**深 圳 大 学 实 验 报 告**

|  |  |
| --- | --- |
| **课程名称** | **：并行计算** |
| **实验项目名称** | **：高斯-约旦消去法的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28日** |
| **实验报告提交时间** | **：2025年05月30日** |

**教务部制**

## 一、实验目的

1. 学会编写高斯-约旦消去法的MPI并行程序；

2. 掌握MPI\_Allgather、MPI\_Bcast、MPI\_Reduce等MPI语句；

3. 对并行程序进行性能分析。

## 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编写编写选主元的高斯-约旦消去法的MPI并行程序，求解*n*元线性方程组***Ax***=***b*。**为了验证结果正确性，将求解得到的***x***带入原方程组，求出最大误差，即向量***Ax***-***b***中绝对值最大的分量。

2. 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1000，进程数分别取1、2、4、8、16、32、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 <cmath>  #include <algorithm>  using namespace std;  inline void parallel(double(\*a)[1001], int rows, int cols, int rank, int procNum,  int\* pivot, double\* x, int\* which, double\* picked\_row) {  // 遍历每一列，逐步进行主元选择和消元  for (int step = 0; step < cols; step++) {  // 当前进程在本地寻找第step列的最大绝对值元素，作为主元候选  double localMaxVal = -1.0;  int localMaxRow = -1;  for (int i = 0; i < rows; i++) {  if (pivot[i] == -1 && fabs(a[i][step]) > localMaxVal) {  localMaxVal = fabs(a[i][step]);  localMaxRow = i;  }  }  // 结构体，用于同时传递主元值和进程编号  struct {  double value;  int rank;  } in, out;  in.value = localMaxVal;  in.rank = rank;  // 所有进程进行归约操作，找出具有最大绝对值主元的进程  MPI\_Allreduce(&in, &out, 1, MPI\_DOUBLE\_INT, MPI\_MAXLOC, MPI\_COMM\_WORLD);  int globalPivotProc = out.rank; // 全局主元的进程编号  which[step] = globalPivotProc; // 第step列主元来自哪个进程  // 只有拥有主元的进程负责提取对应行并将其广播  if (rank == globalPivotProc) {  pivot[localMaxRow] = step; // 标记该行为第step步的主元行  for (int j = 0; j <= cols; j++) picked\_row[j] = a[localMaxRow][j]; // 复制主元行  }  // 广播主元行的第step列及之后的数据给所有进程  MPI\_Bcast(&picked\_row[step], cols + 1 - step, MPI\_DOUBLE, globalPivotProc, MPI\_COMM\_WORLD);  // 对每一行进行消元（除了主元行），将第step列的值归零  for (int i = 0; i < rows; i++) {  if (pivot[i] != step && picked\_row[step] != 0) {  double factor = a[i][step] / picked\_row[step]; // 计算因子  a[i][step] = 0; // 第step列置零  for (int j = step + 1; j <= cols; j++) {  a[i][j] -= factor \* picked\_row[j]; // 执行消元  }  }  }  }  // 每个进程根据已知的主元位置求解对应的解  for (int i = 0; i < rows; i++) {  if (pivot[i] != -1) {  x[pivot[i]] = a[i][cols] / a[i][pivot[i]]; // x\_i = b / A[i][i]  }  }  // 将每个解广播到所有进程  for (int i = 0; i < cols; i++) {  MPI\_Bcast(&x[i], 1, MPI\_DOUBLE, which[i], MPI\_COMM\_WORLD);  }  }  int main(int argc, char\* argv[])  {  int rank, p;  const int n = 1000;  MPI\_Init(&argc, &argv);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &p);  int m = rank < n % p ? n / p + 1 : n / p; // 该进程处理的矩阵行数  double(\*a)[n + 1] = new double[m][n + 1]; // b[i]存储在a[i][n]中  double(\*z)[n + 1] = new double[m][n + 1]; // a的备份，用于计算最大误差  int\* pivot = new int[m];  double\* max\_coefficient = new double[p];  double x[n];  int which[n];  double picked\_row[n + 1];  for (int i = 0; i < m; i++)  pivot[i] = -1;  srand(rank \* 100);  for (int i = 0; i < m; i++)  for (int j = 0; j < n + 1; j++)  a[i][j] = z[i][j] = rand() \* 1.0 / RAND\_MAX;  double t0 = MPI\_Wtime();  parallel(a, m, n, rank, p, pivot, x, which, picked\_row);  double t1 = MPI\_Wtime();  double max\_dif;  double local\_dif = 0;  for (int i = 0; i < m; i++)  {  double dif = z[i][n];  for (int j = 0; j < n; j++)  dif -= z[i][j] \* x[j];  local\_dif = max(local\_dif, abs(dif));  }  MPI\_Reduce(&local\_dif, &max\_dif, 1, MPI\_DOUBLE, MPI\_MAX, 0, MPI\_COMM\_WORLD);  if (rank == 0)  {  cout << "max difference is " << max\_dif << endl;  cout << "time is " << t1 - t0 << " seconds" << endl;  }  delete[] a;  delete[] z;  delete[] pivot;  delete[] max\_coefficient;  MPI\_Finalize();  return 0;  } |

