第 28 卷 第 1 期 力 学 进 展 Vol. 28 No. 1
1998 年 2 月 25 日 ADVANCES IN MECHANICS Feb. 25, 1998

并行计算机和计算流体力学并行算法 *

Dirk Roose Rafael Van Driessche
Dept. Computer Science, K. U. Leuven, Belgium

摘 要 对研制计算流体力学高效并行算法及软件具有重要意义的并行计算问题提出了导引性的看法. 首先综述了并行计算机的主要设计特征并简要描述了市场现有的几种并行系统. 接着介绍了一些有关研制并行算法及评价其性能的重要概念. 然后讨论了如何使分布式内存 并行计算机的运行 负载不平衡和通信开销达到最小. 最后列举了计算流体力学某些算例的测试结果. 本文的重点是结构网格和分程序结构网格的应用,但这些概念和方法对非结构网格同样有效.

关键词 并行系统,并行算法,并行性能分析,计算流体力学

1 并行计算

1.1 并行计算机结构

1.1.1 Flynn 分类法

许多年来高性能计算机的分类使用的是 Flynn 分类法. 这种分类法的基础是指令流和数据流的处理方式(单个或多个指令/数据流),它将计算机分成如下三大类[1]:

单指令单数据(SISD)系统 它是包含一个中央处理器(CPU)的常用系统,如工作站和计算服务器,因此能以串行模式处理一个指令流. 现在许多大型计算服务器或巨型机具有不止一个 CPU,但它们大多用于同时处理一些彼此无关的作业(指令流). 因此,这种系统应该作为一组 SISD 机器看待.

单指令多数据(SIMD)系统 这种系统具有许多(简单的)处理元件,元件数目从 1024 至 64 K,可以在锁定的步骤里对不同数据执行同一指令. 即以单个指令并行地操作多个数据项目. 过去,SIMD 机器如 Thinking Machines 的 ConnectionMachine CM-2 和 MasPar 已经相当成功. 如今,除了还存在于某些受高结构化数据系统和数据访问方式支配的特殊应用领域(如图像处理)的系统外,SIMD 结构几乎消失了.

多指令多数据(MIMD)系统 在"多指令、多数据"系统中,各处理器对各自的数据独立地执行不同的指令流,所设计的硬件和软件能使各处理器有效地配合. 当由不同处理器执行的任务共同形成一个作业时,就发生了并行处理.

向量处理器 经常可认为是 SIMD 系统的子系统. 向量处理器具有特殊硬件("向量元件")

*本工作由国家自然科学基金和国家教委青年教师资金资助.
为了便于读者理解论文内容,译文中有些地方由译者作了一些引申性说明
收稿日期: 1996-11-18,修回日期: 1997-10-08

以流水线方式来完成对含有相似数据的数组的运算. 这些向量元件能够以每时钟周期一次、两次和在特殊情况下三次的速率输出结果. 在编程者看来, 向量处理器在执行向量运算方式时, 是以几乎并行的方式(SIMD 方式) 来操作它们的数据的. 向量处理器用于 Cray Y-MP C90, J916 和 T90 系列, ConvexC 系列, Fujitsu VP 系列, NEC SX 系列, 等等.

浮点运算的流水线操作在用于高性能工作站的 RISC 处理器中也是一个关键的概念. 一些先进的 RISC 处理器也能并行地执行几个指令(如"双指令模式").

1.1.2 MIMD 系统的内存组织形式

并行计算机也可根据其他标准来分类. 根据内存组织方式可以进一步划分 MIMD 系统.

共享内存 MIMD 系统 在共享内存 MIMD 系统中, 所有处理器使用一个公共内存. 在共享内存系统中的主要设计问题是处理器与存储器(或存储模块) 之间的连接. 当增加处理器数量时, 存储器连接的总带宽在理想情况下应随处理器数目 P 线性地增加. 遗憾的是, 完全相互连接代价很高, 需 $O(P^2)$ 个连接. 于是, 人们采用了各种不同的连接网络, 其中一些如图 1 所示. 用十字形网络需 P^2 个连接, 用 Ω 形网络需 $P\log_2 P$ 个连接, 而一个中心芯片只对应于一个连接. 在当今所有的多处理器向量机器中都使用十字形网络. 由于受互连网络容量或造价的限制, 共享内存并行计算机的处理器数目无法达到非常之大.

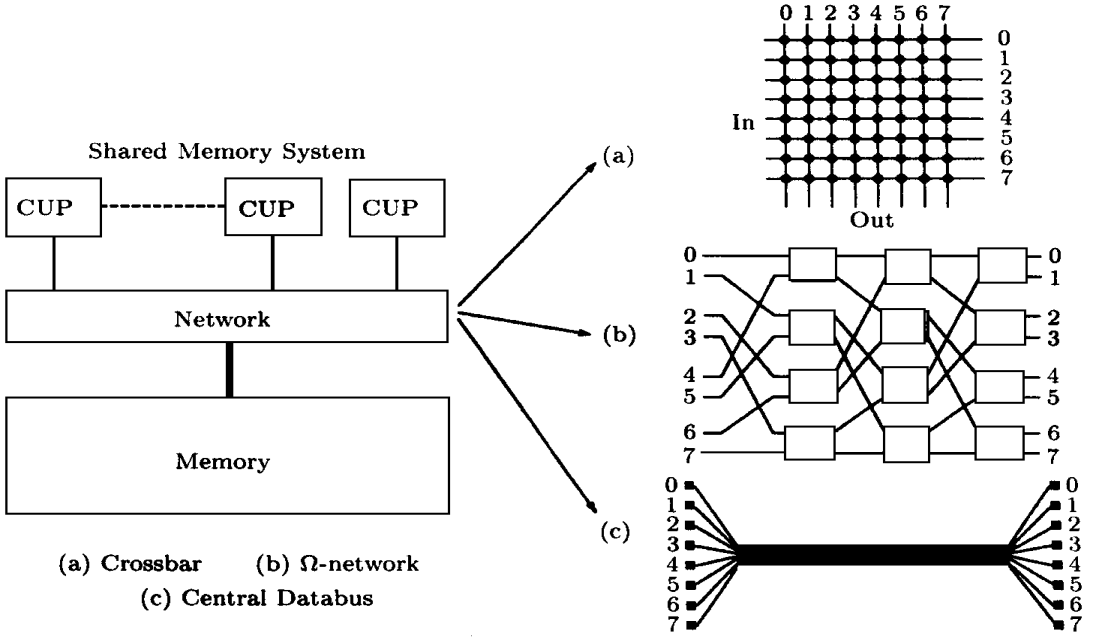


图 1 共享内存 MIMD 并行计算机 的可能连接方式

共享内存概念已经在多处理器向量机器(Cray X-MP, IBM 3090, 它们的后继者和其他厂商的类似系统) 中使用了约 10 年. 然而这些系统并不被经常用作真正的并行系统: 大部分作业只用一个处理器. 其中一个原因是处理器为有限数目(经常为 4 个或 8 个), 这限制了可能的并行加速比. 而且, 由于分时, 正常情况下用户在一个指定时刻下不能完全控制分配给他工作的处理器数目. 这也会限制所能得到的加速比.

现今, 一些厂商(Convex, Silicon Graphics, ...) 提供的基于 RISC 处理器的共享内存 MIMD 系统, 处理器数目多达 20 个.

分布式内存 MIMD 系统 一台分布式内存 MIMD 并行计算机由一些处理器组成, 每个处理器都有自己的局部内存, 它们由通信网络来相互连接. 每个处理器和它的局部内存合称处理

结点. 每个处理结点实际上就是一个完整的计算机,它运行时独立于其他结点. 处理结点只能通过通信网络传递消息来进行通信.

对于分布式内存机器,通信网络结构也是极重要的. 理想情况下,人们想要一个全连接系统,其中每个处理结点可直接连接于其他每个结点. 但是当结点数目很大时这是不可能的. 因此处理结点的布置是以某种互连布局来调度的. 需要在各种连接方式和它们的造价之间进行权衡以选取其中之一.

