

# Parallel-Programming Task1

刘森元, 21307289

中山大学计算机学院

Codes on <https://github.com/Myocardial-infarction-Jerry/Parallel-Programming/tree/main/Task1>.

Project built by CMake.

```
1 | > cd Task1
2 | > cmake . && make
3 | > mpiexec MPIMatMul -n 8 // Customize running
4 | > mpirun MPIMatMul // Using default setting
```

## 1 Environment

11th Gen Intel(R) Core(TM) i7-11700KF @ 3.60GHz

NVIDIA GeForce RTX 3080 Ti O12G

Windows Subsystem for Linux @ Ubuntu 22.04 LTS

## 2 Task

使用 MPI 点对点通信方式实现并行通用矩阵乘法 (MPI-v1), 并通过实验分析不同进程数量、矩阵规模时该实现的性能.

### Note

**输入:**  $m, n, k$  三个整数, 每个整数的取值范围均为  $[128, 2048]$ .

**问题描述:** 随机生成  $m \times n$  的矩阵  $A$  以及  $n \times k$  的矩阵  $B$ , 并对这两个矩阵进行矩阵乘法运算, 得到矩阵  $C$ .

**输出:**  $A, B, C$  三个矩阵, 及矩阵计算所消耗的时间  $t$ .

**要求:**

- 使用 MPI 点对点通信实现并行矩阵乘法, 调整并记录不同线程数量 (1~16) 及矩阵规模 (128~2048) 下的时间开销, 填写下表, 并分析其性能.
- 根据当前实现, 在实验报告中讨论两个优化方向:
  - 在内存有限的情况下, 如何进行大规模矩阵乘法计算?
  - 如何提高大规模稀疏矩阵乘法性能?

## 3 Theory

由于矩阵的可分性, 使用并行通用矩阵乘法的朴素思想, 即将矩阵分割为  $size$  等份, 在多个线程中进行计算, 最后由 master 核进行拼接.

$a_{00}$	$a_{01}$	$\cdots$	$a_{0,n-1}$	$\begin{matrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{matrix}$	$=$	$y_0$
$a_{10}$	$a_{11}$	$\cdots$	$a_{1,n-1}$			$y_1$
$\vdots$	$\vdots$		$\vdots$			$\vdots$
$a_{i0}$	$a_{i1}$	$\cdots$	$a_{i,n-1}$			$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
$\vdots$	$\vdots$		$\vdots$			$\vdots$
$a_{m-1,0}$	$a_{m-1,1}$	$\cdots$	$a_{m-1,n-1}$			$y_{m-1}$

如上图的逐行计算, 对于  $m \times n$  的矩阵  $A$ , 可将其切分为若干个  $subM \times n$  子矩阵分别与矩阵  $B$  相乘, 最后进行拼接, 其中  $subM = (m + size - 1)/size$ , 保证均分.

## 4 Code

### ⚠ Caution

源代码详见 [MPIMatMul.cpp](#).

### 4.1 进行朴素矩阵乘法

```

1 void matMul(float *A, float *B, float *C, int m, int n, int k) {
2     for (int i = 0; i < m; i++) {
3         for (int j = 0; j < k; j++) {
4             C[i * k + j] = 0;
5             for (int l = 0; l < n; l++) {
6                 C[i * k + j] += A[i * n + l] * B[l * k + j];
7             }
8         }
9     }
10 }
```

### 4.2 master 核进行分发操作

```

1 // Core 0
2 int subM = (m + size - 1) / size;
3 std::cerr << "subM = " << subM << std::endl;
4 matMul(A, B, C, subM, n, k);
5
6 // Core 1 ~ size - 1
7 for (int i = 1; i < size; ++i) {
8     if (subM * i >= m)
9         break;
10
11     int rem = std::min(subM, m - subM * i);
12
13     MPI_Send(&subM, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
14     MPI_Send(&n, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
15     MPI_Send(&k, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
16     MPI_Send(&A[subM * i * n], rem * n, MPI_FLOAT, i, 0, MPI_COMM_WORLD);
17     MPI_Send(B, n * k, MPI_FLOAT, i, 0, MPI_COMM_WORLD);
18 }
```

```

19
20 for (int i = 1; i < size; ++i) {
21     if (subM * i >= m)
22         break;
23
24     MPI_Recv(&C[subM * i * k], subM * k, MPI_FLOAT, i, 0, MPI_COMM_WORLD, &status);
25 }

