《时空大数据高性能处理》上机实验

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 章节 | **第一次上机：矩阵计算** | | | | | 序号 | 1 |
| 姓 名 | 甘劲博 | 系院专业 | 计算机 | 班级 | 21计科5 | 学 号 | 2110551202 |
| 日期 | 2024-9-24 | | 指导教师 | 刘舟 | | 成 绩 |  |
| **一、内容**  回顾矩阵基本运算，重点包括矩阵与向量相乘、矩阵与矩阵相乘、矩阵转置、矩阵求逆，进行编程实现；在此基础上，讨论矩阵操作的性能。 | | | | | | | |
| **2.1主要c/c++代码**  包含：矩阵和向量的相乘、矩阵和矩阵的相乘、矩阵转置、矩阵求逆  开源地址：<https://github.com/Duxingmengshou/HPCLab>  矩阵和向量的相乘：  #include <stdio.h>  #include <mpi.h>  #include <stdlib.h>    #define M 10 // 矩阵行数  #define N 11 // 矩阵列数    void Matrix\_print(double \*A, int m, int n)  {  for (int i = 0; i < m; i++)  {  for (int j = 0; j < n; j++)  printf("%.1f ", A[i \* n + j]);  printf("\n");  }  printf("\n");  }    int main(int argc, char \*\*argv)  {  int my\_rank, comm\_sz, line;  double start, stop; // 计时时间  MPI\_Status status;    double \*Matrix\_A, \*Vector\_B, \*result, \*buffer\_A, \*ans;    MPI\_Init(&argc, &argv);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &comm\_sz);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank);    line = M / comm\_sz; // 每个进程分多少行数据  Matrix\_A = (double \*)malloc(M \* N \* sizeof(double));  Vector\_B = (double \*)malloc(N \* sizeof(double));  result = (double \*)malloc(M \* sizeof(double));  buffer\_A = (double \*)malloc(line \* N \* sizeof(double)); // A的均分行的数据  ans = (double \*)malloc(line \* sizeof(double)); // 保存部分数据计算结果    // 给矩阵A和向量B赋值  if (my\_rank == 0)  {  start = MPI\_Wtime();  for (int i = 0; i < M; i++)  {  for (int j = 0; j < N; j++)  Matrix\_A[i \* N + j] = i + 1; // 示例初始化  }  for (int i = 0; i < N; i++)  {  Vector\_B[i] = i + 1; // 示例初始化  }    // 输出矩阵A和向量B  Matrix\_print(Matrix\_A, M, N);  printf("\n");  for (int i = 0; i < N; i++)  {  printf("%.1f ", Vector\_B[i]);  }  printf("\n");  }    // 数据分发  MPI\_Scatter(Matrix\_A, line \* N, MPI\_DOUBLE, buffer\_A, line \* N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);  // 数据广播  MPI\_Bcast(Vector\_B, N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);    // 计算结果  for (int i = 0; i < line; i++)  {  double temp = 0;  for (int j = 0; j < N; j++)  {  temp += buffer\_A[i \* N + j] \* Vector\_B[j];  }  ans[i] = temp;  }    // 结果聚集  MPI\_Gather(ans, line, MPI\_DOUBLE, result, line, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);    // 计算剩余行数据  if (my\_rank == 0)  {  int rest = M % comm\_sz;  if (rest != 0)  {  for (int i = M - rest; i < M; i++)  {  double temp = 0;  for (int j = 0; j < N; j++)  {  temp += Matrix\_A[i \* N + j] \* Vector\_B[j];  }  result[i] = temp;  }  }    // 输出结果  printf("\n");  for (int i = 0; i < M; i++)  {  printf("%.1f ", result[i]);  }  printf("\n");  stop = MPI\_Wtime();  printf("rank:%d time:%lfs\n", my\_rank, stop - start);  }    free(Matrix\_A);  free(Vector\_B);  free(result);  free(ans);  free(buffer\_A);    MPI\_Finalize();  return 0;  }  矩阵和矩阵的相乘：  #include <stdio.h>  #include <mpi.h>  #include <stdlib.h>    #define M 10 // 矩阵维度  #define N 11  #define K 12    void Matrix\_print(double \*A, int m, int n)  {  for (int i = 0; i < m; i++)  {  for (int j = 0; j < n; j++)  printf("%.1f ", A[i \* n + j]);  printf("\n");  }  printf("\n");  }    int main(int argc, char \*\*argv)  {  int my\_rank, comm\_sz, line;  double start, stop; // 计时时间  MPI\_Status status;    double \*Matrix\_A, \*Matrix\_B, \*Matrix\_C, \*ans, \*buffer\_A, \*buffer\_C, \*result\_Matrix;  double alpha = 2, beta = 2; // 系数C=aA\*B+bC    MPI\_Init(&argc, &argv);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &comm\_sz);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank);    line = M / comm\_sz; // 每个进程分多少行数据  Matrix\_A = (double \*)malloc(M \* N \* sizeof(double));  Matrix\_B = (double \*)malloc(N \* K \* sizeof(double));  Matrix\_C = (double \*)malloc(M \* K \* sizeof(double));  buffer\_A = (double \*)malloc(line \* N \* sizeof(double)); // A的均分行的数据  buffer\_C = (double \*)malloc(line \* K \* sizeof(double)); // C的均分行的数据  ans = (double \*)malloc(line \* K \* sizeof(double)); // 保存部分数据计算结果  result\_Matrix = (double \*)malloc(M \* K \* sizeof(double)); // 保存数据计算结果    // 给矩阵A B,C赋值  if (my\_rank == 0)  {  start = MPI\_Wtime();  for (int i = 0; i < M; i++)  {  for (int j = 0; j < N; j++)  Matrix\_A[i \* N + j] = i + 1;  for (int p = 0; p < K; p++)  Matrix\_C[i \* K + p] = 1;  }  for (int i = 0; i < N; i++)  {  for (int j = 0; j < K; j++)  Matrix\_B[i \* K + j] = j + 1;  }    // 输出A,B,C  Matrix\_print(Matrix\_A, M, N);  Matrix\_print(Matrix\_B, N, K);  }    // 数据分发  MPI\_Scatter(Matrix\_A, line \* N, MPI\_DOUBLE, buffer\_A, line \* N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);  MPI\_Scatter(Matrix\_C, line \* K, MPI\_DOUBLE, buffer\_C, line \* K, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);  // 数据广播  MPI\_Bcast(Matrix\_B, N \* K, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);    // 计算 结果  for (int i = 0; i < line; i++)  {  for (int j = 0; j < K; j++)  {  double temp = 0;  for (int p = 0; p < N; p++)  temp += buffer\_A[i \* N + p] \* Matrix\_B[p \* K + j];  ans[i \* K + j] = alpha \* temp + beta \* buffer\_C[i \* K + j];  }  }  // 结果聚集  MPI\_Gather(ans, line \* K, MPI\_DOUBLE, result\_Matrix, line \* K, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);    // 计算A剩下的行数据  if (my\_rank == 0)  {  int rest = M % comm\_sz;  if (rest != 0)  {  for (int i = M - rest - 1; i < M; i++)  for (int j = 0; j < K; j++)  {  double temp = 0;  