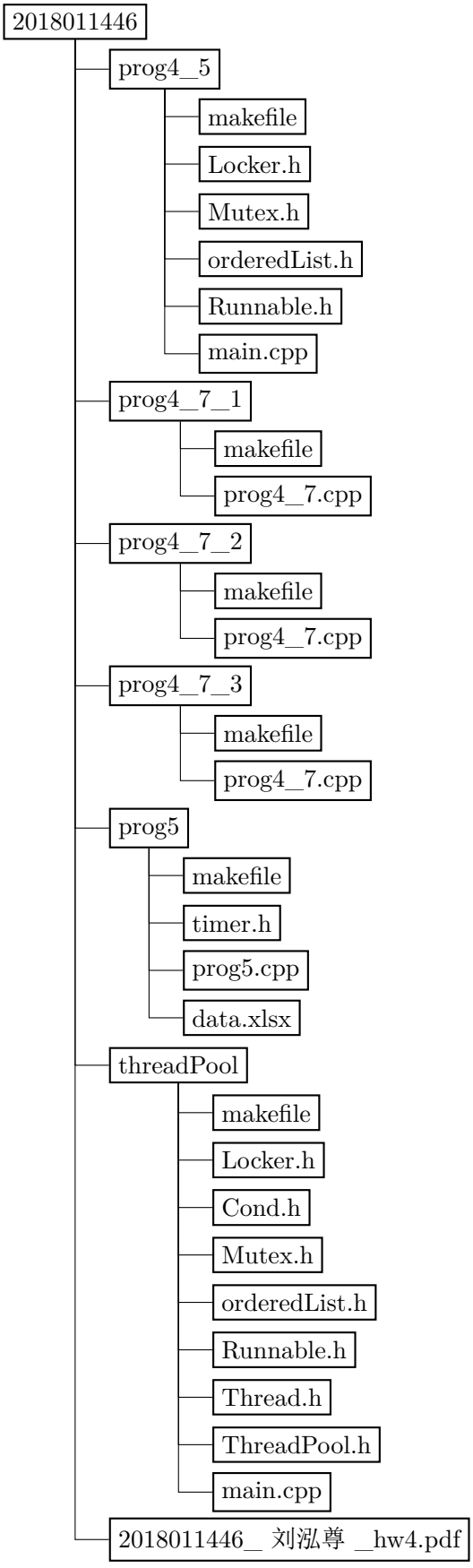
高性能计算导论:hw4

刘泓尊　2018011446　计 84

2020 年 3 月 31 日

目录

1 File Structure



2 Exercise 4.7

2.1 一个生产者 + 一个消费者

因为只有两个线程，且每个线程责任清晰，所以我将生产者和消费者分别放在一个函数中，让结构更清晰。

生产者首先获得互斥量，然后发送信息到缓冲区，随后将标志位 (ok) 置为 1; 消费者采用忙等待 (busy-waiting) 机制，检测到标志位为 1 后，读取消息缓冲区中的内容，此过程中也采用互斥锁.(实际上，本例中“消费者”加锁的意义不大，除非编译器或 CPU 对指令进行了重排，也就是“内存序”改变的问题。)

此部分代码在./prog4_7_1 下

共享变量

```
const int MAX_LENGTH = 100;
pthread_mutex_t mu;//互斥量
bool ok;//标志位
char msg[MAX_LENGTH];//消息缓冲区
```

Consumer

```
//消费者
void* consumer_func(void* rank){
    long my_rank = (long)rank;
    while(1){
        pthread_mutex_lock(&mu);//加锁
        if(ok){
            printf("message: %s\n", msg);
            pthread_mutex_unlock(&mu);
            break;
        }
        pthread_mutex_unlock(&mu);
    }
    return nullptr;
}
```

Producer

```
//生产者
void* producer_func(void* rank){
    long my_rank = (long)rank;
    pthread_mutex_lock(&mu);//加锁
    sprintf(msg, "Hello from thread %ld!", my_rank);
    ok = 1;
    pthread_mutex_unlock(&mu);
    return nullptr;
```

```
}
```

main

```
pthread_t consumer, producer;
pthread_mutex_init(&mu, nullptr);

pthread_create(&consumer, nullptr, ::consumer_func, (void*)0);
pthread_create(&producer, nullptr, ::producer_func, (void*)1);
pthread_join(consumer, nullptr);
pthread_join(producer, nullptr);

pthread_mutex_destroy(&mu);
```

运行结果 执行 make run 之后可以运行程序，可以看到程序实现了“生产者-消费者”同步。

main

```
[2018011446@bootstraper prog4_7_1]$ make run
./prod_cons1
message in thread 0: Hello from thread 1!
```

