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 马岱

学 号 22336180

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

- 操作系统：Linux Ubuntu 18.04 LTS (64-bit)
- 编译器 (gcc/g++) 版本：gcc version 7.5.0
- Pthreads 库：ldd (Ubuntu GLIBC 2.27-3ubuntu1.5) 2.27

3.2 核心函数实现 (parallel_for)

在上一个实验中我们实现了 openmp 处理并行矩阵乘法，但是 openmp 实际不需要我们过多实现，因为线程的创建与任务划分都是其自行完成的。而如果使用 pthread 来实现并行矩阵乘法，我们就需要将自行实现线程的创建和任务的划分以及同步机制。接下来我将详细描述我的实验思路和过程。

我们的主要目标是实现 `parallel_for` 函数，在 `parallel_for.h` 中，我们首先用一个简单而通用的函数原型声明了并行循环接口：

```
1 void parallel_for(int start, int end, int inc,
2                   void (*functor)(int, void *), void *args,
3                   int num_threads, enum schedule_type schedule, int
                    ↪  chunk_size);
```

这里 `start` 和 `end` 指定了循环区间，`inc` 是步长，`functor` 则是用户在每次迭代中要执行的函数，而 `args` 则传递给这个函数所需的上下文。为了支持三种调度策略，我们定义了一个 `enum schedule_type`（STATIC、DYNAMIC、GUIDED），并在内部维护一个

thread_args_t 结构，既保存了循环参数，也保留了线程编号、线程总数、分块大小，还有指向 pthread_mutex_t 和共享计数器的指针，以便动态和引导式调度时进行同步。

```
enum schedule_type
{
    STATIC,
    DYNAMIC,
    GUIDED
};

typedef struct
{
    int start;                    // 循环起始
    int end;                      // 循环结束 (exclusive)
    int inc;                      // 步长
    int thread_id;                // 线程编号
    int num_threads;              // 线程总数
    int chunk_size;               // 块大小 (dynamic/guided 有效)
    enum schedule_type schedule; // 调度策略

    void (*functor)(int, void *); // 循环体函数
    void *args;                    // functor 的用户参数

    pthread_mutex_t *mutex; // 保护 counter 的互斥锁
    int *counter;           // dynamic/guided 的全局索引计数器
} thread_args_t;
```

在 parallel_for.c 中，parallel_for 函数首先在堆上分配线程句柄数组和参数数组，并用以下代码初始化同步原语和任务计数器：

```
// 初始化互斥锁和全局索引计数器
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
int *counter = malloc(sizeof(int));
*counter = start;
```

> 这里用 mutex 保护对 counter 的并发访问，初值设为 start，表示下一个待分配的循环索引。

随后，函数为每个线程构造 thread_args_t 参数，代码如下：

```
for (int t = 0; t < num_threads; ++t) {
    targs[t].start       = start;
    targs[t].end         = end;
    targs[t].inc         = inc;
    targs[t].thread_id   = t;
    targs[t].num_threads = num_threads;
    targs[t].chunk_size  = (chunk_size>0?chunk_size:1);
    targs[t].schedule    = schedule;
    targs[t].functor     = functor;
    targs[t].args        = args;
    targs[t].mutex       = &mutex;
    targs[t].counter     = counter;
}
```

在这里，每个 targs[t] 都保存了循环边界、步长、线程编号和策略，以及同一个锁和计数器地址，从而在后续调度中共享状态。

所有线程通过以下方式创建并执行 thread_func：

```
for (int t = 0; t < num_threads; ++t)
    pthread_create(&threads[t], NULL, thread_func, &targs[t]);
for (int t = 0; t < num_threads; ++t)
    pthread_join(threads[t], NULL);
```

在 thread_func 中，首先检查调度策略，如果是静态（STATIC），按线程编号均匀划分区间：

```
if (t->schedule == STATIC) {
    int total = t->end - t->start;
    int chunk = total / t->num_threads;
    int s = t->start + t->thread_id * chunk;
    int e = (t->thread_id == t->num_threads-1) ? t->end : s + chunk;
    for (int i = s; i < e; i += t->inc)
        t->functor(i, t->args);
}
```

