物质点并行算法研究

Studies on Parallel Algorithm of Material Point Method

（申请清华大学工学硕士学位论文）

培 养 单 位 ： 航天航空学院

学　　科 ： 力学

研 究 生 ： 张 衍 涛

指 导 教 师 ： 张 雄 教 授

二〇一一年六月

物质点并行算法研究

张衍涛

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容。

本人保证遵守上述规定。

（保密的论文在解密后遵守此规定）

作者签名：_____ 导师签名：_____

日　　期：_____ 日　　期：_____

摘 要

物质点法采用物质点与背景网格双重离散。与拉格朗日有限元法相比，它避免了因网格畸变而产生的数值困难；与欧拉法相比，它避免了因非线性对流项而产生的数值困难，且容易处理自由表面和材料界面，因此适合于求解流固耦合，以及涉及大变形的问题。然而，普通电脑的计算能力限制了可以研究的问题的规模。模拟大规模问题时，离散常常不够密致，模拟结果也常常失真。为了扩大计算规模，有必要研究物质点法的并行算法。同时，为了避免浪费计算资源，要求并行算法具有良好的负载平衡。

与所有的网格质点类方法如 PIC 方法相似，物质点法中的质点在网格内是自由移动的，相对欧拉格式和有限元格式而言更难以实现负载平衡。本文主要针对常用的两种并行计算模型 MPI 和 OpenMP，研究了物质点并行算法中的一些问题，并讨论了如何进一步改进负载平衡。

MPI 模型使用一个丰富的函数库实现进程间消息传递。它功能强大、灵活，但实现 MPI 并行的工作量比较大。本文基于静态分区思想利用 MPI 模型将三维物质点法软件 MPM3D（计算机软件著作权登记号 2009SRBJ4761）并行化，编制了并行物质点程序 MPM3D_MPI，并验证了其正确性与稳定性，极大地提高了物质点程序的计算规模。OpenMP 是基于线程的并行模型，主要通过指导性语句实现并行。使用 OpenMP 时，在背景网格更新阶段我们提出了网格交替更新法以避免数据竞争。该方法适用范围广、实施简单、易于实现负载平衡且不浪费内存。在此基础上本文利用 OpenMP 将 MPM3D 并行化，编制了 MPM3D_OMP。算例表明 MPM3D_OMP 具有良好的并行效率。

关键词：物质点法　　MPI　　OpenMP　　网格交替更新法　　并行算法

Abstract

Material Point Method discretizes material domains by both regular grid and particles. Compared with Lagrangian FEM, MPM avoided numerical difficulty induced by grid distortion which is common in FEM. And compared with Eulerian method, there is no numerical difficulty caused by nonlinear convective term, and it's easy for MPM to handle free surface and the material interface. As a result of these advantages, MPM is widely used in coupled fluid-solid problem and other dynamic problems that involve large deformation. However, the computing capablity of common PCs greatly restrains the scale of problem that can be solved. When solving large scale problems, the bodies are often discretized with insufficient particles and the results will definitely be very inaccurate. To enlarge the problem scale, parallelization of MPM is required. Meanwhile, load balance strategy also needs to be raised to avoid wasting computing resources.

Similar to all methods in which bodies are discretized by both grid and particles, such as PIC method, the particles of MPM are free to move anywhere within the grid. This feature will cause great difficulty in achieving load balance. This article addressed some issues in the parallel algorithm of MPM using MPI or OpenMP and discussed how to furthur improve load balance.

MPI provides a lot of funtions to pass messages between processes. It is powerful, flexible but requires a lot of work to implement. In this article, the 3D explicit material point method code, MPM3D, is parallelized employing static decomposition strategy and is named MPM3D_MPI. The article also verifies the correctness and stability of MPM3D_MPI. The parallel program greatly increases the scale of problems that can be solved.

OpenMP is a parallel model based on threads, which mainly uses

directives to realize parallelism. We proposed an alternated grid updating method to avoid data races when OpenMP is used in the stage of grid updating. This method can be widely used, easy to implement, easy to realize load balance and wastes no memory. Based on this method, MPM3D_OMP is developed. Some examples show that the efficiency of MPM3D_OMP is very good.

Keywords: Material Point Method MPI OpenMP
alternated grid updating method parallel algorithm

目 录

主要符号对照表

ADI	Alternating Direction Implicit
AMR	自适应网格(Adaptive Mesh Refinement)
CFD	计算流体动力学(Computational Fluid Dynamics)
FLIP	隐式流体质点方法(Fluid Implicit Particle)
HJC	Holmoquist-Johnson-Cook
HPF	高性能 Fortran(High Performance Fortran)
JWL	Jones-Wilkins-Lee
MPI	消息传递接口(Message Passing Interface)
MPM	物质点法(Material Point Method)
MUSL	Modified Update Stress Last
NPB	NAS Parallel Benchmark
NUMA	不均匀内存存取机(Non-Uniform Memory Access)
PIC	网格质点方法(Particle In Cell)
RCB	递归坐标二分法(Recursively Coordinate Bisection)
SMP	对称多处理机(Symmetric MultiProcessing)
SPH	光滑质点流体动力学(Smooth Particle Hydrodynamics)

第 1 章　引 言

1.1　课题背景

并行计算主要目的是提高求解速度，缩短计算时间，扩大计算规模等。从应用的角度来看，并行计算有着更深层次的意义。若某算法收敛，那么加密离散(如，缩小有限元中的单元尺寸，或者物质点法中的背景网格尺寸)会使计算结果更接近问题的解。但加密离散就意味着计算规模增大，在无法使用理论验证时，大规模并行计算是验证算法是否收敛的一个手段。现实世界是复杂的，单纯使用理论工具常常不足以研究这些问题，并行计算由于能够提供大规模的数值运算，为我们理解现实中的很多复杂数学模型提供了工具。

物质点法是无网格法的一种，它在求解大变形问题时具有有限元法无法比拟的优势。有限元法用于求解显示动力学问题时，其时间步长和单元尺寸线性相关。当单元畸变到一定程度，时间步长将小到无法继续求解。相比而言，物质点法的网格只起辅助作用，所有的物理信息由物质点携带，网格则只用来求解动量方程。每计算完一步后变形网格都会被抛弃，并在下一步计算中重构规则网格。因而每一步所使用的网格都是规则的，时间步长也相对稳定。由于物质点法的这个优势，它在动力学尤其是大变形问题模拟中有着广泛应用。

物质点法能够处理大变形，包括断裂、破碎等传统有限元难以模拟的问题，已经广泛地用于求解冲击、爆炸等问题，并用于模拟粒子材料、泡沫材料、土壤材料、混凝土材料等等。然而普通个人电脑的计算能力对于物质点法模拟的规模的限制阻止了我们对于更多问题的探索。以今天的计算机发展水平，普通 PC 能够计算的问题规模仅限于数百万物质点。对于三维物质点程序，在每个维度上也只能铺设 10^2 量级数量的物质点，能够模拟的问题的规模是很有限的。若需要研究大尺度问题，如模拟楼层在地震荷载、冲击荷载等作用下的倒塌过程，每个物质点所代表的现实尺度可达数米，其结果显然将严重失真，因此有必要研究物质点法的并行算法。

1.2　物质点法

物质点法最初起源于 PIC[1]方法。在等离子体模拟、流体模拟等领域里，PIC

方法有着广泛的应用。但 PIC 方法只是半拉格朗日格式，其质点只携带了质量与位置变量。为了减少 PIC 方法的数值弥散，Brackbill 等提出了 FLIP[2, 3]格式，并应用于流体的模拟中。FLIP 格式里的粒子携带着流体全部的物理量，易于求解使用历史变量的大变形问题，初具了物质点法的雏形。Sulsky 等将 FLIP 推广到固体问题，模拟了弹塑性体碰撞问题、弹丸高速碰撞成坑问题等，提出了物质点法(MPM)[4, 5]。物质点法继承了欧拉法和拉格朗日方法的双重优点，避免了欧拉法处理对流项的困难，以及拉格朗日单元在大变形情况下的畸变问题，而且该方法自动实现了无滑移接触算法，在计算穿透、爆炸、冲击等涉及大变形、破碎的问题中与有限元相比具有很大的优势。因此在随后的发展过程中，物质点法被广泛应用于多种多样的问题，如金属成形问题[6], 高速碰撞问题[5, 6, 7]，爆炸过程[8, 9, 10]等等。同时，由于应用的需要，物质点法的若干个子问题也得到了深入研究，比较重要的有对于物质点接触算法的研究[11-16]，对于自适应算法的研究[10]等等。

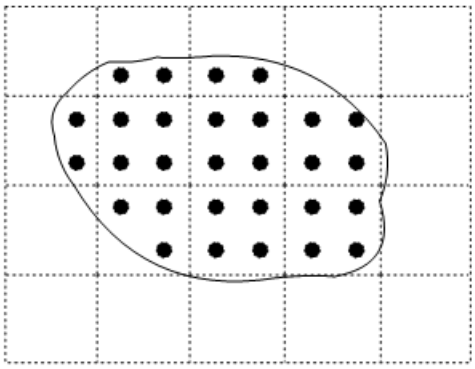


图 1.1　物质点法离散示意图

物质点法使用物质点与背景网格对物体进行双重描述，如图 1.1，虚线为背景网格，黑实点为离散得到的物质点，实曲线为物体的轮廓。物质点携带了物体的所有变量，如位置、质量、动量、能量和变形历史。在每一步的计算过程中，物质点先把自身携带的物理量通过形函数映射到背景网格上，在背景网格上求解控制方程，最后再把网格上的求解结果映射回物质点以更新物质点携带的变量。在下一步计算时将抛弃已经变形的网格，而使用重新构造的规则网格。

在下面的叙述中，上标 k、k+1 分别表示第 k 和 k+1 个时间步，下标 i 表示背景网格结点编号，p 表示物质点编号，则物质点法在一次循环过程中的计算流

程如下：

(1) 物质点上的质量和动量通过形函数映射到背景网格上：

$$m_i^k = \sum_p m_p S_{ip}^k \tag{1-1}$$

$$\boldsymbol{p}_i^k = \sum_p m_p \boldsymbol{v}_p^k S_{ip}^k \tag{1-2}$$

其中 m、$\boldsymbol{p}$、$\boldsymbol{v}$ 分别为质量、动量和速度，S_{ip}^k 为网格结点 i 的形函数在物质点 p 处的值。

(2) 对背景网格结点施加边界条件。

(3) 计算背景网格结点力：

$$\boldsymbol{f}_i^k = \boldsymbol{f}_i^{\text{int},k} + \boldsymbol{f}_i^{\text{ext},k} \tag{1-3}$$

其中，

$$\boldsymbol{f}_i^{\text{int},k} = -\sum_p \boldsymbol{\sigma}_p^k G_{ip}^k \frac{m_p}{\rho_p^k} \tag{1-4}$$

$$\boldsymbol{f}_i^{\text{ext},k} = \sum_p m_p S_{ip}^k \boldsymbol{b}_p^k + \int_\Gamma S_{ip}^k \boldsymbol{t}^k d\Gamma \tag{1-5}$$

而 G_{ip}^k 为网格结点 i 的形函数的导数在物质点 p 处的值，外力为体力 $\boldsymbol{b}$ 和面力 $\boldsymbol{t}$ 的和。

(4) 在背景网格结点上积分动量方程：

$$\boldsymbol{p}_i^{k+1} = \boldsymbol{p}_i^k + \boldsymbol{f}_i^k \Delta t \tag{1-6}$$

其中 Δt 为时间步长。

(5) 将速度和加速度映射回物质点：

$$\bar{\boldsymbol{v}}_p^{k+1} = \sum_i \frac{\boldsymbol{p}_i^{k+1}}{m_i^k} S_{ip}^k \tag{1-7}$$

$$\boldsymbol{a}_p^k = \sum_i \frac{\boldsymbol{f}_i^k}{m_i^k} S_{ip}^k \tag{1-8}$$

更新物质点的位置和速度：

$$\boldsymbol{x}_p^{k+1} = \boldsymbol{x}_p^k + \bar{\boldsymbol{v}}_p^{k+1} \Delta t \tag{1-9}$$

$$\boldsymbol{v}_p^{k+1} = \boldsymbol{v}_p^k + \boldsymbol{a}_p^k \Delta t \tag{1-10}$$

(6) 对于 MUSL[14]格式，上步更新后的物质点速度重新映射回背景网格：

$$\boldsymbol{v}_i^{k+1} = \frac{1}{m_i^k} \sum_p m_p \boldsymbol{v}_p^{k+1} S_{ip}^k \tag{1-11}$$

(7) 再次对网格结点施加边界条件。

(8) 最后使用 MUSL 更新得到的背景网格结点的速度计算物质点上的应变增量和旋度增量：

$$\Delta \boldsymbol{\varepsilon}_p = \frac{\Delta t}{2} \sum_i \left[\left(G_{ip}^k \boldsymbol{v}_i^{k+1} \right)^T + G_{ip}^k \boldsymbol{v}_i^{k+1} \right] \tag{1-12}$$

$$\Delta \boldsymbol{\omega}_p = \frac{\Delta t}{2} \sum_i \left[\left(G_{ip}^k \boldsymbol{v}_i^{k+1} \right)^T - G_{ip}^k \boldsymbol{v}_i^{k+1} \right] \tag{1-13}$$

然后调用本构关系更新物质点上的应力、压力以及声速等。

在前面算法的基础上，我们实验室使用 C++语言研发了冲击爆炸三维物质点法数值仿真软件 MPM3D (计算机软件著作权登记 2009SRBJ4761，2009.7)，采用了面向对象的设计思想。在此有必要声明一下，在实现过程中上述步骤中的(1)、(3)、(5)、(6)和(8)步通过对物质点进行循环实现，而(2)、(4)和(7)步通过对背景网格结点进行循环实现。这种实现方式对使用 OpenMP 的物质点并行算法的影响比较明显。

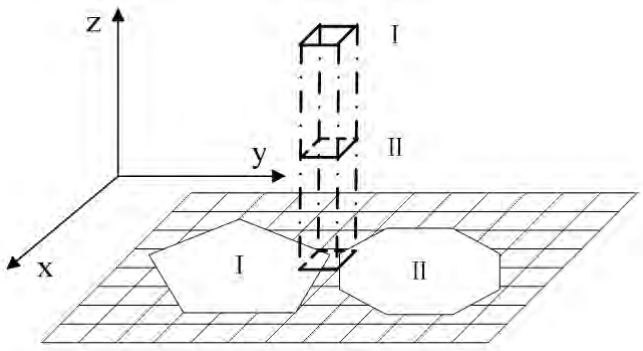


图 1.2　接触算法示意，局部多重网格[16]

