**湖南大学信息科学与工程学院本科生实验报告**

**（2022学年秋季学期）**

课程名称：**高性能计算导论**  任课教师：**罗辉章 批改人：**

|  |  |  |  |
| --- | --- | --- | --- |
| 年级+班级 | **计科1901** | 专业（方向） | **计算机科学与技术** |
| 学号 | **201908010102** | 姓名 | **刘德龙** |
| Email | **1589075757@qq.com** | 完成日期 | **2022年9月28日星期三** |

# 实验目的(20分)

学习mpi，使用mpi实现并行通用矩阵乘法

# 实验过程和核心代码(40分)

## 通过MPI实现通用矩阵乘法

### 描述

通过 MPI 实现通用矩阵乘法（实验1）的并行版本，MPI并行进程（rank size）从 1 增加至 8，矩阵规模从 512 增加至 2048.

**输入：**M , N, K 三个整数（512 ~2048）

随机生成 M\*N 和N\*K 的两个矩阵 A,B,对这两个矩阵做乘法得到矩阵 C.

**输出：**A,B,C 三个矩阵以及矩阵计算的时间

### 思路

第0个进程生成矩阵A，B,把参数M,N,K以及矩阵A的部分行，完整的矩阵B发送给所有其他进程。

除了0号进程，每个进程接受M,N,K，以及矩阵A的部分行，矩阵B的所有列数据。

所有进程负责计算出C矩阵的M/C行（C为进程数量）。然后把数据都交给0号进程。

通过if-else 来区分不同进程，执行不同的逻辑。

为了使得矩阵A的行能够平均分配给各个进程，需要把M调整到进程数的整数倍，拓展的行用0补充。

### 核心代码

第0号进程

int fix\_M=ceil(M/(double)world\_size)\*world\_size; //使得行数是进程数的整数倍，拓展的行用0补充。方便给每个核平均分配行。

int \*\*A=create\_matrix(fix\_M,N);

int \*\*B=create\_matrix(N,K);

int \*\*C=create\_matrix(fix\_M,K);

init\_matrix(M,N,A);

memset(A[M],0,sizeof(int)\*(fix\_M-M)\*N);

init\_matrix(N,K,B);

timer.begin();

int wn=(fix\_M/world\_size); // wn是各个进程实际需要计算的行数

// 发送3个参数以及矩阵A,B

for(int d=1;d<world\_size;d++)

{

    int temp=M;

    M=wn;

    MPI\_Send(M\_N\_K,3,MPI\_INT,d,0,MPI\_COMM\_WORLD);

    M=temp; // //实际发送到其他核的M值是其他核需要算的行数，不是fix\_M,也不是真正的M。

    MPI\_Send(A[d\*wn],wn\*N,MPI\_INT,d,1,MPI\_COMM\_WORLD);

    MPI\_Send(B[0],N\*K,MPI\_INT,d,2,MPI\_COMM\_WORLD);

}

// 计算自己要算的那部分

for(int i=0;i<wn;i++)

{

    for(int j=0;j<K;j++)

    {

        C[i][j]=0;

        for(int k=0;k<N;k++)

        {

            C[i][j]+=A[i][k]\*B[k][j];

        }

    }

}

//接收其他进程算的那部分到相应的位置

for(int d=1;d<world\_size;d++)

{

    MPI\_Recv(C[d\*wn],wn\*K,MPI\_INT,d,3,MPI\_COMM\_WORLD,MPI\_STATUS\_IGNORE);

}

auto cost=timer.end\_ns();

其他进程

MPI\_Recv(M\_N\_K,3,MPI\_INT,0,0,MPI\_COMM\_WORLD,MPI\_STATUS\_IGNORE);

int \*\*A=create\_matrix(M,N);

int \*\*B=create\_matrix(N,K);

int \*\*C=create\_matrix(M,K);

//cout<<"process\_rank"<<world\_rank<<" mnk:"<<M<<" "<<N<<" "<<K<<endl;

