**深 圳 大 学 实 验 报 告**

|  |  |
| --- | --- |
| **课程名称** | **：并行计算** |
| **实验项目名称** | **：矩阵乘法的MPI并行程序** |
| **学院** | **：计算机与软件学院** |
| **专业** | **：计算机科学与技术** |
| **指导教师** | **：陆克中** |
| **报告人** | **：何泽锋** |
| **学号** | **：2022150221** |
| **实验时间** | **：2025年05月14日** |
| **实验报告提交时间** | **：2025年05月15日** |

**教务部制**

## 一、实验目的

1. 学会编写Cannon矩阵乘法的MPI并行程序；

2. 掌握MPI\_Isend、MPI\_Irecv、MPI\_Wait、MPI\_Sendrecv、MPI\_Reduce等MPI语句；

3. 通过非阻塞通信实现计算和通信的同时处理；

## 二、实验环境

1. 硬件环境：6×32核CPU、400GB内存的分布内存并行计算平台；

2. 软件环境：Ubuntu Linux、mpicc、mpic++（mpic++ -O3 -o a.out a.cpp）、mpiexec（mpiexec -n 36 -f machinefile /home/bxjs/a.out）

3. 远程登录：本地PowerShell中执行ssh bxjs@hpc.szu.edu.cn；

4. 传输文件：本地PowerShell中执行scp c:\a.cpp [bxjs@hpc.szu.edu.cn:/home/bxjs/](mailto:bxjs@hpc.szu.edu.cn:/home/bxjs/)或<ftp://hpc.szu.edu.cn>。

## 三、实验内容

1. 用MPI编写两个*n*阶方阵*A*和*B*的Cannon矩阵乘法程序，结果存放在方阵*C*中。为了简单起见，假设进程数*p*为完全平方数，且*n*可以被整除。*a*、*b*、*c*都采用块棋盘划分存储，每个子块的大小都为。*A*和*B*的初始值为：*Ai*,*j*=*Bi*,*j*=*in*+*j*，0≤*i*, *j*<*n*。在计算完*C*=*AB*后，求*C*中所有元素的和sum，即sum=，然后进程0在终端输出sum值。

2. 可以采用以下优化方法：①利用缓存命中率和程序的局部性原理来优化矩阵乘法；②通过非阻塞通信实现计算和通信的同时处理。

3. 测试并行程序在不同进程数下的执行时间和加速比（与进程数=1时的执行时间相比）。其中，*n*固定为1680，进程数分别取1、4、9、16、25、36、49、64。为减少误差，每项实验进行5次，取平均值作为实验结果。