超立方布局 在过去曾用于一些系统中. 超立方布局的一个有趣的特征是,如果处理结点有 $P = 2^d$ 个,则网络"直径"(即任意两结点之间连接的最大数目)是 d. 因此,直径增长只是结点数目的对数. 另外,许多其他布局(例如树、环、二维和三维网格) 可以通过超立方布局来模拟,这是因为这些布局是超立方布局的子系统.

在现今的并行系统中,网络布局和通信直径不太重要,因为这些系统使用了某种形式的消息"小通道". 这意味着当两个结点间通信路径一建立,数据就通过这条路径输送而不用妨碍中间结点的操作. 除了建立结点间通信路径的一小部分时耗外,通信时耗已完全与结点间距离无关了.

一些系统对网络使用了 二维或三维网格 结构. 其基本原因是这种互连布局对大部分用于大规模科学计算的算法已经足够了,而使用更高级的互连结构很难获得补偿. 在另一些系统中使用一种多级网络,如图 1 和 4 所示的Ω形网络. 多级网络的优点是,二等分通信带宽与处理器数目成线性关系而同时保持每个处理器通信连线的数目固定. 分布式内存系统的二等分通信带宽定义为系统的一半($P/2$ 个处理器) 与另一半相连接时所有通信连线的有效带宽.

分布式内存系统的一个重要优点是这种结构比共享系统更少受到可扩展性问题的干扰. 网络(在限定带宽时) 只在处理结点间有通信时才被用上,而不是在每次内存取时使用. 缺点是网络上通信的延迟大大高于共享内存系统中使用共享数据引起的延迟. 当问题的特征决定了处理器间需进行频繁的通信时,也许只能达到理论峰值速度的一小部分.

第一代分布式内存 DM-MIMD 系统是基于简单,便宜的微处理器. 因此,即使相互连接的处理器数目有 100 至 1000 个,这些机器的峰值性能也低于典型的向量处理器和共享内存超级并行计算机的性能. 如今,分布式内存并行计算机的性能经常超过传统的超级计算机. 这是因为分布式内存系统所用的 RISC 处理器性能的快速提高和网络技术的大大改进. 此外,许多系统现在装有允许快速并行磁盘输入/输出的高级硬件和软件(即"并行 I/O 系统"). 因此,分布式内存并行系统在侧重于计算速度的领域如计算流体力学中很快占有重要地位.

分布式内存机器有 Intel Paragon, CM-5, Cray T3D, IBM SP2. 分布式内存系统处理器数目可达上千个,但大多数系统在 16 至 128 个之间.

一些年来,网络或工作站机群被当作"廉价"的分布式内存并行计算机来使用. 工作站机群的使用使得一些普通情况下未被使用的计算能力得到利用. 当然,如果工作站通过 Ethernet(或甚至通过快速 FDDI 互连) 简单连接在一起,则能够像并行系统那样有效使用的工作站数目就受到限制,这是因为这类连接使通信性能很低. 一些工作站厂商提供了互连开关来达到高速通信(如 Digital). 这种群体工作方式填补了与"真正并行计算机"的差距.

因此现今被用作并行计算机的系统很广泛,它包含从小工作站机群到由许多处理器和高级通信网络技术组成的大系统.

混合内存系统 尽管共享内存系统与分布式内存系统似乎有明显的区别,但许多并行系统都使用一种混合型内存组织. 在共享内存系统中,每个处理器可能有一个大型超高速缓存存储器,它可以看作一个局部内存. 一些系统有两级组织:处理器以共享内存模块组织成组,并通过通信网络来连接不同的处理器组. 最后,分布式内存系统可以包含取得其他处理器内存中数据的软硬件支持,并使之对用户透明化. 根据这种支持的具体形式,它们分别称为"虚拟共享内

存”,“全局共享内存”,“全局虚内存”,等等.

内存等级和性能 在向量处理器和高级 RISC 处理器中,处理器运算速度都比主存储器中的数据读写快得多. 向量处理器中的向量寄存器和 RISC 处理器中的大型高速缓存存储器置于处理器与主存储器之间. 这些速度非常高的存储模块用于维持处理器不间断地计算而不需要频繁地访问主存储器.

向量寄存器,高速缓存,局部存储器和/或全局(共享)内存一起形成内存等级. 对一个给定的应用程序来说,它能达到的性能决定性地依赖于内存等级中“较高级”的存储器中的数据的(再)利用. 因此,为了取得高性能,算法对数据的访问在(地址)空间上和时间上都应该具有局部性.

1.2 并行计算机编程

我们已经指出在共享内存和分布式内存并行体系结构中并没有明显的区别,还指出,一些新的并行系统采用了混合型内存机制. 然而,我们可以明显地区别两种不同的编程模式,即共享内存编程模式和分布式内存编程模式.

在两种模式中,程序在运行时分裂成一些并行执行的子进程. 大部分情况下,一个处理器只处理一个子进程,因此我们将在下面的讨论中使用“处理器”这个词,虽然“子进程”这个词更为精确些.

1.2.1 共享内存编程模式

不论内存的物理组织形式如何,共享内存编程模式的基础是公共的即全局的地址空间的存在,也就是每个处理器能通往每个内存地址. 因此,处理器之间的通信通过访问(写或读)共享数据来实现. 访问共享数据所花费的时间可能很不一样,这依赖于共享数据的物理位置(高速缓存,局部存储器,另一个处理器的存储器).

也有可能出现这样的情况,不同处理器要同时使用公共内存的同一部分. 在此情况下,有必要进行进程的同步控制. 在执行一个程序的串行部分之前也需要同步控制,以确定所有的处理器在串行部分之前已完成它们的并行操作.

这样,可能由于“内存冲突”、同步控制和“串行瓶颈”而使性能严重地恶化. 另外,(并行)任务的生成导致的额外开销会变得很高.

目前,在共享内存式编程模式中有能自动实现并行的 Fortran 和 C 编译器. 程序员也可以通过命令影响并行控制(如同对向量处理那样). 并行控制可以针对循环,也可以针对任务(子程序),前者也称为“细粒度”或“微观”并行控制,后者也称为“粗粒度”或“宏观”并行控制.

1.2.2 分布式内存编程模式

在分布式内存编程或消息传递模式中,处理器只能访问自己的内存. 无论何时,一个处理器需要另一个处理结点中的数据时,数据必须在处理结点间传送. 这样一个消息传递或通信步骤包括发送结点的消息准备,通过通信网络的输送和目标结点的消息接收. 当消息传递模式是用于共享内存系统中时,输送实际上被共享内存的消息存储所代替.

另外,在分布式内存模式中也有同步控制问题. 可能一个处理器中还没有另一个处理器所需要的数据;在此同步点处后一个处理器必须等前一个处理器赶上来. 为确保各处理器的处理正确进行也可能需要同步控制.

尽管每个处理器能运行一个不同的程序,但最常用的是“单程序、多数据(SPMD)”编程形式:所有处理器执行同一程序但作用于一套数据的不同部分,这就需要对数据和对其进行的操作的占用进行分配. 数据分配必须使得各处理器的工作负载很好地平衡,并且使通信和同步控制量最小.

分布式内存编程模式经常比共享内存编程模式难. 程序员必须知道在局部内存中的数据

位置,并且,当需要时,直接地移动或分配这些数据. 有关数据分配和所有必要的通信的指令必须显式地写在程序中. 一个串行程序常常需要做重大改造才能使之并行.

分布式内存程序用常规语言(Fortran, C,C + +, …)写成,用一个通信库来进行通信和同步操作. 基本通信指令允许消息在任意处理结点间发送和接收. 送来的消息通常在目标结点由操作系统缓存,直到应用程序需要这条消息. 也有各类"较高级"指令,如对分布于结点间的一组数据作全局运算(求和和求最大值)和同步控制.

除依赖于机器的通信库外,还发展了一些独立于机器的通信库. 广泛应用的系统有PVM[2], MPI[3]和 PARMACS[4]. 这些库或环境中有的(如 PARMACS)允许自动分裂和并合向量和矩阵以及对性能进行监控和分析. PVM 环境允许用网络把异构工作站连起来当做并行机使用,如同分布式内存并行计算机那样.

