

中国科学技术大学

硕士学位论文

MapReduce并行编程模式的应用与研究

姓名：吴晓伟

申请学位级别：硕士

专业：计算机软件与理论

指导教师：郑启龙

20090501

摘 要

当前,科学与工程计算中大规模数据处理的需求与日俱增。与此同时,高性能并行机的发展和硬件价格的下降使得高性能机器得到了广泛推广,与之相伴的是各种并行编程模式和并行编程语言的纷纷出现,并行计算的发展为各学科带来福音的同时也带来了一系列的问题。并行机体系结构以及并行编程语言和模式的多样性为并行应用的开发带来了极大的困难,增加了普通程序员的编程难度。因此,简化并行编程,提高并行程序开发效率就成为了一个关键的问题。

很多并行编程模型和语言抽象层次太低,导致程序员要关心很多底层实现细节。而 MapReduce 模式具有高层次的抽象,在 MapReduce 运行时系统的支持下程序员只需要关注对具体数据的处理,以及数据相关性的处理,极大的降低了并程序的开发难度。但是除了网络搜索领域外,如何使用 MapReduce 对其它领域(尤其是数值计算领域)的具体应用进行设计与分析还没有系统、详细的研究和阐述。为此,本文围绕这些环节展开研究工作,具体研究工作与成果如下:

第一,研究了并行算法空间中由任务组织、由数据分解组织和由数据流组织这几种并行问题组织原则与 MapReduce 模式的关系,综合 MapReduce 模式以及几种原则的特性,给出了如何用 MapReduce 对这几种不同原则描述的问题进行表述。

第二,将任务分解、分治策路、流水线等并行问题分析模式与 MapReduce 模式相结合,用 MapReduce 来描述这些并行模型,并且对每个模式中出现的典型应用或算法进行分析。并通过对 MapReduce 在这些模式上的应用和对具体应用的分析,给出了使用 MapReduce 模式分析问题的步骤及方法。

第三,提出了在用 MapReduce 模式解决问题时所用到的关键数据结构,一方面是输入数据的表示,其中包括输入数据和输入数据的特征,并结合矩阵数据划分的分析过程,给出了数据结构的设计要点和方法。另一方面,分析了 key 值的作用,研究了 key 值与 MPI 中通信模式的关系,给出了选择和设计 key 值的原则。

第四,将 MapReduce 引入数值计算领域,在 HPMR 上研究了数值计算的应用。使用 MapReduce 模式在 HPMR 上设计和实现了以矩阵计算为典型代表的若干数值计算应用,包括矩阵的 LU 分解与 QR 分解。在对具体问题的实现和分析过程中研究了使用 MapReduce 的步骤和各种数据结构的设计,体现了这

种模式的简洁性和易用性。

实验测试表明，在 HPMR 上运用我们所提出的方法和步骤使用 MapReduce 模式来分析和解决有关并行计算问题有着很好的简洁性和易用性，且具有良好的运行时性能。

关键词：MapReduce 数值计算 并行编程模式 HPMR

ABSTRACT

At present, the demands of science and engineering which need to deal with large amounts of data are growing strongly. At the same time, the development of high performance parallel computer and the decline of hardware prices make the high-performance machines used widely, and various parallel programming models and parallel programming languages have emerged. However, the development of parallel computing is good news for the various subjects, at the same time it also brought a series of problems. The diversity of parallel computer architecture, parallel programming language and parallel programming models cause enormous trouble for the development of parallel applications and increase the difficulty of the programming. Therefore, simplifying parallel programming and increasing the efficiency of parallel program development becomes the key problem.

Many parallel programming models and language's abstraction level is so low that the programmers should be concerned about the details of the bottom implement. MapReduce is a high level abstraction model. Supporting by the run-time system, in MapReduce the programmers only need to pay attention to the specific data processing and the processing of the data relevance, it greatly reduces the difficulty of parallel program development. In addition to the field of web search, however, there is no systematic research and description of how to use MapReduce to design and analysis applications in other areas (especially the domain of numerical calculation). Hence, this paper carried out in these aspects of research, specific research work and achievements are as follows:

First, we research the relationship between MapReduce model and organizing principles in the parallel algorithm space such as organized by tasks, organized by data decomposition and organized by data flow. By understanding the characteristics of MapReduce and algorithm space model, given how to use describe problems of these principles with MapReduce.

Second, using MapReduce to describe task partition, divide and conquer, pipeline with, and analysis the typical application or algorithm. Via the MapReduce model in these applications and analysis of specific applications, given the steps and methods witch used MapReduce model to solve the problem.

Third, proposed the key data structure using in MapReduce. On the one hand, it's the input data, which should include input data and the characteristics of the

input data, and according to the analysis of division of matrix data, given the key points and methods of the data structure design. On the other hand, by analyzing the key roles and researching the relations between key and MPI communication mode, give the principles of selection and design of key.

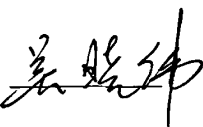
Forth, we introduced MapReduce into numerical computation, and implemented LU decomposition and QR decomposition on HPMR. In the process of analysis and implement the algorithms we checked our method which use to solve problems with MapReduce, and this models embodies this simplicity and ease of use.

Through the practical tests, using our method and procedures to analyze and solve problems with MapReduce on HPMR is simplicity and ease of use, and has a good run-time performance.

Key Words: MapRedcue, numerical computation, parallel programming model, HPMR

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文,是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外,论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名: 

签字日期: 2009.6.1

中国科学技术大学学位论文授权使用声明

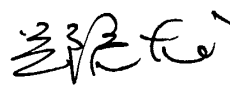
作为申请学位的条件之一,学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权,即:学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版,允许论文被查阅和借阅,可以将学位论文编入有关数据库进行检索,可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

保密的学位论文在解密后也遵守此规定。

☒ 公开 ☐ 保密 (____年)

作者签名: 

签字日期: 2009.6.1

导师签名: 

签字日期: 2009.6.1

第1章 绪论

1.1 并行机体系结构的发展趋势

随着各学科的发展和工业以及服务业的发展,当前,无论是在物理、化学、生物等理论研究领域还是在制造业、军工、电信服务等工业产业领域都需要强大的计算能力和对海量数据的分析、分析能力。很多情况下,对计算和数据处理的要求不仅是能够完成工作得到结果,而且在计算速度、实时性和精确度方面都有很严格的要求和限制。传统的单处理器和串行程序在很多方面已经显得捉襟见肘,在很多大规模数据处理和计算中已经显得无能为力,在这种情况下并行机和并行编程应运而生。在并行机的体系结构方面,并行机一般需要多台计算机或者多个处理器进行协同工作,每个处理器或者计算机完成一部分任务;在并行算法方面,先要把整个计算任务划分成几个阶段或者几部分,它们之间可以并发的或者交叉并发的进行计算,然后把它们分布到各个处理器或者机器上。通常并行计算的过程需要进行数据交换,一般采用网络通信或者共享存储的方法来进行完成,通信过程也体现了多机协作的特点。当所有计算都完成以后,再由某个机器完成对各部分结果的合并工作,从而得出最终结果。

出于对各种不同因素的考虑,并行计算机在体系结构上也存在着很大的差别。有的通过网络发送消息的方式实现各处理器之间的通信,有的通过在公用存储器中使用共享变量的机制来实现通信;有些多核处理器设计为同构的,有些则设计为异构的。并行体系机构可以抽象的分为并行计算机结构模型和并行计算机访存模型。其中并行计算机结构模型又可分为单指令多数据流(SIMD)(Eric E. Johnson et al. 1998)以及多指令多数据流(MIMD)(Howard Jay Siegel et al. 1978)。多指令多数据流的计算机模型又可分为PVP, MPP, COW, DSM等计算机;根据计算机的访存模型可以将并行计算机分为UMA、NUMA、COMA、CC-NUMA、NORMA。

二十世纪七十年代 Illiac-IV 系统的出现标志着并行机时代的到来,在过去的将近四十年里并行机的体系结构不断发展,这期间每个时代都有其代表性的体系结构。七十年代到九十年代的二十年中向量机成为了并行计算机界的主流架构,Cray 公司出的 Cray 系列机器是当时的代表性机器,其中典型机器有 C-90。九十年代初 Cray 公司被 SGI 公司兼并,从此向量机退出了主流行列。随之出现的是 MPP 结构的并行机,从二十世纪九十年代初到中期,MPP 系统逐渐崭露头角并显示出取代和超越向量机的趋势。其中著名的 Intel Paragon、IBM SP2、Intel TFLOPS 和我国的曙光-1000 等分布式存储的 MIMD 计算机就出现在那个时代。MPP 的高端机器的代表有 Intel 公司的 ASCI Red 和 SGI Cray 公司的 T3E900,它

们的运算能力都达到了万亿次。90年代中期集群系统的概念被提出,1995年以后各种体系结构(PVP, MPP, SMP等)并存,共同发展。但是集群系统迅速崛起占领了大多数并行机的席位,目前我国的曙光4000以及自主研发的KD-50-I万亿次机器基本都属于多核多处理器的集群系统。随着计算需求的增长各种计算能力更强的大型机在不断刷新着前人的记录,IBM的超级计算机Roadrunner已经达到了1.026PetaFLOPS的计算能力,而我国则由曙光公司推出了自己的计算能力达到230万亿次的曙光5000超百万亿次大型计算机。

高性能的同时意味着高价格和高功耗,虽然计算机硬件设备的价格有了显著的下降,但是一台高性能并行计算机其价格高达百万到千万,这样的价格对于研究所和学校等研究机构来说还是很难承受的。而应用规模和精度的不断增加,并程序规模的不断增大都导致各行业对高性能计算机硬件环境的需求越来越迫切。而集群系统可以使用廉价的兼容计算机通过网络进行连接,可以统一使用各主机的资源、统一调度、协调处理以实现一个高性能系统。在这一方面集群系统在降低成本的同时,还提供了较高的计算能力,受到广泛的欢迎。而由我国自主研发的KD-50-I万亿次高性能计算机具有体积小、功耗低、成本低等诸多的优点,它必将在高性能领域占据一席之地。

近些年来,处理器厂商也注意到,受到处理工艺的限制,通过提高频率来提高处理器计算能力的做法已经遇到了很大困难。多核处理器在这种情况下出现了,从真假双核之争到现在四核乃至八核十六核及更多核的处理器器的出现,使得多核技术逐渐成熟并出现了同构多核和异构多核。目前,由多核机器组成的集群系统已经成为了当前高性能计算机界的领军人物,但是如何高效的利用这种并行机系统是当前研究人员和编程人员急需解决的一个重要问题。

1.2 并行编程模式的发展

并行机体系结构的多样性也为并行编程带来了很大的麻烦,如何方便、合理、高效的利用这些机器成为了急需解决的问题。采用什么样的体系结构能够带来更快的速度和更高的精度,而采用什么样的编程方式才能使其发挥到极致,这一系列的问题一直伴随着并行计算的发展过程。

就目前比较流行的编程模式来看,从宏观上讲,MPI是基于消息传递的程序设计模式,因此它更能适应底层基于网络通信的并行架构;而OpenMP的通信方式则采取的是共享内存法,因此它更应用共享存储架构下的并行体系结构。最核心的就是以下两个问题:(1)设计什么样的编程模型能够充分发挥特定的并行体系结构所带来的优势;(2)不同的编程模型都有自己的特点,特点越多意味着编程越复杂,怎样掩盖这种复杂度提高并行开发效率。

总之, 为了方便合理高效的利用并行计算环境, 必须要有一种并程序设模式与体系结构相配合。在并行编程语言中 OpenMP 主要应用于共享地址空间的计算机上, 还有一大类编程语言, 它们的实现可以映射到任意并行计算机上, 这样的语言可以分为两大类: local view languages 和 global view languages。MPI (Message Passing Interface), 它作为一个库已经成为了 local view languages 的一个标准, 并得到了广泛的支持和应用, 几乎所有的集群系统都会安装 MPI。比 MPI 更高层的抽象语言有 Co-Array Fortran, Unified Parallel C, 和 Titanium 等, 这些 local view languages 一直发展到现在。此外 MPI 与 OpenMP 的混合编程也一度成为热点。

让普通的串行程序员来学习这些语言开发并行应用无疑增加的它们的负担, 而且并行程序运行过程中存在的数据和任务的分配, 进程或线程间的消息通信, 容错处理等各种细节都会使程序员不能集中精力解决并行任务的表述。在这种情况下 Google 提出了 MapReduce 模型, 它将用户需要处理的工作分解为两个功能模块 Map 和 Reduce。用户通过指定的 Map 函数对输入的一系列键值对(key/value 对), 产生中间键值对; MapReduce 底层实现将所有具有相同键值的键值对合并到一起, 并传递给 Reduce 函数; 用户通过指定的 Reduce 函数处理得到的一系列键值相同的中间结果对, 这种模式极大的简化了并行编程。各种不同形式的 MapReduce 系统纷纷出现, 对其应用的研究也逐步展开。

1.3 本文的工作和章节安排

1.3.1 本文的工作内容及其成果

本文首先介绍了目前并行机的体系结构以及当前主流的并行编程语言和模式。接着介绍了 MapReduce 模式的几种实现, 特别我们自己实现的 HPMR。然后, 介绍了 MapReduce 模式与其它并行编程模式的关系, 以及 MapReduce 在这些模式上的应用。最后, 将 MapReduce 引入数值计算领域, 并在 HPMR 上实现了两个矩阵分解算法。

本文主要研究了各种不同并程序设模式和并行编程模式与 MapReduce 模式的关系, 研究了如何将 MapReduce 与这些模式相结合来描述不同的并行应用, 在较高的层面对 MapReduce 的应用进行了研究, 提出了使用 MapReduce 分析问题和设计应用的方法步骤, 以及数据结构的设计等关键点。然后, 在面向高性能计算的 MapReduce 平台—HPMR 上实现了以矩阵计算为典型代表的并行数值计算, 探讨了 MapReduce 模型应用于高性能数值计算的可行性及相关方法。实践表明, 作为一种新型的并行分布式编程模型, MapReduce 具有较高的并行表述抽象性, 可有效地降低并行编程的难度, 提升并行编程生产率。

1.3.2 本文的组织

第二章，我们将介绍并行体系结构的基本知识和目前流行的并行体系结构；对并行编程模式和行编程语言的基本知识进行介绍，这一章的内容是并行编程的基础知识。

第三章，介绍 MapReduce 编程模式的起源，同时介绍了几种不同体系结构上的 MapReduce 实现，包括我们自己的 MapReduce 系统——HPMR，以及它们的编程特点和运行过程，展示了 MapReduce 模式的几个关键概念和系统的关键部分。

第四章，介绍了 MapReduce 模式与其它并行编程模式的联系和区别，研究了如何用 MapReduce 来表述这些模式，给出了使用 MapReduce 的方法和步骤，介绍了数据结构的设计以及 key 值的设计，结合矩阵运算进行了深入分析，并将 MapReduce 引入数值计算领域。

第五章，给出了运用 MapReduce 模式在 HPMR 上设计并实现矩阵 LU 分解和 QR 分解的详细步骤，以及在数值计算中使用 MapReduce 编程模式的关键点。

第六章，对 HPMR 上实现的两个矩阵分解程序进行性能分析，并针对两种不同模式的程序体现出的不同性能给出分析。

第七章，总结本文做出的贡献，并对 MapReduce 模式在数值计算领域的应用提出一些展望。

第 2 章 并行机体系结构及并行程序开发环境

2.1 并行机体系结构简介

随着并行机技术的发展,并行机的体系结构也在不断的更新、发展。以下将对主流的并行体系结构做简要的介绍。

2.1.1 SMP

SMP 是 Symmetrical Multi-Processing 的简称,也就是对称多处理机系统,是指由多个处理器(CPU)做成的计算机系统,这些处理器之间共享总线和存储系统。SMP 是一种被广泛应用的并行机体系结构,在这种架构中,一台计算机又同时运行统一操作系统单一副本的多个处理器组成,处理器间共享内存等其他计算机资源。典型的 SMP 系统使用商业化微处理器(具有片上或外置高速缓存),它们经由高速总线(或者交叉开关)连向共享存储器。从使用者的角度来看,一台 SMP 机器跟一台普通的单机一模一样,多个处理器的管理都有操作系统来完成。由于 SMP 系统具有对称性,所以所有 CPU 对本地资源的访问都是平等的,操作系统负责任务的分配、多 CPU 的调度以及系统中各种资源的分配,已达到负载平衡,使整个系统拥有很高的性能。对称使得 SMP 系统能够开拓较高的并行度,但是共享存储限制了系统中的处理器不能太多(一般少于 64 个),而且总线和交叉开关互联一旦做成也难于进行新的扩展。图 2.1 为一个典型的 SMP 体系结构图。

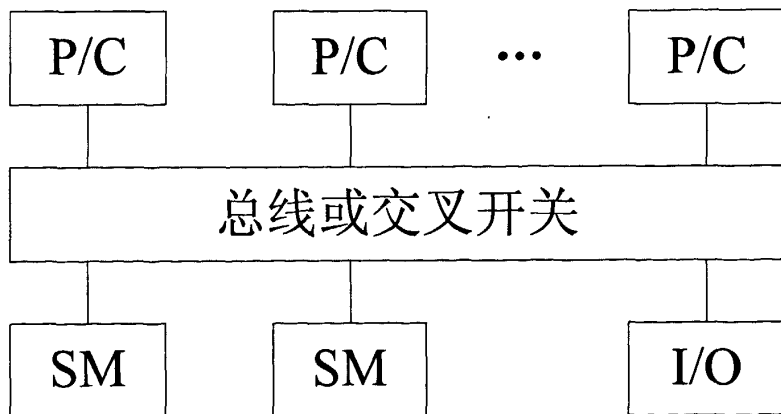


图 2.1 SMP 结构图

2.1.2 Cluster

集群(Cluster)(John C. Hunter et al. 1995)由多个成为节点的计算机组成,

一般情况下这些节点通过网络连接起来作为一个系统来工作，这个系统为用户提供计算能力、数据存储和通信资源等。同时，集群中每个节点都以一个完整的系统，可以提供所有的单机服务，在集群中各节点相互协作，作为一个整体它提供了强大的性能和极高的可用性。在高可用性集群中，单个节点的故障并不会影像整个集群的工作，通过备份等技术高可用集群可以提供连续不间断的各种服务。最近几年集群系统发展的异常迅猛，随着多核处理器的出现，多核集群也在大量增长，已经成了当前集群系统的主要形式。图 2.2 为一个典型的集群系统结构图。

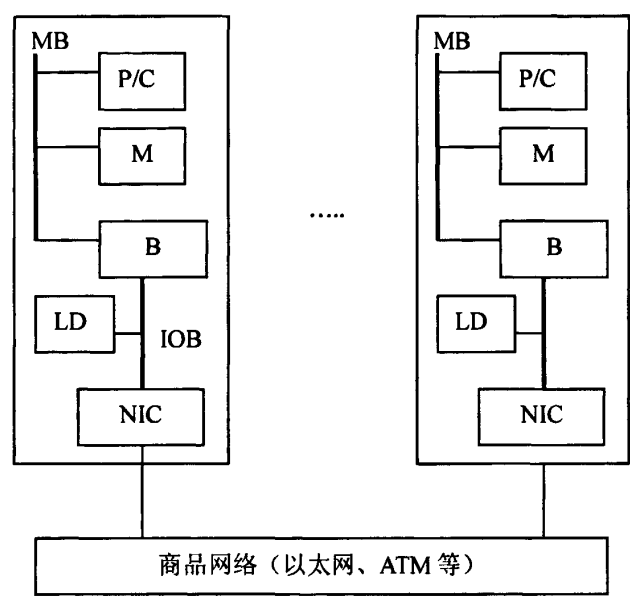


图 2.2 cluster 结构图

2.1.3 其它并行机体系结构

SMP 和 Cluster 结构是当前最流行的两种并行机结构，除此以外还有很多其它体系结构。其中比较著名的有并行向量处理机（PVP），这样的系统中包含少量专门设计定制的高性能向量处理器 VP，每个 VP 至少具有 1Gflops 的处理能力，其结构如图 2.3 所示。系统中使用了专门设计的高带宽交叉开关网络将 VP 连向共享存储模块，存储器能够以每秒兆字节的速度向处理器提供数据。这样的机器通常不使用高速缓存，而是使用大量的向量寄存器和指令缓冲器。著名 PVP 机器有 Cray C-90、Cray T-90、NEC SX-4，而我国国防科技大学自主研发的银河系列大型机也属于 PVP 机器。

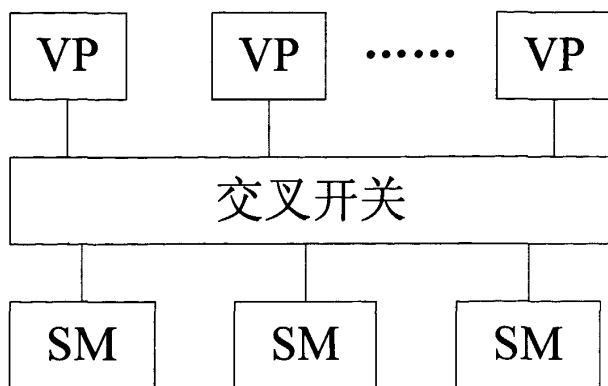


图 2.3 PVP 结构图

大规模并行处理机（MPP）一般是指超大型计算机系统（Very Large-scale），它的处理节点一般采用商业化微处理器，系统中有物理上的分布式存储器，采用高通信带宽和低延迟的互联网络，能够扩放至成百上千乃至上万个处理器，它是一种异步的 MIMD 机器，程序由多个进程组成，每个进程都有其私有地址空间，进程间采用传递消息的方式互相作用。MPP 的主要应用是科学计算、工程模拟和信号处理等以计算为主的领域。如今 cluster 系统迅速发展，尤其是多核多处理器的集群迅猛发展，计算能力和可扩放性突飞猛进，MPP 与 cluster 之间的界限越来越模糊，而且集群相对于 MPP 有着更高的性价比，因此集群在发展可扩放并行机计算机方面呼声很高。

共享存储多处理机（DSM）的高速缓存目录 DIR 用以支持分布式高速缓存的一致性。DSM 和 SMP 的主要差别是，DSM 在物理上有分布在各个节点中的局部存储器，它们形成了一个共享的存储器。而对于程序员来讲系统的硬件和软件共同提供了一个单地址的变成空间，相对于 MPP 来说 DSM 的优越性在于编程比较容易。Stanford DASH、Cray T3D 和 SGI/Cray Origin2000 都属于此类结构。

