目录

[1 引 言 3](#_Toc482576144)

[1.1 课题背景 3](#_Toc482576145)

[1.2 研究意义 4](#_Toc482576146)

[1.3 论文组织结构 4](#_Toc482576147)

[2 相关研究和理论技术背景 6](#_Toc482576148)

[2.1 国内外相关研究 6](#_Toc482576149)

[2.1.1 分子动力学方法相关研究 6](#_Toc482576150)

[2.1.2 并行计算机的发展 6](#_Toc482576151)

[2.2 理论背景 7](#_Toc482576152)

[2.2.1 MD模拟基本思想 7](#_Toc482576153)

[2.2.2 空间分解 10](#_Toc482576154)

[2.3 技术背景 10](#_Toc482576155)

[2.3.1 并行计算 10](#_Toc482576156)

[2.3.2 MPI并行编程模型 11](#_Toc482576157)

[2.3.3 各种通信？？？ 13](#_Toc482576158)

[2.3.4 并行程序性能评价 13](#_Toc482576159)

[3 MD模拟空间分解并行化实现 15](#_Toc482576160)

[3.1 MD模拟过程分析 15](#_Toc482576161)

[3.1.1 原子信息存储 15](#_Toc482576162)

[3.1.2 计算相互作用力 16](#_Toc482576163)

[3.1.3 更新原子位置和速度 17](#_Toc482576164)

[3.2 MD模拟程序并行化实现 18](#_Toc482576165)

[3.2.1 任务划分 18](#_Toc482576166)

[3.2.2 数据结构 20](#_Toc482576167)

[3.2.3 通信分析 20](#_Toc482576168)

[3.2.4 宏观量提取 21](#_Toc482576169)

[4 MD模拟并行化通信策略研究 24](#_Toc482576170)

[4.1 基本的点对点通信 24](#_Toc482576171)

[4.2 两种MD并行程序中的通信优化策略 25](#_Toc482576172)

[4.3 通信聚合和单边通信结合的通信优化策略 29](#_Toc482576173)

[4.4 改进的聚合-单边通信优化策略 34](#_Toc482576174)

[5 实验结果分析 38](#_Toc482576175)

[6 结 论 39](#_Toc482576176)

1. 引 言
   1. 课题背景

[introduction\_to\_md\_标记]

自20世纪电子计算机出现以来，计算机模拟技术就被应用在各个专业的研究领域。计算机模拟指的是以数学模型为基础，通过计算机计算来研究某种特定系统的行为的一种方法。它已经成为计算物理、生物化学、工程领域和社会科学等多个专业领域的一种高效直观的研究方法。在各类研究过程中，实验总会消耗一定的时间和资源，由于这两个因素的限制，很多真实的实验往往很难得到实施，因此人们就提出利用计算机工作的高效性来对这些实验进行模拟。实践证明，计算机模拟已经对各科研领域做出了重大贡献。

分子模拟就是一种计算机模拟方法。分子模拟从分子体系的结构和分子间微观相互作用的角度来理解其性质，它是一种和传统实验截然不同的方法，可以说是传统实验方法的补充。目前主要的分子模拟方法有蒙特卡洛、分子力学和分子动力学方法等。其中，蒙特卡洛与分子动力学方法最为常见。蒙特卡洛方法在模拟一些晶格相对规则的物质上显得更为成功，而分子动力学则更多地用来模拟大量不规则运动的微观粒子。此外，虽然分子动力学可模拟的时间跨度较短，在真实的世界中，一些时间跨度大的过程如材料的老化等，就超越了分子动力学方法的能力，但是在描述的精确度上它比蒙特卡洛更加精确。在一些领域实际的科学研究中，有很多现象很难被直接观测到，这就给科研人员带来了诸多不便，从而间接地造成了某些科研的瓶颈。分子模拟技术使得科研人员能获取或者直接观测到微观世界中的粒子行为，既方便了科学研究，使不可能变为可能，也突破了时间、资源和经济因素方面上的限制。

分子动力学方法是一种分子模拟技术。它是一种通过数值模拟技术来模拟多粒子系统动态行为的方法，研究对象是分子或分子体系结构与性质。它的模型基础是经典力学模型，体系中粒子的运动可以看成是符合牛顿运动定律的，通过计算运动方程，得到体系中原子或分子各个时刻的位置和速度等微观运动状态，再利用某些积分统计计算，就不难获得模拟物质对象宏观上的特性。如今，MD方法已被用来研究生物、化学和材料领域的物质特性分析。以材料领域为例，从空间尺度的角度来看，材料性能的模拟可分级为原子、微观、介观和宏观。分子动力学就是一种在微观尺度上进行模拟的方法。

* 1. 研究意义

分子动力学（MD）模拟是计算模拟，必定带来一定的计算量。从空间角度和时间角度分别来分析其计算量：1）从空间角度来说，MD模拟是在原子或者分子的尺度上进行的。我们知道，在真实的世界中，即使是物质非常小的一部分，其中也包含了数以亿计的分子或者原子。所以说，MD方法的模拟对象的数量是极其庞大的，对计算性能就必然会有很高的需求。2）从时间角度来说，由于MD方法需要一定的精确性和稳定性，其整个模拟迭代过程所使用的步长通常在飞秒（即10-15秒）级，而它模拟的是真实世界中一些物质的变化，时间跨度通常比飞秒要大得多。因此，MD模拟过程的总迭代次数是非常巨大的。这就是导致MD模拟计算量需求大、成为一个典型的计算瓶颈问题的两点主要原因。可以看出，MD模拟方法对计算机的性能需求是无穷尽的，对于一些稍微复杂点的模拟体系，即使使用巨型计算机，也会消耗人们大量的时间。

为了提高计算性能，除了单方面提高计算机硬件的能力外，并行计算就是另一个值得深入研究的方向。并行计算将原任务拆分成一些子任务，并合理分配至各个处理器，各处理器同时地进行计算，再通过相互之间的协作与通信，共同形成一个有机的整体得到最终的计算结果。并行计算是突破单处理器计算性能限制的一个重要技术。另外，从并行的角度看，对它的研究主要是对计算节点协调、通信机制的研究，其发展也主要在这两个方面上体现出来。因此，对分子动力学方法的并行化研究及其通信机制的研究意义重大。

* 1. 论文组织结构

本论文共六章，整体提纲如下：

第一章主要阐述了本课题的背景、研究意义，研究内容？

第二章对分子动力学和并行计算的国内外研究现状进行了总结，同时对分子动力学的理论基础和并行计算的技术背景做了介绍。

第三章主要分析了MD模拟方法的并行化实现方法。

第四章针对MD模拟并行化程序的通信机制，研究了两种已有的通信策略，提出了将两种策略结合的方法，并在此基础上对其进行了改进，形成了一种新的通信策略。

第五章主要分析了实验结果，对前面的方法与策略进行了分析。

第六章是结论，是对本课题研究成果的总结。

1. 相关研究和理论技术背景
   1. 国内外相关研究
      1. 分子动力学方法相关研究

MD模拟方法是通过数值模拟技术来模拟多粒子系统动态行为的一种方法。它被广泛地用来研究固体、液体和气体物质的一些宏观特性。

Wainwright和Alder[]为了研究气液体在硬球模型下的状态方程，在1957年通过计算机模拟，分析了由一些刚性小球组成的分子体系的运动，因此成为了通过MD模拟方法来研究物体宏观特性的开创者。Lees[]等人在1972年将MD方法用在了速度存在变化的非平衡体系中。1980年，Andersen[]等人开创了恒压分子动力学方法，并对其进行了实验验证。Gillan[]等在1983年将MD方法用在了温度存在变化的非平衡体系中，与Lees等人提出的方法共同形成了非平衡体系下的分子动力学方法体系。1984年，Nose[]等人提出了恒温分子动力学方法，并对其进行了实验验证，与Andersen等人提出的方法相呼应。1985年，Hoover等人改进和优化了Nose提出的方法，提高了此方法的稳定性和可靠性。同样在1985年，Car[]等人针对金属和半导体，研究了它们的势函数，提出将电子论和分子动力学方法结合，开创了一种新方法，即第一原理分子动力学方法。1991年，Cagin[23]在提出了巨正则系综分子动力学，该方法则主要处理吸附问题。其实，从1980年开始，随着多体势函数的开创和计算机技术的发展，MD模拟方法体系越来越完善。如今，MD方法已经覆盖了诸多的专业领域与应用场景，各个领域的研究人员都利用MD方法更方便、更直观地进行科学研究。

* + 1. 并行计算机的发展

[并行计算导论——张林波]

早在20世纪40年代就出现了现代计算机，开始了计算机从串行计算时代到并行计算时代的一个发展历程。在20世纪60年代，磁芯存储器和晶体管相关技术的发展使得并行计算机逐渐出现在人们的视野中，随之而来的是流水线技术的开创。这使得并行计算机的性能大幅度提高，也推动了计算机技术发展。

