

# 并行与分布式计算

## 第四次作业

姓名：张洪宾

班级：18 级计科超算

学号：18340208

# 1 问题描述

利用 Foster 并行程序设计方法计算  $1000 \times 1000$  的矩阵与  $1000 \times 1$  的向量之间的乘积, 要求清晰地呈现 Foster 并行程序设计的四个步骤。

## 2 实验环境

### 2.1 软件

- macOS Catalina
- Apple clang version 11.0.3 (clang-1103.0.32.29)  
Target: x86\_64-apple-darwin19.4.0  
Thread model: posix
- 采用 MPI 编程

### 2.2 硬件

所用机器型号是 MacBook Pro 2015

- 2.5 GHz 四核 Intel Core i7
- 16 GB 1600 MHz DDR3

## 3 解决方案

### 3.1 Foster 并行程序设计方法的具体步骤

- Partitioning
- Communication
- Agglomeration
- Mapping

### 3.2 串行解决矩阵乘向量的问题

为了便于分析问题，我先用串行的方法完成了矩阵乘向量的方法。首先我们先明确我们要解决的矩阵与向量相乘的方法：

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 5 \\ 11 \end{bmatrix}$$
$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = 1 \begin{bmatrix} 1 \\ 3 \end{bmatrix} + 2 \begin{bmatrix} 2 \\ 4 \end{bmatrix} = \begin{bmatrix} 5 \\ 11 \end{bmatrix}$$

图 1: 矩阵乘向量示例

可以看到向量的维度需要和矩阵的列数相同。

用串行的方法编写了矩阵乘向量的程序 (\src\serial.c)，代码如下：

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #define row 1000
4  #define col 1000
5
6  int matrix[row][col];
7  int vector[col];
8
9  int res[row];
10 int main(){
11     for(int i = 0; i < col; i++){
12         vector[i] = rand() % 10; //Initialize Vector
13     }
14     for(int i = 0; i < row; i++){
15         for(int j = 0; j < col; j++){
16             matrix[i][j] = rand() % 10; //Initialize Matrix
17         }
18     }
19     for(int i = 0; i < row; i++){
20         for(int j = 0; j < col; j++){
21             res[i] += matrix[i][j] * vector[j]; //Multiplication
```

```
22     }  
23 }  
24 return 0;  
25 }
```

通过该算法我们就可以对该问题使用 Foster 并行程序设计的方法。

### 3.3 Partitioning

首先我们要对问题进行划分。我们可以观察到：结果的向量不同项之间并不存在数据相关。也就是说，我们可以将问题划分成为矩阵的第  $i$  行和向量做点积得到结果向量的第  $i$  个元素。

所以我们可以将整个问题划分为 1000 个任务：1000 个  $1 \times 1000$  的向量和  $1000 \times 1$  的向量之间的点积，这一千个问题之间不存在数据相关。

### 3.4 Communication

我们需要执行的通信主要有两个方面：在主线程将矩阵和向量初始化结束后，将各个线程要执行所需要的数据发送给各个线程。在共享内存的编程模型中可以通过全局变量来完成通信，为了使得通信可以得到充分地体现，我选择了 MPI 编程。

MPI 编程需要用 MPI\_Send 和 MPI\_Recv 来进行点到点的通信，而对于矩阵乘以向量中的向量，该向量为所有进程所需，所以可以用 MPI\_Bcast 进行广播。

### 3.5 Agglomeration

如果每个任务都要开辟一个线程进行执行，那显然是对资源的一种浪费，不仅无法改善性能，还会因为开辟线程使得性能大大降低。

为此我们需要将子任务聚合。假设我们的资源允许我们开辟  $p$  个线程，那么我们就可以让一个程序执行  $1000/p$  个子任务。这样的话既可以实现并行，也可以减少资源调度对性能的影响。

### 3.6 Mapping

最后我们要将任务映射到资源上。为了充分利用局部性，我将连续到  $1000/p$  个任务分配到一个进程，然后在各个进程进行计算，最后将结果汇总。

为了证明计算结果的正确性，我在结果汇总完毕之后再用串行的方法计算了结果，并把它与并行计算得到的向量做比较，并打印结果是否正确信息。

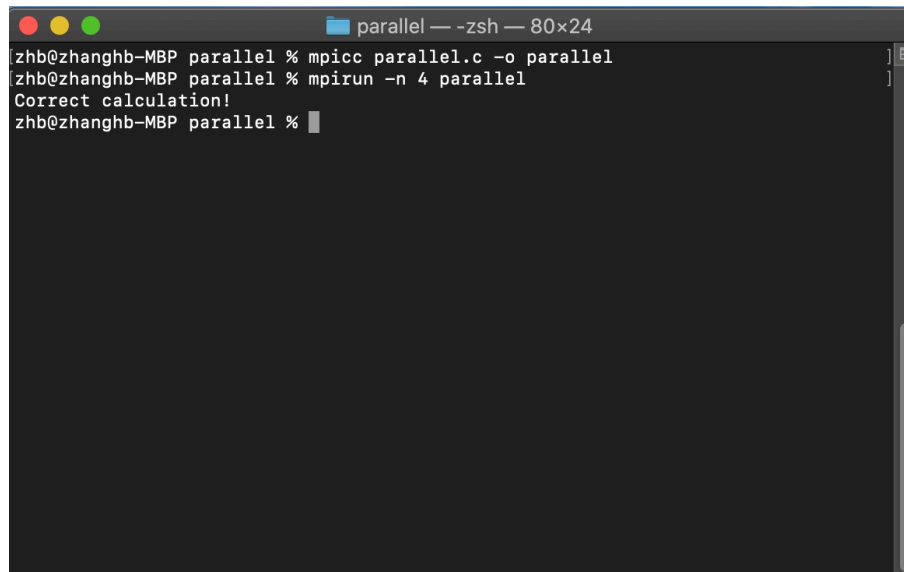
并行程序的主要代码如下，完整代码见 \src\parallel.c。