2.1.4 多核处理器

处理器在不断提升频率以达到更高性能的同时，它的功耗和散热量也在不断增加，并且成为未来高性能体系结构设计的重要制约因素（R. Kumar et al. 2003）。例如，Pentium-4 系列的 CPU 的功耗已经超过了 50W，预计到 2015 年这一系列 CPU 的功耗将达到 300W。功耗和散热量的增加不仅在封装形式、风扇、供电和空调方面提出了更多的要求，而且会导致高耗能系统频繁出现各种问题（Linda Wilson et al. 2002）。

计算机产业在 2005 年发生了重大的转变，Intel 跟随着 IBM 的 Power 4 芯片和 Sun 公司的 Niagara 芯片的产生发出了这样的声明：微处理器性能的提升从今

以后要依赖多处理器 (processors) 或者核 (cores) (K. Asanovic et al. 2006)。

CMP (Single-Chip Multiprocessor) (Kunle Olukotun et al. 1996; Multi-Core processors 2005; Ron Kalla et al. 2004) 是典型的多核处理器结构, 几年各大芯片厂商都大力推出自己的多核处理器, 并占领了大部分的市场。硬件加工工艺的不断进步和应用软件规模的不断增大是微处理器向多核方向发展的两个重要因素。

由于电路设计复杂性及目前很多应用程序都有一定的并行性, 目前超标量处理器的复杂设计导致芯片内的延迟越来越大, 功耗也不断增加。通过提高处理器主频来提高性能的做法带来了诸如散热和加工工艺等多方面的问题, 因此多核技术应运而生。这项技术在一个处理器内增加较低频率的处理器个数, 减少处理器的复杂度, 以提高性能降低功耗。

通超标量处理器相比, 多核处理器有着独特的优势。在六发射动态调度超标量处理器与四核两发射的多核处理器上作比较: 如果运行相同的串行程序, 前者比后者的性能要高 30% 左右; 但运行有一定细粒度并行的程序是前者之比后者高 10% 左右; 如果运行同时具有粗粒度和细粒度的并行程序后者性能则比前者高了 50%——100%。

目前, CMP 在性能和价格等方面都有不错的表现, 无论是在科研工作中还是在日应用中, CMP 都逐步替代了传统的单核 CPU, 成为了处理器市场上的主力军。

在当前的商用 CMP 中, Intel 和 AMD 都推出了各自的双核、四核处理器, 更多核的处理器还在开发之中。这些多核处理器已经成为了处理器市场上的主力军。通常将核数比较少的处理器 (64 个核以下) 称为多核 (multi-core), 将核的数目超过 64 的处理器称为众核 (many-core) 处理器, 这些多核处理器的每个核都是一样的没有从属与控制的关系。而另一种对众核的定义则是一个通用 CPU 加上多个如图形处理器 GPU 之类的专用处理单元所构成的处理器, 有关 GPU 的内容我们将在后面介绍。现在来看另一个著名的处理器 Cell, 它就是由一个通用核加上几个专用处理单元组成的。

Cell (B. Flachs et al. 2005; D. Pham et al. 2005; M. Gschwind et al. 2006; H. P. Hofstee 2005; J. A. Kahle et al. 2005) 处理器是由日本索尼、新力电脑娱乐、东芝、美国国际商业机器 (IBM) 公司联合开发还有用于 PlayStation3 游戏系统的高速运算处理器。与传统的多处理器或多核体系结构不同, 它不是由相同的几个核组成, 它包含一个通用的性能很高的 PowerPC 核, 这个核用来控制其它八个叫做协作核——SPE (synergistic processing element) 的简单 SIMD 核, 每个 SPE 包含一个协作处理单元——SPU (synergistic processing unit) 和本地存储器。每个 SPE 所要运行的指令和要处理的数据都存储在自己的本地内存中。PowerPC 核除

了完成虚拟地址和物理地址的转化外还要管理没有高速缓存（cache）一致性的 SPE 本地内存的内容，因此 PowerPC 核要为每个 SPE 完成 DMA 的初始化，以便其能将程序和数据到 DRAM 读入本地内存。DMA 初始化一旦完成，PowerPC 核就会启动各个 SPE（SamuelWilliams et al. 2006）。Cell 的概要图如图 2.4 所示：

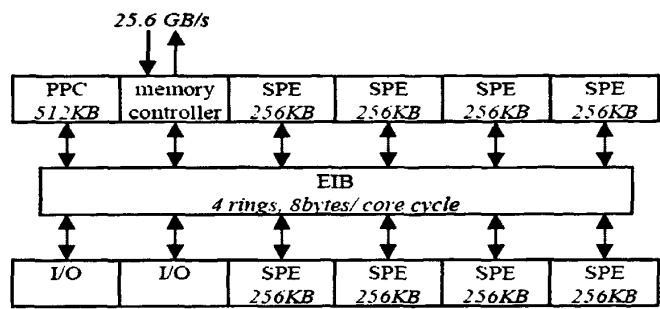


图 2.4 Cell 结构概要图

注:八个 SPE 和一个 PowerPC 核，一个内存控制器，两个 I/O 控制器，每个 SPE 有自己的本地内存，它所运行的程序都存储在这里面

2.2 并行程序开发环境

MPI, OpenMP 以及 MPI+OpenMP 混合编程是目前流行的三种开发模式。除此之外还存在着全局地址空间划分语言（Partitioned Global Address Space Languages）也就是 PGAS 语言，三种主要的 PGAS 语言是 Co-Array Fortran, Unified Parallel C 和 Titanium。Titanium 被扩展成为相应的 Fortran, C 和 Java（Calvin Lin et al. 2008）。另外还有 ZPL 和 NESL 等全局空间视角语言。每种语言根据自身设计上的特点分别适合于特定的运行平台。在运行的操作系统上，它们主要也是支持类 Unix 系统，如 Linux。当然微软也发布了一套专为高性能计算(HPC)领域设计的集群服务器操作系统“Windows HPC Server 2008”。但是在高性能计算领域类 Unix 系统依然牢牢占据着不可动摇的统治地位。

2.2.1 MPI 简介

MPI（Message Passing Interface）（Snir M. et al. 1996; Michael J. Quinn 2005; 陈国良 et al. 2004）是一个消息传递接口的标准，用于开发基于消息传递的并行程序，其目的是为用户提供一个实际可用的、可移植的、高效的和灵活的消息传递接口库。MPI 是一个库，而不是一门语言，因此对 MPI 的使用必须和特定的语言结合起来进行。在 MPI-1 中明确提出了 MPI 与 FORTRAN 77 和 C 语言的绑

定, 并且给出了通用接口和针对 FORTRAN 77 与 C 语言的专用借口说明。MPI-1 的成功说明了 MPI 选择的语言绑定策略是正确的、可行的。MPI 程序可以不做修改的运行在单机 PC 机、单台工作站、机群系统和 MPP 系统上, 无论这些机器由谁制造, 使用何种操作系统, 这也是它被广泛接受的原因。

MPI 并行程序由多个进程组成, 程序运行时一组固定数目的进程会在程序初始化的过程中生成。每个进程都会属于一个或多个通信域 (Communicator)。通信域由通信组、通信上下文等内容组成。进程组就是通信域中所有进程的集合, 该进程组的每个进程分别由整数 0、1、……、N-1 (N 为通信组的进程总数) 来标识, 这些进程通过相互收发消息来进行通信。一个 MPI 并行程序至少需要六个最基本的函数, 他们分别完成 MPI 环境的初始化和结束、获取当前进程编号及指定通信域的进程数、消息的发送和接收等功能。目前主流的 MPI 软件有 MPICH (MPICH2 2005) 和 Lam MPI (LamMPI 2007), Boost.MPI (Douglas Gregor et al. 2007) 是支持泛型消息传递的 MPI 库。

2.2.2 OpenMP 简介

OpenMP (OpenMP 2008; Rohit Chandra et al. 2000; 陈国良 et al. 2004) (OpenMP standard) 标准是共享存储体系结构上的一个编程模型。它是基于编译制导的, 具有简单、移植性好和可扩展等优点, 是共享存储系统编程的一个工业标准。OpenMP 并不是一种新的语言, 它是对基本语言 (如 Fortran 77、Fortran 99、C、C++等) 的扩展。OpenMP 为自己制定了一个目标, 总结概括起来为四方面: ①标准性: 即在共享存储系统平台上提供一个标准; ②简洁实用: 建立一个简单、有限的编译制导语句集, 使得主要的几个并行操作只需 3、4 个命令就可以完成; ③使用方便: 能够将串行程序并行化, 不仅能完成粗粒度的并行, 也能完成细粒度并行; ④可移植性: 支持 FORTRAN、C/C++语言。

OpenMP 规范了一系列的编译制导 (Compiler Directive)、运行库 (Runtime Library) 和环境变量 (Environment Variables), 并将他们提供给用户, 用于与编译器和运行系统进行交流, 使得用户能够控制并行程序的行为, 完成用户需要的操作。例如用户可以显式地指定代码中需要并行执行的部分, 该部分又将由多少个线程并行执行等, 但是用户也承担了进程的创建、同步和通信以及各种变量的存储属性的设定等任务。OpenMP 的体系结构如图 2.5 所示:

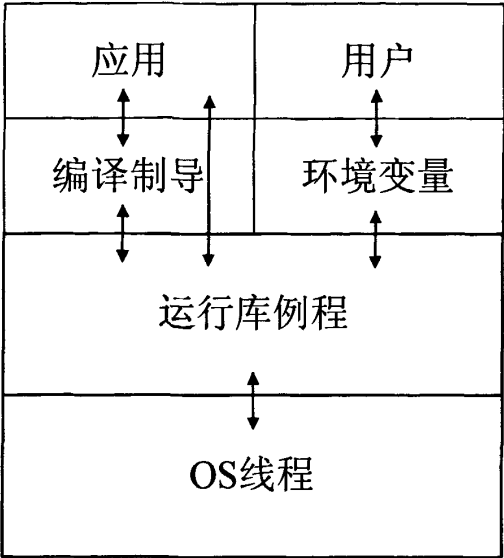


图 2.5 OpenMP 体系结构

OpenMP 是基于线程的并行编程模型 (Y. Charlie Hu et al. 2000), 一个进程由多个线程组成。OpenMP 使用 Fork-Join 并行执行模型。OpenMP 程序初始时只有一个主线程, 在程序运行过程中主线程会在进入并行域时创建 (FORK) 子线程, 个子线程并行执行并行域中的代码; 当离开并行域时, 所有子线程都会被同步或者被中断 (JOIN), 只有主线程继续执行。所有的子线程都共享主线程的内存变量等运行时上下文资源, 它们通过设定主线程中的共享变量来进行通信。通过应用编程接口用户还可以动态改变不同并行域中所产生的线程数, 能够很好的控制并行粒度。

2.2.3 混合编程

随着硬件技术的进步, 利用小规模配置的 SMP 机器构建的 SMP 集群大量出现如 IBM ASCI White、曙光 3000 等 (CHEN Yong et al. 2004), 而随着多核技术的发展, 利用多核处理器构建的多核机群也随之大量出现如曙光 4000、曙光 5000 等。在这种类型的集群中多个 CPU 或者多个核之间通过高速总线或者交叉开关连接起来, 并通过它们访问共享的内存区域和 I/O 设备等。机群中的各个节点间通过通信网络 (例如 Myrinet、Infiniband、以太网) 连接起来, 并且节点间可以通过消息传递进行通信 (Tanaka Y et al. 1998)。单一的编程模式很难同时发挥集群和多核多处理的优势。由并行体系结构的交叉使用而引发了并行开发模型的改变, 混合编程 (Lorna Smith and Mark Bull 2001) 就是其中一例。

所谓混合编程就是将 MPI 和 OpenMP 在一个并行应用程序开发中同时使用。兼顾了消息传递以及共享内存的编程模型, 适合于开发基于类似 CC-NUMA 并

行体系结构的应用程序，可以发挥这种体系结构的长处，提高并行体系结构的使用率，加速了应用程序的运行性能。因为 CC-NUMA 是 SMP+Cluster 或者 Multicore+Cluster 的混合体，这个并行平台是由多个 SMP 机器或者多核机器组成的机群系统。而基于 Cluster 的体系结构适合使用 MPI 并行模型，而在单个节点中适合使用 OpenMP 编程模型 (Frank Cappello, Deniel Etiemble 2000)。对于混合编程来说，使用 MPI 来开发节点间的并行性，同时使用 OpenMP 来开发节点内的并行性。也就是说先由 MPI 程序在各计算节点上产生进程，再由 OpenMP 在节点内部产生线程。即以 MPI 为主程序框架，在每个 MPI 进程内部可以并行化的区域采用 OpenMP 来对其进行编译制导的并行化。混合编程的正确性、高效性和简洁性已经被证明。混合程序运行时刻进程与线程间的从属关系如图 2.6 所示：

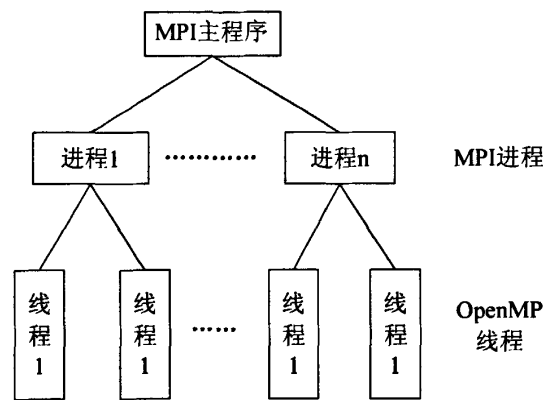


图 2.6 混合编程模型中进程与线程的关系

2.2.4 全局地址空间划分语言

分布式存储上的编程一般都采用消息传递，但是完全可以在分布式存储模型之上进行更高级的抽象。在过去的二十年里，一些研究小组已经建立了这样的抽象，并且开发了一系列的全局地址空间划分语言 (Partitioned Global Address Space Languages) 也就是 PGAS 语言。

全局地址空间是指某种语言把分布式机器的内存抽象成单一的虚拟内存。这些语言不提供共享内存，因为并没有指望硬件会保证内存的一致性。但是全局地址空间并没有赋予程序员定义全局数据结构的能力，一个分布式存储模型上的改进就是程序员必须自己维护一组分散的数据结构，然而这些数据结构的行为却像全局的数据结构一样。在 PGAS 语言中，程序员的精力放在一个进程的行为上，而且必须区分本地和非本地数据，但是这些语言剔除了消息传递使编程简单了；编译器产生所有的通信调用，每个调用都与程序员指定的非本地引用对应。此外，

这些语言底层通信采用了比消息传送更有效的单边通信。

三种主要的 PGAS 语言是 Co-Array Fortran, Unified Parallel C 和 Titanium, 其中 Titanium 被扩展成为相应的 Fortran, C 和 Java 版本。

Co-Array Fortran (CAF) 是 Numrich 和 Reid 在 1990 年用 Fortran 语言扩展而得到的, 其中最主要的一个语言扩展是 co-array。co-array 是一种处理器间通信的机制 (“co” 是 communication (通信) 的缩写)。利用 Fortran 语言对所有的数组引用都加括号的特点, CAF 对 co-array 的变量引用都加上方括号。co-array 的标识说明这些变量所在的内存是分布在每一个进程中的, 在 CAF 叫镜像 (image)。编译器为这些远程引用产生通信调用, 极大的简化了编程复杂度。最初的 CAF 实现采用了 Cray-proprietary, 单边通信库——Shmem, 但是现在的实现使用了 ARMCI 和 GASNet, 它们都是单边通信库的标准。CAF 为分布式存储的并行计算机提供了干净的优雅的用户指定的接口。

统一的并行 C 语言 (Unified Parallel C) 即 UPC, 它由 El-Ghazawi, Carlson 和 Draper 组成的小组在 2000 年左右开发出来。它为程序员提供了全局地址空间视图。但是不像 CAF 那样, 当 UPC 数组变量被声明为共享 (shared) 的时候, 它会被分布到程序实例的内存中去。线性数组中的元素会被循环或者按块循环分配。循环或者按块循环分配会达到较好的负载平衡, 但是同时也降低了本地性。UPC 程序员也可以为了重新获得本地性而直接把数组的划分分配到进程中去。

Titanium 在 UC Berkeley 由 Kathy Yelick 领导的一个的研究小组在分布式存储的并行计算机上的对 Java 进行扩展开发了一种语言这就是 Titanium (Ti)。它的存储模型跟 UPC 类似, 也很像其它的 PGAS 语言, 它使用单边通信库来生成通信代码。Ti 与其它 PGAS 语言最大的差别就是它是面向对象的语言。而它与 Java 最大的不同就是添加了区域的概念, 这个概念致使安全的面向效率的内存管理从而取代了垃圾回收机制。Ti 中最能提高效率的特性是它的分解循环——foreach, 这个特性同时减轻了程序员和编译器的工作。Titanium 能通过使用路障 (barriers) 和一个叫做 single (它被所有进程共享) 的共享变量概念强制保证全局同步。

2.2.5 全局视角语言

在全局视角语言中, 一个并行问题会被当作一个整体来看待, 而不会关注程序的某个进程或某一个数据划分。这种高抽象层次的语言支持隐式并行, 也就是说它的编译时会接管进程创建、进程间通信和同步等操作, 极大的简化了程序员的工作。全局视角语言包括 ZPL 和 NESL 等。

ZPL (Zope Public License) (C. Lin and L. Snyder 1994; B.L. Chamberlain, S.-E. Choi, et al. 2000; ZPL 2009) 的开发目标是使程序拥有良好的可移植性和性能并

且描述简洁。它把重点放在数组的定义和处理上,同时也拥有隐式并行的一切特点。作为“数组语言”ZPL 将整个数组作为一个处理单元,例如在对向量 A 中的元素值全部加一是通常写成下面的形式: $A:=A+1$ 或者 $A+=1$, 而向量 A 中各个元素的具体操作则由编译器安排为并行执行。在 ZPL 中区域(regions)是一个很重要的核心概念,ZPL 要求所有的数组操作都在一个区域内进行,例如: $[1..n/2] A:=A+1$, 方括号里面的内容代表了一个区域这条语句的作用是对数组 A 中前二分之一元素值加一。区域也可以用来描述多维数组中的元素,例如 $[1..8, 1..8]$ 则表示这个区域是某个 8 阶二维方阵。

NESL (Nested Parallel Language) (BLELLOCH, G. E. et al. 1993; Guy E. Blelloch et al. 1989; Guy E. Blelloch 1995; John Greiner and Guy E. Blelloch 1995) 是上世纪九十年代中期由 Guy Blelloch 领导的研究小组在 Carnegie Mellon 所实现的一种高层次的数据并行语言,它松散的基于 ML (Meta-Language) 语言 (Robin Milner et al. 1990), 完全支持嵌套的串行和嵌套的并行机制,也就是说它能够为一个并行函数产生多个实例并行运行。嵌套的并行对实现不规范的嵌套循环(内层的循环次数依赖于外层)算法和分治算法很重要。NESL 同时提供了一个性能分析模型能够分析一个程序在不同并行机模型上的运行效率。

NESL 中主要的数据类型就是序列 (sequences), 即一串用方括号扩起来的数据如 $[6, 14, 0, -5]$, 或者是双引号里面的字符串如 “NESL allows sequences of characters”。序列还可以嵌套使用, 如 $[“a” “sequence” “can be made of” “sequences”]$ 。最基本的 NESL 操作就是对括号里面的每一个元素进行统一的操作如 $\{a+1 : a \text{ in } [5, 14, 0, -5]\}$, 这个操作对序列里的每一个元素加一。更复杂一点的如两个序列相加 $\{a+b : a \text{ in } [6, 14, 0, -5]; b \text{ in } [4, -4, 10, 15]\}$, 它会将 a 和 b 中对应的元素相加最后得到的结果为序列 $[10, 10, 10, 10]$ 。下面的例子将展示如何使用函数对数据进行操作:

```
function dotprod(a, b) =
    sum({x*y : x in a; y in b});
dotprod([2, 3, 1], [6, 1, 4]);
```

NESL 作为面向全局视角语言,在对数组数据进行处理的时候有其特有的优势,但其语法过于复杂,不利于推广。

2.3 小结

本章介绍首先当前并行机中常用的两种体系结构硬件平台——SMP 和 Cluster, 接着介绍了一些曾经出现并辉煌一时的体系结构。然后介绍了 MPI 和 OpenMP 这两种并行程序开发模型,对其优缺点进行了分析。并根据目前多核机

器大量出现和多核机群的迅速崛起的现状介绍了混合编程。最后介绍了分布式存储上的三种主要的 PGAS 语言和两种全局视角语言。从目前的趋势来看, 由多核 CPU 组成的机群系统已经成为了并行机的主流, 并且还会持续很长时间。而使用什么样的编程方法和模式来适应这样的系统也必将成为人们所关注的热点。

第3章 MapReduce 的发展和现状

3.1 MapReduce

3.1.1 MapReduce 简介

MapReduce(Jeffrey Dean, Sanjay Ghemawat 2004; Ralf Lämmel, 2007)是一种编程模式,源自对 Lisp 以及其它很多面向功能语言中的 map 和 reduce 概念的抽象,由 Google 公司提出并首先应用到大型集群上,应用于对海量数据集的产生与处理。在 MapReduce 中,用户指定一个 map 函数,通过这个 map 函数处理输入的 key/value (键/值)对,并且产生一系列的中间 key/value 对,然后使用 reduce 函数来合并所有的具有相同 key 值的中间键/值对中的值部分,并产生输出数据。现实生活中的很多任务的实现都是基于这个模式的。

使用 MapReduce 实现的程序可以自动分布到一个由普通机器组成的超大集群上并发执行。运行时系统会解决输入数据的分布细节,跨越机器集群的程序执行和调度,处理机器的失效,并且管理机器之间的通讯请求。这样的模式允许程序员不需要有什么并发处理或者分布式系统的经验,就可以使用超大的分布式系统的资源。

3.1.2 Google 公司的 MapReduce

Google 的应用绝大部分计算都是概念上很简单的,不过输入数据通常是非常巨大的,并且为了能在合理的时间内执行完毕,数据上的计算必须分布到上百个或者上千个计算机上去执行。如何并发计算,如何分布数据,如何处理失败等等相关问题合并在一起就会导致原本简单的计算淹没在为了解决这些问题而引入的很复杂的代码中。

