《高性能程序设计基础》
实验报告

实验序号：实验三

实验名称：MPI 矩阵向量乘法的并行算法

姓　　名：劳马东

学　　号：16337113

1 实验题目

(1) 用 MPI 完成稠密矩阵向量乘法的并行算法

(a) 任务划分按照数据划分方法

i. 按输出数据划分

ii. 按输入数据划分

(b) 矩阵和向量从磁盘读入

(c) 结果输出到磁盘

(d) 矩阵和向量文件格式统一按三元组存放

(e) 矩阵很大，可能一个节点存不下

(2) 用 MPI 完成稀疏矩阵向量乘法的并行算法

(a) 根据非零元分布划分矩阵

2 实验目的

(1) 熟悉 MPI 全局聚集函数的使用；

(2) 掌握常见任务划分方法。

3 实验要求

(1) 计算算法的加速比，并列表

4 实验过程

4.1 稠密矩阵向量乘法

1、 矩阵划分

(a) 按行划分

输入文件的矩阵格式是按列给出，因此根进程每次读入一列，然后将这一列分发到各个进程。由于一列的大小不一定是进程数的整数倍，因此分发不是平均的，各个进程负载大小（行数）通过一个行负载计算模块得到。收到的数据需要经过一次矩阵变换——转置，因为一列在物理上存储为一维数组，本质上是一个行向量，而收到的数据逻辑上是列向量。如图 1。

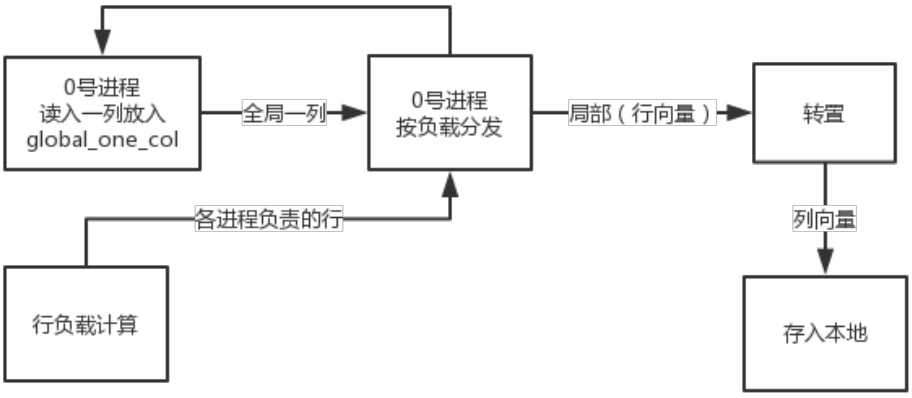


图 1: 行划分流程

图 2、3、4、5 是该过程的主要代码。*global_one_col* 是一个一维数组，一行结束标志为文件末尾或读入的列号为下一列；细节之处在于如果是由于读入了下一列而中止循环，那么下一列的第一行已经被读走了；一列的分发是异步的，因为 0 号进程之后不需要用到这次读的一列，如此可以提高性能。

```
if (info.rank == 0) {
    // 读入一列
    while (cnt--) {
        fscanf(file, "%d %d %lf", &row, &col, &entry);
        if (col == i + 1) {
            global_one_col.A[row - 1] = entry;
        } else    // 一列结束
            break;
    }
}
```

图 2: 读入一行

```
// 列结束推出循环时已经读入了下一列的第一行
if (info.rank == 0) {
    clear(global_one_col);
    global_one_col.A[row - 1] = entry;
}
```

图 3: 细节处理

```
// 异步分发
MPI_Request request;
Iscatterv(global_one_col, tmp, comm, 0, &request);
MPI_Wait(&request, MPI_STATUS_IGNORE);
```

图 4: 分发

```
// 转置
for (int j = 0; j < tmp.size; ++j)
    res.A[j * n + i] = tmp.A[j];
```