程序将系数矩阵按行分块到多个进程，采用部分主元策略进行消元。每个进程维护本地矩阵块，在每一步消元中，各进程先在本地未选行中找出当前列绝对值最大的元素，通过归约操作确定全局主元所在进程及行号。主元进程广播主元行，其他进程用其进行消元。消元完成后，各进程计算本地解分量并广播，最终所有进程获得完整解向量。

## 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*=1000）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 线程数  执行时间 | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| 第1次 | 0.40092 | 0.56988 | 0.92287 | 1.28379 | 1.32571 | 1.29839 | 1.15515 |
| 第2次 | 0.40254 | 0.55909 | 0.92913 | 1.31481 | 1.34770 | 1.23001 | 1.14440 |
| 第3次 | 0.40078 | 0.55081 | 0.96802 | 1.26065 | 1.30454 | 1.21551 | 1.17914 |
| 第4次 | 0.39791 | 0.58751 | 0.91919 | 1.30477 | 1.30663 | 1.16455 | 1.10567 |
| 第5次 | 0.39731 | 0.59404 | 0.97021 | 1.25877 | 1.30944 | 1.22559 | 1.13167 |
| 平均值 | 0.39989 | 0.57227 | 0.94188 | 1.28456 | 1.31880 | 1.22681 | 1.14321 |
| 加速比 | 1.00000 | 0.69878 | 0.42457 | 0.31131 | 0.30322 | 0.32596 | 0.34980 |

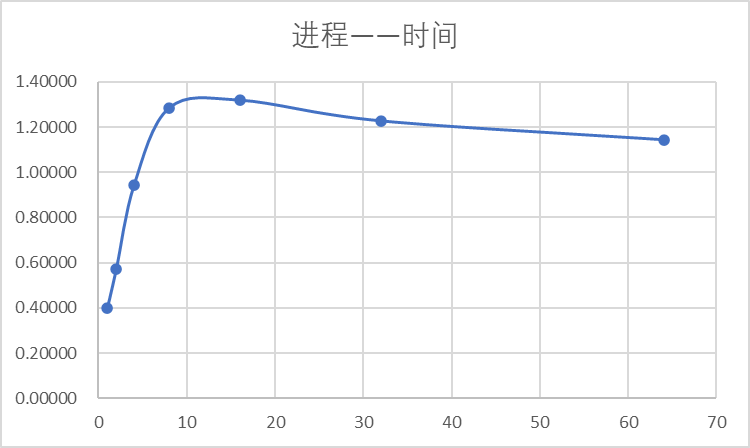
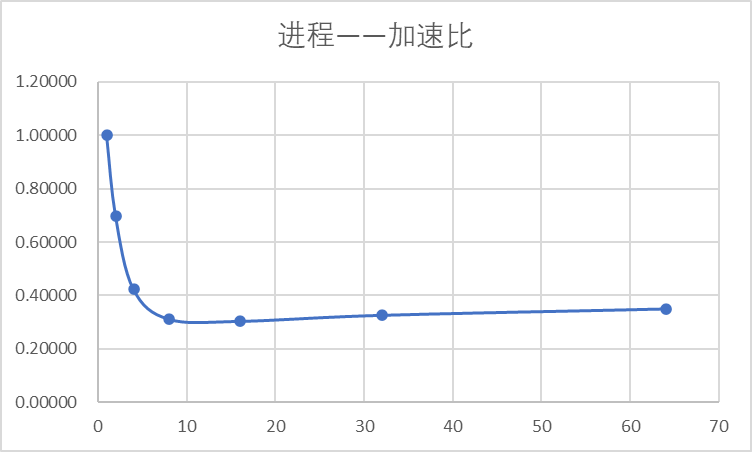
 

图1 进程——时间比 图2 进程——加速比

（1）分析实验数据

当进程数从1增至16时，平均执行时间从0.39989秒逐步上升至1.31880 秒，加速比从1降至0.30322，随着进程增多，耗时反而增长。进程数继续增至32、64时，执行时间略降至1.22681秒、1.14321秒，加速比微升至0.32596、0.34980，但仍远低于理论加速效果。

结合实验6的分析可以推断出这是因为频繁的进程通信造成的，具体来说：每处理一列需执行1次MPI\_Allreduce用于全局选主元、2次MPI\_Bcast分别广播主元行号与数据，而 n=1000时需循环1000次此类操作。当进程数增加，单个进程分配的行数呈线性减少，计算量随之减少，但通信次数与数据传输量未同比下降，此时计算时间的缩短远不如通信带来的延迟。