为了降低这种复杂度,Google 设计了一种新的能够方便处理这类简单计算的模式——MapReduce。这些简单计算原本很简单,但是并发处理细节,容错细节,以及数据分布细节,负载均衡等等细节问题导致代码非常复杂。所以可以把这些公共的细节抽象到一个库中,由一个运行时系统来负责。而将对数据的操作抽象为 map 和 reduce 两个概念,这种抽象是源自 Lisp 以及其它很多函数式语言的 map 和 reduce 概念。大部分对数据的操作都和 map 相关,这些 map 负责处理输入记录中的每个逻辑“record”,并且会产生一组中间的 key/value 键值对,接着在所有具有相同 key 的中间结果上执行 reduce 操作,这样就能将适当的数据进行合并。

在 MapReduce 使用用户自定义的 Map 函数和 Reduce 函数对数据进行处理。

MapReduce 的主要贡献在于提供了一个简单强大的接口,通过这个接口,可以把大尺度的计算自动的并发和分布执行。使用这个接口,可以将应用部署到由普通 PC 构成的巨大集群上,从而获得极高的性能。

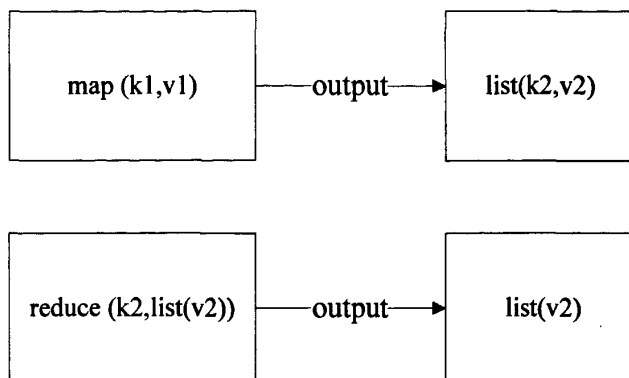


图 3.1 map 和 reduce 函数的相关类型

从概念上讲,使用者提供的 Map 和 Reduce 函数的相关类型如图 3.1 所示:显然,输入的键/值和输出的键/值是属于不同的域的。进一步说,中间的键/值和输出的键/值属于相同的域的,比如 map 的输出就是作为 reduce 的输入。Google 的 MapReduce 系统由 C++语言来实现,把字符串作为用户定义函数的输入和输出,由用户自己来识别字符串并转化为合适的类型。

Google 中广泛使用的计算环境为:用交换机网络(Luiz A. Barroso et al. 2003)连接的由普通兼容 PC 机构成的超大集群。在这种环境下,网络带宽资源是相对缺乏的,因此系统设计者尽量让输入数据保存在构成集群机器的本地硬盘上,通过 GFS (Google File System) 管理 (Sanjay Ghemawat et al. 2004) 的方式来减少网络带宽的开销。图 3.2 是 Google 的 MapReduce 运行示意图。

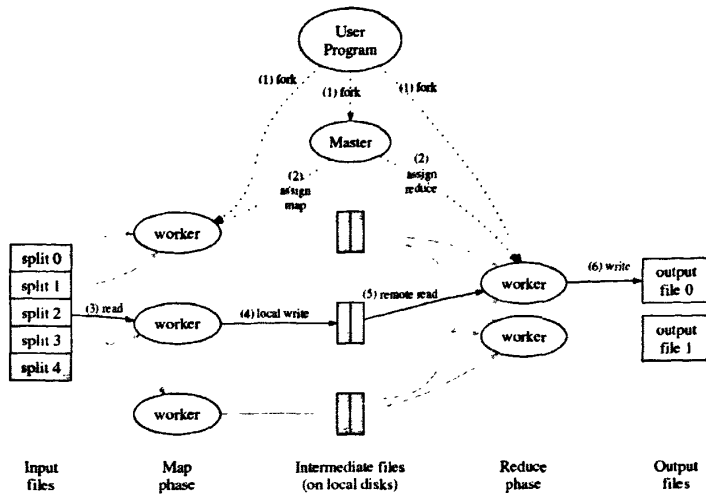


Figure 1: Execution overview

图 3.2 Google MapReduce 运行示意图

Google 公司的 MapReduce 系统运行在一个由普通机器组成的大型集群上，并且有很强的扩展性：一个典型的 MapReduce 计算处理通常分布到上千台机器上来处理上 TB 的数据。已经开发出了上百个 MapReduce 程序，并且每天在 Google 的集群上有上千个 MapReduce 任务在执行。

3.2 MapReduce 其它实现

MapReduce 模型被提出以后，越来越多围绕着它的研究工作已经展开，有些是将其应用在机器学习（C. Chu et al. 2007）等领域，但更多的工作是针对自己的应用或者不同的体系结构来实现自己的 MapReduce 系统，下面将介绍几种已有的 MapReduce 实现。

3.2.1 Hadoop

Hadoop (Hadoop 2008) 源于 apache 的 Lucene 项目。Lucene 是一个用 Java 开发的开源高性能全文检索工具包，它不是一个完整的应用程序，而是一套简单易用的 API。在全世界范围内，已有无数的软件系统，Web 网站基于 Lucene 实现了全文检索功能，后来 Doug Cutting 又开创了第一个开源的 Web 搜索引擎 Nutch，它在 Lucene 的基础上增加了网络爬虫和一些和 Web 相关的功能，一些解析各类文档格式的插件等，此外，Nutch 中还包含了一个分布式文件系统用于存储数据。从 Nutch 0.8.0 版本之后，Doug Cutting 把 Nutch 中的分布式文件系统以及实现 MapReduce 算法的代码独立出来形成了一个新的开源项 Hadoop。

Nutch 也演化为基于 Lucene 全文检索以及 Hadoop 分布式计算平台的一个开源搜索引擎。Hadoop 由分布式文件系统 HDFS 和 MapReduce 这两个部分组成，是一个开源的可运行于大规模集群上的分布式并行编程框架。

Hadoop 把 Google 的 MapReduce 模式应用在了具有扩展性的领域，例如 web 搜索，它提供了一个支持 MapReduce 并行框架的部件，用一个自己的分布式文件系统 HDFS 代替了 Google File System (Sanjay Ghemawat et al. 2003)，并且还有一个分布式的数据库 HBase，它基于 Google 的一个叫做 Big Table 的数据库 (Fay Chang et al. 2006)。

在 Hadoop 中所有数据都被存储在分布式文件系统 HDFS 上，而 HDFS 由一个管理结点 (NameNode) 和 N 个数据结点 (DataNode) 组成，每个结点均为一台普通计算机。HDFS 在使用上与单机上的文件系统非常类似，一样可以建目录，创建、复制和删除文件，查看文件内容等。但其底层实现却是把文件切割成 Block，然后这些 Block 分散地存储在不同的 DataNode 上，每个 Block 还可以复制数份存储于不同的 DataNode 上，达到容错容灾的目的。NameNode 是整个 HDFS 的核心，它通过维护一些数据结构，记录了每一个文件被切割成了多少个 Block，这些 Block 可以从哪些 DataNode 中获得，各个 DataNode 的状态等重要信息。

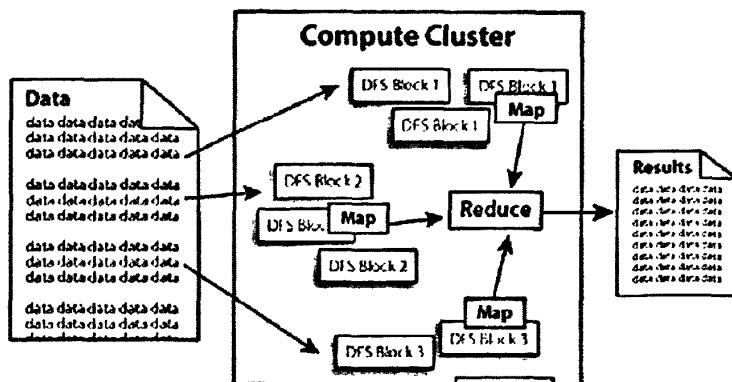


图 3.3 Hadoop 中分布存储与并行计算

Hadoop 中的分布式并行计算：Hadoop 中有一个作为主控的 JobTracker，用于调度和管理其它的 TaskTracker。JobTracker 可以运行于集群中任一计算机上。TaskTracker 负责执行任务，必须运行于 DataNode 上，即 DataNode 既是数据存储结点，也是计算结点。JobTracker 将 Map 任务和 Reduce 任务分发给空闲

的 TaskTracker, 让这些任务并行运行, 并负责监控任务的运行情况。如果某一个 TaskTracker 出现故障, JobTracker 会将其负责的任务转交给另一个空闲的 TaskTracker 重新运行。图 3.3 为 Hadoop 中分布式存储与并行计算之间的关系图。

Hadoop 中的本地计算: 数据存储在哪一台计算机上, 就由这台计算机进行这部分数据的计算, 这样可以减少数据在网络上的传输, 降低对网络带宽的需求。在 Hadoop 这样的基于集群的分布式并行系统中, 计算结点可以很方便地扩充, 因而它所能提供的计算能力近乎是无限的, 但由于数据需要在不同的计算机之间传输, 所以网络带宽往往容易变成瓶颈, “本地计算”成为了最有效的一种节约网络带宽的手段, 业界把这形容为“移动计算比移动数据更经济”。

3.2.2 GPU 上的 MapReduce

Mars (B. He et al. 2008) 是一个设计并实现在图形处理器 (GPU) 上的 MapReduce 系统。与通用的 CPU 相比, GPU 拥有更高的计算能力和更宽的带宽, 但是由于它的体系结构是按照协处理器 (co-processor) 来设计的, 并且它的程序设计接口是为图形处理而专门设计的, 因此对 GPU 编程还是比较复杂的。作为第一种尝试使用 GPU 为硬件而实现 MapReduce 系统, Mars 所使用的硬件是 NVIDIA G80 GPU, 这个 GPU 包含一百个图形处理单元。

由于 GPU 的计算能力和可编程性越来越高, 很多众核处理器将 GPU 作为一个协作处理器来完成计算任务 (A. Ailamaki et al. 2006), 由 GPU 组成的众核处理器基本结构如图 3.4 所示。GPU 由很多 SIMD 的多核处理单元组成, 并且支持上千个线程的并发执行。与 CPU 的线程相比, GPU 中线程的上下文切换开销要小的多。每个多核处理单元上的所有线程被划分成一个组进行管理, 该组中的所有线程共享这个多核处理单元上的诸如寄存器等资源。

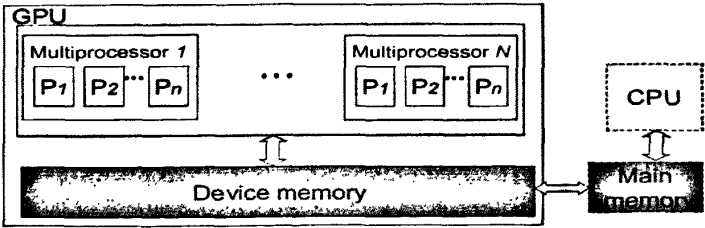


图 3.4 基于 GPU 的众核 (many-core) 结构

Mars 系统实现在一台普通的 PC 机上, 这台 PC 有一个 NVIDIA GeForce8800 GPU (G80) 和一个 Intel quad-core CPU。相对于 CPU 上的 MapReduce 系统, Mars 的 API 很少。Mars 的运行系统会利用众多的 GPU 线程来完成 Map 和 Reduce 的

工作，它会自动为每个线程分配一小部分 key/value 对，这样就能很好的并发利用 GPU 的大量线程。为了避免同时写带来的冲突，Mars 的运行时系统使用了一种开销很低的 lock-free 机制，这个机制会保证拥有少量同步开销的并行操作的正确行。另外，由于在网页数据分析中存在着大量的文本操作，因此 Mars 系统的开发者还开发了一个有效的字符串处理库。

Mars 系统有两个基本设计原则：

(1) 简单原则。简化 GPU 编程会使越来越多的开发者使用 GPU 来实现他们的程序。

(2) 高效原则。在 GPU 上实现的 MapReduce 系统的性能至少应该不输于 CPU 上的 MapReduce 系统。

Mars 所提供的 API 与其它已存在的 MapReduce 差不多，但是数量略少。这些由 C/C++ 来实现的 API 分为两类，一类是用户实现的 API，它们由用户来实现；另一类是系统提供的 API，它们像链接库一样被用户调用。

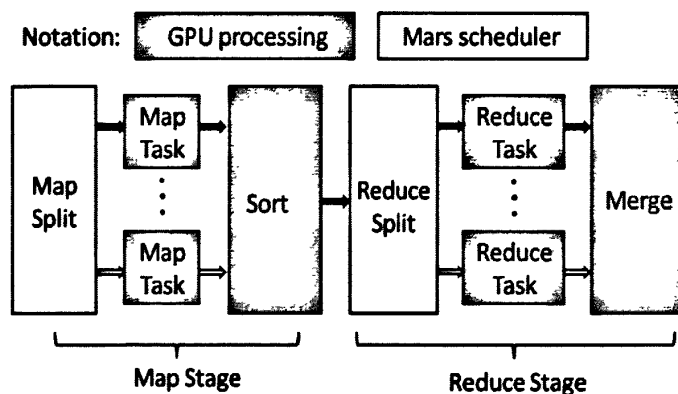


图 3.5 Mars 的工作流程图

图 3.5 给出了 Mars 上的 MapReduce 流程。在 map 阶段，划分器把输入的 key/value 对划分成与线程数目一样多的数据块，每个 GPU 线程处理一个数据块。这样使得所有线程在 map 阶段可以达到负载均衡。map 阶段完成以后，产生的中间结果 key/value 会被重新组织，具有相同 key 的 value 会被有序存储。在 reduce 阶段，划分器会把存储的中间结果 key/value 对划分成更小的数据块，具有相同 key 的 key/value 对会被分在同一个块里面，同样数据块的数目与线程的数目是相同的。每个进程有一个很长的 ID，它与每个数据块拥有的一个很长的 key 所对应。reduce 阶段通常也是负载均衡的，除非 value 列表的长度不同。而由于当前的 GPU 并不支持动态线程调度，因此 Mars 也不支持动态的负载均衡。在最后一

步，所有进程产生的输出都会被存入同一个缓冲区。

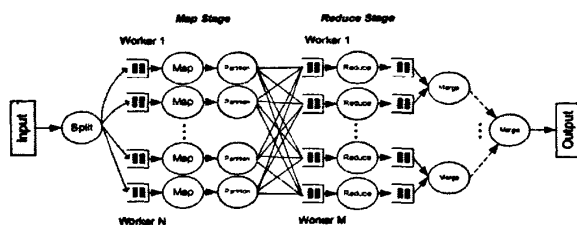
3.2.3 Phoenix

Phoenix (C. Ranger et al. 2007) 是共享内存系统上的一个 MapReduce 实现，它同样包括一组编程接口 API 和一个运行时系统。Phoenix 的运行时系统会自动的管理线程创建，动态任务调度，数据划分和容错处理。

Phoenix 采用线程来实现 Map 和 Reduce 任务，同时使用共享内存缓冲区的方法来实现通信，这样就避免了数据拷贝产生的开销。运行时系统动态地在各个可用核上调度任务从而达到负载均衡，通过并行粒度调整和并行任务分配来保证数据的本地性。运行时系统还通过任务重做或者重新分配来处理传输或者参数错误，然后把输出与其它输出进行合并。

Phoenix 的 API 由 C 和 C++ 编写，而这些 API 也可以用 Java 或 C# 这样的语言来定义。它一共可以分为两类：第一类可以被程序用来使用完成初始化系统和产生输出；第二类包括一系列用户定义的函数。

Phoenix 运行时的任务调度器可由用户配置，它将创建并控制线程完成所有的 Map 和 Reduce 任务，同时管理用于通信的缓冲区。在初始化完成以后，调度器决定有多少个核要参与计算，并为每个核产生一个线程用来动态的执行 Map 和 Reduce 任务。Map 阶段分割器 (Splitter) 先把输入数据分成大小相同的块，然后交由 map 任务线程去操作，Partition 函数会把中间结果按 key 进行统计。调度器必须等所有的 map 工作完成以后才初始化 Reduce 阶段，动态分配 redcue 任务并确保该阶段负载均衡。最后一步就是将所有的最终输出结果合并并放入连续的内存缓冲区。图 3.6 为 Phonix 运行时系统的基本数据流图。



型来解决消息传递过程中的异构性问题以及数据不连续问题。MPI 的消息数据类型可以分为两种：预定义数据类型和派生数据类型。虽然可以通过打包并以 MPI_PACKED 为数据类型来发送地址空间不连续的数据项，但是这种方式比较繁琐，而且容易出错。MPI 引入派生数据类型的概念，允许定义由数据类型不同且地址空间不连续的数据项组成的消息。然而这些操作在面对发送接收的消息为对象时变得很苍白无力。而 Boost.MPI 在泛型消息传递方面的优势正是它被选择用来构建 HPMR 的原因。

Boost.MPI 是一个高性能并行计算的消息传递库。一个 Boost.MPI 程序由一个或多个可以通过点对点或集群消息传递进行通信的进程组成。不像线程环境中的通信和共享内存的库，Boost.MPI 的进程可以分布在很多不同的机器上，这些机器可能拥有不同的操作系统和底层系统结构。

Boost.MPI 不是一个全新的并行编程库，而是一个 C++ 友好的面向标准 MPI (standard Message Passing Interface) 的程序接口。MPI 是高性能和分布式计算领域最流行的库，它定义了面向 C, Fortran 和 C++ 库接口，并且已经存在了大量的以这三种语言实现的 MPI 程序。虽然现在已经有了很多 C++ 编写的 MPI 程序，但是相比使用 C 编写的程序它们并没有什么新的功能和不同。而 Boost.MPI 提供了另一种 C++ 的 MPI 接口，这种接口可以很好的支持 C++ 风格的开发风格，其中包括完全支持用户自定义的数据类型和 C++ 的标准库中的各种类型，集体算法中的任意函数对象，还可以使用先进的 C++ 库技术来保证最大的效率。

到现在，Boost.MPI 已经能够支持 MPI 1.1 中的大部分功能了。Boost.MPI 的抽象性不高使得程序员能混合使用底层的 C MPI 库和 BoostMPI 库。Boost.MPI 现在支持的特性有：

通信域：Boost.MPI 支持通过手动操作进程组来创建、销毁、克隆和分拆 MPI 的通信域。

点对点通信：Boost.MPI 支持通过阻塞和非阻塞的发送接收操作来实现的点到点的操作，而且支持原始数据类型和用户自定义的数据类型。这个自定义的类型无需打包之类的操作。

集群通信：Boost.MPI 支持使用原始数据类型和用户自定义数据类型的归约 (reduce) 和收集 (gather) 等集群通信。

MPI 数据类型：Boost.MPI 能够通过 Boost 序列化库 (Boost.Serialization) 建立用户自定义的 MPI 数据类型。

将结构和内容分开：在传输复杂数据结构 (list、map 等) 的时候，Boost.MPI 可以先将它们的结构传送过去然后在将它们的内容传送过去。如果一个数据结构是一个非常大的且静态的数据结构，那么在普通的 MPI 中将会经过多次的传输

操作才能完成，而在 Boost.MPI 中则相当简单。

对 Boost.MPI 的使用可以通过内建的 C++ 绑定也可以通过其它的方式，它在对 C++ 方面的得强大支持是它被选择用来实现 HMPR 系统的一个重要原因。下面将介绍 HPMR 的设计与实现。

3.3.2 HPMR 的设计与实现

到目前为止已经出现了很多关于 MapReduce 模式应用的研究，但是关于在多核集群上适于数值计算的 MapReduce 系统的研究还不多，在总结了已有 MapReduce 系统特别是 Hadoop 的优缺点以后，我们设计并实现了自己的 High-Performance MapReduce。

HPMR (郑启龙 et al. 2008) 的结构可以分为客户端和服务端，客户端是 MapReduce 程序集成开发环境，服务器端是 MapReduce 程序运行环境。HPMR 的服务器端包含编译器，运行器和 MapReduce 库。MapReduce 库是 HPMR 的核心，它在程序运行时扮演两种角色：master 和 worker，两者的关系图 3.7 所示。master 负责管理所有的 worker，worker 负责执行子任务 task，包括 Mapre 任务、Reduce 任务和通信任务。

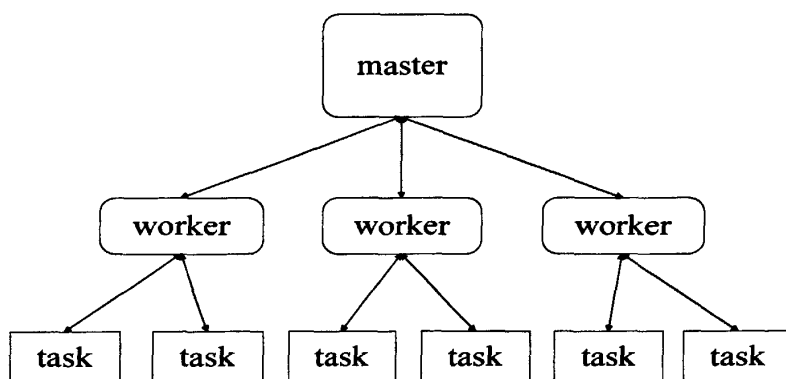


图 3.7 master 和 worker 的关系

HPMR 的 MapReduce 库由三个模块组成，如图 3.8 所示。数据管理模块的职责：(1) master 从输入文件中读入数据块，把它们分发给各个 worker；(2) worker 管理 map 数据和 reduce 数据。任务管理模块的职责主要是控制任务并行，提高程序执行效率。通信管理模块的职责主要是根据 map 阶段产生的 key/value 信息产生 kv 路由，并将其发送给各个 worker，而后个 worker 之间进行消息传递和通信。

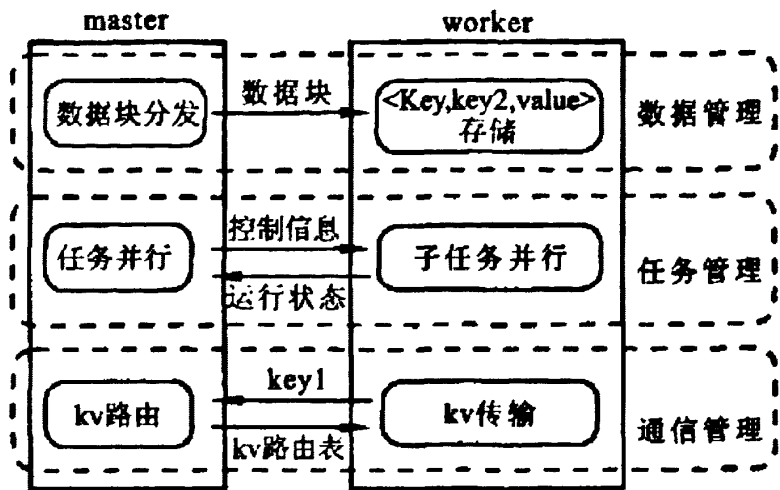


图 3.8 HPMR 的三个模块

HPMR 综合利用多进程和多线程技术在多核集群上构造了一个分布式并行系统。通信管理模块的职责包括 kv 路由和 kv 传输。kv 路由是指，在 Map 阶段后，所有 worker 把自己的<key1,key2,value>信息发送给 master，master 调用 kv 路由算法生成 kv 路由表，kv 路由表中记录了每个<key1,key2,value>应该被发送给哪个 worker。Kv 传输是指，worker 根据 kv 路由表把自己的<key1,key2,value>发送给其它 worker。