for (int p = 0; p < N; p++)  temp += Matrix\_A[i \* N + p] \* Matrix\_B[p \* K + j];  result\_Matrix[i \* K + j] = alpha \* temp + beta \* Matrix\_C[i \* K + j];  }  }    Matrix\_print(result\_Matrix, M, K);  stop = MPI\_Wtime();    printf("rank:%d time:%lfs\n", my\_rank, stop - start);  }    free(Matrix\_A);  free(Matrix\_B);  free(Matrix\_C);  free(ans);  free(buffer\_A);  free(buffer\_C);  free(result\_Matrix);    MPI\_Finalize();  return 0;  }  矩阵转置：  #include <stdio.h>  #include <stdlib.h>  #include <mpi.h>    #define N 4    void printMatrix(int \*matrix, int rows, int cols)  {  for (int i = 0; i < rows; i++)  {  for (int j = 0; j < cols; j++)  {  printf("%d ", matrix[i \* cols + j]);  }  printf("\n");  }  }    int main(int argc, char \*argv[])  {  int rank, size;  MPI\_Init(&argc, &argv);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);    int local\_rows = N / size;  int \*local\_matrix = (int \*)malloc(local\_rows \* N \* sizeof(int));  int \*global\_matrix = NULL;  int \*transposed\_matrix = NULL;    if (rank == 0)  {  global\_matrix = (int \*)malloc(N \* N \* sizeof(int));  for (int i = 0; i < N \* N; i++)  {  global\_matrix[i] = i;  }  printMatrix(global\_matrix, N, N);  }    MPI\_Scatter(global\_matrix, local\_rows \* N, MPI\_INT, local\_matrix, local\_rows \* N, MPI\_INT, 0, MPI\_COMM\_WORLD);    int \*local\_transposed = (int \*)malloc(local\_rows \* N \* sizeof(int));  for (int i = 0; i < local\_rows; i++)  {  for (int j = 0; j < N; j++)  {  local\_transposed[j \* local\_rows + i] = local\_matrix[i \* N + j];  }  }    if (rank == 0)  {  transposed\_matrix = (int \*)malloc(N \* N \* sizeof(int));  }    MPI\_Gather(local\_transposed, local\_rows \* N, MPI\_INT, transposed\_matrix, local\_rows \* N, MPI\_INT, 0, MPI\_COMM\_WORLD);    if (rank == 0)  {  printf("Transposed Matrix:\n");  printMatrix(transposed\_matrix, N, N);  free(global\_matrix);  free(transposed\_matrix);  }    free(local\_matrix);  free(local\_transposed);    MPI\_Finalize();  return 0;  }  矩阵求逆：  #include <stdio.h>  #include <mpi.h>  #include <stdlib.h>    #define N 3 // 矩阵维度    void Matrix\_print(double \*A, int n)  {  for (int i = 0; i < n; i++)  {  for (int j = 0; j < n; j++)  printf("%.2f ", A[i \* n + j]);  printf("\n");  }  printf("\n");  }    void invert\_matrix(double \*A, double \*I, int n)  {  for (int i = 0; i < n; i++)  {  for (int j = 0; j < n; j++)  {  I[i \* n + j] = (i == j) ? 