MPI\_Recv(A[0],M\*N,MPI\_INT,0,1,MPI\_COMM\_WORLD,MPI\_STATUS\_IGNORE);

MPI\_Recv(B[0],N\*K,MPI\_INT,0,2,MPI\_COMM\_WORLD,MPI\_STATUS\_IGNORE);

for(int i=0;i<M;i++)

{

    for(int j=0;j<K;j++)

    {

        C[i][j]=0;

        for(int k=0;k<N;k++)

        {

            C[i][j]+=A[i][k]\*B[k][j];

        }

    }

}

MPI\_Send(C[0],M\*K,MPI\_INT,0,3,MPI\_COMM\_WORLD);

### 遇到困难

#### 困难1：

如果只用一个进程运行，可以跑通。如果有3个进程运行。则出现有一个核没接收到数据的现象。然后程序提前停止。代码以及现象如下：

第0个进程：

printf("please input M N K:");

int M\_N\_K[3];int &M=M\_N\_K[0];int &N=M\_N\_K[1];int &K=M\_N\_K[2];

scanf("%d %d %d",&M,&N,&K);

int \*\*A=create\_matrix(M,N);

int \*\*B=create\_matrix(N,K);

int \*\*C=create\_matrix(M,K);

init\_matrix(M,N,A);

print\_matrix(A,M,N);

init\_matrix(N,K,B);

timer.begin();

for(int d=1;d<world\_size;d++)

{

    MPI\_Send(M\_N\_K,3,MPI\_INT,d,0,MPI\_COMM\_WORLD);

    MPI\_Send(A[0],M\*N,MPI\_INT,d,1,MPI\_COMM\_WORLD);

    MPI\_Send(B[0],N\*K,MPI\_INT,d,2,MPI\_COMM\_WORLD);

}

cout<<"process\_rank"<<world\_rank<<" on call"<<endl;

用随机数填充完矩阵A后，打印矩阵A。给其他进程发送完数据之后，打印调试信息。

其他进程：

cout<<"process\_rank"<<world\_rank<<" on call"<<endl;

int M\_N\_K[3];int &M=M\_N\_K[0];int &N=M\_N\_K[1];int &K=M\_N\_K[2];

MPI\_Recv(M\_N\_K,3,MPI\_INT,0,0,MPI\_COMM\_WORLD,MPI\_STATUS\_IGNORE);

int \*\*A=create\_matrix(M,N);

int \*\*B=create\_matrix(N,K);

int \*\*C=create\_matrix(M,K);

MPI\_Recv(A[0],M\*N,MPI\_INT,0,1,MPI\_COMM\_WORLD,MPI\_STATUS\_IGNORE);

MPI\_Recv(B[0],N\*K,MPI\_INT,0,2,MPI\_COMM\_WORLD,MPI\_STATUS\_IGNORE);

cout<<"process\_rank"<<world\_rank<<" on call2"<<endl;

print\_matrix(A,M,N);

进入的时候就打印这个进程on call，接收完打印on call2并打印接收到的矩阵A。

运行结果：

liu@manager:~/HPClab3$ make

mpicxx -o mpi\_gemm src/MPI\_GEMM.cpp src/common/matrix\_tool.cpp

mpirun -np 3 -hosts localhost ./mpi\_gemm

please input M N K:process\_rank1 on call

process\_rank2 on call

3 4 5

         0          4         15          8

        14          7         15          8

         8         11         13         14

process\_rank0 on call

process\_rank1 on call2

         0          4         15          8

        14          7         15          8

         8         11         13         14

试了多次，都是rank2不打印接收到的数据。

并且，程序运行到这里直接退出。并没有其他打印了（后面0号进程应该要打印矩阵A,B,计算好的C和用时的。

**解决：**

这个问题是因为程序的最后没有这行代码，加上即可：

MPI\_Finalize();

