计算机科学与工程中的并行编程技术大作业

—— 分子动力学程序并行优化

1 背景介绍

分子动力学是一种分子模拟的方法，通过计算机对分子和原子体系运动进行模拟，结合牛顿力学、统计力学、分子间相互作用等原理模拟微观粒子系统随时间演化的行为，通过微观粒子的运动来计算宏观性质。这种分子动力学的模拟在磷脂膜自组装、碳纳米管拉伸断裂等化学和生物领域有着广泛的应用，这些问题需要先通过分子动力学模拟计算出微观粒子间的相互作用，得到合理的模拟预测结果，进行调整才能够在真实材料上实验时达到目标效果。

为了便于计算机模拟过程中的计算与存储，首先需要对粒子进行网格化，将计算区域划分为多个元胞（cell），每个 cell 中包含 10 个粒子。由于模拟的自由度数非常大，设定一个截断半径，当两个粒子间的距离大于截断半径时视为这两个粒子间不再有相互作用力，只考虑截断半径内粒子间的相互作用。在本次的模拟程序中，设置截断半径为 2，即一个粒子需要计算与自己所在同一 cell 中的粒子以及 8 个相邻 cell 中粒子的相互作用。整体的分子动力学模拟过程如图1所示，对粒子的位置和速度进行初始化后，根据截断半径计算粒子间的相互作用力和加速度，对粒子的速度和坐标进行更新，重复迭代直到达到迭代次数。

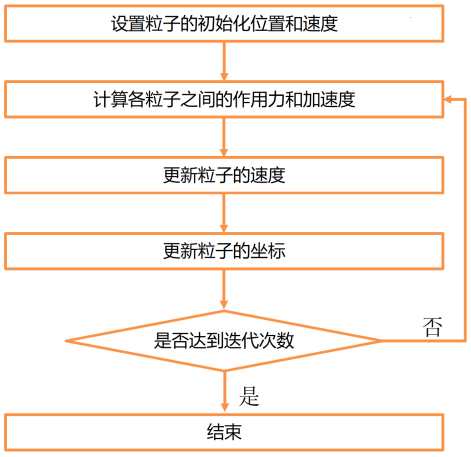


Figure 1: 分子动力学模拟流程

分子动力学模拟的核心在于，对系统中每一个粒子考虑其他所有粒子对这一个粒子的影响，对粒子状态进行更新，如此往复迭代。一个待模拟的微观粒子系统中粒子的数目往往是十分庞大的，要计算的粒子间相互作用的计算量十分庞大，因此在模拟的过程中使用并行计算对程序进行优化是必要的，也是本次作业中的目的。

2 算法热点分析

首先对所给的串行版本的分子动力学模拟程序 `md.cpp` 的主要耗时部分进行分析。程序的计算热点主要在于每个粒子计算与同一个 cell 中其他粒子的相互作用以及与邻居 cell 中粒子的相互作用这两个 `for` 循环中（图2）。

```
// For each cell...
for (int cx1 = 0; cx1 < NCellRows; cx1++) {
  for (int cy1 = 0; cy1 < NCellCols; cy1++) {
    for (int i = 0; i < NPartPerCell; i++) {
      for (int j = 0; j < NPartPerCell; j++) {
    if(i!=j)
      TotPot += interact(particles[cx1][cy1][i],particles[cx1][cy1][j]);
  }
}


// Consider each other cell...
for (int cx2 = 0; cx2 < NCellRows; cx2++) {
  for (int cy2 = 0; cy2 < NCellCols; cy2++) {
    if ((abs(cx1-cx2) < 2) && (abs(cy1-cy2) < 2) &&
    (cx1 != cx2 || cy1 != cy2)) {
      for (int i = 0; i < NPartPerCell; i++) {
        for (int j = 0; j < NPartPerCell; j++) {
            TotPot += interact(particles[cx1][cy1][i],particles[cx2][cy2][j]);
        }
      }
    }
  }
}
```

Figure 2: 耗时循环代码

本次计算规模为 163840 个粒子，共 16384 个 cell，将其划分为大小 128×128 的网格。在每一个迭代步中，对于同 cell 中的相互作用，每个粒子需进行 9 次计算，所有粒子需进行 $163840 \times 9 = 1474560$ 次计算；对于相邻 cell 间的相互作用，每个粒子需与截断半径内共 $8 \times 10 = 80$ 个粒子的相互作用进行计算（除边界 cell 外），估算大约需进行 $126 \times 126 \times 10 \times 80 + 126 \times 4 \times 10 \times 50 + 4 \times 10 \times 30 = 12954000$ 次运算，以上所有的运算单位为一次相互作用函数 `interact()` 的调用。在原始的程序中这两部分是串行执行的，所以执行时间为两部分的累加和，总次数是非常巨大的，成为的程序主要的计算耗时热点。

