并行与分布式计算基础：作业 1 报告

吕洋

2022 年 11 月 12 日

目录

1 问题叙述及理论分析

1.1 问题叙述

考虑的边值问题为

$$\begin{cases} -\Delta u + \alpha u = f \text{ in } \Omega, \\ \qquad\qquad u = 0 \text{ on } \partial\Omega. \end{cases}$$

其中，$\Omega = (-1,1)^2$，

$$f(x,y) = 2\left(1-x^2\right) + 2\left(1-y^2\right) + \alpha\left(1-x^2\right)\left(1-y^2\right)$$

1.2 简要理论分析

这是一个 Poisson 边值条件的二阶线性椭圆 PDE。该方程有解析解

$$u(x,y) = \left(1-x^2\right)\left(1-y^2\right)$$

$\alpha \geq 0$ 时，这是该方程唯一的古典解，利用椭圆方程的极值原理立刻得证。

2 数值求解算法

我们使用有限差分法将问题离散化，利用 $Jacobi$ 迭代法求解该线性系统。该方法的一般表达式如下：

对于线性系统 $Ax = b$，选定初值 $x^{(0)}$ 后，

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j\neq i} a_{ij}x_j^{(k)}\right), \quad i = 1,2,\ldots,n$$

时间复杂度为 $O(n^2)$ (对于稀疏的线性系统)

对于这个问题来说，记 U_{ij}^k 为以 h_x, h_y 为精度的离散解，那么更新公式为

$$U_{ij}^{k+1} = \begin{pmatrix} a_1 & a_2 & a_1 \\ a_3 & a_0 & a_3 \\ a_1 & a_2 & a_1 \end{pmatrix} \bigotimes \begin{pmatrix} U_{i+1,j-1}^k & U_{i+1,j}^k & U_{i+1,j+1}^k \\ U_{i,j-1}^k & U_{i,j}^k & U_{i+1,j-1}^k \\ U_{i-1,j-1}^k & U_{i-1,j}^k & U_{i-1,j+1}^k \end{pmatrix}$$

其中

$$\begin{aligned} a_0 &= \frac{1}{h_x^2+h_y^2}\left(2 + 2\frac{h_y^2}{h_x^2} + 2\frac{h_x^2}{h_y^2}\right) \\ a_1 &= -\frac{1}{2*\left(h_x^2+h_y^2\right)} \\ a_2 &= -\frac{h_x^2}{h_y^2*\left(h_x^2+h_y^2\right)} \\ a_3 &= -\frac{h_y^2}{h_x^2*\left(h_x^2+h_y^2\right)} \end{aligned}$$

$\bigotimes$ 表示矩阵 $Hadamard$ 积复合对矩阵所有元素求和。

3 实验设定

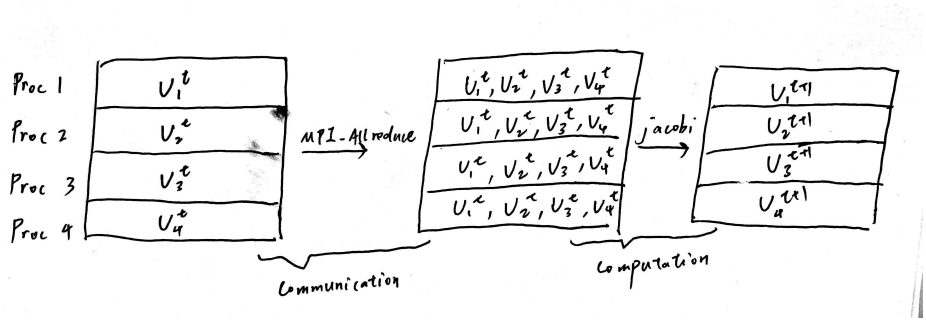
实验时，固定参数 $\alpha = 0.05$, $h_x = h_y = \frac{1}{128}$, 迭代次数 20000。后验误差使用格点上的 L^∞ 范数估计。

4 并行策略 1：集合通信

4.1 算法描述

使用 C++ 二维数组记录当前时刻的近似值。假设有 N 个进程可以使用，让第 i 个进程负责坐标在 $[\frac{-1+i}{N}, \frac{i}{N}) \times [-1, 1]$ 的长方形连续区域内的计算，这样做的好处是数值储存在连续的内存空间，方便 MPI 进行信息传递。

由于 Jacobi 迭代每一轮都要用到上一轮近似得到的线性系统的结果，我们可以让每个 process 在每一轮迭代前，利用`MPI_Allgather()`来获取整个系统在上一轮的近似值，之后分别进行各自负责部分的更新计算。整个算法的示意图如下：



代码结构大致如下：(源文件为 `collective.c`)

```
int main(int argc, char* argv)
{
    MPI_init(&argc, &argv);
    do_some_initializaion();
    while(k <= max_iter)
    {
            MPI_Allgather(&send_buff,send_size,MPI_FLOAT, receive_buff,receive_size,MPI_FLOAT,
    MPI_COMM_WORLD);

            do_local_update();

    }
    post_processing();
    MPI_Finalize();
}
```