在之后的迅速发展历程中，先是虚拟存储和缓存机制的开创，再到半导体存储器、集成电路的大规模采用，对向量计算机指令集的优化和完善，这些都对并行计算机的发展起到了里程碑的作用。从20世纪80年代起，微处理器技术的研究就得到了广泛关注，共享存储的多处理器系统被大量采用。90年代初，斯坦福大学的DASH计划实现了分布式共享存储器的缓存一致性，IEEE也为此提出了统一的协议标准。

如今的并行计算机已经进入了一个新时代，大量的高性能计算机厂商都有能力制造出高性价比的并行计算机，并行计算机正在逐渐向人们的生活进行靠拢，并行计算的应用发展也处于一个繁荣昌盛的局面。

* 1. 理论背景
     1. MD模拟基本思想

分子动力学方法[]是通过数值模拟技术来模拟多粒子系统动态行为的一种方法，研究对象是分子或分子体系结构与性质。它的模型基础是经典力学模型，体系中粒子的运动可以看成是符合牛顿运动定律的，通过计算运动方程，得到体系中原子或分子各个时刻的位置和速度等微观运动状态，利用某些积分统计计算，就不难获得模拟物质对象宏观上的特性。

1. MD模拟的原理

MD模拟的有效性基于两个假设[]：1）牛顿运动定律在体系中所有粒子的运动上均适用。2）叠加定理在体系中粒子相互作用上均适用。

假设一个原子数为N的模拟体系，原子*i*的位置矢量为**r***i*，质量是m*i*，速度是**v***i*，受到的力是**F***i*。原子*i*和原子*j*的相互作用力为*fij*，势能为*uij*，距离为r*ij*=| **r***i* - **r***j* |。

以原子*i*为例，在确定模拟体系中粒子的初始位置和速度后，我们可以求出原子*i*的势能*Ui*：



整个系统的总势能*U*则为：



而力是势能函数关于位移的负梯度，由此可以得到原子*i*受到的力**F***i*：

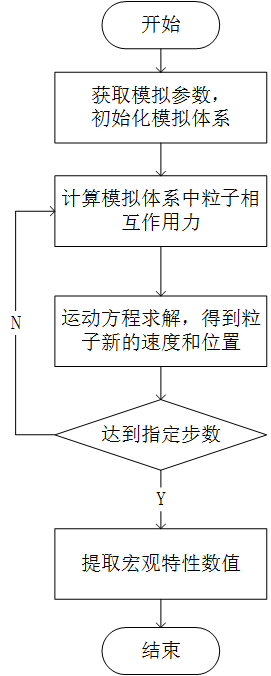


在经典力学模型的基础上，粒子运动符合牛顿运动定律，故可以对原子*i*在*t*时刻的速度**v***i*(*t*)和位置**r***i*(*t*)进行求解：





总结起来，整个模拟过程主要为四步：1）初始化模拟体系，包括粒子速度和位置等。2）计算粒子的相互作用力。3）根据经典力学模型，得到新的速度和位置。4）待体系稳定，进行宏观量的计算。其中，步骤2和步骤3形成一个循环迭代体，对应的流程图如图：



1. 势函数

在MD模拟过程中，计算原子受到的作用力是主要的部分，为了达到这个目的，往往需要先求出原子的势能，研究人员为了表达原子的势能与其所在位置的关系，引入了势函数[分子动力学中势函数研究—陈强]。最早的势函数为对势，在初期，MD模拟采用的都是对势模型。对势又包括间断对势和连续对势，连续对势作为主要的模型被用在MD模拟中。常见的对势有Lennard-Jones(LJ)势[]，Morse势[]，Born-Mayer势等。经过多年的发展，有了多体势，典型的多体势有EAM势[]和Finnis-Sinclair(F-S)势[]。

LJ，Morse 公式，各势函数特性。。。。。

1. 运动方程的求解

在计算得到了粒子所受的作用力后，紧接着做的工作则是根据力来计算粒子下一步的新速度和位置。解决这一问题的算法主要有Verlet算法[]、速度Verlet算法[]、Leap-frog算法[]和Beeman算法[]等。这里主要对Leap-frog算法进行介绍。

1. Leap-frog算法：

假设在时刻*t*，粒子的位置是r(*t*)，速度是v(*t*)，受力是F(*t*)，质量为m，模拟过程迭代的步长为∆*t*，并且****已知。

1. 根据F(*t*)，能得到时刻*t*的加速度a(*t*)：



1. 由a(*t*)计算****：

****

1. 由****计算****：

****

1. ****则可由平均值得出：

****

1. 由步骤3得到的****，又可得到****，转到步骤1进行循环迭代。
   * 1. 空间分解
   1. 技术背景
      1. 并行计算

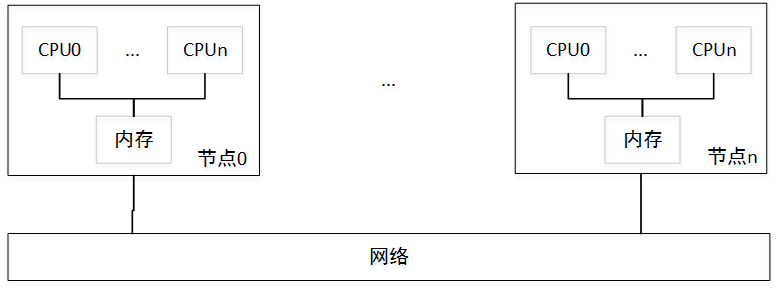
[高性能并行计算——迟学斌]

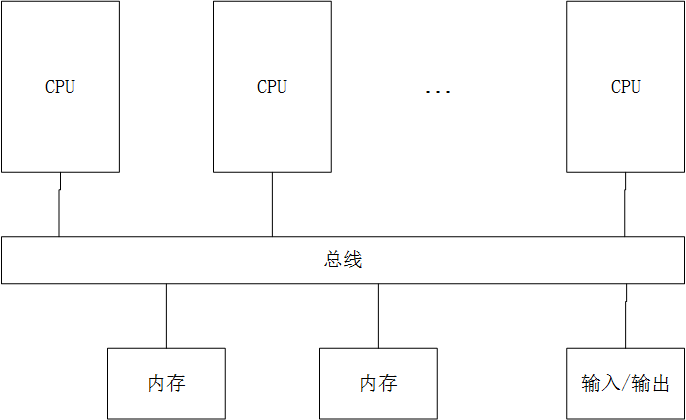
并行计算[]是为了解决单处理机的计算性能问题，由于物理层面上的限制，单处理机在硬件方面是存在瓶颈的，在一些复杂问题的求解中，并行计算则显得必不可少。其基本思想是将一个规模比较大的计算问题分割为一些小计算问题，然后将各个小任务合理分配至各个处理单元，它们同时地进行计算，再通过相互之间的协作与通信，共同形成一个有机的整体得到最终的计算结果。并行计算相当于有多个单处理机同时地服务于同一个计算问题，因此相比于串行计算来说，其时间消耗必然会大大减少。

从20世纪计算机问世以来，并行计算经历了从概念的提出到体系完善的一个发展过程。并行计算所带来的日益增长的计算性能，对人类生活和科学研究的影响都越来越大。如今的并行计算机进入了一个新的时代，在并行计算机的研制技术上，我国已然在世界名列前茅。如曙光、天河和神威等超级计算机都为国家科研打下了物质基础。

* + 1. MPI并行编程模型

从存储方式的角度来看，并行计算系统可分为两大类：分布式存储系统（DMP）和共享内存系统（SMP）。如图:

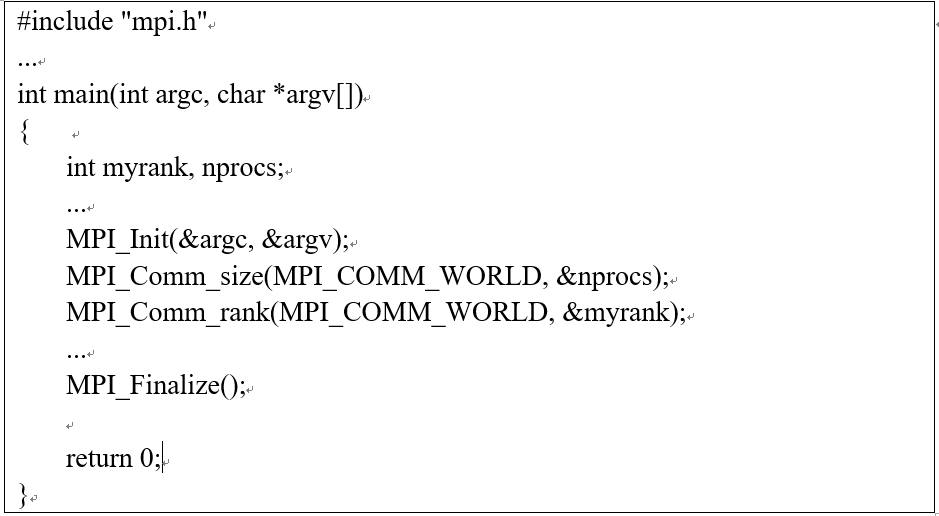




在DMP结构下的并行编程模型中，MPI（。。。。）[]使用最广泛。它是人们提出来的一种编程标准，旨在提高以消息传递为基础的并行程序的效率、一致性、可移植性和可拓展性。