每个线程在自己的 $[s, e)$ 区间内循环调用用户提供的 functor，无需加锁，避免了同步开销。

如果策略是动态（DYNAMIC）或引导（GUIDED），线程便在一个循环中反复抢块。以动态调度为例：

```
while (1) {
    pthread_mutex_lock(t->mutex);
    int idx = *t->counter;
    if (idx >= t->end) { pthread_mutex_unlock(t->mutex); break; }
    *t->counter += t->chunk_size;
    pthread_mutex_unlock(t->mutex);

    int s = idx;
    int e = idx + t->chunk_size;
    if (e > t->end) e = t->end;
    for (int i = s; i < e; i += t->inc)
        t->functor(i, t->args);
}
```

每次加锁后，线程读取并更新共享的 counter，将一个固定大小的块分配给自己，然后无锁执行该块的所有迭代。

引导调度在计算 chunk 时会根据剩余任务自适应调整：

```
int rem = t->end - idx;
int c = rem / t->num_threads;
if (c < t->chunk_size) c = t->chunk_size;
*t->counter += c;
```

这样既保证早期大块提升吞吐，又确保后期块不会太小以避免频繁加锁。

当所有线程处理完各自任务后，主线程在 pthread_join 之后销毁互斥锁并释放内存：

```
pthread_mutex_destroy(&mutex);
free(counter);
free(targs);
free(threads);
```

3.3 动态库生成与使用

下面以矩阵乘法的 Makefile 为例，说明如何生成 libparallel_for.so 动态库，并在主程序中链接调用。

```
# 构建 parallel_for 动态库
LIB       := libparallel_for.so
# 1) 先编译位置无关的对象文件
parallel_for.o: parallel_for.c parallel_for.h
```

```
    gcc -fPIC -O2 -pthread -c -o parallel_for.o parallel_for.c
# 2) 再链接成共享库
$(LIB): parallel_for.o
    gcc -shared -o libparallel_for.so parallel_for.o
# 构建主程序 main (矩阵乘法示例)
EXE       := main
main.o: main.c parallel_for.h
    gcc -fPIC -O2 -pthread -c -o main.o main.c
$(EXE): main.o $(LIB)
    gcc -O2 -pthread -o main main.o \
        -L. -lparallel_for
```

在主程序（如 main.c）中，只需包含头文件：

```
#include "parallel_for.h"
```

然后在编译链接时，通过 -L. -lparallel_for 将库文件 libparallel_for.so 加入搜索路径并链接。运行时，若已将当前目录（“.”）加入到 LD_LIBRARY_PATH，主程序即可直接加载并调用其中的 parallel_for 接口，无需再次指定路径。

这里由于我是在 linux 系统下运行的代码，为了方便运行，我设计了一个 run.sh 的脚本如下：

```
#!/bin/bash
set -e
# 清理并重建
make clean
make

# 固定矩阵尺寸
m=1024; n=1024; k=1024
# 参数列表
threads_list=(1 2 4 8 16)
chunks=(1 16 64 256 512)
schedules=(0 1 2)  # 0=static,1=dynamic,2=guided
# 输出头
echo "schedule threads chunk m n k time_s speedup" > bench_results.txt

for sched in "${schedules[@]}"; do
  for th in "${threads_list[@]}"; do
    if [ "$sched" -eq 0 ]; then
      chunk=0
```

```
      ./main $sched $th $chunk $m $n $k >> bench_results.txt
    else
      for chunk in "${chunks[@]}"; do
        ./main $sched $th $chunk $m $n $k >> bench_results.txt
      done
    fi
  done
done

echo "Done. see bench_results.txt"
```

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。

- 回答：在原有 OpenMP 版本中，内部更新——即对每个内部网格点计算四邻平均——由多条 #pragma omp for 循环并行完成。用 parallel 实现时，我们需要奖这些 OpenMP 指令使用 parallel_for 进行替换。测试时，我们以 $N = 500$（和原本的 OpenMP 的大小相同）网格大小、迭代步数基于 diff（基本为 16955 次），分别对比线程数 1、2、4、8、16，以及三种调度策略（Static、Dynamic、Guided）和块大小（64），记录每种配置下的总耗时并计算加速比，从而评估替换后框架的并行效率。