```
1  if(my_id == 0){
2      for(int i = 1; i < p; i++){
3          MPI_Send(matrix[i * row / p], col * row / p, MPI_INT, i, i, MPI_COMM_WORLD);
4      }
5  }
6  else{
7      MPI_Recv(matrix[my_id * row / p], col * row / p, MPI_INT, 0, my_id,
8      MPI_COMM_WORLD, &status_p);
9  }
10
11
12 MPI_Bcast(vector, col, MPI_INT, 0, MPI_COMM_WORLD);
13
14 for(int i = my_id * row / p; i < (my_id + 1) * row / p; i++){
15     for(int j = 0; j < col; j++){
16         res[i] += matrix[i][j] * vector[j];
17     }
18 }
19 if(my_id == 0){
20     for(int i = 1; i < p; i++){
21         MPI_Recv(res + i * row / p, row / p, MPI_INT, i, i, MPI_COMM_WORLD, &status_p);
22     }
23     serial();
24     int flag = 1;
25     for(int i = 0; i < row; i++){
26         if(res[i] != s_res[i]){
27             printf("Error!\n");
28             flag = 0;
29         }
30     }
31     if(flag){
32         printf("Correct calculation!\n");
33     }
34 }
35 else{
36     MPI_Send(res + my_id * row / p, row / p, MPI_INT, 0, my_id, MPI_COMM_WORLD);
37 }
```

## 4 实验结果

### 4.1 并行执行的正确性

打开终端，在程序中用 `mpicc` 对程序进行编译，用 `mpirun` 运行程序，在这里用了 4 个线程运行程序：



