

# 课程设计报告

## 1. B+树bulkloading过程的理解

对于串行构造的bulkloading（源代码）的理解：

源代码自底向上构建B+树，因为将键值对一对一对写入文件中开销较大，所以使用bulkloading来构建，过程如下：

- 1) 创建叶子结点，在创建每一个叶子结点时将该结点所需要的、固定大小（本例为512字节）的空间，通过append\_block函数附加到文件末尾，修改第一个block中记录的"块数"。
- 2) 将其与前一个兄弟结点相连（第一个结点除外），若不是第一个结点，则连接完之后会对前一个结点进行delete操作，于是会调用析构函数，析构函数中的write\_block函数将该叶子结点的数据写入文件。
- 3) 向结点中批量填充键值对，直到一个结点被填满或者数据使用完毕，若被填满，则在下一个循环中会创建新的叶子结点，所以当前结点就变成了"上一个结点"，而"leaf\_act\_nd"被置为NULL，以通知程序上一个结点已满。
- 4) 在构建叶子结点的过程中标记了"start\_block" 和 "end\_block"，以在构建索引结点的过程中使用。

而构建索引结点的过程与构建叶子结点的过程十分相似，不同的是：

- 1) 区分level == 1/非1 的情况：当level == 1时，意味着其子结点为存储数据的叶子结点，反之为索引结点。
- 2) 判定结束的条件：last\_end\_block 和 last\_start\_block分别记录当前层最后一个和第一个结点，当值相等时自然意味着该层仅剩一个结点，即为根结点，所以索引结点的构建结束。

## 2. 算法并行的设计思路

方案一：

我们提出了一个索引结点的构建算法：设索引共需要有n层，每一层由一个线程建立，从第1层开始建立，若第k层结点数量超过1，则说明 $k < n$ ，索引未完成，需建立k+1层。第k层增加新结点时，应发信号给第k+1层的线程，将新结点插入，若因此需要新建结点，则同样上传信号到k+2层的线程，以此类推。当索引全部建立完成后应进行平衡，因为某层的最后一个结点容量可能不足一半，只需将最右边的两个结点中的key平均分即可，因为 $B(\text{容量}) \times x (x \geq 1) > B/2$ 。

方案二：

输入线程数目，按照原有的顺序，先并行构建叶子结点，再自底向上并行地构建每一层索引结点。其中，在并行构建叶子节点时，安排好每个叶子结点的位置，而不是顺序地append到文件末尾，例如当某个线程构建第n个叶子结点时，应当选择附加到位置  $(n + 1) * \text{block\_length}$ 处，而不是自然地附加到结尾（或者说提前算好每个结点的block）。

方案三：

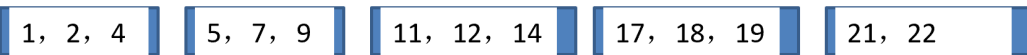
但由于经过多次尝试，不断修改，仍无法正确完成此算法，获取正确的结果，所以我们选择了另一个比较简单的方法。通过阅读代码，可以得知一个叶子结点需要读入115个id值，创建一个叶子结点数组，通过计算可以得知一共需要多少个叶子结点，然后将这些叶子结点先初始化，最后多个线程并行往叶子节点中写入数据。索引结点与叶子结点也类似，由于索引结点数量较少，可以使用两个线程并行构建一层索引结点，方法与叶子节点相同，逐层往上，最后根节点只需要一个线程进行构建。

多线程的实现通过线程池完成。


### 3. 算法流程图

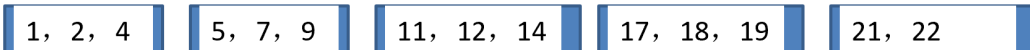
所提出的索引结点构建算法流程图：

1

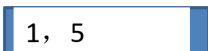
叶子 

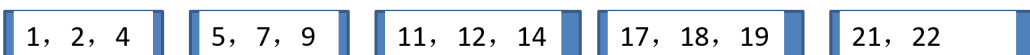
2

Pth1 

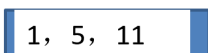
叶子 

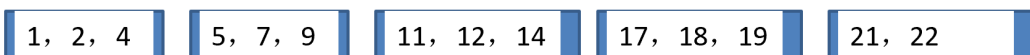
3

Pth1 

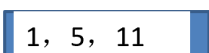

叶子 

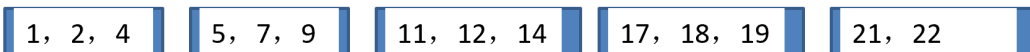
4

Pth1 

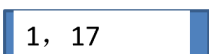
叶子 

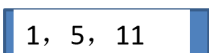
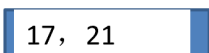
5

