Parallel-Programming Task5

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/Task5.

Project built by CMake.

```
> cd Task5
> cmake . && make
> ./OpenMPMatMul
```

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

2.1 OpenMP 通用矩阵乘法

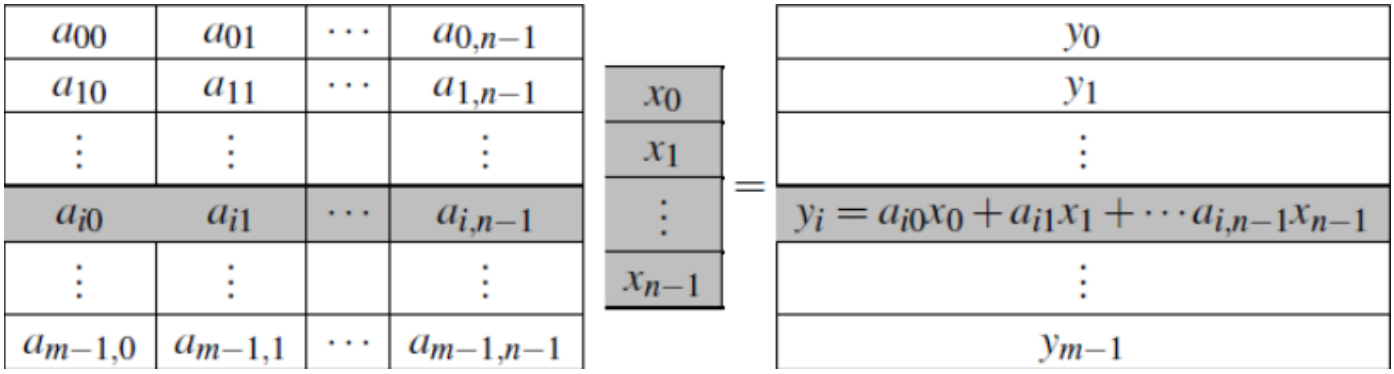
使用OpenMP实现并行通用矩阵乘法，并通过实验分析不同进程数量、矩阵规模、调度机制时该实现的性能。

> **Note**
>
> **输入**：m, n, k 三个整数，没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 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 OpenMP 多线程实现并行矩阵乘法，设置不同线程数量（1-16）、矩阵规模（128-2048）、调度模式（默认、静态、动态调度），通过实验分析程序的并行性能。选做：根据运行时间，对比使用 OpenMP 实现并行矩阵乘法与使用 Pthreads 实现并行矩阵乘法的性能差异，并讨论分析。

3 Theory

3.1 OpenMP 通用矩阵乘法

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。



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

4.1 OpenMP 通用矩阵乘法

> **Caution**
>
> 源代码详见 OpenMPMatMul.cpp

4.1.1 朴素矩阵乘法

```
void matMul(float *A, float *B, float *C, int m, int n, int k) {
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < k; ++j) {
            float sum = 0.0f;
#pragma omp parallel for num_threads(num_thread) reduction(+:sum)
            for (int l = 0; l < n; ++l) sum += A[i * n + l] * B[l * k + j];
            C[i * k + j] = sum;
        }
    }
}
```