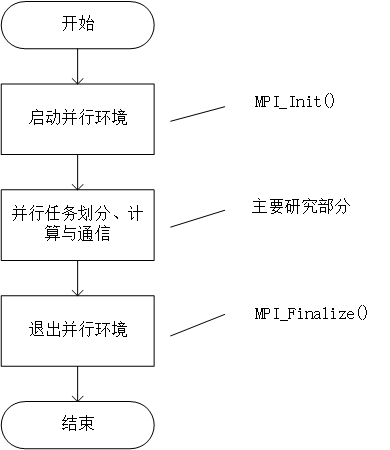
在MPI标准中，定义了一些方便用于进程间消息传递的接口。如今，一些通用的MPI系统已经被研发出来，开源系统中比较著名的有MPICH[]和LAM MPI[]。其中，MPICH使用最广泛，目前许多的机群系统上也主要是以MPICH为并行环境。它提供的接口能容易地被多种语言调用，如C、C++和Fortran语言。[并行计算导论——79]早期的MPI实现是基于MPI 1.0的，它是MPI标准的第一个版本。而在1998年，MPI 2.0版本推出，相比于第一版，它增加了一些功能，如动态进程管理、单边通信、线程安全和并行I/O等。

MPI作为一种编程模型，实际应用中，程序必然会有与模型对应的典型结构。以C语言为例，MPI程序的结构一般如图：



MPI中所有函数接口及常量都以MPI\_开头。在使用MPI函数前，程序中必须包含头文件mpi.h。此外，在所有其它MPI函数调用的开头和结尾，必须分别有MPI\_Init()与MPI\_Finalize()这两个函数，前者用来初始化MPI，后者则是退出MPI系统。最常用的函数有MPI\_Comm\_size()和MPI\_Comm\_rank()，它们分别用于获取指定通信域中进程的总数量和本进程对应的标识号。除图中展示的函数之外，MPI还提供了大量编程接口，如MPI\_Send()和MPI\_Recv()，它们用于进程间的点对点通信，MPI\_Send()将数据信息发送至指定的进程，MPI\_Recv()则是接收指定进程发送来的数据信息。

从程序总体结构的角度来看，MPI并行程序分为三个部分：1）启动并行环境。包括通信器和通信环境的启动。2）并行任务划分、计算和通信。这部分是我们主要的研究对象，研究如何合理地分配任务和进行高效的通信。3）退出并行环境。在退出后，就不能再调用任何MPI函数了。总体结构可用图来表示：



* + 1. 各种通信？？？
    2. 并行程序性能评价

实际编程中，在完成了并行程序的编写后，一个必需的工作就是做程序的性能做出评价。目前已有几种评价指标[]：

1. 并行加速比和效率

假设程序在串行的情况下，单处理器完成任务所需的时间为T。同样的问题，在经过并行化后，由N个进程并行完成计算所需的时间为T*N*，则可计算其加速比S*N*和并行效率E*N*：





通俗地来说，加速比指的是计算任务在引入了N个进程并行计算后，与其串行程序相比，所消耗时间的减少倍数。并行效率则指的是此N个进程或处理器资源的利用程度。

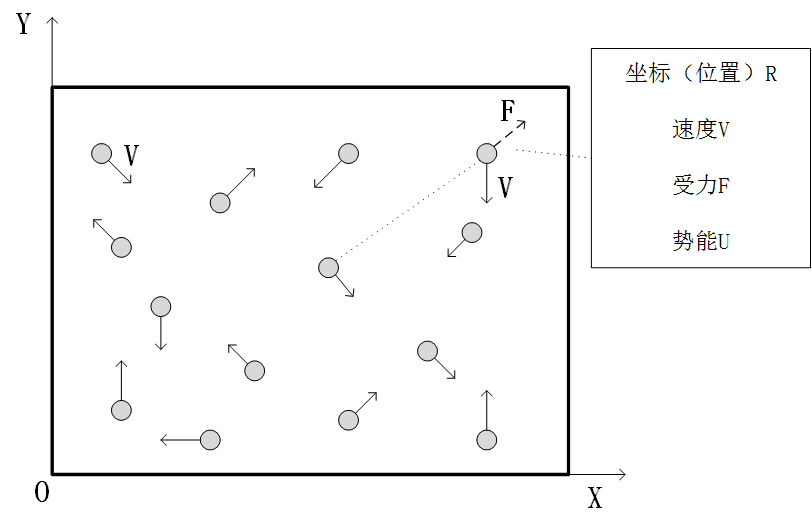
2）。。。？

1. MD模拟空间分解并行化实现
   1. MD模拟过程分析

在2.2节中已经介绍，MD整个模拟过程主要为四步：1）初始化模拟体系，包括粒子速度和位置等。2）计算粒子的相互作用力。3）根据经典力学模型，得到新的速度和位置。4）待体系稳定，进行宏观量的计算。其中，步骤2和步骤3形成一个循环迭代体，迭代一定次数后退出循环。

* + 1. 原子信息存储

MD模拟是在微观尺度上进行的，其对象是原子，因此必须记录原子的行为和状态信息。如图，将一个模拟体系赋予坐标系（二维体系），则每一个体系中的原子都有唯一的坐标值即位置；每一个原子时刻都在运动，所以有唯一的速度值；其次，体系中两两原子间都有相互作用，这带来了原子的受力和势能，也需要记录。此外，为了防止受力的重复计算，还需要一个能够唯一标识原子的序号。故得到存储原子信息的结构体Atoms（三维体系），如表：

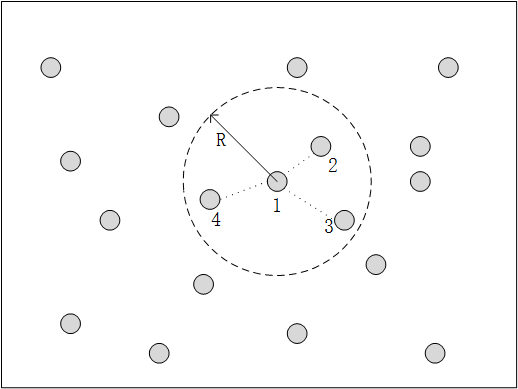


Atoms

|  |  |
| --- | --- |
| 位置pos | X方向pos[0] |
| Y方向pos[1] |
| Z方向pos[2] |
| 速度vel | X方向vel[0] |
| Y方向vel[1] |
| Z方向vel[2] |
| 受力force | X方向force[0] |
| Y方向force[1] |
| Z方向force[2] |
| 势能u | |
| 序号id | |

* + 1. 计算相互作用力

在体系初始化完成后，就需要对体系中每个原子的受力进行计算。对于MD方法的串行程序，其计算方法是遍历体系中的每个原子，对每个原子，又需要计算它与其它所有原子的受力叠加，所以该算法的复杂度为O(N2)。其计算量大，为了减少计算量，。。。引入了截断半径[]这一概念。如图，在计算原子1时，我们认为只有与原子1距离不超过截断半径R的其他原子，才与原子1有相互作用，如图中的原子2、3、4，这就减少了大量的计算。



在串行程序中，计算原子受力的算法如下：

|  |
| --- |
| **MD串行程序计算作用力伪代码** |
| **atoms：**原子信息结构体数组  **atomnum：**总原子数量  **r0：**截断半径 |
| 1: for(int i=0 ;i < atomnum;i++)  2: for(int j=0 ;j < atomnum;j++)  3: **if( atoms[i].id >= atoms[j].id )**  4: continue;  5: r = ( atoms[i].pos- atoms[j].pos ) \* ( atoms[i].pos - atoms[j].pos );  6: r = sqrt(r);  7: **if( r > r0 )** //距离大于截断半径  8: continue;  9: u = calu(r); //计算势能  10: f = calforce(r); //由势函数的梯度函数求得  11: atoms[i].force - = f;  12: atoms[j].force + = f;  13: atoms[i].u + = 1/2\*u;  14: atoms[j].u + = 1/2\*u; |

其中，第3行用于判断原子i和原子j是否已经参与计算，防止重复计算。第7行则判断两原子的距离是否大于截断半径，若大于则不计算。第9、10行则利用在。。节介绍的势函数来计算势能和作用力，11~14行对原子的受力和势能进行叠加。