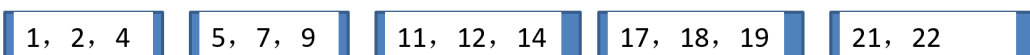
Pth1   结点多于1说明此非root

叶子 

6

Pth2  叶子全部加入到树中，此层只有1个结点，所以是root

Pth1   结点多于1说明此非root

叶子 

## 4. 关键代码描述

### 线程池的实现

在多线程的实现中，我们小组实现了一个线程池来辅助实现结点构建的并行化。线程池结构包括一个任务队列结构，一个线程状态结构，最大线程数，空闲线程数，运行线程数，线程池退出标志。初始时线程都处于空闲状态，当有新的任务进来，从线程池中取出一个空闲的线程处理任务，然后当任务处理完成之后，该线程被重新放回到线程池中，供其他的任务使用，当线程池中的线程都在处理任务时，就没有空闲线程供使用，此时，若有新的任务产生，只能等待线程池中有线程结束任务空闲才能执行。

线程池的基础结构，针对执行需要的线程任务，可对线程结构进行修改，添加参数（本次实验就使用到了四个参数）

```
typedef struct task{
    void *(*run)(void *args);    //需要执行的任务
    void *arg;                   //参数
    struct task *next;
}task;

typedef struct condition{
    pthread_mutex_t pmutex;
    pthread_cond_t pcond;
}condition;

typedef struct threadpool{
    condition condition_;    //状态量
    task *first;             //任务队列中第一个任务
    task *last;              //任务队列中最后一个任务
    int run_thread;          //运行线程数
    int space_thread;        //空闲线程数
    int max_thread;          //最大线程数
    int quit;                //是否退出标志
}threadpool;
```

线程池初始化：对线程状态和各种变量进行初始化

```
int condition_init(condition *cond){
    int status;
    if((status = pthread_mutex_init(&cond->pmutex, NULL))) {
        return status;
    }
    if((status = pthread_cond_init(&cond->pcond, NULL))) {
        return status;
    }
    return 0;
}

//线程池初始化
void threadpool_init(threadpool *pool, int threads){
    condition_init(&pool->condition_);
    pool->first = NULL;
    pool->last = NULL;
    pool->run_thread = 0;
    pool->space_thread = 0;
    pool->max_thread = threads;
    pool->quit = 0;
}
```

线程池中添加一个任务：首先对线程池加锁，然后把新加入的任务进队；如果线程池中有空闲的线程，唤醒；当线程池中运行的线程数少于最大线程数，创建一个新的线程；线程池解锁。

```
//增加一个任务到线程池
void threadpool_add_task(threadpool *pool, void *(*run)(void *arg), void *arg){
    task *newtask = (task *)malloc(sizeof(task));
    newtask->run = run;
    newtask->arg = arg;
    newtask->next = NULL;

    //线程池的状态被多个线程共享，操作前需要加锁
    pthread_mutex_lock(&(pool->condition_).pmutex);

    if(pool->first == NULL){
        pool->first = newtask;
    }
    else{
        pool->last->next = newtask;
    }
    pool->last = newtask;

    //线程池中有线程空闲，唤醒
    if(pool->space_thread > 0){
        pthread_cond_signal(&(pool->condition_).pcond);
    }
    //当前线程池中线程个数没有达到设定的最大值，创建一个新的线程
    else if(pool->run_thread < pool->max_thread){
        pthread_t tid;
        pthread_create(&tid, NULL, thread_run, pool);
        pool->run_thread++;
    }
    //结束，访问
    pthread_mutex_unlock(&(pool->condition_).pmutex);
}
```