```
parallel --zsh-- 80x24
zhh@zhanghb-MBP parallel % mpicc parallel.c -o parallel
zhh@zhanghb-MBP parallel % mpirun -n 4 parallel
Correct calculation!
zhh@zhanghb-MBP parallel %
```

图 2: 运行程序

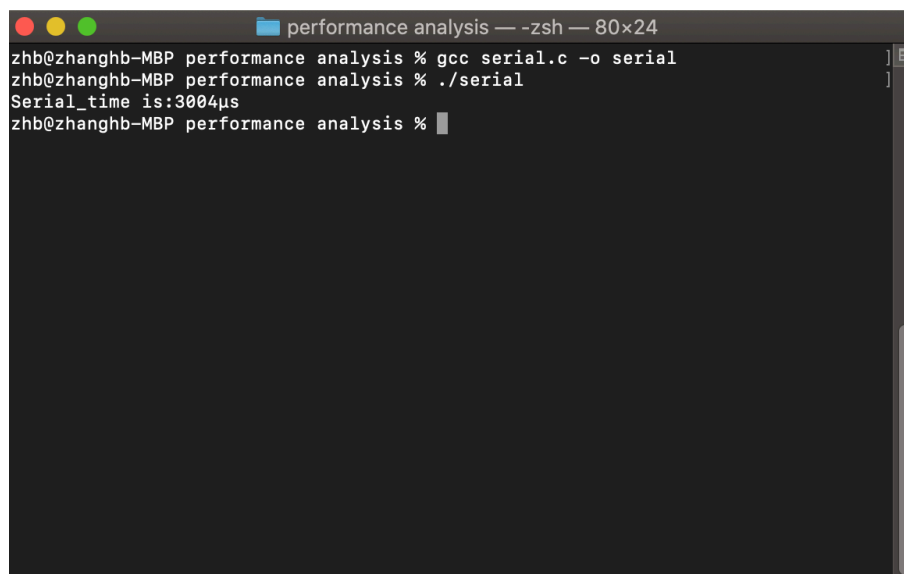
可以看到并行程序对随机生成的矩阵和向量的运算结果与串行程序的运算结果相同，说明并行程序运行可以得到正确的结果。

### 4.2 性能分析

在资源允许的情况下，我们可以用运行时间对程序的性能进行分析。在串行程序中，我们用 C 语言的 `gettimeofday` 函数对运行时间进行分析。而在并行程序中，我们使用 MPI 程序带有的 `MPI_Barrier(MPI_COMM_WORLD)` 对运算的起始时间进行同步，然后用 `MPI_Wtime` 函数得到每个线程的运行时间，最后用 `MPI_Reduce` 获取最大的线程运行时间，来得到整个程序的运行时间。

我把稍微修改后的代码，在其中删除了对并行程序正确性验证的部分。放到了 `performance analysis` 文件夹下，用与性能的分析。

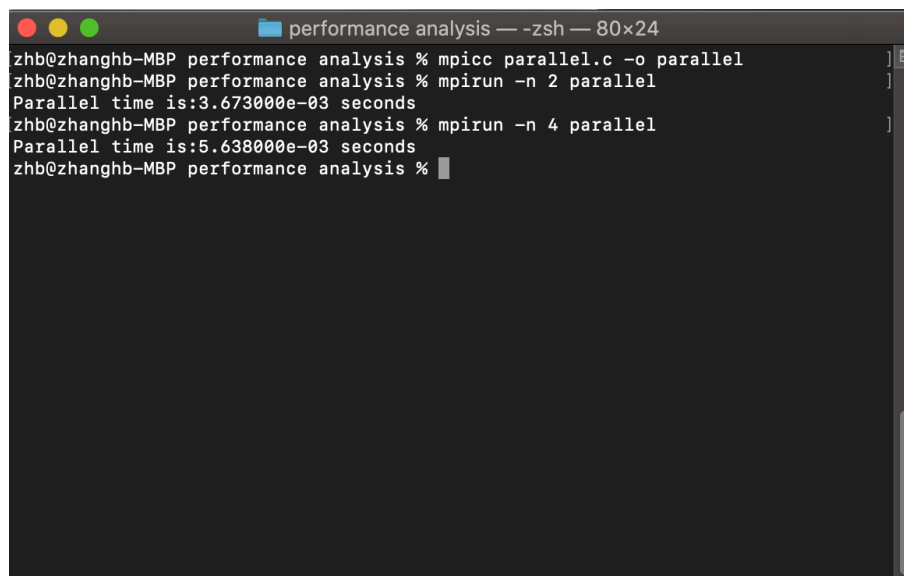
对串行程序进行性能分析：

A terminal window titled "performance analysis — zsh — 80x24" showing the execution of a serial program. The user runs 'gcc serial.c -o serial' and then './serial'. The output shows 'Serial\_time is:3004μs'.

```
zhh@zhanghb-MBP performance analysis % gcc serial.c -o serial
zhh@zhanghb-MBP performance analysis % ./serial
Serial_time is:3004μs
zhh@zhanghb-MBP performance analysis %
```

图 3: 串行程序的运行时间

而对于并行程序，我分别让进程数量为 2 和 4，进行运算：

A terminal window titled "performance analysis — zsh — 80x24" showing the execution of a parallel program with 2 and 4 processes. The user runs 'mpicc parallel.c -o parallel', then 'mpirun -n 2 parallel', and finally 'mpirun -n 4 parallel'. The output shows 'Parallel time is:3.673000e-03 seconds' for 2 processes and 'Parallel time is:5.638000e-03 seconds' for 4 processes.

```
zhh@zhanghb-MBP performance analysis % mpicc parallel.c -o parallel
zhh@zhanghb-MBP performance analysis % mpirun -n 2 parallel
Parallel time is:3.673000e-03 seconds
zhh@zhanghb-MBP performance analysis % mpirun -n 4 parallel
Parallel time is:5.638000e-03 seconds
zhh@zhanghb-MBP performance analysis %
```

图 4: 并行程序的运行时间

发现结果不仅没有提升，反而随着线程数量的增多，运行时间也增多。

这也在意料之中。MPI 使用了大量的通信，通信的开销其实非常巨大，然后由于问题的规模并不是很大，仅为  $1000 \times 1000$  的矩阵规模，所以并行化的性能提升并不能抵消通信带来的巨大开销。

## 5 遇到的问题及解决方法

在测量运行性能的时候，我发现如果在 MPI 中使用 `gettimeofday` 函数，随着线程数量的增加，运行时间没有明显的变化，不符合我的预期。于是我参考了《并行程序导论》，用 `MPI_Barrier` 和 `MPI_Wtime`，测量得到了比较准确的时间。