对 DM-MIMD(分布式内存 MIMD 系统)机器进行(半)自动并行的编译器和软件工具现在已经存在. 高性能 Fortran (HPF)是 Fortran 90 用于编写并行应用软件的扩展[5]. HPF 包括的特征有,分配多维数组(即结构数据块)至并行处理器和指定数据并行操作. 目前正在扩展 HPF 使得对更复杂的数据结构如多分区网格具有类似功能[6]. FORGE 90 是一个用于对已有的串行程序进行分析和(半)自动并行化的软件工具,其基础是用户定义的数组分配, FORGE 允许以交互方式或自动选择循环来进行并行化[7].

1.3 几种并行系统

Intel Paragon Intel Paragon 是一种分布式内存系统,它的处理结点以二维网格相互连接,见图 2. 一个有 1874 个处理器的 Paragon 系统在 Sandia 国家实验室已投入运行. 有两种处理结点可以使用,它们都是基于 Intel i860 处理器. 一般性任务的结点有两个处理器(一个用于计算,一个用于通信)和一个输入/输出(I/O)扩充口. 多处理器结点有两个计算处理器(带共享内存)和一个通信处理器. 通过网络传送的 wormhole 消息,是由每个结点之一的网格发送程序芯片来完成的. 处理结点在逻辑上分为计算部分(并行程序执行)、I/O 部分(用于 I/O、连网时磁盘存取的结点)和服务部分(人机对话、编译).

分布式操作系统提供单一的操作系统环境(单一的进程标识符空间,单一的文件系统,等等)和对作业进行自动调度. 它采用 Intel's NX 通信系统或 SUNMOS 环境(Sandia)来支持分布式内存编程模式. 并行开发环境包括各种软件开发和性能监控工具. 更多的信息见[8].

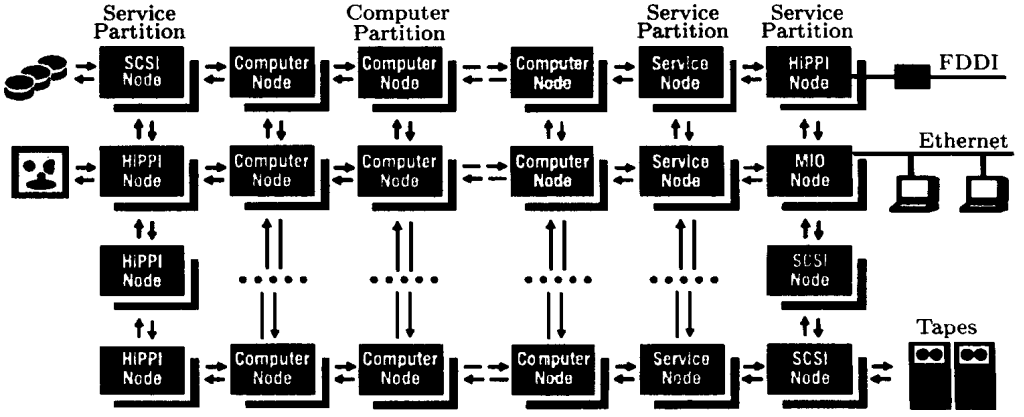


图 2 Intel Paragon 结构,其基础是二维互连的网络

Cray T3D Cray T3D 是有 32 至 2048 个处理结点的分布式内存并行系统. 处理结点(DEC

Alpha 处理器)由一个双向三维环面(周期网格)网络来连接(网络的每个开关由两个结点共享),见图 3. 其中使用了各种技巧来降低互连网络的通信开销和进行处理结点的同步控制. 从物理结构上看内存是分布式的,但它却是全局可寻址的. 因此,它支持三种编程模式: SIMD(数据并行),共享内存 MIMD 和分布式内存 MIMD 编程模式. 软件环境包括一个具有 Fortran 90 特征(数组句法,等等)的 Fortran 编译器,允许用户在一个程序中混合使用三种编程模式. 还包括 PVM,一个性能分析器,等等. T3D 系统需要一个 Cray 向量处理器作为主机系统.

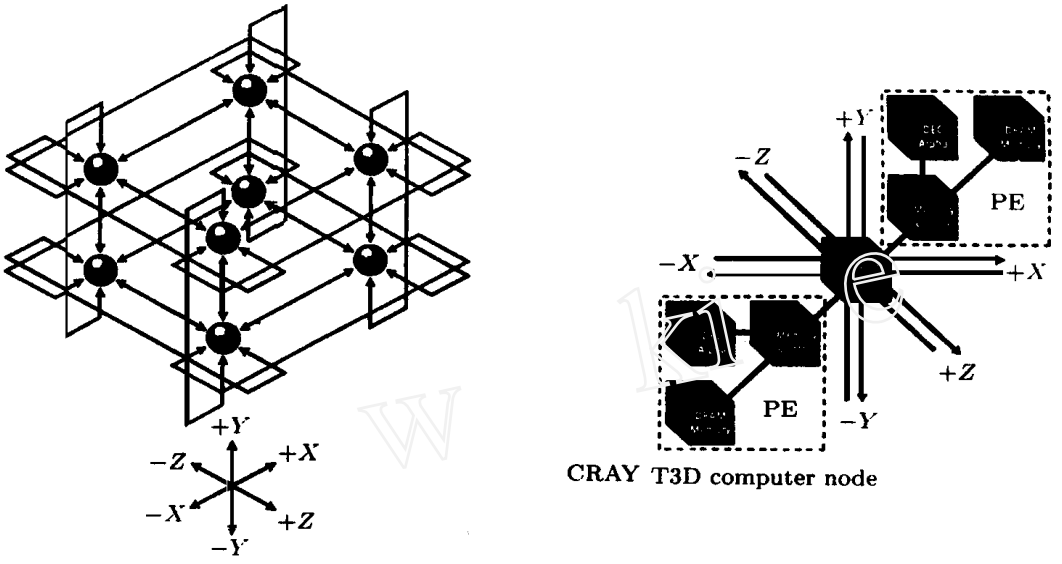


图 3 Cray T3D 结构(三维互连网络)

IBM SP2. IBM SP2 是具有 128 个处理结点的分布式内存系统. 有两类处理结点:"窄结点"和"宽结点",都是基于 POWER2 处理器的. 宽结点允许有更大的内存,提供更快的处理器与内存之间的连接和允许挂接各种存储设备. 结点由"高性能开关"相互连接,见图 4. 开关是一个进行 wormhole 路径选择的多步 Ω 形网络. 开关的可用通信带宽随处理器数目线性增长. 提供对低延迟和最小附加开销的短消息支持. 更多的信息见[9].

AIX 并行环境包括消息传递库(MPL)、性能监控和可视化工具. PVM 的优化版本也有了. 只支持分布式内存编程模式,作业调度支持由"Loadleveler"软件提供.

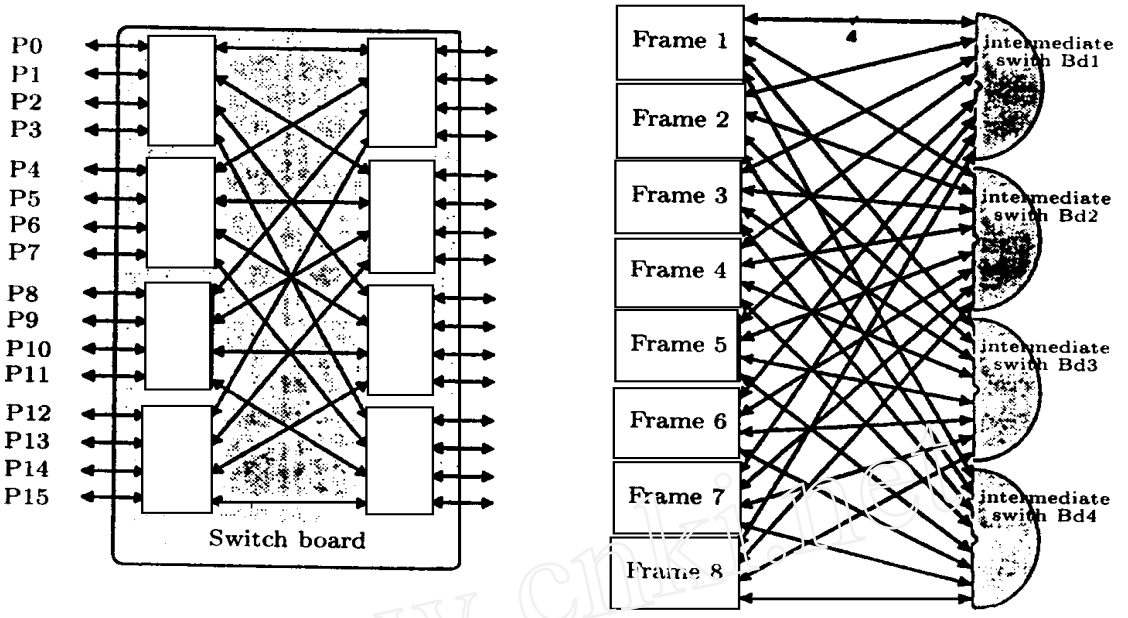
Silicon Graphics Power Challenge SGI Power Challenge 系统是共享内存多处理器,最多具有 18 个处理器(MIPS R8000),带有多个能同时运行的功能部件. 每个处理器上的高速缓存系统由一个芯片内的小的高速缓存和一个芯片外的大的流水线结构 高速缓存构成. 主存系统最多可达 8 路交替方式.