接触算法使用局部多重网格思想[16]，如图 1.2。在背景网格初始化阶段，每个物质点分别映射到其所在网格的结点上，每个结点记录下对该结点有贡献的

物质点所属物体的编号。如果结点探测到有多个物体对该结点均有贡献，则判定该结点处于接触状态。此时就在原来网格基础上，创建若干层局部的网格，并把不同物体的物质点贡献的变量分别存储到相应层网格。Hu[15]提出的接触算法使用的是全局多重网格，每个物体分别在自己所在的网格运动，局部多重网格的使用极大地减少了内存使用量。网格初始化完成后，处于接触状态的网格结点根据不同物体对该结点贡献的大小调整接触力，实现接触算法。添加接触算法后的计算流程见图 1.3。

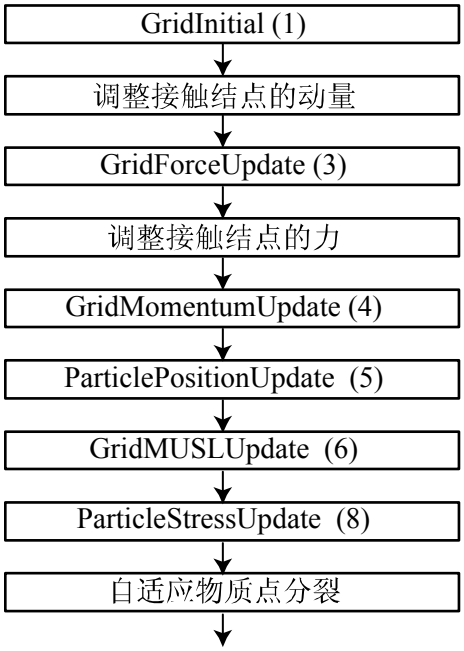


图 1.3　添加了接触算法和自适应物质点分裂算法的计算流程

动态网格也是为了减少内存使用量。如在计算侵彻问题时，板与弹药所覆盖的网格占所有网格比例自始至终都比较小，未被覆盖的网格占用了大量内存，因此如果不为不存在物质点的网格分配内存就会节省大量内存，此即动态网格。

自适应物质点分裂算法是为了避免发生数值断裂。如在模拟金属射流时，射流尖端由于物质点的数量很少，很容易出现紧邻的物质点间距大于网格尺寸。由于物质点间的作用力是通过网格传递的，两者之间就在数值上发生了断裂，物理上可能并未断裂。自适应物质点法让物质点根据一定的原则分裂为若干个较小的物质点，减弱了数值断裂对结果的影响。其算法基本思想[10]是通过累积塑性应变计算出质点在某个方向上的伸长量(图 1.4)，如果伸长 L_i 大于一个定值，

则该物质点在该方向上发生分裂，分裂后的子物质点的质量、体积、应力、累积塑性应变等根据一定的规则得到赋值。

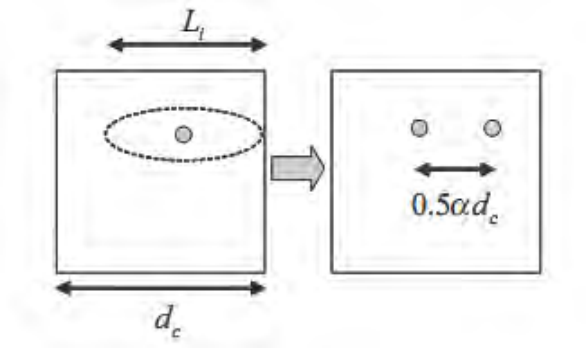


图 1.4　自适应物质点算法示意[10]，L_i 为物质点在 i 方向上伸长的长度，d_c 为网格尺寸

1.3　并行计算机与编程模型

并行计算机的结构大致可分为两类。第一类是通过高速网络互联的并行计算机群(Computer Cluster)，见图 1.5。每台计算机有自己私有的处理器(CPU)和内存，称为一个节点。节点间通过互联网连接。连接方式多种多样，可以有网格拓扑、星形拓扑、直线拓扑、环形拓扑等，图 1.5 中所示即为直线拓扑。第二类为共享存储并行机，其特征为所有内存统一索引为一个地址空间，由所有处理器共享使用，如图 1.6。共享存储计算机又可分为两类，一类是对称多处理机(SMP)，这类机器的不同处理器与内存的距离是相同的，均为本地内存。另一类为不均匀内存存取机器(NUMA)，这种结构的每个处理器既可以读写本地内存(local memory)，也可以读写远程内存(non-local memory)。由于读写远程内存所花费的时间要比读写本地内存花费的时间多，因此称为“不均匀”。 实际上 NUMA 计算机可以看做紧密耦合的第一类并行计算机，即把每个节点的内存统一编址的计算机群。随着并行技术的发展，涌现出越来越多的 SMP 机群，即每个节点是一个独立的 SMP 计算机，节点之间通过高速网络互联，融合了两种并行计算机的优势。

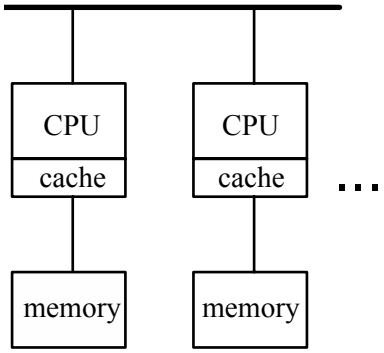


图 1.5　Computer Cluster 的结构，图中为直线拓扑结构

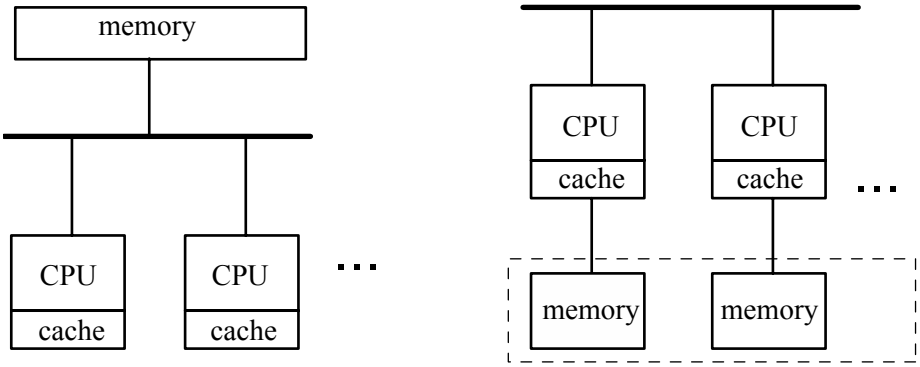


图 1.6　共享存储并行机的结构，左侧为 SMP 结构，右侧为 NUMA 结构，虚线表示统一在同一地址空间内

相应地，并行计算模型总体上可分为两大类：消息传递模型与共享内存模型。消息传递模型(MPI)是基于进程的并行模型，指不同进程在并行计算过程中通过传递消息实现彼此协同合作。这类程序主要运行在并行计算机群上，常用的软件包有 MPICH、OpenMPI 和 LAM/MPI 等。消息传递模型提供了大量的消息发送与进程控制函数，因此编程灵活，可控性强，但实施的工作量较大。共享内存模型是基于线程的并行模型，指不同线程通过读写同一块内存实现线程间协同合作，这类程序主要运行在共享内存并行机上，典型软件有 OpenMP 和 HPF。共享内存模型通过读写内存实现线程间信息传递，通过添加到串行程序中的指导性语句(directive)控制线程的步伐。注意到上面两种模型分别使用了名词“进程”和“线程”。通常一个进程可以包含若干个线程，它们可以利用进程所拥有的资源。进程是分配资源的基本单位，而线程是独立运行和独立调度的基本单位。由于线程比进程更小，基本上不拥有系统资源，故对它的调度所付出

的开销就会小得多，能更高效的提高程序的并发执行的程度。

1.4　并行计算中的几个重要概念

为了衡量并行程序的效率需要引入加速比与效率的概念。加速比就是程序在并行化之前的运行时间与并行化之后的运行时间之比。并行效率为加速比与使用的处理器个数只比。记，T_s 为串行程序的运行时间，T_p 为并行程序的运行时间，n 为运行并行程序所使用的处理器的个数，则加速比的定义为：

$$S=\frac{T_s}{T_p} \tag{1-14}$$

效率的定义为：

$$E=\frac{S}{n}\times 100\% \tag{1-15}$$

如果记串行程序中可以并行的部分的运行时间为 T_{sp}，不可以并行化的部分的运行时间为 T_{ss}，则并行加速比可以写作：

$$S=\frac{T_{sp}+T_{ss}}{\frac{T_{sp}}{n}+T_{ss}} \tag{1-16}$$

让 n 趋于无穷大，

$$S_{max}=\frac{T}{T_{ss}}=\frac{1}{1-P} \tag{1-17}$$

其中 $T=T_{ss}+T_{sp}$ 为串行计算时间，$P=T_{sp}/T$ 为可并行化部分所占比例。这个规律称为 Amdahl 规律。它表明串行程序可并行化的比例越高，则能达到的最大加速比也越高。一个串行程序如果没有全部并行化，则无论使用多少处理器，其最大加速比不会超过 S_{max}。从图 1.7 可以看到，当可并行化比例为 75%，使用 16 个进程时，其最理想加速比仅为 3，因此并行化部分的比例会严重影响程序的可扩展性。如果比例较低，随着使用进程数目的增多，其并行效率会迅速下降。

Amdahl 规律没有考虑程序的额外负载，比如因传递消息而花费的时间、同步点前的等待时间等，记这部分时间为 T_o，则加上这部分时间后，并行加速比为：

$$S=\frac{T_{\mathrm{sp}}+T_{\mathrm{ss}}}{\frac{T_{\mathrm{sp}}}{n}+T_{\mathrm{ss}}+T_{\mathrm{o}}} \tag{1-18}$$

因此如果额外负载过多，将进一步恶化程序的效率。这段论述表明，并行程序要达到较高的效率需要满足：一、并行化部分占串行程序的比例要高；二、并行带来的额外工作要尽量少。

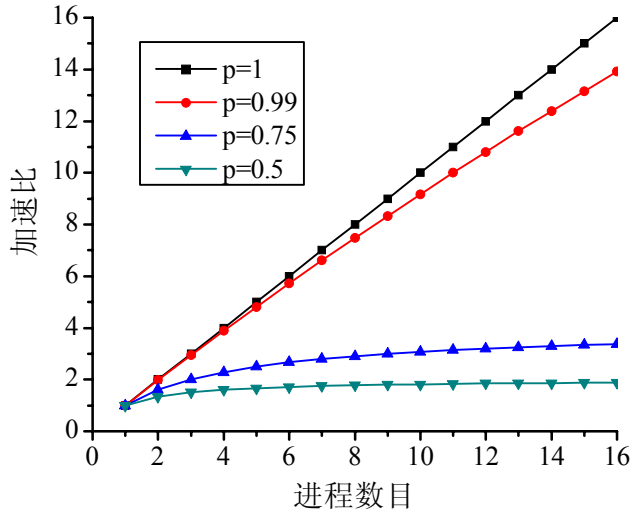


图 1.7　Amdahl 规律对于计算加速比的影响

Amdahl 规律描述了负载平衡时的理想情况，很多情况下负载是不平衡的。如果负载不平衡，在同步点前会出现任务少的处理器在闲置，而任务多的处理器在全速运转的情况，如此则会造成计算资源的浪费。负载平衡的研究也是当前并行算法研究中比较热门的一块领域。

1.5　并行计算在力学中的应用

各种并行算法的思想都是大同小异，即把原来由一个处理器做的工作拆开由不同的处理器共同同时完成，并尽量减少因拆分而引起的额外工作，保持各处理器的工作量大小相同，尽量让总的运行时间最短。然而开发并行算法需要因程序制宜，要根据原有串行算法的特征、运行平台的硬件结构等选择合适的开发工具，提出相应的并行算法。

计算流体力学程序多采用欧拉格式，如有限差分法、有限体积法等。这类算法计算网格固定，一般是规则网格(不规则网格也可以映射到规则网格上求解)，这些特征给并行计算带来了很多方便。如果采用消息传递模型，这类算法一般采用区域分解算法。首先，由于其网格连接方式的固定性，消息传递的源与目的很明确。其次，因为每个网格或节点上的计算量是固定的，只要在开始区域分解的时候尽量保持进程间任务的平衡，在随后的计算中各进程的负载强度是不会变化的，自始至终都会维持较好的负载平衡。再次，欧拉网格多为规则的六面体(在二维程序中为长方形网格)，分区比较简单，按照某条网格线划分开即可，如图 1.8。

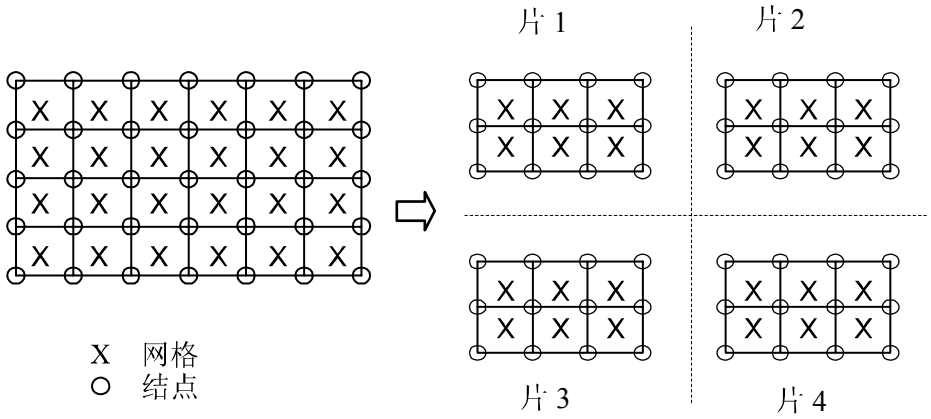


图 1.8　区域分解的思想

NPB 是美国 NASA 开发的一套使用 MPI 的 CFD 大规模并行软件，并且一直用作检验并行机器性能的测试软件，其核心算法是基于 Beam-Warming 格式和 ADI 格式的有限差分法。Jin 等获得了该软件的串行版本，并使用 OpenMP 将其并行化[17], 取得了很好的并行效率，被公认为 OpenMP 的一个典型应用。Jin 重点强调了使用 OpenMP 的并行程序，只有串行程序得到足够优化才有可能达到较高的并行效率。这是因为 OpenMP 将所有内存包装起来，隐藏了数据的局部性，不利于缓存重用，而较高的并行效率需要各个线程分配的数据储存在连续的内存块上，以最大限度的重复利用缓存。

