# Project 1: 矩阵相乘的并行计算实现

李冯福
学号：201418000206054

2015-6-8

## Contents

**1 问题描述** **1**

**2 问题解答** **2**
2.1 设计方法验证程序的对错 . . . . . 3
2.2 考察计算时间与矩阵大小的关系 . . . . . 3
2.2.1 串行执行时间 . . . . . 4
2.2.2 MPI 并行计算求解时间 . . . . . 4
2.2.3 MPI 并行计算加速比 . . . . . 5
2.3 改进的 MPI 矩阵乘积算法 . . . . . 6
2.4 论证上述关系的原因 . . . . . 7
2.4.1 计算时间与 $np$ 的关系 . . . . . 7
2.4.2 计算时间与 $m$ 的关系 . . . . . 7

**3 附录** **8**
3.1 代码文件说明 . . . . . 8
3.2 常用 MPI 函数介绍（C++） . . . . . 9

## 1 问题描述

**写程序完成矩阵相乘**

1. 设计方法验证程序的对错；
2. 考察计算时间与矩阵大小的关系；
3. 论证上述关系的原因；
4. 完成类 SCVector 的定义；
5. 设计并完成定义类 SCMatrix；
6. 使用 SCMatrix 完成矩阵乘积。

# 2 问题解答

**矩阵乘法问题描述如下：**

给定两个矩阵 $A$ 和矩阵 $B$。其中 A 的大小为 $m \times p$，$B$ 的大小为 $p \times n$。现在要求 $A$ 和 $B$ 的乘积：

$$C = A \times B$$

这里 $C$ 的大小为 $m \times n$。

如果用 $a_{ij}$ 和 $b_{ij}$ 分别表示 $A$ 和 $B$ 的第 $i$ 行 $j$ 列元素的话，上述问题等价于：

$$c_{ij} = \sum_{k=1}^{p} a_{ik} \times b_{kj}, \quad i = 1, 2, \cdots, m; j = 1, 2, \cdots, n \tag{1}$$

**解决方案：原始**

在 C++ 里面，问题 (1) 可以使用一个简单的多重循环来实现。算法如下：

**Input:** $A$、$B$
**Output:** $C$
1: **for** $i = 0$ to $m - 1$ **do**
2:  **for** $j = 0$ to $n - 1$ **do**
3:  按照公式 (1) 计算出 $c_{ij}$
4:  **end for**
5: **end for**

**算法 1:** 矩阵乘积的原始算法

**解决方案：改进成并行算法**

仔细观察上述问题 (1) 可以发现，求解 $C$ 的各行元素是独立进行的。因此，对于 $A$ 和 $B$ 比较大的情况，可以用并行计算来求解 $C$ 的各行。这样可以节省不少时间。当然，任何程序的时间复杂度和空间复杂度都是一个 trade-off 的过程，这就需要牺牲一定的内存空间。在内存足够的情况下，可以用 C++ 的 MPI 接口来实现多 CPU 核心的并行计算。

假设现在有 $np$ (number of processors) 个进程同时来并行的求解问题 (1)。由于 C++ 中的矩阵是按行存储的，因此，可以对 $A$ 按行进行分块，分块矩阵记为 $A_i (i = 1, 2, \cdots, np)$。即，

$$A = \begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_{np} \end{pmatrix} \tag{2}$$