Fortran 和 C 编译器能够重组程序,在嵌套循环的情况下可通过互换循环、"覆盖"或"分块"等等来减少错过高速缓存中的数据(循环分块是优化内存等级性能的一种技术,如用于矩阵运算). 另外,编译器支持 Fortran 和 C 程序自动的或通过用户设定(通过指导语句)的并行控制. 更多的信息见[10].

(基于开关的)通信网络能连接多达 8 个 Power Challenge 系统,形成一个"CHALLENGE 阵列"系统. 通过这个网络的通信必须以分布式内存方式编程,使用消息传递库(PVM,MPI)或高性能 Fortran.

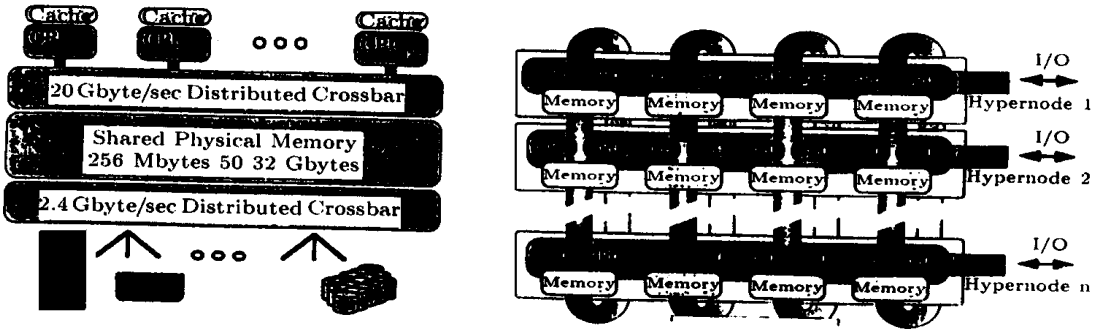
Convex Exemplar Convex Exemplar 由一些超结点组成,通过一个具有四路交替连线的低延迟环形网络来相互连接. 每个超结点是一个共享内存多处理器,含 8 个处理器(HP PA-RISC

7200)，由一个交叉开关与 4 个内存模块相连，见图 5.



(a) 16 结点双向多级网络，形成 IBM SP2 中高性能开关的基本组织　　(b) 用 12 个开关来互连 128 个处理器

图 4



(a) 逻辑系统　　(b) 物理系统

图 5　Convex Exemplar 结构

Exemplar 编程环境对共享内存和分布式内存编程都提供支持. PVM 通信库用于消息传递. 内存共享编程环境是通过所谓“全局共享的分布式虚拟内存”实现的. 以共享内存方式编写的一个应用程序可使用位于不同超结点处的处理器. 在这种情况下，用到了内存等级中的三个级：某个处理器上的高速缓存，处理器所属的超结点的全局内存和位于不同超结点处的内存.

访问位于不同超结点处的数据所需时间比访问同一个超结点内的数据所花的时间要长. 为了减少由于使用环形网络引起的延迟，每个超结点包括一个记录通过网络进行的内存访问的高速缓存区. 该高速缓存区保存的信息能用于直接获取已保存在高速缓存区中的任何全局数据. 系统自动维护不同超结点间高速缓存区的一致性.

1.4　并行性能参数

并行实现的质量经常用得到的 加速比 或 效率 来衡量.

一种并行算法在 P 个处理器上运行所得的 并行加速比 由这个并行算法在单个处理器上运行时间与在 P 个处理器上运行时间之比来定义. 并行效率等于加速比除以 P. 由此我们得到如下并行加速比 $S(n,P)$ 和并行效率 $E(n,P)$ 的定义:

$$S(n,P) = \frac{T(n,1)}{T(n,P)}, E(n,P) = \frac{S(n,P)}{P} = \frac{T(n,1)}{PT(n,P)} \quad (1)$$

式中 n 表示问题的尺度, $T(n,1)$ 和 $T(n,P)$ 分别指算法在一个和 P 个处理器上的运行时间.

注意式(1)没有给出任何有关并行算法质量的信息. 它只衡量一个算法并行化的好坏程度. 因此,还得补充衡量并行算法 数值效率 的数据. 该数值效率可以定义成单个处理器运行时间的以下比率: $T_{best}(n)/T(n,1)$. 此处, $T_{best}(n)$ 指由并行计算机的一个处理器运行已知速度最快的串行算法所需时间. 并行加速比(或效率)与数值效率相结合得到了 绝对加速比 和 绝对效率 的概念,定义如下:

$$\bar{S}(n,P) = \frac{T_{best}(n)}{T(n,P)}, \bar{E}(n,P) = \frac{\bar{S}(n,P)}{P} = \frac{T_{best}(n)}{PT(n,P)} \quad (2)$$

实际考虑限制了定义(2)的使用. 首先是难以确定哪个算法是最好的串行算法;这可能依赖于问题尺度 n,所使用的硬件,程序编写方式等. 另外当可以得到更好的算法时,"最好"的算法这个概念便随时改变. 而且,实现该算法的好的程序也不一定总是有. 实际中人们可以用一个好的具有好结果的算法运行的时间来代替 $T_{best}(n)$,以定义绝对加速比.

如果我们假定一个有 P 个处理器的机器运行速度不能超过一个单处理器机器的 P 倍,则显然有 $S(n,P) \leq P$ 和 $E(n,P) \leq 100\%$. 现在列举一些可能引起偏离线性加速比关系的开销.

® **串行部分** 一台并行计算机能得到的加速比可能因存在一小部分固有的不能并行化的串行部分而被大大地限制. Amdahl 定律表述了这个问题[11]:

令 α 为在计算中必须串行操作的部分所占的比例, $0 \leq \alpha \leq 1$. 一台有 P 个处理器的并行计算机所能得到的最大加速比如下:

$$S(n,P) \leq \frac{1}{\alpha + (1-\alpha)/P} \leq \frac{1}{\alpha} \quad (3)$$

例如,当 10 %的程序必须按串行模式运行时,最大加速比被限制在 10 以内,而与有效处理器数目无关.

Amdahl 定律成为人们怀疑大规模并行系统用处的中心论题. 他们的批评在只考虑解一定尺度的指定问题(即 α 为常值)时是有道理的. 但实际上,很少有这种情况,因为问题的尺度趋于与处理器数目和计算机的性能成比例(大规模并行系统解的问题比小规模系统大).