图 5: 矩阵转置

2、 局部稠密矩阵与向量相乘

由于本地矩阵的每一列都是完整的，因此该算法就是普通的矩阵-向量乘法。

需要强调的是，向量 x 也按行划分的方式分布在各个进程之中，依然是为了性能。假设各个进程接收字节的速率为：

$$d_{min} = \min\{d_0, d_1, ..., d_i, ..., d_{N-1}\} \tag{1}$$

无论如何每个进程在做矩阵-向量乘法的时候都需要得到完整的 x。假如 0 号进程读取向量并用广播的方式发送给其他进程。假设进程数为 N，向量字节数为 F，那么该过程至少需要的时间为：

$$t_{broadcast} = \frac{N \times F}{d_{min}} = O(N) \tag{2}$$

而采用先分发再蝶形收集的方法，至少需要的时间为：

$$t_2 = t_{scatter} + t_{all\_gather} = \sum_{i=1}^{N} \frac{\frac{F}{N}}{d_i} + \sum_{i=1}^{\log_2 N} \frac{i \times F}{d_{min}} = O((\log_2 N)^2) \tag{3}$$

此外，分发过程与 0 号进程读向量的过程可以并发，进一步提高性能。

```
Allgatherv(local_x, x, MPI_COMM_WORLD);

int local_i, j;
for (local_i = 0; local_i < local_A.size / n; local_i++) {
    local_y.A[local_i] = 0;
    for (j = 0; j < n; j++) {
        local_y.A[local_i] += local_A.A[local_i*n+j] * x.A[j];
    }
}
return local_y;
```

图 6: 向量聚集与乘法

3、 局部结果聚合

使用 MPI 的 gatherv 函数收集每个进程本地的 y 向量即可。

4.2 稀疏矩阵向量乘法

1、 矩阵划分——按元素划分

简单起见，首先读入完整的矩阵，然后调用 MPI 的 scatterv 函数按元素顺序分发。效率更高的方法是根据每个进程的负载读入相应个数的元素，然后异步 send 到对应进程。但由于在实验中发现读取整个矩阵的时间极短（不足 1 秒），因此采用简单方法。

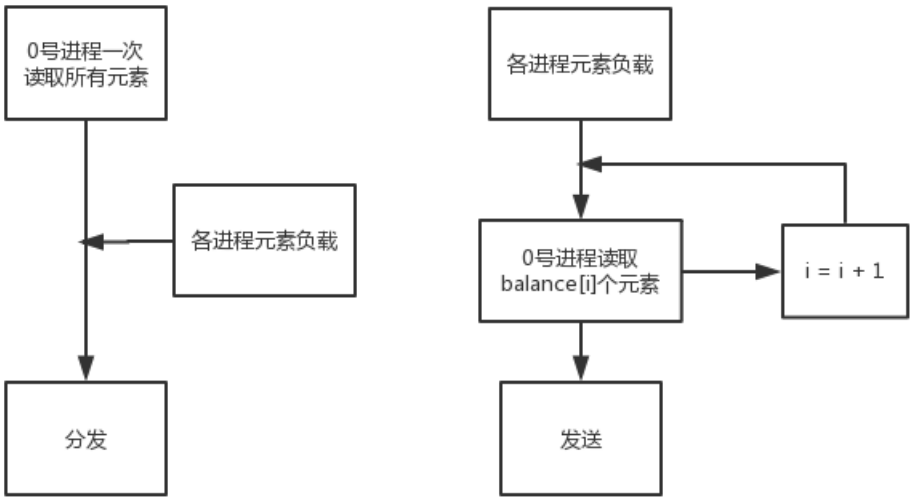


图 7: 按元素划分的两种方法

2、 局部矩阵与向量相乘

遍历局部矩阵中的每个元素，获得其行列下标，与全局向量 x 对应位置相乘求和，放入局部结果矩阵 *local_y*（与全局 y 大小相同）即可。

```
for (int i = 0; i < size; ++i) {
    int row = local_A[i].i;
    int col = local_A[i].j;
    local_y.A[row - 1] += local_A[i].value * x.A[col - 1];
}
```