2.2 偶数生产者 + 奇数消费者

我将生产者和消费者写在同一函数中，便于区分奇数与偶数。共享变量设置与第一部分相同。

对于每一个线程，采用忙等待 (busy-waiting) 机制，首先获得互斥量: 如果是奇数线程，检测标志位 (ok) 是否为 1,$ok == 1$ 则读取缓冲区中的内容, 并**将标志位 (ok) 置为 0**，以便后续线程写入缓冲区；如果是偶数线程，在标志位为 0 的情况下，将消息写入缓冲区，并将标志位置为 1。消费者和生产者完成操作后均要释放锁。

代码放在./prog4_7_2 下

Producer&Consumer

```
void* run(void* rank){
    long my_rank = (long)rank;
    while(1){
        pthread_mutex_lock(&mu);//加锁
        if(my_rank % 2 == 1){//奇数消费者
            if(ok){
                printf("message received by %ld: %s\n", my_rank, msg);
                ok = 0;
                pthread_mutex_unlock(&mu);
                break;
            }
        }else{//偶数生产者
```

```
            if(!ok){
                sprintf(msg, "Hello from thread %ld", my_rank);
                ok = 1;
                pthread_mutex_unlock(&mu);
                break;
            }
        }
        pthread_mutex_unlock(&mu);
    }
    return nullptr;
}
```

main

```
pthread_mutex_init(&mu, nullptr);
pthread_t* threads = new pthread_t[threadnum];
for(size_t i = 0; i < threadnum; i++){
    pthread_create(&threads[i], nullptr, ::run, (void*)i);
}
for(size_t i = 0; i < threadnum; i++){
    pthread_join(threads[i], nullptr);
}
pthread_mutex_destroy(&mu);
```

运行结果 执行 make run n=10 后得到的结果如下 (n 是线程个数, n 必须为偶数!):

main

```
[2018011446@bootstraper prog4_7_2]$ make run n=10
./prod_cons2 10
create 10 threads...
message received by 3: Hello from thread 0
message received by 1: Hello from thread 4
message received by 7: Hello from thread 2
message received by 9: Hello from thread 6
message received by 5: Hello from thread 8
```

2.3 每个线程既是生产者又是消费者

采用共享变量 *recv* 标记应接收消息的消费者。

依然采用忙等待机制，每个线程初始时不设置角色。线程获得互斥锁后，当发现标志位 (ok) 为 0 时，成为生产者，发送消息并将标志位置为 1，指定消息接收方 (recv) 为 $(my_rank+1)\%num_threads$。当有线程发现标志位为 1 且自己是接受方 (recv) 时，成为消费者，读取缓冲区并将标志位置 0。

代码放在./prog4_7_3 下

共享变量

```
const int MAX_LENGTH = 100;
bool ok;//标志位
char msg[MAX_LENGTH];//消息缓冲区
pthread_mutex_t mu;//互斥量
int recv;//下一个消费者的编号
int num;//线程个数
```

Producer&Consumer

```
void* run(void* rank){
    long my_rank = (long)rank;
    bool is_recv = false, is_send = false;
    while(!is_send || !is_recv){
        pthread_mutex_lock(&mu);
        if(ok){
            if(!is_recv && my_rank == recv){
                printf("receiver %ld: %s\n", my_rank, msg);
                ok = false;
                is_recv = true;
            }
        } else if (!is_send){
            sprintf(msg, "Hello from thread %ld", my_rank);
            ok = true;
            is_send = true;
            recv = (my_rank + 1) % num;
        }
        pthread_mutex_unlock(&mu);
    }
    return nullptr;
}
```

main

```
pthread_mutex_init(&mu, nullptr);
pthread_t* threads = new pthread_t[num];
for(size_t i = 0; i < num; i++){
    pthread_create(&threads[i], nullptr, ::run, (void*)i);
}
for(size_t i = 0; i < num; i++){
    pthread_join(threads[i], nullptr);
}
pthread_mutex_destroy(&mu);
```

运行结果 执行 make run n=10 的结果如下 (此题 n 任意):

main

```
[2018011446@bootstraper prog4_7_3]$ make run
./prod_cons3 10
created 10 threads
receiver 1: Hello from thread 0
receiver 2: Hello from thread 1
receiver 9: Hello from thread 8
receiver 5: Hello from thread 4
receiver 8: Hello from thread 7
receiver 7: Hello from thread 6
receiver 0: Hello from thread 9
receiver 4: Hello from thread 3
receiver 3: Hello from thread 2
receiver 6: Hello from thread 5
```

3 Exercise 4.11

注: 下面所说的 *prev* 不代表是双向链表，只是代表位置关系在“前”

3.1 两个 Delete 同时执行

如果两个线程试图删除同一个节点，执行顺序可能如下:

- Thread1 找到节点 N
- Thread2 找到节点 N
- Thread1 将 N→prev→next 赋值为 N→next
- Thread1 释放节点 N
- Thread2 将 N→prev→next 赋值为 N→next。此时访问了已经释放的内存，发生**段错误**.