进程 $i$ 求解 $A_i$ 与 $B$ 的乘积。另外，注意到 $m$ 不一定被 $np$ 整除，因此最后可能会剩余一小块矩阵。这部分可以由主线程负责求解。

有了上面的准备后，矩阵乘法的并行化可以分解为如下四个步骤：

step 1 由主进程对 $A$ 进行分块，并发送给所有次进程；

step 2 由主进程将 $B$ 发送给所有次进程；

**Input:** $A$、$B$、$np$
**Output:** $C$
1: 声明矩阵 $A$、$B$、$C$ 和 $bA$、$bC$，这里 $bA$ 和 $bC$ 为 $A$ 和 $C$ 按行分块的分块矩阵
2: 并行开始（调用MPI_Init 函数）
3: 获取总进程数和当前进程号（调用MPI_Comm_size 函数和MPI_Comm_rank 函数）
4: 给每个进程分配 $bA$、$B$ 和 $bC$ 的空间
5: **if** 当前进程为主进程 **then**
6: 给 $A$、$C$ 分配空间并设定 $A$ 和 $B$ 的值
7: **end if**
8: **Require**：同步（MPI_Barrier实现）
9: /* 矩阵乘积求解开始 */
10: 将主进程的 A 分配到其他进程（包括主进程）的 $bA$（MPI_Scatter 函数实现）
11: 将主进程的 B 复制给其他进程的 $B$（MPI_Bcast 函数实现）
12: 各个进程按照各自的 $bA$ 和 $B$ 计算出相应的 $bC$：$bC = bA \times B$
13: **Require**：同步
14: 将各个进程求出的结果 $bC$ 返回到主进程的 $C$（MPI_Gather 函数实现）
15: **if** 当前进程为主进程 **then**
16: 求解 $A$ 分块后多余的块和 $B$ 的乘积
17: **end if**
18: /* 矩阵乘积求解结束 */
19: 释放 $bA$、$B$ 和 $bC$ 的空间
20: **if** 当前进程为主进程 **then**
21: 释放 $A$、$C$ 的空间
22: **end if**
23: 并行结束（调用MPI_Finalize 函数）
24: **return** $C$

**算法 2:** 矩阵乘积的 MPI 并行算法

step 3 所有进程分别计算各自的矩阵乘积；

step 4 由主进程收集结果。

具体的，上述思路的 C++ MPI 接口实现如算法 2 所示。

## 2.1 设计方法验证程序的对错

验证并行算法2 算出的结果 $C$ 的对错，需要先知道正确的矩阵相乘结果。为此，假设用单个进程算出的结果是正确的。于是，用主进程算出真实的 $C_{true} = A \times B$，然后让算法2算出的 $C$ 和 $C_{true}$ 对比即可知道程序的正确性。

矩阵相等意味着它们的每一个元素都相等。假设数据是实数，在 C++ 中用 float 实现。那么比较两个 float 类型的变量是否相等的方法是设定一个较小的阈值，例如 $10^{-6}$。当这两个数的差的绝对值小于该阈值时，认为这两个数相等；否则不相等。

## 2.2 考察计算时间与矩阵大小的关系

前一部分已经验证了程序的正确性。为了考察计算时间，可以用时间函数 clock()。它在 time.h 中声明。由于并行程序有多个进程，因此，各个进程有各自的计算时间。而整个程序的计算时间就是这些进程中计算时间最多的那个时间。

注意到这里这里的计时只考虑矩阵求解的时间，不包括 $A$、$B$ 等内存的分配时间。

在后面的实验中，假设矩阵都是方阵（$m = p = n$），分为五种尺寸：$m$ = 1000、2000、3000、4000 和 5000。启动进程数 $np$ = 2、4、8、16、32、64。所有实验都是在拥有四块 Intel(R) Xeon(R) E5-2637 v2 @ 3.50GHz CPU 的机器（8 核 16 线程）上运行的。

### 2.2.1 串行执行时间

首先考察单个 CPU 核心下计算时间与矩阵大小的关系。（相当于 $np$ = 1）。以此作为基准来考察并行计算的效果。

理论上，矩阵计算的时间复杂度为 $O(mnp)$。因此，以 $m$ = 1000 为基准，$m$ = 2000、3000、4000 和 5000 的计算时间应该分别是 $m$ = 1000 时的 8、27、64 和 125 倍。

实际中的运算时间和矩阵大小的关系如下表：

| $m$ | 1000 | 2000 | 3000 | 4000 | 5000 |
|---|---|---|---|---|---|
| 运算时间（秒） | 3.6 | 31.5 | 126. 8 | 331.3 | 727.8 |
| 理论倍数 | 1 | 8 | 27 | 64 | 125 |
| 真实倍数 | 1 | 8.86 | 35.63 | 93.05 | 204.44 |