图 3.9 展示了 HPMR 的执行流程，其中既包括任务流（T1~T3），也包括数据流（D1~D9）。

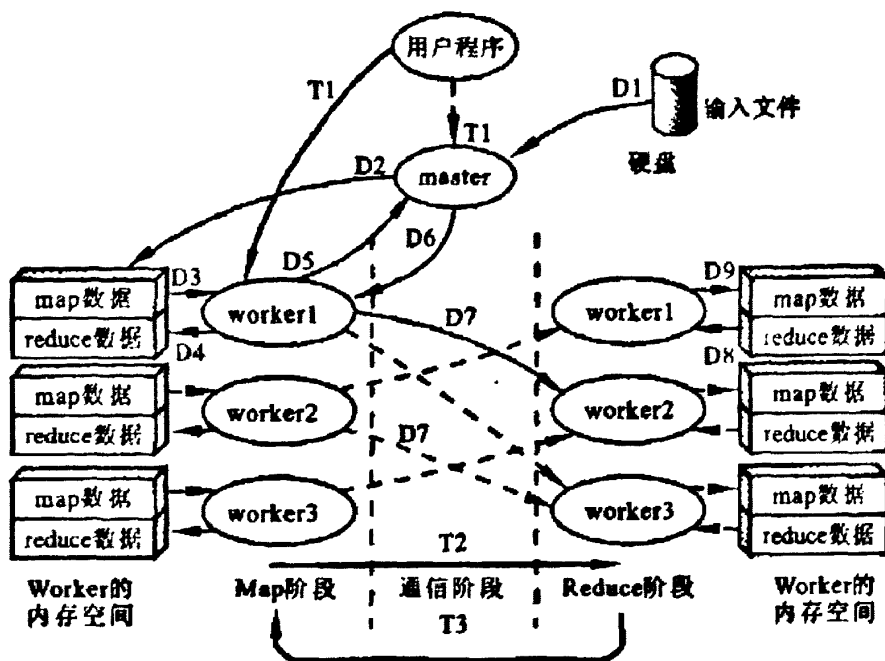


图 3.9 HPMR 执行流程图

任务流：<T1>启动用户程序，生成 master 和 worker。<T2>worker 执行议论 Map-Reduce 过程，分为三个阶段：Map 任务、通信任务、Reduce 任务。<T3>如果程序没有结束，启动新一轮 Map-Reduce 过程。

数据流：<D1>master 从输入文件中读入程序的输入数据。<D2>master 把输入数据分给各个 worker。<D3>map 函数读入输入数据。<D4>map 函数的输出数据将作为 reduce 函数的输入数据。<D5>worker 把<key1,key2,value>信息发送给 master。<D6>master 返回 kv 路由表。<D7>每个 worker 把自己的数据发送给其它 worker。<D8>reduce 函数读入输入数据。<D9>reduce 函数的输出数据作为下一轮 Map-Redcue 过程中 map 函数的输入数据。

HMPR 相对与其它的 MapReduce 系统一下三个主要的特点：（1）使用 <key1,key2,value>代替<key,value>。因为在数值计算中通常会用到来自不同数据域的两组或几组数据，比如在矩阵乘法中，来自矩阵 A 和来自矩阵 B 的数据在相乘的时候发挥的作用是不同的，因此就需要把它们区分出来。在 HPMR 中 key1 通常是用来表示相关性的，而 key2 则是用来区分来自不同数据域的数据的。在矩阵向域中可以另 key2 为 A 或 B 来区分不同的矩阵。当然在其它的应用中 key2 也可以用来表示数据的其它属性。使用 key2 可以方便的表达与数据某些属性相关的信息。（2）禁止 Map 过程与 Reduce 过程并发。由于在并行的数值计算中，

通常先将原始数据划分成各个子数据块然后分别由不同的进程或线程对其进行计算操作。而在多轮的迭代计算中,不但各数据子块间需要进行数据交换,而且每轮之间经常会有明显的依赖关系,所以每轮计算完成以后,每个数据子块都要进行数据交换,只有当数据交换的过程完成以后才能进行下一步的计算。由此可以看出,数据交换需要一个同步过程,而在 HPMR 中 Map 过程完成以后,紧接着的数据通信过程就是数据交换过程,所以要进行同步,因此 Map 过程与 Reduce 过程不能并发进行。(3)支持多轮 Map-Reduce 过程。相对 wordcount(Jeffrey Dean, Sanjay Ghemawat 2004)这种一轮 Map-Reduce 过程就能的出结果的算法,在数值计算中,多数应用都是要经过多轮的循环迭代才能完成。所以高效的支持多轮 Map-Reduce 也就成了 HPMR 必然要考虑的。为了让程序员能够控制程序的运行,HPMR 提供了控制循环结束的方法,程序员可以设置循环迭代的轮数、循环结束的条件。而在默认情况下 HPMR 会判断每个 worker 是否还有工作要做,如果都处于无事可做的状态则结束本程序。

3.4 总结

本章介绍了 MapReduce 编程模式起源和发展,由 Google 所实现的 MapReduce 以及其特点,引出其它三种 Hadoop、Mars 和 Phoenix。它们属于在不同类型机器上的 MapReduce 实现。当然 MapReduce 还有其它形式的实现如 Cell 上的实现(M. Kruijf and K. Sankaralingam 2007)等。通过对这些系统的介绍阐述了 MapReduce 模式的基本运行模式和编程方式,在分析了这些 MapReduce 实现的优缺点之后,引入了我们面向高性能计算领域而在多核集群上设计的 MapReduce 平台——HPMR 并介绍了它适应多轮迭代的数值计算的特点。

第 4 章 并行算法模式与 MapReduce 描述

4.1 并行算法设计模式与 MapReduce

4.1.1 并行算法设计模式

设计一个并行算法的第一阶段由分析问题以识别开发的并行性组成，通常通过利用“寻找并发性”设计空间中的模式实现（Timothy G. Mattson 2005）。“寻找并发性”设计空间的输出是问题的分解，即将问题分解为多个设计元素：

- 任务分解，识别出多个能够并发执行的任务；
- 数据分解，识别出从属于每个任务的局部数据；
- 分组任务和排序分组，以满足时间约束；
- 任务间的相关性分析。

在定义算法结构的无数方式中，大多数遵循六种基本的设计模式。这些模式组成了“算法结构”设计空间。“算法结构”设计空间的总体框图和它在语言中的位置如图 4.1 所示：

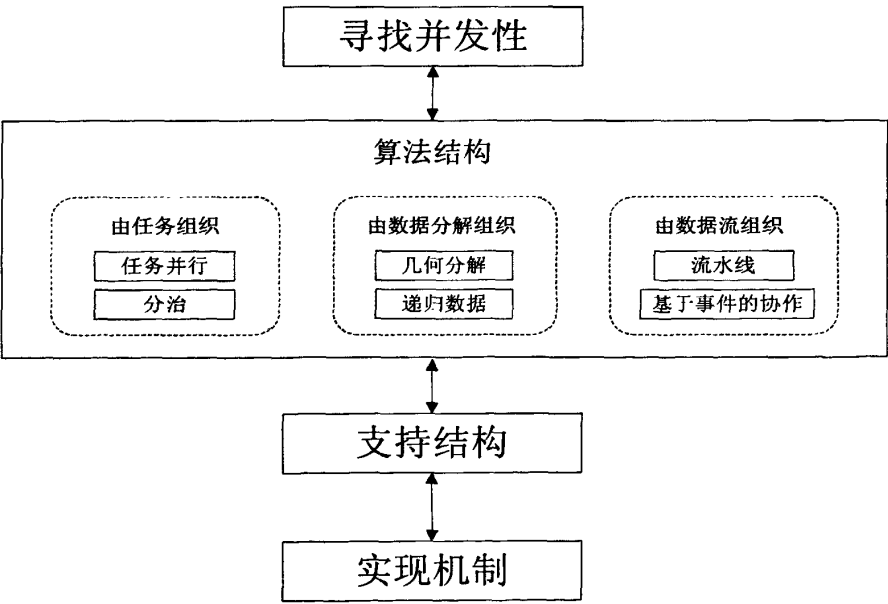


图 4.1 “算法结构”空间的总体框图和它在模式语言中的位置

这个阶段的关键问题是确定哪一个模式或者那些模式最适合这个问题。其中由任务组织、由数据分解组织和由数据流组织都是主要的组织原则，而每种组织

原则下都有两种不同的模式，这些模式所代表的编程求解方式是不一样的。如图 4.2 所示。

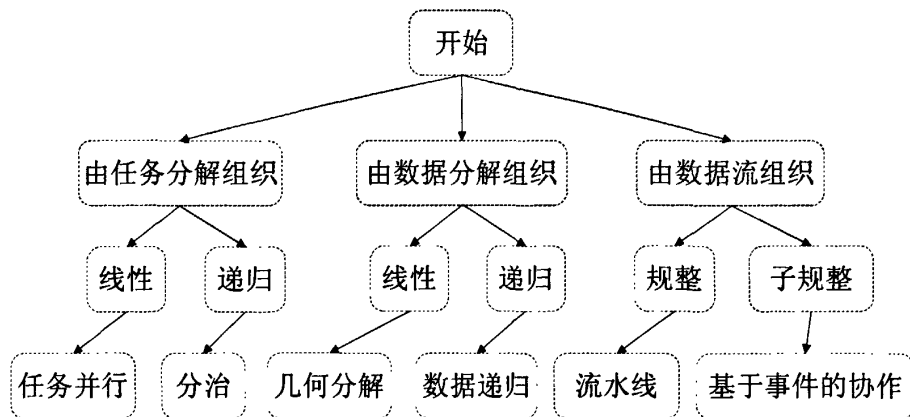


图 4.2 算法结构空间决策树

以上各种编程模式用来分析并行问题以确定怎样简单高效的将问题分解，并不涉及具体的如何编程实现这些模式，而各种模式的实现是依赖于底层的实现机制诸如 OpenMP、MPI、Titanium 等语言。下面将讨论 MapReduce 中这些模式的实现方式。

由任务组织，当任务的执行是最好的组织原则时，选择由任务组织。然后确定任务是如何被枚举的。如果这些任务可以被聚集成为一个线性集合时，选择任务并行模式。这种模式包含两种情形：一种是任务间相互独立的情形（即所谓的易并行算法）；另一种是任务间具有某些相关性的情景，具体表现为任务需要访问共享数据或者需要交换信息。如果任务由一个递归过程枚举，选择分治（divide and conquer）模式。在这种模式中，问题的求解方式是：将问题递归的划分为多个问题子集，然后将子解决方法组合为原始问题的一个解决方法。在这里可以看到，由任务的相互独立情况分为两种不同的模式，因此在 MapReduce 中对其进行实现时也要考虑这两种情况。当任务间相互独立时，每个 map/reduce 的执行者都在自己的 map 或者 reduce 中独立完成自己的任务。如果任务间存在某些联系，正如上述——原始问题的解决方法要由问题子集的解来组合，那么使用 MapReduce 来实现的时候应该将各个不同的子任务分布在 map 中，通过产生统一的 key 将子任务的解决方法汇聚到一起，在 reduce 中由一个执行者将这些解决方法组合为原始问题的一个解决方法。

由数据分解组织，当数据分解是主要的组织原则时，选择由数据分解组织分支。在这一组中存在两种模式，不同点是分解的构造方式——线性方式或递归方

式。当问题空间被划分为多个离散的子空间，并且问题的求解通过计算每个子空间的解决方法实现，而且每一个子空间的解决方法通常需要使用其它几个子空间的数据是，选择几何分解。可以在科学计算中发现这种模式的许多实例，它在基于网格的并行计算中作用明显。当问题根据一个递归数据结构（例如，二叉树）定义时，选择递归数据模式。MapReduce 模式本来就是用于处理大规模数据的，因此在处理数据分解时相对比较简单，几何分解中问题空间被划分成多个离散子空间，每个子空间由一个 map/reduce 的執行者进行处理，如果子空间需要进行数据交换，则当 map 处理完数据后让交换数据与对应的子空间产生相同的 key，在 reduce 中相关数据就会聚合到一起，具体可见后面要提到的 LU 分解和 QR 分解的详细实现过程。对于递归数据模型也是采取相同的思路，只是原始数据和 key 值的产生会有所不同。

由数据流组织，当主要的组织原则是根据数据流对任务分组强制规定了一种顺序的方式时，选择由数据流组织。这种模式由两种成员组成，当顺序是规则的和静态的时，应用其中一种，当它是不规则和/或动态的是，应用另外一种。当任务分组间的数据流是规则的、一路的，并且在算法间不发生变化时（即任务分组可以被安排为一个流水线，数据流流过该流水线），选择流水线模式。流水线模式是一种很特殊的模式，其中的工作进程一个接一个的启动，且除第一个进程外每个工作进程都需要前一个进程发送过来的数据，除最后一个进程外每个工作进程都要把自己处理完的数据发送给下一个进程进行处理。在 MapReduce 中，这种模式的实现并不困难，即在 map 结束的时候将自己处理完的数据与下一个 map/reduce 的執行者产生相同的 key 这样就完成了数据交换的任务。现在的 HPMR 系统不能很好的支持 map 或者 reduce 任务的动态启动和撤销，也就是说在流水线模式下不管要不要处理数据每个 map/reduce 的執行者都处于工作状态，直到所有工作全部完成。后面 QR 分解的实现中将用到这种模式。

不难看出，上述的各种编程模式都能采用 MapRedcue 来对其进行描述，并实现在 MapReduce 系统上。因此可以说 MapReduce 能很好的描述并行编程中的问题，是一种良好的编程模型。

4.1.2 MapReduce 对并行算法模式的描述

这一节主要描述 MapReduce 在各种并行模式中的应用，以及用其对特定应用进行分析和描述的步骤。

任务并行模式中，问题可以很好的被分解为能够并发执行的任务集合，每一个并行算法由一个并发任务集合作为基础。关注于这些任务和它们的交互可能不是组织算法的最好方式，在某些情形中，根据数据（如在几何分解模式中所做的）或者根据并发任务见的的数据流（如在流水线模式中所做的）是可行的。但是，在

多数情形中，最好直接利用任务本身来组织，当设计是直接基于任务是，算法被称为任务并行算法。典型的任务并行问题有：医学成像的射线跟踪问题，每一条“射线”相关的计算成为一个单独的并且完全独立的任务；分子动力学问题，每一个原子上的非化学键作用力的更新是一个任务，通常通过将作用力数组复制到每一个子任务中，用于存储每一个原子的作用力部分和来处理任务间的相关性，当所有的任务计算完它们对非化学键作用力的贡献后，每一个作用力数组被组合为一个存储每一个源自的非化学键作用力总和的数组。

上述两种应用与采用按数据划分并不能更好的体现按任务划分的特点，这里我们假设一种应用，它的输入是 n 组数据，分别对每组数据进行的操作并不一样，比如第一组要求先做乘法后做加法，第二组要求先做加法后做乘法，第三组……，这样每一组数据上的任务都不相同，让它们并行来做可以看作是典型的按任务划分。其并行结构如图 4.3 所示：

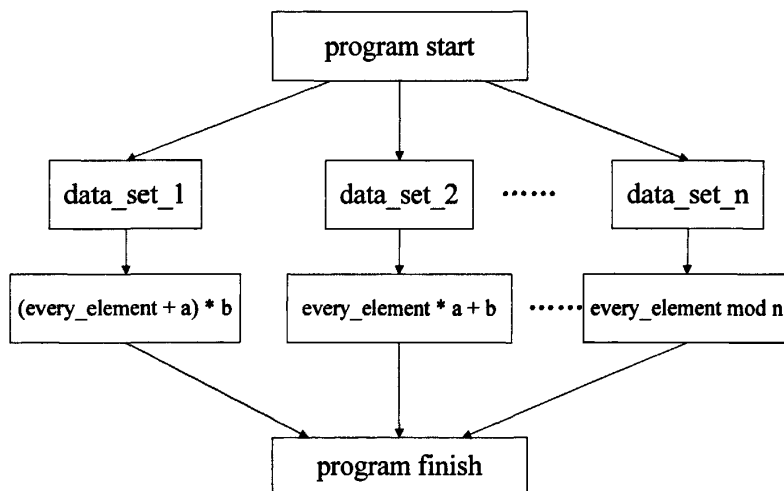


图 4.3 按任务划分并行结构图

现在我们用 MapReduce 模式来对这个问题进行描述，可以看到在这个问题中计算步只有一个，且不存在数据依赖。首先从数据划分方面来看，每一个数据集合对应着不同的操作，所以每一个数据集合作为一个单独的 map 输入 key/value 对中的 value。其次，从计算任务的分配角度来看，由于只有一个数据无关的计算步，所以在设计 MapReduce 的两个主要步骤 map 和 reduce 时，可以将计算任务放入其中任何一个步骤。最后，从数据结构的设计方面来看，一方面在 map 的输入数据中要包含数据集合中的数据，另一方面每个数据集合所要做的计算是不一样的，所以要有一个标识能够表示次数据集合要做什么计算，比如用 1 作为第一组数据的标识，当发现标识为 1 则对每个元素做先加后乘的运算，而且还要

有加法参数 a 和乘法参数 b 的信息。由此看来在设计数据结构的时候由两部分信息要考虑, (1) 数据集合的参数和标识信息, 即数据的特征信息; (2) 数据集合本身的数据, 即原始数据。明确了这几方面, 我们来看 MapReduce 的伪码。

```

Map (key, value)
{
    if (value.operation == 1)                //判断操作标识
    {
        for each element in value.data       //对数据集中每个元素做计算
            element += value.para_a;
            element *= value.para_b;
        }
    else if (value.operation == 2)
    {
        for each element in value.data
            element *= value.para_a;
            element += value.para_b;
        }
    .....
    else if (value.operation == n)
    {
        for each element in value.data
            element = element mod n;
        }
    output(key, value);                      //产生中间结果 key/value 对
}

```

MapReduce 系统已经将以某种格式输入的数据读入并产生 Map 的输入, 由于中间结果没有数据依赖, 所以不产生数据交换, 还是用输入的 key 作为中间结果的 key 值。此时, 所有的计算任务已经完成, 而 Reduce 只需将结果输出:

```

Reduce(key,value_list)
{
    output(key, value_list.value_1);
}

```

Reduce 的输入应该是 key/value_list 的形式, value_list 中包含多个 value, 它们是中间结果中 key 值相同的 value 的一个列表, 比如现在又 key1/value1、

key1/value2、key1/value3，则某个 Reduce 的输入为 key1/value_list，其中 value_list 中有三个元素分别是 value1、value2 和 value3。对于这种只有一个计算步且无数数据交换的应用我们倾向于将计算过程放在 Map 中执行。

分治模式在很多串行算法中得到了应用。利用这种策略，问题被划分为大量较小的子问题，独立的求解每一个子问题，并将所有的子解决方法合并为整个问题的一个解决方法，从而求解整个问题。子问题可以直接被求解，或者可以再使用相同的分治策略求解它们，这样导致一个整体的递归程序结构。对于大量的计算稠密问题来说，这个策略是非常有价值的。对于很多问题来说，它们的数学描述可以很好的被映射为一个分治算法，例如著名的快速傅立叶变换（William H. Press et al. 1993）和 Cholesky 分解（Philip Alpatov et al. 1997）等。因为子问题是被独立求解的，它们的解决方法可以被并行的执行，这就是分治模式并行性所在，如图 4.4 所示：

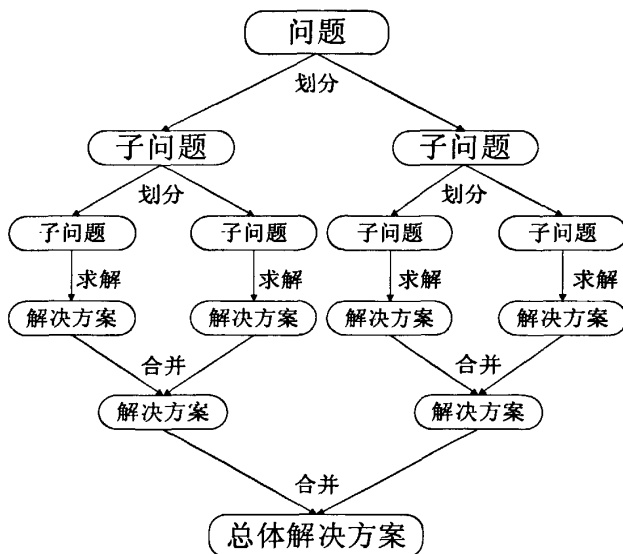


图 4.4 分治策略

下面我们会单独介绍递归在 MapReduce 模式下的实现方法，因此这里我们所描述的是一个简单的例子——计算 π 值。在这个问题中我们只将其向下分解到第一层子问题，这样子问题的解合并就会得到总问题的解。其串行算法核心代码如下：

```

#define N 100000
double pi = 0.0, w, temp = 0.0;
w = 1.0 / N;
for( i = 0; i < N; i++)

```

```

{
    temp = (i + 0.5) * w;
    pi = 4.0 / (1.0 + temp * temp) + pi;
}
pi = pi * w;