图 8: 稀疏矩阵与向量乘法

3、 局部结果聚合

每个进程本地的 *local_y* 向量相加就是全局 y，因此调用 MPI 的 reduce 函数求和即可。

5 实验结果及分析

5.1 正确性验证

测试进程数为 4，测试矩阵与向量为：

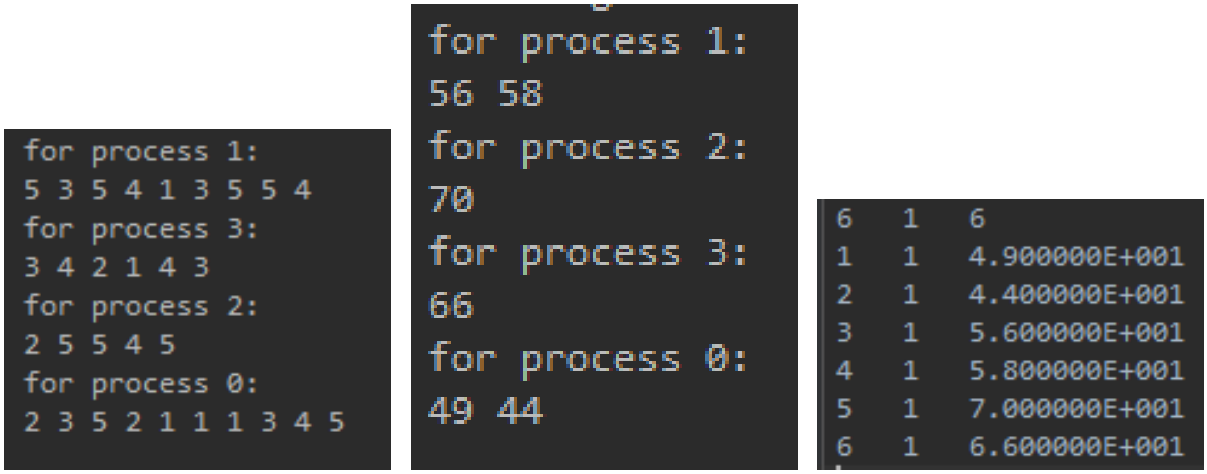
$$A = \begin{bmatrix} 2 & 3 & 5 & 2 & 1 & 1 \\ 1 & 0 & 3 & 4 & 5 & 0 \\ 5 & 3 & 5 & 4 & 1 & 0 \\ 0 & 0 & 3 & 5 & 5 & 4 \\ 2 & 5 & 5 & 4 & 0 & 5 \\ 3 & 4 & 2 & 1 & 4 & 3 \end{bmatrix} \qquad x = \begin{bmatrix} 3 \\ 4 \\ 4 \\ 1 \\ 5 \\ 4 \end{bmatrix} \tag{4}$$

1、 稠密矩阵

总共 6 行，进程 0 分到前两行（2 3 5 2 1 1 1 3 4 5，只输出非 0 元素），依此类推，结果正确；

进程 0 本地 y 为 $(49 = 2*3+3*4+5*4+2*1+1*5+1*4, 44)$，依此类推，结果正确；

全局 y 为 $(49, 44, 56, 58, 70, 66)$，正确。



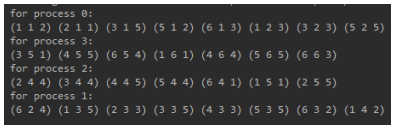
(a) 行划分结果　　(b) 进程本地 y　　(c) 全局 y

2、 稀疏矩阵

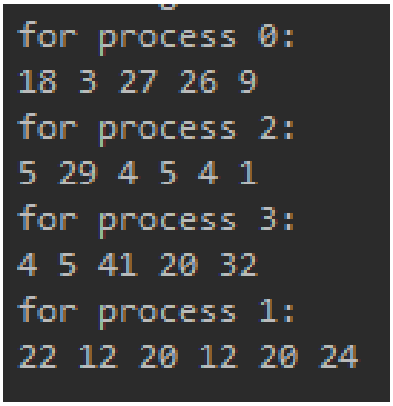
总共 30 个元素，进程 0 和 1 按顺序分到 8 个，进程 2、3 按顺序分到 7 个，正确；