线程运行函数：用一个while循环不断地对任务队列中任务进行处理，进入运行函数，先对线程池加锁，然后等待任务队列的添加，设置一个等待睡眠时间，超时退出线程池。当任务队列不为空即有任务加入时，由于处理任务需要时间，线程池解锁允许其他线程访问线程池，然后处理任务，处理完任务后，线程池重新加锁。如果当线程池退出标志为1并且线程池中的任务队列为空时，如果运行线程数为0将线程唤醒，然后线程池解锁，该线程退出线程池。或者如果当线程睡眠时间超时，线程池解锁，该线程退出线程池。如果该线程继续运行，线程池解锁，继续while循环。

```
//创建的线程执行
void *thread_run(void *arg){
    struct timespec abstime;
    int timeout;
    printf("thread %d is starting\n", (int)pthread_self());
    threadpool *pool = (threadpool *)arg;
    while(1){
        timeout = 0;
        //访问线程池之前需要加锁
        pthread_mutex_lock(&(pool->condition_).pmutex);
        pool->space_thread++;
        //等待队列有任务到来
        while(pool->first == NULL && !pool->quit){
            printf("thread %d is waiting\n", (int)pthread_self());

```

```

        //获取从当前时间, 并加上等待时间, 设置进程的超时睡眠时间
        clock_gettime(CLOCK_REALTIME, &abstime);
        abstime.tv_sec += 2;
        int status;
        //该函数会解锁, 允许其他线程访问, 当被唤醒时, 加锁
        status = pthread_cond_timedwait(&(pool->condition_).pcond, &(pool->condition_).pmutex, &abstime);
        if(status == ETIMEDOUT){
            printf("thread %d wait timed out\n", (int)pthread_self());
            timeout = 1;
            break;
        }
    }

    pool->space_thread--;

    if(pool->first != NULL){
        task *t = pool->first;
        pool->first = t->next;
        //由于任务执行需要消耗时间, 先解锁让其他线程访问线程池
        pthread_mutex_unlock(&(pool->condition_).pmutex);
        //执行任务
        t->run(t->arg);
        //执行完任务释放内存
        free(t);
        //重新加锁
        pthread_mutex_lock(&(pool->condition_).pmutex);
    }

    //退出线程池
    if(pool->quit && pool->first == NULL){
        pool->run_thread--;
        //若线程池中没有线程, 通知等待线程全部任务已经完成
        if(pool->run_thread == 0){
            pthread_cond_signal(&(pool->condition_).pcond);
        }
        pthread_mutex_unlock(&(pool->condition_).pmutex);
        break;
    }

    //超时, 退出线程池
    if(timeout == 1){
        pool->run_thread--;
        pthread_mutex_unlock(&(pool->condition_).pmutex);
        break;
    }
    pthread_mutex_unlock(&(pool->condition_).pmutex);
}
printf("thread %d is exiting\n", (int)pthread_self());
return NULL;
}

```

经过测试, 线程池可以正常运行, 不会发生冲突冒险, 实现了互斥。

## 并行构建叶子结点

设定并行分裂输入数据table的块大小

```
int merge_size = 115000; //并行读取table分割数目, 115 * 1000, 1000个叶子结点
int num_task = 0;        //线程数目
```

计算需要的线程数量, 计算需要建立的叶子结点数量, 初始化叶子结点数组

```
threadpool pool;
num_task = (int)(n / merge_size) + 1; //计算需要的线程数
int max_thread = num_task + 1;
node_num = n / 115 + 1; //计算需要建立的叶子结点数
BLeafNode **BLeafNodeArray = new BLeafNode*[node_num]; //叶子结点数组

for(int i = 0; i < node_num; i++){
    BLeafNodeArray[i] = NULL;
}

//建立叶子结点, 往叶子结点中添加初始数据, 得到该叶子的key值, 获取叶子结点block值, 每115个数据
//创建一个叶子结点
for(int i = 0, j = 0; i < node_num; i++, j = j + 115){
    id = table[j].id_;
    key = table[j].key_;
    BLeafNode *leaf_act_nd = NULL;
    if(!leaf_act_nd){
        leaf_act_nd = new BLeafNode();
        leaf_act_nd->init(level, this);
    }
    leaf_act_nd->add_new_child(id, key);
    BLeafNodeArray[i] = leaf_act_nd;
}
```

初始化线程池, 然后往线程池中添加并行构建叶子结点线程函数task\_leaf\_node

```
//初始化线程池
threadpool_init(&pool, max_thread);
//往线程池中添加任务
for (int i = 0; i < num_task; i++) {
    int *arg3 = (int*)malloc(sizeof(int));
    *arg3 = merge_size * (i + 1);
    if(i == num_task - 1){
        *arg3 = n;
    }
    //将table, 叶子结点数组, end, 树本身传入线程函数
    threadpool_add_task(&pool, task_leaf_node, (void*)&(*table), (void*)&
    (*BLeafNodeArray), arg3, (void*)&(*this));
}
```