```

为了得到更高精度的 π 值，就必须增加循环次数 N ，因此由规模为 N 的问题分解得到 p 个规模为 $m = N/p$ 的问题，为了不在 N 次循环中重复对某个 i 之计算，每个子问题有一个递增步 $step = p$ 。每个子问题分别求解最后合并解。现在我们从 MapReduce 模式出发对这个问题进行描述。首先还是从数据划分方面来看，在这个问题中没有什么数据，对每个子问题来说有三个关键的数据循环次数 m ，乘法参数 w ($w = 1.0 / N$) 和递增步 $step$ ，当然还要有一个对局部计算结果进行存储的变量 pi ，因此在 Map 的输入 value 数据结构中会有这四个成员。其次，从计算任务的分配角度来看，计算任务是计算局部 pi 值，但是与上面我们在任务划分中提到的问题不一样，这里需要将所有局部 pi 值合并求出最后结果，因此它们是具有相关性的，并且要把所有 pi 值送给某个 reduce 任务来计算最终结果。所以，只能将计算任务放入 Map 阶段，通过产生相同的 key，将中间结果的所有 value 值放入同一个 value_list 中，在 Reduce 阶段对其处理产生最终 π 值。最后，从数据结构的设计方面来看，参数 m 和 w 可以看作是数据的特征信息，而局部 pi 值可以看作是原始数据。由此我们可以得到如下 Map 伪码：

```

Map(key, value)
{
    double temp = 0.0;
    int i = 0, j = value.step;
    for (; i < value.m; i++, j += value.step)           //计算局部  $\pi$  值
    {
        temp = (j + 0.5) * value.w;
        value.pi = 4.0 / (1.0 + temp * temp) + value.pi;
    }
    Key key = 1;                                       //产生同一 key 值
    output(key, value);
}

```

当然，如果不是用 $step$ 这样跳步的方法，可以在数据结构中加入起始循环 i 值，然后从这个值开始循环 m 次。在产生的中间结果对所有 key 值都为 1，也可以设置为其它值，这里只是为了方便设置此值。所有 value 值经过 MapReduce

系统的处理都会到同一个链表中。Reduce 的过程如下：

```

Reduce(key, value_list)
{
    Value value_final;
    for each value in value_list
        value_final.pi += value.pi;
    value_final.pi = value_final.pi * value.w;
    output(key, value_final);
}

```

Reduce 将所有子问题的解合并成为原问题的解。由此可以清楚的看出在分治模式中，Map 所扮演的角色就是求解各子问题的解，并为它们产生关联，而 Reduce 则是把关联到一起的子问题解合并，求出原问题的解。

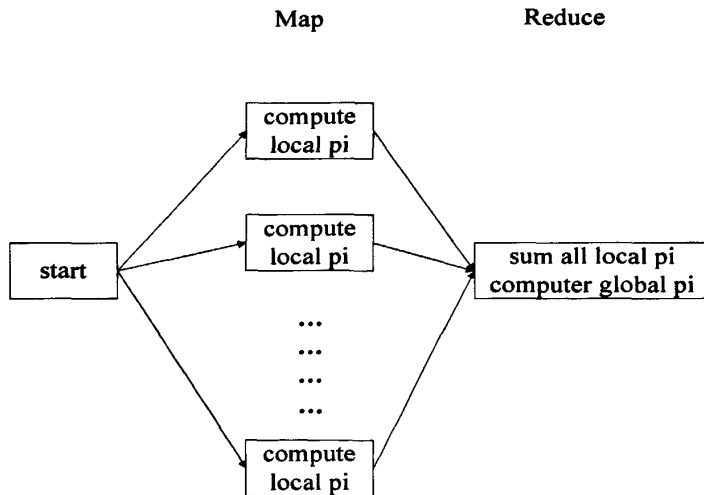


图 4.5 分治策略的 MapReduce 描述

注:通过产生相同的 key 局部 pi 值会被汇聚到同一个 reduce 的输入中

某些具有递归特性的问题，可以很自然的使用分治策略来利用它们的潜在的并发性。对于递归模式我们不做过多的介绍，跟上面一样通过一个具体的实例来进行分析。这里我们以快速排序为例。快速排序的一个特点是通过一个中间值将输入数据集合不断拆分直到每个集合都有序，则整个按顺序合并每个数据集合有序。用 MapReduce 来对其进行分析可以发现，与上面的两个例子相比，快排需要多轮迭代，并不是一轮 map-reduce 过程就能够得出结果。我们仍然先从数据划分方面来着手，由快排的特性可知原始的输入数据只能作为一个整体不能被划

分，这个数据要不断一步一步的被划分成更小的数据作为下一轮 Map 的输入。接着从任务划分方面来看，快排有两个计算任务，一个是根据给定的中间值把数据集划分成两部分，这两部分分别进行快排；另一个就是将所有有序集合按顺序合并，如果是在共享内存系统上实现则可以省略此步骤。不难发现数据划分的工作应该在 Map 中进行，而数据集合并以后就可以直接给出结果，所以要放在 Reduce 中。最后从数据结构的设计来看，首先要按顺序合并的子数据集，所以要有参数 flag 表示这个顺序。对于排序操作来说当数据规模小于一定规模的时候使用一般的排序方法性能就已经很好，因此当数据集的规模小于 N 的时候直接交换或者插入排序即可，所以参数里要多一个数据规模 size 和阈值 N。前面这些都是数据特征，还要使用一个数据结构来待排序存储数据。这里用到了多轮迭代，每次都会产生更多的 key 值，所以对 key 值的设计也很重要，其中有两个原则：（1）唯一性，避免将无关联数据关联在同一 key 值下；（2）特征性，key 值最好能反映数据的某个特征。flag 在这里表示子数据集顺序的，举一个例子来说明 flag 的设置，由于每次划分时产生左右两个子数据集，而中间值会比左子数据集中的所有值都大，因此将其作为右数据集 flag 的值，而左数据集 flag 可赋值为其第一个元素的值，这样足以体现两个数据集的顺序关系，而且其值唯一也可作为 key 值，所以用它做 key 值既体现了唯一性也体现了 key 值的特征性。如图 4.6 所示：

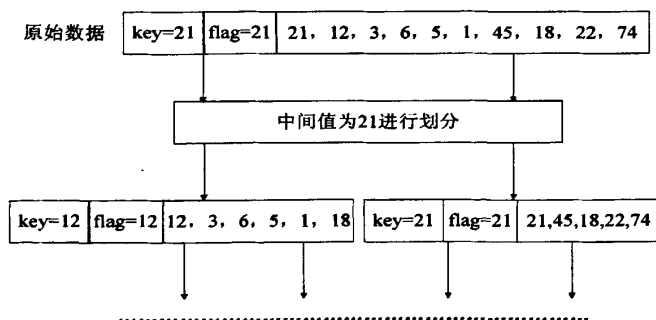


图 4.6 flag 值和 key 值的设置

当子数据集的 size 小于 N 是可以采取平凡排序得到有序的数据集，然后产生一个特殊的 key 值将其合并到最终结果集合中，Map 伪码如下：

```

Map(key,value)
{
    if(value.size <= value.N)           //对数据集排序
    {

```

```

do sort value.data;
Key key = 's';           //设置特殊 key 值
output(key,value);
}
else                      //对数据集合进行划分
{
    mid = value.data[0]
    Value value1, value2;
    partition(mid, value,value1,value2); //划分数据集合设置 size, flag
    Key key1 = value1.flag;
    Key key2 = value2.flag;
    output(key1, value1);
    output(key2, value2);
}
}

```

在 reduce 中，主要的工作就是将 key 值为特殊值（s）的所有 value 合并，而对其它 key 值的 vaule 不做任何改动：

```

Reduce (key, value_list)
{
    if(key == 's')
    {
        Value value;
        merge(value_list, value);
    }
    output(key, value);
}

```

在快排的这个例子中，不能判断迭代的次数，因此就需要 MapReduce 系统来判断程序结束条件，或者给用户提供程序结束手段，HPMR 中就提供了用户可设置的退出条件和退出函数其中同时包括针对固定迭代次数的和不固定迭代次数的两种情况。

流水线模式是一类比较特殊的模式，尽管对单个数据元素的操作具有顺序约束性，但是可以同时对不同的数据元素执行不同的操作，这即是其并发性的所在。一个典型的工业流水是汽车的组装，一辆汽车在流水线中移动过程中每个工人负责一个零件的安装。我们可以把汽车看作是数据，把每个工人看作一个进程，虽

然同一个数据在一个时刻只能由一个进程处理,但是同一时刻可以有多个进程对不同的数据进行处理,而且每个数据都是依次被传递到下一个进程。从 MapReduce 模式出发,在数据划分方面,对处理数据的处理可能是对本地数据的处理,也可能是对流水线上发送过来的数据处理,但初始时只有一个 map-reduce 进程可以运行并进行数据操作,然后它将流水数据发送给下一个 map-reduce 进程,直到最后所有进程完成处理退出。从任务分配角度来看,每个进程完成了对流水数据的处理后才将其发送给下一个,因此要在 map 中完成对数据的处理,而后通过 key 值将其关联到下一个步骤去。从数据结构方面分析,由于不涉及具体数据,所以参数可以不考虑,这里要强调 key 值的设置,初始时每个 Map 都有自己的输入数据,此时将 key 值按流水顺序设置为 0、1、2……不针对具体应用的伪码如下:

```
Map(key, value)
{
    compute(value, value.pipe_data);
    Key key2 = key + 1;
    Value value2(value.pipe_data);
    output(key, value);
    output(key2, value2);
}
Reduce(key, value_list)
{
    replace_pipe_data(value_list, value);
    output(key,value);
}
```

我们将在后面给出矩阵 QR 分解的实例,它是一个流水线模式的程序,因此这里不再过多介绍此种模式。

4.1.3 MapReduce 模式应用总结

从我们用 MapReduce 模式对几种并行模式的描述和对其中具体问题的应用和分析中可以清楚的看出,用此模式来分析一个问题通常都是分为三个步骤:(1)从数据划分来分析。如果有数据,则对数据进行什么样的划分,比如矩阵的按行和按列是完全不同的两种划分方法,划分以后的数据有怎样的数据关联,这都是需要认真考虑的。此外,数据划分以后会有很多特性通称为数据特征信息,要明确子数据集有哪些特征信息,这对数据处理有很重要的作用。如果没有数据,那么应该有哪些信息作为原始数据作为 Map 的输入是很值得考虑的,例如前面求

π 的问题中会将迭代信息和计算参数等信息作为原始输入数据。(2) 从计算任务的分布来分析。因为在 Map 和 Reduce 两个步骤中都可以安排计算任务, 因此根据特定的情况来把计算任务安排在不同的步骤是很重要的。如果各个子数据集之间存在数据依赖, 比如某项数据经过处理后才能发送给后面的子数据集进行处理, 这需要将计算任务放在 Map 中; 或者要先计算各部分的子结果, 然后将它们合并得到最终结果那么计算子结果的任务只能放在 Map 中, 因为这样可以通过让子结果产生相同的 key 而在 Reduce 中关联起来, 为合并工作做好准备。后面我们将要提到的 QR 分解是将计算任务放在 Map 中进行的典型代表。在有些应用中要将计算任务放入 Reduce 中进行, 其特点是所有子数据集要使用某项数据对自身进行操作, 相当于先广播某项数据, 然后对自身数据进行操作。后面将有提到的 LU 分解就是一个典型代表, 这里的 Map 就完全是一个完成数据依赖的过程。(3) 数据结构的设计。有了前面对数据的划分, 要设计合理的数据结构来表达划分得到的数据, 首先这个数据结构要有一个存放数据的容器, 其次这个数据结构包含这个数据所有的特征信息。其结构如图 4.7 所示

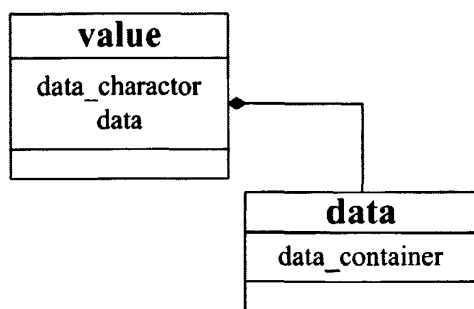


图 4.7 MapReduce 数据结构设计

完成用 MapReduce 模式来分析问题以后还有一个重要的问题就是对 key 值的设计。在 MapReduce 中具有相关性的数据都是通过设置相同的 key 值从而汇聚到一起的。因此, 简单来说 key 值的作用相当于通信标识, 标识相同的 value 最终会被放入同一个 value 链表中。key 值的设计也很重要, 其中有两个原则: (1) 唯一性, 避免将无关联数据关联在同一 key 值下; (2) 特征性, key 值最好能反映数据的某个特征。如在快速排序中提到的 key 值的设计, 而在矩阵计算这种比较特殊的应用中, 子矩阵通常都有自己的编号, 而这个编号是很好的 key 值的候选人。在 MPI 中存在着点到点、广播、散播等多种通信方式, 而 MapReduce 模式中只提供了一个 key 值概念, 因此, 要用 key 值来表答这些不同的通信方式。设计 key 值表达几种简单的通信方式, 如表 4.1 所示。在 HPMR 中, master 要调

用 kv 路由算法生成 kv 路由表, 不同的 key 值越多, 路由算法所耗时间越长。因此, 在产生 key 值的时候不光要考虑它所表达的意义, 同时也要尽量减少不同 key 值的数量。

表 4.1 通信方式与 key 值关系表

通信方式	key 值的表示方式
点对点	$value_1$ 和 $value_2$ 产生关联: 两个 value 分别产生相同的 key, 如 $\langle key, value_1 \rangle$ 和 $\langle key, value_2 \rangle$
广播	$value_1$ 通过各 $value_i$ ($i=1 \cdots n$) 产生关联: 多个 $value_i$ 产生各不相同的 key_i , 如 $\langle key_i, value_1 \rangle$, $value_1$ 则以每个 key_i 为 key 产生如 $\langle key_i, value_1 \rangle$ ($i=1 \cdots n$)
多播	与广播类似, 只是 i 为某些特定值, 因此只要 $value_1$ 与特定的 $value_i$ 产生相同的 key 即可

4.2 数值计算的特点

数值计算 (Numerical Computation) 是指基于代数关系运算的一类, 诸如矩阵运算、多项式求值、解线性方程组等计算问题。基本上属于数值分析 (对以数字形式表示的问题求数值解) 的范畴。它被广泛应用于物理、化学等学科, 产生了诸如计算物理、计算化学等分支学科, 成为推动这些学科发展的一大动力。数值计算与非数值计算有着很大的区别。

诸如 web 搜索服务, 排序, 数据挖掘等非数值计算均属于数据密集型计算, 即, 在大规模的输入数据上进行简单运算。而数值计算的输入数据规模不确定, 但是数据间具有复杂的依赖关系。在并行数值计算中由于输入数据分布在不同的进程或线程中, 因此要通过进程间消息传递或者共享内存等方法进行数据交换, 以满足数据依赖。而这正是其数据分布的特点。

数值计算属于计算密集型计算, 在矩阵运算、解线性方程组等问题程序设计中主要的计算任务集中在循环迭代中。而在并行编程中每次迭代后各进程需要进行必要的数据交换以满足数据依赖。因此, 并行数值计算程序设计的特点可以归结为: 带有数据交换的循环迭代。

4.3 矩阵计算的特点

MapReduce 已被用来服务于 Google 的产品, 但几乎不涉及数值计算, 而在并行计算中, 数值计算是很重要的一大类问题, 矩阵计算则是数值计算中最重要

的一类运算。特别是在线性代数和数值分析中，它是一种基本的运算，因此我们选取很有代表性的矩阵计算为突破口，尝试采用 MapReduce 编程模式在 HPMR 上解决数值计算问题。

4.3.1 数据划分

矩阵并行计算前需对矩阵进行划分，即，把一个大矩阵划分成若干子矩阵，每个子矩阵分配给不同的进程或线程做处理。矩阵划分中最主要的两种方法就是带状划分和棋盘划分。图 4.8 和图 4.9 为典型的带状矩阵划分方法，分为按行划分和按列划分两种情况，并且还能细分为顺序划分和交叉划分（陈国良 2002）。

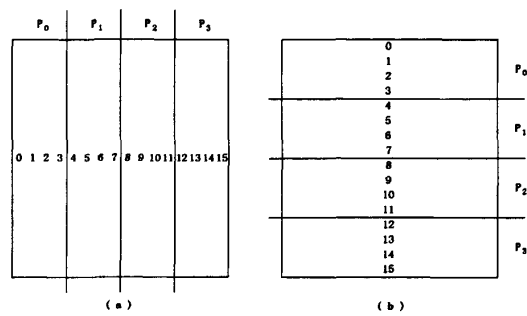


图 4.8 矩阵的顺序带状划分

注：图（a）为按列的顺序带状划分，图（b）为按行的顺序带状划分

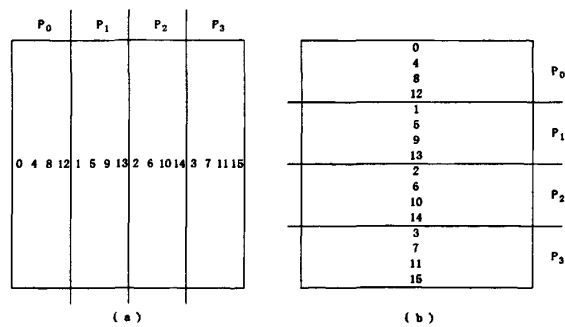


图 4.9 矩阵的交叉带状划分

注：图（a）为按列的交叉带状划分，图（b）为按行的交叉带状划分

4.3.2 主要类的设计

HPMR 目前提供的矩阵划分方式包括了按行/列连续划分或按行/列交错划

分，以及行-列交叉的棋盘式划分，并支持包括稠密和三角等矩阵存储形式。在 HPMR 中，每个矩阵子块称为 `sub_matrix` 并用一个类进行描述。如图 4.10 所示 `sub_matrix` 类包含两部分内容：子矩阵描述和子矩阵数据。子矩阵描述包含划分相关的信息，如划分方式、总块数、本块块号、子块起始/终止行（或列）号等，以及和矩阵计算特征有关的信息。而子矩阵数据所含有的信息则相对比较简单，除了分块矩阵的数据之外通常还有行数和列数，以便对该矩阵数据访问时提供循环控制信息。

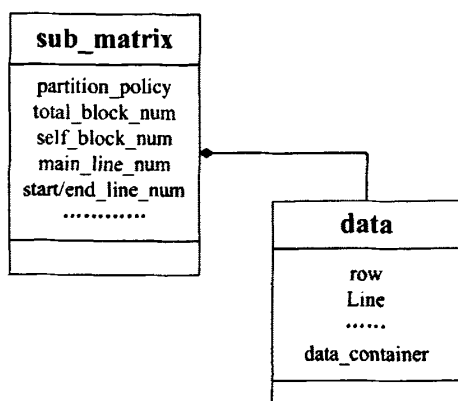


图 4.10 `sub_matrix` 类

4.4 总结

本章首先介绍了各种常见的并行开发模式，然后使用 MapReduce 模式对这几类模式下的应用进行分析，并给出了 MapReduce 的表达式，然后总结了使用这种模式分析和解决问题的方法，最后对数值计算的特点进行了介绍，并注重分析了矩阵运算的特点，为下一章的两个实现打好基础。

第5章 两种矩阵分解的分析与实现

5.1 LU 分解的实现

5.1.1 LU 分解简介

矩阵的 LU 分解是基本、常用的一种矩阵运算，它是求解线性方程组的基础，尤其在解多个同系数阵的线性方程组的时候特别有用。LU 分解主要是把一个给定的 n 阶方阵 A 分解成一个下三角阵。分解的过程中，主要的计算是利用主行 k (k 从 1 到 n) 依次对其余各行 i ($i > k$) 做初等变换，各行计算之间没有数据相关性，因此可以对矩阵 A 按行划分来实现并行计算 (陈国良 et al. 2004)。LU 分解主要串行代码图 5.1 所示。

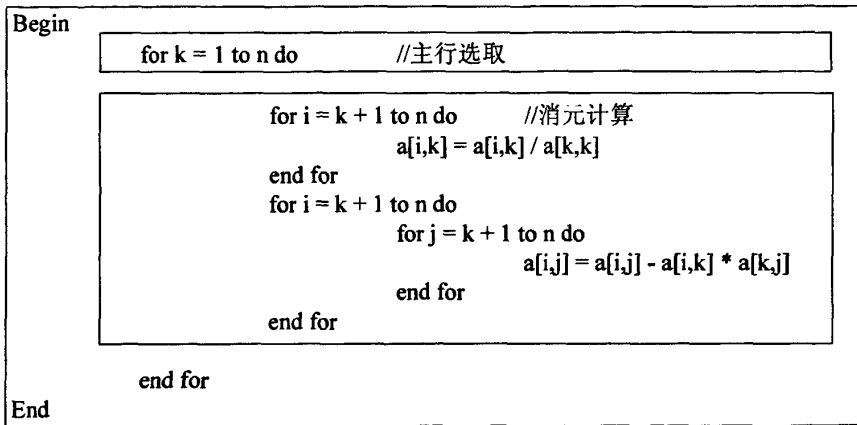


图 5.1 串行 LU 分解代码

图 5.4 中第一部分是最外层循环，为选取主行。第二部分中，单循环是为各行(行号 i 大于主行号 k 的行)计算消元参数；紧接着的两层循环为每一行进行消元计算。存在数据依赖的数据为要进行消元计算的行和当前主行。

5.1.2 使用 MPI 实现的 LU 分解

我们对矩阵 A 的采用按行顺序分块法来实现该算法，这样具有数据相关性的数据单元就变成了当前主行和含有需要变换行的矩阵块。设处理器个数为 p ，矩阵 A 的阶数为 n ， $m = \lceil n/p \rceil$ ， A 对矩阵按行顺序划分后，编号为 i ($i=0,1,\dots,p-2$) 的处理器存有 A 的第 $m*i$ ， $m*i+1$ ， \dots ， $m*i+m-1$ 行，而最做一个处理器 (编号 $i=p-1$) 存有 A 的第 $m*i$ ， \dots ， n 行。然后依次以第 0, 1, \dots ， $n-1$ 行作为逐行，拥有者将其广播给所有处理器，各处理器利用主行对其行向量进行变换。

若以编号为 `my_rank` 的处理器的主行元素作为主行，并将它广播给所有的处理器，则编号大于 `my_rank` 的处理器利用主行元素对其子矩阵的各向量数据做行变换，而 `my_rank` 处理器则对其向量中行号大于 `i` 的数据做行变换。具体并行算法框架描述如图 5.2 所示：

```

对所有处理器my_rank(my_rank=0,...,p-1)同时执行如下算法：
for i=0 to n-1 do /* 当前的主行号 */
    if(i>=my_start_line&& i<=my_end_line)
        /* 当前处理的主行在本处理器 */
        v=i-my_start_line      /* v为主行号在当前处理器中的编号 */
        for k=0 to n-1 do
            f[k]=a[v,k]      /* 逐行元素存到数组f */
        end for
        向其他所有处理器广播主行元素
    else /* 当前处理器的主行不在本处理器 */

        接收主行所在处理器广播来的主行元素并存到数组f

    end if

    if(i<my_end_line) /* 对行元素进行变换 */
        k=my_start_line
        if(i>=my_start_line) /* 拥有主行的进程只对行号大于主行的向量变换 */
            k=i-my_start_line+1;
            mul=a[k,i]/f[i]
            for j=k to my_end_line do
                for m=i to n-1 do
                    a[j,m]=a[j,m]-f[m]*mul
                end for
            end for
        end if
    end for
end for