表 1: 单个核心下矩阵的运算时间与矩阵大小的关系表。

从表1中可以看出，随着矩阵大小的增加，矩阵乘积的运算时间急剧增加。并且，矩阵越大，运算时间偏离理论上的计算时间越大。

### 2.2.2 MPI 并行计算求解时间

利用算法2的 MPI 矩阵相乘计算时间与矩阵大小的关系如下表。

| $m$ \ $np$ | **1** | 2 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|
| 1000 | **3.6** | 1.8 | 0.9 | 0.7 | 0.7 | 0.5 | 0.4 |
| 2000 | **31.5** | 17.3 | 8.4 | 6.1 | 4.3 | 2.9 | 2.1 |
| 3000 | **126.8** | 63.2 | 31.9 | 20.6 | 16.8 | 10.9 | 7.5 |
| 4000 | **331.3** | 173.3 | 85.8 | 61.9 | 39.2 | 20.6 | 14.4 |
| 5000 | **727.8** | 355.2 | 178.5 | 131.3 | 78.7 | 46.1 | 23.7 |

表 2: 计算时间与矩阵大小 $m$ 以及进程数 $np$ 的关系表。单位：秒。

从表2中可以看出，随着 $np$ 的增加，计算时间呈下降趋势。当进程数达到 64 时，计算时间可以显著的减少。然而，试验中进程数为 64 时的总运行时间（包括内存分配时间、计算时间、内存回收时间等时间）并不比进程数为 32 时显著减少。这可能是因为进程数太多时，占用的系统内存太大的缘故。另外，由于机器资源的限制，没有跑更大的 $np$。

按照表2可以画出不同 $np$ 下的计算时间与矩阵大小的关系图，如图1所示。从图1中可以更直观的看出不同 $np$ 下计算时间随矩阵大小的变化图。

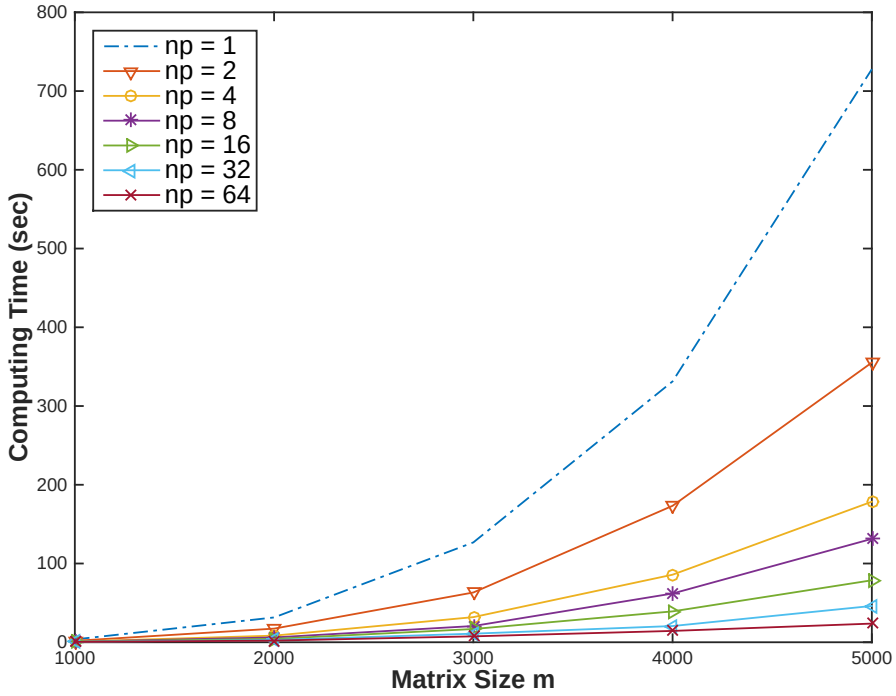


图 1: 不同进程数下矩阵相乘的计算时间与矩阵大小的关系图。

### 2.2.3 MPI 并行计算加速比

进一步，由并行计算时间和串行计算时间可以求出相应的并行加速比。计算公式为：

$$\text{speed up} = \frac{\text{单个进程的求解时间}}{\text{np 个进程的求解时间}}$$

从而，将表2中 np = 1 的那一列除以其他各列即可求得加速比。结果如表3所示。

| $m$ \ $np$ | **1** | 2 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|
| 1000 | **1** | 2.0 | 4.0 | 5.1 | 5.1 | 7.2 | 9.0 |
| 2000 | **1** | 1.8 | 3.8 | 5.2 | 7.3 | 10.9 | 15.0 |
| 3000 | **1** | 2.0 | 4.0 | 6.2 | 7.5 | 11.6 | 16.9 |
| 4000 | **1** | 1.9 | 3.9 | 5.5 | 8.5 | 16.1 | 23.0 |
| 5000 | **1** | 2.0 | 4.1 | 5.5 | 9.2 | 15.8 | 30.7 |