并行构建叶子结点线程函数task\_leaf\_node, 叶子结点满了就构建下一个叶子结点

```
//并行读取table, 写入叶子结点中
void* task_leaf_node(void *arg1, void *arg2, void *arg3, void *arg4){
    Result *table = (Result*)arg1;
    BLeafNode **LeafArray = (BLeafNode**) arg2; //叶子结点数组
```

```

int *numptr = (int *)arg3;
int end = *numptr;;
int start = end - merge_size;
if(end == 1000000){
    start = merge_size * (num_task - 1);
}

int id = -1;
float key = MINREAL;
int thread = (start / merge_size);
int j = thread * (merge_size / 115);

//数据写入叶子结点
for(int i = start; i < end; i++){
    if(i % 115 == 0){
        continue;
    }
    //获取table中的id和value值
    id = table[i].id_;
    key = table[i].key_;
    //如果当前叶子结点已满, 进入下一个
    if(LeafArray[j]->isFull()){
        j++;
    }
    LeafArray[j]->add_new_child(id, key);
}
//任务结束, wait自增
wait++;
return NULL;
}

```

## 并行构建索引结点

使用一个while循环, 一次循环为一层索引结点的构建, 当start\_block = end\_block时, 说明此时结点只剩下一个根节点, 不再需要构建索引结点, 循环结束。

```
while(last_start_block < last_end_block){}
```

先计算得到该层所需要的索引结点数量, 初始化索引结点数组, 如果当前层数为一, 此时的儿子结点为叶子结点, 通过block值重加载叶子结点并且获取其key值。如果当前层数大于一, 此时的儿子结点为索引结点。

```

block = -1;
key = MINREAL;
bool first_node = true;
node_num = (last_end_block - last_start_block) / 62 + 1; //计算改层需要构建的索引结
点数量

BLeafNode *leaf_child = NULL;
BIndexNode *index_child = NULL;
BIndexNode **BIndexNodeArray = new BIndexNode*[node_num]; //索引结点数组

//构建索引结点
for (int i = last_start_block, j = 0; i <= last_end_block; i = i + 62, j++) {
    block = i;
    //根据block值, 从儿子结点中获取key值

```

```

if(level == 1){
    //第一层，儿子结点为叶子结点
    leaf_child = new BLeafNode();
    leaf_child->init_restore(this, block);
    //获取key值
    key = leaf_child->get_key_of_node();
    if (leaf_child != NULL) {
        delete leaf_child; leaf_child = NULL;
    }
}
else{
    //其他层次，儿子结点为索引结点
    index_child = new BIndexNode();
    index_child->init_restore(this, block);
    key = index_child->get_key_of_node();
    if (index_child != NULL){
        delete index_child; index_child = NULL;
    }
}

//生成索引结点，进行初始化和添加数据
BIndexNodeArray[j] = new BIndexNode();
BIndexNodeArray[j]->init(level,this);
BIndexNodeArray[j]->add_new_child(key, block);
}

```

并行构建索引结点，由于索引结点数量较少，只需要两个线程同时构建即可。但如果当前的索引结点为根结点，则只需要一个线程进行构建。

```

wait = 0;
num_task = 2; //由于索引结点数目较少，所以选择两个线程进行同时构建即可
int start = last_start_block;
int end = last_start_block + (node_num / 2) * 62 - 1;
//如果当前为根节点，则只需要一个线程进行构建
if(BIndexNodeArray[0]->get_block() == BIndexNodeArray[node_num - 1]-
>get_block()){
    num_task = 1;
    end = last_end_block;
}
//开始多线程并行构建索引结点
for (int i = 0; i < num_task; i++) {
    int *arg2 = (int*)malloc(sizeof(int));
    *arg2 = start;
    int *arg3 = (int*)malloc(sizeof(int));
    *arg3 = end;
    start = end + 1;
    end = last_end_block;
    printf("level:%d start:%d end:%d\n", level, *arg2, *arg3);
    //将索引结点数组，树本身，start和end传入线程函数
    threadpool_add_task(&pool, task_index_node, (void*)&(BIndexNodeArray), arg2
, arg3, (void*)&(this));
}

```

并行构建索引结点线程函数task\_index\_node，构建索引结点方法与初始化索引结点数组相同

```

void* task_index_node(void *arg1, void *arg2, void *arg3, void *arg4){