如果两个线程释放前后相邻的节点，可能的执行顺序如下:

- Thread1 找到节点 N1
- Thread2 找到节点 N2，且 N2 紧邻在 N1 后面
- Thread1 将 N1→prev→next 赋值为 N1→next
- Thread2 将 N2→prev→next(即 N1→next) 赋值为 N2→next.

此时 N1→prev 的后继为 N2, 而不是 N2→next.

- Thread1 删除节点 N1
- Thread2 删除节点 N2

此时**链表发生断裂**。

3.2 一个 Insert 和一个 Delete 同时执行

插入的线程恰好插入在待删除点的前面:

- Thread1 找到待删除节点 N1
- Thread2 创建新节点 N2, 找到插入位置，在 N1 的前面

• Thread2 将 N2 的后继标记为 N1
• Thread1 删除 N1 节点
此时发生链表**断裂**, 节点 N2 之后的部分将丢失。

3.3 一个 Member 和一个 Delete 同时执行

Member 和 Delete 是同一节点时:
• Thread1 调用 Member 访问节点 N1，发现 N1 存在
• Thread2 删除 N1
• Thread1 函数返回报告 N1 存在，实际上却已经被删除.

Member 的节点在 Delete 之后:
• Thread1 查找 (Memver) 节点 N1, 当前正在节点 N2(在 N1 之前)
• Thread2 删除节点 N2
• Thread1 调用 cur = N1→next, 访问已经释放的内存，发生**段错误**。

3.4 两个 Insert 同时执行

两个线程 insert 在同一位置:
• Thread1 找到插入位置的前一个节点 N1
• Thread2 找到插入位置的前一个节点 N1
• Thread1 插入: N1→next = newNode1
• Thread2 插入: N1→next = newNode2
此时相当于只成功插入了一个节点，而不是两个。

3.5 一个 Insert 和一个 Member 同时执行

Insert 线程插入节点恰好是 Member 查询的.
• Thread1 查找节点 N1, 未找到
• Thread2 插入节点 N1
• Thread1 调用返回，报告 N1 不存在，而实际上已经在链表中。

4 Exercise 4.12

不安全
• 两个线程删除同一节点时:
第一阶段: 由于读写锁可以同时读，所以两个线程都将找到该节点.
第二节点: 线程 1 加“写锁”，删除该节点，释放锁；之后线程 2 获得“写锁”再删除该节点。出现**重复释放**的问题。
• 两个线程同时插入时，也会与进行上例相同的第一阶段，因此在第二阶段可能在同一位置插入两个节点。

两种情况的**本质**都是: 在两次加锁的中间存在时间差，这段时间内，链表的状态可能发生改变。因此不安全。

5 Programming Assignment 4.5

代码在./prog4_5 下

5.1 线程安全的有序链表类

为了便于管理锁的获取与释放，我简单封装了互斥量与信号量，并实现了 Locker 来自动管理锁的释放，见附录 A。之后我实现了线程安全的有序链表 orderedList，为了简洁性，内部数据结构使用 STL，代码如下:

orderedList.h

```
template <class T>
class orderedList{
    list<T> alist;
    ReadWriteMutex rwmu;
public:
    void push(T t){
        WriteLocker locker(&rwmu);//写锁
        alist.insert(std::upper_bound(alist.begin(), alist.end(), t), t);
    }
    bool pop(T t){
        WriteLocker locker(&rwmu);//写锁
        auto iter = std::find(alist.begin(), alist.end(), t);
        if(iter != alist.end()){
            alist.erase(iter);
            return true;
        } else {
            return false;
        }
    }
    bool find(T t){
        ReadLocker locker(&rwmu);//读锁
        auto iter = std::find(alist.begin(), alist.end(), t);
        if(iter != alist.end()){
            return true;
        } else {
            return false;
        }
    }
    string toString(){
        string res = "head";
        ReadLocker locker(&rwmu);//读锁
        for(auto iter: alist)
            res += "->" + std::to_string(iter);
        res += "\n";
        return res;
```

```
    }
};
```

