并行程序设计与算法实验1

实验	MPI点对点通信矩阵乘法	专业	计算机科学与技术
学号	21311525	姓名	马梓培
Email	mazp@mail2.sysu.edu.cn	完成日期	2024/03/28

1. 实验目的

- 使用MPI点对点通信实现并行矩阵乘法
- 设置线程数量（1-16）及矩阵规模（128-2048）
- 根据运行时间，分析程序并行性能

2. 配置安装MPI

1. 添加MPICH的官方仓库：

```
sudo sh -c 'echo "deb http://www.mpich.org/mpich-3.4/ubuntu/ $(lsb_release -cs) main"
> /etc/apt/sources.list.d/mpich3.4.list'
```

2. 导入公钥：

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 0xd9345623
```

3. 更新包管理器的包列表：

```
sudo apt update
```

4. 安装MPICH：

```
sudo apt install mpich
```

5. 验证安装

```
mpichversion
```

3. MPI点对点通信并行矩阵乘法实现

代码见附件或github.

3.1 MPI通信初始化

首先，引入头文件

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

初始化MPI参数：

```
int comm_sz, my_rank;
MPI_Comm comm = MPI_COMM_WORLD;// MPI通信子
MPI_Init(NULL, NULL);          // 初始化MPI
MPI_Comm_size(comm, &comm_sz); // 获取通信子大小（进程数量）
MPI_Comm_rank(comm, &my_rank); // 获取进程编号
```

3.2 点对点通信

在将矩阵A和矩阵B随机初始化后，我们的目标是要计算$C = AB$，因此，将矩阵A均匀地划分，并分发给当前通信子中所有的进程，同时，将整个矩阵B发送给所有的进程。也就是说，若要计算出矩阵C的某一行，相当于是找到矩阵A中对应那行与整个矩阵B相乘。

注意到实验数据的简单性（均为2的指数），我们可以朴素的将矩阵行和列的大小，N，根据processor的数量划分，即：

$$row_{avg} = \frac{N}{p} \tag{1}$$

3.2.1 MPI_Send

核0(master core)分发矩阵至[1 to comm_sz - 1]核：

```
avg_rows = N / comm_sz;
...
// master core:
for (int i = 1; i < comm_sz; i++) {
    // send avg_rows of matA to [1 to comm_sz - 1] cores respectively, tag = 0
    MPI_Send(&A[i * avg_rows * N], avg_rows * N, MPI_FLOAT, i, 0, MPI_COMM_WORLD);
    // send whole matB to [1 to comm_sz] cores respectively, tag = 1
    MPI_Send(B, N * N, MPI_FLOAT, i, 1, MPI_COMM_WORLD);
}
```

根据MPI_Send函数，进行参数的解释

```
int MPI_Send(
    void*         msg_buf_p            /* in */,
    int           msg_size             /* in */,
    MPI_Datatype  msg_type             /* in */,
    int           dest                 /* in */,
    int           tag                  /* in */,
    MPI_Comm      communicaotor        /* in */);
```

e.g., `MPI_Send(&A[i * avg_rows * N], avg_rows * N, MPI_FLOAT, i, 0, MPI_COMM_WORLD);`

将矩阵A第i行第0个元素的地址作为msg_buf_p，大小为avg_rows * N，数据类型为MPI_FLOAT，目标processor为i，tag=0，通信子为MPI_COMM_WORLD。

在发送与接收的实现中，我用tag=0和1来区分矩阵A和矩阵B。

3.2.2 MPI_Recv

[1 to comm_sz -1]核接受核0(master core)数据：

```
// recive part of A
MPI_Recv(localA, avg_rows * N, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
// receive whole B
MPI_Recv(localB, N * N, MPI_FLOAT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

