浙江大學

《多线程排序》

实验报告

题　　目：	多线程排序
课程名称：	C程序设计专题
授课教师：	翁恺
姓　　名：	黄文翀
学　　号：	3200100006
班　　级：	强基数学2001班
专业学院：	数学科学学院

多线程排序

黄文翀，3200100006 （学号）

浙江大学 数学科学学院 强基数学2001班

摘　要： 多线程算法是提升运算效率的有力工具。本实验主要研究了多线程在排序算法中的应用，并利用分块排序后多路归并的方式，实现了多线程的快速排序与归并排序。此外，本文还研究了多线程归并的另一种实现方式，即将归并排序中的第一个递归调用改为派生线程，本文从理论上对此算法进行分析，并通过实验进行检验，最后指出理论复杂度难以实现的原因。

关键词： 计算机；排序算法；多线程；

目录

1 实验要求

自学pthread库，实现多线程的快速排序与归并排序。

2 基本思路

将数组均匀地分为`K`份，对每一份使用一个单独的线程进行排序（快速排序或归并排序），待所有线程同步后，进行`K`路归并以得到最后排好序的数组。`K`的取值由实验结果确定。

3 实现

3.1 pthread库的基本函数

使用以下函数创建一个线程：

```
int pthread_create(pthread_t* tid, const pthread_attr_t* attr, void* thread, void* args);
```

其中`tid`是线程的句柄；`attr`是线程的一些特性，一般传`NULL`即可；`thread`是线程主函数；`args`是传给线程主函数的参数，参数表通常封装在结构体中，然后将结构体指针强制转换为`void*`类型传入。

使用以下函数等待一个线程结束，以实现同步：

```
int pthread_join (pthread_t tid, void ** retval);
```

其中`tid`是线程的句柄；`retval`是用户定义的指针，用于存储线程的返回值。

3.2 数组的划分与子线程的创建、同步

```
SIZE = n / THERAD_NUM;
for(int i = 0; i < THERAD_NUM; i ++){
    b[i] = (struct Array *) malloc(sizeof(struct Array));
    b[i] -> a = a + i * SIZE;
    b[i] -> n = (i == THERAD_NUM - 1) ? (n - i * SIZE) : SIZE;
    pthread_create(&tid[i], NULL, sort_thread, (void*)(b[i]) );
}
for(int i = 0; i < THERAD_NUM; i++){
    pthread_join(tid[i] , NULL);
}
```

将数组分成`THREAD_NUM`份，计算平均大小`SIZE`，然后将每一块的起始地址、长度封装到结构体中，传给排序子线程`sort_thread`，特别注意数据的数量不一定是线程数的整数倍，因此最后一块的大小不一定是`SIZE`，需要通过计算余量来确定。

主线程需要调用`pthread_join`函数等待所有排序子线程结束，同步之后进行后续处理。

3.3 排序子线程

```
void* sort_thread(void *args){
    struct Array *arr = (struct Array *) args;
    if(MODE == 'Q'){
        printf("[Thread] Mode: Quick Sort    length = %d\n", arr -> n);
        quick_sort(arr -> a, arr -> n);
    }
    else if(MODE == 'M'){
        printf("[Thread] Mode: Merge Sort    length = %d\n", arr -> n);
        merge_sort(arr -> a, arr -> n);
    }
    else printf("undefined mode.");
}
```

子线程得到数组的起始地址与长度，然后直接调用单线程的快速排序或归并排序函数，将数组完成排序即可，无需与其它线程进行交互。

3.4 快速排序

```
void _quick_sort_(int *a, int n){
    int t = Rand() % n;
    swap(&a[0], &a[t]);
    int l = 0, r = n - 1, x = a[0];
    while(l < r){
        while(l < r && a[r] >= x) r--;
        if(l < r) a[l] = a[r];
        while(l < r && a[l] <= x) l++;
        if(l < r) a[r] = a[l];
    }
    a[l] = x;
    if(l) _quick_sort_(a, l);
    if(n - l - 1) _quick_sort_(a + l + 1, n - l - 1);
}
```

基本思想：在每一段中选取一个数，一般选取第一个，将比它小的放在它左边，比它大的放在它右边。

这可以不需要借助辅助数组，只要左右指针交替向中靠拢即可。具体地，找到从右往左第一个比x小的数，放到左指针的位置，然后左指针向右找到第一个比x大的数，放到右指针的位置，不断反复，直到两指针相遇，把x放到相遇的位置即完成分类。然后对两边递归处理即可。