* + 1. 更新原子位置和速度

在2.2.1中，对Leap-frog算法进行了介绍。我们利用此算法来更新原子的坐标和速度，伪代码如下：

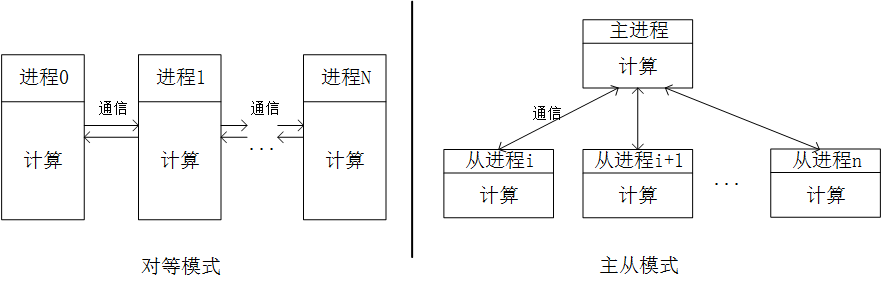
|  |
| --- |
| **MD更新原子坐标、速度伪代码** |
| **atoms：**原子信息结构体数组  **atomnum：**总原子数量  **n：**当前迭代次数  **N：**总迭代次数  **m：**原子的质量  **dt：**迭代步长 |
| 1: while( n < N )  2: {  3: for(int i=0 ;i< atomnum; i++)  4: {  5: atoms[i].vel + = atoms[i].force/ m \* 0.5 \* dt;  6: atoms[i].pos + = atoms[i].vel \* dt;  7: }  8:  9: **ComputeForce();** //重新计算原子的受力，即F(t+dt)  10:  11: for(int i=0 ;i< atomnum; i++)  12: atoms[i].vel + = atoms[i].force/ m \* 0.5 \* dt;  13: } |

其中，第9行ComputerForce()指的是3.1.2中描述的算法，即计算相互作用力。

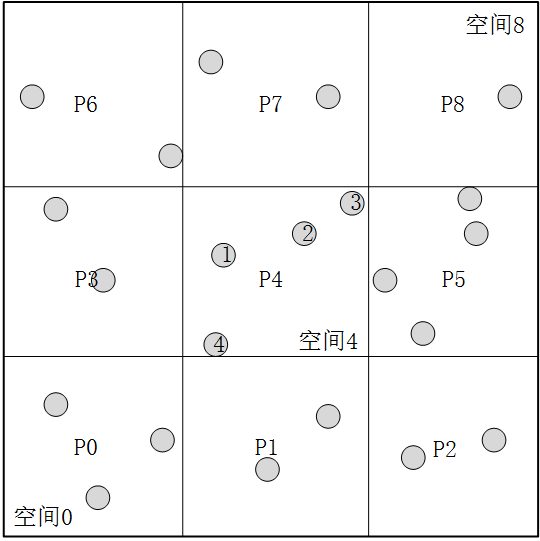
* 1. MD模拟程序并行化实现
     1. 任务划分

把一个程序进行并行化，通常有两种任务划分方式，分别是按数据划分和按功能划分。数据划分指的是将需要处理的总数据划分成一些子任务，将数据分割后再分配到每个进程；功 能划分指的是按程序各部分功能的不同来划分给不同进程，每个进程完成不同的功能。出于对并行程序效率的考虑，一般我们要尽可能地让各进程的相互依赖性降低，避免因为进程间的相互等待而影响整个并行程序的效率，而在功能划分方法中，程序各个功能模块一般都存在相关性和依赖性，一个功能模块的启动可能需要等待另一个功能模块的完成。所以，功能划分一般会给并行程序带来一定的限制，我们采用数据划分的方法来进行并行化。

在MPI编程中，程序的基本设计模式有两种：对等模式和主从模式，大部分的MPI程序都是基于这两种设计模式的。对等模式中，所有进程的任务和工作量大致相同，各自完成自己的计算部分，所有进程的地位大致相等；主从模式中，则存在主从进程之分，由主进程控制和管理各个从进程，或者将数据从各个从进程收集和向各个从进程散发。对等模式和主从模式的比较如图。在MD模拟中，将数据划分后，各个进程的计算并没有很大差别，故可以采用对等模式进行并行化。

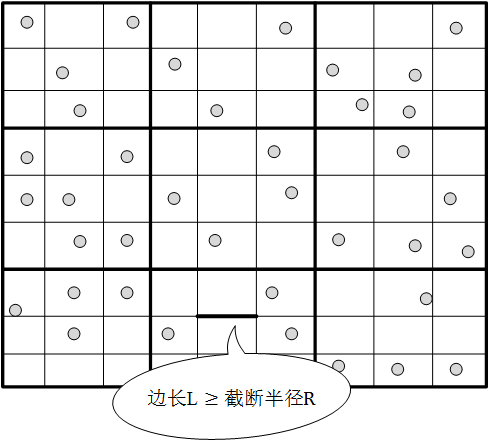


采用数据划分和对等的任务划分方法和设计模式，将整个模拟体系在实际的物理层面上平均划分为N个子空间，N为指定的并行进程数，每个进程只负责其对应子空间内部原子的计算。以二维体系为例，如图，将整个体系平均分成了9个子空间（空间0~8），每个子空间对应一个进程。以进程P4来举例，P4进程只计算坐标分布在空间4的原子1、2、3、4。如此，每个进程只计算自己负责的原子，在原子分布均匀的情况下，这就大致把计算量降低为原任务的1/N，N个进程同时进行计算，时间将大大减少。



* + 1. 数据结构

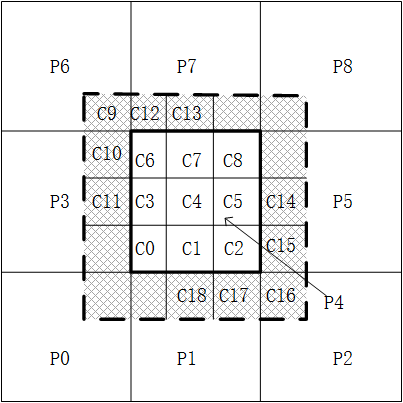
MD模拟中常见的数据结构有两种：Neighbor List[]和Linked Cell[]。为了方便管理原子的数据信息，同时减少计算量，在并行程序中采用Linked Cell数据结构。但与串行程序不同的是，并行程序中元胞（cell）需要根据进程对应子空间的大小来进行划分。在划分元胞时，值得注意的是，需要保证元胞的边长大于等于截断半径，即满足L≥R。因为在满足L≥R的条件下，我们在计算某一元胞中原子的受力时，只需要关注与其相邻的元胞中的原子坐标信息，而不需要其它的原子坐标信息，这就省去了大量不必要的进程间通信。以二维体系为例，如图，将模拟体系划分成了9个进程（粗实线分割），每个进程内又划分了9个元胞（细实线分割），这些元胞以链表的形式联系起来，每个进程以元胞为单位进行原子受力的计算。



计算力代码变化？

* + 1. 通信分析

在3.2.2中，我们对数据结构做了说明。接下来，将对基于这种数据结构的并行化程序做通信分析，说明通信量所在。以二维体系为例，如图，我们将体系分成9个子空间，分别分配给9个进程（图中P0~P8），每个进程划分9个元胞（图中C0~C8）。通信分为两个部分：第一，进程P4在计算其原子受力时，以元胞为单位，依次计算C0~C8中原子的受力，但是在计算边界元胞原子受力时，需要其它进程的原子坐标信息。比如，在计算元胞C2原子受力时，共需要C1、C2、C4、C5、C14~C18的原子坐标数据，其中C14、C15、C16、C17、C18的数据均来自于其它进程，这就需要与其它进程发生通信，接收这部分数据，同时发送本进程边界元胞中原子数据给其它进程。第二，在模拟过程中，原子是不断运动的，本进程中的原子很可能在某一时刻运动到了其它进程内。例如，进程P4的元胞C2中某原子A由于不规则运动进入了进程P2所计算的范围内，这时就需要将原子A的所有数据（包括坐标、速度、受力和势能等）发送给进程P2，进程P2接收后将原子A加入，最后再从进程P4中删除原子A的数据。因此，这也占用了部分的通信量。故可用一部分存储空间（图中阴影部分）来存储来自相邻进程的原子数据，同时也可用于存储运动至相邻进程的原子的数据，这部分区域可以称为通信区。



更新速度位置代码变化?