Jay Hoeflinger 等使用 OpenMP 对两套 CFD 软件 ROCFIRE 和 ROCFLO 做了并行化的工作[18]。Jay 认为在使用 OpenMP 并行化程序的过程中不仅要不断的提高程序已并行化部分占整个程序的比例(即 1.4 节中的 P)，还要不断地寻找已并

行化部分中效率最差的代码，并改进它的效率，OpenMP 的增量并行模式为这样做提供了方便。他在文献中对使用 OpenMP 并行化的过程以及每一步的效果做了详细描述，并且得到了良好的结果。

有限单元法中的单元之间互联方式也是固定的，与欧拉格式类似，也比较容易达到负载平衡。有限元法常常采用三角形、四面体单元等，互联方式比较复杂，分区的时候不仅要尽量做到负载平衡，还要尽量减少分区断面的大小以减少通信量，因此相对欧拉网格分区也比较复杂。目前比较常用的有限元网格分区软件有 Metis[22]和 Chaco[23]。Metis 由美国明尼苏达大学的 George Karypis 和 Vipin Kumar 开发，而 Chaco 由美国 Sandia 国家实验室开发。Metis 还提供了并行版本 ParMetis。

在有限单元分区时，可以从单元的边或面切开，称为分割单元法，也可从结点处切开。从结点处切开比较直观，也比较常见。Krysl 等对分割单元法的可行性进行了研究[21]，认为分割单元法尽管额外负载较多，但仍然是可扩展的。

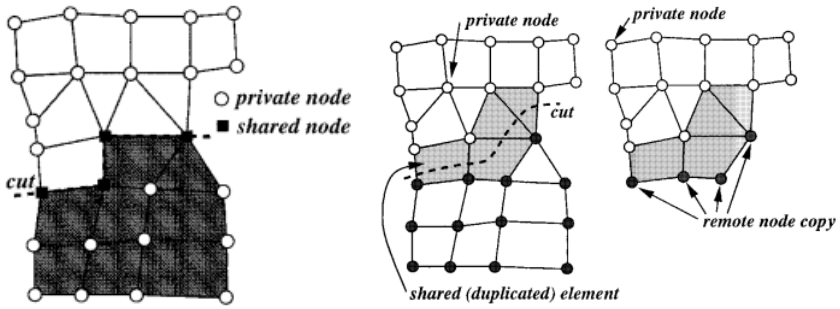


图 1.9　有限单元法的不同分区方式[21]

有限元接触算法要搜索界面单元，以判断物体间是否存在穿透，这一步要耗费大量的计算时间。在并行计算中，因为有可能两个分区分别包含有两个要接触的物体，这样分区间就需要交互大量信息。Malone 等对有限元并行接触算法做了详细的研究[24]。Kevin 等对有限元、SPH 耦合的接触算法，做了详细总结[25]，并提出了一种新的实施接触算法时的负载平衡思想，取得了良好的效果。

然而质点类方法的并行算法研究遭遇着很大的困难，其原因在于质点会在计算区域内部移动，负载难以维持平衡。Lammps 是一款常用的分子动力学软件。由于 Lammps 计算的问题中原子的移动不是很明显，如纳米压痕模拟中原子可以认为是准静态的，这样即使采用静态分区策略，负载平衡到模拟后期也不会有显著恶化。

PIC 方法和 MPM 方法很类似，都采用了质点和网格双重离散的办法，因此 PIC 方法的并行算法研究对 MPM 的并行算法研究很有借鉴意义。该类算法最大的困难在于质点在计算区域中是动态移动的，实现负载平衡就会遇到很大困难。Othmer 提出了使用任务池思想(taskfarm)[26]，并使用了 MPI-2 标准中提供的远程内存读取函数实现该思想。Plimpton 则将计算区域分为很多小片并分配给不同进程[27]，当有负载不平衡出现时，过载的进程将部分小片内质点的信息发送给空闲的进程进行计算。此方法的效果很好，但是涉及到复杂的通信操作，较难实施。

Uintah 是美国犹他大学的 Parker 等开发的一套并行计算的基础设施[19, 20]。它提供了一套在成千上万个 CPU 上操纵自适应网格(AMR)和质点的函数，可以方便地实施求解大规模偏微分方程算法。 Uintah 采用了消息传递模型，并创造性地提出了任务图设计思想。它将算法看做任务之间的依赖图，每个任务消化一些数据并生产一些数据，任务之间建立数据流。此外，还存在一个数据仓库，所有数据对每个任务可见，这样就可以方便地把任务调度和数据调度分开来，达到负载平衡的目的。目前为止，在 Uintah 上实现的算法有欧拉 CFD 算法和物质点法等。关于负载平衡，Uintah 实现了简单的静态平衡策略，但它为实现更复杂的负载平衡算法提供了平台。尽管平衡算法简单，但文献[20]中的几个算例却表现出了良好的可扩展性，当 CPU 数目达到 1000 的时候，加速比几乎呈线性增长，但其文献中的算例仍然基本是准静态的。

1.6　本文的主要工作

本文的主要目标是对基于 MPI 和 OpenMP 对物质点法的并行算法进行初步探索，并试图提出针对质点网格双重离散格式，且质点在网格内快速移动的算法的负载平衡算法。

本章对于进行物质点并行算法研究的意义、并行计算的背景知识以及并行计算在计算力学中的应用做了简要介绍。

第二章主要介绍了使用 MPI 的并行物质点法的实现细节，主要内容为如何使用消息传递函数实现背景网格的并行更新和物质点的跨区移动，并基于 MPI 将 MPM3D 并行化，编制了并行程序 MPM3D_MPI，通过算例验证了程序的正确性和稳定性，不同的算例显示了程序在不同情况下的并行效率。

第三章介绍使用 OpenMP 的并行物质点法的实现细节，提出了一种新的分区方式，它易于实现，具有良好的负载平衡，并且具有较广的适用性，并基于 OpenMP 将 MPM3D 并行化，编制了并行程序 MPM3D_OMP。部分算例验证了该方法的良好效果，并针对其固有的缺陷提出了改进办法。

第四章对本文工作做了简单总结，并为物质点并行算法提出了展望。

第 2 章　基于 MPI 的物质点并行算法

2.1　MPI 简介

MPI 是一组库函数，用于在进程间发送消息，整个 MPI 函数库包含一百二十五条函数，然而常用的不到十条，包括点对点通信、群组通信、障碍函数、计时函数等。

(1) MPI 程序在开始和结尾的时候要调用下面的函数以创建和结束 MPI 程序的运行环境(以 C++为例，下同)：

```
void MPI::Init(int argc, char **argv)
void MPI::Finalize()
```

(2) 点对点通信含有若干种模式，阻塞型通信函数最为常用：

```
void MPI::Comm::Send(const void* buf, int count, const
                MPI::Datatype& datatype, int dest, int tag)
void MPI::Comm::Recv(void* buf, int count, const
                MPI::Datatype& datatype, int source, int tag)
```

其中 Comm 是一个通信子，即一组进程的集合。通信子定义了发送消息的域，程序中可以存在若干个通信子，每个进程在通信子中由进程号定位。第一个函数实现的功能是将 count 个存储在 buf 里面的 datatype 类型的数据发送给进程 dest。如果两个进程之间有多次发送和接受，可以将 tag 分别设为不同的值，以区别对待不同次发送的数据。第二个函数实现的功能是从编号为 source 的进程接收 count 个 datatype 类型的数据，并存储在 buf 内。datatype 是 MPI 自定义的数据类型，包括 MPI_INT，MPI_DOUBLE，MPI_CHAR 等。

阻塞式发送是指只有当缓存里面的数据得到转存，如转存到系统的缓存里或者成功发送给接收进程，函数才会返回。如果操作系统不提供缓存，则发送函数只有等到接收进程调用了接收函数并且已经成功接受的情况下，才会返回。如果操作系统提供缓存，但发送的数据量超过了系统缓存大小，发送函数也会做同样等待。这种情况下，两互相发送消息的进程，就容易永远等待彼此，称为死锁。为了避免程序死锁，同时为了提高程序性能，让消息发送与其他工作

同步进行，就需要使用非阻塞式发送函数：

```
MPI::Request MPI::Comm::Isend(const void* buf, int count,
             const  MPI::Datatype& datatype, int dest, int tag)
MPI::Request MPI::Comm::Irecv(void* buf, int count, const
             MPI::Datatype& datatype, int source, int tag)
```

非阻塞式发送函数，只是初始化了发送或接收操作，并没有完成它，因此其返回值为一个 MPI 请求 `MPI::Request`。 当需要完成通信时，该请求调用等待函数：

```
void MPI::Request::Wait()
```

等待函数直到完成了通信操作才会返回。在消息发送函数的初始化函数与等待函数之间便可以插入其他函数模块，使进程在发送消息时还可以同时做其他工作。本文在更新背景网格以及跨区移动物质点时所用的 MPI 通信函数均为非阻塞式通信函数。

(3) 群组通信函数有广播函数、障碍函数、归约函数等，更详细的介绍请参阅文献[33, 34]。

(4) MPI 程序在执行时，用如下命令：

```
mpirun -np nproc -hostfile hostfile ./a.exe arguments
```

其中参数 `np` 指定并行程序由 `nproc` 个进程执行，其主机结点由 `hostfile` 确定。“`./a.exe`”是可执行文件，`arguments` 是可执行文件的其他参数，如输入文件等。

2.2　使用 MPI 的物质点并行算法

物质点法大约一半计算量存在于与物质点相关的计算，如物质点的位置以及应力的更新，另外一半的计算在于物质点与网格之间的相互映射。与背景网格结点相关的计算量只有动量积分和施加边界条件，占所有计算量中很小的一部分。因此如果对物质点进行分区，理论上会实现比较好的负载平衡。但物质点在背景网格内的分布是极其不均匀的、动态的，很有可能其初始分布在网格的一端，而到了模拟的结尾，物质点移动到了网格的另一端，而且其空间位置可能与编号顺序完全不符，因此必须将按照网格分区考虑进来。

Plimpton 在 PIC 模拟[27]中使用了 RCB [32] 分区算法，分区得到的结果是不规则的网格区域(图 2.1)，分区间的互联方式复杂多样，这样就需要一套寻找邻

居进程的算法，通信会非常复杂。其负载平衡算法为将过载进程里部分质点的相关信息发送给空闲进程，让空闲进程只负责临时的计算，计算完毕即回发给过载进程。Parker 等在物质点法并行模拟[19-20]中采用了静态分区的办法，并且取得了良好的效果。综合上述考虑，并行程序 MPM3D_MPI 采用了静态均匀分区算法。

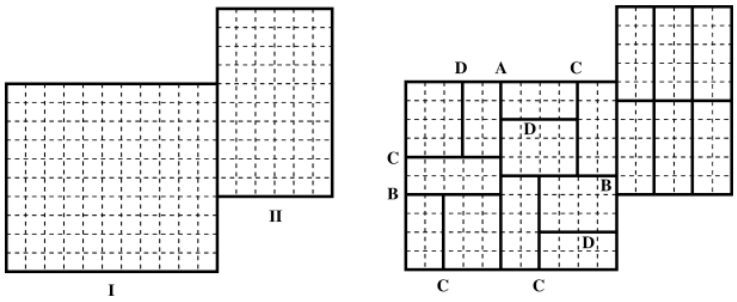


图 2.1　RCB 分区办法：先沿 A 线切开，得到的两个分区分别沿 B 线切开，之后依次沿 C 和 D 线分开，努力保证每次划分两侧的分区的任务量一致

2.2.1　文件输入方式

串行程序 MPM3D 输入文件时，定义了 `CParseXMP` 类。该类存储了程序运行需要的所有数据。该类比较复杂，其数据成员与程序的计算模块一一对应，内部含有许多 C++标准模板类的对象，如 `map`，`vector` 等。程序在运行之初，首先将输入文件的信息读入并存储在该类的对象内。然后各个计算模块依次调用该对象进行初始化。

并行程序输入信息可以采用两种方式，一种是只有 0 号进程读取输入文件并打包，然后将打包后的信息发送给其他进程。串行程序已经把输入的数据打包好，存储在 `CParseXMP` 的对象内，但由于该类的复杂性，不能直接使用 MPI 提供的发送字节流的函数将该对象发送给其他进程。即使能够直接发送给其他进程，接收之后同样要由各计算模块依次调用该数据对自身进行初始化，与实际使用的方法相比不仅麻烦而且耗时。

考虑到不同进程所需数据除物质点以及背景网格不同外，其他数据完全相同。因此 MPM3D_MPI 让所有进程同时读入输入文件，在读入背景网格的时候，按照前面提到的分区办法进行分区；读入物质点的时候则判断该物质点是否属于本分区，如果不属于，则舍弃该物质点。

2.2.2　分区方式以及进程拓扑

串行程序 MPM3D 为物质点和背景网格分别建立了数据结构，MPM3D_MPI 也利用了这些数据结构，用于存储自己所负责的子分区的物质点和背景网格。我们采用对背景网格进行静态均匀分区方式，每个子分区由一个进程负责。分区边界为某个背景网格面。按照背景网格的空间位置，物质点分别划分到相应的子分区，并由相应的进程负责。为了便于寻找每个进程的邻居，分区分别沿三个维度方向均匀划分若干份，彼此交错形成分区，图 2.2 右侧为背景网格的分区方式。

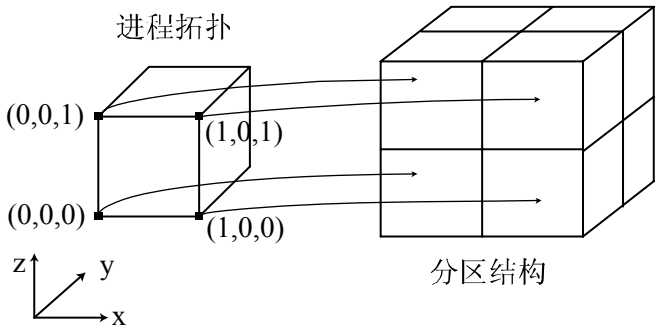


图 2.2　进程拓扑与分区示意图

分区信息通过输入文件获得。根据分区信息在进程之间建立相应的空间位置关系，称之为进程拓扑。由于网格一般沿直角坐标系的坐标轴方向，程序 MPM3D_MPI 使用下面的函数创建进程的笛卡尔拓扑：

```
MPI::Cartcomm MPI::Intracomm::Creat_cart(int nDim, const int
                Dims[],const bool IsPeriodic[], bool ReOrder)
```