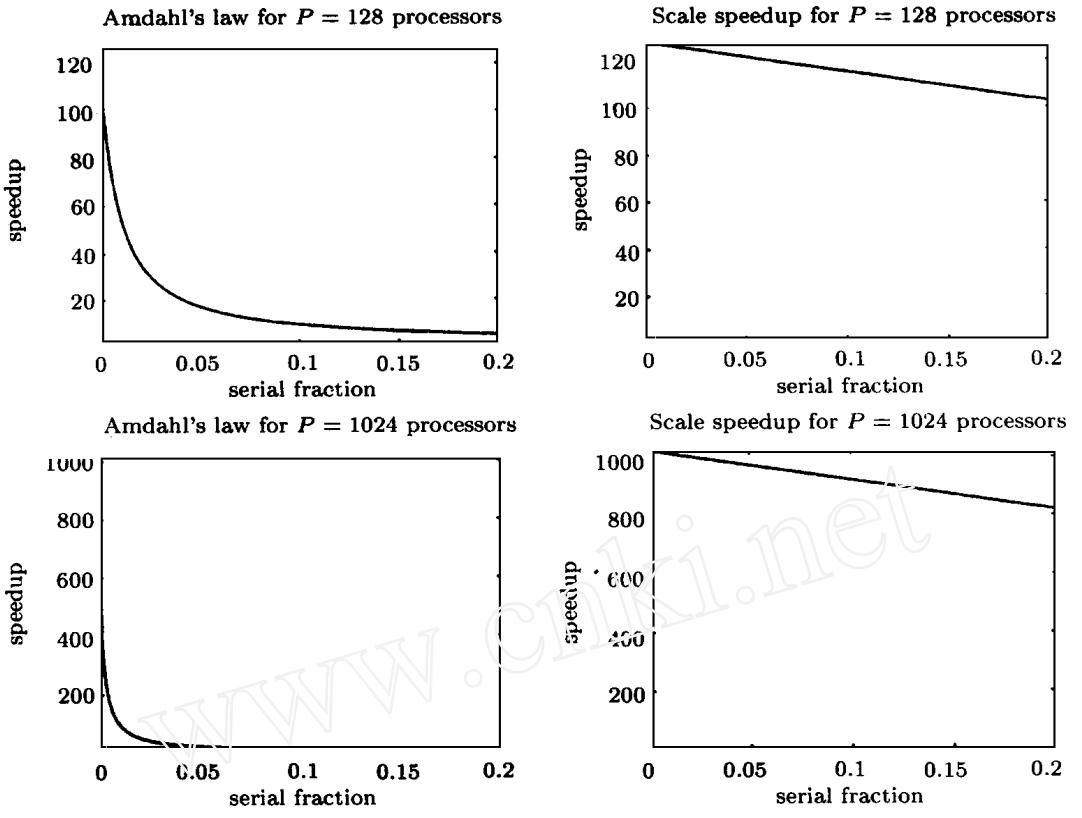
对于许多计算问题,当问题增大时,串行部分 α 很快趋于 0. 因此,当问题足够大时, α 依赖于处理器数目,式(3)的重要性减弱了. Gustafson 人用另一法则代替了 Amdahl 定律[12,13]:

令 $\bar{\alpha}$ 等于一个有 P 个处理器的并行系统在计算期间内的串行部分. 于是能得到的最大加速比如下:

$$S'(n,P) \leq P(1-\bar{\alpha}) + \bar{\alpha} \quad (4)$$

$S'(n,P)$ 通常叫做 比例加速比. 它等于 $T'(n,1)$ 对 $T'(n,P)$ 的比值, $T'(n,1)$ 是并行程序在单个处理器上要运行的时间,只要这个处理器有足够的有效资源(内存).

在大规模应用中, $\bar{\alpha}$ 值经常很小,在大规模并行处理器上可以得到非常高的比例加速比. 图 6 表示了 $S(n,P)$ 和 $S'(n,P)$ 对串行分数 α 和 $\bar{\alpha}$ 的依赖关系.



(a) 并行加速比 $S(n, P)$　　(b) 比例加速比 $S'(n, P)$

图 6　$P = 128$ 和 $P = 1024$ 时加速比与串行部分的关系

® **非最佳算法和算法开销**　最好的串行算法经常很难或不可能并行化(如解三对角线性系统的托马斯算法). 在这种情况下并行算法可能比串行算法需要更大的计算量. 另外,为了避免通信所带来的开销,人们可能希望在不同处理器上重复一些计算,而不是让一个处理器计算后分配结果(如‘双通量计算’,见后).

® **软件开销**　并行控制经常引起(有关的)软件开销增加,如索引、程序调用等的开销. 这种方法也常引起循环变短,因而限制了向量长度. 这降低了向量计算的使用潜力.

® **负载不平衡**　并行算法的运行时间由负担最多的处理器的运行时间决定. 当计算负载不是均匀分布时,就会引起负载不平衡,产生处理器空转:处理器必须等待别的处理器完成特定计算.

® **通信和同步控制开销**　最后,任何用于通信和同步控制的时间都是纯开销.

在下一节中,我们将详细讨论这些不同来源的开销.

2　使用网格问题的并行处理

2.1　引言

在本文的其余部分,由于两个原因,我们将着重讨论 分布式内存并行化问题. 第一,并行系统中只有分布式内存系统有“最高”并行结构. 第二,在分布式内存编程模式中,并行控制语句必须在应用程序中明确给出. 为分布式内存系统设计的算法也能在共享内存(或混合)系统中很好地运行. 数据分配对分布式内存系统是必要的,对共享内存系统也是有益的. 例如,矩阵通

常不能全部存入高速缓存中.通过将矩阵运算分解成对子矩阵的运算,并选择适应高速缓存操作的子矩阵大小,就可以优化内存等级的性能.

如今,并行算法设计的基本问题已经被很好地理解了,并且已在不少书籍和论文中作了介绍. G. Fox 等人的书[14]是一本重要的参考书(尽管有点过时). E. Vande Velde[15]和 C. De Moura[16]的书也可作为很好的入门材料.一年一度的并行计算流体力学会议的文集提供了这个领域内的研究和成果的综述[17~19].另外,可扩展高性能计算会议[20,21], SIAM 科学计算并行处理会议[22,23]和高性能计算与网络会议[24,25]等均为有价值的信息来源.

对于使用网格的问题,例如偏微分方程的数值解,数据被定义在离散点(网格点、有限体积或有限元)上.本文即用网格点来总称网格点、有限体积和有限元及其数据.

带网格的应用问题的并行化因每个网格点上的计算一般只涉及几何相邻的网格点而变得很容易.通过将网格分裂成子区域(子网格)和分配这些子区域到并行系统的处理器,就能达到并行计算的目的.每个处理器对分配到该处理器的(单个或几个)子区域进行有关计算.子区域之间的沟通(和通信)只取决于它们的周界线.

在带网格问题的并行计算及并行算法的性能分析中所涉及的许多重要概念可以通过研究"模型问题"的并行计算来理解,即用显式时间积分来求解结构网格上偏微分方程的有限差分或有限体积离散.

假定一个二维结构网格分裂成大小相等的子区域,使得每个处理器处理 $n_x \times n_y$ 个网格点或单元.再假定显式时间积分基于五点格式,即

$$v_{i,j}^{(k+1)} = f(u_{i-1,j}^{(k)}, u_{i+1,j}^{(k)}, v_{i,j-1}^{(k)}, u_{i,j+1}^{(k)})$$

由于这个计算的局部化特性,每个处理器能够对所有的内网格点(图 7 的白色区域)进行计算.其他的子区域网格点称为(子区域)"边界网格点".为了计算子区域边界网格点上的值,处理器还必须知道子区域边界另一边的网格点处的函数值 $u_{ij}^{(k)}$.这个数值必须从相邻处理器中取得并存储在位于图 7 的覆盖区中.另一方面,边界网格点上的值必须送到相邻区域中位于覆盖区的点上.因此在每步积分之前,相邻的处理器互相交换消息,见图 8.