* + 1. 宏观量提取

在体系达到平衡（即原子总势能比较稳定）后，我们可以开始提取感兴趣的宏观量，下面以温度这一宏观值为例子，说明在MD并行化中提取宏观量的方法。

要计算体系的宏观值，需要一个从微观到宏观转换的一个途径。在计算温度时，就可以通过计算体系中所有原子的总动能来进行转换，总动能和温度的关系满足公式：



所以有：

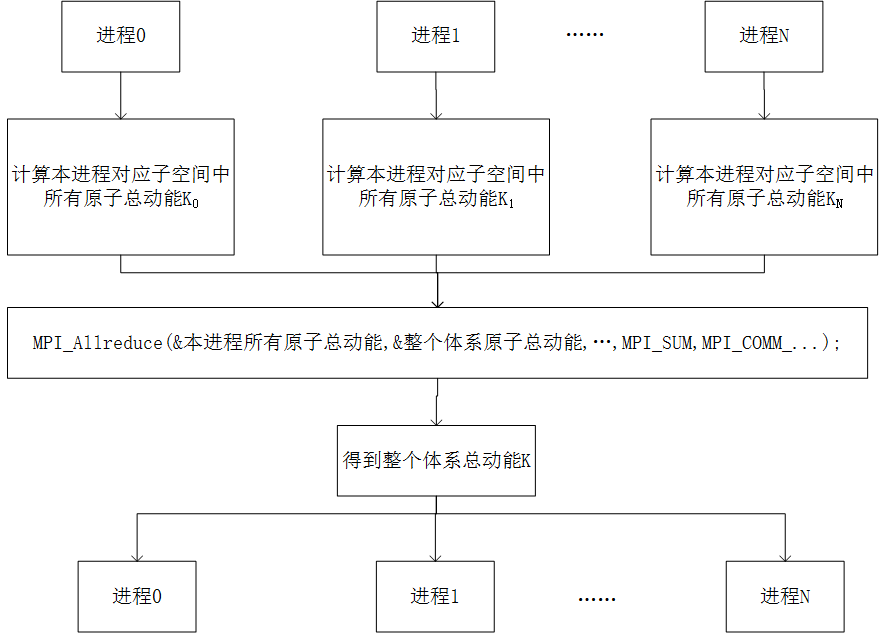


其中，K为总动能，N为体系中的总原子数，kB是波尔兹曼常数[]，T为温度。因此，在已知总原子数N和常数kB的情况下，温度T可以容易地通过总动能K来求得。

要获得整个体系的总动能，首先要计算每个进程中原子的总动能，再将所有进程的这一值相加。假设进程0~N中原子总动能分别为K0，K1，…，KN，则可求得体系总动能K：



在MPI编程模型中，函数MPI\_Allreduce()可完成这一工作。MPI\_Allreduce()是MPI中一种组归约操作，归约操作指的是用某一特定的操作来对某些进程输入缓冲区的数据进行计算，并将累积的结果返回给某一进程；组归约操作只与归约操作有一点不同，其计算结果会返回给参与计算的每一个进程。在计算总动能时，我们需要利用组归约操作把所有进程计算出的总动能相加，得到体系的总动能，再将结果返回给每一个进程。组归约示意图？因此，可按图所示的程序结构来计算体系的总动能，从而通过式求得温度这一宏观量。



MD并行化程序中提取体系温度的算法如下：

|  |
| --- |
| **MD并行程序宏观量温度的提取伪代码** |
| **atoms：**原子信息结构体数组  **MAXCELLNUM：**常数，元胞中原子数量最大值  **cellnum：**本进程中的总元胞数量  **cellatom：**各元胞中原子数量的数组  **myKe：**本进程中原子总动能  **globalKe：**整个体系原子总动能  **T：**温度  **N：**整个体系原子总数量  **kb：**波尔兹曼常数  **m：**原子的质量 |
| 1:double myKe=0.0, globalKe=0.0, temp, T=0.0;  2:  3:for(int i=0 ; i < cellnum; i++ )  4:for(int j= MAXCELLNUM\*i ; j < cellatom[i] + MAXCELLNUM\*i ; j++)  5:{  6: **temp = 0.5 \* m \* atoms[j].vel \* atoms[j].vel;**  7: **myKe = myKe + temp;**  8:}  9:  10: **MPI\_Allreduce(&myKe, &globalKe, 1,**  **MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_...);**  11:  12: **T = 2 \* globalKe / ( 3 \* N \* kb);** |

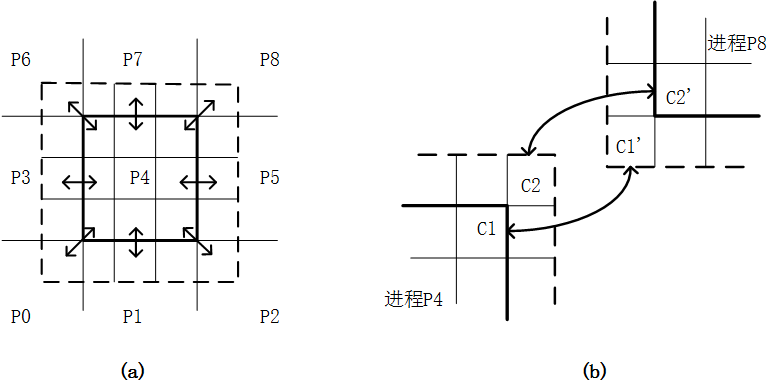
其中，第3~8行计算本进程中的原子总动能Kn，第10行利用MPI\_Allreduce()组归约计算得到整个体系的总动能，第12行则计算宏观量温度值。

总体程序伪代码：

1. MD模拟并行化通信策略研究
   1. 基本的点对点通信

第三章3.2.3节中说明到，MD模拟并行化程序中通信是不可避免的。每个进程在计算对应子空间中边界原子的受力时，需要其它的邻居进程中相应一部分原子的坐标信息，这部分的信息需要通过与其它进程进行通信才能获得；另外，在原子运动过程中，部分原子运动到了其它进程内时，也需要进行通信。这就引起了我们对其通信策略的研究。

最基本的通信方式就是最直接的点对点通信。其它某个进程需要本进程的数据，或者本进程需要其它某个进程的数据，就直接地将所需数据通过MPI\_Send()、MPI\_Recv()发送给相应进程或从相应进程接收。以二维体系为例，如图（a），进程P4需要和其邻居8个进程进行通信，需要将边界元胞和运动至其它进程的原子数据发送给其它进程，并接收其它进程边界元胞和运动至本进程的原子数据。再以进程P4和进程P8之间的通信为例，如图（b），进程P4的元胞C1保存了一部分进程P8所需要的原子数据，C2保存了运动至进程P8的原子的数据；同理，进程P8的元胞C2’保存了一部分进程P4所需要的原子数据，C1’保存了运动至进程P4的原子的数据。两者通信时，进程P4将C1的数据发送至C1’，将C2的数据发送至C2’，同时接收来自C1’和C2’的数据；进程P8做同样的发送和接收。同理，进程P4与其它进程P0、P1、…、P7也需要做同样的通信，每与一个其它进程通信就需要调用一次点对点通信操作，共需要8次。



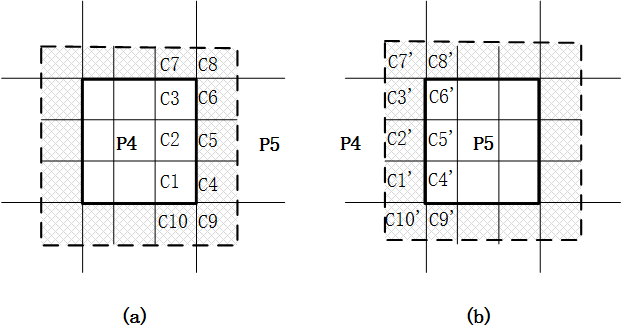
* 1. 两种MD并行程序中的通信优化策略

除最基本的八方向点对点通信策略外，通过调研了解到，在2016年，。。。[]提出了几种在动力学蒙特卡洛（KMC）并行化程序中的通信优化策略。其中，有两种优化策略思想可借鉴用于分子动力学并行化程序，即通信聚合优化与单边通信优化。

1）MD模拟并行化中的通信聚合优化

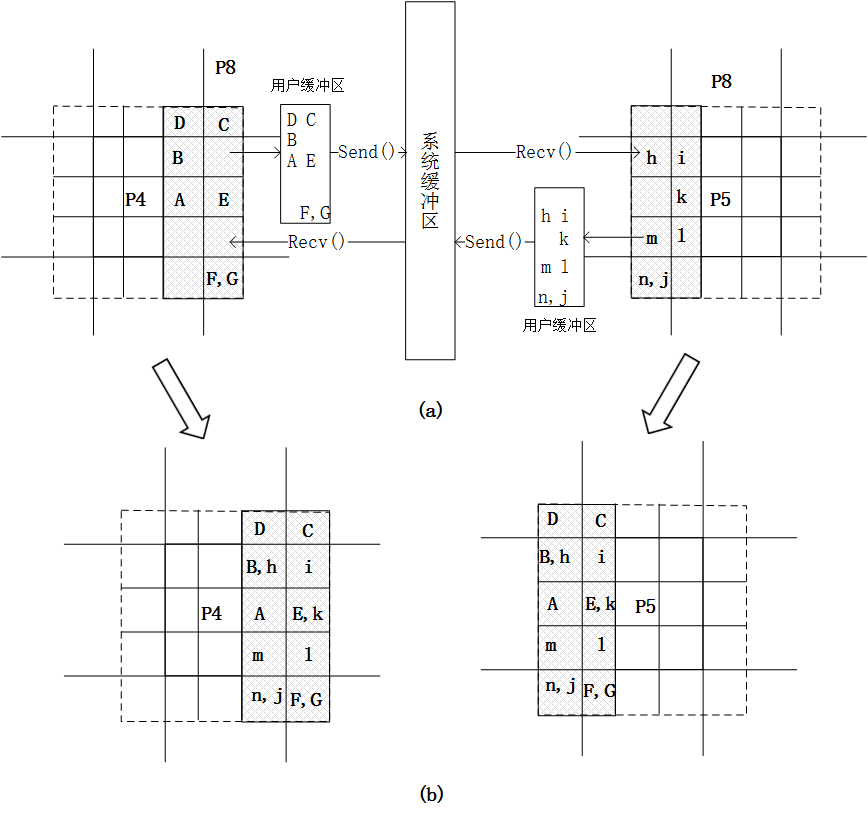
在动力学蒙特卡洛（KMC）并行化程序中，由于其算法的性质，为了消除模拟时的数据冲突，需要将每个进程再次分割为若干个区段，进行通信的单位不是一个进程而是一个区段。而在MD模拟程序的并行化中，不需要考虑进程内部的数据冲突，其通信的单位是进程。所以，通信聚合优化策略在MD方法中有所不同，其主要思想是将某个进程与其它某几个进程的通信合并成一次，每次调用发送或接收操作时尽可能地多处理一些数据，使得总通信次数减少，从而减少了系统函数调用的开销，减少通信时间。