其中，`nDim` 为创建的拓扑的维数，`Dims` 为分别在 `nDim` 个维度上的长度，`IsPeriodic` 在某维度上是否周期，`ReOrder` 表示是否对进程重新编号。根据进程拓扑的空间位置关系，可以方便寻找每个进程在各个方向上的邻居的进程号。同时计算网格也按照输入的分区信息进行切割。切割后的计算网格的空间位置关系与分区信息一致，也就与进程拓扑一致。根据分区后的计算网格的空间位置，把子网格的相关描述信息发送给相应的进程，这样就成功地把背景网格分配给了不同的进程。图 2.2 左侧的小正方体代表进程拓扑，每个顶点代表一个进程，右侧的正方体代表整体背景网格，每个小方块代表一个子分区，进程的拓

扑位置与网格分区相对位置一一对应。

尽管 MPM3D_MPI 仅实现了按照空间位置平均划分网格的算法，但可以很容易地添加考虑物质点对计算量的贡献的分区算法以及一些更复杂的负载平衡算法，因为一旦背景网格分区确定下来，物质点的跨区移动都是自动进行的。

分区完成后，可以使用 MPI 提供的库函数寻找到自身在每个方向上的邻居进程的进程号：

```
void MPI::Cartcomm::Shift(int direction, int disp, int&
                rank_source, int& rank_dest)
```

该函数获得了编号为 `rank_source` 的进程，沿进程拓扑的 `direction` 方向移动 `disp` 个位置的进程编号，结果存储在 `rank_dest` 内。程序 MPM3D_MPI 中建立了数组 `neighbour[3][2]`，分别存储三个维度方向上左右两个邻居的编号。

2.2.3　背景网格的并行更新

相邻的分区需要共同更新彼此共有的网格边界。背景网格总共要更新三次，分别为 `GridInitial`，`GridForceUpdate`，`GridMUSLUpdate`。如果启用接触算法，还要更新网格上的接触力。这几次更新都对相同的背景网格边界结点遍历，区别在于更新的是不同的变量。为了减少更新时间，我们创建了背景网格边界结点列表，更新背景网格时都使用该列表。进程对列表遍历，依次将结点的所需变量存入缓存。

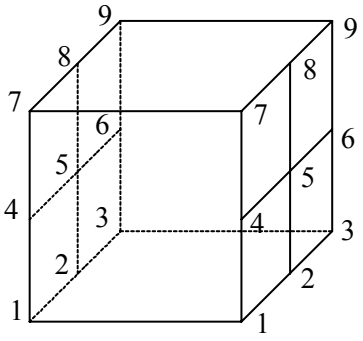


图 2.3　背景网格结点列表示意

建立背景网格边界结点列表时相邻分区共享的背景网格边界结点的排列顺序要一致。假设图 2.3 中网格为某一进程负责，该进程要创建它的六个网格边界面的结点列表，则在某一维度正负方向的两个面上的结点需按照相同顺序排列。

如图，其左面的结点顺序要与其右面的结点顺序一致。所有进程都按照此顺序建立自己的结点列表。在接收网格信息的时候，只要调用自己对应面的结点列表，把接收到的信息依次填充给结点即可。

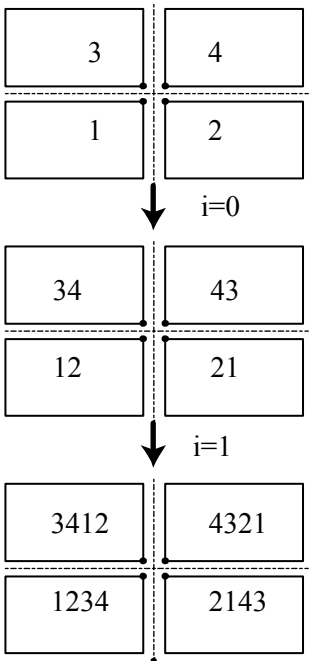


图 2.4　背景网格并行更新模式

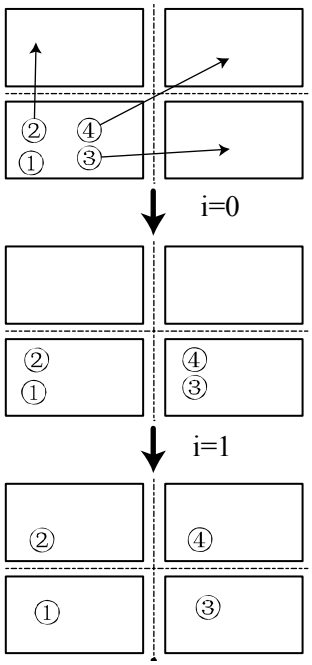


图 2.5　跨区移动物质点示意图

未添加接触算法的 MPM 在背景网格并行更新时因为每个结点上数据的长度是一致的，存放到缓存里时，只要按照列表遍历，将需要的变量存入即可。解包时，同样地将缓存里一定的长度作为一个结点的变量，依次读取。将每个面上结点的数据打包好后，按照下面的模式将相应面的数据发送给相应的邻居进程：

```
For(int i=0;i<3;i++)
   send to neighbour[i][0]
   receive from neighbour[i][1]
   send to neighbour[i][1]
   receive from neighbour[i][0]
End
```

按照上面的消息发送模式，位于背景网格边界面上中间的结点(如图 2.3 中的 5 号结点)，只要通过接收邻居的消息，即可得到正确的更新。那么位于角点上(如

图 2.3 中的结点 9)的结点呢？图 2.4 是一个二维的例子，四个进程负责四个子分区，黑色实结点应该由与它连接的四个进程共同更新。数字为进程号，同时也表示本区的黑色实结点拥有来自该进程的信息。在更新前，每个进程的黑色结点上只有自己分区的物质点对它的贡献。经过两次更新后，每个进程的黑色结点上都含有了来自其他进程所有的贡献，因而得到了正确的更新。

2.2.4　物质点的跨区移动模式

每个分区的周围最多有 26 个邻居子分区。由于时间步长的限制，任一子分区内的物质点经过一步计算,可能移动到与其相邻的 26 个子分区中的任何一个，但不会越过这 26 个子分区，移动到更远的子分区里去。MPM3D_MPI 不加判断它到底移动到了哪一个邻居所属的子分区里，而是采用了与背景网格的并行更新类似的方法：

```
For(int i=0;i<3;i++)
   For(int j=0;j<nb_particle;j++)
      p=particle_list[j]
      if(p.x[j]<grid.spanlow[i])
         add p to plist_l
      if(p.x[j]>grid.spanup[i])
         add p to plist_r
   End

   send plist_l to neighbour[i][0]
   receive from neighbour[i][1]
   send plist_r to neighbour[i][1]
   receive from neighbour[i][0]
End
```

其中,p 表示物质点,p.x 表示物质点 p 的位置,grid 为局部网格,grid.spanlow 和 grid.spanup 分别为局部网格的上下界，plist_l 表示向某维的负方向发送的物质点列表，相应地 plist_r 表示向某维的正方向发送的物质点列表。如图 2.5 所示，分别在所有维度方向上的移动后，物质点被成功地移动到了相应的分区。

2.2.5　对部分算法的支持

MPM3D_MPI 添加了对接触算法、动态网格以及自适应物质点分裂算法的支持。添加了这些算法支持后的 MPM3D_MPI 的计算流程见图 2.7。

接触算法：并行程序对接触算法的支持体现在两个方面。更新背景网格边界结点时，打包函数需要将每个存在接触的网格结点上所有局部层上的数据全部放入缓存；收到邻居传递来的背景网格信息后，解包函数同样把存在于每个结点上的若干层网格，依次存入本地相应的局部网格，因而只需要对存放背景网格结点上数据的缓存进行重新设计。启用接触算法后缓存的数据结构见图 2.6。每个结点分配的数据段开头都会存储一个整型数，表示该结点的局部网格层数，之后依次存储所有层网格的数据，包含组件编号等。

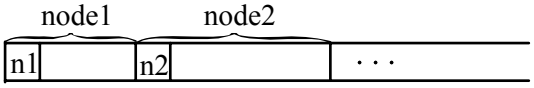


图 2.6　启用了接触算法后，背景网格的缓存的数据结构

动态网格：并行程序对于动态网格的支持与对接触算法的支持实现思想类似，并采用了同样的缓存结构。在更新背景网格时，打包函数如果遇到背景网格边界某结点为空，则设置该结点的层数为 0；解包函数如果遇到某结点层数为 0，则直接跳过。

并行自适应物质点分裂法：由引言知添加了自适应物质点分裂法的计算流程只是比原计算流程在程序最后多调用一个函数。考虑到物质点分裂会改变物质点的位置，因此只要在跨区移动物质点之前，先调用自适应算法即可，否则有可能某些物质点会被分裂到子分区外面，引起计算错误。

2.2.6　并行文件输出

MPM3D_MPI 使用 h5part 和 vtk 的并行文件输出功能。

2.3　算例及结果

并行程序与串行程序的结果常常不一致，原因在于并行程序改变了原程序的计算流程。在分区边界上运算顺序与串行程序不同，在消息传递过程中也会因为对数据进行截断而引入微小误差。那么如何测试并行程序，保证结果的正

确性？此外，并行程序除了可以扩大原程序的规模，还需要保持较高的并行效率，以充分利用已有的计算资源。本文主要采用下面几种方案验证并行程序的正确性、稳定性，最后测试了静态分区算法的并行效率。

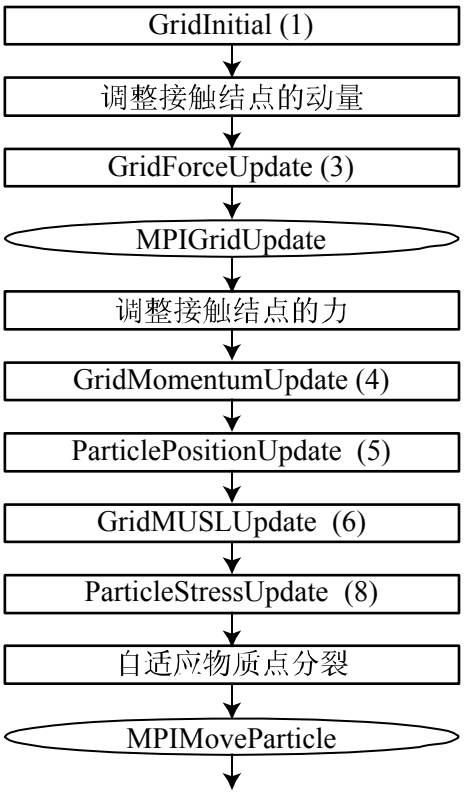


图 2.7　MPM3D_MPI 的计算流程(MUSL 格式)

2.3.1　程序正确性的验证

由于分区边界的存在会给计算程序带来一定的误差，为了检查边界是否会对计算结果产生致命的影响，我们对同一规模的同一问题，使用不同的进程数目与拓扑结构进行计算，观察结果与串行程序结果是否一致：如损伤、塑性变形、压力云图，以及能量曲线等。

水下爆炸模拟的目的主要是为了观察冲击波在水中的传播，研究其对岸边建筑、舰船等的破坏作用。该算例可以通过观察比较冲击波形，考察分区边界对计算结果的影响。我们使用 JWL 状态方程模拟炸药，使用 Grüneisen 状态方程模拟水。JWL 状态方程的表达式为：

$$p = A(1-\frac{\omega}{R_1 V})e^{-R_1 V} + B(1-\frac{\omega}{R_2 V})e^{-R_2 V} + \frac{\omega E}{V} \tag{2-1}$$

其中，A、B、R_1、R_2、ω 为材料参数，V 为材料的体积，E 为材料的初始单位体积内能。模拟中采用的材料参数见表 2.1。

表 2.1　TNT 炸药的参数

A(MPa)	B(MPa)	R_1	R_2	ω	E(MJ/m^3)
3.73e5	3.74e3	4.15	0.9	0.35	6000

Grüneisen 状态方程的表达式为：

$$p = p_{\mathrm{H}} + \frac{\gamma}{v}(e - e_{\mathrm{H}}) \tag{2-2}$$

其中，p_{H} 和 e_{H} 分别为 Hugoniot 曲线上的点的压力和单位质量的内能，γ 为 Grüneisen 常数且$\frac{\gamma}{v} = \frac{\gamma_0}{v_0} =$常数。模拟中采用的参数见表 2.2。

表 2.2　水的参数

C_0(m/s)	S	γ_0
1647	1.921	0.1

本算例使用一层背景网格，在网格内均匀布点。由于对称性，我们采用 1/4 模型计算，其布点如图 2.8，起爆后冲击波将以水为介质向四周传播。离散后总物质点数目约为四万。使用并行程序计算时，分区设置为 4×2×1 和 8×8×1，分别表示在三个维度上平均分区的份数，两种情况分别记为工况二和工况三，串行模拟为工况一。我们将观察当冲击波跨过分区边界的时候，分区边界的存在对冲击波的传播造成的影响。由于该算例的规模较小，主要目的为验证并行程序的正确性，因此在一台八核工作站上完成。

当计算步数接近 1000 时，冲击波将达到边界，此后波将会沿边界发生镜面反射，计算区域内的波形会变得比较紊乱。为了观察到较好的计算结果，我们只观察第 800 步的结果。图 2.9 为计算步数为 800 时三种工况的压力云图。