**总结：**

这个问题输入程序异常退出，没有执行完整个代码流程，但也没有提示任何错误就结束了。

## 基于 MPI 的通用矩阵乘法优化

### 描述

分别采用 MPI 点对点通信和 MPI 集合通信实现矩阵乘法中的进程之间通信，并比较两种实现方式的性能。

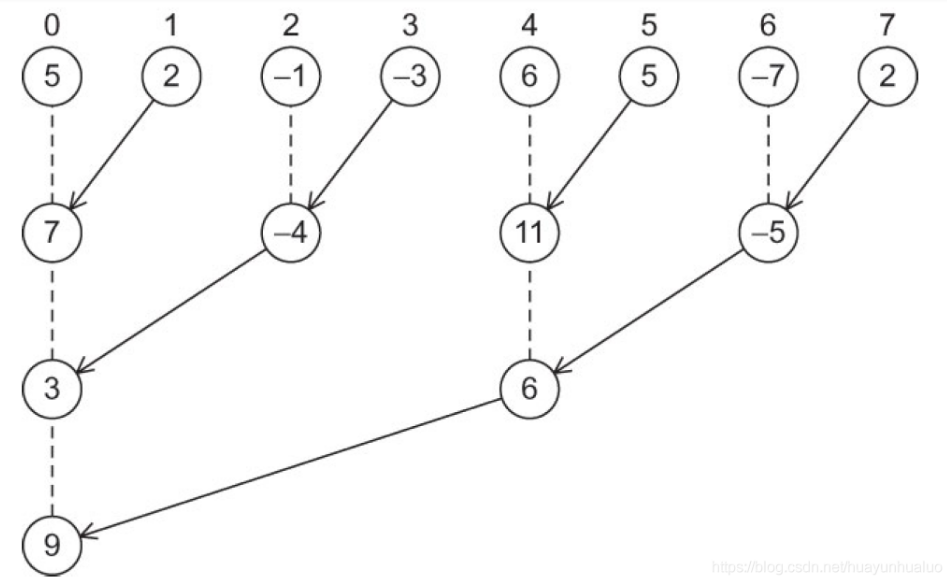
### 分析

2.1中的通信方式就是点对点通信。下面讨论集合通信。

集合通信就是把所有进程集中看待，通过一些策略加速信息整合过程。集合通信有reduce和allreduce。

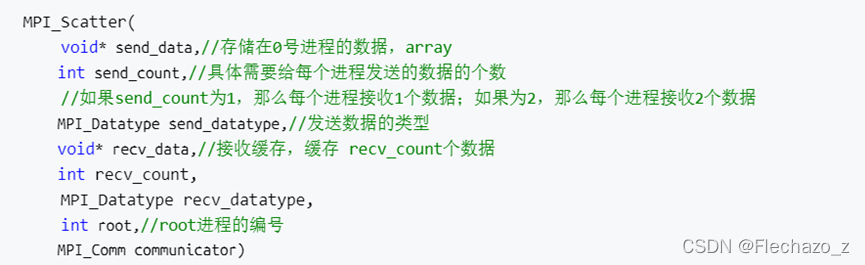
reduce是把所有数据汇总到一个核上，比如每个进程都有一个整数，要把所有进程的整数求和，得到的结果保存在第1个进程中。 allreduce就是得到的和要保存到所有的核中。

一些策略可以加快全局就和的过程。如图：

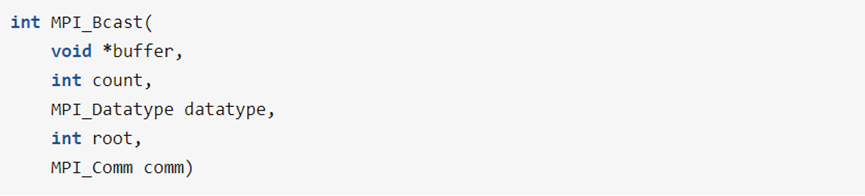


这个方案就比1-7进程向0号进程发送数据，0号进程计算和要快。带宽上，上图方案0号进程只需承担4次通信，而朴素方案要承担7次通信。计算次数上，0号进程只需要计算3次。而朴素方案要进行7次求和。