```

图 5.2 并行 LU 分解核心代码

其中主行的发送采取的是广播形式，即，那些子矩阵已经完成变换的进程也要做出接收动作，这造成了一定的开销，但是因为一个进程总是要跟很多进程进行通信，如果采用点到点的通信，则性能会出现下降，而且进程间通信的控制也会变得复杂。因此在这种情况下广播通信是一种不错的选择。然而，这样的设计使程序员必须了解主行所在进程，以及各种进程间的关系，这些逻辑关系必须体现在通信的细节上。

5.1.3 在 HPMR 上实现的 LU 分解

在用 MapReduce 模式在 HPMR 上来实现 LU 分解算法时，依然采取按行顺序划分的方法对矩阵进行划分。根据前述 `sub_matrix` 类的设计方法，分块的信息：总块数、本块的块号、当前主行的行号、起始行的行号；矩阵数据则由我们自己实现的 `matrix` 类来存储，`matrix` 中除了矩阵元素还有两个成员变量行数和列数用来表明当前的子矩阵块的大小。通过起始行行号和 `matrix` 中的行数可以计算出终止行行号，这个数据结构包含了足够的矩阵信息。

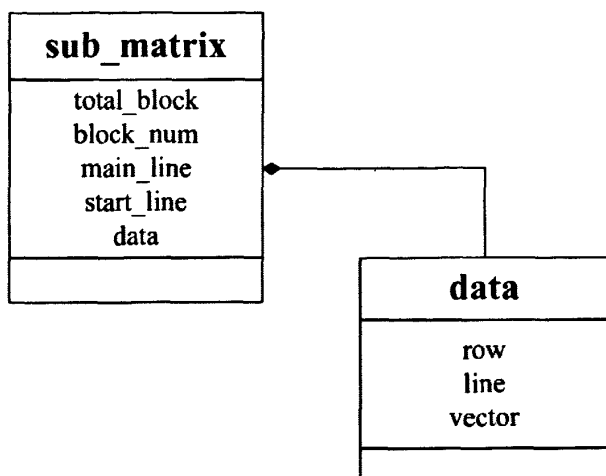


图 5.3 LU 分解中数据结构

由 MPI 上 LU 分解的实现可知，当前主行和需要做变换的行之间存在数据依赖关系。因此，在用 MapReduce 模式来实现 LU 分解算法时，我们需要在 map 中通过让当前主行和包含需要变换行的矩阵分块产生相同的 key 将它们关联起来。经过 HPMR 对中间结果的处理，reduce 的时候具有相同 key 的主行和需要进行消元变换的子矩阵就已经在了一起，只需要进行消元变换。

在 MapReduce 编程模式中最重要的一点就是 key 值的选取与设置。这里我们要先回顾一下 HPMR 在 key 值设置方面的特点。首先在设置方面，跟其它 MapReduce 系统中 key 的定义有一点不同，在 HPMR 中有两个 key: key1 和 key2。在处理中间 key/value 对的时候，将 key1 相同的 value 按照 key2 的不同放在同一个 value 集合中。因此我们在用 key1 来表示关联关系，用 key2 来标识 value 中数据的特性。

有了上述对 key 作用的定位，我们来设计合理的 key 值实现数据的关联。在 LU 分解中，map 的输入参数 value 为包含<总块数><本块的块号><当前主行的行号><起始行的行号><matrix>数据项的 `sub_matrix` 类。map 每次处理一个这样的数据块并为它产生一个唯一的 key 值以与主行数据产生关联。可以很清楚的看

到, 本块块号(`self_block_num`)对于每一个子矩阵分块(`sub_matrix`)来说是各不相同的, 因此我们把它作为 `key1` 值。有上述对 `key2` 值作用的分析可知, `key2` 可以用来区分关联到同一 `key1` `value` 集中性质不同的 `value`。在这里, 性质不同主要指的是主行数据和子矩阵块数据在计算中的角色不同。因为它们都是以 `sub_matrix` 类的对象来表示的, 且在进行消元变换的时候是后者用前者进行计算, 所以一定要有标识来对它们进行区分。虽然可以在 `sub_matrix` 的子矩阵描述部分利用对诸如本块块号等信息赋予特殊值以区分主行, 但是这需要对每个 `value` 值进行解析后才能确认, 而采用 `key2` 无疑是一种更快捷的方式。我们为 `key2` 赋予两种值 `m` 和 `s`, `m` 代表 `main_line_block`, `s` 代表 `self_block`, 由此来区分主行和子矩阵。`key1` 值的选取通常与 `value` 数据的特性相关, 一方面这样的值相对容易确定, 另一方面可以表示数据的某些信息为 `reduce` 或者最终结果的处理提供便利。`key2` 值的作用通常是标识符, 除了本例中的用法, 在矩阵乘法中我们通过对 `key2` 赋值“`A`”和“`B`”来区分同一个 `value` 集中来自不同矩阵的数据。

有了对矩阵数据的划分和 `key` 值的设置以后就可以较容易地写出 `Map` 和 `Reduce` 处理过程, 描述如下:

Map: 检查对当前数据子块是否需要使用当前主行进行变换。即, (a) 如果 `Map` 持有的数据子块不包含主行号 `i` 的数据行且主行号 `i` 没有超越本地数据子块所持有行的行号范围, 则产生 `<key1=本地数据子块号#current_block, key2=m, Val=本地数据子块>`; (b) 如果 `Map` 持有的数据子块包含了主行号 `i` 的数据行, 则首先产生 `<key=本地数据子块号#current_block, key2=s, Val=本地数据子块>`, 如果主行不是本数据子块的最后一行则产生 `<key=本地数据子块号#current_block, key2=s, Val=主行数据>`, 然后针对所有其它需要变换的划分子块分别产生 `<key=其它数据子块号#other_block, Val=主行数据>`。`Map` 伪码如下:

```
Map(Key key, Value sub_matrix){
    //得到主行行号
    main_line_num = sub_matrix.main_line_num;

    if( sub_matrix.higher_than(main_line_num) ){
        //子矩阵最小行号大于主行号, 需要做变换, 产生与主行的关联
        key = (sub_matrix.block_num);
        value = (s, sub_matrix);
    }
    else if( sub_matrix.lower_than(main_line_num) ){
        //子矩阵最大行号小于主行, 变换完毕, 什么都不用作
```

```
}
else if( sub_matrix.has(main_line_num) ){
//子矩阵包含主行，把主行跟自己和其它 block_num 比自己大的子矩阵//
做关联
    int block = sub_matrix.block_num
    if(main_line_num != sub_matrix.high_line_num){
//如果不是本子矩阵的最后一行，则自己未变换完毕，产生跟自己
//关联
        key = (sub_matrix.block_num);
        value = (s,sub_matrix);
    }
    else{
//不产生跟自己的关联
        block += 1;
    }
    for(block; block < sub_matrix.total_block; block++)
    {
//产生主行跟其它子矩阵的关联
        key = (block);
        value = (m,sub_matrix.main_line);
    }
}
}
```

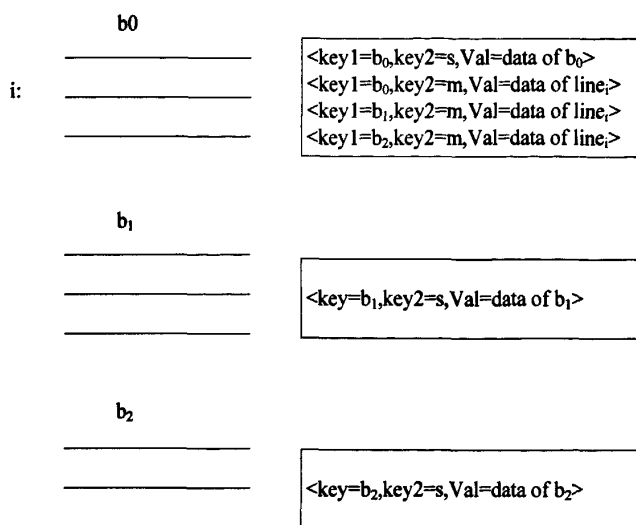


图 5.4 某一轮 MapReduce 过程中 Map 产生的 key/Val 对

图 5.4 显示了某轮 MapReduce 过程中 Map 产生的 key/val, 其中左边给出了进行 LU 分解的矩阵划分, 如子块 b_0 、 b_1 和 b_2 ; 而右边则给出了 Map 产生的 key/val 偶对序列, 其中 b_0 因含有主行 i 而产生 4 个 key/val 偶对, 其中一个为子矩阵块, 三个为需要关联到其它子矩阵块的主行数据。而其余划分分别只产生 1 个 key/val 偶对与主行产生关联。

如过采用行交叉划分, sub_matrix 所包含行的行号非连续。因此在 map 中判断主行是否在本子矩阵内的条件和方法都有所不同。

顺序划分判断是否含有主行:

```
if(main_line_num > sub_matrix.endne)
{
    //do not have mian_line, and not need to compute
    .....
}
else if(main_line_num < sub_matrix.start_line)
{
    //do not have mian_line, but need to compute
    do creare key = self_block value = value
}
else if((main_line_num >= sub_matrix.start_line)
        &&(main_line_num <= sub_matrix.end_line))
{
```

```
//have main_line
do create key = block_num value = main_line
}
```

因为在交叉划分的矩阵块中不能通过起始行号和终止行号两个边界条件来判断是否含有主行，所以只能对子矩阵中的每一行进行测试，其实现形式如下所示：

```
if(main_line_num < sub_matrix.end_line)
{
    bool has=false;
    for each line_num in sub_matrix
        if(main_line_num = line_num)
        {
            //compare every line_num
            has = true;
        }
    if(has)
    {
        do create key = block_num value = main_line
    }
    else
    {
        do create key = self_block value = value
    }
}
```

由上可见，除了判断条件以外没有什么不同，如果将它们封装在一个名为 `has_main_line()` 的函数中则在程序中完全看不出两者有什么区别。而对数据的处理，即，产生 `key` 和 `value` 的方法完全相同。从这一点也可以看出，数据结构的变化对 MapReduce 的算法设计影响是很有限的。

HPMR 的 master 节点会对个 worker 产生的 `key1` 产生路由，`value` 值小的将会把自己的 `value` 发送给 `value` 值最大的 worker。这一点在 LU 分解中是很有优势的，因为主行数据相对来说是比较小的，所以总是会被发送，因此减少发送子矩阵数据块所造成的开销。中间结果处理完成以后，`key1` 相同的矩阵块和主行就会被关联到在同一个 `value` 集中，而这个 `value` 集所对应的 `key1` 就是矩阵块自己的块号，这就是 reduce 过程的输入。

Reduce: 对需要变换的数据子块实施相应的主行变换。即, (a)对经过 MapReduce 运行时汇聚后的<key=数据子块号#block, Val1= 子块号为#block 的数据子块, Val2=主行数据>中的数据子块进行相应的主行变换;更新 Val1 中主行号。(b)并产生<key=数据子块号#block, Val=变换后的子块号为#block 的数据子块>作为下一轮 MapReduce 的输入。伪码如下:

```
Reduce(Key key, Value value_list){
    //从 value_list 得到主行和需要变换的子矩阵
    main_line = value_list.get_main_line();
    sub_matrix = value_list.get_self_matrix();

    //子矩阵用主行对自己进行变换
    sub_matrix.lu(main_line);

    //改变主行行号, 产生输出 key/value 对
    su_matrix.sub_main_line_num();
    key = (sub_matrix.block_num);
    value = (sub_matrix);
}
```

经过一轮的 map/reduce 过程, 就完成了以某一行为主行的 LU 变换, 对于 n 阶的方阵来说, 完成整个 LU 变换的过程仅要进行 $n-1$ 轮这样的变换。HPMR 为用户提供了控制迭代次数的接口, 通过这些接口可以设置 MapReduce 循环的轮数和循环停止的条件。

5.2 QR 分解的实现

5.2.1 QR 分解简介

矩阵 $A=[a_{ij}]$ 是一个 n 阶实矩阵, 对 A 进行 QR 分解, 就是求一个非奇异 (Nonsingular) 方阵 Q 和一个上三角方阵 R , 使得 $A=QR$ 。其中方阵 Q 满足: $Q^T=Q^{-1}$, 称为正交矩阵 (Orthogonal Matrix), 因此 QR 分解又被称为正交三角分解^[51]。

具体过程是, 初始矩阵 A 和单位矩阵 Q 。顺序使用 A 矩阵的每行 $A[i]$ 和对应的 Q 矩阵的每行 $Q[i]$ 对行号大于 i 的 A 矩阵和 Q 矩阵中的各行依次做变换, 直到 i 等于最后一行的行号。单处理器上矩阵的 QR 分解串行算法如下:

输入: 矩阵 $A_{n \times n}$, 矩阵 Q (值为单位矩阵 I)

Begin


```

for j=1 to n do
  for i=j+1 to n do
    (1) sq=sqrt(a[j,j]*a[j,j]+a[i,j]*a[i,j])
    c=a[j,j]/sq
    s=a[i,j]/sq
    (2) for k=1 to n do
      aj[k]=c*a[j,k]+s*a[i,k]
      qj[k]=c*q[j,k]+s*q[i,k]
      ai[k]=-s*a[j,k]+c*a[i,k]
      qi[k]=-s*q[j,k]+c*q[i,k]
    end for
    (3) for k=1 to n do
      a[j,k]=aj[k]
      q[j,k]=qj[k]
      a[i,k]=ai[k]
      q[i,k]=qi[k]
    end for
  end for
endfor
End

```

这是 QR 分解的主体过程，步骤（1）用来计算参数 c 和 s ，步骤（2）用对向量 a_i 和 a_j 以及 q_i 和 q_j 做计算，步骤（3）将计算结果赋值给原来两个矩阵矩阵中对应的行。整个程序中，除了用到输入矩阵 A 和 Q 还用到了四个向量 aj 、 qj 、 ai 、 qi 用来存放临时计算结果。以上程序的运行结果中矩阵 A 的数据就是上三角矩阵 R ，而矩阵 Q 经过转制以后即可得到奇异方阵。由串行算法可以看出，当计算 i ($j < i < n$) 的时候，也同样会更新第 j 行的内容。因此在以第 j 行为基础进行计算的时候，只能是行号大于 j 的行按顺序依次变换，而不能同时进行并行的计算。

5.2.2 使用 MPI 实现的 QR 分解

由于 QR 分解中消去原始矩阵 $A_{n \times n}$ 中的 a_{ij} 的时候，同时要改变第 i 行和第 j 行两行的元素，而在 LU 分解中，仅利用主行 i 变更第 j ($j > i$) 行的元素。因此 QR 分解并行计算中对数据这处理以及交换与 LU 分解不太一样。设处理器个数为 p ，对矩阵 A 按行划分为 p 块，每块含有连续的 m 行向量， $m = \lceil n/p \rceil$ ，这些行块依次记为 A_0 、 A_1 、 \dots 、 A_{p-1} ，分别存放在标号为 0 、 1 、 \dots 、 $p-1$ 的处理器中。

在 0 号处理器中, 计算开始时, 以第 0 行数据作为主行, 依次与第 1、 \dots 、 $m-1$ 行数据左旋转变换, 计算完毕将第 0 行数据发送给 1 号处理器, 以使 1 号处理器将收到的第 0 行数据作为主行和自己分配到的 m 行数据依次做旋转变换; 与此同时, 0 号处理器进一步以第 1 行数据作为主行, 依次与第 2、3、 \dots 、 $m-1$ 行数据做宣传变换, 计算完成后将第 1 行数据发送给 1 号处理器; 如此循环下去。一直到以第 $m-1$ 行数据作为主行参与旋转变换为止。

在 1 号处理器中, 首先以收到的 0 号处理器的第 0 行数据作为主行和自己的 m 行数据做旋转变换, 计算完将该主行数据发送给 2 号处理器; 然后以收到的 0 号处理器的第 1 行数据作为主行和自己的 m 行数据做旋转变换, 再将该行数据发送给 2 号处理器; 如此循环下去, 一直到以收到的 0 号处理器的第 $m-1$ 行数据作为主行和自己的 m 行数据做旋转变换并将第 $m-1$ 行数据发送给 2 号处理器为止。然后, 1 号处理器以自己的第 0 行数据作为主行, 依次与第 1、2、 \dots 、 $m-1$ 行数据做旋转变换, 计算完毕将第 0 行数据发送给 2 号处理器; 接着以第 1 行数据作为逐行, 依次与第 2、3、 \dots 、 $m-1$ 行数据左旋转变换, 计算完毕将第 1 行数据发送给 2 号处理器; 如此循环下去。一直到以第 $m-1$ 行数据作为主行参与旋转变换为止。除了 $p-1$ 号处理器以外的所有处理器都按此规律先顺序接收前一个处理器发送过的数据, 并作为主行和自己的 m 行数据做旋转变换, 计算完毕将该主行数据发送给后一个处理器。然后依次以自己的 m 行数据作为逐行对后面的数据做旋转变换, 计算完毕将主行数据发送给后一个处理器。

在 $p-1$ 号处理器中, 现已接收到的数据作为主行参与旋转变换, 然后依次以自己的 m 行数据作为主行参与旋转变换。所不同的是, $p-1$ 号处理器作为最后一个处理器, 负责对经过计算的全部数据做汇总。具体的并行算法框架描述如下所示:

输入: 矩阵 $A_{n \times n}$, 矩阵 Q (值为单位矩阵 I)

Begin

对所有处理器 my_rank (my_rank=0, \dots , $p-1$) 同时执行如下的算法:

if(my_rank=0) then /*0 号处理器*/

for j=0 to m-2 do

for i=j+1 to m-1 do

Turnningtransform() /*旋转变换*/

end for

将旋转变换后的 A 和 Q 的第 j 行传送到第 1 号处理器

end for

将旋转变换后的 A 和 Q 的第 $m-1$ 行传送到第 1 号处理器

```

end if
if((my_rank>0)and(my_rank<(p-1))) then /*中间处理器*/
    for j=0 to my_rank*m-1 do
        接收左邻处理器传送过来的 A 和 Q 子块中的第 j 行
        for i=0 to om-1 do
            Turnningtransform() /*旋转变换*/
        end for
        将旋转变换后的 A 和 Q 子块中的第 j 行传送到右邻处理器
    end for
    for j=0 to m-2 do
        z=my_rank*m
        for i=j+1 to m-1 do
            Turnningtransform() /*旋转变换*/
        end for
        将旋转变换后的 A 和 Q 子块中的第 j 行传送到右邻处理器
    end for
    将旋转变换后的 A 和 Q 子块中的第 m-1 行传送到右邻处理器
end if
if(my_rank=(p-1)) then /*p-1 号处理器*/
    for j=0 to my_rank*m-1 do
        接收左邻处理器传送来的 A 和 Q 子块中的第 j 行
        for i=0 to m-1 do
            Turnningtransform() /*旋转变换*/
        end for
    end for
    for j=0 to m-1 do
        for i=j+1 do m-1 do
            Turnningtransform() /*旋转变换*/
        end for
    end for
end if
End

```

由 QR 分解的特点, 程序启动的时候只有第一个处理器在做旋转变换, 接着第二个处理器开始, 第三个处理器开始……直到各个处理器依次全部启动, 过程

如图 5.5 所示：

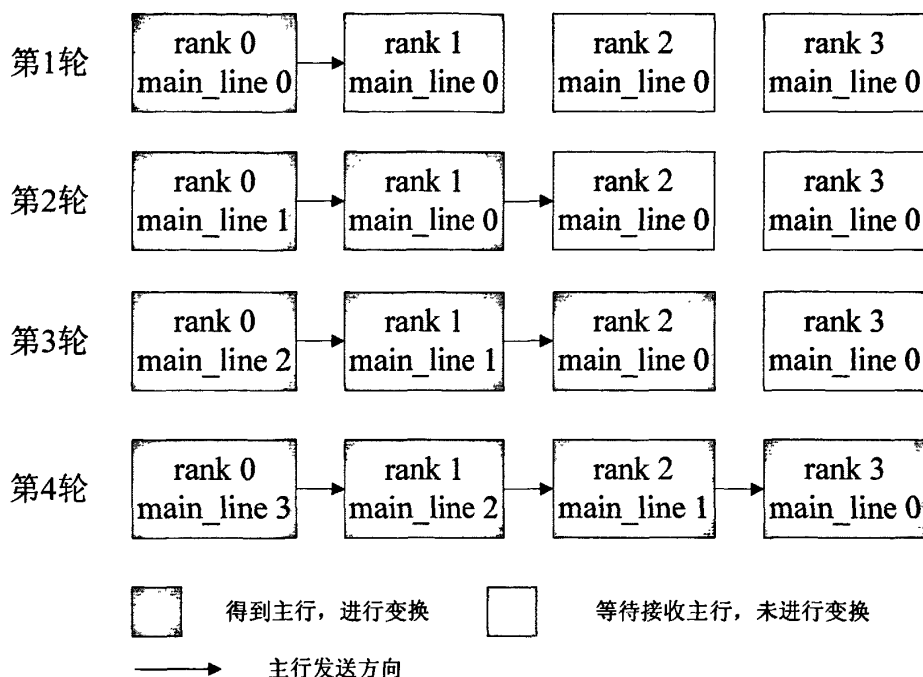


图 5.5 QR 分解流水启动

如果采用行交叉划分, 那么最后一个处理器处理完第一行以后还要把它发送给第一个处理器, 这一行实际发送的次数为 $p(m-1)$ 次, 远远大于按行顺序划分 $p-1$ 次。因此在此情况下, 顺序划分更能减少由于消息通信带来的延迟和开销, 效率更高。

5.2.3 在 HPMR 上实现的 QR 分解

使用 MapReduce 模式来实现 QR 分解, 最先也是最重要的工作就是分析数据依赖, 然后使具有依赖的数据产生相同 key 值以聚合在一起。我们同样采取矩阵的顺序分块方法来划分原始矩阵。在 QR 分解中如果某一个子矩阵块需要进行分解变换, 它一定与某个主行存在数据关联, 这个主行从第 0 行直到这个子矩阵块的最后一行。在 LU 分解中, 所有需要进行分解变换的子矩阵块都与同一主行有依赖关系, 而在 QR 分解中则有所不同, 每个子矩阵块与上一个子矩阵块刚刚变换完的主行产生关联, 并且是两个主行一个来自矩阵 A 一个来自单位矩阵 Q。只要让子矩阵块和这两个数据产生相同的 key 就能让具有相关性的数据聚合到一起。显然, 数据相关性的判断完全遵循数学意义, 由此可以看出 MapReduce 是一种自然的并行, 是一种符合特定应用数学原理的并行模型。它的简洁性在于