从图 2.9 基本上观察不到工况二和工况三中压力波的轮廓与工况一的区别。此外三个工况的能量曲线包括总能量、动能以及内能完全一致，因此在此不再

列出。由此可以看出，并行程序的正确性是可以保证的。

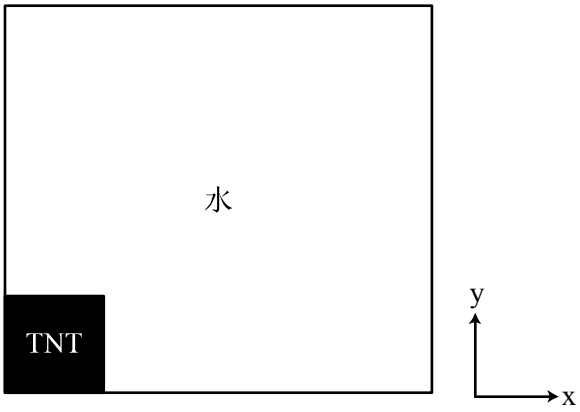


图 2.8　水下爆炸离散模型示意

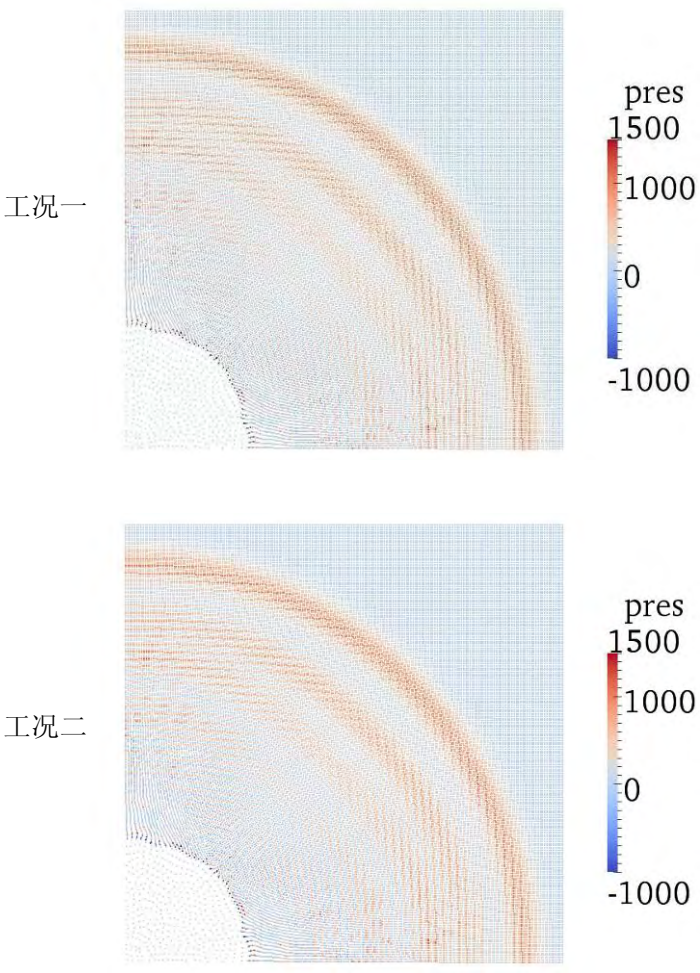
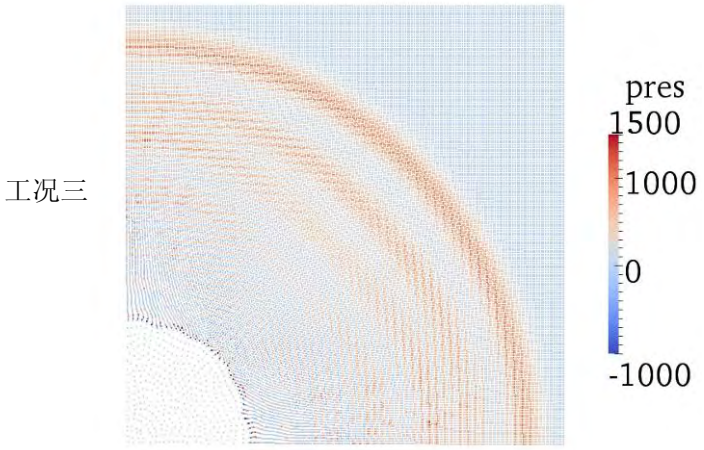


图 2.9　水下爆炸的压力云图



续图 2.9　水下爆炸的压力云图

2.3.2　程序稳定性的验证

稳定性主要验证当程序计算大规模问题时，不会出现如缓存不足以致程序中途退出等异常。此次验证采用机场地下爆炸的算例。由于对称性，只模拟了一半模型，其具体离散方式见图 2.10，各层混凝土采用 HJC 强度模型，炸药使用 JWL 状态方程模拟。离散后的总物质点数目约为五千万。

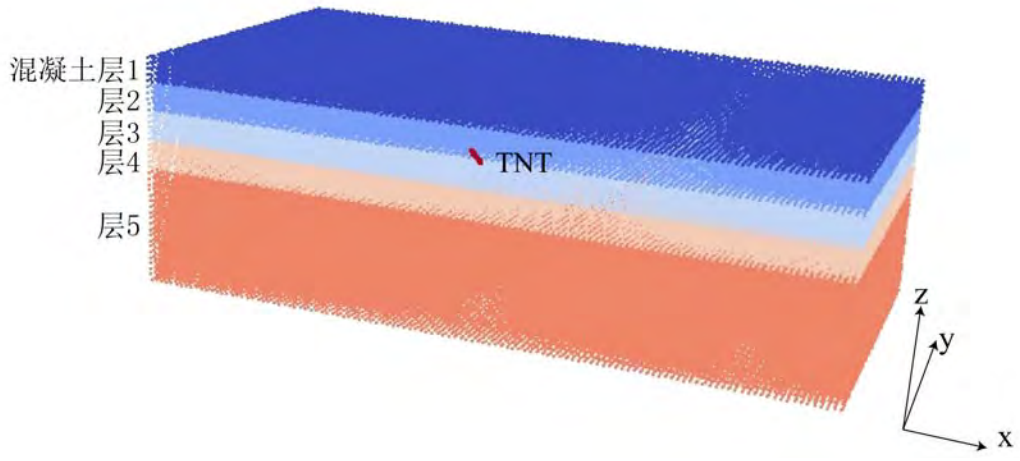


图 2.10　机场爆炸离散模型示意

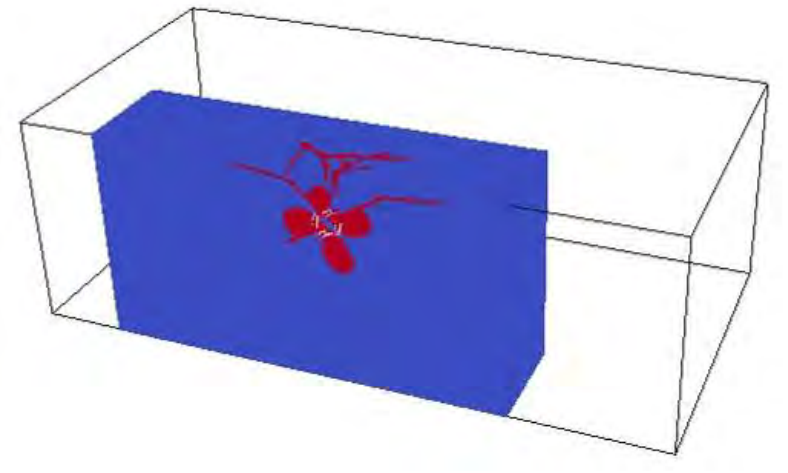


图 2.11　8 毫秒时的损伤云图

此次模拟在大规模并行计算平台上进行，每个结点由两个四核处理器，8 GB 内存组成，共使用了三个结点，约 20 GB 内存。并行分区方式为 8×3×1。图 2.11 为 8 毫秒时的损伤云图，其中黑线框为所有混凝土的体积。由于后处理软件的内存限制，后处理时只观察了爆点附近的几个分区的损伤情况。计算未现异常，验证了 MPM3D_MPI 的稳定性。

2.3.3　并行程序的效率

MPM3D_MPI 采用静态分区算法，随着时间的推进，有些问题中物质点的分布变得极不均匀，而有些问题中物质点依然保持均匀分布，两种问题的效率是完全不同的。本文在两类问题中各挑选了一个代表性的算例，并分别测试了其效率。

2.3.3.1　泰勒杆高速碰撞

泰勒杆高速碰撞刚性壁实验常用于测定材料的在高应变率情况下的参数。在本测试中，材料为铜，采用 Johnson-Cook 强度模型更新应力：

$$\sigma_{\mathrm{y}} = (A + B\varepsilon^{\mathrm{p}^{n}})(1 + C\ln\dot{\varepsilon}^{*})(1 - T^{*m}) \tag{2-3}$$

其中，A、B、C、n、m 为材料常数，ε^{p} 为累积塑性应变，$\dot{\varepsilon}^{*} = \dot{\varepsilon}^{\mathrm{p}} / \dot{\varepsilon}^{0}$ 为无量纲

应变率，$T^* = (T - T_{\text{room}}) / (T_{\text{melt}} - T_{\text{room}})$ 为无量纲温度，其中 T_{melt} 为材料的融化温度，T_{room} 为室温。具体的材料参数见表 2.3。

表 2.3　铜的模型参数

参数名称	参数值	参数名称	参数值
ρ_0(g/mm^3)	8.9e-3	C	0.025
E(MPa)	117.0e-3	M	1.09
v	0.3	T_{room}(K)	293
A(MPa)	98	T_{melt}(K)	1900
B(MPa)	368	c_{v}	385
N	0.7	$\dot{\varepsilon}_0$	1.0e-3

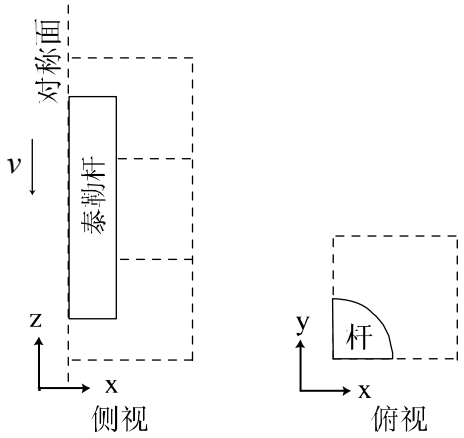


图 2.12　泰勒杆离散示意图，虚线为背景网格

由于泰勒杆的对称性，我们使用 1/4 模型计算，布点情形见图 2.12。沿着 z 方向随着泰勒杆的运动，物质点将覆盖背景网格的不同区域，进程间的负载平衡状况也将随之改变。沿 x 和 y 方向，物质点分布亦很不均匀。取串行模拟为工况一，分区为 1×1×2 为工况二，分区为 1×1×4 为工况三，分区为 1×1×6 为工况四，分区为 1×1×8 为工况五。实测得到的效率见图 2.13。工况五的效率仅为 20%。可见，当负载不平衡时，并行程序的效率可以比较低。

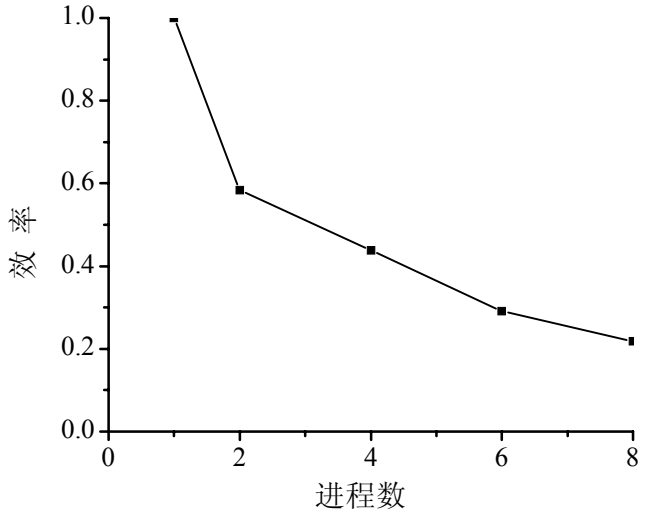


图 2.13　泰勒杆问题的并行效率

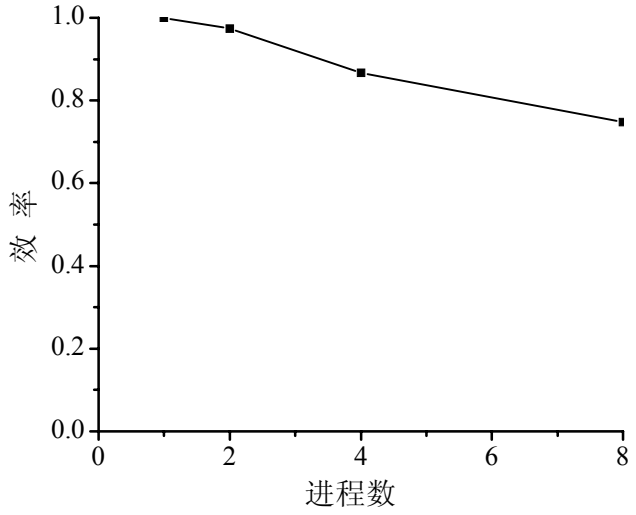


图 2.14　水下爆炸的效率

2.3.3.2　水下爆炸

算例描述见 2.3.1 小节，此处只列出其并行效率。由图 2.8 可见该算例中物质点的分布开始时均匀地铺满了计算网格，此时负载平衡较好，只有到计算后期爆炸形成球形空腔导致局部的物质点分布不均匀才会出现一定程度的负载不平衡，可以预见其并行效率应该比较高。图 2.14 分别是使用 2、4、8 个进程时的效率，分区分别采用 2×1×1, 2×2×1 和 4×2×1。1 个进程指串行程序计算的结果。

此处统计了计算 4000 步的时间。当使用 8 个进程计算时，效率仍高达 75%。可见，当负载平衡能够得到保证时，效率也可以很高。

2.4 小结

本章详细给出了使用 MPI 的物质点并行算法。为了尽量减少对程序的改动，以及方便进程间通信，采用了规则的均匀的静态分区方式，并通过若干算例验证了程序的正确性、稳定性，在此基础上编制了并行程序 MPM3D_MPI。

由于物质点法使用物质点和背景网格双重描述，而物质点在网格内的移动是不可预知的，这就给负载平衡造成了极大的困难。MPM3D_MPI 采用了静态分区策略，每个进程负责一个子分区。对于部分物质点分布比较均匀的问题，MPM3D_MPI 的表现比较令人满意；而对于物质点分布不均匀的问题，可以针对具体问题提出具体的负载平衡算法，但难以提出一个可扩展性较好且具有广泛适用性的负载平衡算法，这些问题使用较少进程进算时可以通过人为分区尽量减小负载不平衡，使用较多进程时常常会出现不同程度的负载不平衡。

尽管如此，MPM3D_MPI 极大地提高了物质点法可以解决的问题的规模。一般的个人电脑能够解决的问题规模大约为 400 万物质点，随着计算机技术的飞速发展，这一数字也会随之改变，但基于 MPI 的并行程序可以将规模提高到并大于 5000 万物质点。同时 MPM3D_MPI 也实现了对接触算法、动态网格和自适应质点分裂算法的支持。

