实验目的

1. 通过MPI实现通用矩阵乘法
2. 基于MPI的通用矩阵乘法优化
3. 改造Lab1成矩阵乘法库函数
4. 构造MPI版本矩阵乘法加速比和并行效率表

实验过程和核心代码

点对点通信

```
srand((unsigned)time(NULL));
for(i=0;i<size;i++)
    for(j=0;j<size;j++)
        a[i*size+j] = rand()%5;

for(i=0;i<size;i++)
    for(j=0;j<size;j++)
        b[i*size+j] = rand()%5;
start = MPI_Wtime();
//将矩阵N发送给其他从进程
for (i=1;i<numprocs;i++)
{
        MPI_Send(b,size*size,MPI_INT,i,0,MPI_COMM_WORLD);
}
//依次将a的各行发送给各从进程
for (l=1; l<numprocs; l++)
{
    MPI_Send(a+(l-1)*line*size,size*line,MPI_INT,l,1,MPI_COMM_WORLD);
}
//接收从进程计算的结果
for (k=1;k<numprocs;k++)
{
    MPI_Recv(ans,line*size,MPI_INT,k,3,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    //将结果传递给数组c
    for (i=0;i<line;i++)
    {
        for (j=0;j<size;j++)
        {
            c[((k-1)*line+i)*size+j] = ans[i*size+j];
        }

    }
}
//计算a剩下的数据
for (i=(numprocs-1)*line;i<size;i++)
{
    for (j=0;j<size;j++)
```

```
            {
                int temp=0;
                for (k=0;k<size;k++)
                    temp += a[i*size+k]*b[k*size+j];
                c[i*size+j] = temp;
            }
        }
        //统计时间
        stop = MPI_Wtime();
```

这段代码是将矩阵b发送给其他进程，将a的各行数据一部分一部分的发送给其他进程，然后把a的剩余部分再进行计算。

```
//接收广播的数据(矩阵b)
        MPI_Recv(b,size*size,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

        MPI_Recv(buffer,size*line,MPI_INT,0,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        //计算乘积结果，并将结果发送给主进程
        for (i=0;i<line;i++)
        {
            for (j=0;j<size;j++)
            {
                int temp=0;
                for(k=0;k<size;k++)
                    temp += buffer[i*size+k]*b[k*size+j];
                ans[i*size+j]=temp;
            }
        }
        //将计算结果传送给主进程
        MPI_Send(ans,line*size,MPI_INT,0,3,MPI_COMM_WORLD);
```

这段代码是分进程的计算，然后将计算结果返回给主进程。

集合通信

```
    // 使用MPI_Scatter将矩阵'a'分发给各个进程
    MPI_Scatter(a, size * local_size, MPI_INT, local_a, size * local_size,
MPI_INT, 0, MPI_COMM_WORLD);

    // 使用MPI_Bcast广播矩阵'b'给所有进程
    MPI_Bcast(b, size * size, MPI_INT, 0, MPI_COMM_WORLD);

    // 每个进程对其本地数据执行矩阵乘法
    for (i = 0; i < local_size; i++) {
        for (j = 0; j < size; j++) {
            int temp = 0;
            for (k = 0; k < size; k++)
                temp += local_a[i * size + k] * b[k * size + j];
            local_c[i * size + j] = temp;
```

```
        }
    }

    // 使用MPI_Gather将结果收集到主进程
    MPI_Gather(local_c, size * local_size, MPI_INT, c, size * local_size, MPI_INT,
0, MPI_COMM_WORLD);
```

1. `MPI_Scatter`：使用MPI_Scatter函数，将矩阵 `a` 分发给各个进程。这个函数将矩阵 `a` 分割成多个块，每个进程接收其中一块。
2. `MPI_Bcast`：使用MPI_Bcast函数，广播矩阵 `b` 给所有进程。这确保所有进程都有相同的矩阵 `b` 的副本。
3. 循环计算：每个进程对其本地数据执行矩阵乘法。循环计算部分使用本地数据 `local_a` 和全局数据 `b` 来计算本地结果 `local_c`。
4. `MPI_Gather`：最后，使用MPI_Gather函数将各个进程的本地结果 `local_c` 收集到主进程中，形成最终的结果矩阵 `c`。