5.2 线程池的实现

线程池的实现主要是填充了题给的框架。先使用 pthread_create() 创建给定数量的线程。然后生成一定数量的任务，压入任务队列，同时用信号量唤醒线程。当任务产生完毕的时候，设置标志位 finished = true, 之后进行广播唤醒所有线程。

main.cpp: main()

```
//Create Thread Pool
pthread_t* thread_handles = new pthread_t[thread_num];

/* Initialize mutexes and conditional variables */
pthread_mutex_init(&mu, nullptr);
pthread_cond_init(&cond, nullptr);
finished = false;

/* Start threads */
for(int i = 0; i < thread_num; i++){
    pthread_create(&thread_handles[i], nullptr, ::runtask, nullptr);
}

/* Generate Tasks */
for(int i = 0; i < task_num; i++){
    pthread_mutex_lock(&mu);
    taskQueue.push( Task( rand() % total, rand() % task_num) );//加入随机任务
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mu);
}


/* Wait fot threads to compelete */
finished = true;
pthread_cond_broadcast(&cond);
for(int i = 0; i < thread_num; i++){
    pthread_join(thread_handles[i], nullptr);
}

/* Destroy mutex and conditional variables */
pthread_cond_destroy(&cond);
pthread_mutex_destroy(&mu);
delete[] thread_handles;
```

线程执行的函数 runTask(), 使用类似忙等待的机制，一直循环直至有信号唤醒该线程，之后该线程执行随机操作的任务。具体代码如下:

runTask()

```
//线程函数
void* runtask(void* args){
    while(1){
        pthread_mutex_lock(&mu);
        while(taskQueue.empty() && !finished){//线程等待被唤醒
            while( pthread_cond_wait(&cond, &mu) != 0)
                ;
        }
        if(taskQueue.empty()){
            pthread_mutex_unlock(&mu);
            break;
        }
        Task t = taskQueue.front();
        taskQueue.pop();
        pthread_mutex_unlock(&mu);
        run(t);//执行对链表的随机操作
    }
    return nullptr;
}
```

为了减少耦合，对链表的随机操作写在 run() 函数中，比例为“插入: 删除: 查询: 打印 = 4:2:8:1”。为了便于显示多线程效果，每次执行完该函数后 **sleep 1s**，并在每次操作后**输出线程编号和任务执行结果**。代码如下:

run()

```
//定义操作比例
constexpr int findRate = 8;
constexpr int deleteRate = 2;
constexpr int insertRate = 4;
constexpr int printRate = 1;
constexpr int total = findRate + deleteRate + insertRate + printRate;
//随机调用
void run(const Task& task){
    if (task.op < findRate) {//find for 8/15 [0, 7)
        if(mylist.find(task.value)){
            printf("threadid %u, find value %d success\n", (unsigned int)
                pthread_self(), task.value);
        } else {
            printf("threadid %u, find value %d failed\n", (unsigned int)
                pthread_self(), task.value);
        }
```

```
15      } else if (findRate <= task.op && task.op < findRate + deleteRate) {//
            delete for 2/15 [7, 9)
16          if(mylist.pop(task.value)){
17              printf("threadid %u, delete value %d success\n", (unsigned int)
                    pthread_self(), task.value);
18          } else {
19              printf("threadid %u, delete value %d failed\n", (unsigned int)
                    pthread_self(), task.value);
20          }
21      } else if (findRate + deleteRate <= task.op && task.op < total -
            printRate) {//insert for 4/15 [9, 14)
22          mylist.push(task.value);
23          printf("threadid %u, inserted value %d\n", (unsigned int)
                pthread_self(), task.value);
24      } else { // print for 1/15 [14, 15)
25          std::cout << mylist.toString();
26      }
27      sleep(1);//sleep for 1 sec
28  }
```

5.3 实验结果

执行 make run t=2 n=20 后的输出结果如下，可以看到任务可以分配给 2 个线程。由于每次执行后 sleep(1)，加上线程个数的限制，形成了周期性输出的表现，每隔约 1s 则较快地输出 2 个。从多次测试的结果来看，并行程序是正确的。

输出结果

```
1  [2018011446@bootstraper threadPoolNor]$ make run t=2 n=20
2  ./main 2 20
3  threadid 1075492608, delete value 5 failed
4  threadid 1067099904, delete value 1 failed
5  threadid 1075492608, delete value 13 failed
6  threadid 1067099904, find value 6 failed
7  threadid 1075492608, inserted value 19
8  threadid 1067099904, find value 4 failed
9  threadid 1075492608, inserted value 16
10 threadid 1067099904, find value 1 failed
11 threadid 1075492608, inserted value 4
12 threadid 1067099904, delete value 11 failed
13 threadid 1075492608, find value 1 failed
14 threadid 1067099904, find value 18 failed
15 head->4->16->19
16 threadid 1067099904, delete value 17 failed
17 threadid 1075492608, find value 12 failed
18 threadid 1067099904, find value 18 failed
```

```
threadid 1075492608, find value 17 failed
threadid 1067099904, find value 12 failed
threadid 1075492608, find value 15 failed
head->4->16->19
```

为了后续编程方便，我还对线程 (Thread) 和线程池 (ThreadPool) 进行了封装，并重写了本题，极大地做到了代码的简化，具体内容请看附录 A。

6 Pthread Programming 5

代码在./prog5 下

6.1 派生线程计算 fib(n-1)

我实现了两个版本的 fib 函数，其中一个只是普通递归求值, 复杂度是 $O(2^n)$; 第二个版本使用记忆化方法，将复杂度为 $O(n)$.