```



```

BIndexNode **IndexArray = (BIndexNode**) arg1; //索引结点数组
int *numptr = (int *)arg2;
int start = *numptr;
int *numptr1 = (int *)arg3;
int end = *numptr1;

int block = -1;
float key = MINREAL;
BLeafNode *leaf_child = NULL; //儿子叶子结点
BIndexNode *index_child = NULL; //儿子索引结点

int j = 0;
if(end == last_end_block){
    j = node_num / 2;
}
//通过block值, 读入儿子结点的key值
for (int i = start; i <= end; i++) {
    if((i - start) % 62 == 0){
        continue;
    }

    block = i;
    //获取儿子结点的key值
    if(level == 1){
        //第一层, 儿子结点为叶子结点
        leaf_child = new BLeafNode();
        leaf_child->init_restore((BTree*)arg4, block);
        //获取key值
        key = leaf_child->get_key_of_node();
        if (leaf_child != NULL) {
            delete leaf_child; leaf_child = NULL;
        }
    }
    else{
        //其他层次, 儿子结点为索引结点
        index_child = new BIndexNode();
        index_child->init_restore((BTree*)arg4, block);
        key = index_child->get_key_of_node();
        if (index_child != NULL) {
            delete index_child; index_child = NULL;
        }
    }

    //如果当前索引结点已满, 进入下一个
    if (IndexArray[j]->isFull()) {
        //printf("level:%d part:%d j:%d\n", level, part, j);
        j++;
    }
    IndexArray[j]->add_new_child(key, block);
}
//printf("level:%d part:%d j:%d\n", level, part, j);
wait++;
return NULL;
}

```

该层索引结点建立完成, 更新last\_start\_block和last\_end\_block, 进入下一层的索引结点构建, 如果last\_start\_block和last\_end\_block相等, 说明此时为根节点, 结束索引结点的构建

```
//更新下一层的start_block和end_block值
last_start_block = BIndexNodeArray[0]->get_block();
last_end_block = BIndexNodeArray[node_num - 1]->get_block();
level++;
```

## 检查函数

首先读取root\_block，也就是root的位置，顺便确认root的level，然后对root的子结点进行遍历，遍历函数

```
void check_node(int block){
    char level;
    int num_entries, num_keys, left_sibling, right_sibling, son;
    float key;
    fseek(tree_fp, (1 + block)*block_length, SEEK_SET); // 定位到块的位置
    fread(&level, SIZECHAR, 1, tree_fp); // 通过块的level就可以判断是叶子节点还是索引节
    点
    if(level != 0){ // level != 0 时, 为索引节点, 1char + 3int + num_entries (51) 对
    key和son
        fread(&num_entries, SIZEINT, 1, tree_fp);
        fread(&left_sibling, SIZEINT, 1, tree_fp);
        fread(&right_sibling, SIZEINT, 1, tree_fp);
        for(int i = 0; i < num_entries; i++){
            fread(&key, SIZEFLOAT, 1, tree_fp);
            fread(&son, SIZEINT, 1, tree_fp);
            check_node(son); // son 即子节点, 继续遍历子节点
            fseek(tree_fp, (1+block)*block_length+SIZECHAR+SIZEINT*3+(i+1)*
            (SIZEFLOAT+SIZEINT), SEEK_SET);
        }
    }
    else if(level == 0){ // level == 0 时, 为叶子节点, 1char + 4int + 8key + 115id
        fread(&num_entries, SIZEINT, 1, tree_fp);
        fread(&left_sibling, SIZEINT, 1, tree_fp);
        fread(&right_sibling, SIZEINT, 1, tree_fp);
        fread(&num_keys, SIZEINT, 1, tree_fp);
        float keys[num_keys]; // 8
        int ids[num_entries]; // 115
        fread(keys, SIZEFLOAT, num_keys, tree_fp);
        fread(ids, SIZEINT, num_entries, tree_fp);
        for(int i = 0; i < num_entries; i++){
            if(ids[i] != 0){
                if(ids[i] == table_copy[id_total].id_) {
                    id_correct++;
                }
                else printf("\n %d and %d", ids[i], table_copy[id_total].id_);
                id_total++;
            }
        }
    }
}

void check(){
    printf("\ncomparing ids...\n");
    int num_blocks;
    int root_block;
    tree_fp = fopen(tree_file, "r");
```

```

if(!tree_fp) exit(1);
fread(&block_length, SIZEINT, 1, tree_fp);
fread(&num_blocks, SIZEINT, 1, tree_fp);
fread(&root_block, SIZEINT, 1, tree_fp); // 得到root_block
fseek(tree_fp, (1 + root_block) * block_length, SEEK_SET); // 定位到root
fread(&root_level, SIZECHAR, 1, tree_fp); // 取得root的level, 主要是验证有多少层
printf(" block_length = %d\n num_blocks = %d\n root_block = %d\n root_level
= %d\n", block_length, num_blocks, root_block, root_level);
check_node(root_block);
fclose(tree_fp);
printf("\naccuracy of id = %d/%d\n", id_correct, id_total);
}

