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 李源卿

学 号 22336128

学 院 计算机学院

专 业 计算机科学与技术

2025 年 5 月 10 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。

- 编写环境: 本地 WSL2；运行环境：本地 WSL2；gcc version 13.3.0；
- POSIX Threads 支持: 200809 ； POSIX 标准版本: 200809

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。我的矩阵乘法版本的 functor 和热传导版本的 functor 在参数上不同（按道理应该尽量保持同一个形式，但是我想探索下函数指针的用法，所以就写作了不同的形式，但其他的基本一致）。
此处以矩阵乘法为例：
我在 parallel_for 函数中创建和销毁线程 thread_func，在这里执行拆解后的第一层 for 循环：

Listing 1: parallel_for 函数核心代码片段

```
1
2 static void *thread_func(void *arg) {
3     ThreadArgs *args = (ThreadArgs *)arg;
4     int current = args->start_idx;
5     //第一层for循环
```

```
    for (int t = 0; t < args->num_iters; t++) {
        //第一层for循环中要做的
        args->functor(current, args->arg);
        current += args->inc;
    }
    return NULL;
}
```

值得注意的点在于，由于每轮循环都会重新调用 functor，所以 functor 内部的变量都是暂时的（我当时写晕了用函数内部的变量来存储 local_max_diff）

下面说一下 parallel_for 的划分 for 循环的方式，此处只考虑了 for(int i = st; i < ed; i+=inc) 的 for 循环，判定条件是 i<=ed 的情况只需要在计算 st 与 ed 之间的距离时加一即可。

这里的划分参考了大模型给出的写法，首先上取整算出总共需要迭代的次数，然后把迭代次数分给各个线程，然后计算 for 循环起始位置，记录函数指针，步长等。

Listing 2: parallel_for 函数核心代码片段

```
if (start >= end || inc <= 0 || num_threads <= 0) return;

    // 计算总迭代次数
    int total = ((end - start) + inc - 1) / inc;
    if (total <= 0) return;

    // 限制线程数不超过总迭代次数
    if (num_threads > total) num_threads = total;

    int base = total / num_threads;
    int remainder = total % num_threads;

    pthread_t *threads = malloc(num_threads * sizeof(pthread_t));
    ThreadArgs *args = malloc(num_threads * sizeof(ThreadArgs));

    for (int k = 0; k < num_threads; k++) {
        // m是迭代次数
        int m = base + (k < remainder ? 1 : 0);
        //用于计算for循环起始位置
        int sum_prev = base * k + (k < remainder ? k : remainder);
        args[k].start_idx = start + sum_prev * inc;
        args[k].num_iters = m;
        args[k].inc = inc;
```

```
        args[k].functor = functor;
        args[k].arg = arg;

        pthread_create(&threads[k], NULL, thread_func, &args[k]);
    }
```

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。

在 Makefile 中，生成动态链接库（.so 文件）的核心规则如下：

```
$(LIBRARY): parallel_for.o
    $(CC) -shared -o $@ $^
```

- `-shared`: 告诉编译器生成共享库（动态库）。
- `o $@`: 输出文件名为目标名（即 $(LIBRARY)，如 libparallel_for.so）。
- `$^`: 依赖的所有.o 文件（这里是 parallel_for.o）。

主程序通过以下规则链接动态库：

```
$(TARGET): main.o
    $(CC) -o $@ $< -L. -lparallel_for -fopenmp -Wl,-rpath=.
```

- `-L.`: 指定库文件的搜索路径为当前目录（.）。
- `-lparallel_for`: 链接名为 libparallel_for.so 的库（省略 lib 前缀和.so 后缀）。
- `-Wl,-rpath=.`: 运行时动态库搜索路径为当前目录（避免 LD_LIBRARY_PATH 环境变量设置）。(没有这个会找不到.so 文件)

```
%.o: %.c
        $(CC) $(CFLAGS) -c $<
```

