中山大學
SUN YAT-SEN UNIVERSITY

并行程序设计与算法实验

Lab1-基于 MPI 的并行矩阵乘法

姓 名 李源卿

学 号 22336128

学 院 计算机学院

专 业 计算机科学与技术

2025 年 3 月 26 日

1 实验目的

- 掌握 MPI 程序的编译和运行方法。
- 理解 MPI 点对点通信的基本原理。
- 了解 MPI 程序的 GDB 调试流程。

2 实验内容

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

3 实验步骤

3.1 使用 MPI 进行矩阵乘法

这两个函数在本次实验中可以承担所有的通讯职能，并且使用方法简单。需要达到类似广播的功能只需要在根进程用 for 循环对每个进程都使用 Send 发送一个消息，然后在其他进程调用 Recv 接收就好了。收集结果的时候也使用 for 循环达到了类似 Scatter 的效果。

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// A:m*k B:k*n C:m*n
void matrix_multiply(double *A, double *B, double *C, int rows, int n
    , int kk) {
     for (int i = 0; i < rows; ++i) {
         for (int k = 0; k < kk; ++k) {
             double a = A[i*kk+k];
             for (int j = 0; j < n; ++j) {
                 C[i*n+j] += a * B[k*n+j];
             }
         }
     }
}
```

```
16 void pirnt_mat(double* A,int m,int n){
17     for (int i = 0; i < m; i++)
18     {
19         for (int j = 0; j < n; j++)
20         {
21             printf("%lf␣",A[i*n+j]);
22         }
23         printf("\n");
24     }
25 
26 }
27 
28 int main(int argc,char **argv){
29     // printf("%d\n",argc);
30     // printf("%s\n",argv[2]);
31     //初始化
32     MPI_Init(&argc, &argv);
33     double *C = NULL;
34     int rank, p;
35     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
36     MPI_Comm_size(MPI_COMM_WORLD, &p);
37     int m , n , k ;
38     double *A = NULL;
39     double *B = NULL;
40 
41     if(rank==0){
42         //输入维度
43         printf("input␣the␣m,n,k:\n");
44         scanf("%d%d%d",&m,&n,&k);
45         int mnk[3]={m,n,k};
46         MPI_Bcast(mnk,3,MPI_INT,0,MPI_COMM_WORLD);
47         m=mnk[0];n=mnk[1];k=mnk[2];
48         //初始化A和B矩阵
49         A=(double*)malloc(m*k*sizeof(double));
50         B=(double*)malloc(k*n*sizeof(double));
51         C=(double*)malloc(m*n*sizeof(double));
52         for (int i = 0; i < m*k; i++)
53         {
54             A[i]=i+1;
55         }
```

```
56          for (int i = 0; i < k*n; i++)
57          {
58              B[i]=i+1;
59          }
60          //规划数据的分组
61          int *rows_per_rank = (int *)malloc(p * sizeof(int));
62          int *start_row_per_rank = (int *)malloc(p * sizeof(int));
63          int remainder = m % p;
64          int base_rows = m / p;
65          int current_start = 0;
66          for (int i = 0; i < p; i++)
67          {
68              //前remainder组多算一行，刚好不多余
69              rows_per_rank[i]=(i<remainder)?base_rows+1:base_rows;
70              start_row_per_rank[i]=current_start;
71              current_start+=rows_per_rank[i];
72          }
73          //发送切好片的A矩阵和整个B矩阵，发rows_per_rank[i]是因为接收
                方需要用来接收矩阵A和发送矩阵C
74          for (int i = 1; i < p; i++)
75          {
76              MPI_Send(&rows_per_rank[i],1,MPI_INT,i,0,MPI_COMM_WORLD);
77              MPI_Send(A+start_row_per_rank[i]*k,rows_per_rank[i]*k,
                    MPI_DOUBLE,i,1,MPI_COMM_WORLD);
78              MPI_Send(B,k*n,MPI_DOUBLE,i,2,MPI_COMM_WORLD);
79          }
80          //矩阵乘（调整过循环顺序）
81          matrix_multiply(A,B,C,rows_per_rank[0],n,k);
82          for (int i = 1; i < p; i++)
83          {
84              //收集结果
85              MPI_Recv(C+start_row_per_rank[i]*n,rows_per_rank[i]*n,
                    MPI_DOUBLE,i,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
86          }
87          pirnt_mat(C,m,n);
88          free(A);
89          free(B);
90          free(C);
91          free(rows_per_rank);
92          free(start_row_per_rank);
```

```
93      }
94      else{
95          //获取矩阵维度
96          int mnk[3]={};
97          MPI_Bcast(mnk,3,MPI_INT,0,MPI_COMM_WORLD);
98          m=mnk[0];n=mnk[1];k=mnk[2];
99          //获取当前进程需要算的行数
100         int local_rows=0;
101         MPI_Recv(&local_rows,1,MPI_INT,0,0,MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
102         A=(double*)malloc(local_rows*k*sizeof(double));
103         B=(double*)malloc(n*k*sizeof(double));
104         double* C=(double*)malloc(local_rows*n*sizeof(double));
105         //接受矩阵A, B
106         MPI_Recv(A,local_rows*k,MPI_DOUBLE,0,1,MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
107         MPI_Recv(B,n*k,MPI_DOUBLE,0,2,MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
108         matrix_multiply(A,B,C,local_rows,n,k);
109         //返回结果给根进程
110         MPI_Send(C,local_rows*n,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
111         free(A);
112         free(B);
113         free(C);
114     }
115
116     MPI_Finalize();
117     return 0;
118 }
```

4 实验结果

4.1 计时

- 由于我暂时没有调整进程的上限个数, 所以运行环境是在超算习堂
- 测试时间的函数使用了 MPI_Wtime，这是一个高分辨率、经过 (或墙) 时钟。单位为 s。
- 测试结果是统计的最慢的进程所用的时间

计时函数参考：https://learn.microsoft.com/zh-cn/message-passing-interface/mpi-wtime-function

测试代码：