```

### 4.3 sub 核进行计算

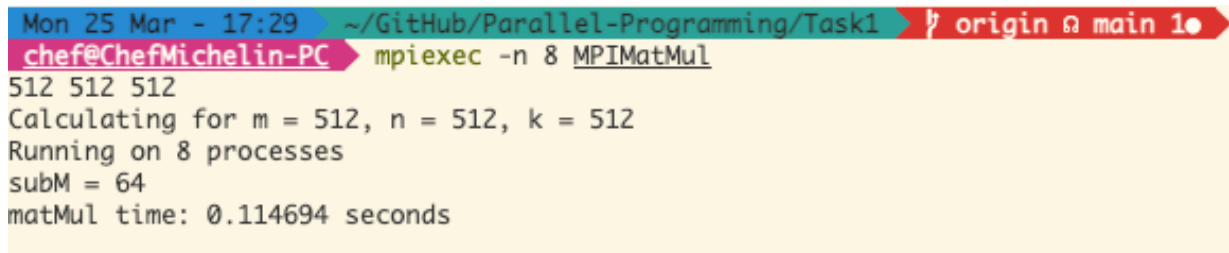
```

1 int subM, n, k;
2 MPI_Recv(&subM, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
3 MPI_Recv(&n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
4 MPI_Recv(&k, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
5
6 auto A = new float[subM * n];
7 auto B = new float[n * k];
8 auto C = new float[subM * k];
9
10 MPI_Recv(A, subM * n, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
11 MPI_Recv(B, n * k, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
12
13 matMul(A, B, C, subM, n, k);
14
15 MPI_Send(C, subM * k, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);

```

## 5 Result

以 8 核心 512 矩阵为例



```

Mon 25 Mar - 17:29 ~/GitHub/Parallel-Programming/Task1 origin main 1
chef@ChefMichelin-PC: ~$ mpiexec -n 8 MPIMatMul
512 512 512
Calculating for m = 512, n = 512, k = 512
Running on 8 processes
subM = 64
matMul time: 0.114694 seconds

```

其中 `matMul time` 为并行矩阵乘法运行时间, `Running time` 为串行矩阵乘法运行时间.

进程数量/矩阵规模 (s)	128	256	512	1024	2048
1	0.00606344	0.047099	0.464436	3.86655	40.2521
2	0.0030436	0.0235597	0.300408	2.32682	39.563
4	0.003084	0.0186486	0.218002	1.79658	28.0431
8	0.00186049	0.0124097	0.114694	0.980662	16.3615
16	0.00168044	0.00781224	0.07891	0.63462	12.9322

- 随着进程数量的增加, 程序的运行时间在大多数情况下都有所减少.
- 当进程数量增加到16时, 对于规模为2048的矩阵, 运行时间并没有显著减少. 这是因为进程间的通信开销开始超过了并行计算带来的收益, 即 Amdahl 定律.

- 随着矩阵规模的增加, 程序的运行时间也在增加. 矩阵乘法的计算复杂度为  $O(n^3)$ , 所以当矩阵规模增加时, 所需的计算量也会显著增加.
- 程序在多进程下表现出了良好的性能提升, 但当进程数量增加到一定程度后, 通信开销可能会开始影响性能.

## 6 Optimization

### 6.1 内存有限情况下如何进行大规模矩阵乘法计算?

1. **分块计算 (Block Multiplication)**: 将大矩阵分解为多个小矩阵 (块), 然后分别计算这些小矩阵的乘积, 最后再将结果组合起来. 这种方法可以减少内存的使用, 因为在任何时候, 你只需要在内存中存储一小部分的数据.
2. **使用磁盘存储**: 使用磁盘存储. 你可以将矩阵写入磁盘文件, 然后在计算时只读取所需的部分. 这种方法的缺点是磁盘读写速度远低于内存, 可能会降低计算速度.
3. **使用稀疏矩阵**: 使用稀疏矩阵格式来存储和计算. 稀疏矩阵只存储非零元素, 可以大大减少内存使用.

### 6.2 如何提高大规模稀疏矩阵乘法性能?

1. **使用迭代方法**: 对于大规模稀疏矩阵, 直接方法 (如高斯消元) 可能会非常慢. 迭代方法, 如共轭梯度法 (Conjugate Gradient) 和 GMRES.
2. **预条件**: 预条件可以改善矩阵的条件数, 使迭代方法收敛得更快. 对于稀疏矩阵, 常用的预条件方法包括不完全 LU 分解 (Incomplete LU decomposition) 和不完全 Cholesky 分解 (Incomplete Cholesky decomposition) .
3. **优化内存访问**: 稀疏矩阵的存储和访问模式可能会导致大量的缓存未命中, 降低计算性能. 通过重新排序矩阵的行和列, 可以优化内存访问模式, 提高缓存利用率, 从而提高计算性能.