2.2 通信开销的分析

对上述简单模型问题,假设问题尺度为 n,算法在一个有 P 个处理器的并行系统上的运行时间可写为

$$T(n, P) = T_{\text{calc}} + T_{\text{comm}}$$

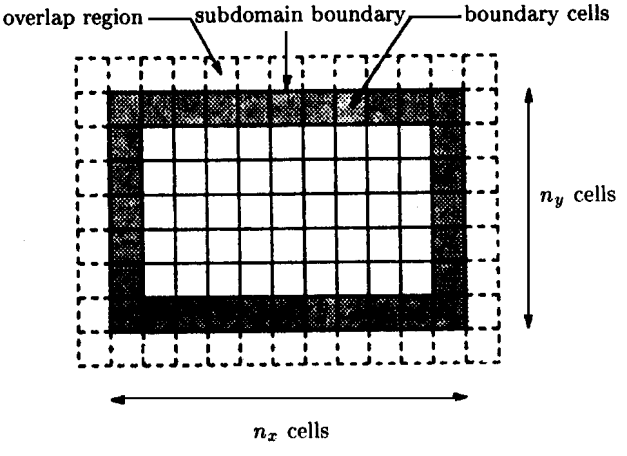


图 7 网格分裂和通信

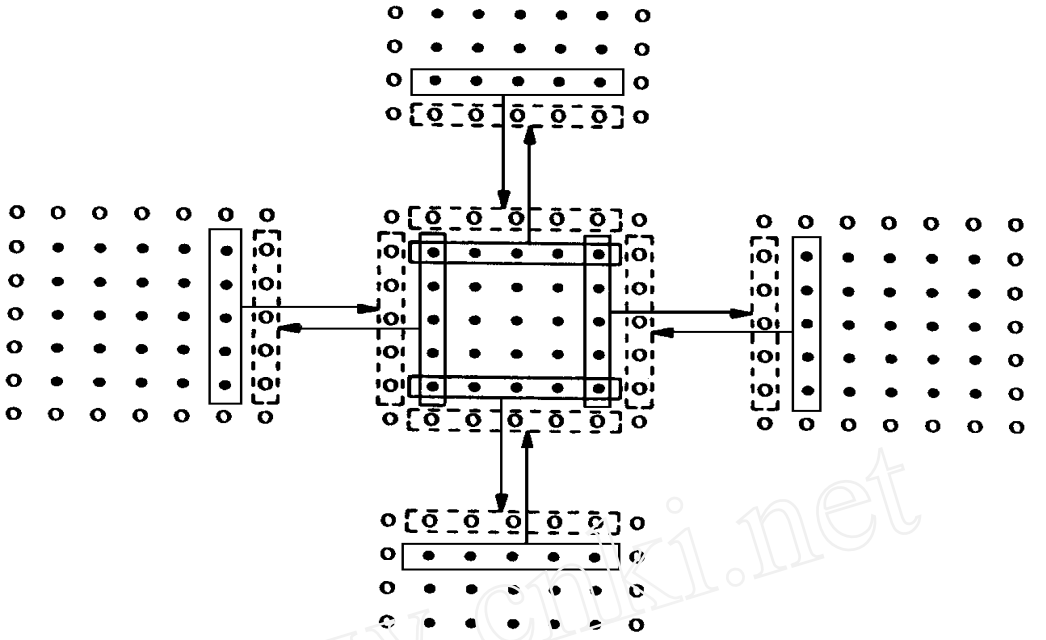


图 8　局部边界通信

T_{calc}指计算时间，T_{comm}指通信时间. 假定除覆盖区的通信外没有其他开销，则串行运行模式所用的计算时间是

$$T(n,1) = P \cdot T_{calc}$$

因而加速比和并行效率由下式给出：

$$S(n,P) = \frac{PT_{calc}}{T_{calc} + T_{comm}} = \frac{P}{1 + T_{comm}/T_{calc}} = \frac{P}{1 + f_c}$$

$$E(n,P) = \frac{1}{1 + T_{comm}/T_{calc}} = \frac{1}{1 + f_c}$$

式中 $f_c = T_{comm}/T_{calc}$为通信开销. 如果 f_c 很小，则可得到接近最优的加速比（$S(n,P) \simeq P$）和效率（$E(n,P) \simeq 1$）.

每个处理器发送和接收数据的数量与边界单元数成比例，而每个处理器所作的计算量则与内单元数成比例. 对于该模型问题，我们有

$$T_{calc} = c_1 n_x n_y t_{calc}$$

$$T_{comm} = c_2 \cdot 2(n_x + n_y) t_{comm}$$

式中 t_{calc}代表一次浮点运算所需时间，t_{comm}指通信一个浮点数所需时间，c_1，c_2 为常数. 由此得到一个重要公式

$$f_c = \frac{c_2}{c_1} \frac{2(n_x + n_y)}{n_x \times n_y} \frac{t_{comm}}{t_{calc}} \tag{5}$$

从这个公式可以看出，通信开销依赖于 3 个因素：

(1) 几何特征 $2(n_x + n_y)/n_x n_y$：衡量“周界线对表面积”的比率，比率越小 f_c 的值越小；

(2) 机器特性 t_{comm}/t_{calc}：衡量通信速度相对于浮点运算速度的快慢程度；

(3)算法特征 c_2/c_1:如果相对于每个网格点通信运算量(由 c_2 表示)而言,浮点运算量很大(c_1 大),则这类问题的开销 f_c 小.

注　大多数通信系统有个重要特性,即它们具有相当高的消息启动时间.相邻处理器间的一条消息发送开销可写成

$$T(n) = t_{\text{start}} + nt_{\text{send}}$$

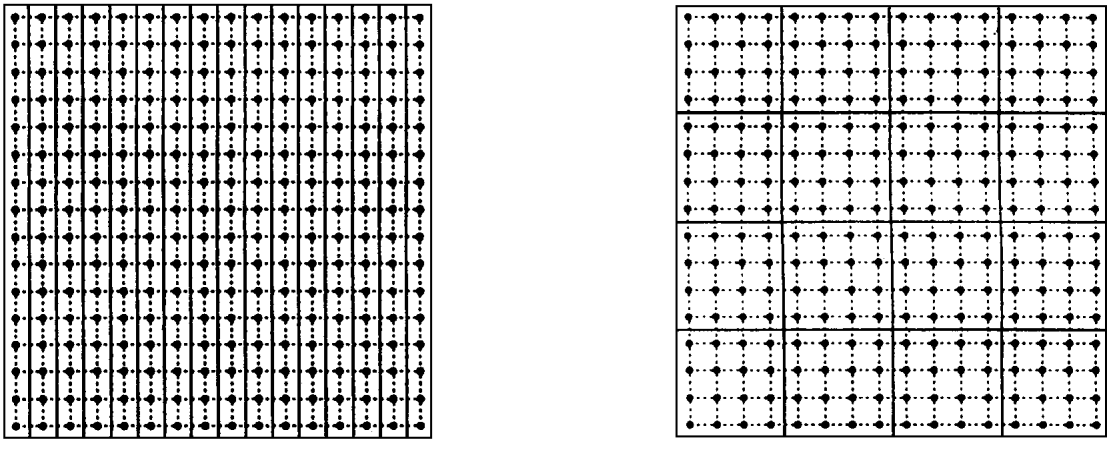
式中 n 指消息长度(传递的字数),t_{start}是消息启动时间(由软硬件延迟引起),t_{send}是每个字的传送时间.许多系统的 t_{start}大大超过 t_{send}(甚至 1000 倍).一个马上可以得出的结论是:应尽可能避免发送许多短消息.

在式(5)中 t_{comm}指一个字通信的平均时间.它明显地大大依赖于发送消息的平均长度:对于短消息有 $t_{\text{comm}} \simeq t_{\text{start}}$,而对于非常长的消息有 $t_{\text{comm}} \simeq t_{\text{send}}$.当用式(5)分析并行算法时必须考虑到这一点.

这个模型问题的进一步分析揭示了并行 CFD 算法必须考虑的一些重要指导性原则.

不同网格划分方法　对于这个模型问题,通信量与(内)子区域边界的网格点数成比例,即与子区域"周界线"长度成比例.当子区域大小一定时,如果每个方向网格点数相等,即 $n_x = n_y$,则周界线长度(也就是通信量)最小,我们将这样的子区域称为"正方形子区域".因此,划分成正方形子区域可得到最小通信量.

由此可以总结出以下规律.条状(或一维)划分(图 9(a))使子区域产生长边界,但最多与两个子区域相邻.块状(或二维)划分(图 9(b))使子区域的边界较短,但却有多达 4 个相邻子区域.因而块状划分使总通信量最小,而条状划分使消息数最少.至于最好的选择则依赖于问题和并行计算机的特性.当消息启动时间在每条消息通信时间占主导地位时,条状划分是有利的.