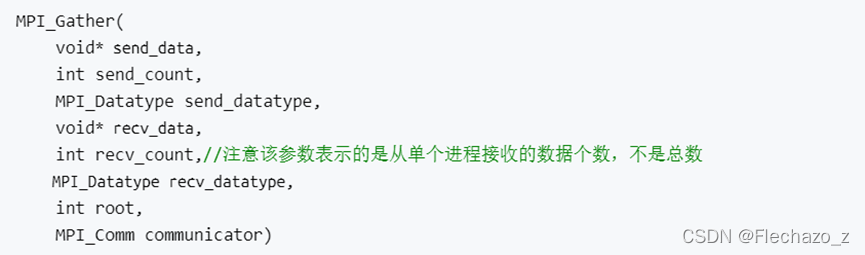
Mpi提供了若干集合通信函数，我们可以使用这些集合函数，而不必知道底层是怎么运作的。理论上，集合通信要快于点对点通信。下面介绍3个本实验用到的集合通信



scatter用来把数据平均分给集合里面的各个进程。可以用来发布A矩阵。数据按进程号分发，第0号进程获得的是第一组数据。



Bcast用来广播数据，用来发布B矩阵。



Gather用来收集数据。每个进程(包含根进程)将自己send数据缓存中的数据发送给根进程，根进程接收 所有发来的消并按进程标识排序。收集数据时，无论时0号进程还是其他进程，都是调用这个函数。

### 核心代码

在集合函数里，比如scatter和gather，会给自己也发送一份数据，同时自己也会收到数据。分发的数据在send\_data中，收到的数据存在recv\_data中。所以0号节点除了分发和收集结果，自己也是需要参与计算的。

代码中，带t前缀的变量是指需要汇总或者分发的数据。要注意的是，无论哪个进程，都要有send\_data和recv\_data，一开始我以为只有0号进程需要有send\_data,其他线程的send\_data是nul，这是不对的l。每个进程都需要开辟完整的tA,tB矩阵空间，但只有0号进程需要初始化tA,tB矩阵。

另外，为了使得scatter函数能够把A矩阵平均分给各个进程，需要对A矩阵拓展，使得A矩阵的行数是进程数的整数倍，拓展的行用0填充。

代码的核心思路是

1. 开辟存储5个参数oM,N,K,M,fM的空间。oM是A矩阵实际的行数，M是每个进程需要计算的行数，fM是拓展行后，A矩阵的行数的空间。
2. 0号进程接收并计算参数oM,N,K,M,fM。
3. 用Bcast广播0号进程的5个参数。
4. 所有进程根据5个参数开辟内存空间tA,B,tC,A,C。tA和tC的行数是fM。A和C的行数是M,
5. 0号进程用随机数填充tA，tB矩阵。
6. 用scatter分发矩阵tA，用Bcast分发矩阵tB。
7. 计算矩阵乘法。
8. 把结果用gather汇总到0号进程的tCz中。

int \*\*tA;int \*\*B;int \*\*tC;

//1.开辟存储5个参数oM,N,K,M,fM的空间。oM是A矩阵实际的行数，M是每个进程需要计算的行数，fM是拓展行后，A矩阵的行数的空间。

int arg[5];int &M=arg[0];int &N=arg[1];int &K=arg[2];int &oM=arg[3];int &fM=arg[4];

/\*

M是工作进程要计算的矩阵A和C的行数，N是矩阵A的列数，K矩阵B的列数。

oM是矩阵A本来应该的行数，

fM是拓展后的矩阵A的行数（拓展矩阵A的行数是为了使得能够平均分给每个进程）

\*/

// 2.0号进程接收并计算参数oM,N,K,M,fM。

if(world\_rank==0)

{

    if(argc==4)

    {

        oM=atoi(argv[1]);