一开始必须进行一次随机交换，否则遇到递减的数据效率会退化为$O(n^2)$，随机化之后期望复杂度是$O(n\log n)$。

3.5 归并排序

```
void merge_sort(int *a, int n){
    int *tmp = (int*) malloc(sizeof(int) * n);
    _merge_sort_(a, n, tmp);
    free(tmp);
}

void _merge_sort_(int *a, int n, int *tmp){
    int mid = n / 2;
    if(mid >= 2) _merge_sort_(a, mid, tmp);
    if(n - mid >= 2) _merge_sort_(a + mid, n - mid, tmp + mid);
    int pl = 0, pr = mid, idx = 0;
    while(pl < mid || pr < n){
        if(pl == mid || pr < n && a[pr] < a[pl]) tmp[idx++] = a[pr++];
        else tmp[idx++] = a[pl++];
    }
    memcpy(a, tmp, sizeof(int) * n);
```

```
}
```

基本思路：先把两半分别做好，然后二路归并得到排好序的数组，随后返回。可以认为快速排序是“自上而下”的，而归并排序是“自下而上”的。

具体实现由于每次递归都要申请辅助空间很慢，所以一开始先申请好一个长为n的辅助空间，然后把地址传给递归，共用这个辅助空间。二路归并，即不断挑出两路开头中较小的放进结果数组，挑完即止。

复杂度是稳定的$O(n\log n)$。

3.6 多路归并

```
void nth_merge(int *a, int n, struct Array *b[]){
    int p[THERAD_NUM] = {0};
    int* tmp = (int *) malloc(sizeof(int) * n);
    int cnt = 0;

    while(1){
        //找到每一路开头元素中最小的那个
        int idx = -1;
        for(int i = 0; i < THERAD_NUM; i++){
            if(p[i] == b[i] -> n) continue;
            if(idx == -1 || b[i] -> a[p[i]] < b[idx] -> a[p[idx]]) idx = i;
        }
        tmp[cnt++] = b[idx] -> a[p[idx]++];

        //检查是否每一路都已完成归并
        int check = 1;
        for(int i = 0; i < THERAD_NUM; i++){
            if(p[i] < b[i] -> n){
                check = 0;
                break;
            }
        }
        if(check) break;
    }

    memcpy(a, tmp, sizeof(int) * n);
    free(tmp);
}
```

在所有的线程完成排序之后，需要合并得到一个排好序的数组，这就要进行多路归并。完全类似归并排序中的二路归并，每一路都设一个指针`p[i]`用于记录当前这一路比较到的位置，比较每一路剩余部分的第一个元素，找到最小的，放到结果数组的下一个位置，直到每一路的指针都走到末尾。

4 算法效率

4.1 复杂度理论分析

设待排序元素有n个，线程数为k

空间复杂度是简单的，不论是快速排序还是归并排序，总是只需要线性大小的额外辅助空间，空间复杂度$\Theta(n)$

不论是快速排序还是归并排序，单线程排m个元素的时间复杂度都是$\Theta(m\log m)$。每个线程都负责$\frac{n}{k}$个（最后一块可能会有出入，但相差不多，直接近似处理）元素的排序，所以第一部分的时间复杂度为$\Theta\left(\frac{n}{k}\log\frac{n}{k}\right)$。

第二部分是k路归并，由于结果数组每增加一个元素，都需要进行k次的比较，因此总共的比较次数为nk，所以第二部分的时间复杂度为$\Theta(nk)$。

总的时间复杂度为$\Theta\left(\frac{n}{k}\log\frac{n}{k}+nk\right)$，现在来分析$k$的最佳取值

设$f(n,k)=\frac{n}{k}\log\frac{n}{k}+nk$，求偏导得$\frac{\partial f}{\partial k}=\frac{n(-\ln(n/x)+x^2\ln 2-1)}{x^2\ln 2}$

利用几何画板求$\frac{\partial f}{\partial k}(n,k)$的零点（$n$作为参数，$k$作为变量），发现零点关于$n$是递增的，且当$n$的取值在$10^5$至$10^7$之间时，零点的取值都在$4.00$至$4.75$中，而$n$再小一些，程序运行是非常快的，不做考虑。