(a)　(b)

图 9　网格的条状和块状划分

注意,各个方向所需通信不总是"等量"的,但它们可能依赖于问题特性或数值算法.这些可能会影响区域划分方式的选择.例如,考虑环绕机翼的可压 Navier-Stokes 方程的求解.使用代数湍流模式会导致对垂直于机翼的那个方向的全局性的依赖(和通信),因而适合使用条状划分.

对子区域大小的依赖　当每个子区域包括 $N = n_x \times n_y$ 个网格点,并且用正方形子区域

($n_x = n_y$)进行块状划分时,由方程(5)可得

$$f_c \propto \frac{1}{\sqrt{N}} \tag{6}$$

这表明只要子区域大小恒定,通信开销 f_c 便保持恒定,而与处理器数目无关!当然这意味着当处理器数目 P 增大时,为了保持并行效率不变,总问题尺度 M 也增大,因为 $M = N \times P$. 关系式 $f_c \propto \sqrt{P}/\sqrt{M}$ 也指明,对于给定的(总)问题尺度 M,效率和加速比随处理器数目的增加而降低(参见 Amdahl 定律). 这个分析只当通信只是相邻处理器之间的消息交换时有效. 任何"全局通信"(例如,将局部残值加起来计算总残值)意味着随着处理器数目增加而增长的开销. 然而,这种全局通信的相对重要性经常很低,并不真正影响总加速比和效率.

扩展至多点格式和三维网格 上述分析在用其他计算格式代替五点格式时仍有效[14]. 这可能有必要用较大的覆盖区(如用 2 点宽度). 在这种情况下通信量(和常数 c_2)增加,但每个网格单元上的运算数量(从而常数 c_1)也增加. 因此,通信开销不一定提高.

在三维网格情况下,可使用一维,二维和三维划分. 于是,通信量由子区域"表面积对体积"的比率决定,方程(6)中的分母相应改为 $N^{1/3}$,见[14].

对机器特性的依赖 一个给定算法的加速比和并行效率与机器特性 $t_{\mathrm{comm}}/t_{\mathrm{calc}}$ 成比例. 现有的不同并行系统所具有的这个特性值很不相同. 因而,不同机器的通信开销根本不同.

注意,计算机生产厂家也许对他们系统里的处理器和通信网络交替升级. 处理器升级而不同时提高通信速度,会产生很大比率 $t_{\mathrm{comm}}/t_{\mathrm{calc}}$ 的"不平衡"系统.

对问题特性的依赖 计算流体力学应用的特性是,在每次迭代中每个网格点或网格单元上有大量浮点运算,而每个点上却只定义少量的(需进行交换的)变量. 所以在通信开销中的因子 c_2/c_1 将很小.

结果是,通信开销达到最小并不总是大大影响加速比和并行效率. 然而,使负载不平衡减到最小总是重要的. 下面的研究表明块状划分一般也使负载不平衡最小. 因此在许多情况下通信开销最小和负载不平衡最小是相辅相成的.

2.3 负载不平衡分析

令第 i 个处理器的计算时间为 T_i^{calc}, $i = 1, \ldots, P$,并且令 $T_{\mathrm{aver}}^{\mathrm{calc}}$ 和 $T_{\mathrm{max}}^{\mathrm{calc}}$ 分别为 P 个处理器上的平均和最大计算时间. 我们定义 负载平衡因子 为

$$\lambda(n, p) = \frac{T_{\mathrm{aver}}^{\mathrm{calc}}}{T_{\mathrm{max}}^{\mathrm{calc}}}$$

如果运算数量(按先后顺序计算)不依赖于处理器的数目,并且通信时间和处理器空转时间可以忽略,则并行程序的计算时间可由处理器最大计算时间决定. 在这种情况下并行效率由下式给出:

$$E(n, p) = \frac{T(n, 1)}{PT(n, P)} \approx \frac{\sum_{i=1}^{P} T_i^{\mathrm{calc}}}{P\max(T_i^{\mathrm{calc}})} = \frac{T_{\mathrm{aver}}^{\mathrm{calc}}}{T_{\mathrm{max}}^{\mathrm{calc}}}$$

从而有 $E(n, p) \approx \lambda(n, p)$.

注意,最容易犯的错误是,用子区域的最大计算时间和最小计算时间的比来测量负载(不)平衡.

一种常见的情形是,每个网格点上的计算量为常数,而处理器由于更新覆盖区中的数据而在每步迭代后(隐含地)被同步. 如果通信时间可以忽略,则有

$$E(n, p) = \frac{T(n, 1)}{PT(n, P)} \approx \frac{M}{PN_{\max}} = \frac{N_{\mathrm{aver}}}{N_{\max}}$$

式中 M 为网格点总数, N_{max}为子区域的最大网格点数, $N_{aver} = M/P$.

假定一个矩形网格在 P 个处理器中分配. 如果网格不能在处理器中相等地分配,则块状划分将引起比条状划分更大的负载平衡因子. 划分成正方形子区域将导致最大的负载平衡因子,即最小的负载不平衡.

边界条件处理也是负载不平衡的可能来源. 实际上, 一般而言,对边界单元作的计算与对内单元作的不同. 为了使边界条件处理引起的负载不平衡最小,边界单元必须尽可能相等地分配到处理器中. 这在子区域(近似)为正方形时能达到.

每个网格点或网格单元的计算量为常值这个假设在 CFD 中不总是有效的. 例如,位于亚音速区和超音速区的网格单元上的计算量就不同. 这会导致不能事先准确预知负载不平衡. 当在区域的不同部分的数学模型不同时,例如考虑到高温区的化学反应时 , 会产生相似的问题.

2.4 并行算法的数值效率

到现在,我们通过比较并行运算时间与 相同算法 在一个处理器上所需时间,讨论了并行开销如何影响计算性能.

在许多情况下用于并行机器的算法不同于用于串行机器的典型算法. 为了得到可接受的并行效率,常对串行算法作修改,以减少通信或同步点数量. 这可能使算法的 数值 效率恶化. 当串行算法不能简单和有效地并行化时,甚至有必要使用数值特性(运算数量、收敛性. . . . .)很不相同的算法.

显式格式 如果执行了所有必要的通信, 则显式格式是内在并行的而且并行(网格划分)不影响其数值特性. 例如, Runge- Kutta 法每一分步后需要进行通信. 为了减少通信开销,可以只在完全时间积分步后对覆盖区更新数据. 省略一些通信会得到稍微差些的收敛性,但能导致更高的"总加速比". 这个影响主要由问题的性质决定. 注意,这种技术导致了"结构分块"方法,但这里分块的数目是由处理器数目决定的,而不是由区域的几何特性决定.

隐式格式 当使用隐式方法时情况较复杂.

® 假定用点松弛格式解生成的线性系统. 雅可比松弛 是内在并行的. 在这种情况下通信需要与上面的模型问题(覆盖区通信)完全一样. 高斯-赛得尔松弛通常有更好的收敛性. 在串行计算机上,典型的高斯-赛得尔迭代按自然顺序扫描网格单元. 在向量或并行计算机上,网格点需按 红-黑交替顺序排列. 先对所有"红"点并行运算, 然后计算"黑"点. 自然(字典式)顺序的高斯-赛得尔法和红-黑顺序的高斯-赛得尔法在收敛速度上是完全不同的. 下一节将举例说明.

® 当使用 线松弛 格式时, 必须求解(分块)三对角系统. 这样便有同一网格线上网格点之间的相互依赖. 如果只对一个方向扫描, 那么三对角系统(以及相应的数据依赖)只定义在这个方向上. 通过使用条状划分, 可保证每个三对角系统只属于一个处理器. 于是每个系统仍可用最佳串行算法托马斯算法(即高斯消去法)求解. 如果使用块状划分, 或者对不同方向使用线松弛, 则线松弛的并行化处理不会如此容易. 于是(某些)三对角系统被分布于处理结点之中. 基于子结构化的高斯消去法和/或循环约简, 已经研制出了对(分块)三对角系统的并行解法[26,27]. 然而,这些解法的运算量比托马斯算法高约 1 倍,从而它们的数值效率很低,并且它们也包含串行部分. 因为必须解许多三对角系统,所以后者的缺陷可以通过平均分配串行部分于各处理器中来避免(以一些通信为代价). 人们也试图通过构造近似解法来减少并行算法的计算时间[28,29].