进程 0 本地 y 为 $(18 = 2*3+3*4, 3, 27, 26, 9)$，以此类推，结果正确；

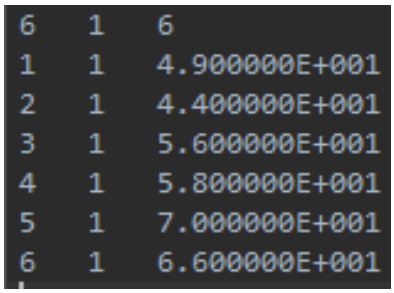
全局 y 正确。



(d) 行划分结果



(e) 进程本地 y

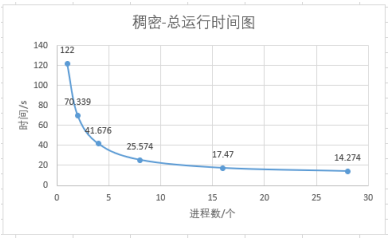


(f) 全局 y

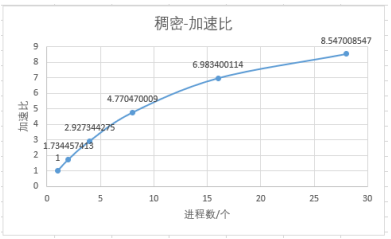
5.2 加速比计算

1、 稠密矩阵

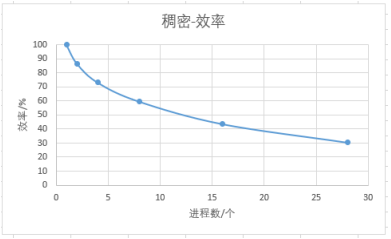
进程数	运行时间/秒	加速比	效率/%
1	122	1	100
2	70.339	1.73	86.72
4	41.676	2.92	73.18
8	25.574	4.77	59.63
16	17.47	6.98	43.64
28	14.274	8.55	30.52



(g) 时间



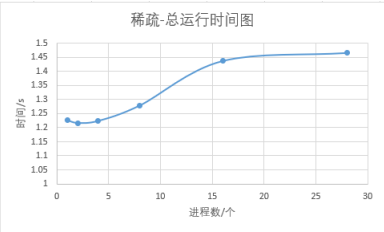
(h) 加速比



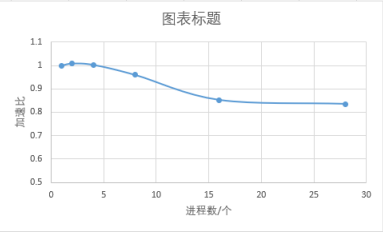
(i) 效率

2、 稀疏矩阵

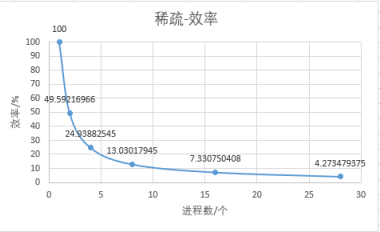
进程数	运行时间/秒	加速比	效率/%
1	1.226	1	100
2	1.216	1.008	50.41
4	1.223	1.002	25.06
8	1.278	0.959	11.99
16	1.438	0.852	5.33
28	1.467	0.836	2.98


稀疏-总运行时间图
时间/s
进程数/个

(j) 时间


图表标题
加速比
进程数/个

(k) 加速比


稀疏-效率
效率/%
进程数/个
49.59216966
24.93882545
13.03017945
7.330750408
4.273479375

(l) 效率