由于创建进程会占用大量时间，在串行递归时，我设置在 $fib(n), n < 15$ 的时候使用串行函数计算、返回，而不再创建新线程。实际上，在求解较大 n 时 (比如 $n > 40$)，由于搜索树过深且计算节点的创建是指数级增长的，所以到达 $n = 15$ 的时候也不会再创建新线程。因此，此做法可以认为是合理的。

此外，我将 fib(n) 的最大 n 设为 60，因为 $n > 60$ 的串行版本将消耗大量的计算时间.

6.1.1 普通递归 (并行)

基于串行版本的递归做了修改。核心是设置一个共享变量 cur_thread_num 计算当前线程数，当当前线程数 < 最大线程数的时候，创建一个新线程计算 fib(n-2), 本线程计算 fib(n-1). 这里考虑到了均衡，因为 fib(n-1) 的计算时间较大，所以将计算时间短的 fib(n-2) 交给新线程去做，以平衡创建线程的开销。

fib_para(n)

```
void* fib_para(void* n){
    long my_n = (long)n;
    if(my_n < 15){
        return (void*)fib_serial(my_n);
    }
#   ifdef DEBUG
    printf("thread %d: fib(%ld)\n", (unsigned int)pthread_self(), my_n);
#   endif
    bool can_create = false;
    pthread_mutex_lock(&mu);
    if(++cur_thread_num < total_thread){//判断是否可以派生新线程
        can_create = true;
    }
    pthread_mutex_unlock(&mu);
```

```
    if(can_create){
        pthread_t new_thread;
        long firstres, secondres;
        //create a new thread to calculate fib(n-2)
        pthread_create(&new_thread, nullptr, fib_para, (void*)(my_n-2));
        //this thread to calculate fib(n-1)
        firstres = (long)fib_para((void*)(my_n-1));
        pthread_join(new_thread, (void**)&secondres);
        pthread_mutex_lock(&mu);
        cur_thread_num--;//修改共享变量
        pthread_mutex_unlock(&mu);
        return (void*)(firstres + secondres);
    } else {
        return (void*)( (long)fib_para( (void*)(my_n-1) ) + (long)fib_para(
            (void*)(my_n-2) ) );
    }
}
```

6.1.2 记忆化递归 (并行)

针对上述并行递归版本做了记忆化，改动的部分并不多，只是加了一个记忆数组 dp[].

fib_para_fast(n)

```
long fib[50] = {0};//共享变量, 记录dp状态
bool finished[50] = {0};//记录是否被计算
void* fib_para_dp(void* n){
    long my_n = (long)n;
    if(finished[my_n]){
        return fib[my_n];
    }
    if(my_n < 15){
        return (void*)fib_serial(my_n);
    }
#   ifdef DEBUG
    printf("thread %d: fib(%ld)\n", (unsigned int)pthread_self(), my_n);
#   endif
    bool can_create = false;
    pthread_mutex_lock(&mu);
    if(++cur_thread_num < total_thread){
        can_create = true;
    }
    pthread_mutex_unlock(&mu);
    if(can_create){
        pthread_t new_thread;
        long firstres, secondres;
```

```
23          //create a new thread to calculate fib(n-2)
24          pthread_create(&new_thread, nullptr, fib_para_dp, (void*)(my_n-2));
25          //this thread to calculate fib(n-1)
26          firstres = (long)fib_para_dp((void*)(my_n-1));
27          pthread_join(new_thread, (void**)&secondres);
28          pthread_mutex_lock(&mu);
29          cur_thread_num--;
30          fib[my_n] = firstres + secondres;//共享变量dp[]
31          finished[my_n] = true;//共享变量finished[]
32          pthread_mutex_unlock(&mu);
33          return (void*)(fib[my_n]);
34      } else {
35          long result = (long)fib_para_dp( (void*)(my_n-1) ) + (long)
                fib_para_dp( (void*)(my_n-2) );
36          pthread_mutex_lock(&mu);
37          fib[my_n] = result;//共享变量dp[]
38          finished[my_n] = true;//共享变量finished[]
39          pthread_mutex_unlock(&mu);
40          return (void*)( result );
41      }
42  }
```