同样，根据MPI_Recv函数，进行参数的解释

```
int MPI_Recv(
    void*         msg_buf_p            /* out */,
    int           buf_size             /* in  */,
    MPI_Datatype  buf_type             /* in  */,
    int           source               /* in  */,
    int           tag                  /* in  */,
    MPI_Comm      communicator         /* in  */,
    MPI_Status*   status_p             /* out */);
```

e.g., `MPI_Recv(localA, avg_rows * N, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);`

localA指向接收buf的内存地址，avg_rows * N表示buf大小，数据类型为MPI_FLOAT，源processor为i，tag=0，通信子为MPI_COMM_WORLD，最后的MPI_Status设置为MPI_STATUS_IGNORE，我们并不关心。

3.3 部分A矩阵与完整B矩阵相乘

```
void matrix_multiply(float* A, float* B, float* C, int size, int avg_rows) {
    for (int i = 0; i < avg_rows; ++i) {
        for (int j = 0; j < size; ++j) {
            float sum = 0;
            for (int x = 0; x < size; ++x) {
                sum += *(A + i * size + x) * *(B + x * size + j);
            }
            *(C + i * size + j) = sum;
        }
    }
}
```

采用最朴素的方式，针对于A的部分行与完整矩阵B相乘。不过多赘述，已在作业0实现了。

3.4 矩阵相乘计时

并行时间取决于“最慢”进程话费的时间，换句话说，我们关心从开始到最后一个进程完成的时间开销。

我们可以通过MPI的集合通信函数MPI_Barrier，确保同一个通信子中的所有进程都完成调用该函数之前，没有进程可以提前返回。其语法：

```
int MPI_Barrier(MPI_Comm  comm    /* in */)
```

计时实现：

```
double local_start, local_end, local_elapsed;
MPI_Barrier(MPI_COMM_WORLD);
local_start = MPI_Wtime();
matrix_multiply(localA, localB, localC, N, avg_rows);
local_end = MPI_Wtime();
local_elapsed = local_end - local_start;

MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
```

注意，最后我偷了个懒，直接调用了MPI_Reduce集合通信函数并用操作符MPI_MAX得到所有进程运行的最长运行时间。

3.5 运算结果发回Core 0

与2.2部分一样，在各个进程都得到运算结果后，发回master core，不过多赘述。

Send:

```
MPI_Send(localC, avg_rows * N, MPI_FLOAT, 0, 2, MPI_COMM_WORLD);
```

Receive:

```
for (int i = 1; i < comm_sz; i++) {
    MPI_Recv(&C[i * avg_rows * N], avg_rows * N, MPI_FLOAT, i, 2, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
}
```

4. 实验结果

4.1 验证正确性：

我采用2个2*2大小的矩阵相乘，同时设置线程为2，进行验证。

```
PP1 ) mpiexec -n 2 ./mpimm.out
Matrix A:
6.258609 1.616280
8.761180 6.367680
Matrix B:
8.216056 0.600904
8.762606 7.654576
Matrix C:
65.583908 16.132759
127.779816 54.006512
Cost time: 0.000000
```

经手动计算验证，C确实是等于AB的。虽然不够严谨，但不失一般性，我们的并行运算是正确的。

4.2 完成表格

根据表格，线程数由1至16，矩阵规模由128至2048进行实验。

N = 128：

```
Parallel-Programming ) cd PP1/
PP1 ) mpiexec -n 1 ./mpimm.out
Cost time: 0.005887
PP1 ) mpiexec -n 2 ./mpimm.out
Cost time: 0.002998
PP1 ) mpiexec -n 4 ./mpimm.out
Cost time: 0.001508
PP1 ) mpiexec -n 8 ./mpimm.out
Cost time: 0.001147
PP1 ) mpiexec -n 16 ./mpimm.out
Cost time: 0.000482
```

N = 256：

```
PP1 ) mpiexec -n 1 ./mpimm.out
Cost time: 0.048902
PP1 ) mpiexec -n 2 ./mpimm.out
Cost time: 0.024930
PP1 ) mpiexec -n 4 ./mpimm.out
Cost time: 0.012310
PP1 ) mpiexec -n 8 ./mpimm.out
Cost time: 0.007849
PP1 ) mpiexec -n 16 ./mpimm.out
Cost time: 0.008462
```

