浙江大学实验报告

专业：计算机科学与技术
姓名：熊子宇
学号：3200105278
日期：2021.6.23

课程名称：C 程序设计专题　　指导老师：翁恺　　成绩：

实验名称：作业 4：并行归并排序与快速排序

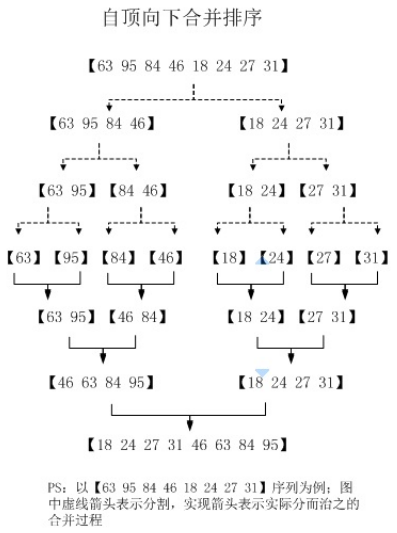
一、实验题目要求

- 自学pthread库
- 用pthread库做多线程实现的归并排序和快速排序
- 比较两种并行排序的性能

二、实验思路和过程描述

1. 归并排序思路

- 先将长度为N的无序序列分割平均分割为两段
- 然后分别对前半段进行归并排序、后半段进行归并排序
- 最后再将排序好的前半段和后半段归并



PS：以【63 95 84 46 18 24 27 31】序列为例；图中虚线箭头表示分割，实现箭头表示实际分而治之的合并过程

2. 快速排序思路

- 取出基准数*pivot*，使pivot左边的数比它小，右边的数比它大
- 对左边和右边分别快速排序(递归过程)

3. 并行计算——判别递归树深度*Tree Depth*

如上图，呈现了归并排序过程的递归树（快速排序类似，只不过不是均分）。本实验希望达到的目的是：第二层左树和右树同时计算，第三层四个子树同时计算，.....*（最大并行线程数量取决于计算机CPU性能）*

最大递归树深度和最大线程数满足：`ThreadNum =2^TreeDepth`

在归并或快速排序标程的基础上，增加`tree_dep`参数，判别树深度。若`tree_dep<最大树深度`，则将其中一个递归函数放到新开的线程中进行，直到达到最大树深度，停止裂变。

4. 比较性能

4.0 比较前提：数据量和伪随机数组

- `MAXSIZE = 1E4`
- `MAXNUM = 1E4`,
- 每次由`srand(0) rand()%MAXNUM`生成，因此是固定的随机数。

4.1 纵向比较：排序算法不变时，运行时间和线程数的关系

- `TOTAL_TREEDEPTH = 5`

理论上，运行时间和线程数应该成反比例。但实际上并非如此。当线程数较少（如从1->2->4）时，呈较好的反比关系，而线程数目增加至8-16时，运行时间可能不会明显缩短。可能有以下原因：

- 计算机执行程序时CPU正忙，开启线程数目不稳定，可能会变少。反过来，若CPU较为空闲，反比关系保持较好。
- 线程数过多后，线程间切换时间增加。
- 每个线程分配到的数据所需时间不同，存在“木桶效应”，即一个线程等待另一个线程的情况。

Serial_Time	Parallel_Time	Ratio(S/P)	Thread_Num
3394	3408	1.0	1
3394	1662	2.0	2
3394	1168	2.9	4
3394	1562	2.2	8
3394	1489	2.3	16

归并排序 运行示例1*线程多反而增加运行时间*

Serial_Time/us	Parallel_Time/us	Ratio(S/P)	Thread_Num
2286	2386	1.0	1
2286	1294	1.8	2
2286	929	2.5	4
2286	898	2.5	6
2286	875	2.6	14

归并排序 运行示例2 *最大线程数取决于CPU空闲状况*

4.2 横向比较：相同线程数，平均性能：归并排序>快速排序

理论上，归并排序用时应小于快速排序，原因是：归并排序稳定，而快排不稳定。意思是归并排序的每一个线程内的数据是均分的，而快排基准（pivot）位置不稳定，线程数据量不同。所以快排并行时一个线程等另一个线程的情况更普遍，造成了时间的浪费。