因此取$k=4$或5，总是能使效率最高，这就决定了我们将确定线程数为4或5。

4.2 实际测试

4.2.1 测试环境

操作系统：`Ubuntu 20.04.2 LTS`

操作系统类型：`64位`

处理器：`Intel® Core™ i7-8550U CPU @ 1.80GHz × 8`

编译器：`gcc`

计时方法：`库 <sys/time.h> 中的函数 gettimeofday()`

4.2.2 测试过程与结果

随机生成了10^7个`int`范围内的整数，附数据生成器代码如下：

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(){
    FILE *fp = fopen("data.in", "w");
    srand(time(0));
    int n = 10000000;
    fprintf(fp, "%d\n", n);
    for(int i = 0; i < n; i++){
        fprintf(fp, "%d ", rand());
    }
    return 0;
}
```

随后分别选取线程数为$4,5,6$，对两种排序算法，分别进行100组测试，统计两个阶段的平均耗时，以及总平均耗时，结果如下：

线程数	算法	分组排序平均耗时(ms)	多路归并平均耗时(ms)	总平均耗时
4	快速排序	629	327	957
5	快速排序	600	385	986
6	快速排序	527	450	978
4	归并排序	627	337	964
5	归并排序	647	383	1031
6	归并排序	556	423	981

可以看到，虽然理论分析的最优线程数更加接近5，但不论是何种算法，5线程总是比4线程劣，甚至劣于6线程。此外，除5线程之外，快速排序与归并排序效率相差不大。

但是，根据理论分析，随着线程数增加，分组排序耗时减少，多路归并耗时增加，归并排序算法的实验结果明显不符合我们的预期。

需要指出的是，归并排序算法在5线程下的反常表现并不是一个偶然结果，我们反复进行了实验，且中途没有进行重新编译，没有进行其它人为干扰实验的操作，结果都表明5线程归并排序的分组排序耗时反常增加，更换数据之后仍然如此。

我们猜测也许这和分组之后每一段的长度与2的整数次幂的接近程度有关，分为4组更接近2^{18}，而分为5组恰好卡在2^{17}与2^{18}之间。当然这一定有涉及到计算机底层处理的更加复杂的问题，这一问题我们将在以后进一步研究。

5 多线程归并排序的第二种实现

```
void* merge_sort(void* args){
    struct Array* arr = (struct Array*) args;
    int* a = arr -> a;
    int n = arr -> n, mid = n / 2;

    pthread_t tid = 0;
    if(mid >= 2) pthread_create(&tid, NULL, merge_sort, make_array(a, mid));
    if(n - mid >= 2) merge_sort(make_array(a + mid, n - mid));
    if(tid) pthread_join(tid, NULL);
    merge(arr, mid);
}
```

这是算法导论中多线程归并的实现，这种实现更加简洁，看起来效率也更高一些，本质上是把原本串行进行的两个子递归并行起来了，如果假设并行能力无限大，理论上分析，这个算法的时间复杂度甚至是$\Theta(n)$的。

但很不巧，由于线程数的膨胀，实际测试中，这种实现的耗时甚至是单线程排序的两倍，因此，我们对具体实现进行优化。具体来说，如果左半部分的长度不超过2^{20}，那么我们就不另开线程：

```
void* merge_sort(void* args){
    struct Array* arr = (struct Array*) args;
    int* a = arr -> a;
    int n = arr -> n, mid = n / 2;

    pthread_t tid = 0;
    if(mid >= 2){
        if(mid < SIZE) merge_sort(make_array(a, mid));
        else pthread_create(&tid, NULL, merge_sort, make_array(a, mid));
    }
    if(n - mid >= 2) merge_sort(make_array(a + mid, n - mid));
    if(tid) pthread_join(tid, NULL);
    merge(arr, mid);
}
```

这个实现是优秀的，10^7规模下的数据，实际测试确实比分组排序后多路归并会快上约100ms

6 实验心得

多线程排序的复杂度分析比单线程更加复杂，与实际测试的差异也更大些。这是因为多线程排序的分析，如果要确切分析，还涉及到工作量、持续时间、线性加速、并行度等各种量的分析。而实际测试上，会与计算机的底层硬件有较大关系。我们还有很多理论知识需要学习。