6.1.3 串行部分的优化

实际上，计算 fib(n) 可以不采用递归函数，使用循环迭代完全可以实现 O(n) 的复杂度，并且只有 O(1) 空间。此策略可以极大减少创建线程和递归调用的开销。其串行版本如下:

fib_serial_fast(n)

```
1  long fib_serial_fast(int n){
2      long a = 0, b = 1, c;
3      for(int i = 1; i < n; i++){
4          c = b; b = a + b; a = c;
5      }
6      return b;
7  }
```

将此优化加入到并行版本的 fib(n) 求解中，可以实现较大改进，性能测试见 6.2 部分。

6.2 运行结果

执行 make run 命令后，依次输入线程个数和 n，之后分别计算串行版本和以上三个版本的计算时间. 示例如下:

运行示例

```
1  [2018011446@bootstraper prog2]$ make run
2  ./prog2
```

```
Enter thread_num:
8
Enter n of fib(n):
46
para_normal: fib(n) = 1836311903, time: 1.353691s
para_fast:   fib(n) = 1836311903, time: 0.210368s
para_dp:     fib(n) = 1836311903, time: 0.000023s
serial:      fib(n) = 1836311903, time: 2.867669s
```

从多次输出结果来看, 并行函数的正确性是得到验证的.

下面对不同 threadnum 和 n 值的性能进行统计:

p\n	10	20	25	30	35	40	45	50	55
串行	0.000001	0.000036	0.000339	0.003685	0.034168	0.22032	1.840282	19.100321	218.093399
1	0.000199	0.000251	0.00052	0.003439	0.032831	0.205708	1.653113	17.722985	196.093399
2	0.000202	0.000257	0.000536	0.003444	0.032849	0.205942	1.653773	17.792085	195.970721
4	0.000224	0.000373	0.000452	0.001664	0.014966	0.115644	0.778252	7.663938	85.838119
8	0.000221	0.002438	0.003865	0.002151	0.014333	0.109306	0.754843	8.905228	93.74916
16	0.000224	0.006023	0.007535	0.006285	0.015639	0.106378	0.802445	9.339412	99.025345
24	0.000222	0.007819	0.011001	0.012939	0.017517	0.103107	0.779098	8.518038	89.329015

表 1: 计算时间/s(普通递归并行版本)

p\n	10	20	25	30	35	40	45	50	55
串行	0.000001	0.000036	0.000339	0.003685	0.034168	0.22032	1.840282	19.100321	218.093399
1	0.00008	0.000087	0.000062	0.000257	0.001684	0.008121	0.08914	0.989676	11.026279
2	0.000066	0.000073	0.000072	0.000311	0.001682	0.008167	0.089213	0.989144	10.566794
4	0.000053	0.000072	0.000072	0.00026	0.001909	0.011863	0.089168	1.033208	10.468103
8	0.000053	0.0000054	0.000106	0.00023	0.00119	0.012483	0.15095	1.055574	10.605881
16	0.000071	0.001384	0.000116	0.000251	0.00194	0.012322	0.154355	1.051279	10.544585
24	0.000054	0.005715	0.000112	0.000249	0.001911	0.013263	0.150649	1.076399	10.509891

表 2: 计算时间/s(递归优化为迭代的并行版本)

p\n	10	20	25	30	35	40	45	50	55
串行	0.000001	0.000036	0.000339	0.003685	0.034168	0.22032	1.840282	19.100321	218.093399
1	0.00007	0.000034	0.000045	0.000044	0.000052	0.000033	0.000028	0.000083	0.000062
2	0.000058	0.000066	0.000045	0.000084	0.000051	0.000048	0.000036	0.000068	0.000079
4	0.000033	0.000072	0.0000061	0.000055	0.000054	0.000027	0.000071	0.000097	0.000079
8	0.00004	0.000062	0.000055	0.000047	0.000034	0.000045	0.000053	0.000073	0.000091
16	0.000035	0.000043	0.000056	0.00004	0.000060	0.000047	0.000061	0.000076	0.000071
24	0.000031	0.004317	0.000049	0.000038	0.000063	0.00005	0.000068	0.000088	0.000080

表 3: 计算时间/s(记忆化递归的并行版本)

从上表可以看到，普通递归并行版本计算时间约为串行版本的一半; 采用记忆化的并行版本实现了极大的优化，几乎为瞬间完成，这也验证了 $O(n)$ 与 $O(2^n)$ 的巨大差距; 采用优化后的递归并行版本也实现了 10-20 倍的速度提升。与串行版本相比，三个版本均实现了较高的效率，对并行版本的一点点改进直至获得百倍的加速也是很有成就感的。值得注意的是，在单线程情况下的计算时间居然比串行版本还要快，推测是编译器对并行版本做了更多的递归优化，减少了递归调用的开销。