        N=atoi(argv[2]);

        K=atoi(argv[3]);

    }

    else

    {

        printf("please input M N K:");

        scanf("%d %d %d",&oM,&N,&K);

    }

    M=ceil(oM/(double)world\_size);

    fM=M\*world\_size;

}

//3.用Bcast广播0号进程的5个参数。

MPI\_Bcast(arg,5,MPI\_INT,0,MPI\_COMM\_WORLD);

//4.所有进程根据5个参数开辟内存空间tA,B,tC,A,C。tA和tC的行数是fM。A和C的行数是M,

tA=create\_matrix(fM,N);

B=create\_matrix(N,K);

tC=create\_matrix(fM,K);

int \*\*A;int \*\*C;

A=create\_matrix(M,N);

C=create\_matrix(M,K);

//5.0号进程用随机数填充tA，tB矩阵。

if(world\_rank==0)

{

    init\_matrix(oM,N,tA);

    memset(tA[oM],0,sizeof(int)\*(fM-oM)\*N);

    init\_matrix(N,K,B);

    timer.begin();

}

//6.用scatter分发矩阵tA，用Bcast分发矩阵tB。

MPI\_Scatter(tA[0],M\*N,MPI\_INT,A[0],M\*N,MPI\_INT,0,MPI\_COMM\_WORLD);

MPI\_Bcast(B[0],N\*K,MPI\_INT,0,MPI\_COMM\_WORLD);

//计算矩阵乘法。

for(int i=0;i<M;i++)

{

    for(int j=0;j<K;j++)

    {

        C[i][j]=0;

        for(int k=0;k<N;k++)

        {

            C[i][j]+=A[i][k]\*B[k][j];

        }

    }

}

//8.把结果用gather汇总到0号进程的tC中。

MPI\_Gather(C[0],M\*K,MPI\_INT,tC[0],M\*K,MPI\_INT,0,MPI\_COMM\_WORLD);

if(world\_rank==0)

{

    auto cost=timer.end\_ns();

    printf("%d %d %lld\n",oM,world\_size,cost/1000000);

}

## 改造实验1 成矩阵乘法库函数

### 描述

将Lab1 的矩阵乘法改造为一个标准的库函数 matrix\_multiply（函数实现文件和函数头文件），输入参数为三个完整定义矩阵（A,B,C），定义方式没有具体要求，可以是二维矩阵，也可以是 struct 等。在 Linux 系统中将此函数编译为.so 文件，由其他程序调用。

### 步骤

只需把实验1的4个矩阵乘法的.cpp文件拷贝到实验3的工程目录，然后编写一个头文件如下：

#ifndef MATRIX\_MULTI\_H

#define MATRIX\_MULTI\_H

#include <vector>

using namespace std;

void CoppersmithWinograd(int n,int\*\* A,int \*\*B,int\*\* C);

void loop\_expansion(int n,int \*\*A,int \*\*B,int \*\*C);

int Strassen(int N, int \*\*MatrixA, int \*\*MatrixB, int \*\*MatrixC);

void universal(int N, int \*\*A, int \*\*B, int \*\*C);

void init\_matrix(vector<vector<int>> &A);

void init\_matrix(int N,int\*\* A);

void init\_matrix(int M,int N,int \*\*A);

int\*\* create\_matrix(int N);

int\*\* create\_matrix(int M,int N);

void delete\_matrix(int \*\*A);

void print\_matrix(vector<vector<int>> &A);

void print\_matrix(int \*\*A,int N);

void print\_matrix(int \*\*A,int M,int N);

void copy\_matrix(int \*\*src,int \*\*dst,int N);

void copy\_matrix(int \*\*src,int \*\*dst,int M,int N);

#endif

再编写一个demo程序链接到这个库

#include"matrix\_multi.h"

int main()

{

    int \*\*A=create\_matrix(3,3);

    int \*\*B=create\_matrix(3,3);

    int \*\*C=create\_matrix(3,3);

    universal(3,A,B,C);

