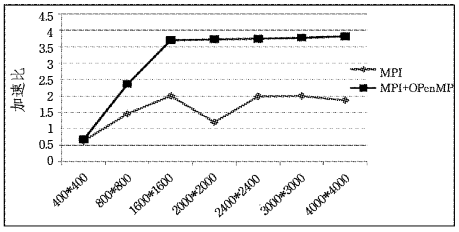

图 5　MPI 与 MPI＋OpenMP 运行加速比曲线

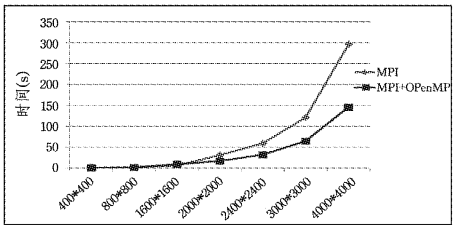

图 6　MPI 与 MPI＋OpenMP 程序运行时间曲线

结束语　笔者的体会是，对于多核集群系统而言，目前在并行应用程序设计中，使用 MPI＋OpenMP 混合编程模式更接近系统硬件的结构性能，因此它不失为一个可行的途径。

毕竟条件有限，实验所用的矩阵规模和节点数还远远不够，实验结果也难于以点带全。我们只是把实验结果和体会写出来，以飨读者。

参 考 文 献

[1] Hwang K. Advanced Computer Architecture: Parallelism Scalability Programmability[M]. New York: McGraw-Hill Inc., 1993

[2] Group W, Skjellum E L A. Using MPI——Portable Parallel Programming with the Massage Interface(Second Edition)[M]. CambridgeMassachusetts, London, England: The MITPress, 1999

[3] 都志辉. 高性能计算之并行编程技术 MPI 并行程序设计[M]. 北京：清华大学出版社，2001

[4] Robert S A J. Multi-core Programming: Increasing performance Through Software Multi-threading [M]. 李宝峰，富弘毅，李韬，译. 北京：电子工业出版社，2007：145-283

[5] Chandra R, Dagum L, Kohr D, et al. Menon: Parallel programming in OpenMP[M]. Morgan Kaufmann Publisher, Inc., San Francisco, CA, USA, 2001

[6] OpenMP C and C++ Application Program Interface[OL]. version3. May 2008. http://www.openmp.org

[7] Brown R. Performance and Productivity Comparison Between OpenMP and MPI[J]. Int Parallel Prog, 2007, 35: 441-458

[8] 章隆兵，吴少刚，蔡飞. 适合集群 OpenMP 系统的制导扩展[J]. 计算机学报，2004，27(8)：1129-1135

[9] 陈永健. OpenMP 编译与优化技术研究[D]. 北京：清华大学，2004

[10] Core i7 QPI 技术解密[OL]. http://wenku.baidu.com/view/63e77d160b4e767 f5acfcea -e. html. Wang D T. The CELL microprocessor. Real World Technologies, 2005

[11] Smith L, Bull M. Development of mixed mode MPI＋OpenMP applications [J]. Scientific Programming, 2001, 9: 83-98

[12] Lusk E, Chan A. Early Experiments with the OpenMP/MPI Hybrid Programming Model [C]// IWOMP'08 Proceedings of the 4th International Conference on Open MP in a New era of Parallelism. Springer, 2008, 5004: 36-47

[13] 叶晓敏. 基于多核处理器并行 EDA 算法研究[D]. 复旦大学

(上接第 10 页)

[12] Wang T, Wei T, Zou W. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution [C]// Network and Distributed System Security Symposium. USA: Internet Society, 2009

[13] Godefroid P, Levin M, Molnar D. Automated whitebox fuzz testing[C]// NDSS. 2008

[14] Hamadi Y. Disolver: A Distributed Constraint Solver[R]. Technical Report MSR-TR-2003-91. Microsoft Research, December 2003

[15] Ganesh V, Dill D. A Decision Procedure for Bit-vectors and Arrays [M]. Computer Aided Verification. Berlin: Springer-verlag, 2007: 524-536

[16] Moura L, Bjorer N. Z3: An Efficient SMT solver. Tools and Algorithms for the Construction and Analysis of Systems[M]. Berlin: Springer-Verlag, 2008: 337-340

[17] Godefroid P, Klarlund N, Sen K. DART: Directed Automated Random Testing[C]// Proceedings of PLDI'2005 (ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation). Chicago, June 2005: 213-223

[18] Sen K, Marinov D, Agha G. CUTE: A Concolic Unit Testing Engine for C[C]// European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering. USA: ACM Press, 2005: 263-272

[19] Godefroid P. Compositional Dynamic Test Generation[C]// Proceedings of POPL'2007 (34th ACM Symposium on Principles of Programming Languages). Nice, January 2007: 47-54

[20] Molnar D, Wagner D. Catchconv: Symbolic Execution and Runtime Type Inference for Integer Conversion Errors[R]. USA: Unversity of California Berkeley, 2007

[21] Fuzzgrind[OL]. http://esec-lab.sogeti.com/pages/Fuzzgrind