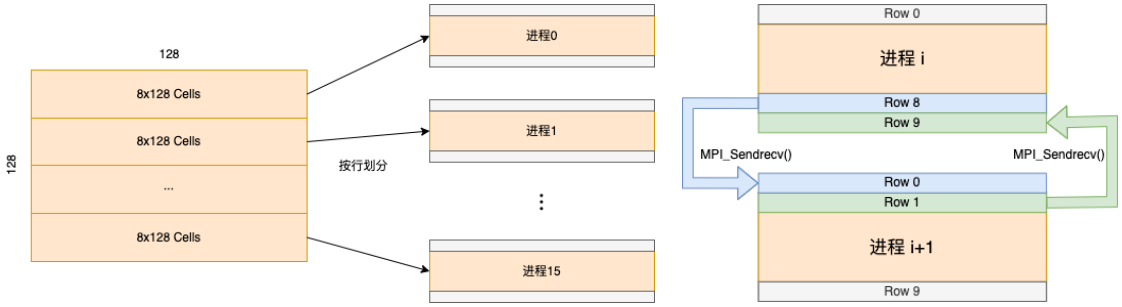
在每一次迭代中，先计算粒子间的相互作用，在所有计算完成后再对粒子状态进行统一更新，因此这两个主要耗时循环中的每一次计算是相互独立、无依赖关系的，可以并行进行以提高程序的效率，下面将从几个不同的方面对该程序进行并行优化。

3 优化方法

运行串行版本的程序，得到运行时间为 $39.4844s$，作为优化对比的基础，后续所有版本的优化均在保证程序正确性的条件下进行，最终的程序输出与串行程序保持一致。**以下的并行优化方法在 x86 平台下进行。**

3.1 MPI 并行

首先将待计算的网格进行一维划分，基于 MPI 进行进程级并行优化，使用 8 个计算节点，每个节点使用 2 个主核，共 16 个 MPI 进程。将尺寸为 128×128 的网格按行划分到 16 个进程中，每个进程 8 行，负责计算大小 8×128 的网格，如图3(a)所示。由于截断半径的存在，使得计算分子相互作用的过程中具有很好的局部性，各进程负责的分块内部的分子可以完全独立地计算；对于位于分块边界（第一行和最后一行）的 cell，需要与相邻进程进行通信获取邻居进程的边界分子信息，计算相互作用力。为了使每个 cell 的计算过程一致，对每个进程中负责计算的网格的两端进行扩展，用于存放通信得到的数据（图3(a)中灰色部分），即每个进程中本地存储的数据块大小为 $(8+2) \times 128$ 个 cell。



(a) 一维划分　　(b) 进程间通信

Figure 3: MPI 并行

在每轮计算开始前，各进程先从相邻进程处获得所需的边界 cell 信息（如图3(b)所示），将自己负责分块的第 1 行发送给前一个进程的第 9 行（绿色部分），将第 8 行发送给后一个进程的第 0 行（蓝色部分），实现两个不同方向的数据通信流。因为每个进程要向相邻的进程发送数据，同时从相邻的进程接收数据，所以使用 `MPI_Sendrecv()` 进行数据的发送和接收操作。首进程和末尾进程比较特殊，只需要和一个相邻进程通信，为了使这两个进程在通信过程中和其他进程统一看待，所以使用虚拟进程 `MPI_PROC_NULL` 作为通信的目标。最终全部进程的通信使用相同的代码实现，但在计算过程中要根据进程号判断当前是否为首尾进程，从而忽略虚拟进程对应的分块中的两行空数据。

使用 MPI 进行进程级优化后，并行版本的程序算法的流程如1所示，优化后的程序运行时间为 $1.89388s$，相比串行版本程序的**加速比为** $s = \frac{39.4844}{1.89388} = 20.8484$，程序运行效率有了较大的提升。

3.2 OpenMP 并行

在前面的基础上，使用 OpenMP 进行线程级并行优化。OpenMP 是共享内存式的编程，所以一个进程内的所有线程共享本地的分块数据 `local_particles`，无需进行通信。线程级并行的方式体现在对循环进行拆分，主线程 fork 出多个线程后每个线程负责一组独立的迭代，是数据并行的一种方式。

在 OpenMP 并行过程中尝试了两种方式，一种是将编译制导语句添加到最外层 for 循环外（对进程内分块的行循环），另外一种是对分块进行列数据拆分，添加到列循环的外侧，这两种不同的并行方式分配到每个线程的计算量基本一致。实验发现添加到最外层循环外的效果更好，因

