中山大學
SUN YAT-SEN UNIVERSITY

并行程序设计与算法实验

Lab6-Pthreads **并行构造**

姓 名 王志杰

学 号 22331095

学 院 计算机学院

专 业 计算机科学与技术

2025 年 4 月 30 日

1 实验目的

- 深入理解 C/C++ 程序编译、链接与构建过程
- 提高 Pthreads 多线程编程综合能力
- 并行库开发与性能验证

2 实验内容

- 基于 Pthreads 的并行计算库实现：parallel_for+ 调度策略
- 动态链接库生成和使用
- 典型问题的并行化改造：矩阵乘法，热传导问题

3 实验过程

3.1 环境与工具

简要说明实验所使用的操作系统、编译器 (gcc/g++) 版本、以及 Pthreads 库。

- 回答：Ubuntu18.04，gcc (GCC) 11.5.0，NPTL 2.38

3.2 核心函数实现 (parallel_for)

简要描述 parallel_for 函数的关键设计思路和实现。重点说明线程创建、任务划分和同步机制。

- **静态调度（SCHED_STATIC）**：
  - 将总迭代次数 total_iters 均分给所有线程，每个线程执行连续的迭代块。
  - 若总迭代数无法整除线程数，前 remaining 个线程额外执行一次迭代，保证负载均衡。
  - 每个线程通过 static_thread_work 函数处理其分配的迭代块，无同步开销。
- **动态调度（SCHED_DYNAMIC）**：
  - 使用共享的 current_iter 记录当前待分配的迭代索引,通过互斥锁(pthread_mutex_t)保证线程安全。
  - 每个线程每次从共享区获取 chunk_size 个迭代任务，循环直到所有任务完成。

– 动态适应负载不均场景，但锁操作引入额外开销。

- **静态调度无需同步**：任务预先划分，线程独立执行，仅需 pthread_join 等待所有线程结束。
- **动态调度依赖互斥锁**：线程通过竞争锁获取任务块，确保共享变量 current_iter 的原子性更新。

```
void parallel_for(int start, int end, int inc, Functor functor, ...) {
    // 计算总迭代次数和实际线程数
    int total_iters = (inc > 0) ? (end - start + inc -1)/inc : ...;
    num_threads = (num_threads > total_iters) ? total_iters : num_threads;

    if (sched_mode == SCHED_STATIC) {
        // 静态调度: 均分迭代块
        int basic_chunk = total_iters / num_threads;
        int remaining = total_iters % num_threads;
        for (int tid = 0; tid < num_threads; tid++) {
            // 计算线程的迭代范围 [iter_start, iter_end)
            int iter_start = basic_chunk * tid + (tid < remaining ? tid : remaining);
            int iter_end = iter_start + basic_chunk + (tid < remaining ? 1 : 0);
            // 创建线程执行 static_thread_work
            pthread_create(&threads[tid], NULL, static_thread_work, targs);
        }
        // 等待所有线程完成
        for (int tid = 0; tid < num_threads; tid++) pthread_join(...);
    }
    else if (sched_mode == SCHED_DYNAMIC) {
        // 动态调度: 共享任务队列 + 互斥锁
        struct dynamic_shared_data shared = { .mutex = PTHREAD_MUTEX_INITIALIZER, ...
            };
        for (int tid = 0; tid < num_threads; tid++) {
            // 所有线程竞争获取 chunk
            pthread_create(&threads[tid], NULL, dynamic_thread_work, &shared);
        }
        // 等待线程并销毁锁
        pthread_mutex_destroy(&shared.mutex);
    }
}

// 动态调度工作函数: 通过锁保护共享迭代计数器
static void* dynamic_thread_work(void *args) {
    struct dynamic_shared_data *shared = (struct dynamic_shared_data *)args;
    while (1) {
        pthread_mutex_lock(&shared->mutex); // 加锁
```

```
        int local_iter = shared->current_iter;
        if (local_iter >= shared->total_iters) {
            pthread_mutex_unlock(&shared->mutex);
            break;
        }
        shared->current_iter += shared->chunk_size; // 更新迭代计数器
        pthread_mutex_unlock(&shared->mutex); // 解锁
        // 执行当前 chunk 的迭代任务
        for (int iter = local_iter; iter < local_iter + chunk_size; iter++) {
            functor(start + iter * inc, arg);
        }
    }
    return NULL;
}
```

Listing 1: parallel_for 函数核心代码片段

3.3 动态库生成与使用

说明生成动态链接库 (.so) 的主要命令或 Makefile 规则，并简述如何在主程序 (如矩阵乘法、热传导) 中链接和调用该库。

```
# Makefile 示例：生成动态库
CC = gcc
CFLAGS = -fPIC -Iinclude -pthread -O3
LDFLAGS = -shared
SRC = src/parallel_for.c
TARGET = libparallel_for.so

$(TARGET): $(SRC)
    $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $^
```

主程序链接动态库的关键步骤：

- **编译选项**：通过 -L 指定库路径，-l 指定库名：

  ```
  # Makefile 示例：链接动态库
  LDFLAGS_PTHREAD = -L../parallel_for -lparallel_for -pthread -Wl,-rpath=../paralle
  heated_plate_pthread: src/heated_plate_pthread.c
      $(CC) $(CFLAGS) $< -o $@ $(LDFLAGS_PTHREAD)
  ```