表 3: 加速比与矩阵大小以及进程数的关系表。单位：倍。

可以从两个角度来观察表格3。首先，固定 $m$，观察加速比与 $np$ 的关系（横向）。可以看出，当 $np$ 较小 ($np \leq 4$) 时，加速比与 $np$ 的关系近似成正比。此后，随着 $np$ 的增加，加速比的上升变得缓慢，不再与 $np$ 成正比关系。

其次，固定 $np$，观察加速比与矩阵大小的关系 (纵向)。当 $np$ 较小 ($np \leq 8$) 时，加速比对几种尺寸的矩阵几乎保持不变。而当 $np$ 较大时，矩阵尺寸越大，加速比越大。这说明，对于大尺寸矩阵的乘积，开辟越多的进程来计算越有利。当然，需要指出的是，进程数越多，需要的内存空间越大，这是程序实际运行中必须要考虑的问题。

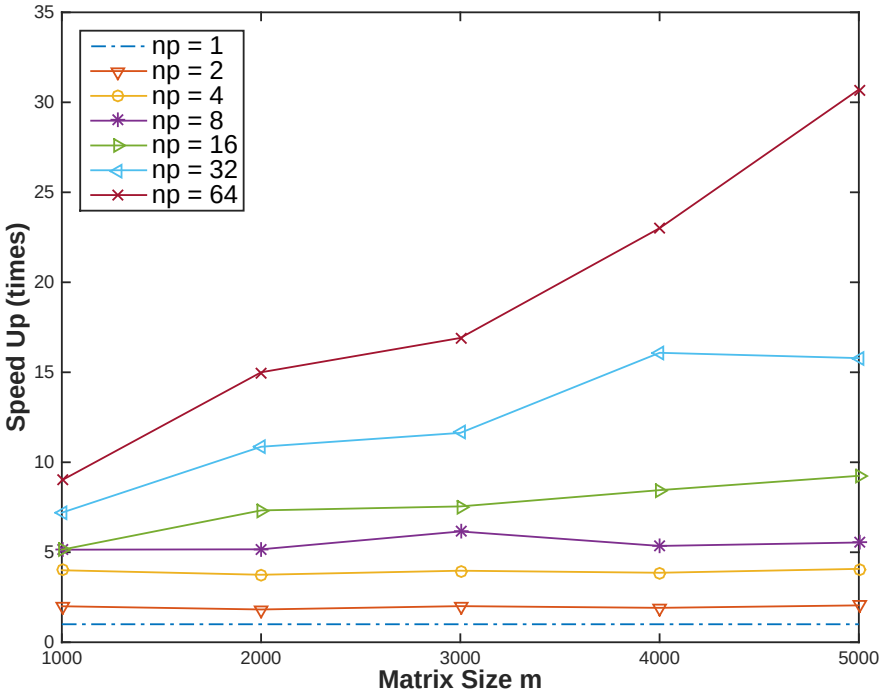


图 2: 不同启动进程数下矩阵相乘的加速比与矩阵大小的关系图。

## 2.3 改进的 MPI 矩阵乘积算法

前面说到程序运行时间和存储空间之间有一个 trade-off。算法2为了节省进程之间的信息交换，提前将各个进程的 $B$ 的空间分配好。这是一种牺牲空间换取时间的方法。对于特大矩阵的乘法，这种方法行不通。因为进程越多，需要的 $B$ 的备份越多。对于大尺寸的 $B$，这种方法代价太大。