A 对线程 (Thread) 和线程池 (ThreadPool) 的封装

代码在./threadPool 下。

为了方便创建线程，免去频繁书写 pthread 函数的繁琐，我实现了 Thread 类、ThreadPool 类，并对锁 (mutex, rwlock) 和信号量 (cond) 进行了简单的封装，在此基础上重写了 Programming Assignment 4.5。

A.1 对互斥量 (Mutex) 与信号量 (Cond) 以及 Locker 的封装

在写程序过程中，我发现“释放锁”的过程比较繁琐，在每一次 return 或 break 语句前都需要手动释放。因此我实现了 Locker 类，Locker 对象在**构造时加锁，析构时解锁**，这样就可以利用作用域来管理锁的释放。同时对 pthread 的 mutex, rwlock 和 cond 进行了封装。代码如下:

Mutex.h

```
class Mutex{//互斥量
    pthread_mutex_t m_mutex;
public:
    Mutex(){ pthread_mutex_init(&m_mutex, nullptr); }
    virtual ~Mutex(){ pthread_mutex_destroy(&m_mutex); }
    void lock(){ pthread_mutex_lock(&m_mutex); }
    void unlock(){ pthread_mutex_unlock(&m_mutex); }
};

class ReadWriteMutex{//读写锁
    pthread_rwlock_t m_rwlock;
public:
    ReadWriteMutex(){ pthread_rwlock_init(&m_rwlock, nullptr); }
    virtual ~ReadWriteMutex(){ pthread_rwlock_destroy(&m_rwlock); }
    void readlock(){ pthread_rwlock_rdlock(&m_rwlock); }
    void writelock(){ pthread_rwlock_wrlock(&m_rwlock); }
    void unlock(){ pthread_rwlock_unlock(&m_rwlock); }
};
```

Locker.h

```
class MutexLocker{
    Mutex* m_mutex;
public:
    MutexLocker(Mutex *mu): m_mutex(mu){ m_mutex->lock(); }
    virtual ~MutexLocker() {m_mutex->unlock(); }
};

class ReadLocker{
    ReadWriteMutex* m_rmutex;
public:
    ReadLocker(ReadWriteMutex *mu): m_rmutex(mu){ m_rmutex->readlock(); }
```

```
12      virtual ~ReadLocker() { m_rmutex->unlock(); }
13  };
14
15  class WriteLocker{
16      ReadWriteMutex* m_wmutex;
17  public:
18      WriteLocker(ReadWriteMutex *mu): m_wmutex(mu){ m_wmutex->writelock(); }
19      virtual ~WriteLocker() { m_wmutex->unlock(); }
20  };
```

Cond.h

```
1   class Cond{
2       pthread_cond_t m_cond;
3   public:
4       Cond(){ pthread_cond_init(&m_cond, nullptr); }
5       virtual ~Cond(){ pthread_cond_destroy(&m_cond); }
6       int wait(Mutex *mu){return pthread_cond_wait(&m_cond, &mu->getpmutex())
            ;}
7       void notify_one(){ pthread_cond_signal(&m_cond); }
8       void notify_all(){ pthread_cond_broadcast(&m_cond); }
9   };
```

A.2 对 Thread 的封装

仿照 c++ 标准的的线程库，我的 Thread 类继承 Runnable. 创建线程时，只需要继承 Thread 类重写 run() 方法即可。run() 方法负责在新线程中执行，只有 run() 方法内的变量是线程独有的。具体代码请见./Threadpool

Runnable.h

```
1   //Runable can be shared by multi-threads
2   class Runnable{
3       static int taskNumber;
4       int taskId;
5   public:
6       Runnable(){ taskId = taskNumber++; }
7       virtual void run() = 0;
8   };
```

Thread.h

```
1   //Thread can't be shared by multi-threads
2   class Thread: public Runnable{
3       pthread_t m_thread;
4       int m_state;//pthread_create()返回的线程状态
5       bool is_running;//标记线程是否正在执行
```

```
6       int thread_id;//线程id
7       void* m_args;//静态func()函数的参数
8       Runnable* runnable;//指向Runnable创建的线程,配合线程池使用
9       static int thread_tot;//线程总数
10      static void *func(void *args);//配合pthread_create()成为回调函数
11  public:
12      Thread(): m_state(0), m_args(nullptr), is_running(false), runnable(
            nullptr){
13          thread_id = thread_tot++;
14      }
15      Thread(Runnable* r): m_state(0), m_args(nullptr){
16          is_running = false;
17          thread_id = thread_tot++;
18          runnable = r;
19      }
20      virtual ~Thread(){
21          m_state = 0;
22          m_args = nullptr;
23      }
24      virtual void run(){
25          if(runnable){ runnable->run(); }
26      };
27      void start(){
28          m_state = pthread_create(&m_thread, nullptr, Thread::func, this);
29          is_running = true;
30      }
31      void join(){
32          if(is_running){
33              pthread_join(m_thread, nullptr);
34              is_running = false;
35          }
36      }
37      int threadid()const{ return thread_id; }
38  };
39  int Thread::thread_tot = 0;
40  void* Thread::func(void* args){
41      Thread* p = static_cast<Thread*>(args);
42      p->run();
43      return nullptr;
44  }
```