第 3 章　基于 OpenMP 的物质点并行算法

3.1　OpenMP 简介

OpenMP 并非一种新的编程语言，而是在已有编程语言的基础上提供的适合共享内存并行结构的一套指导性语句、函数库以方便实现线程级的并行性。OpenMP 使用 fork/join 模型，见图 3.1。在程序的起始只有一个线程在执行，称为主线程，此时执行的代码块称为串行区。当遇到分叉点的时候，在 C 或 C++ 程序中可以添加指导性语句“`#pragma omp parallel num_threads(nthread)`”，将主线程分叉为 `nthread` 个子线程，进入并行区，每个线程执行不同的任务。其中“`#pragma omp`”表示其后的语句为 OpenMP 指导性语句，“`parallel`”表示将创建一个并行区，“`num_threads(nthread)`”是 OpenMP 的子句，用于设置“`parallel`”的参数，定义了并行区内线程的个数。

其他子句有：“`private，public，shared，threadprivate`”等用于设置并行区域内的变量的公有或私有的属性；“`for，sections，single`”等用于规定在并行区域内如何将工作分配给不同的线程；“`critical，master，barrier，atomic`”为设置同步机制的命令；“`reduction`”对并行区内的变量实行归约操作。

当任务执行完毕，所有线程自行终结，只有主线程会继续执行下去。由图 3.1 可以看出，一个程序可以存在若干并行区域。这就给我们提供了逐段并行化串行程序的方便。另外一方面，也可以逐段检测并提高各并行区域的效率，来逐渐提高程序的整体并行效率。

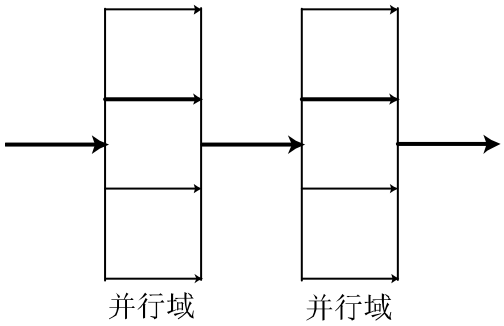


图 3.1　fork/join 模型，较粗的箭头为主线程

OpenMP 最常见的应用是对循环的并行处理，这可以用上面提到的“for”命令来实现，其实现方式为：

```
#pragma omp parallel num_threads(nthread)
{
      #pragma omp for private(私有变量列表)
      For(int i=0; i<end; i++)
         循环体
      End
}
```

或者将二者组合起来：

```
#pragma omp parallel for num_threads(nthread)  private(私有变量列表)
For(int i=0; i<end; i++)
   循环体
End
```

其功能为将循环体中的不同迭代分配给不同的线程，以此实现并行。由于所有线程可以读取所有内存，如果多于一个线程读写内存的同一位置，只要有一个线程是写操作，就很容易出现数据竞争。这时就需要将可能出现数据竞争的变量私有化(见上面代码中 private 子句的用法)，即每个线程为该变量创建一个私有的副本，每个线程只修改自己的副本。

当程序中存在归约变量时，如下面的 sum 变量：

```
For(int i=0; i<end; i++)
   sum+= (…)
End
```

使用 OpenMP 并行时可以方便地使用归约命令实现：

```
#pragma omp parallel for num_threads(n) reduction(+:sum)
For(int i=0; i<end; i++)
   sum+= (…)
End
```

目前为止 OpenMP 的 C、C++实现尚不支持数组归约，其 Fortran 实现支持数组归约。MPM3D 是用 C++编制的，在使用 OpenMP 并行化程序过程中需要人为编

制代码实现数组归约。

控制线程执行步伐的子句，如障碍(barrier)的用法为：

```
#pragma omp barrier
```

这个子句可以让并行域中的线程在此障碍处同步，先到达的线程将等待后到达的线程，直到所有线程都到达，它们才会继续执行。在并行域的结尾处默认存在一个障碍，但可以用 nowait 命令移除。如果障碍之前不同线程所执行的工作量大小不均，这种等待会造成时间的浪费，因此要尽量保证线程间负载的平衡。

子句“critical”常用来避免数据竞争。它可以创建一个临界域，让所有线程顺序执行临界域内的代码，其实质就是串行执行临界域内的代码。如果线程数较多或临界域内代码较长，使用临界域将严重影响程序的并行效率。其他详细的 OpenMP 子句和函数的介绍参见文献[29, 30]。

引言已经强调了数据局部性的重要性。图 3.2 显示程序从主内存中读数据以 cache line 为单位读入。若一次读入的数据不仅满足了本次计算的需要，在之后的若干次计算中也能用到，就会节省大量的内存读操作，称为缓存友好的(cache friendly)。如果 cache line 中的其他部分不能在之后的计算中用到，程序就不得不继续从内存中读数据，称之为缓存不命中(cache miss)。频繁出现的缓存不命中会极大影响程序的效率，这在 OpenMP 并行程序中尤其显著。内存对 OpenMP 程序的所有线程是可见的，所有线程都认为内存为本地的，而事实上不同线程使用的内存可能彼此交错，分布会比较复杂，在执行中就会出现较多的缓存不命中，进而影响程序效率，因此 OpenMP 并行编程时尤其要注意数据的局部性。

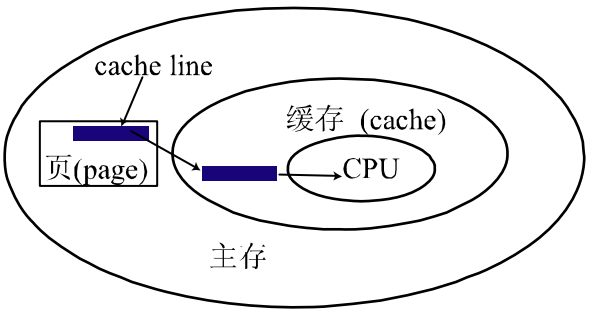


图 3.2　cache 详解

3.2　使用 OpenMP 的物质点并行算法

物质点法中计算量最大的部分分别是更新背景网格和更新物质点。更新背景网格部分包括第二章中所列出的步骤：(1)、(2)、(3)、(4)、(6)和(7)，背景网格上的变量，如质量、动量和力等得到更新。在 MPM3D 中(2)、(4)、(7)步是对网格结点进行循环，循环内不同的迭代是独立的，这段代码就可以直接在循环前添加“`#pragma omp parallel for num_threads(nthread)`”实现并行。而(1)、(3)、(6)步是通过对物质点进行循环。更新背景网格时，如果简单的在循环前添加 OpenMP 提供的指导性语句，就有可能不同的线程同时修改一个网格结点上的变量，这样就会出现数据竞争。一个可行的解决方案是将代码修改为对网格结点循环。文献[31]在 GPU 上执行了 PIC 的并行算法，它采用了“particle pull”的策略(图 3.3)，并提出了一种新的按照物质点位置管理其列表的算法，获得了比较高的并行效率，但如果应用于 MPM3D_OMP，则程序需要做根本改动。

```
//loop over vertices first
Foreach vertex v_s∈G do
  Find P(v_s);
  F(v_s)←0;
  Foreach p_i∈P(v_s) do
    F(v_s)←F(v_s)+ω_iK(v_s, p_i)
  End
End
```

//particle pull

```
//initialize F(v_s)
Foreach vertex v_s∈G do
 F(v_s)←0;
End
//loop over particles first
Foreach particle p_i∈D do
 Find v(p_i);
 Foreach v_s∈v(p_i) do
   F(v_s)←F(v_s)+ω_iK(v_s, p_i)
 End
End
```

//particle push

图 3.3　对于网格更新方式的选择[31]：particle pull 或 particle push

文献[28]提出了两种思想，其一是数组扩展法。其思想是为每个线程创建一套网格，也即为每个线程创建一套背景网格的数据结构，每个线程只修改私有的数据结构，这个过程彼此独立，可以并行执行。之后再把所有线程私有的网格上的数据累加起来，其本质是对数组进行归约。该算法的优点是，算法容易

实现，对串行程序改动量小，且容易通过对物质点分区实现负载平衡；缺点也比较明显，因为本来串行物质点算法中网格使用的内存占程序使用的所有内存的比例很高，如果为每个线程创建一套网格，其内存耗费量势必随线程数目的增加而线性增长。程序的可扩展性就比较差。

另外一种是区域分解法。其基本思想是将背景网格划分为若干块，分配给不同线程，每个线程只更新自己所属区域网格上的物质点。由于区域之间不重叠，各个线程是独立的。这种思想的可扩展性好，额外的内存耗费较少。但其论文中没有提到如何如何实现区域分解的细节，而且与 MPI 分区办法类似，不同的问题的区域分解办法是不同的，也比较难以实现负载平衡。通过吸取前面提到的一些方法的优势和劣势，本文在 3.2.1 节提出了网格交替更新法。

更新物质点包括第二章中列出的步骤(5)、(8)，物质点的位置、速度以及应力应变得到更新。这段代码通过对物质点循环实现，不同的循环迭代之间独立，可以直接使用 OpenMP 的指导性语句实现并行。需要注意避免某些变量出现数据竞争，这将在 3.2.2 小节中详细阐述。

3.2.1　背景网格更新

假设 MPM3D 中实现背景网格更新的伪代码为：

```
For(int i=0; i<nb_point; i++)
   Particle = particle_list[i];
  <map variables on Particle to all grid nodes of the cell
   containing the particle>
End
```

则新的分区和负载平衡的算法如下面所述：首先选定一个方向，作为进行分区的方向。为了更好地获得负载平衡，分区的方向一般沿背景网格最长的边。假设有 N 个线程，则沿分区方向首先分 N 个区，边界为某一背景网格面，尽量让所有个分区内部的物质点数目相等，每个线程分别负责一个区域。然后每个区继续沿分区方向分为两个子区域，边界亦为某一背景网格面。子区域内部的物质点成组，分别命名为 R group 和 L group，同样尽量让不同线程负责区域的 L group 和 R group 内的物质点数目分别相等。在更新背景网格的时候，因为背景网格更新之后的值与对物质点进行循环的顺序无关，我们先让所有线程对自己

负责区域的 L group 内的物质点循环，将其上的物理量通过形函数映射到背景网格上；设置一个障碍；再让所有线程对自己负责区域的的 R group 内的物质点进行循环。图 3.4 示意了当有两个线程的时候，分区的具体办法。

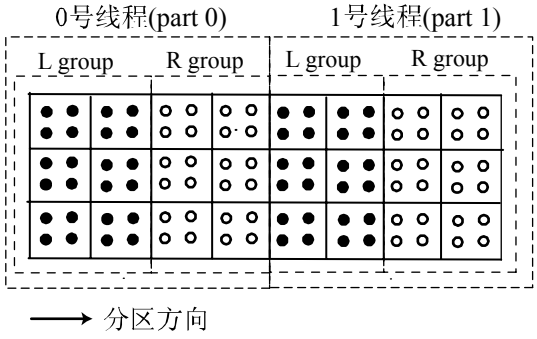


图 3.4　分区方式：实线为背景网格，虚线为分区边界；实点为 L group，虚点为 R group

在障碍之前，每个线程分别更新自己的 L group 内的物质点，由于 L group 覆盖的空间区域不重叠，因此各个线程需要修改的背景网格结点也不重叠，不会出现数据竞争。在障碍之后，类似地也不会出现数据竞争。由于 L group 和 R group 是交替更新的，这个方法称为网格交替更新法。在 L group 与 R group 之间设置一个障碍也是为了避免可能出现的数据竞争。这样操作后，网格内的所有物质点都被映射到背景网格上，保证了结果的正确性。为了更清楚地描述上面的算法，将其写为伪代码如下：

```
#pragma omp parallel num_threads(nthread)
{
   int ithread=omp_get_thread_num();
   For(int i=0; i<nb_point; i++)
      Particle = particle_list[i];
      if(Particle belongs to L group of ithread)
          <map variables on Particle to all grid nodes
          of the cell containing the particle>
   End
```

```
    #pragma omp barrier
    For(int i=0; i<nb_point; i++)
       Particle = particle_list[i];
       if(Particle belongs to R group of ithread)
          <map variables on Particle to all grid nodes
          of the cell containing the particle>
    End
}
```

nb_point 为物质点的数目。由引言知道，在物质点法的计算流程中，背景网格将更新三次，分别为质量、动量、力以及 MUSL 格式中的动量。那么上面的代码将浪费很多不必要的时间，因为在障碍前后每个线程都对所有物质点进行循环，也就是说与串行程序相比对物质点循环多了一次。为了减少运行的时间，我们对每个线程负责的两组物质点进行编号，编号之后每个线程只要对自己负责的编号序列进行循环即可，这样会极大地减少时间浪费。这个过程可以并行执行如下：

```
int *LIndex, *RIndex;
<allocate memory for LIndex and RIndex>
#pragma omp parallel num_threads(nthread)
{
   int ithread=omp_get_thread_num();
   int m,n;
   For(int i=0; i<nb_point; i++)
      Particle = particle_list[i];
      if(Particle belongs to L group of ithread)
         LIndex[m++]=i;
      elseif(Particle belongs to R group of ithread)
         RIndex[n++]=i;
   End
}
```

从上面的描述可以看出该算法与前面提到的几种算法相比具有明显优势。

首先其内存使用量与串行程序相比相差无几，唯一额外的内存耗费来自对物质点建立的索引。其次，该算法实施起来很简单，每个线程只要为自己负责的两组物质点建立索引，并将对物质点的循环重写一遍即可，只是两次循环对象为不同组的物质点。最后，由于采用了一维分区，该算法也比较容易实现普遍适用的负载平衡算法。

3.2.2　物质点更新

物质点更新包括对物质点位置、速度的更新以及物质点应力、应变的更新，均通过对物质点循环实现。由于对不同的物质点更新是独立的，因此可以直接在对物质点更新的循环前添加 OpenMP 的指导性语句：

```
#pragma omp parallel for
```

由于我们的程序采用 C++语言编制，使用了面向对象的思想，类内的数据成员是不能够被 OpenMP 的语句私有化的。如下面的例子：

```
class example{
   public:
   void a, b;
   void func(){
      #pragma omp parallel num_threads(nthread)
      <change a and b>
   }
}
```

a,b 是类 example 的数据成员，example 内的函数成员 func()会修改 a，b。当 func()被并行化的时候就会出现数据竞争。如果 a，b 是临时变量，可以移到函数 func()内，如果 a，b 非临时变量，则这段代码是不能够被并行化的。

MPM3D_OMP 采用了面向对象的思想，在执行物质点更新时各线程均调用了同一个材料类的对象，那么该对象内的数据成员就有可能出现数据竞争，需要把有可能出现数据竞争的临时变量移到函数成员内。

