# Lab6: 基于 Pthreads 的并行热导模拟实验报告

**姓名：蔡可豪**
**学号：22336018**

# 1. 引言

本实验旨在将一个已有的热导模拟程序（heated plate问题）从 OpenMP 版本（或一个串行基准）改造为使用我们此前构造的 `parallel_for` Pthreads 并行结构的应用程序。实验的目标是实现该 Pthreads 版本，测试其在不同线程数量下的并行性能，并分析其表现。

原始问题描述：模拟规则网格上的热传导，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，公式如下：
$w_{i,j}^{t+1} = \frac{1}{4}(w_{i-1,j-1}^{t} + w_{i-1,j+1}^{t} + w_{i+1,j-1}^{t} + w_{i+1,j+1}^{t})$

实验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，并与原始 `heated_plate_openmp.c` 实现对比。

# 2. 实验环境

- 操作系统: macOS (Darwin)
- CPU: Apple M2
- 编译器: GCC (用于 Pthreads 版本)
- 并行库: Pthreads (通过自定义的 `parallel_for` 结构)
- 相关文件:
  - `lab6/src/heated_plate_pthreads.c`: Pthreads 实现版本
  - `lab6/src/parallel_for.c`: 自定义 `parallel_for` 实现
  - `lab6/src/parallel_for.h`: 自定义 `parallel_for` 头文件
  - `lab6/Makefile`: 编译脚本

# 3. Pthreads 版本实现 (`heated_plate_pthreads.c`)

## 3.1 主要思路

Pthreads 版本的 `heated_plate_pthreads.c` 基于以下思路将热导模拟并行化：

1. **数据共享**：主计算网格 `grid_w` (当前状态) 和 `grid_w_new` (下一状态) 被声明为全局变量，以便 `parallel_for` 的工作线程能够访问它们。
2. **并行区域**：核心的迭代计算（更新网格内部点的值）通过 `parallel_for` 实现并行。`parallel_for` 的循环索引 `i` 直接对应网格的行号（从 `1` 到 `N-2`）。
3. **工作函数 (`heated_plate_functor`)**：传递给 `parallel_for` 的工作函数 `heated_plate_functor` 负责处理单行 `i` 的所有列 `j` (从 `1` 到 `N-2`) 的计算。它根据给定的热传导公式更新 `grid_w_new[i][j]`。
4. **参数传递**：为 `parallel_for` 定义了一个空的结构体 `heated_plate_args`，因为此场景下，主要的迭代变量 `i` 由 `parallel_for` 直接提供给 `functor`，而共享数据（网格）是全局的。

5. **收敛性判断与数据同步**：
    - 在每次 `parallel_for` 调用（完成所有行计算）之后，程序串行计算 `grid_w` 和 `grid_w_new` 之间的最大差值 (`max_cell_diff`)，用于判断是否收敛。
    - 同时，在计算差值的循环中，将 `grid_w_new` 的值拷贝回 `grid_w`，为下一次迭代做准备。
    - 边界条件在初始化后固定，计算只针对内部点。

## 3.2 关键代码片段

**Functor 定义:**

```
// Global variables for the grid (accessible by functor)
double **grid_w;    // Current iteration grid
double **grid_w_new; // Next iteration grid

// ...

void *heated_plate_functor(int i, void *arg_bundle) {
    for (int j = 1; j < N - 1; j++) {
        grid_w_new[i][j] = 0.25 * (grid_w[i-1][j-1] + grid_w[i-1][j+1] + grid_w[i+1][j-1] + grid_w[i+1][j+1]);
    }
    return NULL;
}
```

**主循环中的 parallel_for 调用:**

```
    heated_plate_args p_args;
    // ...
    while (iter < MAX_ITER && diff >= TOLERANCE) {
        parallel_for(1, N - 1, 1, heated_plate_functor, &p_args, num_threads);

        double max_cell_diff = 0.0;
        for (int i = 1; i < N - 1; i++) {
            for (int j = 1; j < N - 1; j++) {
                double cell_diff = fabs(grid_w_new[i][j] - grid_w[i][j]);
                if (cell_diff > max_cell_diff) {
                    max_cell_diff = cell_diff;
                }
                grid_w[i][j] = grid_w_new[i][j];
            }
        }
        diff = max_cell_diff;
        iter++;
    }
```

使用的 `parallel_for` 实现 (`lab5/src/parallel_for.c` 中的版本) 采用静态分块的方式将迭代分配给不同的 Pthreads 线程。

# 4. 实验结果与分析

## 4.1 实验设置

- 网格大小 (N): 100
- 最大迭代次数 (MAX_ITER): 1000
- 收敛阈值 (TOLERANCE): 1e-4
- 测试的线程数: 1, 2, 4, 8