N = 512:

```
PP1 ) mpiexec -n 1 ./mpimm.out
Cost time: 0.398231
PP1 ) mpiexec -n 2 ./mpimm.out
Cost time: 0.201551
PP1 ) mpiexec -n 4 ./mpimm.out
Cost time: 0.114946
PP1 ) mpiexec -n 8 ./mpimm.out
Cost time: 0.073404
PP1 ) mpiexec -n 16 ./mpimm.out
Cost time: 0.080081
```

N = 1024:

```
PP1 ) mpiexec -n 1 ./mpimm.out
Cost time: 4.502774
PP1 ) mpiexec -n 2 ./mpimm.out
Cost time: 2.131441
PP1 ) mpiexec -n 4 ./mpimm.out
Cost time: 1.166734
PP1 ) mpiexec -n 8 ./mpimm.out
Cost time: 0.643795
PP1 ) mpiexec -n 16 ./mpimm.out
Cost time: 0.696056
```

N = 2048:

```
PP1 ) mpiexec -n 1 ./mpimm.out
Cost time: 42.735498
PP1 ) mpiexec -n 2 ./mpimm.out
Cost time: 21.767297
PP1 ) mpiexec -n 4 ./mpimm.out
Cost time: 11.761027
PP1 ) mpiexec -n 8 ./mpimm.out
Cost time: 6.799767
PP1 ) mpiexec -n 16 ./mpimm.out
Cost time: 6.566152
```

最终，我们可以得到如下的表格:

P\N	128	256	512	1024	2048
1	0.005887	0.048902	0.398231	4.502774	42.735498
2	0.002998	0.024930	0.201551	2.131441	21.767297
4	0.001508	0.012310	0.114946	1.166734	11.761027
8	0.001147	0.007849	0.073404	0.643795	6.799767
16	0.000482	0.008462	0.080081	0.696056	6.566152

- 横向来看，对于给定的线程数，随着矩阵规模的增加，执行时间增加。这是因为工作量随着矩阵规模的增长呈指数增加。
- 纵向来看，对于给定的矩阵规模，增加线程数通常会减少执行时间，这体现了并行计算的优势。
- 而由于我的虚拟机只分配了8核，因此，使用超过8个线程可能不会给性能带来太多改善，甚至可能会因为线程上下文切换的开销而降低性能。
- 对于较小的矩阵（例如128x128），线程数的增加对执行时间的改善不大。但对于较大的矩阵（2048x2048），增加线程数可以显著降低执行时间，直到达到物理核心的数量。

5. 实验感想

在完成并行程序设计与算法实验后，我深有感触。这次实验通过MPI点对点通信实现并行矩阵乘法，不仅加深了我对并行计算理论的理解，还锻炼了我动手实践的能力。

实验过程中，我首先按照指导书配置并安装了MPI环境，这个过程比我预想的简单，但也让我意识到在并行计算实践中，环境的配置是基础也是关键。随后，我按照步骤一步步实现了点对点通信的并行矩阵乘法。在这一过程中，我不仅要理解MPI通信的机制，还要深入理解并行计算的概念，如进程的分配和同步，数据的分割和汇总等。特别是在实现矩阵乘法的分块、分发、计算和结果汇总过程中，我遇到了不少挑战。但通过查阅资料和不断尝试，我最终解决了这些问题。

实验结果部分让我颇为兴奋。我通过设置不同的线程数量和矩阵规模，观察并分析了程序的并行性能。结果显示，随着线程数量的增加，执行时间有显著下降，特别是在矩阵规模较大时，这种效果更为明显。这不仅验证了并行计算在处理大规模数据时的高效性，也让我体会到了优化算法和提升计算性能的成就感。

总结一下，这次实验让我意识到了并行计算在实际应用中的重要性和潜力。在数据量越来越大的今天，串行计算已经难以满足需求，而并行计算能够有效地解决这一问题。我也意识到，作为一名计算机科学与技术专业的学生，我应该更加深入地学习和掌握并行计算的知识，为将来解决更加复杂的问题做好准备。