1.0 : 0.0; // 初始化单位矩阵  }  }    for (int i = 0; i < n; i++)  {  double pivot = A[i \* n + i];  for (int j = 0; j < n; j++)  {  A[i \* n + j] /= pivot; // 归一化行  I[i \* n + j] /= pivot; // 同时归一化单位矩阵  }  for (int k = 0; k < n; k++)  {  if (k != i)  {  double factor = A[k \* n + i];  for (int j = 0; j < n; j++)  {  A[k \* n + j] -= factor \* A[i \* n + j]; // 消元  I[k \* n + j] -= factor \* I[i \* n + j]; // 更新单位矩阵  }  }  }  }  }    int main(int argc, char \*\*argv)  {  int my\_rank, comm\_sz;  double \*Matrix\_A, \*Matrix\_I;    MPI\_Init(&argc, &argv);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &comm\_sz);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank);    Matrix\_A = (double \*)malloc(N \* N \* sizeof(double));  Matrix\_I = (double \*)malloc(N \* N \* sizeof(double));    // 给矩阵A赋值  if (my\_rank == 0)  {  Matrix\_A[0] = 4;  Matrix\_A[1] = 7;  Matrix\_A[2] = 2;  Matrix\_A[3] = 3;  Matrix\_A[4] = 6;  Matrix\_A[5] = 1;  Matrix\_A[6] = 2;  Matrix\_A[7] = 5;  Matrix\_A[8] = 3;    // 输出原矩阵  printf("Original Matrix:\n");  Matrix\_print(Matrix\_A, N);  }    // 广播矩阵A  MPI\_Bcast(Matrix\_A, N \* N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);    // 矩阵求逆  invert\_matrix(Matrix\_A, Matrix\_I, N);    // 收集结果  MPI\_Gather(Matrix\_I, N \* N / comm\_sz, MPI\_DOUBLE, Matrix\_I, N \* N / comm\_sz, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);    // 输出逆矩阵  if (my\_rank == 0)  {  printf("Inverse Matrix:\n");  Matrix\_print(Matrix\_I, N);  }    free(Matrix\_A);  free(Matrix\_I);    MPI\_Finalize();  return 0;  }  Makefile：  MPICXX = mpicc  CXXFLAGS = -Wall -O2    all: 1-s 2-s 3-s 4-s    1-s: 1-s.c  $(MPICXX) $(CXXFLAGS) -o 1-s 1-s.c    2-s: 2-s.c  $(MPICXX) $(CXXFLAGS) -o 2-s 2-s.c    3-s: 3-s.c  $(MPICXX) $(CXXFLAGS) -o 3-s 3-s.c    4-s: 4-s.c  $(MPICXX) $(CXXFLAGS) -o 4-s 4-s.c    clean:  rm -f 1-s 2-s 3-s 4-s | | | | | | | |
| **2.2 运行截图**    矩阵和向量的相乘：    矩阵和矩阵相乘：    矩阵转置：    求逆矩阵： | | | | | | | |
| **2.3 结果分析**  在并行计算中，矩阵运算的性能受到多个因素的影响，主要包括计算复杂度、数据传输与通信开销、负载均衡和内存访问模式。首先，计算复杂度是评估算法效率的重要指标。例如，矩阵与向量相乘的时间复杂度为O(m n)，而矩阵与矩阵相乘的时间复杂度为O(mnp)，矩阵求逆的时间复杂度通常为O(n^3)。这些复杂度表明，随着矩阵规模的增大，计算所需的时间和资源也会显著增加。  其次，在分布式计算中，数据传输的开销可能成为性能瓶颈。MPI进程之间的通信延迟和带宽限制会直接影响整体性能，尤其是在矩阵乘法等需要频繁数据访问的操作中，数据在不同进程之间的传输会增加延迟，从而影响计算效率。此外，负载均衡也是一个关键因素，确保每个进程处理的计算量大致相同，以避免某些进程处于空闲状态。不均匀的负载分配可能导致性能下降，尤其是在处理大规模矩阵时，某些进程可能会完成任务而其他进程仍在计算，造成资源浪费。  内存访问模式同样对性能有显著影响。矩阵的存储方式（如行主序或列主序）会影响内存访问的局部性，进而影响缓存性能。优化内存访问模式可以减少缓存未命中，提高性能，从而加速计算过程。  关于 MPI 进程数量与性能的关系，增加进程数并不总是能带来更快的计算速度。虽然更多的进程可以并行处理更多的计算任务，但由于通信开销和负载不均衡，性能提升可能会减小。在某些情况下，增加进程数可能导致性能下降，特别是在进程数超过硬件资源（如 CPU 核心数）时，过多的进程可能会导致资源竞争和上下文切换的开销，从而影响整体性能。  影响矩阵计算的因素还包括矩阵的大小、硬件架构和算法选择。较大的矩阵通常能更好地利用并行性，但也会增加通信开销。不同的硬件架构（如多核 CPU、GPU 和集群）具有不同的性能特征，这会影响并行算法的选择。此外，不同的算法在并行环境下的表现差异较大，例如，分块矩阵乘法在某些情况下比传统算法更有效。  综上所述，在进行矩阵运算时，理解基本运算的实现及其性能特征是至关重要的。通过合理的并行策略、负载均衡和优化内存访问模式，可以显著提高计算性能。然而，过多的 MPI 进程并不总是能带来更快的计算速度，性能的提升依赖于具体的应用场景和硬件环境。在实际应用中，需要根据矩阵的特性和计算需求选择合适的算法和并行策略，以达到最佳性能。 | | | | | | | |