```

## 补充：方案二的核心实现代码

// 根据线程数目，计算每个线程需要负责多少个结点，然后计算每个线程负责的范围 (arg2~arg3)，分配给每个子线程

//往线程池中添加任务

```

for (int i = 0; i < numOfThread; i++) {
    int* arg2 = (int*)malloc(sizeof(int));
    int* arg3 = (int*)malloc(sizeof(int));
    *arg2 = i * num_of_node_of_a_thread;
    *arg3 = (i + 1) * num_of_node_of_a_thread;
    threadpool_add_task(&pool, create_leaf, (void*)&(*table), arg2, arg3);
}

```

// 在构造叶子结点的子线程中，对于init函数，多传入一个参数 i+1，以指定叶子节点的block位置，  
// 与此同时也相应的修改了BLeafNode::init函数和append\_block函数

```

for (int i = start; i < end; ++i) {
    id = table[i].id_;
    key = table[i].key_;

    if (!leaf_act_nd) {
        leaf_act_nd = new BLeafNode();
        leaf_act_nd->init(0, tree, i + 1);
        ...
    }
    ...
}

```

// 多线程还要处理start\_block的取值问题，修改代码，使得只有i == 0时才可以修改start\_block即可

```

if (first_node) {
    first_node = false;
    if (i == 0){
        start_block = leaf_act_nd->get_block();
    }
    else{ /* 与正常的叶子结点处理方式无异 */ }
}

```

## 5. 实验结果

并行构建的情况下运行截图如下：

```
accuracy of id = 999997/999997
lgx@ubuntu:~/Desktop/Tree_project$ ./run
data_file   = ./data/dataset.csv
tree_file   = ./result/B_tree
level:1 start:1 end:4340
level:1 start:4341 end:8696
level:2 start:8697 end:8758
level:2 start:8759 end:8837
level:3 start:8838 end:8840
运行时间: 0.129521 s

comparing ids...
block_length = 512
num_blocks = 8842
root_block = 8841
root_level = 3

accuracy of id = 999997/999997
```

可看出运行时间为0.12秒，root\_block是第8842个块，处于8841的位置（首位为0），根结点的level为3，经过check函数按深度优先遍历的方式检查，所有结点的值与table数组中的顺序一致。