Algorithm 1: MPI 优化版本

```
MPI_Init();
Copy data to local particles array;
foreach Time step t do
    leftproc = (rank>0) ? (rank-1) : MPI_PROC_NULL;
    rightproc = (rank<size-1) ? (rank+1) : MPI_PROC_NULL;
    MPI_Sendrecv(row[1], leftproc, row[9], rightproc);
    MPI_Sendrecv(row[8], rightproc, row[0], leftproc);
    foreach cell do
        foreach particle i,j from the same cell do
            Calculate the interaction between i, j;
        end
        foreach Neighbour cells do
            foreach particle i,j from different cell do
                Calculate the interaction between i, j;
            end
        end
    end
    Update particles' status;
end
MPI_Finalize();
```

为线程在 fork 和 join 的过程中有开销，对内层列循环进行线程拆分时需要进行多次的 fork-join 过程，而对最外层循环并行只需进行一次该过程，额外开销更小。在循环计算的过程中需要累加总能量值 `Totpot_local`，进行 OpenMP 线程拆分后通过制导语句中的 `reduction` 指定规约变量，来保证最终结果的正确性。

将编译制导语句 `#pragma omp parallel for reduction(+:Totpot_local)` 添加到外层循环处实现 omp 并行优化。因为每个节点中有 14 个主核，已有两个主核用于启动 MPI 进程，为了充分发挥计算资源，每个 MPI 进程可以 fork 6 个线程用于 OpenMP 并行，因此设置环境变量 `OMP_NUM_THREADS=6`（设置为 7 时程序运行时间会出现较大波动）。使用 OpenMP 进一步优化后，程序的运行时间为 $0.524479s$，**加速比为** $s = \frac{39.4844}{0.524479} = 75.2831$，性能取得了三倍以上的提升。

3.3 计算与通信重叠

在之前优化后的程序中，粒子相互作用的计算需要等待各个进程通信完成后再进行，但每个进程的计算中只有首尾两行需要相邻进程的数据，内部 cell 的计算相互独立，可以与通信同步进行，因此考虑使用非阻塞通信方式，让计算与通信重叠，进一步提高程序的效率。

将之前的阻塞通信方式 `MPI_Sendrecv()` 进行替换，使用非阻塞通信函数 `MPI_Isend()` 和 `MPI_Irecv()`，通信和计算的流程变为：

1. 使用非阻塞通信函数进行通信，函数被调用后立即返回；
2. 在通信进行的过程中，计算每个 cell 内部粒子的相互作用以及分块中部 2-7 行的 cell 数据（图4中黄色部分）；

3. 调用 `MPI_Wait()`，等待之前的非阻塞通信结束；

4. 得到通信的数据，计算分块首尾两行的结果（图4中绿色部分）。

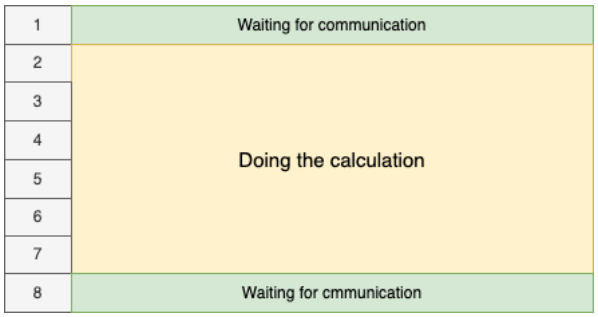


Figure 4: 计算与通信重叠

将通信与计算重叠进一步优化后，程序的运行时间为 $0.426527s$，**加速比提升至：**$s = \frac{39.4844}{0.426527} = 92.5719$。

3.4 循环展开

通过观察图2代码可以发现，形式为多层 for 循环嵌套，循环体内真正参与计算的只有一个语句，且该条语句在循环中对前后无依赖，因此采用循环展开的方式对程序的性能进行进一步优化。循环展开的目的在于将循环体中的迭代次数转换为多个重复的代码块，减少循环控制的开销，同时对数组不同位置的重复操作有利于向量化，提高指令级的并行性。具体循环展开的形式如代码1所示，将最内层遍历 cell 内粒子迭代次数为 10 的 for 循环展开为一条加法指令（经过测试比展开为 10 条重复的加法指令效率更高）。

```
// 循环展开前代码
for(int j=0;j<NPartPerCell;j++){
    if(i!=j)
       TotPot_local += interact(local_particles[cx1][cy1][i],local_particles[cx1][cy1][j]);
}
// 循环展开后代码
TotPot_local += interact(local_particles[cx1][cy1][i],local_particles[cx2][cy2][0])
                +interact(local_particles[cx1][cy1][i],local_particles[cx2][cy2][1])
                +interact(local_particles[cx1][cy1][i],local_particles[cx2][cy2][2])
                +interact(local_particles[cx1][cy1][i],local_particles[cx2][cy2][3])
                +interact(local_particles[cx1][cy1][i],local_particles[cx2][cy2][4])
                +interact(local_particles[cx1][cy1][i],local_particles[cx2][cy2][5])
                +interact(local_particles[cx1][cy1][i],local_particles[cx2][cy2][6])
                +interact(local_particles[cx1][cy1][i],local_particles[cx2][cy2][7])
                +interact(local_particles[cx1][cy1][i],local_particles[cx2][cy2][8])
                +interact(local_particles[cx1][cy1][i],local_particles[cx2][cy2][9]);
```