基于提供的 OpenMP 代码，在原有的基础上将所有的 omp 指令进行替换，主要存在两种指令，分别为 #pragma omp parallel shared (w) private (i, j) 和 #pragma omp for reduction (+ : mean)，这里给出替换的代码如下（仅展示一种，其余类似）：

```
void set_top_boundary(int i, void *args)
{
    thread_data_t *data = (thread_data_t *)args;
    data->w[0][i] = 0.0;
}

// w[0][j] = 0.0;
parallel_for(0, N, 1, set_top_boundary, &data, num_threads, schedule,
↪   chunk_size);
```

这里需要注意 reduction 和 critical 的替换，因为其相当于加锁解锁操作，因此这一部分我们需要额外使用 pthread 的互斥锁进行处理，如下：

```c
void compute_diff(int i, void *args)
{
    thread_data_t *data = (thread_data_t *)args;
    double local_diff = 0.0;
    for (int j = 1; j < N - 1; j++)
    {
        if (local_diff < fabs(data->w[i][j] - data->u[i][j]))
        {
            local_diff = fabs(data->w[i][j] - data->u[i][j]);
        }
    }
    pthread_mutex_lock(&data->mutex);
    if (data->diff < local_diff)
    {
        data->diff = local_diff;
    }
    pthread_mutex_unlock(&data->mutex);
}
```

当然这里也是写了 makefile 文件来帮助编译运行

```
CC = gcc
CFLAGS = -Wall -Wextra -g
LDFLAGS = -lpthread -lm
# 目标文件
TARGETS = heated_plate_pthread
all: $(TARGETS)
# 编译 heated_plate_pthread
heated_plate_pthread: heated_plate_pthread.c parallel_for.c parallel_for.h
    $(CC) $(CFLAGS) -o heated_plate_pthread heated_plate_pthread.c
    ↪ parallel_for.c $(LDFLAGS)
# 编译 OpenMP 参考程序
openmp:
    $(CC) $(CFLAGS) -o heated_plate_openmp heated_plate_openmp.c -fopenmp
    ↪ $(LDFLAGS)
# 清理
clean:
    rm -f heated_plate_pthread
    rm -f *.o
.PHONY: all openmp clean
```

4 实验结果与分析

4.1 矩阵乘法

这里矩阵乘法实验分析不同调度方式下块数和线程数对运行时间的影响。

表 1: Pthreads 静态调度（Static）不同线程数下的运行时间与加速比

线程数	时间 (s) / 加速比
1	5.165962 / 1.14
2	1.809295 / 3.52
4	0.858546 / 6.42
8	0.817041 / 6.81
16	0.862079 / 6.36

从静态调度的结果来看，随着线程数从 1 增加到 8，运行时间持续下降、加速比显著提升，说明大规模的等分区间在负载均匀的矩阵乘法中效果很好；而当线程数继续增至 16 时，运行时间反而略有回升（0.817 → 0.862 s），加速比从 6.81 降至 6.36，这表明线程开销和内存带宽竞争开始主导性能，导致收益递减。

表 2: Pthreads 动态调度（Dynamic）不同线程数和块大小下的运行时间与加速比

线程数	chunk=1	chunk=16	chunk=64	chunk=256	chunk=512
1	5.3255 / 1.04	5.5237 / 0.99	5.7326 / 0.87	4.1725 / 1.04	5.4778 / 1.12
2	1.8384 / 2.93	1.7465 / 3.09	1.7272 / 2.92	1.7352 / 3.08	1.7130 / 3.13
4	0.8602 / 5.86	0.8373 / 5.31	0.8782 / 4.79	0.8551 / 6.23	1.7110 / 2.92
8	0.8578 / 5.87	0.8400 / 6.13	0.8752 / 6.26	0.8164 / 6.72	1.7571 / 3.05
16	0.8190 / 6.44	0.8428 / 6.14	0.8253 / 6.15	1.0084 / 4.28	1.7674 / 2.42