将所有.c 结尾的文件编译为.o 结尾的文件。

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。

3.5 reduce 的实现

我还写了 parallel_sum 和 parallel_max_diff 来模拟 reduce 的行为，把结果存储在线程 id 对应的数组位置处，最后汇总。

```
double parallel_sum(int start, int end, int inc,
                    void (*functor)(int, int,void*),
                    int num_threads, SumData *data) {
    // 分配局部和数组（第一个元素存储线程数）
    data->local_sums = calloc(num_threads, sizeof(double));
    data->thread_count = num_threads;

    // 执行并行计算
    parallel_for(start, end, inc, functor, data, num_threads);

    // 汇总结果
    double total = 0.0;
    for (int i = 0; i < num_threads; i++) {
        // printf("%lf ",data->local_sums[i]);
        total += data->local_sums[i];
    }
    // printf("\n");
    free(data->local_sums);
    return total;
}
```

然后其余的地方和矩阵乘法类似，定义多个结构体作为多个函数的参数，然后传递到 parrallel_for。

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: 矩阵乘法问题性能对比 (Pthreads vs OpenMP, 矩阵大小: 1024 x 1024)

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	9.6565	9.5752
Pthreads: 静态调度 (Static)			
2	Static	5.0383	5.3081
4	Static	2.6210	2.4506
8	Static	3.1723	3.0804
16	Static	3.5882	3.3858

表 2: 热传导问题性能对比 (Pthreads vs OpenMP, 网格大小: 500 x 500, 误差 0.005)

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	11.853832	9.991275
Pthreads: 静态调度 (Static)			
2	Static	6.240613	5.823293
4	Static	3.641210	3.150620
8	Static	5.005029	4.239361
16	Static	8.068534	6.936898

4.2 结果分析与总结

4.3 矩阵乘法

- 矩阵乘法的并行性能明显优于热传导问题，因为热传导需要多次创建/销毁线程，而矩阵乘法只需单次线程操作
- 选择 1024×1024 矩阵尺寸既能体现并行性能，又保持合理实验时间
- 我的 Pthreads 实现性能略低于 OpenMP（差距约 5-15
- 所有并行结果均通过串行代码验证，在 1/2/4/8/16 线程下结果正确
- 矩阵乘法展示了更好的可扩展性，在 4 线程时达到最佳加速比（Pthreads 3.68x，OpenMP 3.91x）
- 当线程数超过 4 时，两种问题的加速比均下降，这是物理核数仅为 4 导致的（前面多次实验都提到了）

表 3: 矩阵乘法性能对比 (Pthreads vs OpenMP, 矩阵尺寸: 1024×1024)

线程数	调度方式	Pthreads(s)	加速比	OpenMP(s)	加速比
1 (串行)	N/A	9.6565	1.00x	9.5752	1.00x
静态调度 (Static)					
2	Static	5.0383	1.92x	5.3081	1.80x
4	Static	2.6210	3.68x	2.4506	3.91x
8	Static	3.1723	3.04x	3.0804	3.11x
16	Static	3.5882	2.69x	3.3858	2.83x

4.4 热传导

- 最佳加速比出现在 4 线程时（Pthreads 2.79x，OpenMP 3.17x）
- 相比理想线性加速（4x），实际效率仅达 69.7%（Pthreads）和 79.3%（OpenMP）, 这里不像矩阵乘法那样，我的程序可以和 openmp 的实现拥有相似的并行性能。这是因为我的 parallel_for 每次调用都伴随着线程的销毁的创建，而 openmp 可以先使用 #pragma omp parallel 创建，然后使用 #pragma omp for 调用，这里会存在很多开销（特别是在 while 循环里）。这里也有解决方案，就是使用线程池，但是由于时间原因，我就只实现了基础版本。
- 当线程数超过 4 时，两种问题的加速比均下降，这是物理核数仅为 4 导致的（前面多次实验都提到了）。我们可以发现，当线程数增多时，我的 Pthread 版本和 Openmp 版本的差距变得很明显，我觉得这也是线程的频繁删除和创建导致的。

表 4: 热传导性能对比 (Pthreads vs OpenMP, 网格尺寸: 500×500)

线程数	调度方式	Pthreads(s)	加速比	OpenMP(s)	加速比
1 (串行)	N/A	10.153832	1.00x	9.991275	1.00x
静态调度 (Static)					
2	Static	6.240613	1.63x	5.823293	1.72x
4	Static	3.6412	2.79x	3.1506	3.17x
8	Static	5.005029	2.03x	4.239361	2.36x
16	Static	8.068534	1.26x	6.936898	1.44x

收敛误差: 0.005