用特定应用的数学模型对其数据相关性进行分析,然后在 map 中用 key 表示这种关联,剩下的消息收发操作交给实现了 MapReduce 的系统去解决。

从上面的分析可以清楚的看出,用 MapReduce 模式来实现并行算法最重要的就是首先要发掘数据相关性,而数据相关性往往跟具体应用的数学原理联系的很紧密。一旦数据相关性确定下来,就要设计特定的 key 值来表达和解决了数据相关性,让具有相关性的数据产生关联而且聚合到一起。对关联数据的处理与串行算法中对数据的处理是一致的,相对来说就比较简单了。因此,运用 MapReduce 来解决问题的关键就是数据相关性的发现。

我们已经用 MapReduce 并行编程模型对 QR 分解进行了数据相关性的分析,现在我们来考虑如何在 HPMR 上来实现 QR 分解算法。首先,对输入矩阵进行处理,将矩阵进行按行顺序分块,每个矩阵分块都有若干行,在每一个分块前面加入分块的描述信息:总块数、本块块号、当前主行行号、起始行行号,这跟 LU 分解很类似,只不过这里我们要处理两个矩阵初始矩阵,使用 matrix 类的对象来存储分块矩阵数据 A_i 和 Q_i 。跟前面 LU 分解所采取的数据结构中的分块描述信息是一样的。

map 的输入参数 value 为<总块数><本块块号><当前主行行号><起始行行号><matrix><main_a><main_q>,map 每次处理一个这样的数据块。main_a 和 main_q 都是 matrix 类的一个对象,只不过它们都是一行 n 列的矩阵。初始化的时候, matrix 会读入分块数据内容,而作为两个主行的 main_a 和 main_q 会初始化为空 (NULL)。在 map 中首先通过起始行号和 matrix 中的行数可以计算出本块的终止行号。如果主行在本块内(主行号大于等于起始行号,小于等于终止行号),则从 A_i 和 Q_i 中按主行号来提取出主行 main_a 和 main_q,如果主行不是终止行,则将它们与行号大于主行号的各行进行旋转变换,然后将当前主行行号加一,产生一个关联,其中 value 为自身矩阵块, key1 = 本块块号, key2=s,与 LU 分解一样这个 s 用来表明本 value 包含有子矩阵块数据,另外产生两个 key/value 对用于将主行 main_a 和 main_q 关联给下一子矩阵块, key1=本块块号+1, key2=ma/mq; 否则如果主行是终止行则不做不变换,直接产 main_a 和 main_q 关于下一子矩阵块数据的关联,即设置 key1 = 本块块号+1, key2=ma/mq,且将自身的两个主行都设置为空 (NULL)。如果主行号大于终止行行号,则整个子矩阵块已经转换完毕,但仍然产生自身对主行的关联,只不过这是后的主行为 n (最大行号为 n-1),不会有跟它关联的主行产生。如果主行号小于自身的起始行号,且主行 main_a 和 main_q 均不为空(为空的时候说明它要等待的主行数据还没有关联过来,只能产生自身数据和当前主行的一个关联),则用这两行为主行对所有行进行变换,并且把主行号加一,如果是最后一块则直接输出结果,否则

然后主行加一，然后用 `main_a` 和 `main_q` 中已经变换过的数据为两个主行产生一个与下一个矩阵块的关联，再产生一个自身矩阵块与下一个主行的关联，即设置 `key1=主行行号`，`key2=s`，`value=sub_matrix`。Map 和 Reduce 的伪码如下：

```
Map(Key key, Value sub_matrix){
    if(sub_matrix.has_main_line())
    {
        main_line = sub_matrix.main_line;
        if(sub_matrix.main_line_num != sub_matrix.end_line)
        {
            compute(main_line,sub_matrix.data);
        }
        else
        {
            sub_matix.main_line = NULL;
        }
        output(key = sub_matix.block + 1, main_line);
        output(key = sub_matrix.block, sub_matrix);
    }
    else if(sub_matrix.start_line > sub_matrix.main_line_num)
    {
        if(sub_matrix.main_line != NULL)
        {
            compute(main_line,sub_matrix.data);
            if(sub_matix.block != sub_matrix.total_block)
                output(key = sub_matix.block + 1, main_line);
            else
                print(main_line);
        }
        output(key = sub_matrix.block, sub_matrix);
    }
    else
    {
        output(key = sub_matrix.block, sub_matrix);
    }
}
```

```

}
Reduce(Key key, Value value_list){
    if(value of key2 == 'a' && key2 == 'q' in value_list is not NULL)
    {
        replace value of key2 == 's' with other two
    }
    output(key, value);    //this value's key is 's'
}

```

由 LU 分解中对 HPMR 中间结果处理特点的介绍可知, 通过 master 节点发送过来的路由信息, 每个 worker 都会将主行 key/value 对发送给拥有下一个子矩阵块数据的 worker。因此, 在 QR 分解中通过 HPMR 对中间结果的处理, 除了含有主行的 worker, 其它能够进行旋转变换的那些 worker 的 key 下会有三个 value: 自身子矩阵, 主行 a 和主行 q。

QR 分解中 reduce 的输入 key/values 有以下几种情况: 第一, 已经做完旋转变换的 reduce, 只有一个 value, 其内容是本身子矩阵, 且两个主行都已设为空 (NULL); 第二, 包含主行的 reduce, 只有一个 value, 其内容是本身子矩阵, 但两个主行不为空; 第三, 需要做变换但是主行还没有匹配过来, 比如第 n-1 块在第二轮的时候主行号为一, 但此时这个主行只传递到了第一块子矩阵, 它的 value 也只有一个, 即本身数据; 第四, 需要做变换且主行已经匹配过来了, 如第三中的第一块子矩阵, 这时候 value 一共有三个, 除去一个为自身数据块, 另两个分别为两个主行 (main_a 和 main_q)。

reduce 的主要作用就是将通过 key 匹配来的主行数据替换掉原来的主行数据, 然后产生 key=block_num, value=sub_matrix (本子矩阵块) 的 key/value 对作为下一轮 map 的输入。所以, reduce 只要针对有三个 value 的 key/values, 通过 key2 为 ma、maq 或者 s 确定用主行数据, 再用对应主行替换掉子矩阵块中的相应主行。下一次 map 所处理的输入就会是如下形式: key=block_num, value=(矩阵描述信息, sub_matrix, main_a, main_q), 其中的 main_a 和 main_q 都已经在这次要使用到的主行数据。

由上述 QR 中 map 和 reduce 的功能可知, QR 分解与 LU 分解有很大的不同。首先在 LU 分解中含有主行的各个子矩阵块在 map 的时候产生主行与其它矩阵子块的关联, reduce 的时候的个子块到主行, 并且做变换。而在 QR 分解中含有主行的子块和已经获得上一子块发送过来的主行的子块首先要对自身做变换, 然后才能产生主行对下一子块的关联, reduce 只是用收到的主行来更新现有的主行。前者把具体的计算放在了 reduce 中进行, 后者则是在 map 中进行计算。这

是由算法本身决定的，但是两者的 `map` 完成了相同的工作，那就是产生数据的关联，使对应的主行和子矩阵产生相同的 `key`，这样经过 HPMR 对中间结果的处理，`reduce` 的时候具有相关性的数据就会被放在同一个 `value` 集里面。另外，在通信模式上，LU 分解很像是一个一对多的类广播式的通信模式，拥有主行的子矩阵为所有块号比它大的子矩阵产生主行的 `key/value`，而 QR 中是一个类似点到点的通信，除了第一块和最后一块每一个子矩阵块只从上一个子块得到主行，并把自己刚处理完的主行发送给下一个子块，每个主行数据想在流水线中一样依次通过各个子矩阵块，并参与它的旋转变换，直到经过最后一块后被当作最终结果输出。

5.3 小结

本章在面向高性能计算的 MapReduce 平台——HPMR 上实现了以矩阵计算为典型代表的并行数值计算，并在设计和实现 LU 分解与 QR 分解的同时，根据各自不同的特点对 `map` 和 `reduce` 的功能以及 `key` 值的作用和产生进行了分析。本章通过对编程实现的介绍，探讨了 MapReduce 模型应用于高性能数值计算的可行性及相关方法。实践表明，作为一种新型的并行分布式编程模型，MapReduce 具有较高的并行表述抽象性，可有效地降低并行编程的难度，提升并行编程生产率。

第 6 章 HPMR 上矩阵计算的性能测试及分析

6.1 测试环境

我们使用国家高性能计算中心（合肥）的曙光 TC4000A 作为硬件平台，这台高性能机由 42 个计算节点（node1~node42，node41 和 node42 各有 8G 内存）和两个登陆节点（node43 和 node44）组成，理论峰值达到 2.6752Tflops，每个计算节点配置如下表所示：

表 6.1 测试平台环境表

操作系统	RedHat AS4.6 Kernel 2.6.9-67 .ELsmp		
主机配置	CPU	内存	网络
	双 AMD Opteron2347（4 核，主频 1.9GHz）	4GB	1000M 快速以太网
应用软件	BoostMPI 1.35 g++ (GCC) 3.4.6 HPMR		

由于 HPMR 的底层是采用 BoostMPI 来实现的，所以运行时也是在 BoostMPI 的环境中。HPMR 默认的及其分配策略写在一个名为 machine 的文件中，此文件只包含了 node1~node40 这四十个节点，各进程分将被依次循环分配到 node40~node1。如果想改变此分配策略，可以按照前面 HPMR 介绍中所给出的方法自己写一个 machine 文件来覆盖此文件。

6.2 LU 分解性能测试及分析

6.2.1 矩阵规模增大

对矩阵运算性能测试的方法一方面是不不断增大矩阵数据的规模，另一方面就是在矩阵规模固定的情况下，不断增加进程数目。在 LU 程序的测试中根据这两种方法我们选取不同的矩阵规模，矩阵规模分别为 2000×2000、4000×4000、6000×6000、8000×8000，在每种规模中又分别设定进程数目分别为 10、20、25（30）、40。。根据进程分配策略，各进程会被依次循环分配到 machine 文件中记录的各个机器中去，这里我们采用默认的 machine 文件，即 node40~node1。测试结果如下：

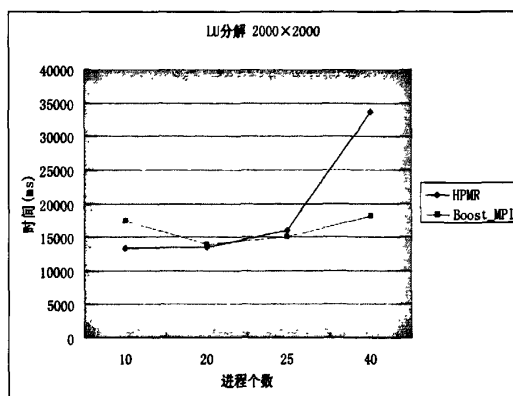


图 6.1 2000 阶方阵 LU 分解性能测试

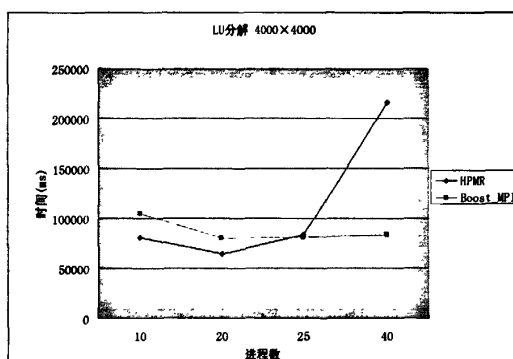


图 6.2 4000 阶方阵 LU 分解性能测试

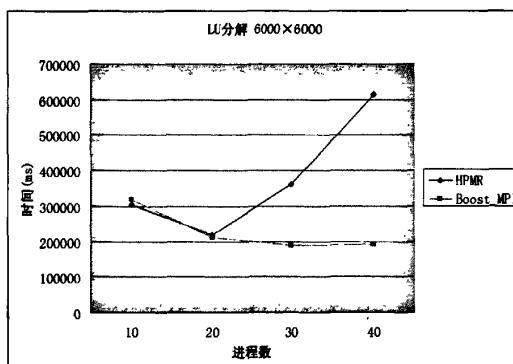


图 6.3 6000 阶方阵 LU 分解性能测试

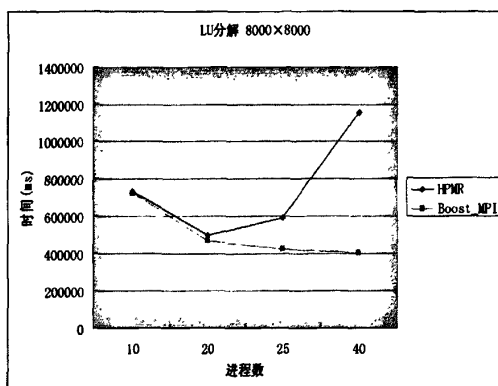


图 6.4 8000 阶方阵 LU 分解性能测试

在四个不同规模的矩阵测试中，不难看出无论是在 BoostMPI 中还是在 HPMR 中，当进程数目由 10 增加为 20 的时候加速效果很明显，然而有 20 增加到 25 到 40 的时候加速效果就不太明显了。这主要是因为进程数目少的时候通信时间相对于计算时间来说是比较小的，当进程数目增大到一定数目的时候，通信代价开始增大，这时候进程数目越多通信代价越高，于是就出现了加速效果减弱的现象，如果继续增加进程数目，例如将进程数目增加到 50 或 60，那么就会出现减速现象。

从实验结果来看，在进程数较少(10 和 20)的时候 HPMR 的性能跟 Boost_MPI 是相当的。在进程数为 10 的时候 HPMR 的性能还要略高。这主要是通信造成的。在 LU 分解中总是有一个进程把主行发送给其它需要做变换的行，这是一个一到多的通信方式。在 Boost_MPI 实现的 LU 分解程序中，这个通信过程是采用广播来实现的。即便是已经对自己的矩阵子块做完 LU 分解的进程也要做出通信动作，这样就增加了通信开销。然而在 HPMR 的通信实现中由于采用的都是点到点的通信方式，所以只有需要做 LU 分解的进程才与当前握有主行的进程进行通信。根据我们的测试来看，在进程数目较少的情况下采用点对点通信方式实现的 LU 分解要比采用广播方式实现的性能好，当进程数目增多的时候广播方式的优点就体现出来了。正如实验结果所展示的当进程数目大于 25 的时候采用广播通信的 Boost_MPI 程序的性能要比 HPMR 好的多了。点到点通信的开销使得 HPMR 的性能开始下降，但是在进程数小于 25 (30) 的时候，已经达到了很好的加速性能。

6.2.2 固定进程数，矩阵规模增大

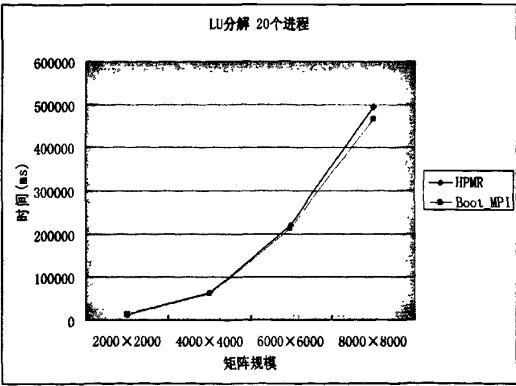


图 6.5 20 个进程下不同规模矩阵 LU 分解性能

图 6.5 为进程数目固定为 20 时，矩阵规模分别为 2000×2000 、 4000×4000 、 6000×6000 、 8000×8000 的时候 BoostMPI 与 HPMR 上实现的 LU 分解的时间对比图。进程数为 20 的时候随着矩阵规模的增大 HPMR 的性能跟 Boost_MPI 还是很相近的，可以看到两条曲线的趋势是一致的，并且 HPMR 那条曲线跟 BoostMPI 很接近，也就说明两者在性能上的差异不是很大。但是如上一节所示在进程数增多的时候用点对点方式实现的影响了一到多通信的性能，如果能够识别这种一到多的通信操作，并且用群集通信的方式来实现，就会在进程数增多的情况下获得良好的性能。而且正如前面使用 MapReduce 模式在 HPMR 上是实现 LU 的过程所展示的，比起使用 BoostMPI 编程，我们能更快速的实现并部署这个程序。

6.3 QR 分解性能测试及分析

6.3.1 矩阵规模增大

QR 的测试方法与 LU 分解一样，只不过在矩阵规模上有所不同，这次我们选取的矩阵规模分别为 1000×1000 、 2000×2000 、 3000×3000 、 4000×4000 和 5000×5000 ，进程数目也同样分别设为 10、20、25、40，只是在矩阵规模为 3000×3000 的时候增加了一个进程数目为 30 的测试。其它测试条件与 LU 分解一摸一样。测试结果如图所示：

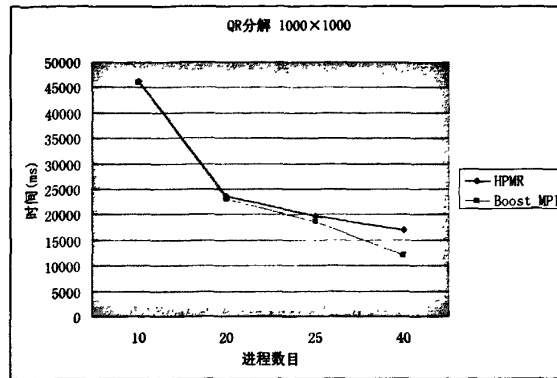


图 6.6 1000 阶方阵 QR 分解性能测试

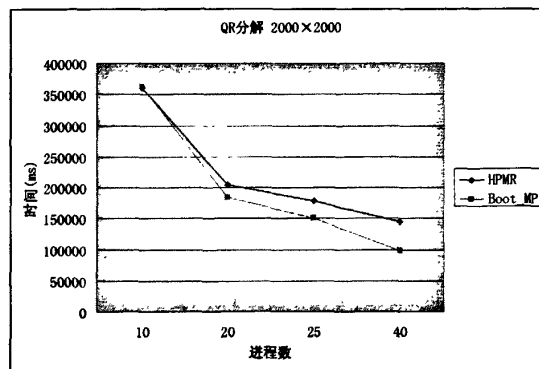


图 6.7 2000 阶方阵 QR 分解性能测试

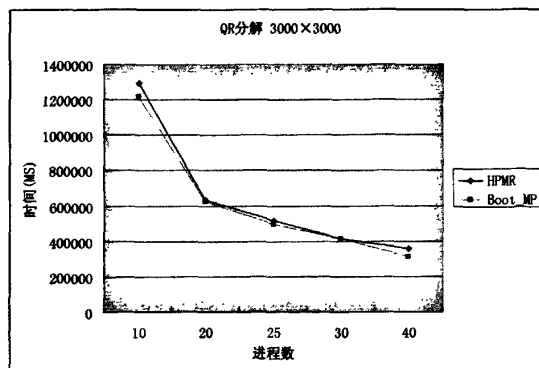


图 6.8 3000 阶方阵 QR 分解性能测试

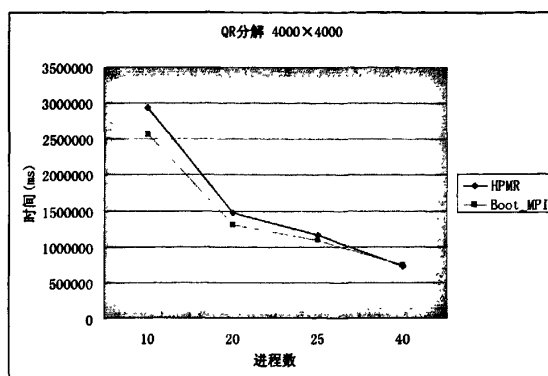


图 6.9 4000 阶方阵 QR 分解性能测试

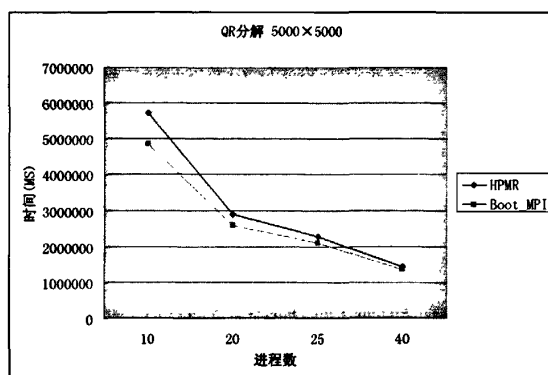


图 6.10 5000 阶方阵 QR 分解性能测试

如图所示,从性能对比来看,QR 分解在 BoostMPI 上的实现跟 HPMR 上的实现差别不是很大,并且没有出现 LU 分解中那种情况,即,在进程数目增大到一定程度时 HPMR 相对于 BoostMPI 性能有显著的下降。这是由 QR 分解本身的特点决定的,在 QR 分解中总是前一个进程把取得的主行发送给后一个进程,不存在 LU 分解那样使用广播操作的通信操作,而这个过程在 BoostMPI 是通过点到点的通信来完成的,而 HPMR 所有的通信都是使用点到点的方式来实现的,所以两者采用了相同的实现方式,因此由于通信产生的性能上的差别并不大。而且相对于 LU 分解中一个进程要对其它所有进程产生通行相比,QR 中的通信代价要小的多,所以在进程数目由 10 向 40 增加的过程中,加速效果虽然有下降但依然很明显。

6.3.2 固定进程数, 矩阵规模增大

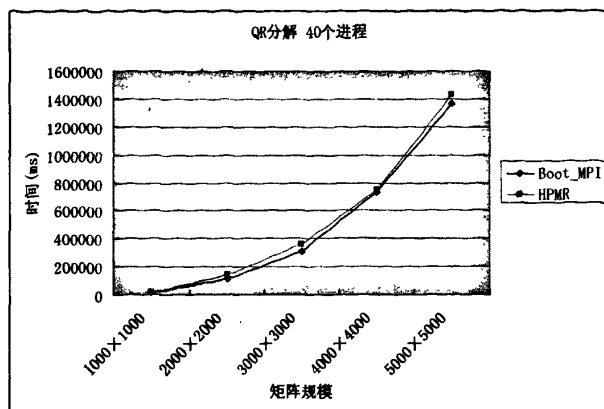


图 6.11 进程数为 40 不同规模矩阵 QR 分解性能

在进程数同为 40 的时候，QR 分解在 HPMR 上实现与在 Boost_MPI 上实现的 QR 分解的性能是很相近的。重要的是 HPMR 的实现相对来说跟符合数学意义，而且简单不用考虑通信的细节。