进程数	1	4	16
总时间 (Wall time)	5.509102	1.971083	1.711334
误差	0.000205021	0.000205021	0.000205021
并行效率	100%	69.9%	20.1%

表 1: 策略 1 实验结果

进程数	1	4	16
总时间 (Wall time)	5.509102	1.971083	1.711334
通信时间	0.067982	0.538782	1.362803
通信占比 (通信/总时间)	1.233989%	27.334329%	79.633946%

表 2: 策略 1 通信时间占比

4.2 实验结果及分析 (profiling)

见表。其中并行效率为 $\frac{P_1}{NP_N} \times 100\%$, 只统计了循环时间。可以看到，进程少的时候表现的很好，进程多的时候虽然也可以并行成功运行解决问题，但效率大大下降。猜想是通信量太大，时间过长导致的。利用`MPI_Wtime()`计时, 印证了这一猜想。因此需要考虑其他更好的并行策略。

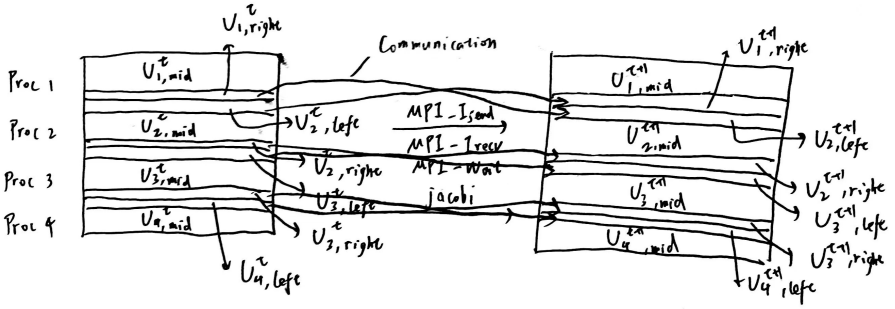
5 并行策略 2：点对点通信

5.1 算法描述

存储数据方法同上

由于 Jacobi 迭代实际上每个点只需要用到相邻 9 个点的结果，因此每个进程实际只需在每个循环知道其相上一个进程最下方的行，和下面进程最上面的行的数据，不需要其他数据。因此，每个进程可以在每个循环开始时，用`MPI_Isend()` 和 `MPI_Irecv()`这样的点对点非阻塞通讯获取这两行的数据，存在某两个 buffer 里，然后进行局部的计算更新。

使用非阻塞通讯的理由是，每个进程中间的格点计算只需自己进程的数据，只有两边的格点需要新的数据。因此使用非阻塞通讯有利于实现 communication-computation overlap，可能会提高效率。当然之后需要使用`MPI_Wait()`, 等返回后再计算两个边缘行的数据。示意图如下:



进程数	1	4	16
总时间 (Wall time)	5.411865	1.613396	0.498471
误差	0.000205021	0.000205200	0.000205021
并行效率	100%	83.9%	67.9%
通信时间	0.0	0.237141	0.113764
通信占比	0.0	14.698256%	22.822622%

表 3: 策略 2 实验结果

代码结构大概如下 (源文件 `p2p.c`)

```
    int main(int argc, char* argv)
{
    MPI_init(&argc, &argv);
    do_some_initializaion();
    while(k <= max_iter)
    {
             MPI_Irecv(left_buffer,M,MPI_FLOAT,rank-1,111,MPI_COMM_WORLD,&request_1);
            MPI_Irecv(right_buffer,M,MPI_FLOAT,rank+1,111,MPI_COMM_WORLD,&request_2);
            MPI_Isend(send_buffer_left,M,MPI_FLOAT,rank-1,111,MPI_COMM_WORLD,&request_3);
            MPI_Isend(send_buffer_right,M,MPI_FLOAT,rank+1,111,MPI_COMM_WORLD,&request_4);

            update_middle_points();

            MPI_Wait(&request_1,&st1);
            MPI_Wait(&request_2,&st2);
            MPI_Wait(&request_3,&st3);
            MPI_Wait(&request_4,&st4);

            update_left_points();
            update_right_points();

    }
    post_processing();
    MPI_Finalize();
}
```

5.2 实验结果及分析 (profiling)

由上表可知,这种策略效果得到了很大提升,比较令人满意。其中通信时间计算的是`MPI_Wait()`函数等待的时间。

6 其他加速策略讨论

下面列举一些针对这个问题的加速策略 (但没有实施)

1. 利用对称性，解 $u(x,y)$ 关于 x ,y 是偶函数，可以将问题的规模缩小 $\frac{1}{4}$，不过需要重新编制解线性系统的算法；
2. 改变迭代中参数 α 的值，使用超松弛加速；
3. 尝试使用专业人员编制的线性代数加速库。

7 总结

我们分析了问题，制定了不同的并行算法，并进行了较为仔细的分析，得出了令人满意的加速策略。关于源代码，请参考文件夹中的 readme 文件。