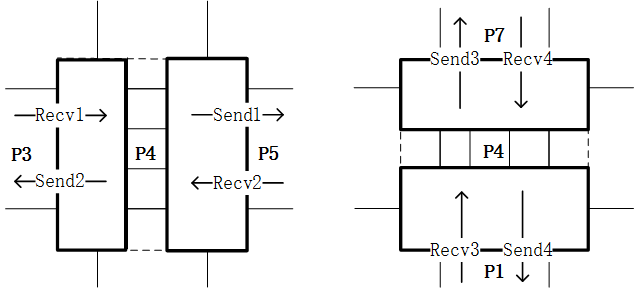
在4.1节中，对MD并行程序中基本的点对点通信方式做了分析。进一步研究发现，在这种方式中，相邻的进程间除了相互进行通信的元胞所表示的数据一致外，还有其它元胞表示的数据也一致。如图，以进程P4、P5为例，在基本的点对点通信中，进程P4与P5之间的通信数据只存在于元胞C1~C6和与之对应的C1’~C6’中。但是实际上，P4所拥有的数据包括了C1~C10的所有元胞，与之对应的，P5拥有的数据包括了C1’~C10’所有元胞。所以，考虑将通信的数据范围从C1~C6扩展到C1~C10。



以上述说明的通信聚合思想为基础，可以设计出MD模拟并行程序中的通信聚合优化策略。如图（a），以进程P4和进程P5通信为例，假设进程P4的边界元胞中包含了原子A、B，从进程P4运动至其它进程中的原子有C~G，进程P5的边界元胞中包含了原子i、k、l，运动至其它进程中的原子有h、m、n和j。将通信的数据范围扩大，进程P4与P5通信时，把A~G所有原子的数据均拷贝至用户缓冲区，再通过MPI\_Send()发送给进程P5；同理，进程P5将h~n所有原子数据拷贝至用户缓冲区，发送给进程P4，其结果如图（b）。

进程P4与其他进程的通信需要以一定的顺序执行。如图，首先，进程P4从P3接收数据并同时向P5发送数据（图中Recv1和Send1）；其次，进程P4从P5接收数据并同时向P3发送数据（图中Recv2和Send2）；然后，进程P4从P1接收数据并同时向P7发送数据（图中Recv3和Send3）；最后，进程P4从P7接收数据并同时向P1发送数据（图中Recv4和Send4）。如此一来，进程P4无需直接与进程P8进行通信，进程P4中若有原子运动至进程P8中，可通过进程P5间接地将原子数据送达，并且每一次通信可以尽可能多地发送数据，减少了总通信次数。





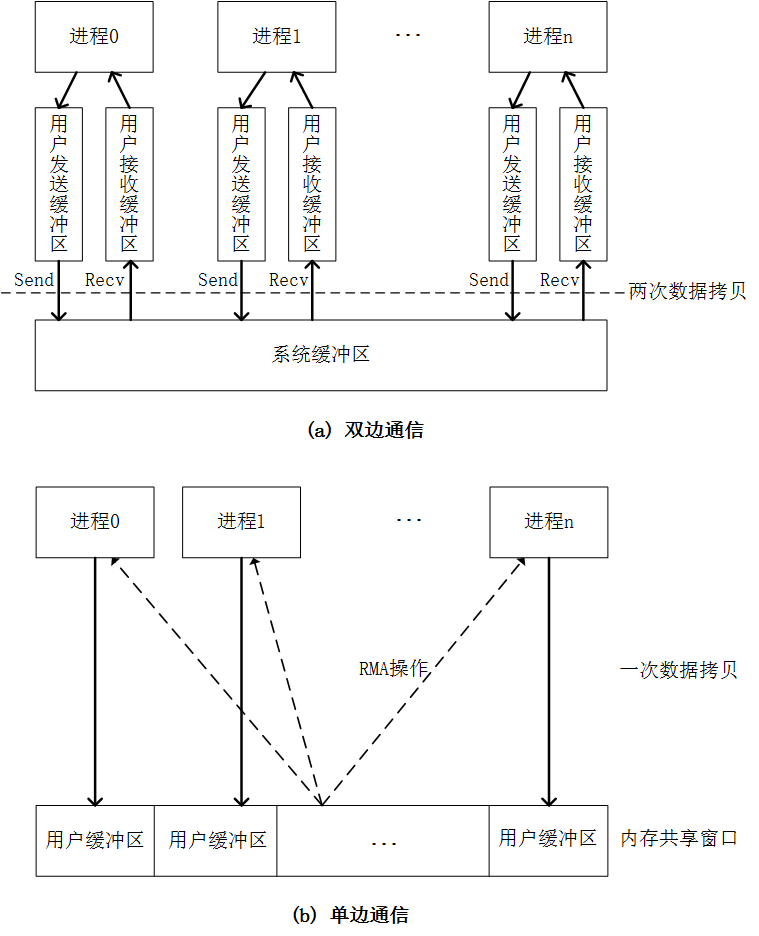
可以看出，进程只需要进行方向与坐标轴正方向相同和相反的通信，不需要再进行对角线方向上上的直接通信。比如进程P4，只需要与P1、P3、P5、P7通信，不直接与P0、P2、P6、P8进行通信。以二维体系为例，其伪代码如下：

|  |
| --- |
| **MD并行程序通信聚合策略伪代码（二维体系）** |
| **neighborProc：**各方向的邻居进程标识数组  **negProc：**沿坐标轴负方向的邻居进程标识  **posProc：**沿坐标轴正方向的邻居进程标识  **sendbuf：**发送缓冲区  **recvbuf：**接收缓冲区 |
| 1:for(int i=0 ; i < 2; i++ ) //二维体系，两个坐标维度  2: {  3: negProc = neighborProc[2\*i];  4: posProc = neighborProc[2\*i + 1];  5:  6: **CopyToBuf(sendbuf, …);**  7: **MPI\_Send(sendbuf, …, posProc, 0, MPI\_COMM\_...);**  8: **MPI\_Recv(recvbuf, …, negProc, 0, MPI\_COMM\_...);**  9: **ProcessData(recvbuf, …);**  10:  11: CopyToBuf(sendbuf, …);  12: **MPI\_Send(sendbuf, …, negProc, 0, MPI\_COMM\_...);**  13: **MPI\_Recv(recvbuf, …, posProc, 0, MPI\_COMM\_...);**  14: ProcessData(recvbuf, …);  15: } |

其中，第6行CopyToBuf()将要发送的数据拷贝至发送缓冲区中，第9行ProcessData()是处理接收缓冲区中的数据。第7、8行是进程向坐标正方向的邻居进程发送数据，并接收来自坐标负方向的邻居进程的数据；第12、13行是进程向坐标负方向的邻居进程发送数据，并接收来自坐标正方向的邻居进程的数据。

2）单边通信优化

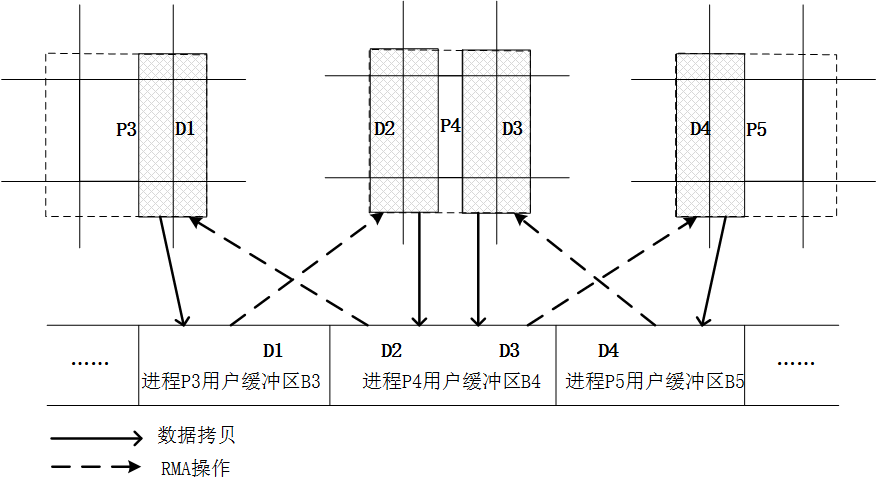
除通信聚合优化外，还可以利用单边通信机制进行优化。MPI的单边通信机制是MPI2.0版本中新加入的特性，它与点对点通信等双边通信机制不同之处在于：在点对点等双边通信机制中，发送方需要先将数据从用户发送缓冲区中拷贝至系统缓冲区，然后接收方再从系统缓冲区中拷贝出来，至用户接收缓冲区；而在单边通信机制中，可以通过开辟一个内存共享窗口将用户缓冲区的内存进行进程间的共享。这样一来，只需要发送方将要发送的数据拷贝至用户缓冲区中，其它进程若需要这部分数据，可以通过远程内存访问（RMA）操作直接获取到共享内存的用户缓冲区中的数据地址，从而直接进行处理，而不需要再消耗时间从系统缓冲区当中拷贝出来。图。。。比较了单边通信机制与双边通信机制的区别，图（a）展示了双边通信机制的数据走向，可以看到在Send发送函数和Recv接收函数处，各进行了一次数据拷贝；图（b）展示了单边通信机制的数据走向，进程只需要将要发送的数据拷贝至内存共享窗口中，其它进程可在需要时，通过远程内存访问（RMA）操作直接对数据进行访问。可明显看出，单边通信机制去除了部分数据的多次拷贝，从而减少了进程间的通信时间。