A.3 对线程池 (ThreadPool) 的封装

仿照 PA4.5 采用信号量实现线程调度的思路，我对线程池进行了封装。之后调用时，只需要继承 Runnable 类重写 run() 函数，之后调用 ThreadPool 类的 addTask() 函数压入任务队列即可。

ThreadPool.h

```
class ThreadPool{
    list<Runnable*> taskq;//任务队列，为了方便管理，使用list而不是queue
    unsigned thread_num;//线程个数
    Mutex mu;//互斥量
    Cond cond;//信号量
    pthread_t* threads;
    std::atomic_bool finished;//标记所有任务是否完成
public:
    ThreadPool(unsigned _n): thread_num(_n), finished(false),
        threads(new pthread_t[thread_num]){
        for(size_t i = 0; i < thread_num; i++){//一次性创建线程
            pthread_create(&threads[i], nullptr, ThreadPool::task, this);
        }
    }
    virtual ~ThreadPool(){
        finished = true;
        cond.notify_all();//没有任务就广播唤醒所有线程
        for(size_t i = 0; i < thread_num; i++){
            pthread_join(threads[i], nullptr);
        }
        for(auto iter: taskq){//销毁任务
            delete iter;
        }
    }
    void addTask(Runnable* t){//将任务压入队列
        MutexLocker locker(&mu);
        taskq.push_back(t);
        cond.notify_one();//唤醒线程执行该任务
    }
    static void* task(void* args);//作为回调函数
};

void* ThreadPool::task(void* args){//由子线程负责执行
    ThreadPool *thp = static_cast<ThreadPool*>(args);
    while(1){
        Runnable* t;
        {//条件等待，实现任务的分配调度
            MutexLocker locker(&thp->mu);
            while(thp->taskq.empty() && !thp->finished){
```

```
                while( thp->cond.wait(&thp->mu) != 0 )
                    ;
            }
            if(thp->taskq.empty()){
                break;
            }
            t = thp->taskq.front();
            thp->taskq.pop_front();
        }
        t->run();
    }
    return nullptr;
}
```

A.4 重写 Programming Assignment 4.5

经过上述封装之后，我重新编写了 4.5 的代码，实现了代码耦合度的减少和逻辑的清晰。首先是由 listTest 类继承 Runable 类重写 run() 方法，实现对 4.5 实现的有序链表的随机操作。

listTest

```
orderedList<int> mylist;//共享变量

class listTest: public Runnable{
    int op, value;
public:
    listTest() = default;
    listTest(int _op, int _val): op(_op), value(_val){}
    void run(){
        if (op < findbound) {//find for 80%
            if(mylist.find(value)){
                printf("threadid %u, find value %d success\n", (unsigned int
                    )pthread_self(), value);
            } else {
                printf("threadid %u, find value %d failed\n", (unsigned int)
                    pthread_self(), value);
            }
        } else if (findbound <= op && op < insertbound){//delete for 10%
            if(mylist.pop(value)){
                printf("threadid %u, delete value %d success\n", (unsigned
                    int)pthread_self(), value);
            } else {
                printf("threadid %u, delete value %d failed\n", (unsigned
                    int)pthread_self(), value);
            }
        } else {//insert for 10%
```

```
22              mylist.push(value);
23              printf("threadid %u, inserted value %d\n", (unsigned int)
                    pthread_self(), value);
24          }
25  #       ifdef DEBUG
26          std::cout << mylist.toString();
27  #       endif
28      }
29  };
```

其次在主线程中开辟线程池，不断创建任务提交给线程池，由线程池实现调度。

main.cpp

```
1  ThreadPool pool(thread_num);//开辟线程池
2  for(size_t i = 0; i < task_num; i++){//提交任务
3      pool.addTask( new listTest( rand() % 100, rand() % 500 ) );
4  }
```

可以看到，通过对 pthread 库进行良好的封装，可以极大方便后续编程任务的实现，降低了开发难度。