进程数为1时，程序无需任何通信，纯计算效率最高；进程数增至2时，虽然计算被拆分，但每次消元需2个进程同步主元信息，通信延迟使加速比下降。当进程数达16，进程间的广播与归约的往返延迟可能超过本地计算时间，导致加速比进一步降低。进程数从32增至64时执行时间略降，可能是因为系统调度上的优化。

表2 并行程序在不同进程数下的数据差值

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 线程数  数值差距 | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| 平均值 | 7.66E-13 | 7.63E-13 | 3.95E-13 | 5.91E-13 | 4.81E-13 | 1.19E-12 | 1.59E-10 |

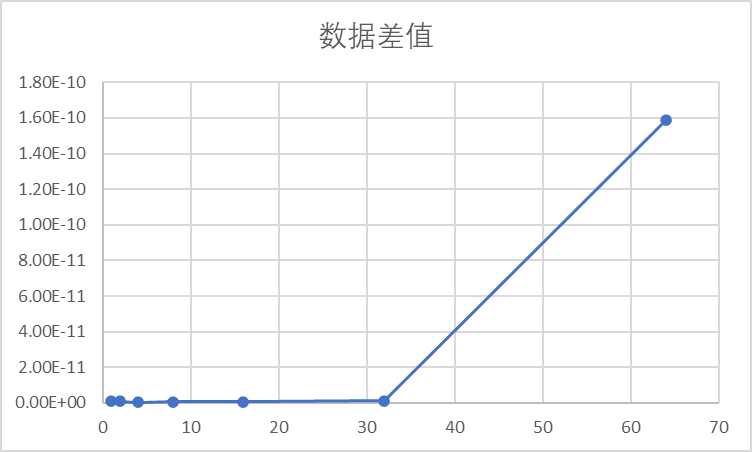


图3 数据差值与进程数量的关系

从实验数据看，线程数为1-32时，数值差距稳定在10−13-10−12数量级，在合理分块下能有效控制浮点运算的舍入误差，且通信开销未显著影响数值稳定性，此时每个线程处理行数较多，主元选择覆盖范围广，消元过程的数值稳定性通过全局选主元机制得以保障。而当线程数增至64时，数值差距骤升至1.59E-10，出现了明显的误差增加。

（2）代码优化部分

本次代码进行了两处的细节优化：

①采用MPI\_Allreduce来收集全局主元，相较于直接使用MPI\_Gather来获取所有的值，MPI\_Allreduce采用树状归约的方式，可以将获取全局主元的时间复杂度降低到，而MPI\_Gather需要手动比较获取的所有的局部主元，并将比较得到的最大值广播，代码上也相对繁琐。

②传递主元行的时候只传cols + 1 – step的部分，此处的cols是行的总长度，step表示这是当前处理的列，采用当前方法计算的情况下，step前的列都已经被处理，因此无需进行广播，可以减少数据传播量，减少广播时间。

（3）时间复杂度分析：

①本地主元查找：每步处理第 step 列，需要在 行中查找最大值，单轮，总体为

②全局主元归约：使用MPI\_Allreduce，执行时间

③广播主元行：每次广播的数据量为 ，平均每轮 ，每轮通信代价，总复杂度为

④本地消元：每一列，对本地 行进行一次消元，消元是，总体为

⑤获得方程解并广播；每个变量x需要一次广播，共 n 次，广播一次的代价是，总复杂度

总复杂度为：

## 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。*

本实验通过实现基于MPI的高斯-约旦消去法并行程序，采用多进程求解大型线性方程组。在正确性方面，程序在大多数进程数下均能保持较高精度，误差控制在10⁻¹²数量级以内，表明全局主元选取机制有效保障了数值稳定性。然而在进程数达到64时，误差出现明显升高，分析发现这是由于单个进程处理的行数大幅减少，主元选取范围变窄，加上浮点数精度误差累积，造成数值解不稳定。

在性能方面，实验结果显示程序的并行加速效果并不理想，进程数从1扩展至16时，总体运行时间不降反升，加速比持续下降，进一步扩展至32和64后仅有小幅回升。结合通信分析可知，程序在每列消元过程中需执行一次全局主元归约与一次主元行广播，当列数较大（n=1000）时，频繁通信成为性能瓶颈。

在优化上，程序采用MPI\_Allreduce代替MPI\_Gather，利用树状归约结构降低选主元的通信复杂度，同时在主元行广播中仅传输必要数据，避免冗余信息传播，从而在一定程度上提升了效率。

实验过程分析了每一步的时间复杂度，最终计算得到的大约复杂度为，前者为并行计算部分，后者反映通信开销对性能的主导影响。实际上并行的开销会比估计的要大，进程间的通信和阻塞的时间会放大等待的时间。