## 4.2 Pthreads 版本性能数据

| 线程数 | 执行时间 (秒) |
| --- | --- |
| 1 | 0.043475 |
| 2 | 0.049158 |
| 4 | 0.053496 |
| 8 | 0.086226 |

所有运行均在 `MAX_ITER` (1000) 次迭代后结束，表明它们达到了最大迭代次数而不是因为差值小于 `TOLERANCE` 而提前收敛（最大差值约为 0.06）。

## 4.3 性能分析

从上表数据可以看出，对于 `N=100` 的网格规模：

1. **无明显加速比**：与单线程版本（0.043秒）相比，增加线程数（2、4、8线程）不仅没有带来性能提升（时间减少），反而导致了执行时间的增加。
2. **性能随线程数增加而下降**：2线程比1线程慢约13%，4线程比1线程慢约23%，8线程比1线程慢约98%。

**可能的原因分析：**

- **问题规模较小**：`N=100` 的网格，其核心计算量（内部点 `(N-2)*(N-2)`）可能不足够大，使得并行带来的计算收益无法覆盖并行本身的开销。
- **并行开销**：
  - `parallel_for` 实现中包含线程创建、参数传递、线程同步 (`pthread_join`) 等操作。对于每次主循环（共 `MAX_ITER` 次）都调用 `parallel_for`，这些开销会累积。
  - 在 `heated_plate_functor` 中，每个线程处理的行数可能较少，导致任务粒度过细。例如，8个线程处理约98行，每个线程约12-13行。这些细粒度任务的频繁调度和管理也会引入开销。
- **数据同步与拷贝**：在 `parallel_for` 完成后，计算最大差值和将 `grid_w_new` 拷贝回 `grid_w` 的操作是串行执行的。虽然这部分耗时不直接随线程数变化，但它构成了总时间中不可并行化的部分（阿姆达尔定律）。
- **内存访问模式与缓存效应**：虽然 `parallel_for` 可能实现了行数据的良好划分，但全局数组的访问以及 `grid_w` 和 `grid_w_new` 之间的切换，在多线程环境下可能会有复杂的缓存交互，但对于当前性能不升反降的现象，主要矛盾更可能是并行开销和问题规模。

为了验证是否是问题规模的原因，可以尝试显著增大 `N` (例如到 500 或 1000)，并重新进行测试。如果问题规模增大后能观察到加速效果，则说明当前的瓶颈主要是并行开销相对于计算量过高。

**调度方式**：
本实验中使用的 `parallel_for` 实现采用的是静态分块调度。实验要求中提到测试不同调度方式，但当前的 `parallel_for` 未提供此功能。如果需要，可以修改 `parallel_for` 以支持例如动态调度等策略，但这超出了本次直接应用 `parallel_for` 的范围。

# 5. 总结与思考

本实验基本完成了将热导模拟问题改造为基于 Pthreads 和自定义 `parallel_for` 结构的并行程序的目标。
通过实验测试，我们观察到在当前 `N=100` 的问题规模下，Pthreads 并行版本并未能展现出预期的加速效果，反而随着线程数的增加，其执行时间有所上升。初步分析认为，这主要是由于并行引入的额外开销（如线程创建与同步、任务切换等）在小规模计算量下，超过了并行计算本身带来的收益。

为了更全面地评估 `parallel_for` 结构的有效性以及 Pthreads 在此类问题上的应用潜力，后续可以从以下几个方面进行更深入的探究：

1. **调整问题规模**：尝试在更大的网格规模（例如 N=500, N=1000甚至更高）下进行测试，观察并行版本是否能展现出良好的加速比。当计算密集度提高后，并行开销占比可能会下降。
2. **优化并行开销**：研究 `parallel_for` 实现，思考有无可能通过例如线程池等技术，来复用线程，减少重复创建和销毁线程的开销，特别是在迭代计算中。
3. **细化性能瓶颈分析**：使用性能分析工具（如 gprof）来更精确地定位程序中的热点和瓶颈，明确开销的具体来源。
4. **探索不同并行策略**：虽然本次实验的 `parallel_for` 是基于静态分块，但可以思考和尝试实现其他调度策略（如动态调度），并比较它们在不同场景下的表现。
5. **完善实验对比**：积极解决 OpenMP 的编译配置问题，以便完成与 OpenMP 版本的直接性能对比。这将为我们理解不同并行编程模型的特点和适用场景提供宝贵数据。

通过本次实验，我们不仅实践了 Pthreads 多线程编程和 `parallel_for` 并行结构的运用，也对并行程序设计中的开销、问题规模、任务粒度等关键因素对性能的影响有了更直观的认识。

# 6. 附录：源代码文件结构

- `lab6/src/heated_plate_pthreads.c`
- `lab6/src/parallel_for.c`
- `lab6/src/parallel_for.h`
- `lab6/Makefile`