3.3　负载平衡算法

MPM3D_OMP 的物质点更新部分因为直接使用了 OpenMP 提供的指导性语句，可以直接使用 OpenMP 内置的线程调度功能实现负载平衡。

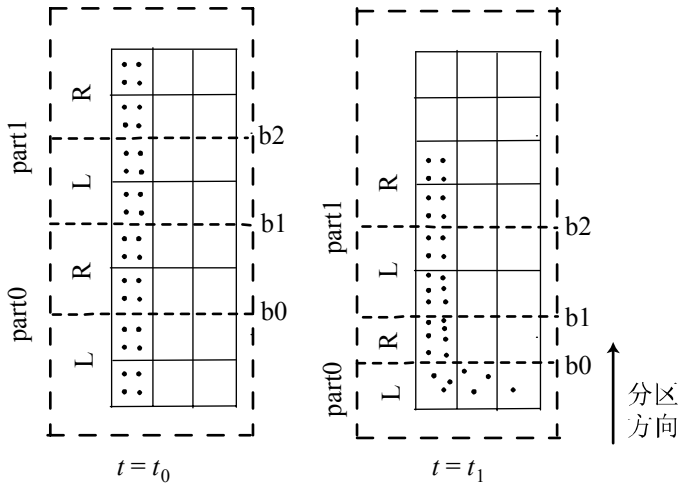


图 3.5　负载平衡策略：泰勒杆碰撞

由于物质点在计算网格内是自由移动的，其分布常常极不均匀，也常常引起背景网格更新部分的负载不平衡。MPM3D_OMP 中背景网格更新部分通过“particle push”策略实现，因此如果能够不断地保持各分区内部的物质点数目平衡，可以预见同时也会达到负载平衡。

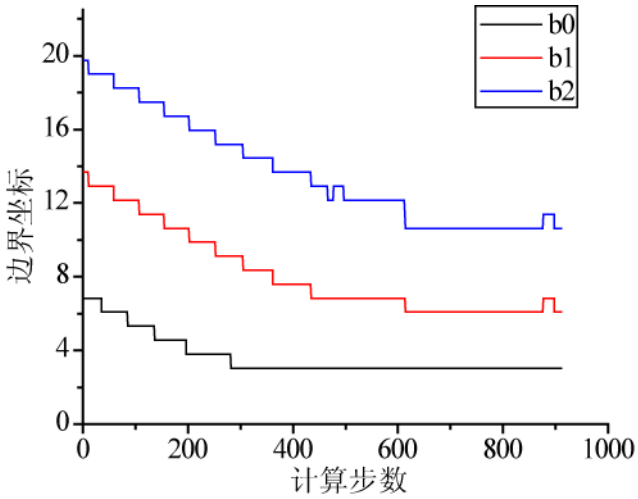


图 3.6　分区边界的变化

网格交替更新法可以很容易地通过重分区来达到负载平衡，只要不断地重

新划分分区边界努力保证不同的子分区含有数目相同或尽量接近的物质点即可。图 3.5 展示了泰勒杆算例的动态分区过程，b1 是两线程之间的边界，b0 和 b2 分别是两线程内左右两分区的边界，此图在两步之内动态地显示了分区边界是如何改善负载平衡的。

图 3.6 表示的是图 3.5 中的泰勒杆算例在每步都重新分区时边界的变化。在计算的后期，分区的边界基本不变。这时如果仍然不断地调用分区算法，会造成计算资源的浪费。因此我们提出了一个阈值，定义为分区内包含的物质点数目的最大值与平均值之比。如果在计算过程中该阈值超过了某一定值，将调用分区算法。阈值的设定需要一定经验的积累，一般设为 0.1~0.3 即可。

3.2.1 小节的分区算法中，分区的数目与线程的数目一致，每个线程被赋予一个分区。这种情况下如果沿多个方向分区，大多数算例都会出现分区间物质点数目不均的现象。全泰勒杆算例可以按照图 3.7 的分区办法使用 2、4 个线程计算，同时严格保持负载平衡。若使用 6 个线程，将不能保持负载平衡。类似地，1/4 泰勒杆算例更难保持负载平衡。因此在物质点沿网格分布不均匀的情况下，沿多个维度的分区算法需要进一步详细讨论。如果分区数目远大于线程数目可以采用任务池策略，即将背景网格划分为很多“片”，每个线程负责一个或多个“片”。不同的“片”里含的物质点数目与背景网格结点数目不同，计算量也不同，因此可以通过在线程间调度“片”以达到负载平衡，但此思想在 MPI 程序中用得较多，很少在 OpenMP 程序中使用。

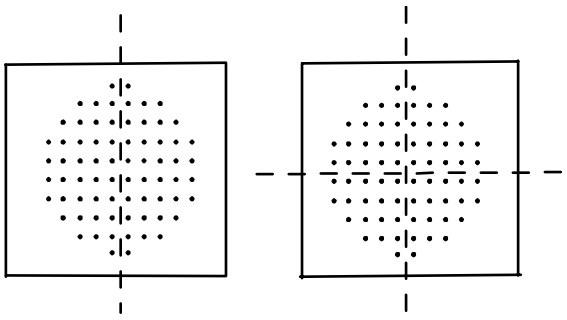


图 3.7　全泰勒杆分区示意(俯视)，实线为网格边界，虚线为分区边界，点为泰勒杆离散的物质点

从上面的讨论可以看出，网格交替更新法有其优势也有其劣势。它比较容易解决负载平衡问题，但一个维度上的背景网格层数毕竟有限，这极大地限制了使用的线程数目，进而影响到程序的可扩展性。

除了前面提到的措施，我们还使用了其他在 OpenMP 程序中常见的办法，如尽量扩大并行区，尽量把所有循环并行化以减弱 Amdahl 规律的影响，交换循环的顺序以充分利用系统缓存，使用“`#pragma omp nowait`”除掉一些不必要的隐式障碍(`barrier`)等，详细请参阅文献[30]。

3.4　算例与结果

为了验证并行程序的效率实施了一系列的数值测试。下面所有的算例在一台八核工作站上进行，其处理器为两个四核 CPU：Xeon 5520，内存为 12GB。操作系统为 CentOS，使用 gcc 4.2.1 编译器。

3.4.1　泰勒杆碰撞

为了测试问题规模对程序效率的影响，我们将泰勒杆离散为三组不同粗细程度的物质点模型。背景网格覆盖区域为(-11.4:11.4， -11.4:11.4，0:26.6)，单位为毫米，三组模型的网格大小分别为 0.76，0.38 和 0.19 毫米。第一组模型含有 34596 个背景网格结点和 21171 个物质点，第二组模型含有 264191 个网格结点和 169376 个物质点，第三组模型含有 2064381 个网格结点和 1346432 个物质点，分别记为工况一、二、三。分区方向沿泰勒杆长，测试得到的总的效率、背景网格更新的效率、物质点更新的效率分别见图 3.8、3.9 和 3.10。

所有结果都启用了 O3 选项。计时采用了 OpenMP 提供的计时函数 `omp_get_wtime()`，并且只统计了计算部分的时间。

结果显示模型越细、物质点数目越多，整个程序的并行效率也越高。这是因为模型越细每个线程持有的背景网格层数越多，更易于实现负载平衡。这一点也可以从背景网格更新部分的并行效率得到验证。当模型固定，使用的线程数增多时，程序的并行效率逐渐减小。这是因为，沿分区方向的背景网格层数是固定的，当使用的线程增多，每个线程持有的背景网格层数就会减少，这样负载不平衡就比较容易发生。图 3.8 显示当使用 4 个线程计算的时候，效率稳定在 80%以上，对于工况三，效率更是高达 90%。物质点更新部分的效率随着线程数增多也在逐渐下降，尽管其效率降低不如背景网格更新的下降显著。这个现象在后面的聚能射流算例中表现更为明显，将在那里探讨效率下降的原因。

本算例揭示了网格交替更新法和负载平衡算法的一些特性，告诉了我们并

行效率随着进程数、模型规模变化的响应。总体上讲，本算例的效果令人满意。在使用八个线程时，总体效率维持在 50%以上，规模较大时效率约为 70%左右。

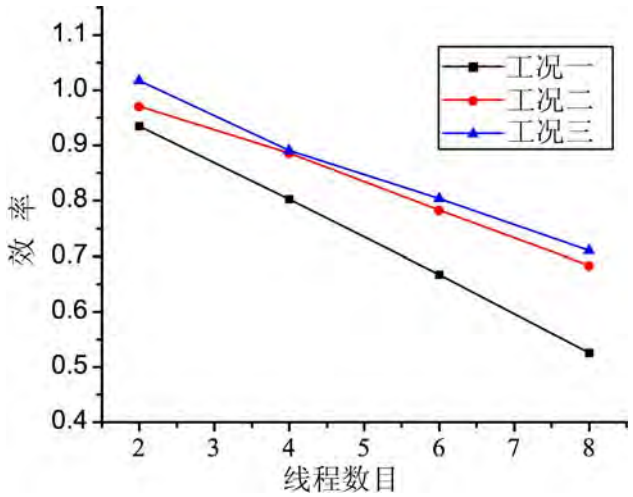


图 3.8　泰勒杆总效率

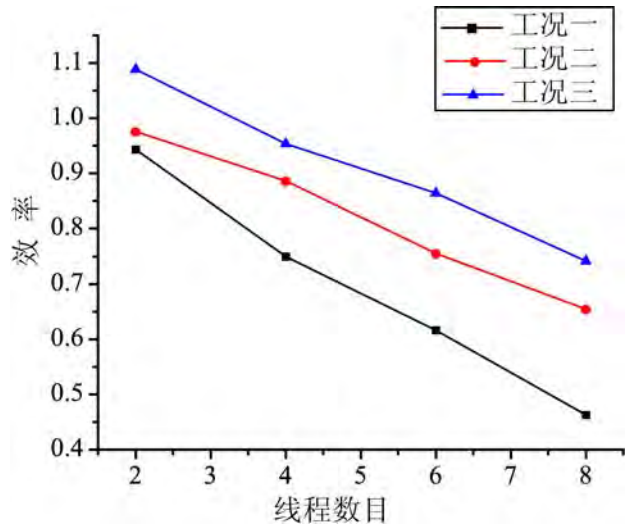


图 3.9　泰勒杆背景网格更新部分的效率

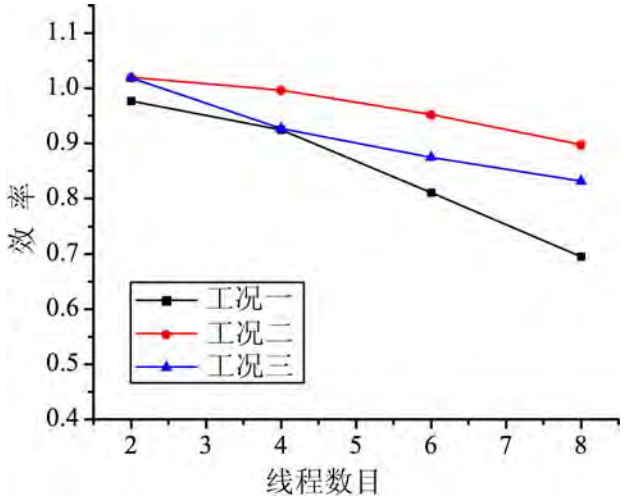


图 3.10　泰勒杆物质点更新部分的效率

3.4.2　二维气体爆炸

二维气体爆炸模拟使用 JWL 状态方程来模拟 TNT 炸药，其参数见表 2.1。为了避免数值断裂，该模拟启用了自适应算法[10]，物质点数目从开始的 5027 增长到最后的 94866。自适应算法将在必要的时候扩展程序的物质点列表，因此会直接在内存中开辟空间。在 OpenMP 标准中，直接操纵内存的代码是不能够并行化的，因此这段代码固有的串行性会在一定程度上降低并行程序的性能。

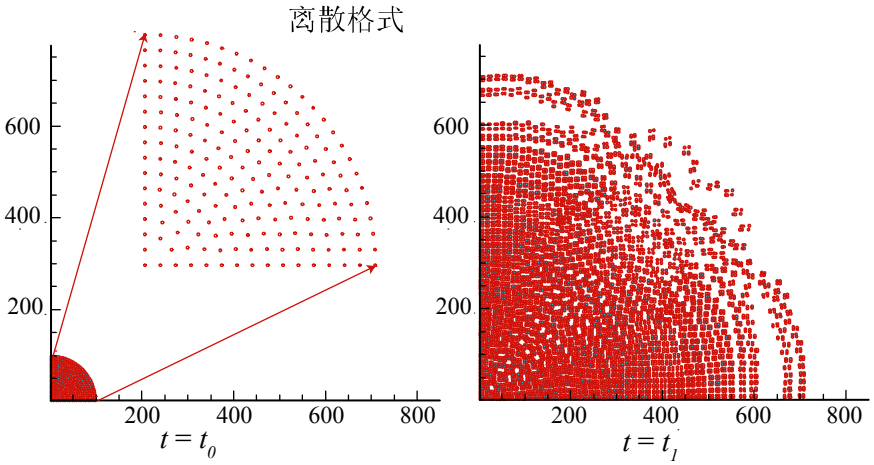


图 3.11　TNT 的离散模型

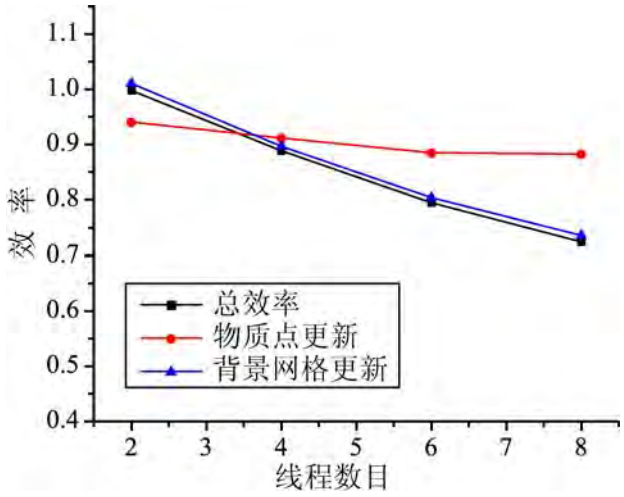


图 3.12　二维 TNT 爆炸的并行效率

由于该问题的对称性，我们采用 1/4 模型。从图 3.11 可以看到此问题的物质点分布极不均匀，沿多个方向分区将会导致严重的负载不平衡。但使用网格交替更新法和与之匹配的负载平衡算法时，图 3.12 显示该算例的并行效率比第一个算例更好，使用八个线程计算时效率约为 75%，充分显示了网格交替更新法的优势。