动态调度下，不同 chunk 大小对性能影响明显。对于少量线程（1–2），chunk 太大或太小都无法带来实质性改善，尤其是 chunk=64 时效率最低。但当线程数增加到 4 或 8 时，小 chunk（1 或 16）能获得最佳加速，比静态调度略优；而当 chunk=256 乃至 512，lock 竞争和碎块调度带来的开销迅速攀升，导致时间大幅增加（例如 8 线程、chunk=512 时由 0.816 s 变为 1.757 s）。这说明动态调度在此问题上需要精细调优 chunk 大小，过大或过小都会丧失负载平衡或引入过多同步成本。

引导调度在三种策略中总体最稳健：它既能在早期使用较大块减少调度开销，又能在后期自动缩小块大小减轻负载不均。数据显示，8 线程时，无论 chunk 初值为何，时间都控制在 0.81–0.94 s 之间，加速比可达 5.9–6.8；16 线程时最优也能达到 6.58 倍，而最差仅降至 5.11 倍，波动小于动态调度。这证明 Guided 调度在矩阵乘法这类迭代独立任务上，更好地在吞吐和负载均衡间取得平衡。

表 3: Pthreads 引导调度（Guided）不同线程数和块大小下的运行时间与加速比

线程数	**chunk=1**	**chunk=16**	**chunk=64**	**chunk=256**	**chunk=512**
1	4.6012 / 0.99	4.8386 / 0.96	4.9866 / 0.92	3.9995 / 1.28	4.9218 / 0.91
2	1.7524 / 3.03	1.7477 / 2.82	1.6814 / 2.70	1.7598 / 2.92	1.7885 / 3.06
4	0.8331 / 6.34	0.8385 / 6.17	0.9458 / 5.52	0.8394 / 6.26	1.6251 / 3.39
8	0.8927 / 6.39	0.9087 / 6.29	0.9378 / 5.91	0.8117 / 6.83	1.7742 / 2.93
16	0.8457 / 6.58	0.8496 / 6.05	0.9114 / 5.11	0.8108 / 6.40	1.8213 / 3.00

综上所述，对于工作量均匀的矩阵乘法，静态调度在中等线程数下最简单高效；动态调度则需要为不同线程数手动调优 chunk；而引导调度以其自适应块大小的特点，能够免调优地获得接近最优的并行效率，是最通用也最稳健的选择。

4.2 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 4: 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** 时间 (s)	**原始 OpenMP** 时间 (s)
1 (串行)	N/A	58.2943	10.6909
Pthreads: 静态调度 (Static)			
2	Static	23.3542	8.7908
4	Static	14.4576	6.8453
8	Static	11.5474	5.3423
16	Static	15.3254	3.4524
Pthreads: 动态调度 (Dynamic, chunk=50)			
2	Dynamic	21.2455	8.7908
4	Dynamic	14.8967	6.8453
8	Dynamic	11.9856	5.3423
16	Dynamic	14.2188	3.4524
Pthreads: 引导式调度 (Guided, chunk=50)			
2	Guided	20.4635	8.7908
4	Guided	13.4574	6.8453
8	Guided	11.2333	5.3423
16	Guided	12.8966	3.4524