    Strassen(3,A,B,C);

    loop\_expansion(3,A,B,C);

    CoppersmithWinograd(3,A,B,C);

}

然后在makefile中加上两个规则：

matrix\_mul:CoppersmithWinograd.cpp loop\_expansion.cpp matrix\_tool.cpp Strassen.cpp universal.cpp

    g++ $^ -fPIC -shared -o lib$@.so

so\_demo:so\_demo.cpp matrix\_mul.so

    g++ $< -L. -lmatrix\_mul -o $@

在命令行上输入make matrix\_mul && make so\_demo就得到了一个libmatrix\_mul.so和so\_demo可执行程序。

# 实验结果(30分)

## 点对点通信

Makefile 脚本4种规模8种进程数通过一个命令make运行。

scale:=512 1024 1536 2048

core\_num:=1 2 3 4 5 6 7 8

mpi\_gemm:MPI\_GEMM.cpp matrix\_tool.cpp

    mpicxx -o $@ $^

    for s in $(scale); do \

        for c in $(core\_num); do \

            mpirun -np $$c -hosts 127.0.0.1 ./mpi\_gemm $$s $$s $$s; \

        done \

    done

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **求和项:用时/ms** | **规模** |  |  |  |
| **进程数** | **512** | **1024** | **1536** | **2048** |
| 1 | 849 | 7850 | 37443 | 145807 |
| 2 | 447 | 4797 | 20418 | 83371 |
| 3 | 347 | 3650 | 16289 | 58299 |
| 4 | 307 | 3129 | 13412 | 49738 |
| 5 | 381 | 3519 | 13118 | 49735 |
| 6 | 499 | 3410 | 13145 | 50944 |
| 7 | 500 | 3509 | 13085 | 51248 |
| 8 | 428 | 3582 | 13418 | 54484 |

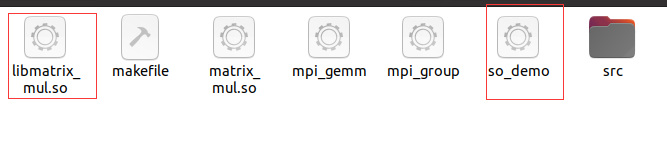
我的虚拟机分配了4个核心，所以对于不同的规模，大概都是到4个进程，再增加进程就不再有性能提升，甚至还因为通信成本增加而性能倒退。

## 集合通信

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **求和项:用时/ms** | **规模** |  |  |  |
| **进程数** | **512** | **1024** | **1536** | **2048** |
| 1 | 840 | 7841 | 36466 | 146536 |
| 2 | 446 | 4634 | 20097 | 81271 |
| 3 | 316 | 3606 | 15410 | 71275 |
| 4 | 286 | 2836 | 12077 | 53646 |
| 5 | 368 | 3194 | 13146 | 57454 |
| 6 | 450 | 3774 | 15143 | 62864 |
| 7 | 454 | 3189 | 15200 | 63642 |
| 8 | 586 | 3517 | 14433 | 51291 |

可以看到，在我的机子上，集合通信的用时，在核心数少时，比p2p好，但核心数变多时，比p2p差，我也不知道为什么，这和理论相反。

## 改造实验1 成矩阵乘法库函数



# 实验感想(10分)

通过本实验实践了mpi的点对点通信和集合通信。其中集合通信通过一个函数达到数据分发和收集的功能，没有分开成send函数和receive函数。我感觉时很妙的，但是也带来一些不方便的地方，比如接收者可能不需要分发者的空间，但需要为接收者开辟空间、所有进程，scatter函数的位置需要在同一位置，不能写在不同进程的代码区（也就是不能写在if(world\_rank==1)里面）。

矩阵乘法不需要使用reduce和allreduce集合通信操作，因此没有体验到这两个操作的美。

我的实验结果：在进程数超过物理核心数时，集合通信比点对点通信还慢。这样我很费解。