- **运行时**：需确保动态库路径在 LD_LIBRARY_PATH 中，或通过 -Wl,-rpath 硬编码路径。

3.4 应用测试 (热传导)

简述如何将 parallel_for 应用于热传导问题，替换原有的并行机制。描述测试设置，如网格大小和线程数。

- **替换原有并行机制**:
  - 原 OpenMP 的 `#pragma omp for` 替换为 `parallel_for` 调用。
  - 边界初始化（如 `set_left`）、均值计算（`sum_loop1`）等操作封装为回调函数。
  - 主循环中的迭代（保存旧值、更新温度场、计算差异）均通过 `parallel_for` 并行化。
- **测试设置**:
  - **网格大小**: 500×500 网格点（`ROWS` 与 `COLS` 宏定义）。
  - **线程配置**: 通过命令行参数指定线程数（`-t`）、调度模式（`-s static/dynamic`）和块大小（`-c`）。
  - **收敛条件**: 温度场最大变化小于 $\epsilon = 0.001$。

关键代码片段:

```
// 主循环: 迭代直至收敛
while (diff >= epsilon) {
    // 并行保存旧温度场
    SaveArgs save_args = {u, w, COLS};
    parallel_for(0, ROWS, 1, save_u, &save_args, num_threads, sched_mode, chunk_size);

    // 并行更新温度场
    UpdateArgs update_args = {w, u, COLS};
    parallel_for(1, ROWS-1, 1, update_w, &update_args, num_threads, sched_mode,
        chunk_size);

    // 并行计算最大差异
    diff = 0.0;
    DiffArgs diff_args = {w, u, ROWS, COLS, &diff_mutex, &diff};
    parallel_for(1, ROWS-1, 1, compute_diff, &diff_args, num_threads, sched_mode,
        chunk_size);

    iterations++;
}
```

Listing 2: 热传导主循环并行化

4 实验结果与分析

4.1 性能测试结果

展示不同线程数和调度方式下，自定义 Pthreads 实现与原始 OpenMP 实现的性能对比。

表 1: 热传导问题性能对比 (Pthreads vs OpenMP, 网格大小: M x N)

线程数	**调度方式 (Pthreads)**	**自定义 Pthreads**	**原始 OpenMP**
		时间 (s)	时间 (s)
1 (串行)	N/A	30.07	5.404996
Pthreads: 静态调度 (Static)			
2	Static	20.29	2.755444
4	Static	19.48	1.540102
8	Static	26.33	1.021470
16	Static	45.91	0.880433
...	...	...	...
Pthreads: 动态调度 (Dynamic, chunk=10)			
2	Dynamic	21.33	2.755444
4	Dynamic	21.14	1.540102
8	Dynamic	24.94	1.021470
16	Dynamic	42.71	0.880433
...	...	...	...

4.2 结果分析与总结

简要分析性能结果，说明是否达到了并行加速效果。从表 1 的性能对比数据可以看出，自定义的 Pthreads 并行实现与 OpenMP 版本在加速效果上存在显著差异：

- **单线程性能差距**：
  - Pthreads 串行版本耗时 30.07 秒，而 OpenMP 仅需 5.40 秒。这也是我做实验的时候非常困扰的地方，反复检查好几遍也没查出错误，我认为应该不是代码写错了，可能是由以下原因导致：
    - Pthreads 版本中冗余的锁操作（如均值计算、差异统计）引入了额外开销。
    - OpenMP 的编译器优化（如循环向量化、内存对齐）显著提升了单线程执行效率。

- **多线程扩展性不足**：
  - Pthreads 版本在 2/4 线程时仅获得有限的加速（静态调度 2 线程：30.07 → 20.29 秒），而 OpenMP 版本表现出线性加速趋势（2 线程：5.40 → 2.76 秒）。
  - 高线程数（如 16 线程）时，Pthreads 版本出现性能退化（静态调度：45.91 秒），可能原因包括：
    * 锁竞争加剧：动态调度中全局迭代计数器的频繁加锁导致线程串行化。
    * 任务划分粒度不合理：静态调度的任务块过大（如 500x500 网格划分至 16 线程）导致负载不均。
    * 内存访问瓶颈：多线程同时访问连续内存区域时带宽受限。
- **调度策略影响**：
  - 动态调度（chunk=10）相比静态调度未显著改善性能，说明细粒度任务划分加剧了锁开销。
  - OpenMP 的 static 可能更适应热传导问题的迭代特性。
- **OpenMP 优势**：通过编译器深度优化和高效的运行时调度，在多线程场景下实现了接近线性的加速比。
- **Pthreads 改进方向**：
  - 减少锁操作：使用原子操作（如 `__atomic_fetch_add`）替代互斥锁。
  - 优化任务粒度：根据网格规模动态调整 `chunk_size`，平衡负载与调度开销。
  - 引入 NUMA 感知：绑定线程到特定 CPU 核以减少内存延迟。

注：实验报告格式参考本模板，可在此基础上进行修改；实验代码以 zip 格式另提交；最终提交内容包括实验报告 (pdf 格式) 和实验代码 (zip 压缩包格式)