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      if (rank == 0) {
23          A = (double*)malloc(n * n * sizeof(double));
24          B = (double*)malloc(n * n * sizeof(double));
25          C = (double*)malloc(n * n * sizeof(double));
26
27          srand(time(NULL));
28          initialize_matrix(A, n, n);
29          initialize_matrix(B, n, n);
30
31          if (n <= 10) {
32              printf("Matrix␣A:\n");
33              print_matrix(A, n, n);
34              printf("Matrix␣B:\n");
35              print_matrix(B, n, n);
36          }
37      }
38
39      local_A = (double*)malloc(rows_per_proc * n * sizeof(double));
40      local_C = (double*)malloc(rows_per_proc * n * sizeof(double));
41      memset(local_C, 0, rows_per_proc * n * sizeof(double));
42
43
44      start_time = MPI_Wtime();
45
46      MPI_Scatter(A, rows_per_proc * n, MPI_DOUBLE,
47                  local_A, rows_per_proc * n, MPI_DOUBLE,
48                  0, MPI_COMM_WORLD);
49
50      if (rank == 0) {
51          MPI_Bcast(B, n * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
52      } else {
53          B = (double*)malloc(n * n * sizeof(double));
54          MPI_Bcast(B, n * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
55      }
56
57      // Perform local matrix multiplication
58      for (int i = 0; i < rows_per_proc; i++) {
59          for (int j = 0; j < n; j++) {
60              for (int k = 0; k < n; k++) {
```

```
                local_C[i * n + j] += local_A[i * n + k] * B[k * n +
                    j];
            }
        }
    }

    // Gather results back to the root process
    MPI_Gather(local_C, rows_per_proc * n, MPI_DOUBLE,
               C, rows_per_proc * n, MPI_DOUBLE,
               0, MPI_COMM_WORLD);

    end_time = MPI_Wtime();

    // Print result matrix if small enough
    if (rank == 0) {
        if (n <= 10) {
            printf("Result Matrix C:\n");
            print_matrix(C, n, n);
        }
        printf("Matrix size: %d x %d\n", n, n);
        printf("Number of processes: %d\n", size);
        printf("Execution time: %f seconds\n", end_time - start_time)
            ;
    }

    // Free memory
    if (rank == 0) {
        free(A);
        free(C);
    }
    free(B);
    free(local_A);
    free(local_C);

    MPI_Finalize();
    return 0;
}
```

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	0.019479	0.170543	1.404218	7.370951	71.796604
2	0.012254	0.082274	0.821596	7.047319	68.997478
4	0.007633	0.381321	0.702947	4.223849	35.688534
8	0.222525	0.351655	0.520122	2.421282	19.263383
16	0.377614	0.390133	0.399883	1.814683	9.326502

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	1.59	2.07	1.71	1.05	1.04
4	2.55	0.45	2.00	1.74	2.01
8	0.09	0.48	2.70	3.04	3.73
16	0.05	0.44	3.51	4.06	7.70

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。