## 四、代码描述

*在下面写出完整的程序代码（文本格式，不可截图），对于其中的关键代码，以注释方式给出必要的描述。*

|  |
| --- |
| #include "mpi.h"  #include <iostream>  #include <vector>  #include <algorithm>  #include <cstring>  #include <math.h>  using namespace std;  void Cannon(int n, int p, int rank, int\* a, int\* b, int\* c,  int\* a\_buf, int\* b\_buf, long long\* sum) {  // 初始化网格参数  int q = int(sqrt(double(p))); // 计算进程块的边长  int m = n / q; // 子块的大小  int row\_num = rank / q; // 当前进程块的行号  int col\_num = rank % q; // 当前进程块的列号  // 复制原始数据到缓冲区  memcpy(a\_buf, a, m \* m \* sizeof(int));  memcpy(b\_buf, b, m \* m \* sizeof(int));  // 对齐A矩阵块：每行向左移动行号次  int a\_dest = row\_num \* q + (col\_num - row\_num + q) % q;  int a\_src = row\_num \* q + (col\_num + row\_num) % q;  MPI\_Sendrecv(a\_buf, m \* m, MPI\_INT, a\_dest, 0,  a, m \* m, MPI\_INT, a\_src, 0,  MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  // 对齐B矩阵块：每列向上移动列号次  int b\_dest = (row\_num - col\_num + q) % q \* q + col\_num;  int b\_src = (row\_num + col\_num) % q \* q + col\_num;  MPI\_Sendrecv(b\_buf, m \* m, MPI\_INT, b\_dest, 0,  b, m \* m, MPI\_INT, b\_src, 0,  MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  // 循环计算矩阵乘法  for (int l = 0; l < q; l++) {  // 矩阵乘法计算  for (int i = 0; i < m; i++) {  for (int k = 0; k < m; k++) {  int a\_val = a[i \* m + k]; // 优化后的循环，将a[i\*m+k]缓存到寄存器  for (int j = 0; j < m; j++) {  c[i \* m + j] += a\_val \* b[k \* m + j];  }  }  }  // 数据移位阶段  if (l < q - 1) {  // A矩阵块向左循环移位  a\_dest = row\_num \* q + (col\_num - 1 + q) % q;  a\_src = row\_num \* q + (col\_num + 1) % q;  MPI\_Sendrecv(a, m \* m, MPI\_INT, a\_dest, 0,  a\_buf, m \* m, MPI\_INT, a\_src, 0,  MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  // B矩阵块向上循环移位  b\_dest = (row\_num - 1 + q) % q \* q + col\_num;  b\_src = (row\_num + 1) % q \* q + col\_num;  MPI\_Sendrecv(b, m \* m, MPI\_INT, b\_dest, 0,  b\_buf, m \* m, MPI\_INT, b\_src, 0,  MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  // 指针交换  swap(a, a\_buf);  swap(b, b\_buf);  }  }  // 计算进程块的和  long long local\_sum = 0;  for (int i = 0; i < m \* m; i++) {  local\_sum += c[i];  }  // 结果归约到0号进程  MPI\_Reduce(&local\_sum, sum, 1, MPI\_LONG\_LONG, MPI\_SUM, 0, MPI\_COMM\_WORLD);  }  int main(int argc, char\*\* argv) {  int n = 1680; //矩阵大小为n\*n  int m; //子块大小为m\*m  long long sum = 0; // 用于存储最终结果  int rank, p, q;  MPI\_Init(&argc, &argv);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &p);  q = int(sqrt(double(p))); // 进程网格边长  m = n / q; // 子块大小  int\* a = new int[m \* m]; // 矩阵A  int\* b = new int[m \* m]; // 矩阵B  int\* c = new int[m \* m]; // 结果矩阵C  int\* a\_buf = new int[m \* m]; // A的通信缓冲区  int\* b\_buf = new int[m \* m]; // B的通信缓冲区  // 初始化数据  for (int i = 0; i < m; i++) {  for (int j = 0; j < m; j++) {  a[i \* m + j] = b[i \* m + j] = (rank / q \* m + i) \* n + (rank % q \* m + j);  c[i \* m + j] = 0;  }  }  // 执行Cannon  double t0 = MPI\_Wtime();  Cannon(n, p, rank, a, b, c, a\_buf, b\_buf, &sum);  double t1 = MPI\_Wtime();  // 输出结果  if (rank == 0) {  cout << "sum is " << sum << endl;  cout << "time is " << t1 - t0 << " seconds" << endl;  }  delete[] a;  delete[] b;  delete[] c;  delete[] a\_buf;  delete[] b\_buf;  MPI\_Finalize();  return 0;  } |

## 五、实验结果和分析

*实验结果以及对实验结果的比较分析和综合概括。*

表1 并行程序在不同进程数下的执行时间（秒）和加速比（*n*=1680）

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 线程数  执行时间 | 1 | 4 | 9 | 16 | 25 | 36 | 49 | 64 |
| 第1次 | 2.70146 | 0.83777 | 0.43981 | 0.33697 | 0.32336 | 0.20243 | 0.25423 | 0.34851 |
| 第2次 | 2.62562 | 0.81428 | 0.43272 | 0.34379 | 0.33305 | 0.20297 | 0.25330 | 0.35733 |
| 第3次 | 2.60219 | 0.80987 | 0.43242 | 0.34754 | 0.32621 | 0.20930 | 0.24165 | 0.34179 |
| 第4次 | 2.60537 | 0.82589 | 0.42961 | 0.33088 | 0.31641 | 0.21007 | 0.24723 | 0.34246 |
| 第5次 | 2.64985 | 0.80414 | 0.41676 | 0.32557 | 0.33745 | 0.20128 | 0.24814 | 0.34473 |
| 平均值 | 2.63690 | 0.81839 | 0.43026 | 0.33695 | 0.32729 | 0.20521 | 0.24891 | 0.34696 |
| 加速比 | 1.00000 | 3.22206 | 6.12858 | 7.82581 | 8.05664 | 12.84974 | 10.59390 | 7.59995 |

