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 林隽哲

学 号 21312450

学 院 计算机学院

专 业 计算机科学与技术

2025 年 4 月 7 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 实验结果

3.1 实验核心代码

```
int main(int argc, char *argv[]) {
    int n = 128;
    int rank, size;
    double *A = NULL, *B = NULL, *C = NULL;
    double *local_A = NULL, *local_C = NULL;
    double start_time, end_time;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (n % size != 0) {
        if (rank == 0) {
            printf("Matrix size (%d) must be divisible by number of
                processes (%d)\n", n, size);
        }
        MPI_Finalize();
        return 1;
    }

    int rows_per_proc = n / size;
```

```
21
22      // Allocate memory for local matrices
23      local_A = (double*)malloc(rows_per_proc * n * sizeof(double));
24      local_C = (double*)malloc(rows_per_proc * n * sizeof(double));
25      memset(local_C, 0, rows_per_proc * n * sizeof(double));
26
27      // Root process initializes matrices
28      if (rank == 0) {
29          A = (double*)malloc(n * n * sizeof(double));
30          B = (double*)malloc(n * n * sizeof(double));
31          C = (double*)malloc(n * n * sizeof(double));
32
33          srand(time(NULL));
34          initialize_matrix(A, n, n);
35          initialize_matrix(B, n, n);
36
37          if (n <= 10) {
38              printf("Matrix A:\n");
39              print_matrix(A, n, n);
40              printf("Matrix B:\n");
41              print_matrix(B, n, n);
42          }
43      }
44
45      start_time = MPI_Wtime();
46
47      // Distribute matrix A using point-to-point communication
48      if (rank == 0) {
49          // Root process sends parts of A to other processes
50          for (int dest = 1; dest < size; dest++) {
51              MPI_Send(A + dest * rows_per_proc * n,
52                       rows_per_proc * n,
53                       MPI_DOUBLE,
54                       dest,
55                       0,
56                       MPI_COMM_WORLD);
57          }
58          // Root process keeps its own part
59          memcpy(local_A, A, rows_per_proc * n * sizeof(double));
60      } else {
```

```
61          // Other processes receive their part of A
62          MPI_Recv(local_A,
63                  rows_per_proc * n,
64                  MPI_DOUBLE,
65                  0,
66                  0,
67                  MPI_COMM_WORLD,
68                  MPI_STATUS_IGNORE);
69      }
70
71      // Distribute matrix B using point-to-point communication
72      if (rank == 0) {
73          // Root process sends B to all other processes
74          for (int dest = 1; dest < size; dest++) {
75              MPI_Send(B,
76                       n * n,
77                       MPI_DOUBLE,
78                       dest,
79                       1,
80                       MPI_COMM_WORLD);
81          }
82      } else {
83          // Other processes allocate memory for B and receive it
84          B = (double*)malloc(n * n * sizeof(double));
85          MPI_Recv(B,
86                  n * n,
87                  MPI_DOUBLE,
88                  0,
89                  1,
90                  MPI_COMM_WORLD,
91                  MPI_STATUS_IGNORE);
92      }
93
94      // Perform local matrix multiplication
95      for (int i = 0; i < rows_per_proc; i++) {
96          for (int j = 0; j < n; j++) {
97              for (int k = 0; k < n; k++) {
98                  local_C[i * n + j] += local_A[i * n + k] * B[k * n +
                        j];
99              }
```

```
100         }
101     }
102 
103     // Gather results using point-to-point communication
104     if (rank == 0) {
105         // Root process receives results from other processes
106         for (int src = 1; src < size; src++) {
107             MPI_Recv(C + src * rows_per_proc * n,
108                     rows_per_proc * n,
109                     MPI_DOUBLE,
110                     src,
111                     2,
112                     MPI_COMM_WORLD,
113                     MPI_STATUS_IGNORE);
114         }
115         // Root process copies its own result
116         memcpy(C, local_C, rows_per_proc * n * sizeof(double));
117     } else {
118         // Other processes send their results to root
119         MPI_Send(local_C,
120                 rows_per_proc * n,
121                 MPI_DOUBLE,
122                 0,
123                 2,
124                 MPI_COMM_WORLD);
125     }
126 
127     end_time = MPI_Wtime();
128 
129     // Print result matrix if small enough
130     if (rank == 0) {
131         if (n <= 10) {
132             printf("Result␣Matrix␣C:\n");
133             print_matrix(C, n, n);
134         }
135         printf("Matrix␣size:␣%d␣x␣%d\n", n, n);
136         printf("Number␣of␣processes:␣%d\n", size);
137         printf("Execution␣time:␣%f␣seconds\n", end_time - start_time)
            ;
138     }
```

```
139
140     // Free memory
141     if (rank == 0) {
142         free(A);
143         free(C);
144     }
145     free(B);
146     free(local_A);
147     free(local_C);
148
149     MPI_Finalize();
150     return 0;
151 }
```

3.2 运行时间

根据运行结果，填入下表以记录不同进程数和矩阵规模下的运行时间：

进程数	矩阵规模				
	128	256	512	1024	2048
1	0.008940	0.075898	0.705587	6.545637	56.328150
2	0.004838	0.033641	0.367404	3.555477	29.233684
4	0.008679	0.049823	0.292343	2.215296	17.302856
8	0.009288	0.074531	0.210484	1.316330	11.103513
16	0.067242	0.024242	0.161879	1.270833	9.155628

表 1: 不同进程数和矩阵规模下的运行时间 (单位: 秒)

根据上表的结果数据，我可以进行以下性能分析：

3.3 加速比分析

将进程数为 1 的情况作为基准，计算不同进程数下的加速比：

进程数	矩阵规模				
	128	256	512	1024	2048
1	1.00	1.00	1.00	1.00	1.00
2	1.85	2.26	1.92	1.84	1.93
4	1.03	1.52	2.41	2.96	3.26
8	0.96	1.02	3.35	4.97	5.07
16	0.13	3.13	4.36	5.15	6.15

表 2: 不同进程数和矩阵规模下的加速比

3.4 性能分析

3.4.1 小规模矩阵（128 256）

- 观察小规模矩阵下的运行时间，可以发现，随着进程数的增加，运行时间减少直到进程数为 16。
- 观察小规模矩阵下的加速比，可以发现，随着进程数的增加，加速比先增加后减少。
- 进程数的增加将会同时导致通信开销的增加，当通信开销大于计算收益时，运行时间将会不降反增，加速比也将会降低。

3.4.2 中大规模矩阵（512 2048）

- 观察中大规模矩阵的运行时间与加速比，可以发现，随着矩阵规模的增加，进程数的增加所带来的收益将会愈发明显。

4 讨论题

- 在内存受限情况下，如何进行大规模矩阵乘法计算？
  - 分块计算：将大矩阵分成多个小块，每次只加载部分数据到内存中进行计算。这种方法可以显著减少内存使用量，但会增加 I/O 操作。
  - 分布式计算：将矩阵分布到多个计算节点上进行计算，每个节点只处理部分数据。
  - 数据压缩：对矩阵数据进行压缩存储，在计算时再解压。
- 如何提高大规模稀疏矩阵乘法性能？

- 使用稀疏矩阵存储格式：如 CSR（Compressed Sparse Row）或 CSC（Compressed Sparse Column）等压缩存储格式，只存储非零元素，减少内存使用和计算量。
- 对数据预处理：在计算前对矩阵进行预处理，如矩阵重排序，将非零元素尽可能聚集在一起，减少计算量。
- 并行计算：使用并行计算技术，如 MPI，将矩阵分布到多个计算节点上进行计算，每个节点只处理部分数据。