3.4.3　聚能射流

聚能射流是利用爆炸产生的高速、高能量金属流体，应用于军事中反坦克武器、石油工业中的钻井器械等。模拟中射流罩、炸药与外罩的放置如图 3.13，其中 α=38°。外罩与射流罩都是铜材料，均使用 Johnson-Cook 强度模型与 Grüneisen 状态方程模拟。TNT 使用 JWL 状态方程模拟，参数见表 2.1。

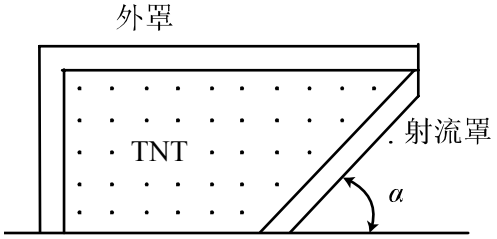


图 3.13　聚能射流的初始构形

起爆后，射流罩被爆炸声称的高温高压气体挤压形成射流。射流的直径很小，速度很快，在物质点数值模拟中极易发生数值断裂，因此模拟也启用了自适应物质点法[10]，物质点的数目从 0μs 的 66912 增加到 20μs 的 112996。20μs 时射流罩的构形见图 3.14，并行效率见图 3.15。相比前两个算例，此模拟的效率较低，使用 8 个线程时物质点更新部分的效率也低至 74%，而前两个算例的物质点更新部分一般都在 90%左右。

本算例效率稍低的原因估计在于缓存不命中。此算例中射流速度极高，到模拟后期，物质点的分布也比较乱。更新物质点需要用到物质点所在的网格结点上的信息。如果连续更新的两个物质点在同一个网格内，则在更新本物质点时仍然能够利用到更新上一个物质点时从内存中取得的网格结点信息，这种情况有利于缓存重用，将减少从内存中读取数据的次数与时间。本次模拟中，物质点的分布极乱，缓存不命中发生的几率相比前两个算例更大，因而效率也稍差。尽管如此，在使用八个线程时效率仍高于 60%。本算例揭示了影响 OpenMP 并行程序的效率的一个非常重要的因素，即缓存不命中。一般来讲，如要让 OpenMP 程序接近理想加速比，需要对串行程序优化做非常多的工作以提高其缓存重用率。

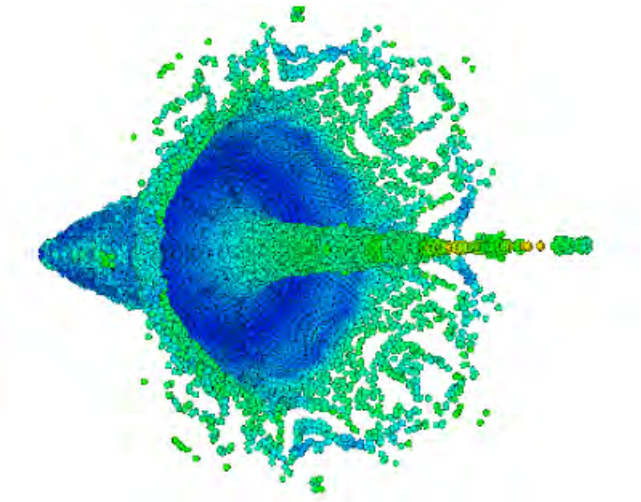


图 3.14　20μs 时射流罩的构形

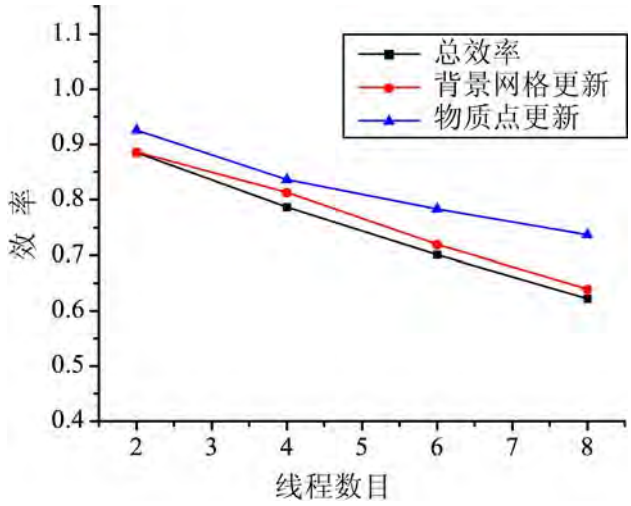


图 3.15　聚能射流的效率

3.5 小结

本章为使用 OpenMP 的物质点并行算法的物质点更新部分提出了网格交替更新法，该办法实施简单、不浪费内存、易于实现负载平衡、适用范围广，在不同种类的算例测试中均取得了良好的效果，使用八个线程时，所测试的算例的效率均在 50%以上。

网格交替更新法中最小的分区单位是背景网格层，粒度较大，容易出现负载不平衡，这在一定程度上影响了并行程序的效率。此外，由于沿选择的维度方向背景网格的层数是有限的，限制了程序使用更多的线程来计算，可扩展性有限。

使用 OpenMP 的并行程序取得理想的并行效率的前提是得到充分优化的串行程序。物质点法的串行程序 MPM3D，在缓存重复使用方面仍有待于改进。可行的优化方案有：采用“particle pull”策略，或者按照物质点的空间位置顺序管理排列物质点列表。

第 4 章　结 论

物质点法的并行算法研究是一个具有挑战性的课题。由于计算量随物质点的移动导致的不确定性，其并行算法研究，相对于有限元格式和欧拉格式更加困难。本文针对两种并行计算模型，即 MPI 和 OpenMP 分别研究了相应的物质点并行算法。使用 MPI 的并行程序采用了静态规则分区的办法，该程序极大地提高了运算规模，且实现了对接触算法和动态网格的并行支持。在使用 OpenMP 的物质点并行算法中提出了网格交替更新法，从根本上解决了负载平衡问题，在各类问题中都表现出了良好的并行效率。由于时间限制，本文的工作还是有很多可以改进的地方。MPI 程序的负载平衡问题有待于深入研究，而 OpenMP 程序的可扩展性需要继续提高。

综合前面两章的讨论以及使用 MPI 和 OpenMP 的并行程序的各自的优缺点，负载平衡问题可以通过两个方案得以解决：一，任务池模式，即将背景网格划分为很多“片”，每个线程负责一个或多个“片”。不同的“片”里含的物质点数目与背景网格结点数目不同，计算量也不同，可以通过在线程或进程间调度“片”以达到负载平衡。二，将 MPI 与 OpenMP 结合起来，在 MPI 的层面上使用粗粒度分区，这时比较容易实现负载平衡。再对每个分区使用 OpenMP 进行细粒度并行计算，通过线程级并行进一步增加程序的并行性。由于当前的并行机群多为 SMP 机群，该方案恰与这种底层硬件结构匹配。但方法二也只是在 MPI 层面上缓解了负载不平衡，没有从根本上解决问题。当使用成百上千或者更多的处理器时，单个结点内的核数是一定的而且常常较小，MPI 的进程数也会相当可观，这种情况下在 MPI 层面通过粗粒度分区实现负载平衡的难度较大。这部分工作也是在今后的研究中需要深化的地方。

最后，本文主要讨论了使用物质点法的并行算法，但上面的讨论对所有的质点网格类方法，如 PIC 方法等，都具有指导意义。与物质点法类似，PIC 方法也采用了质点与网格双重离散，在模拟过程中同样存在负载平衡难以实现的困难，本文在研究过程中也从 PIC 方法的并行算法研究中获得了很多启发。

参考文献

[1] F.H. Harlow. The particle-in-cell computing method for fluid dynamics. Methods in Computational Physics, 1964, 3:319-345

[2] J.U. Brackbill and H.M. Ruppel. FLIP: a method for adaptively zoned, particle-in-cell calcultations in two dimensions. Journal of Computational Physics, 1986, 65:314-343

[3] J.U. Brackbill, D.B. Kothe, and H.M. Ruppel. FLIP: a low-dissipation, particle-in-cell method for fluid flow. Computer Physics Communications, 1988, 48:25-38

[4] D. Sulsky, Z. Chen and H.L. Schreyer. A Particle Method for History-Dependent Materials. Computer Methods in Applied Mechanics and Engineering, 1994, 118:179-196

[5] D. Sulsky, S.J. Zhou and H.L. Schreyer. Application of a Particle-in-Cell Method to Solid Mechanics. Computer Physics Communications, 1995, 87:236-252

[6] D. Sulsky and H.L. Schreyer. Axisymmetric form of the material point method with applications to upsetting and Taylor impact problems. Computer Methods in Applied Mechanics and Engineering, 1996, 139:409-429

[7] X. Zhang, K.Y. Sze and S. Ma. An explicit material point finite element method for hyper-velocity impact. International Journal for Numerical Methods in Engineering, 2006, 66:689–706

[8] W. Hu and Z. Chen. Model-based simulation of the synergistic effects of blast and fragmentation on a concrete wall using the MPM. International Journal of Impact Engineering, 2006, 32:2066–2096

[9] J.E. Guilkey, T.B. Harman and B. Banerjee. An Eulerian–Lagrangian approach for simulating explosions of energetic devices. Computers and Structures, 2007, 85:660–674

[10] S. Ma, X. Zhang, Y. Lian and Xu Zhou. Simulation of high explosive explosion using adaptive material point method, CMES. 2009, 39:101-123

[11] A.R. York, D. Sulsky and H.L. Schreyer. The material point method for simulation of thin membranes. Int. J. Numer. Meth. Engng, 1999, 44:1429-1456

[12] S.G. Bardenhagen, J.U. Brackbill, D. Sulsky. The material-point method for granular materials. Comput. Methods Appl. Mech. Engrg, 2000, 187: 529-541

[13] S.G. Bardenhagen, J.E. Guilkey, K.M. Roessig, J.U. Brackbill, W.M. Witzel and J.C. Foster. An Improved Contact Algorithm for the Material Point Method and Application to Stress Propagation in Granular Material. CMES, 2001, 2(4):509-522

[14] J. A. Nairn. Material Point Method Calculations with Explicit Cracks. CMES, 2003, 4(6): 649-663

[15] W. Hu, Z. Chen. A multi-mesh MPM for simulating the meshing process of spur gears. Computers and Structures, 2003, 81:1991-2002

[16] P. Huang, X.Zhang, S.Ma and X. Huang. Contact algorithms for the material point method in impact and penetration simulation. IJNME, 2011, 85(4):498–517

[17] H. Jin, M. Frumkin and J. Yan. The OpenMP Implementation of NAS parallel Benchmarks and Its Performance. 1999, NAS Technical Report NAS-99-011

[18] J. Hoeflinger, Prasad Alavilli, Thomas Jackson and Bob Kuhn. Producing scalable performance with OpenMP: Experiments with two CFD applications. Parallel Computing, 2001, 27:391-413

[19] Steven G. Parker. A component-based architecture for parallel multi-physics PDE simulation. Future Generation Computer Systems, 2006, 22:204–216

[20] Steven G. Parker, James Guilkey and Todd Harman. A component-based parallel infrastructure for the simulation of fluid–structure interaction. Engineering with Computers, 2006 22:277–292

[21] Petr Krysl, Zdenek Bittnar. Parallel explicit finite element solid dynamics with domain decomposition and message passing: dual partitioning scalability, Computers and Structures. 2001 79:345-360

[22] George Karypis, Vipin Kumar. Metis home page.
http://glaros.dtc.umn.edu/gkhome/views/metis

[23] Sandia Laboratory. Chaco homepage. http://www.sandia.gov/~bahendr/chaco.html

[24] J.G. Malone, N.L. Johnson. A parallel finite element contact/impact algorithm for non-linear explicit transient analysis: Part II-parallel implementation. Intl. J. Num. Methods Eng, 1994, 37: 591-603

[25] Kevin Brown, Steve Attaway, Steve Plimpton, Bruce Hendrickson. Parallel strategies for crash and impact simulations. Comput. Methods Appl. Mech. Engrg, 2000, 184:375-390

[26] C. Othmer, J.Schüle. Dynamic load balancing of plasma particle-in-cell simulations: The taskfarm alternative. Computer Physics Communications, 2002, 147:741–744

[27] Steven J. Plimpton, David B. Seidel, Michael F. Pasik, Rebecca S. Coats, Gary R. Montry. A load-balancing algorithm for a parallel electromagnetic particle-in-cell code. Computer Physics Communications, 2003, 152:227–241

[28] P. Huang, X. Zhang, S. Ma and H.K. Wang. Shared Memory OpenMP Parallelization of Explicit MPM and Its Application to Hypervelocity Impact. CMES, 2008, 38(2):119-147

[29] OpenMP Architecture Review Board. Version 3.0. OpenMP Application Program Interface. May, 2008

[30] Barbara Chapman, Gabriele Jost and Ruud van der Pas. Using OpenMP-Portable shared memory parallel programming. Cambridge, Massachusett, London, England: The MIT Press, 2007

[31] George Stantchev,William Dorland, Nail Gumerov. Fast parallel Particle-To-Grid interpolation for plasma PIC simulations on the GPU. J. Parallel Distrib. Comput, 2008, 68: 1339–1349

[32] M. Berger and S. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. IEEE Trans. Computers, 1987, C-36:570-580

[33] Message Passing Interface Forum. Version 2.1. MPI: A Message-Passing Interface Standard. Jun 23, 2008

[34] William Gropp, Ewing Lusk, and Anthony Skjellum. Using MPI: Portable Parallel Programming with the Message-Passing Interface. US: MIT Press, 1999

致　谢

衷心感谢导师张雄教授对本人的精心指导，他严谨的学术作风将使我终生受益。

感谢计算动力学研究室的全体同学的热情帮助。

感谢在背后默默地支持我的家人。

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____日　期：_____

个人简历、在学期间发表的学术论文与研究成果

个人简历

1986 年 3 月 5 日出生于河南省清丰县。

2004 年 9 月考入北京大学力学与工程科学系，2008 年 7 月本科毕业并获得工学学士学位。

2008 年 9 月免试进入清华大学航天航空学院攻读力学硕士至今。

发表的学术论文

[1] Yantao Zhang, Xiong Zhang, Yan Liu. An Alternated Grid Updating Parallel Algorithm for Material Point Method Using OpenMP. CMES, vol.69, no.2, pp.143-165, 2010