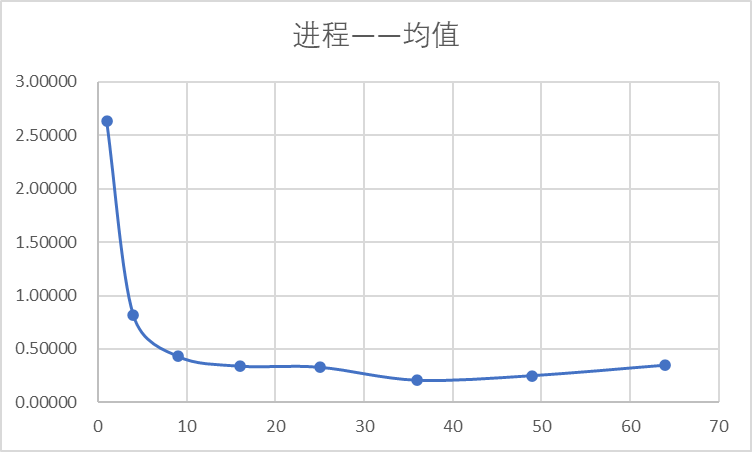
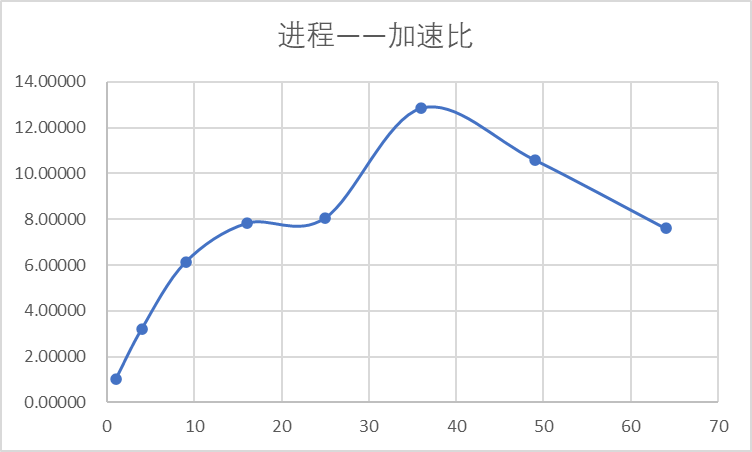
 

图1 进程——均值曲线 图2 进程——加速比曲线

实验通过测试不同进程数下的Cannon矩阵乘法，从数据可以看出，随着进程数的增加，程序的执行时间呈现明显的下降趋势，加速比逐步提升，但在进程数达到一定规模后，性能提升的幅度逐渐减缓，甚至出现下降的情况。具体来看，当进程数为1时，平均执行时间为2.63690秒，将这个时间作为计算的基准。当进程数增加到4时，执行时间降至0.81839秒，加速比达到3.2206，说明并行化带来了显著的性能提升，但此时的加速比不能完全与进程数量一致，这是因为程序还有其他开销，包括通信等时间。进程数进一步增加，此时的时间减少和加速比上升没有刚开始的时候的幅度大，逐渐趋于平缓。

当进程数增加到25时，执行时间的下降趋势开始放缓，相交于16和36进程的加速比提升不明显，没有很好的符合曲线的趋势。当进程数量超过36时时间反而上升了，而加速比开始下降这一现象可能源于通信开销的增加，因为更多的进程意味着更多的数据交换和同步操作，在Cannon算法中，矩阵块需要频繁移位，通信成本逐渐成为性能瓶颈。在当前的实验环境下，36个进程可能是较优的并行规模，超过这一规模后，通信开销问题开始影响整体性能。同时还要考虑到进程间的计算速度差议，最慢的进程会导致整体速度下滑，当进程数过多的时候可能会面临比较严重的同步问题，进程会相互阻塞导致运行速度下降更加严重。

## 六、实验结论

*实验过程中遇到的问题及解决办法，运用了哪些技术方法以提高实验性能，从该实验得到的客观结论，等等。*

在本次实验中，通过MPI实现了Cannon矩阵乘法，验证了其在分布式内存系统中的正确使用。实验过程中，最初的数据对齐和通信逻辑较为复杂，通过调试MPI\_Sendrecv的参数和优化数据交换策略，确保了算法的正确执行。与单独使用MPI\_Send和MPI\_Recv相比，MPI\_Sendrecv能有效避免死锁问题，因为它确保发送和接收的操作是同步的。

Cannon算法的核心在于通过矩阵块的循环移位实现分布式矩阵乘法。每次移位后，各进程执行局部矩阵乘法运算。当进程数增加时，每个进程处理的数据块尺寸呈平方反比减小，计算复杂度从降至。当p较小时，计算量减少带来的收益主导；但当p超过临界点后，通信开销开始主导执行时间。在本实验中，36个进程表现最佳，超过这一规模后，通信开销的增加会导致性能下降。在实际应用中，需要根据具体问题规模和硬件环境选择合适的并行度，以避免资源浪费和性能损失。