考虑到内存问题，一种改进的方案是，让 $B$ 也像 $A$ 一样预先散播在各个进程中。之后，让 $B$ 的各个分块在各个进程之间传递，以此来达到完整的矩阵乘积。这种方案的优点是可以大大的节省存储空间，因为它只需要一份 $B$ 的备份。可能的不足是，这种方法会产生很大的数据传输开销。算法 3 总结了改进后的算法过程。

表 4 是算法 3 的运行结果。从表中可以看出，当矩阵尺寸 $m$ 较小时，时间的变化没有随着 $np$ 的增加成倍数较少，而是比较缓慢。然而，当矩阵尺寸达到 5000 以上时，随着 $np$ 的成倍增长，计算时间也近似的成倍减少。

| $m$ \ $np$ | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|---|---|
| 1000 | 3.6 | 1.5 | 0.8 | 0.6 | 0.4 | 0.3 (12) | 0.4 (9) | 0.6 (6) |
| 2000 | 31.5 | 12.1 | 6.3 | 5.2 | 3.1 | 2.3 (13) | 1.4 (21) | 2.3 (13) |
| 3000 | 126.8 | 41.0 | 20.6 | 13.3 | 10.5 | 7.0 (20) | 5.8 (24) | 4.7 (30) |
| 4000 | 331.3 | 96.7 | 47.3 | 32.4 | 24.2 | 12.7 (25) | 9.6 (34) | 6.7 (48) |
| 5000 | 727.8 | 183.7 | 92.7 | 59.2 | 47.8 | 26.4 (27) | 15 (46) | 8.2 (86) |
| 10000 | 5218.9 | - | - | - | - | 204 (26) | 109 (48) | 61 (85) |
| 15000 | 19590.5 | - | - | - | - | 672 (29) | 360 (54) | 215 (91) |
| 20000 | 43226.7 | - | - | - | - | 1620 (27) | 871 (50) | 511 (85) |

表 4: 计算时间与矩阵大小以及进程数的关系表（单位：秒）。括号里面是加速比。

## 2.4 论证上述关系的原因

### 2.4.1 计算时间与 $np$ 的关系

随着 $np$ 的增多，单个线程的计算量要下降。这导致总的时间下降。然而，进程之间的数据交换需要花费跟多的时间。这样，总计算时间的下降速度相对 $np$ 的增长速度更缓慢。这和实验中的观测一致。

### 2.4.2 计算时间与 $m$ 的关系

在用单个进程计算矩阵乘积时讨论过这个问题。对于多进程，计算时间主要由三部分组成。分别是：

1. 寻址时间;
2. 进程间的数据交换花费的时间;
3. 计算时间。

首先，进程越多，每个进程要处理的矩阵尺寸越小，这会使得寻址时间更少。$m$ 越大，这种效应越明显。其次，数据交换的总量是不变的。时间花费的瓶颈在于线程的等待。随着进程数的增多，等待的时间也会越少。（相当于使得每个进程都能“忙起来”。）最后，每个进程的计算时间和 $m$ 成三次方成正比，和 $np$ 成反比。

由此可以判断，随着 $m$ 的增加，寻址所花费的时间效应很明显。数据交换的时间相对弱一些。通过使用超多线程 $np = 128$，对于 $m = 15000$ 的情况可以达到 90 倍以上的加速比。

**Input:** $A$、$B$、$np$
**Output:** $C$

声明矩阵 A、B、C、bA、bB_send、bB_recv、bC 和 bC_send
并行开始
获取总进程数和当前进程号
每个进程分配 bA、bB_send、bB_recv、bC 和 bC_send 的空间
**if** 当前进程为主进程 **then**
  给 A、B 和 C 分配空间并设定 A 和 B 的值
**end if**
**Require:** 同步
/* 矩阵乘积求解开始 */
将主进程的 A 散播到其他进程（包括主进程）的 bA
将主进程的 B 散播到其他进程（包括主进程）的 bB_recv
复制 bB_send = bB_recv
**for** k = 0 to np -1 **do**
  求解 bC：bC = bA × bB_recv
  将 bC 复制到 bC_send 中
  **if** 当前进程为偶数进程 **then**
    复制 bC_send = bB_recv
    发送 bB_send 到下一个进程
    接收来自上一个进程的 bB_send 到 bB_recv
  **else**
    接收来自上一个进程的 bB_send 到 bB_recv
    发送 bB_send 到下一个进程
    复制 bB_send = bB_recv
  **end if**
**end for**
**Require:** 同步
**if** 当前进程为主进程 **then**
  求解多余分块的矩阵乘积
**end if**
/* 矩阵乘积求解结束 */
释放 bA、bB_send、bB_recv、bC 和 bC_send 的空间
**if** 当前进程为主进程 **then**
  释放 A、B 和 C 的空间
**end if**
并行结束
**return** $C$