通过这种方式，矩阵乘法的计算被分布到多个进程中，从而加速了计算过程。MPI函数用于在进程之间传输数据，并最终将结果收集到主进程中，以获得完整的结果。

改造库函数

1. **编写矩阵乘法库函数**：：

```
// matrix_multiply.h

#ifndef MATRIX_MULTIPLY_H
#define MATRIX_MULTIPLY_H

void matrix_multiply(int *A, int *B, int *C, int rowsA, int colsA, int
colsB);

#endif
```

```
// matrix_multiply.c

#include "matrix_multiply.h"

void matrix_multiply(int *A, int *B, int *C, int rowsA, int colsA, int
colsB) {
    // 矩阵乘法的实现
    for (int i = 0; i < rowsA; i++) {
        for (int j = 0; j < colsB; j++) {
            int temp = 0;
            for (int k = 0; k < colsA; k++) {
                temp += A[i * colsA + k] * B[k * colsB + j];
            }
            C[i * colsB + j] = temp;
```

```
        }
    }
}
```

2. **编写编译脚本**：

```
# 编译矩阵乘法库函数为共享库
gcc -shared -o libmatrixmultiply.so matrix_multiply.c
```

运行这个脚本会生成一个名为 `libmatrixmultiply.so` 的共享库文件。

表格

Comm_size (num of processes)	Order of Matrix (milliseconds)				
	128	256	512	1024	2048
1	0.00422	0.0347	0.303	3.12	72.4
2	0.00269	0.0223	0.169	1.85	39.9
4	0.00185	0.0186	0.0994	1.27	30.3
Comm_size (num of processes)	Order of Matrix (Speedups)				
	128	256	512	1024	2048
1	1	1	1	1	1
2	1.568773	1.556054	1.792899	1.686486	1.814536
4	2.281081	1.865591	3.04829	2.456693	2.389439
Comm_size (num of processes)	Order of Matrix (milliseconds)				
	128	256	512	1024	2048
1	0.00437	0.352	0.333	3.73	105
2	0.002738	0.229	0.187	1.88	59.2
4	0.002329	0.192	0.159	1.6	42.8
Comm_size (num of processes)	Order of Matrix (Speedups)				
	128	256	512	1024	2048
1	1	1	1	1	1
2	1.596056	1.537118	1.780749	1.984043	1.773649
4	1.876342	1.833333	2.09434	2.33125	2.453271

实验结果

因为我的电脑只有四个核心，只能进行四个进程的计算，但是不难发现随着进程的增加，运算时间有着明显的减少，证明了并行计算具有加速的效果。

点对点通信比集合通信更快。

另外虽然没有在表格中展示，我发现再增加进程，运算时间还是没有减少，我猜想这可能是因为进程在轮流使用核心的缘故。

实验感想

本次实验使我深刻认识到并行计算的作用。

源码

1. **集合通信**

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <time.h>

int main(int argc, char **argv) {
    double start, stop, max_time;
    int i, j, k;
    int *a, *b, *c, *local_a, *local_c;
    int size = 128;
    int rank, numprocs;
    int local_size;

    MPI_Init(&argc, &argv); // MPI初始化

    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // 获取当前进程的排名
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs); // 获取总进程数

    a = (int *)malloc(sizeof(int) * size * size);
    b = (int *)malloc(sizeof(int) * size * size);
    c = (int *)malloc(sizeof(int) * size * size);

    if (rank == 0) {
        srand((unsigned)time(NULL));
        for (i = 0; i < size; i++)
            for (j = 0; j < size; j++)
                a[i * size + j] = rand() % 5;

        for (i = 0; i < size; i++)
            for (j = 0; j < size; j++)
                b[i * size + j] = rand() % 5;

        start = MPI_Wtime();
    }

    local_size = size / numprocs;
    local_a = (int *)malloc(sizeof(int) * size * local_size);
    local_c = (int *)malloc(sizeof(int) * size * local_size);

    // 使用MPI_Scatter将矩阵'a'分发给各个进程
    MPI_Scatter(a, size * local_size, MPI_INT, local_a, size * local_size,
MPI_INT, 0, MPI_COMM_WORLD);
```

```
    // 使用MPI_Bcast广播矩阵'b'给所有进程
    MPI_Bcast(b, size * size, MPI_INT, 0, MPI_COMM_WORLD);

    // 每个进程对其本地数据执行矩阵乘法
    for (i = 0; i < local_size; i++) {
        for (j = 0; j < size; j++) {
            int temp = 0;
            for (k = 0; k < size; k++)
                temp += local_a[i * size + k] * b[k * size + j];
            local_c[i * size + j] = temp;
        }
    }