* 1. 通信聚合和单边通信结合的通信优化策略

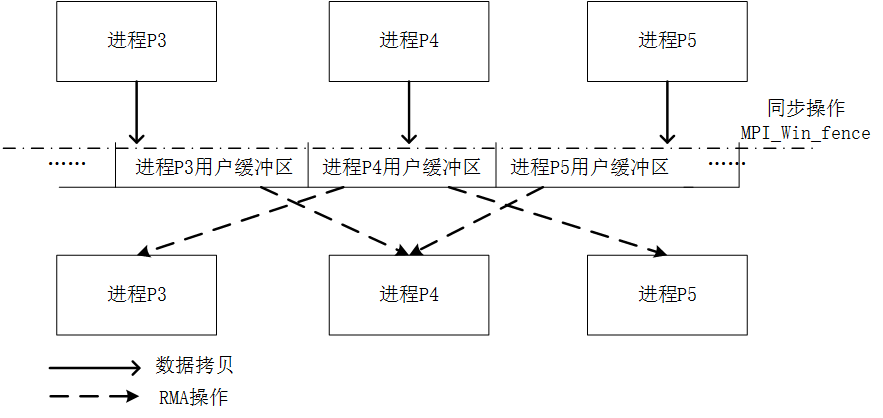
前面分析的两种通信优化策略，都还只是被单独使用。基于这一点，考虑将这两种方法结合，提出了一种新的通信策略——聚合-单边通信优化策略。在4.2节中已经分析，在MD模拟并行化程序中，通信聚合优化策略通过把若干次通信合并成一次，减少了总通信次数和通信调用开销；单边通信优化通过开辟内存共享窗口将用户缓冲区共享，减少了每次通信进程间数据的拷贝次数。现在将这两种方法结合，通信聚合策略中，由于增加了每次通信需要发送的数据量，其用户发送缓冲区和接收缓冲区都会随之增大，每次通信中拷贝至系统缓冲区和从系统缓冲区拷贝出来的数据量增大。若将单边通信机制与之结合，将其用户缓冲区进行内存共享，减少多余的数据拷贝，则可同时发挥两者的优势，获得更好的优化效果。

图。。展示了聚合-单边通信优化策略的主要过程。该图以进程P4与P3、P5的通信为例，假设P3需要发送的数据块为D1，P4发送的数据块为D2、D3，P5发送的数据块为D4。首先，各进程需要为自己的用户缓冲区开辟一个内存共享窗口，这可以通过MPI\_Win\_allocate\_shared()操作实现，使得各进程的用户缓冲区都可以被其它进程直接访问；然后，各进程将需要发送的数据拷贝（图中粗实线箭头）至用户缓冲区中，进程P3将数据块D1拷贝至缓冲区B3，进程P4将数据块D2、D3拷贝至缓冲区B4，进程P5将数据块D4拷贝至缓冲区B5，这一步操作需要进行一次同步操作，即需要保证所有的进程均已经完成了数据的拷贝，才能进行下一步操作；最后，各进程通过RMA操作（图中粗虚线箭头）对共享窗口进行访问，获得对应进程缓冲区的起始地址，处理相应的数据，这一步不需要数据的拷贝，进程P3从缓冲区B4中获取数据块D2，进程P4从缓冲区B3中获取数据块D1、从缓冲区B5中获取数据块D4，进程P5从缓冲区B4中获取数据块D3。完成了这些操作，就完成了进程P4与P3、P5的一次通信。很明显，与单独的通信聚合优化策略相比，聚合-单边通信策略减少了从系统缓冲区到用户缓冲区的数据拷贝。

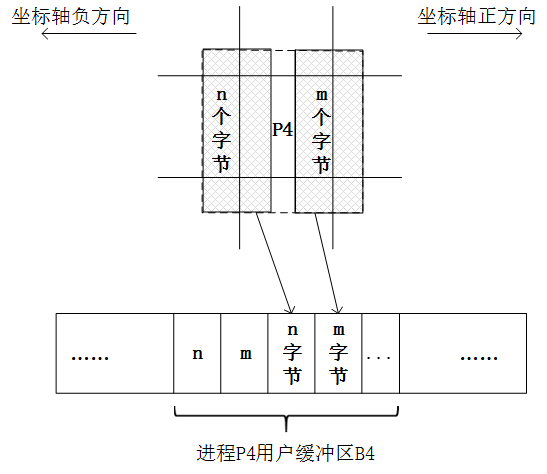


通过上面的分析可以知道，在进程将数据拷贝至用户缓冲区时，需要一个同步操作。该同步操作保证只有在所有进程都已经成功将数据拷贝至缓冲区后，才能进行后面的操作，否则将引起数据的冲突与混乱。仍以图中的过程为例，若没有同步操作，在进程P4将数据块D2、D3拷贝至缓冲区B4后，被允许进行下一步的RMA操作，进程P4可以直接获取缓冲区B3中的数据，但是此时很可能进程P3还没有将数据块D1拷贝至缓冲区B3中，导致进程P4获取到的数据并不是所期望的数据，从而引起数据的混乱。所以，该同步操作是必要的。

图展示了该同步操作在通信过程中的位置，MPI中的MPI\_Win\_fence()可以完成这一同步。MPI\_Win\_fence()操作可以保证，只有在所有进程都调用了此函数后，进程才能继续往下运行，否则将阻塞在此同步操作上。



另外需要注意的一点是，在进程通过RMA操作获取到相应进程的缓冲区起始地址后，需要告诉进程从缓冲区中取多少字节的数据。由于RMA操作只能获取到起始地址，并不能指明有多少字节数据有效，所有需要在缓冲区的数据中显示地指明数据的字节量。考虑在每个进程缓冲区的起始处，空出两个int类型的空间来指定后面有效数据的字节数。如图，以进程P4为例，要将负轴方向的n个字节和正轴方向的m个字节拷贝至缓冲区B4。需要先在缓冲区起始的位置处添加两个int型数据n、m，然后再将n、m个字节的数据按顺序拷贝在其后。这样就指定了有效数据量的大小，方便其它进程访问。



以二维体系为例，MD模拟并行程序的聚合-单边通信优化策略伪代码如下：

|  |
| --- |
| **MD并行程序聚合-单边通信优化策略伪代码（二维体系）** |
| **neighborProc：**各方向的邻居进程标识数组  **bufsize：**缓冲区大小  **databuf：**用户缓冲区  **win：**内存共享窗口  **negProc：**沿坐标轴负方向的邻居进程标识  **posProc：**沿坐标轴正方向的邻居进程标识  **negbufAddress：**沿坐标轴负方向的邻居进程缓冲区起始地址  **posbufAddress：**沿坐标轴正方向的邻居进程缓冲区起始地址  **n：**沿坐标轴负方向的通信数据字节数  **m：**沿坐标轴正方向的通信数据字节数 |
| 1: **MPI\_Win\_allocate\_shared(bufsize, 1,**  **MPI\_INFO\_NULL, MPI\_COMM\_..., &databuf, &win); //** 开辟内存共享窗口  ……  ……  2: for(int i=0 ; i < 2; i++ ) //二维体系，两个坐标维度  3: {  4: negProc = neighborProc[2\*i];  5: posProc = neighborProc[2\*i + 1];  6:  7: memcpy(databuf, &n, sizeof(int));  8: memcpy(databuf+sizeof(int), &m, sizeof(int));  9:CopyToBuf(databuf+2\*sizeof(int), n, …);  10:CopyToBuf(databuf+2\*sizeof(int)+n, m, …);  11: **MPI\_Win\_fence(0,win); //** 同步操作  ……  ……  12: **MPI\_Win\_shared\_query(win, negProc, …, &negbufAddress);**  13: **MPI\_Win\_shared\_query(win, posProc, …, &posbufAddress);**  14: memcpy(&n, negbufAddress, sizeof(int));  15: memcpy(&m, negbufAddress+sizeof(int), sizeof(int));  16: ProcessData(negbufAddress+2\*sizeof(int), n+m, …);  17: memcpy(&n, posbufAddress, sizeof(int));  18: memcpy(&m, posbufAddress+sizeof(int), sizeof(int));  19: ProcessData(posbufAddress+2\*sizeof(int), n+m, …);  20: **MPI\_Win\_fence(0,win); //** 同步操作  21: }  ……  ……  22: MPI\_Win\_free(&win); |