**算法 3:** 矩阵乘积的 MPI 并行算法（改进）

# 3 附录

## 3.1 代码文件说明

源代码在 sources 文件夹下。

- utils.h 和 utils.cpp 一些实用函数的头文件和实现文件，例如初始化矩阵、打印矩阵等；
- SCVector.h 和 SCVector.cpp SCVector 类

- SCMatrix.h 和 SCVector.cpp SCMatrix 类
- mm_single_proc.cpp 算法1的实现
- mm_MPI.cpp 算法 2 的实现
- mm_MPI_update.cpp 算法 3 的实现
- mm_MPI_SCMatrix.cpp 算法 3 用 SCMatrix 类实现

安装和运行请参考 README 文件。

## 3.2 常用 MPI 函数介绍（C++）

**MPI_Init**

功能：初始化 MPI 运行时环境，即启动并行计算
原型：int MPI_Init(int *argc, char ***argv)

**MPI_Finalize**

功能：结束 MPI 运行时环境，即结束并行计算
原型：int MPI_Finalize(void)

**MPI_Comm_size**

功能：获取进程数 np
原型：int MPI_Comm_size(MPI_Comm comm, int *size)

**MPI_Comm_rank**

功能：获取当前进程号，范围：0 到 np - 1
原型：int MPI_Comm_rank(MPI_Comm comm, int *rank)

**MPI_Barrier**

功能：用于同步。只有所有进程都调用了该函数后，它才返回。
原型：int MPI_Barrier(MPI_Comm comm)

**MPI_Send**

功能：点对点阻塞通信的发送函数（阻塞发送：只有发送完成后才返回）
原型：int MPI_Send(void *message, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

- message：待发送消息的首地址
- count：发送的数据个数，如果发送的是向量，即为向量的长度

- datatype：发送的数据类型，如：MPI_INT，MPI_FLOAT，MPI_CHAR 等
- dest：接收消息的进程编号，范围：0 到 np - 1
- tag：消息的标签。对于重复发送同一类型的消息，标签将起到作用
- comm：发送进程和接收进程隶属的通信域，一般为 MPI_COMM_WORLD

**MPI_Recv**

功能：点对点阻塞通信的接收函数

原型：int MPI_Recv(void *message, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

- message：存放接收消息的变量的首地址
- count：接收的数据个数，大小一般应该等于对应的 MPI_Send 中的数据个数
- datatype：接收的数据类型，和 MPI_Send 中的数据类型对应
- source：发送该消息的进程编号
- tag：消息的标签。和 MPI_Send 中的消息标签一致
- comm：通信域
- status：接收消息返回的状态

**MPI_Bcast**

功能：广播（一对多）。用于将主进程内的数据**拷贝**到其他所有进程（包括该进程）。

原型：int MPI_Bcast(void* message, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

- message：待拷贝的数据首地址
- root：主进程

**MPI_Scatter**

功能：散播（一对多）。用于将主进程的数据等份的**分配**到其他所有进程（包括该进程）。

原型：int MPI_Bcast(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

- sendbuf：待分配数据的首地址
- sendcount：每个进程接收数据的个数（不是总个数!）
- recvbuf：待接收数据的首地址
- recvcount = sendcount （一般）
- root：主进程

Note：假设有 np 个进程，则 sendbuf 的大小（占用空间）应该是 np × count × sizeof(MPI_Datatype)。sendbuf 按照存储地址的连贯性分配为 np 等分，第 i 个进程接收第 i 段消息。等效为调用了 np 次 MPI_Send 函数。

**MPI_Gather**

功能：收集（多对一），散播的逆过程。用于将次进程的数据收集到主进程。

原型：int MPI_Gather(void* sendbuf, int sendcout, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)