作为求解大量三对角系统的一种替代方法,可 以流水线方式使用托马斯算法[28]. 因为并行三对角系统解法与托马斯算法相比有低"数值效率",但以通信为代价,也许值得用托马斯算法. 然而这种方法导致许多短消息通信和(流水线进程的开始和结束时的)负载不平衡. 另一种

替代方法是用如下的方式求解双向三对角系统. 我们知道如果对数据采用条状划分,则对其中的一个方向三对角系统可用托马斯算法求解. 如果人们将不同的条状划分用于算法的两个阶段,使得在每一阶段任一三对角系统只存于一个处理器中,则在两个方向都可用托马斯算法. 这要求两个阶段之间要进行"数据转置". 于是两个阶段数据转置引起的通信量与每个处理器含有的网格点数量成比例. 因为计算所消耗的时间与上面有同样的性质,所以并行效率还是可以接受的. 在具有不规则边界的有限差分网格上实现半隐式 ADI 时间积分,以总效率来衡量,上面的第二个方法是最有效的[30].

® 数值处理和并行处理之间相互作用的另一个例子可在多重网格中找到. 用达到收敛所需的工作量来衡量,W-循环通常比 V-循环更有效. 然而在并行计算机中,W-循环导致差的并行效率,所以人们常常采用 V-循环,尽管后者数值特性较差[31].

® 解偏微分方程的并行算法也可以数学意义上的区域分解为基础. 有两种可能方法.

在 Schwartz 区域分解方法中使用覆盖子区域. 在子区域边界上用近似解,分别在每个子区域上解微分方程. 产生的近似解提供了解相邻(覆盖)子区域边界的新的近似值. 必须用迭代方式重复这个进程.

在 Schur 补充方法中使用非覆盖子区域. 子区域问题按定义在子区域边界上的变量求解. 通过求解交界面问题或"Schur complement"问题计算出边界上的变量后,就能确定子区域里面的变量.

注意,这两种区域分解方法与不用分解相比,都需要做额外计算. 这些额外计算必须看作由并行引起的算法开销. 计算流体力学问题的区域分解技术在文献[32～35]中作了介绍.

3 举例

在这一节里,我们举例说明了上面几节介绍的一些概念. 首先讨论了欧拉方程的一种显式算法在 Intel iPSC/2 和 iPSC/860 分布式内存计算机上的并行性能. 除了证明由网格划分引起的"二重通量计算"并行程序中会引起重要的算法开销外,还评论了用来测量并行性能的不同方法并定义了功效以作为测量性能的另一指标.

接着讨论了欧拉方程的一个隐式解法的并行性能. 除讨论能达到的并行加速比和并行效率外,还说明并行数值效率怎样影响总加速比和总效率,后者是实际性能的最好测量.

最后介绍了用分块结构网格求解欧拉方程的结果. 在这里使用了导致新的网格块产生的分块自适应网格加密. 另外还证明了使用启发式分配法可以将结构块有效地分配到处理器中,使负载不平衡和通信开销较小.

3.1 欧拉方程显式格式的并行性能

首先描述一些显式格式的分块并行试验[36～38]. 使用以下格式:

格式 1 采用 Van Leer 向量通量划分的一阶迎风离散,用带当地时间步长的欧拉前差时间积分.

格式 2 二阶 Roe 格式,对特征变量用最小模数限制器,用五步 Runge-Kutta 时间积分.

在 Runge-Kutta 法的并行版本中,对每个时间步只作一次通信,即在五步 RK 方法的第一步之前. 格式 2 的计算时间对通信时间的比率比格式 1 的大得多.

算例为绕 NACA0012 翼型的跨音速流动. 来流条件为: 马赫数 $M = 0.80$,迎角为 1.25°,总温 $T_0 = 278\,\mathrm{K}$,总压 $P_0 = 150000\,\mathrm{Pa}$. 图 10 所示的结构化 C 网格(240 ×19 单元)可以分成 2,4,...,16 个相同大小的块(一维划分,正交于翼型),见图 10(a). 因此所有这些划分允许计算负载几乎完全平衡.

做了两套测试: 第一套用 $N = P$ 块,第二套用 $N = 16$ 块,与处理器数目 P 无关. 在第一种

替代方法是用如下的方式求解双向三对角系统. 我们知道如果对数据采用条状划分,则对其中的一个方向三对角系统可用托马斯算法求解. 如果人们将不同的条状划分用于算法的两个阶段,使得在每一阶段任一三对角系统只存于一个处理器中,则在两个方向都可用托马斯算法. 这要求两个阶段之间要进行"数据转置". 于是两个阶段数据转置引起的通信量与每个处理器含有的网格点数量成比例. 因为计算所消耗的时间与上面有同样的性质,所以并行效率还是可以接受的. 在具有不规则边界的有限差分网格上实现半隐式 ADI 时间积分,以总效率来衡量,上面的第二个方法是最有效的[30].

® 数值处理和并行处理之间相互作用的另一个例子可在多重网格中找到. 用达到收敛所需的工作量来衡量,W-循环通常比 V-循环更有效. 然而在并行计算机中,W-循环导致差的并行效率,所以人们常常采用 V-循环,尽管后者数值特性较差[31].

® 解偏微分方程的并行算法也可以数学意义上的 区域分解为基础. 有两种可能方法.

在 Schwartz 区域分解方法中使用覆盖子区域. 在子区域边界上用近似解,分别在每个子区域上解微分方程. 产生的近似解提供了解相邻(覆盖)子区域边界的新的近似值. 必须用迭代方式重复这个进程.

在 Schur 补充方法中使用非覆盖子区域. 子区域问题按定义在子区域边界上的变量求解. 通过求解交界面问题或"Schur complement"问题计算出边界上的变量后,就能确定子区域里面的变量.

注意,这两种区域分解方法与不用分解相比,都需要做 额外计算. 这些额外计算必须看作由并行引起的 算法开销. 计算流体力学问题的区域分解技术在文献[32～35]中作了介绍.

3 举例

在这一节里,我们举例说明了上面几节介绍的一些概念. 首先讨论了欧拉方程的一种显式算法在 Intel iPSC/2 和 iPSC/860 分布式内存计算机上的并行性能. 除了证明由网格划分引起的"二重通量计算"并行程序中会引起重要的算法开销外,还评论了用来测量并行性能的不同方法并定义了功效以作为测量性能的另一指标.

接着讨论了欧拉方程的一个隐式解法的并行性能. 除讨论能达到的并行加速比和并行效率外,还说明并行数值效率怎样影响总加速比和总效率,后者是实际性能的最好测量.

最后介绍了用分块结构网格求解欧拉方程的结果. 在这里使用了导致新的网格块产生的分块自适应网格加密. 另外还证明了使用启发式分配法可以将结构块有效地分配到处理器中,使负载不平衡和通信开销较小.

3.1 欧拉方程显式格式的并行性能

首先描述一些显式格式的分块并行试验[36～38]. 使用以下格式:

格式 1 采用 Van Leer 向量通量划分的一阶迎风离散,用带当地时间步长的欧拉前差时间积分.

格式 2 二阶 Roe 格式,对特征变量用最小模数限制器,用五步 Runge-Kutta 时间积分.

在 Runge-Kutta 法的并行版本中,对每个时间步只作一次通信,即在五步 RK 方法的第一步之前. 格式 2 的计算时间对通信时间的比率比格式 1 的大得多.

算例为绕 NACA0012 翼型的跨音速流动. 来流条件为: 马赫数 $M = 0.80$,迎角为 1.25°,总温 $T_0 = 278\,\mathrm{K}$,总压 $P_0 = 150000\,\mathrm{Pa}$. 图 10 所示的结构化 C 网格(240 ×19 单元)可以分成 2,4,...,16 个相同大小的块(一维划分,正交于翼型),见图 10(a). 因此所有这些划分允许计算负载几乎完全平衡.

做了两套测试: 第一套用 $N = P$ 块,第二套用 $N = 16$ 块,与处理器数目 P 无关. 在第一种