```
    if(rank==0){
        //输入维度
        printf("input the m,n,k:\n");
        scanf("%d%d%d",&m,&n,&k);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    double start=MPI_Wtime();
    ......
    double time=MPI_Wtime()-start;
    printf("process %d costs %lfs\n",rank,time);
    double max_time;
    MPI_Reduce(&time,&max_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD)
        ;
    if(rank==0){
        printf("the max time: %lfs\n",max_time);
    }
    MPI_Finalize();
    return 0;
```

4.2 运行时间

进程数	矩阵规模				
	128	256	512	1024	2048
1	0.012775s	0.082896s	0.643198s	5.206706s	41.496786s
2	0.005949s	0.042818s	0.324621s	2.629395s	21.415073s
4	0.004561	0.024158s	0.172887s	1.381328s	10.611006s
8	0.004862s	0.015255s	0.255603s	1.826807s	11.766949s
16	0.106172s	0.017869s	0.496006s	2.704538s	18.485188s

4.3 结果分析

我们可以看到，随着进程数的增多，运行时间也不一定越快，结合课上所学和自己的思考，我觉得主要是以下两点原因：

- 超算习堂的实验环境中，真实的物理核数是 4，那么当进程数超过物理核之后，系统就会对多个进程进行调度，让这些进程轮流上处理器运行。所以当进程数小于等于物理核对时候，加速比和我们所设想的增长几乎一致，两个核快两倍，四个核快四倍，但是多余四个核表现不佳，这是因为处于运行中的进程有限，其他的大多处于挂起状态。

- 其次我考虑到的原因是进程间的通信开销，因为进程越多，需要通信的次数也越多，通信的开销也会变大。

我们横向对比前三行（进程数为 1,2,4），会发现每一行的前一个耗时是后一个耗时的 8 倍左右，这是符合我们预期的。因为矩阵乘法是 $O(n^3)$ 开销，那么 $n->2n$，总开销就扩大 8 倍。
当进程数多于物理核的时候，时间开销就和操作系统对进程的调度有更强的关系了，所以 8 倍关系不是特别明显。

5 讨论题

5.1 在内存受限情况下，如何进行大规模矩阵乘法计算？

回答：在内存受限的情况下，如果我们要进行大规模的矩阵乘法，那么我们需要特别关注的地方一定是磁盘 I/O 时间，因为内存受限，我们的内存可能无法存下太多数据，有部分矩阵的数据就要从磁盘中读取，这往往是很费时间的操作；其次，我们也可以关注 cache 的命中率（可以使用循环展开，调整循环顺序等）。

- 矩阵分块：将大矩阵 A（m×n）和 B（n×p）划分为适合内存的子块。例如：

  将 A 按行划分为多个大小为 $m_b \times n$ 的块。

  将 B 按列划分为多个大小为 $n \times p_b$ 的块。

  将结果 C 存储为 $m_b \times p_b$ 大小的块。

  假设矩阵中元素是 double，那么要保证
  $(m_b * n + n * p_b + m_b * p_b) * sizeof(double) \leq$ 内存大小

- 可以固定已经加载进入内存的 A 或者 B 的块，若固定了 B 的块（即 B 的某几列），那么我们可以依次加载 A 的所有块进入内存，即只替换 A 的块，以此来减少 I/O 次数。

- 按行或列顺序存储矩阵数据，减少磁盘寻道时间。例如，将 A 的行块连续存储，B 的列块按访问顺序存储（列优先存储），这样可以让读入内存的块有尽可能多的我们想要的数据。

- 结合循环展开，循环顺序调整，以及 SIMD 指令来计算 C 矩阵中元素来加速计算。

- 结合 Strassen 等快速矩阵乘法算法，减少计算量，间接降低内存压力。

5.2 如何提高大规模稀疏矩阵乘法性能？

回答：

用 CSR 格式压缩存储矩阵 A 和矩阵 B：

- row_ptr：长度为 m+1（m 为行数），row_ptr[i] 到 row_ptr[i+1]-1 为第 i 行的非零元素索引。
- col_indices：非零元素的列索引。
- values：非零元素的值。

此时我们应该把矩阵 B 也按照行优先存储，因为下面的算法是调整过循环顺序的（也就是我们不会在第三次循环中直接算出 $C_{i,j}$）

```
        // 假设C初始化为全零的密集矩阵
    for (int i = 0; i < A_rows; i++) {
        for (int k_ptr = A.row_ptr[i]; k_ptr < A.row_ptr[i+1]; k_ptr
            ++) {
            int k = A.col_indices[k_ptr];
            float A_ik = A.values[k_ptr];
            for (int j_ptr = B.row_ptr[k]; j_ptr < B.row_ptr[k+1];
                j_ptr++) {
                int j = B.col_indices[j_ptr];
                float B_kj = B.values[j_ptr];
                C[i][j] += A_ik * B_kj; // 累加到结果矩阵
            }
        }
    }
```

这样我们就在牺牲了一部分空间的代价下，跳过了对矩阵元素 0 的运算。