Listing 1: 循环展开

由于循环体对程序效率的影响因素比较复杂，取决于循环体中的计算量、循环控制的开销以及流水线指令顺序等处理器的特性，因此对不同的循环展开形式进行测试，包括一个循环的展开程度、对不同位置的循环是否进行展开，从中选择效果最好的方式。最终经过测试，对计算不同 cell 间相互作用时的最内层循环进行完全展开得到的效率最高，程序运行时间为 0.375973s，**加速比为：** $s = \frac{39.4844}{0.375973} = 105.0192$。

4 性能测试与结果分析

优化后的程序位于 `md.cpp` 中，程序的编译和任务提交指令位于 `Makefile` 中。在执行程序时，先通过 `make` 指令进行编译，然后通过指令 `make run` 运行程序。

程序不同阶段优化的运行时间和加速比对比结果如表1所示，几种不同优化方法逐步提升了程序的性能。从加速比增长倍数来看，表现最好的优化方式是使用 MPI 的进程级优化，只开启了 16 个 MPI 进程但取得了大于 20 的加速比。因为初始化过后对于计算过程来说，这种优化方式是对任务最粗粒度进行划分，且各个进程的任务间除了较少的通信外再无其他依赖和共享数据，计算的独立性较高，可以近似线性地提高程序的性能。OpenMP 是表现第二好的，使用 6 个线程并行，加速比取得了大概 3 倍的提升，由于其共享内存的机制，且对于同一个变量的赋值需要进行额外的规约操作，且线程的 fork-join 过程会产生额外的开销。另外两种优化方式属于最细粒度的优化，需要根据程序的特点进行设计和实现：分析进程内数据对通信结果的依赖性，从而使用非阻塞通信的方式将无依赖的数据的计算与通信重叠，通过掩盖通信时间的方式提升性能；对循环进行展开，通过掩盖循环控制开销和更好地利用寄存器和向量化的特性来提高程序的性能。这些细粒度的优化方法需要进行比较多方案的尝试，才能得到实际运行过程中的最优性能。

Table 1: 不同优化阶段的性能对比

	串行版本	MPI 优化	OMP 优化	计算通信重叠	循环展开
运行时间	39.4844	1.89388	0.524479	0.426527	0.375973
加速比	-	20.8484	75.2831	92.5719	105.0192

将最终版本的并行程序运行 5 次，记录其运行的平均时间与最优时间，结果如表2所示，可以看出经过并行优化后的程序相比最初的串行版本性能有了很大的提升，取得了理想的加速比；并且程序运行时间较稳定，说明并行执行过程中任务的划分比较平衡合理，实现效果较好。包含 MPI 初始化部分的程序总运行时间为 0.417385s，初始化部分对整体性能影响较小。

Table 2: 程序优化结果

使用平台	串行程序时间	并行程序平均时间	并行程序最优时间	最优加速比
X86	39.4844s	0.3822744s	0.375973s	105.0192

对于目前的程序来说，仍存在的性能限制可能主要在于对 OpenMP 自动的并行化和任务划分可控性较弱，如果能对多层嵌套的循环进行更高效的拆分并行的话，性能可能还有可提升的空间。

5 结论

在本次任务中，经过进程级、线程级以及一系列细粒度的技术对分子动力学模拟的程序进行了优化，最终取得了很好的性能提升，加速比达到了 105.0192。通过这次优化的过程，总结出了对一个程序进行并行优化的基本流程，首先应该对任务进行粗粒度划分，分配给多个进程执行，通过通信进行数据交换；然后可以将进程内的任务进一步划分给多个线程执行，做数据并行。一般经过这两个步骤就可以得到比较不错的性能了，但要想追求更极致的加速比，就需要对具体的程序和算法进一步的分析，使用更细粒度的优化方法。一般来说程序的并行优化过程中效果和成本是不对等的，在最开始使用简单的优化方式便能够取得很好的加速比，随着优化程度的加深，需要更复杂细化的更改与对代码的理解，才能取得一点性能的提升，在本次作业的实践过程中对这一点的感受十分深刻。