通过对 LU 分解和 QR 分解的性能测试和对比，说明在 HPMR 系统上采用 MapReduce 模式来解决数值计算问题，不仅隐藏进程的通信、任务的分配等并行编程细节，简化了并行编程的难度，提升并行编程生产率，而且也保证了一定的运行效率。

当然通过分析也可以看到 HPMR 还存在很多可以改进的地方。比如，通信的效率是影响整个 HPMR 性能的一个主要方面。现在我们使用 BoostMPI 支持泛型消息传递，可以很方便的传递像链表、树等复杂的数据结构和对象。在每一轮的 map/reduce 过程中通信时间只有十几到几十毫秒，但是在多轮迭代的过程中，这个时间的累计效应是很明显的。所以我们可以采用单边通信来等方法优化消息通信部分。

6.4 HPMR 上的算法时间复杂度分析

6.4.1 BSP 模型简介

(1) BSP 模型的基本参数

大同步并行 BSP (Bulk Synchronous Parallel) (陈国良 et al. 2004; Khachian N 1979) 模型 (相应地，APRAM 模型也叫做“轻量”同步模型)，其早期最简单的版本叫做 XPRAM 模型，它是作为计算机语言和体系结构之间的桥梁，并以下述三个参数描述的分布式存储的多计算机模型：①处理器/存储模块 (下文也简称为处理器)；②施行处理器/存储器模块对之间点到点传递信息的选路器；③执行以时间间隔 L 为周期的所谓路障同步器。所以 BSP 模型将并行机的特性抽象为

三个定量参数 p 、 g 、 L ，分别地迎处理器数、选路器吞吐率（亦称带宽因子）、全局同步之时间间隔。

(2) BSP 模型的计算

在 BSP 模型中，计算系由一系列用全局同步分开的周期为 L 的超级不 (Superstep) 所组成。在各超级步中，每个处理器均执行局部计算，并通过路由器接收和发送消息；然后做一全局检查，以确定该超级步是否已由所有的处理器完成：若是，则前进到下一个超级步，否则下一个 L 周期被分配给未完成的超级步。

(3) BSP 成本模型

在 BSP 的一个超级计算步中，其计算模型如图 6.12 所示：

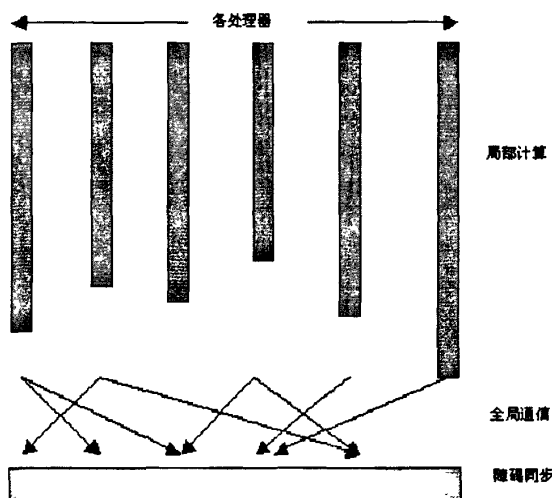


图 6.12 BSP 一个超级计算步中的计算模式

按此可抽象出 BSP 的成本计算模型如下：

$$\text{一超算步成本} = \max_{\text{processes}} \{w_i\} + \max \{h_i g\} + L$$

其中， w_i 是进程 i (process_i) 的局部计算时间， h_i 是 process_i 发送或接受的最大信包数， g 是带宽的倒数（时间步/信包）， L 是路障同步时间（注意，在 BSP 成本模型中并未考虑到 I/O 的传送时间）。所以，在 BSP 计算中，如果用 s 个超级计算部，则总的运行时间为

$$T_{BSP} = \sum_{i=0}^{s-1} w_i + g \sum_{i=0}^{s-1} h_i + sL$$

(4) BSP 模型的性质和特点

BSP 模型是一个分布式存储的 MIMD 计算模型，其特点是：①它将处理器和选路器分开，强调了计算任务和通信任务的分开，而选路器仅施行点到点的消息传递，不提供组合、复制或广播等功能，这样做既掩盖了具体的互连网络拓扑，又简化了通信协议；②采用同步路障方式的以硬件实现的全局同步是在可控的粗粒度级，从而提供了执行紧耦合同步时并行算法的有效方式，而程序员并无过分的负担；③在分析 BSP 模型的性能时，家丁局部操作可在一个事件簿内完成，而在每一超级步中，一个处理器至多发送或接收 h 条消息（称为 h -relation）；④为 PRAM 模型所设计的算法，均可采用在每个 BSP 处理器上模拟一些 PRAM 处理器的方法实现之。

6.4.2 时间复杂度分析

MapReduce 程序在每一轮的运行中都有三个过程。第一是 map 过程，时间用 t_m 表示；第二是中间结果的处理过程，时间用 t_p 表示；第三是 reduce 过程，时间用 t_r 表示。数据分发过程所需时间用 t_d 表示，整个 MapReduce 过程的迭代次数用 n 表示，整个过程所用的时间用 T 表示。

在多次迭代的 MapReduce 程序中，可以把一次 map/reduce 过程看作可在一个时间步内完成的局部操作，且每轮迭代间都会有一个同步过程。这样在 MapReduce 模型中，我们可以使用大同步并行 BSP(Bulk Synchronous Parallel)模型来对其时间复杂度来进行分析。

由上一节 BSP 中一个超级步的计算公式可以得出 MapReduce 中一个超级计算步的成本为：

$$T_i = \max\{t_{mi}\} + \max\{t_{ri}\} + t_p + L$$

这里的 L 为每轮之间的路障同步时间。那么整个 MapReduce 程序所需时间为：

$$T = t_d + \sum_{i=1}^n t_{mi} + \sum_{i=1}^n t_{ri} + \sum_{i=1}^n (t_p + L) + t_q$$

由于 t_m 和 t_r 是数据处理过程所需要的时间，与具体应用的算法有关，在很难在系统中进行优化。 t_p 除了同步以外，key 的统计、产生 kv 路由和 key/value 对的通信占了很大比例。虽然 t_p 也会随着输入数据和不同 key 值的增加而增大，但是通过设计合理的中间结果处理过程是减少 t_p 是提高整个系统运行效率最有效的途径。

对于同一个应用，一次迭代过程中的 t_m 、 t_p 、 t_r 都是固定的。随着数据规模的 n 的增大，时间 T 是线性增长的，显然 MapReduce 算法的时间复杂度为 $O(n)$ 。

第7章 结论

7.1 主要工作及其结论

并行机的发展推动了包括科研、军工、生物工程等国民经济重要领域的发展,同时越来越多的领域对并行计算产生了需求。并行计算已经成为很多领域不可或缺的工具。

并行计算机体系结构的发展给并程序的开发带来了一个又一个的挑战,每种并行编程语言和编程模式的出现都适应了特定的体系结构,但同时也给程序开发人员带来了很大困难。为了能够在不同的体系结构下进行编程,即使是熟悉并程序开发的程序员也要不得不学习、了解新语言的特性,及其应用特点。而对于普通串行程序员来说,这无疑是阻碍它们迈入并行编程领域的一道极高的门槛。因此如何为程序员提供一个简单的并行编程模式,掩盖并行编程的复杂度,降低并程序开发难度,提高其开发效率成为了一个关键问题。为此,本文的主要研究工作围绕着 MapReduce 编程模式与各种并行模式的关系,及其应用和设计方面等几个方面展开。本文的研究工作与成果如下:

- 1、研究了并行算法空间中由任务组织、由数据分解组织和由数据流组织这几种并行问题组织原则与 MapReduce 模式的关系,综合 MapReduce 模式以及几种原则的特性,给出了如何用 MapReduce 对这几种不同原则描述的问题进行表述,体现了 MapReduce 的通用性和易用性。

- 2、将 MapReduce 与任务划分、分治策略等并行问题分析模式相结合,用 MapReduce 来描述这些并行模型,并且对每个模式中出现的典型应用或算法进行分析。通过对 MapReduce 在这些模式上的应用和对具体应用的分析,给出了使用 MapReduce 模式分析问题的三个步骤及方法。

- 3、给出了使用 MapReduce 模式所用到的数据结构,其内容即数据和数据特征。并结合矩阵数据划分的分析过程,给出了数据结构的设计要点和方法。研究了 key 值与通信模式的关系,分析了 key 值的作用,给出了选择、设计 key 值的原則。

- 4、将 MapReduce 引入数值计算领域,在 HPMR 上研究了数值计算的应用。使用 MapReduce 模式在 HPMR 上设计和实现了矩阵计算中的 LU 分解与 QR 分解。在对具体问题的实现和分析过程中研究了使用 MapReduce 的步骤和各种数据结构的设计,体现了这种模式的简洁性和易用性。

通过实验测试表明,在我们所提出的对问题的分析和解决方案下,使用 MapReduce 模式能够简洁的对数值计算中的很多应用进行够简洁的描述,并且能够简明的对数据进行划分和处理,而且在 HPMR 上实现的两种矩阵分解也有着

良好的性能，这种模式在数值计算领域有着强大的生命力。

7.2 未来工作和展望

并行编程模式和语言的多样性增加了并行应用的开发难度，而大多数并行编程语言抽象层次过低，导致使用它们对问题进行描述的复杂度很高。因此，提供一个简单易用的并行编程模型成了迫切的需求。

MapReduce 模式是一个高层次抽象的并行编程模型，本文的工作已经体现出 MapReduce 的通用性和易用性。以后我们的工作将围绕下面三点展开：

首先，进一步研究使用 MapReduce 模式的应用特点，对用这种模式分析问题和设计解决方案的方法进行更深入的探讨，提出更为规范、细致且通用的步骤，使其应用更为方便。

再者，将更多的应用引入 MapReduce 模式，由于使用 MapReduce 对数值计算进行分析和实现还很少，因此把更多数值计算领域的问题引入进来是一个重要的工作。

还要面向各种不同种类的应用比如矩阵计算和计算物理、计算化学、流体力学等工程领域设计通用的数据结构和通用的编程架构，方便具体应用的快速实现和部署。

随着高性能计算需求的不断增长，MapReduce 编程模式的优势必将日渐突出，而不断对其特性进行深入研究，发挥它在各方面的潜力是我们未来工作的目标。

参考文献

- Eric E. Johnson, Completing an MIMD multiprocessor taxonomy, ACM SIGARCH Computer Architecture News, Volume 16 Issue 3, June 1988
- Howard Jay Siegel, S. Diane Smith, Study of multistage SIMD interconnection networks, Proceedings of the 5th annual symposium on Computer architecture ISCA '78, April 1978
- John C. Hunter, John A. Wertz, Multi-node cluster computer system incorporating an external coherency unit at each node to insure integrity of information stored in a shared, distributed memory, Patent number:5394555, Assignee: Bull HN Information Systme Inc, Feb 28, 1995
- R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, and D.M. Tullsen, "Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction," in Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36), Dec. 2003
- Linda Wilson. International Technology Roadmap for Semiconductors, 2002, <http://public.itrs.net>
- K. Asanovic, R. Bodik, B. C. Catanzo, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, U. C., Berkeley, December 18 2006
- Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The Case for a Single-Chip Multiprocessor. Proceedings Seventh International Symp. Achitectural Support for Programming Languages and Operating Systems(ASPLOS VII), Cambridge, MA, October 1996
- Multi-Core processors—the next evolustion in computing. 2005
- Ron Kalla, Balaram Sinharoy, Joel M. Tandler IBM POWER5 CHIP: A DUAL-CORE MULTITHREADED PROCESSOR. IEEE 2004
- B. Flachs et al., A Streaming Processor Unit for a Cell Processor, ISSCC Dig. Tech. Papers, Paper 7.4, 134-135, February, 2005
- D. Pham et al., The Design and Implementation of a First-Generation Cell Processor, ISSCC Dig. Tech. Papers, Paper 10.2, 184-185, February, 2005
- M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell's multicore architecture. IEEE Micro, 26(2):10–24, 2006.
- H. P. Hofstee. Power efficient processor architecture and the Cell processor. In HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture, pages 258–262, 2005.

- J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4):589–604, July 2005.
- Samuel Williams, John Shalf, Leonid Oliker, Shoal Kamil, Parry Husbands, and Katherine Yelick. The Potential of the Cell Processor for Scientific Computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20. ACM Press, 2006
- Calvin Lin, Lawrence Snyder, 并行程序设计原理 (英文版). 北京: 机械工业出版社, 2008
- Snir M. MPI: the complete reference. Cambridge, Mass.: MIT press, 1996, 336
- Michael J. Quinn, *Parallel Programming in C with MPI and OpenMP*, 2005
- MPICH2, <http://www-unix.mcs.anl.gov/mpi/mpich/>, 2005
- LamMPI, <http://www.lam-mpi.org/>, 2007
- Douglas Gregor, Matthias Troyer, Boost.MPI, http://www.boost.org/doc/libs/1_38_0/doc/html/mpi.html#mpi.intro, 2007
- OpenMP, <http://www.openmp.org/drupal/>, 2008
- Rohit Chandra et al, *Parallel Programming in OpenMP*, 2000
- Y. Charlie Hu, Honghui Lu, Alan L. Cox and Willy Zwaenepoel, *OpenMP for Networks of SMPs*. Copyright 2000 by Academic Press.
- CHEN Yong, CHEN Guo-liang, LI Chun-sheng, HE Jia-hua, Hybrid Programming Model Research on SMP Cluster Architecture. *MINI-MICRO SYSTEMS* Vol.25 No.10 Oct.2004
- Tanaka Y, Matsuda M, Ando M, Kazuto K and Sato M. Compas: a Pentium pro PC-based SMP cluster and its experience. *IPPS Workshop on Personal Computer Based Networks of Workstations*. 1998, 486–497.
- Lorna Smith and Mark Bull, Development of mixed mode MPI/OpenMP applications. *Scientific Programming* 9(2001) 83–98
- Frank Cappello, Deniel Etiemble, MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks *Proceedings of the 2000 ACM/IEEE conference on Supercomputing(CDROM) Supercomputing '00* November 2000
- C. Lin and L. Snyder et al., ZPL: an array sublanguage. In Banerjee, U., Gelernter, D., Nicolau, A., and Padua, D. (Eds.). *Languages and Compilers for Parallel Computing*, 6th International Workshop Proceedings, Springer-Verlag, New York, 1994
- B.L. Chamberlain, S.-E. Choi, et al. ZPL: A Machine Independent Programming Language for Parallel Computers. *IEEE Transactions on Software Engineering, Parallel Processing Letters*, vol. 26(3), pp. 197–211, 2000.
- ZPL, <http://www.zope.org/Resources/ZPL>, 2009
- BLELLOCH, G. E. 1993. NESL: A nested data-parallel language. Tech. Rep. CMU-CS-93-129,

- Carnegie Mellon University, April
- Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11):1526-1538, November 1989
- Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. NESL user's manual (for NESL version 3.1). Technical Report CMU-CS-95-169, Carnegie Mellon University, July 1995
- John Greiner and Guy E. Blelloch. The NESL cost semantics. In preparation, 1995
- Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Mass., 1990.
- Jeffrey Dean, Sanjay Ghemawat. MapReduce: simplified data processing on large clusters[C]// 6th Symposium on Operating Systems Design and Implementation. USA, 2004
- Ralf Lämmel, Google's mapreduce programming model-revisited[C]// Data Programmability Team Microsoft Corp. USA, Redmond, 2007.
- Luiz A. Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22-28, April 2003
- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In 19th Symposium on Operating Systems Principles, pages 29-43, Lake George, New York, 2003. To appear in OSDI 2004
- C. Chu, S. Kim, Y. A. Lin, Y. Y. Yu, G. Bratski, A. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *NIPS 19*, 2007
- Hadoop, <http://hadoop.apache.org/>, 2008
- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, New York, NY, USA, 2003. ACM Press
- Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, CA, USA, 2006. USENIX Association
- B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A mapreduce framework on graphics processors. In *PACT*, 2008
- A. Ailamaki, N. K. Govindaraju, S. Harizopoulos, and D. Manocha. Query co-processing on commodity processors. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 1267-1267. VLDB Endowment, 2006
- C. Ranger, R. Raghuraman, A. Penmetsa, G. Bratski, and C. Kozyrakis. Evaluating mapreduce for

- multi-core and multiprocessor systems. In Proc. of HPCA, pages 13–24, 2007
- M. Kruijf and K. Sankaralingam. MapReduce for the Cell B.E. Architecture, TR1625, Technical Report, Department of Computer Sciences, The University of Wisconsin-Madison, 2007
- William H. Press, Saul A. Teukolsky, and William T. Vetterling. Numerical recipes in C: The Art of Scientific Computing, 2nd edition. Cambridge University Press, 1993
- Philip Alpatov, Greg Baker, Carter Edwards, John Gunnel, Greg Morrow, James Overfelt, Robert van de Geijn, and Yuan-Jye J. Wu. PLAPACK: Parallel linear algebra package design overview. In Proceedings of the 1997. ACM/IEEE Conference on Supercomputing, ACM Press, 1997
- Khachian N. A new polynomial time algorithm for linear programming. Soviet Math. Dokl. 1979(4):191~194
- Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill, 并行编程模式, 北京: 清华大学出版社, 2005.11
- 郑启龙, 王昊, 吴晓伟, 房明. HMPR: 多核集群上的高性能计算支撑平台. 微电子学与计算机. 25卷, 第9期, 2008年9月
- 陈国良, 并行算法的设计与分析. 北京: 高等教育出版社, 2002
- 陈国良, 安虹, 陈峻, 郑启龙, 单久龙. 并行算法实践[M]. 北京: 高等教育出版社, 2004.

附录 1 插图索引

图 2.1	SMP 结构图	5
图 2.2	cluster 结构图	6
图 2.3	PVP 结构图	7
图 2.4	Cell 结构概要图	9
图 2.5	OpenMP 体系结构	11
图 2.6	混合编程模型中进程与线程的关系	12
图 3.1	map 和 reduce 函数的相关类型	18
图 3.2	Google MapReduce 运行示意图	19
图 3.3	Hadoop 中分布存储与并行计算	20
图 3.4	基于 GPU 的众核 (many-core) 结构	21
图 3.5	Mars 的工作流程图	22
图 3.6	Phoenix 数据流程图	23
图 3.7	master 和 worker 的关系	25
图 3.8	HPMR 的三个模块	26
图 3.9	HPMR 执行流程图	27
图 4.1	“算法结构”空间的总体框图和它在模式语言中的位置	29
图 4.2	算法结构空间决策树	30
图 4.3	按任务划分并行结构图	32
图 4.4	分治策略	34
图 4.5	分治策略的 MapReduce 描述	36
图 4.6	flag 值和 key 值的设置	37
图 4.7	MapReduce 数据结构设计	40
图 4.8	矩阵的顺序带状划分	42
图 4.9	矩阵的交叉带状划分	42
图 4.10	sub_matrix 类	43
图 5.1	串行 LU 分解代码	45
图 5.2	并行 LU 分解核心代码	46
图 5.3	LU 分解中数据结构	47
图 5.4	某一轮 MapReduce 过程中 Map 产生的 key/Val 对	50
图 5.5	QR 分解流水启动	56
图 6.1	2000 阶方阵 LU 分解性能测试	62
图 6.2	4000 阶方阵 LU 分解性能测试	62
图 6.3	6000 阶方阵 LU 分解性能测试	62
图 6.4	8000 阶方阵 LU 分解性能测试	63
图 6.5	20 个进程下不同规模矩阵 LU 分解性能	64
图 6.6	1000 阶方阵 QR 分解性能测试	65
图 6.7	2000 阶方阵 QR 分解性能测试	65
图 6.8	3000 阶方阵 QR 分解性能测试	65
图 6.9	4000 阶方阵 QR 分解性能测试	66
图 6.10	5000 阶方阵 QR 分解性能测试	66
图 6.11	进程数为 40 不同规模矩阵 QR 分解性能	67
图 6.12	BSP 一个超级计算步中的计算模式	68

附录 2 表格索引

表 4.1	通信方式与 key 值关系表.....	41
表 6.1	测试平台环境表	61

致 谢

在此衷心感谢我的导师郑启龙副教授，他严谨的治学态度，一丝不苟的科研精神和渊博的学识让我受益匪浅。郑启龙副教授在我三年的研究生生涯中在学习、生活和科研的各个方面给了我很大的支持，在他的指引下我不断取得进步，郑老师的言传身教不光在科研方面并且在各个方面的都让我受益终生。

感谢王昊和房明两位同学在 MapReduce 模型的应用、设计与分析的研究过程中所提出的宝贵意见和指导，感谢他们的热烈讨论给我带来的思路；感谢 04 级研究生胡晨光，05 级研究生刘术娟、栾俊，博士生杨晓奇，07 级硕士研究生汪胜、王向前，08 级研究生汪睿、卢世贤、博士生夏霏，他们在组会上积极参与讨论，文中的很多思路都是从这些激烈的讨论中得来的，在此向他们表示诚挚的谢意。

感谢实验室中徐云副教授，博士研究生赵裕众，硕士研究生张坤鹏、王志浩、余林斌、张宏、雷一鸣、邵明芝等同学在学习和生活中给我的帮助。

特别感谢我的家人多年来对我的关心和支持，他们是我学习和工作取得进步的永恒动力。

2009 年 5 月

在读期间发表的学术论文与取得的研究成果

参加的科研项目:

- [1] 当代并行机的并行算法应用基础研究, 国家自然科学基金重点基金(No.60533020), 2006—2009。
- [2] 基于 MapReduce 的统一并行编程模型研究, 安徽省自然科学基金(090412068), 2009-2010。
- [3] 并程序设计模型研究, Intel 高等研究基金(No. 4507336209), 2007。
- [4] 巨型计算机开发与应用 web 环境系统, 北方计算中心, 2005 - 2007。

已发表的论文:

- [1] 郑启龙, 王昊, 吴晓伟, 房明. HPMR: 多核集群上的高性能计算支撑平台. 微电子学与计算机, 2008, Vol25, No.9, pp21—23, 27。
- [2] 郑启龙, 栾俊, 房明, 吴晓伟; CCSim: 基于 Pin 的 CMP Cache 访问模拟器. 微电子学与计算机, 2008, Vol25, No.10, pp5—7, 11。

待发表论文:

- [1] 郑启龙, 吴晓伟, 房明, 王昊, 汪胜, 王向前; 基于 HPMR 的并行矩阵计算.(计算机工程), 2009 (已录用)