实践符合理论推断。

Serial_Time/us	Parallel_Time/us	Ratio(S/P)	Thread_Num
2881	2839	1.0	1
2881	1981	1.5	2
2881	1281	2.2	4
2881	1172	2.5	8
2881	942	3.1	16

归并排序 运行示例3 *较好的单调递减关系*

Serial_Time	Parallel_Time	Ratio(S/P)	Thread_Num
2144	2216	1.0	1
2144	2173	1.0	2
2144	1525	1.4	4
2144	1466	1.5	8
2144	1110	1.9	15

快速排序 运行示例1

三、实验代码解释

为了避免重复，源文件中的串行排序代码均省略

1. 宏定义、全局变量及函数声明

```
#define MAXSIZE 1E4 //数组大小
#define MAXNUM (int)1E4 //数组元素最大值
#define TOTAL_DEPTH 5 //递归树最大深度
/*方便向pthread_create传参数而创建的结构体*/
struct parray {
    int *pBase;//数组的指针
    int tree_dep;//树深度
    int begin;//要排序的起始位置
```

```
    int end;//要排序的末尾位置
};
typedef struct parray parray;

int threadnum,Tree;//记录总线程数和本轮最大树深度

void InitArray(parray *a);//生成随机数组a，大小为MAXSIZE,取值区间[0,MAXNUM-1]
int isIdentical(parray a,parray b);//判断串行排序和并行排序后结果是否相同
void Serial_MergeSort(int a[],int begin, int end);//串行归并排序
void Parallel_MergeSort(int a[],int tree_dep,int begin,int end);//并行归并排序
void Merge(int a[],int begin,int end);//归并排序要用到的合并
void *work1(parray *a);//归并排序的线程函数
void Serial_QuickSort(int a[],int begin, int end);//串行快速排序
void Parallel_QuickSort(int a[],int tree_dep,int begin,int end);//并行快速排序
void *work2(parray *a);//快速排序的线程函数
void swap(int v[],int i,int j);
```

2. 初始化数组

```
void InitArray(struct parray *a) {
    a->pBase = (int *)malloc(sizeof(int)*MAXSIZE);
    a->begin = 0,a->end = MAXSIZE,a->tree_dep = 0;//begin is inclusive, end is
exclusive
    srand(0);
    for (int i=0;i<MAXSIZE;i++) {
        a->pBase[i] = rand() % MAXNUM;
    }
}
```

3. 并行归并排序

```
void *work1(parray *a) {
    threadnum++;//记录总线程数
    Parallel_MergeSort(a->pBase,a->tree_dep,a->begin,a->end);
    return NULL;
}
void Parallel_MergeSort(int a[],int tree_dep,int begin,int end)
{
    if (end-begin < 2) return;
    int mid = (begin+end)/2;
    tree_dep++;
    if (tree_dep <= Tree) {//比较当前树深度和最大树深度，若比最大深度小则可以继续裂变
        pthread_t tid;
        parray temp = {a,tree_dep,begin,mid};//方便传参数
        pthread_create(&tid,NULL,(void*)work1,&temp);//将begin-mid段数据放入新线程中
        Parallel_MergeSort(a,tree_dep,mid,end);
        pthread_join(tid,NULL);
    }
```

```
    else {
        Parallel_MergeSort(a,tree_dep,begin,mid);//若达到最大深度则不再开启新线程
        Parallel_MergeSort(a,tree_dep,mid,end);
    }
    Merge(a,begin,end);
}
```