    // 使用MPI_Gather将结果收集到主进程
    MPI_Gather(local_c, size * local_size, MPI_INT, c, size * local_size, MPI_INT,
0, MPI_COMM_WORLD);

    if (rank == 0) {
        for (i = local_size * numprocs; i < size; i++) {
            for (j = 0; j < size; j++) {
                int temp = 0;
                for (k = 0; k < size; k++)
                    temp += a[i * size + k] * b[k * size + j];
                c[i * size + j] = temp;
            }
        }

        stop = MPI_Wtime();
        max_time = stop - start;
        printf("最大运行时间: %lfs\n", max_time);

        free(a);
        free(b);
        free(c);
    }

    free(local_a);
    free(local_c);

    MPI_Finalize(); // 结束MPI

    return 0;
}
```

2. **点对点通信**

```
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
```

```
#include<time.h>

int main()
{
    double start, stop;
    int i, j, k, l;
    int *a, *b, *c, *buffer, *ans;
    int size = 2048;
    int rank, numprocs, line;

    MPI_Init(NULL,NULL);//MPI Initialize
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);//获得当前进程号
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);//获得进程个数

    line = size/numprocs;//将数据分为(进程数)个块,主进程也要处理数据
    a = (int*)malloc(sizeof(int)*size*size);
    b = (int*)malloc(sizeof(int)*size*size);
    c = (int*)malloc(sizeof(int)*size*size);
    //缓存大小大于等于要处理的数据大小，大于时只需关注实际数据那部分
    buffer = (int*)malloc(sizeof(int)*size*line);//数据分组大小
    ans = (int*)malloc(sizeof(int)*size*line);//保存数据块计算的结果

    //主进程对矩阵赋初值，并将矩阵N广播到各进程,将矩阵M分组广播到各进程
    if (rank==0)
    {


        srand((unsigned)time(NULL));
        for(i=0;i<size;i++)
            for(j=0;j<size;j++)
                a[i*size+j] = rand()%5;

        for(i=0;i<size;i++)
            for(j=0;j<size;j++)
                b[i*size+j] = rand()%5;
        start = MPI_Wtime();
        //将矩阵N发送给其他从进程
        for (i=1;i<numprocs;i++)
        {
                MPI_Send(b,size*size,MPI_INT,i,0,MPI_COMM_WORLD);
        }
        //依次将a的各行发送给各从进程
        for (l=1; l<numprocs; l++)
        {
            MPI_Send(a+(l-1)*line*size,size*line,MPI_INT,l,1,MPI_COMM_WORLD);
        }
        //接收从进程计算的结果
        for (k=1;k<numprocs;k++)
        {
            MPI_Recv(ans,line*size,MPI_INT,k,3,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
            //将结果传递给数组c
            for (i=0;i<line;i++)
            {
                for (j=0;j<size;j++)
```

```
                {
                    c[((k-1)*line+i)*size+j] = ans[i*size+j];
                }

            }
        }
        //计算a剩下的数据
        for (i=(numprocs-1)*line;i<size;i++)
        {
            for (j=0;j<size;j++)
            {
                int temp=0;
                for (k=0;k<size;k++)
                    temp += a[i*size+k]*b[k*size+j];
                c[i*size+j] = temp;
            }
        }
        //统计时间
        stop = MPI_Wtime();
    // for(i=0;i<size;i++)
    // {
    //     for(j=0;j<size;j++)
    //     {
    //         printf("%d ",a[i*size+j]);
    //     }
    //     printf("\n");
    // }
    // printf("\n");
    // for(i=0;i<size;i++)
    // {
    //     for(j=0;j<size;j++)
    //     {
    //         printf("%d ",b[i*size+j]);
    //     }
    //     printf("\n");
    // }
    // printf("\n");
    // for(i=0;i<size;i++)
    // {
    //     for(j=0;j<size;j++)
    //     {
    //         printf("%d ",c[i*size+j]);
    //     }
    //     printf("\n");
    // }

        printf("rank:%d time:%lfs\n",rank,stop-start);

        free(a);
        free(b);
        free(c);
        free(buffer);
        free(ans);
    }
```

```
    //其他进程接收数据，计算结果后，发送给主进程
    else
    {
        //接收广播的数据(矩阵b)
        MPI_Recv(b,size*size,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

        MPI_Recv(buffer,size*line,MPI_INT,0,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        //计算乘积结果，并将结果发送给主进程
        for (i=0;i<line;i++)
        {
            for (j=0;j<size;j++)
            {
                int temp=0;
                for(k=0;k<size;k++)
                    temp += buffer[i*size+k]*b[k*size+j];
                ans[i*size+j]=temp;
            }
        }
        //将计算结果传送给主进程
        MPI_Send(ans,line*size,MPI_INT,0,3,MPI_COMM_WORLD);
    }

    MPI_Finalize();//结束

    return 0;
}
```