# Parallel-Programming Task6

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/Task6.

Project built by CMake.

```
> cd Task6
> cmake . && make
> bin/MatMul
```

## 1 Environment

Apple M1 Pro

macOS Sonoma 14.4.1

## 2 Task

### 2.1 构造基于 Pthreads 的并行 for 循环分解、分配、执行机制

> **Note**
>
> 模仿 OpenMP 的 `omp_parallel_for` 构造基于 Pthreads 的并行 for 循环分解、分配及执行机制。次内容延续上次实验，在本次试验中完成。
>
> **问题描述**：生成一个包含 `parallel_for` 函数的动态链接库（.so）文件，该函数创建多个 Pthreads 线程，并行执行 `parallel_for` 函数的参数所指定的内容。
>
> **函数参数**：`parallel_for` 函数的参数应当指明被并行循环的索引信息，循环中所需要执行的内容，并行构造等。以下为 `parallel_for` 函数的基础定义，实验实现应包括但不限于以下内容：
>
> ```
> void parallel_for(int start,
>                   int end,
>                   int inc,
>                   void *(*functor)(int, void *),
>                   void *arg,
>                   int num_threads);
> ```
>
> - `start, end, inc` 分别为循环的开始、结束以及索引自增量；
> - `functor` 为函数指针，定义了每次循环所执行的内容；
> - `arg` 为 `functor` 的参数指针，给出了 `functor` 执行所需的数据；
> - `num_threads` 为期望产生的线程数量。

> **要求**：完成 `parallel_for` 函数实现并生成动态链接库文件，并以矩阵乘法为例，测试其实现的正确性及效率。

### 2.2 parallel_for 并行应用

> **Note**
>
> 使用此前构造的 `parallel_for` 并行结构，将 `heated_plate_openmp` 改造为基于 Pthreads 的并行应用。
>
> ***heated plate* 问题描述**：规则网格上的热传导模拟，其具体过程为每次循环中通过对邻域内热量平均模拟热传导过程，即：
>
> $$w_{i,j}^{t+1} = \frac{1}{4}(w_{i-1,j-1}^{t} + w_{i-1,j+1}^{t} + w_{i+1,j-1}^{t} + w_{i+1,j+1}^{t})$$
>
> **要求**：使用此前构造的 `parallel_for` 并行结构，将 `heated_plate_openmp` 实现改造为基于 Pthreads 的并行应用。测试不同线程、调度方式下的程序并行性能，并于原始 `heated_plate_openmp.c` 实现对比。

## 3 Theory

1. 在多线程环境中加速执行任务，将一组独立的循环迭代分配给不同的线程，减少整体执行时间。
2. **任务分配**：
   - 将循环迭代范围划分成多个小任务，每个线程负责执行其中的一部分。
   - 各个线程的起始迭代值由线程索引决定，每个线程以固定的增量执行循环。
3. **增量计算**：
   - 根据线程数和循环增量，计算每个线程的迭代增量，使得线程之间没有重叠或遗漏。
   - 线程的起始点：`起始 + 线程索引 * 增量`
   - 线程的步长：`线程数 * 增量`
4. **线程执行**：
   - 每个线程独立执行其任务区间，避免线程间的资源竞争。
   - 使用同步原语（如 `join`）等待所有线程完成。
5. **任务执行函数**：
   - 用户自定义工作函数（“任务执行函数”）接受迭代索引和附加参数。
   - 每个线程调用任务执行函数来完成其分配的循环迭代。
6. **负载均衡**：
   - 通过均匀分配循环迭代次数，保证所有线程的工作量大致均衡。

## 4 Code

### 4.1 OpenMP 通用矩阵乘法

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

```cpp
#include "parallel_for.h"
#include <iostream>

void parallel_for(int start, int end, int inc, void *(*functor)(int, void *), void *args,
int num_threads) {
    std::vector<std::thread> threads;
    for (int rank = 0; rank < num_threads; ++rank) {
        threads.push_back(std::thread([=] {
            for (int i = start + rank * inc; i < end; i += num_threads * inc) {
                functor(i, args);
            }
            }));
    }

    for (auto &thread : threads)
        thread.join();
}
```

# 5 Result

## 5.1 构造基于 Pthreads 的并行 for 循环分解、分配、执行机制

以 8 核心 512 矩阵默认调度模式下为例

```
~/Documents/GitHub/Parallel-Programming/Task6 main !1
bin/MatMul 512 8
Generating random numbers...
Calculating vectors with 8 threads...
Time elapsed: 104.509 ms
Output results to file...
Running Python script to verify the results...
The result is correct!
```

| 进程数量/矩阵规模 (ms) | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|
| 1 | 12.0448 | 58.395 | 473.459 | 3879.63 | 74824.5 |
| 2 | 4.01892 | 29.8944 | 239.457 | 2023.74 | 39383.8 |
| 4 | 2.12513 | 15.1242 | 123.151 | 1059.01 | 21546.9 |
| 8 | 1.83987 | 13.2332 | 86.3407 | 717.779 | 15188.3 |
| 16 | 1.93542 | 11.0887 | 85.989 | 739.387 | 15590.6 |

- 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.

## 5.2 parallel_for 并行应用

> **Caution**
>
> 代码详见 HeatedPlate.cpp

使用 `parallel_for` 改造后有结果

| 实现方式 | 运行时间 |
| --- | --- |
| Pthread | 11.989297 |
| OpenMP | 9.244094 |

1. **OpenMP 的优势：**
   - **线程管理：** OpenMP 具有自动的线程池管理和调度功能，可以根据系统可用的线程资源动态调整线程的分配与执行。
   - **工作分配：** 使用 `#pragma omp parallel for` 指令，OpenMP 能够自动将循环迭代分配给多个线程，实现负载均衡。
   - **线程同步：** OpenMP 提供了一系列同步原语（如 `critical` 和 `reduction`），可以安全高效地管理共享数据。
2. **Pthread 与 OpenMP 的性能差异：**
   - **手动线程管理：** 在 Pthread 中，需要手动创建和管理线程池，增加了编程复杂性。
   - **同步开销：** Pthread 的同步操作需要使用显式的锁，增加了同步开销，尤其是在大量线程竞争时。
   - **负载均衡：** Pthread 的工作分配需要手动控制，在负载均衡上可能不如 OpenMP 自动化的策略高效。