4. 并行快速排序

```
void *work2(parray *a) {
    threadnum++;
    Parallel_QuickSort(a->pBase,a->tree_dep,a->begin,a->end);
    return NULL;
}
void Parallel_QuickSort(int v[],int tree_dep,int left,int right)
{
    int i,last;
    if (left >= right) return;
    /*寻找基准last*/
    swap(v,left,(left+right)/2);//把最中间的元素换到最左边
    last = left; //定位 比划分元素小 的最后一个元素的位置，便于结束本轮快排时将 划分元素 插入last的位置
    for (i = left+1;i <= right;i++) {
        if (v[i] < v[left]) swap(v,++last,i); //将小于划分元素的数移到左边，并标记最后一个小的数的位置
    }
    swap(v,left,last);
    tree_dep++;
    if (tree_dep <= Tree) {
        pthread_t tid;//比较当前树深度和最大树深度，若比最大深度小则可以继续裂变
        parray temp = {v,tree_dep,left,last-1};
        pthread_create(&tid,NULL,(void*)work2,&temp);//将begin-mid段数据放入新线程中
        Parallel_QuickSort(v,tree_dep,last+1,right);
        pthread_join(tid,NULL);
    }
    else {
        Parallel_QuickSort(v,tree_dep,left,last-1);//若达到最大深度则不再开启新线程
        Parallel_QuickSort(v,tree_dep,last+1,right);
    }
}
```

5. 记录运行时间并比较性能 *(以归并排序为例)*

```
    struct timeval start, end;
    gettimeofday(&start,NULL);
    Serial_MergeSort(a.pBase,a.begin,a.end);
    gettimeofday(&end,NULL);
```

```
    long Serial_time = (end.tv_sec-start.tv_sec)*1E6+(end.tv_usec-start.tv_usec);//串行排序时间
    printf("Serial_Time/us\tParallel_Time/us\tRatio(S/P)\tThread_Num\n");//表格表头
    for (;Tree<TOTAL_DEPTH;Tree++) {//本轮最大树深度从0-TOTAL_DEPTH递增
        InitArray(&b);
        gettimeofday(&start,NULL);
        Parallel_MergeSort(b.pBase,b.tree_dep,b.begin,b.end);
        gettimeofday(&end,NULL);
        long Parallel_time = (end.tv_sec-start.tv_sec)*1E6+(end.tv_usec-start.tv_usec);//并行排序时间
        if (isIdentical(a,b)) {//两种排序结果相同
            printf("%ld\t\t%ld\t\t\t%.1f\t\t%d\n",Serial_time,Parallel_time,(int)Serial_time*1.0/(int)Parallel_time,threadnum+1);
            threadnum = 0;//重置线程计数器
        }
    }
```

6. 其他细节

```
/*归并排序中的合并*/
void Merge(int a[],int begin,int end) {
    int mid = (begin+end)/2;
    int i = begin,j = mid,k = i;
    int *b = (int *)malloc(sizeof(int)*(end-begin));
    while (i<mid && j<end) {
        if (a[j] < a[i]) b[k++] = a[j++];
        else b[k++] = a[i++];
    }
    while (i<mid) b[k++] = a[i++];
    while (j<end) b[k++] = a[j++];

    for (int i=begin;i<end;i++) {
        a[i] = b[i];
    }
}
/*判断串行排序和并行排序后结果是否相同*/
int isIdentical(parray a,parray b)
{
    int same = 1;
    for (int i=0;i<MAXSIZE;i++) {
        if (a.pBase[i] != b.pBase[i]) {
            same = 0;
            break;
        }
    }
    return same;
}
```

四、实验体会和心得

1. 最初的想法非常简单：在main函数中手动拆成两段，用两个线程并行。这种程序可行，但比较“傻瓜”，无法根据数据量自动增加线程量。随后又想用迭代的归并排序做，但是发现程序十分不稳定，有时候可以排序正确，但大部分时候排序错误。这可能和线程开启的速度和变量计算速度不一致有关。而且这种方法不能迁移至快速排序的并行，最终放弃。
2. 以上两种原始的想法失败后，我感觉到应该在递归形式的算法中生成新线程。但是我不会判断总线程数。使用递归树深度决定线程数的方法是我学习他人的思路得到的。这种方法适用于归并和快排，可迁移性较好。
3. 程序里还有一个bug：`#define MAXSIZE 1E4`，若设置成`1E5`或更大，会报错段错误。问题原因不明。
4. 总的来说，我学习了pthread库的基本函数的用法，初步尝试了并行计算，进一步巩固了递归思想和两种排序算法。