## 6. 实验分析以及性能调优

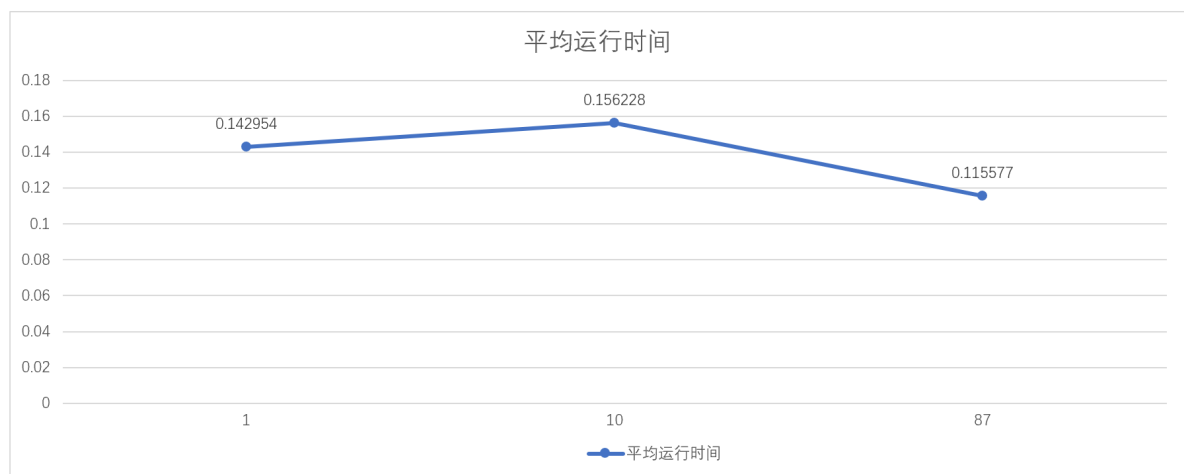
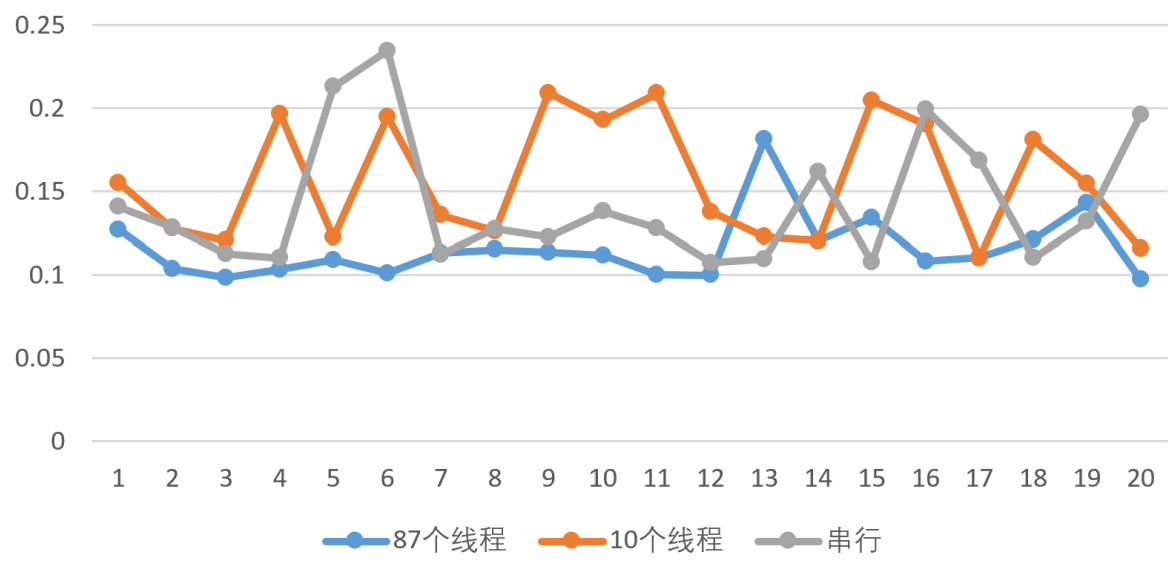
在实现如上算法的基础上，通过调整线程数来进行性能调优

分别将串行、10个线程、87个线程三种情况运行20次并取平均值，结果如下：

|     | 87个线程    | 10个线程    | 串行       |
|-----|----------|----------|----------|
|     | 0.126855 | 0.154979 | 0.140595 |
|     | 0.103872 | 0.127551 | 0.128147 |
|     | 0.098365 | 0.120455 | 0.112783 |
|     | 0.103224 | 0.196538 | 0.110294 |
|     | 0.109053 | 0.122185 | 0.212811 |
|     | 0.101154 | 0.194746 | 0.234365 |
|     | 0.113165 | 0.135571 | 0.112283 |
|     | 0.115153 | 0.126128 | 0.127508 |
|     | 0.113579 | 0.209129 | 0.122488 |
|     | 0.111675 | 0.192809 | 0.137878 |
|     | 0.100298 | 0.208768 | 0.127892 |
|     | 0.09998  | 0.137533 | 0.107354 |
|     | 0.18114  | 0.122781 | 0.109668 |
|     | 0.120178 | 0.120213 | 0.161669 |
|     | 0.134028 | 0.204533 | 0.107851 |
|     | 0.108095 | 0.190071 | 0.198971 |
|     | 0.110473 | 0.110054 | 0.168489 |
|     | 0.121217 | 0.180738 | 0.110291 |
|     | 0.142687 | 0.154283 | 0.131705 |
|     | 0.09734  | 0.115503 | 0.196028 |
| 平均值 | 0.115577 | 0.156228 | 0.142954 |

绘制成折线图可得：

图表标题



可看出，增加线程数到一定数目，运行时间有明显的缩短。

设定线程数为87时，平均运行时间相对于串行执行降低了19.15%