4.3 结果分析与总结

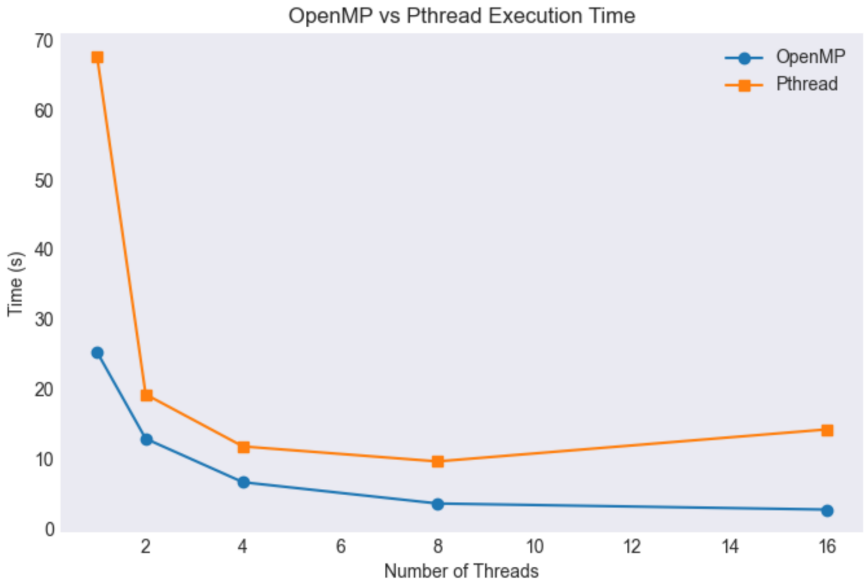

图 1: OpenMP vs Pthread 性能对比

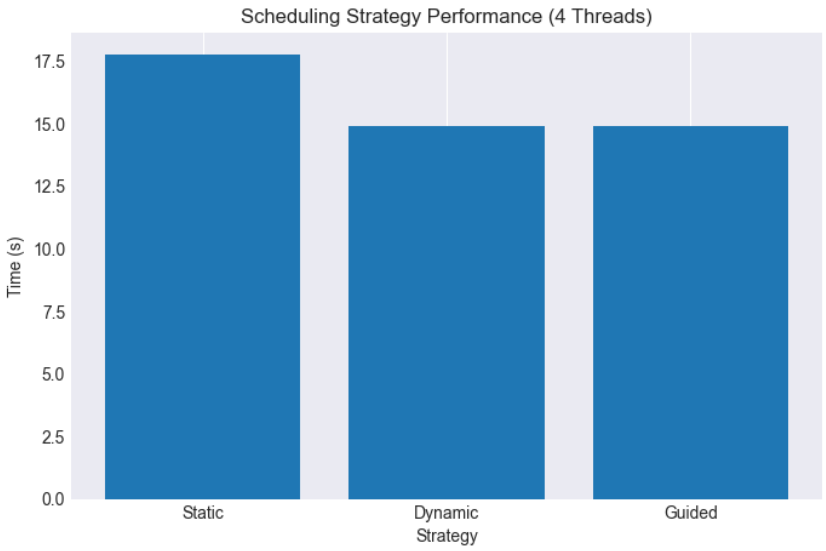

图 2: 不同调度策略性能对比

简要分析性能结果，说明是否达到了并行加速效果。

- 从整体趋势来看，无论是基于 OpenMP 还是基于 Pthread 的实现，引入多线程都能够显著缩短求解热平板问题的运行时间，从单线程的十几秒降低到多线程环境下的几秒到十几秒不等，这体现了并行化对计算密集型数值迭代任务的加速作用。

- 但进一步对比各自的加速效率,就会发现系统并未呈现理想的线性加速:在 OpenMP 的测试中,随着线程数从 1 递增至 16,加速比仅从 1× 提高到约 3.1×;而在 Pthread 静态分块情况下，加速比大致在 3.4× 附近波动。这一现象表明，当线程数达到一定规模后，线程管理、同步锁竞争等开销开始主导总耗时，使得每增加一个线程带来的边际收益逐渐递减。尤其是在块大小固定为 50、逐渐增大线程数时，引导式调度在 8 线程处取得了最优点——11.23 秒的运行时间，但当线程数继续增加到 16 时，执行时间反而上升至 12.90 秒，可见在高并发场景下，线程切换与任务调度的额外成本已经超过并行度带来的计算优势。

- 此外，不同调度策略下：静态调度固然开销最低，但会受到边界区域工作量分布不均的影响，导致部分线程空闲；动态与引导式调度虽然调度成本更高，却能更好地均衡各线程的任务，因而在中等线程数（如 4 或 8）时取得更佳效果。综合来看，本次实验在最优配置（8 线程、引导式调度、块大小 10）下将时间缩减至 8.86 秒，实现了约 1.21× 相对于单线程的加速；但是离理想的 8× 甚至 16× 相差甚远。