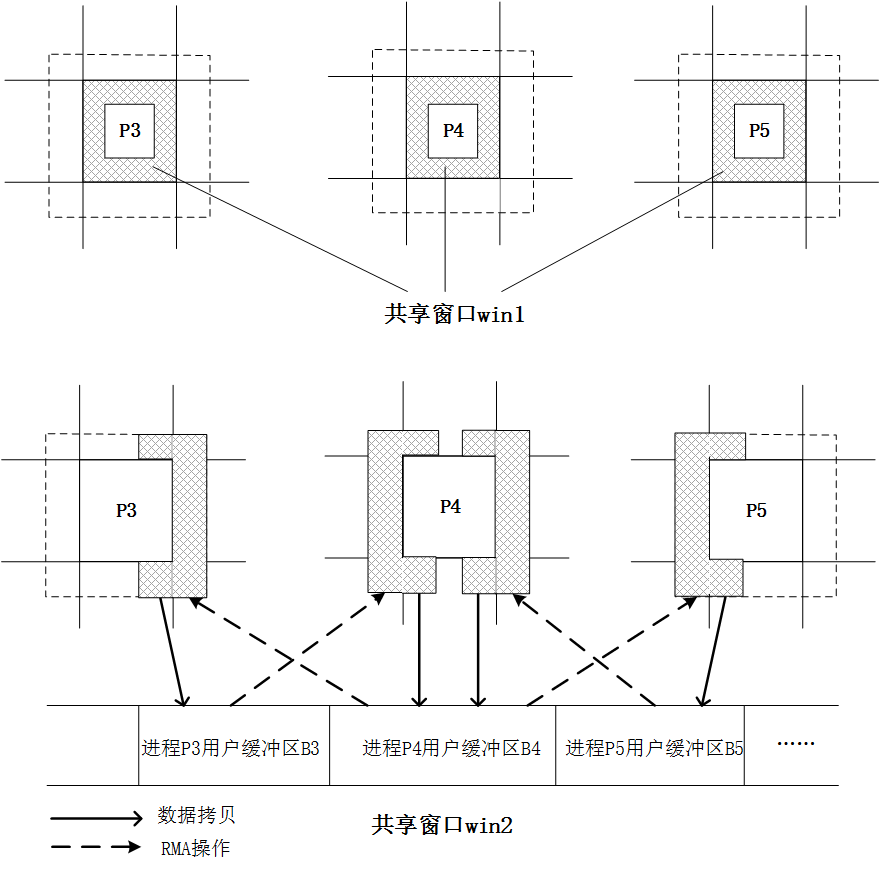
其中，第1行MPI\_ Win\_allocate\_shared()开辟了一个大小为bufsize的内存共享窗口win，起始地址为databuf。第7~10行是将数据拷贝至缓冲区并指定有效数据量，第11行为同步操作MPI\_Win\_fence()。第12、13行是MPI中的RMA操作，MPI\_Win\_shared\_query()可以获取到指定进程的缓冲区起始地址。在获取了其它进程的起始地址后，第14~19行则从内存共享缓冲区中获取数据并直接进行处理，无需拷贝。第20行同样为同步操作。

* 1. 改进的聚合-单边通信优化策略

通过上面的分析，聚合-单边通信优化策略仅仅是直接地将两种通信策略结合起来。单边通信在这种策略中，通过将用户缓冲区内存共享，只能减少用户缓冲区和系统缓冲区之间的拷贝过程，从而减少通信时间。但是在MD模拟并行程序中，用户缓冲区有两个方面的用途：第一个方面，存储运动至其它进程的原子的信息数据，再将其发送至相应进程；第二个方面，存储本进程中边界元胞中的原子的信息数据，再将其发送至相应进程。因此，可以考虑通过直接将各进程中边界元胞原子数据的存储（对应第二个方面）进行内存共享，各进程可以直接通过单边通信操作获取此部分数据，这样就可以减少一部分从进程的存储空间到用户缓冲区的数据拷贝，从而进一步减少通信时间。把这种通信策略称为改进的聚合-单边通信优化策略。

在4.3节中分析到，进程在进行通信时，把准备发送的数据全部都拷贝至内存共享窗口的用户缓冲区当中，这些数据包括两个部分：子空间中边界元胞的原子信息和运动至其它进程的原子的信息。这一步骤引入了一个将数据从进程存储空间拷贝至用户缓冲区的过程，考虑尽量地减少这些数据的拷贝，可以直接将进程的存储空间进行内存共享。由于防止数据的冲突，不可能同时将两部分数据的存储空间都进行内存共享，而又考虑到边界元胞中的原子一般比运动至其它进程的原子多，所以选择对前者的存储空间进行内存共享化，充分利用单边通信机制的优势。

图展示了在聚合-单边通信优化策略基础上做的进一步改进，主要的思想是对存储边界元胞原子信息的空间直接进行内存共享化。这就需要引入一个新的内存共享窗口（图中的窗口win1），该窗口用于共享各进程边界元胞的原子数据信息，使得各进程在通信时可以直接访问其它进程的这部分数据。而原来的窗口（图中的窗口win2）仍可用于共享各进程的用户缓冲区，不同的是，进程只需要将运动至其它进程的原子数据拷贝至缓冲区，这部分数据的通信与改进前的策略是相同的。由此看来，改进后的策略相比于改进前的策略，又进一步减少了一部分数据的拷贝，进一步减少了通信时间。



以二维体系为例，MD模拟并行程序改进的聚合-单边通信优化策略伪代码如下：

|  |
| --- |
| **MD并行程序改进的聚合-单边通信优化策略伪代码（二维体系）** |
| **neighborProc：**各方向的邻居进程标识数组  **bufsize：**缓冲区大小  **databuf：**用户缓冲区  **win1：**内存共享窗口1  **win2：**内存共享窗口2  **negProc：**沿坐标轴负方向的邻居进程标识  **posProc：**沿坐标轴正方向的邻居进程标识  **negbufAddress：**沿坐标轴负方向的邻居进程缓冲区起始地址  **posbufAddress：**沿坐标轴正方向的邻居进程缓冲区起始地址  **n：**沿坐标轴负方向的通信数据字节数  **m：**沿坐标轴正方向的通信数据字节数 |
| 1: **MPI\_Win\_allocate\_shared(…, 1,**  **MPI\_INFO\_NULL, MPI\_COMM\_..., &…, &win1);**  2: **MPI\_Win\_allocate\_shared(bufsize, 1,**  **MPI\_INFO\_NULL, MPI\_COMM\_..., &databuf, &win2);** **//**开辟内存共享窗口  ……  ……  3: for(int i=0 ; i < 2; i++ ) //二维体系，两个坐标维度  4: {  5: negProc = neighborProc[2\*i];  6: posProc = neighborProc[2\*i + 1];  7:  8: memcpy(databuf, &n, sizeof(int));  9: memcpy(databuf+sizeof(int), &m, sizeof(int));  10:CopyToBuf(databuf+2\*sizeof(int), n, …);  11:CopyToBuf(databuf+2\*sizeof(int)+n, m, …);  12: **MPI\_Win\_fence(0,win2); //** 同步操作  ……  ……  13: **MPI\_Win\_shared\_query(win1, negProc, …, &negbufAddress);**  14: **MPI\_Win\_shared\_query(win1, posProc, …, &posbufAddress);**  15: ProcessData(negbufAddress, …);  16: ProcessData(posbufAddress, …);  17:  18: **MPI\_Win\_shared\_query(win2, negProc, …, &negbufAddress);**  19: **MPI\_Win\_shared\_query(win2, posProc, …, &posbufAddress);**  20: memcpy(&n, negbufAddress, sizeof(int));  21: memcpy(&m, negbufAddress+sizeof(int), sizeof(int));  22: ProcessData(negbufAddress+2\*sizeof(int), n+m, …);  23: memcpy(&n, posbufAddress, sizeof(int));  24: memcpy(&m, posbufAddress+sizeof(int), sizeof(int));  25: ProcessData(posbufAddress+2\*sizeof(int), n+m, …);  26: **MPI\_Win\_fence(0,win2); //** 同步操作  27: }  ……  ……  28: MPI\_Win\_free(&win1);  29: MPI\_Win\_free(&win2); |

其中，与**聚合-单边通信优化策略伪代码有所不同的只有一部分**。第1~2行开启了两个共享窗口。第13~16行是从共享窗口win1中直接获得存储空间起始地址，可以直接对边界元胞的原子数据进行处理，避免了此部分数据的拷贝。

计算力、其它代码的变化?

1. 实验结果分析
2. 结 论