5 Result

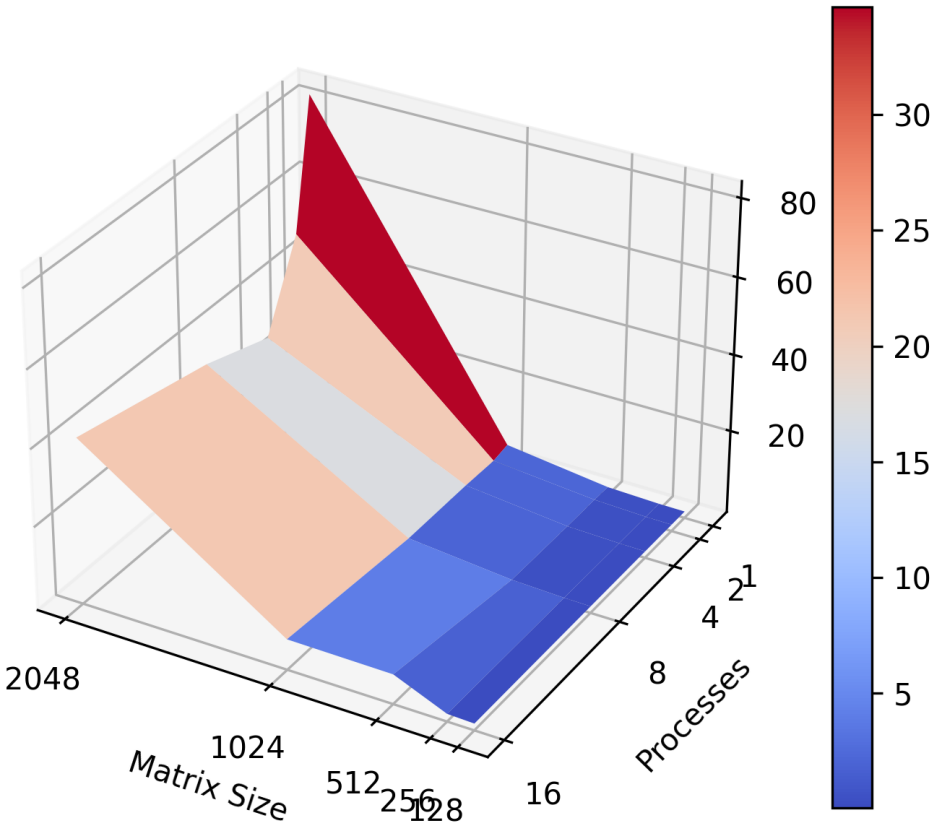
5.1 OpenMP 通用矩阵乘法

以 8 核心 512 矩阵默认调度模式下为例

```
~/Parallel-Programing/Task5  main !6
./OpenMPMatMul 512 512 512 8
Calculating for m = 512, n = 512, k = 512
Running on 8 threads
matMul time: 0.659245 seconds
Verifying with Python script
The matrix multiplication is correct.
```

其中 `matMul time` 为并行矩阵乘法运行时间.

进程数量/矩阵规模 (s)	128	256	512	1024	2048
1	0.014097	0.070119	0.543809	4.048968	82.492855
2	0.017437	0.091779	0.627493	3.168582	48.964278
4	0.020966	0.111330	0.547351	3.426131	27.645359
8	0.038651	0.156202	0.697604	3.350200	33.643892
16	0.054763	0.279289	5.988570	6.243032	41.413419



- 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.
- 代码的扩展性受限于线程数量, 当线程数量有限时, 并行化带来的收益并不显著.

以 8 核心 2048 矩阵为例, `chunk_size = 16`, 不同调度模式下时间对比

默认	静态	动态
31.196983	22.504740	67.854195

- **默认调度模式**：在默认调度模式下，OpenMP 运行时环境决定如何分配迭代。这可能导致负载不均衡，尤其是在迭代的计算成本不均匀的情况下。这可能是默认调度模式下时间较长的原因。
- **静态调度模式**：在静态调度模式下，迭代被均匀地分配给各个线程。如果所有迭代的计算成本都大致相同，那么这种调度模式通常能提供最好的性能，因为它可以最大限度地减少线程之间的同步开销。这可能是静态调度模式下时间较短的原因。
- **动态调度模式**：在动态调度模式下，迭代被动态地分配给各个线程。这种调度模式适用于迭代的计算成本不均匀的情况，因为它可以在运行时动态地平衡负载。然而，动态调度模式的开销通常比静态调度模式要大，因为它需要在运行时进行更多的同步和调度操作。这可能是动态调度模式下时间较长的原因。