

**Harbin Institute of Technology at Weihai**

**并行计算 课程报告**

学 号： 181110315

姓 名： 王少博

设计题目：基于Pthread的Timsort并行设计与实现

院 系： 计算机科学与技术暨软件学院

成 绩：

教师评语：

哈尔滨工业大学（威海）

二零二零年十一月

# 绪论

## 项目背景

TimSort是结合了归并排序（合并排序）和插入排序（插入排序）而得出的排序算法，它在现实中有很好的效率.Tim Peters在2002年设计了该算法并在Python中使用是Python中list.sort的默认实现）。该算法找到数据中已经排好序的块 分区，每一个分区叫一个run，然后按规则合并这些run.Pyhton自从2.3版本以后一直采用Timsort算法排序，现在Java SE7和Android也采用Timsort算法对数组排序。它还被 Java SE7, Android platform, GNU Octave, 谷歌浏览器, 和 Swift 用于对非原始类型的数组排序。

## 项目意义

现实中的大多数据通常是有部分已经排好序的，Timsort利用了这一特点.Timsort排序的输入的单位不是一个个单独的数字，而是一个个的分区。其中每一个分区叫一个run（图1）。针对这个run序列，每次拿一个run出来进行归并。每次归并会将两个run合并成一个run。每个run最少要有2个元素.Timesor按照升序和降序划分出每个run：run如果是是升序的，那么运行中的后一元素要大于或等于前一元素（…）;如果run是严格降序的，即运行中的前一元素大于后一元素（），需要将run中的元素翻转（这里注意降序的部分必须是“严格”降序才能进行翻转。因为TimSort的一个重要目标是保持稳定性的稳定性。如果在这种情况下进行翻转这个算法就不稳定）。

## Pthread的优势和与MPI OpenMP的比较

1. pthread与openmp：

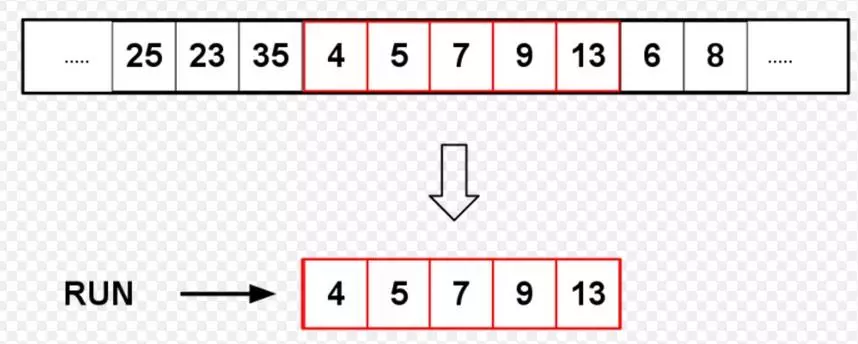
pthread在程序启动时创建一束线程，将工作分配到线程上。然而，这种方法需要相当多的线程指定代码，而且不能保证能够随着可用处理器的数量而合理地进行扩充。OpenMP不需要指定数量,在有循环的地方加上代码，修改设置文件极客。OpenMP 非常方便，因为它不会将软件锁定在事先设定的线程数量中，但是相对的查错更难也更麻烦。

1. pthread和mpi：

首先mpi是基于分布式内存系统，而openmp和pthread基于共享内存系统；也就是说mpi之间的数据共享需要通过消息传递，因为mpi同步的程序属于不同的进程，甚至不同的主机上的不同进程。相反由于openmp和pthread共享内存，不同线程之间的数据就无须传递，直接传送指针就行。同时mpi不同主机之间的进程协调工作需要安装mpi软件（例如mpich）来完成。在openmp和pthread之间的区别主要在编译的方式上，openmp的编译需要添加编译器预处理指令#pragma，创建线程等后续工作要编译器来完成。而pthread就是一个库，所有的并行线程创建都需要我们自己完成算法具体分析。根据以上性质，最终我是用了操作起来比较容易地pthread库。

### 最小运行复杂度

运行是已经排好序的一块分区.RUN可能会有不同的长度，Timesort根据运行的长度来选择排序的策略。例如如果运行的长度小于某一个值，则会选择插入排序算法来排序。运行的最小长度（minrun）取决于数组的大小。当数组元素少于64个时，那么运行的最小长度便是数组的长度，这是Timsort用插入排序算法来排序。当数组元素大于等于63时，对于较大的阵列中，从32到65的范围内选择一个称为minrun的数，使得阵列的大小除以最小游程大小，等于或略小于2的幂。最终的算法只需要数组大小的6个最高有效位，如果剩余的位被设置，则添加一个，并将该结果用作minrun。该算法适用于所有情况，包括数组大小小于64的情况。



### 运行长度的优化

优化运行的长度是指当运行的长度小于minrun时，为了使这样的运行的长度达到minrun的长度，会从数组中选择合适的元素插入运行中。这样做使大部分的运行的长度达到均衡，有助于后面运行的合并操作。

### 归并运行

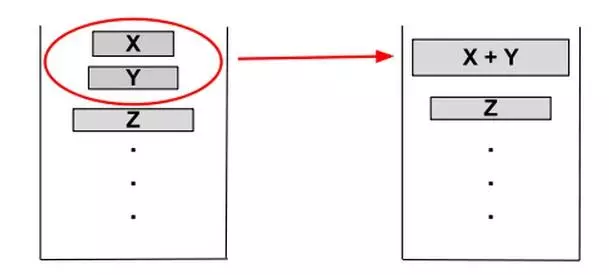
划分run和优化run长度以后，然后就是对各个run进行合并。合并run的原则是run合并的技术要保证有最高的效率。当Timsort算法找到一个run时，会将该run在数组中的起始位置和运行的长度放入栈中，然后根据先前放入栈中的运行决定是否该合并run. Timsort不会合并在栈中不连续的运行（Timsort不会合并非连续运行，因为这样做会导致三条跑道的共同要素相对于中跑来说是失序的。）

Timsort会合并在栈中2个连续的run. 代表栈最上方的3个游程的长度，当同时不满足下面2个条件是，这两个运行会被合并，直到同时满足下面2个条件，则合并结束：

（1）

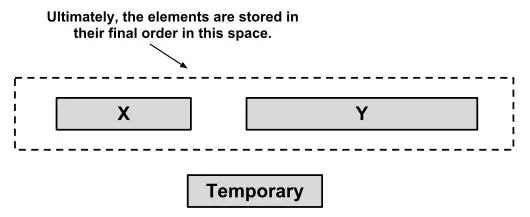
（2）

例如：。如果，那么合并为一个新的运行，然后入栈重复上述步骤，直到同时满足上述2个条件当合并结束后，Timsort会继续找下一运行，然后找到以后入栈，重复上述步骤，及每次运行入栈都会检查是否需要合并2个运行。

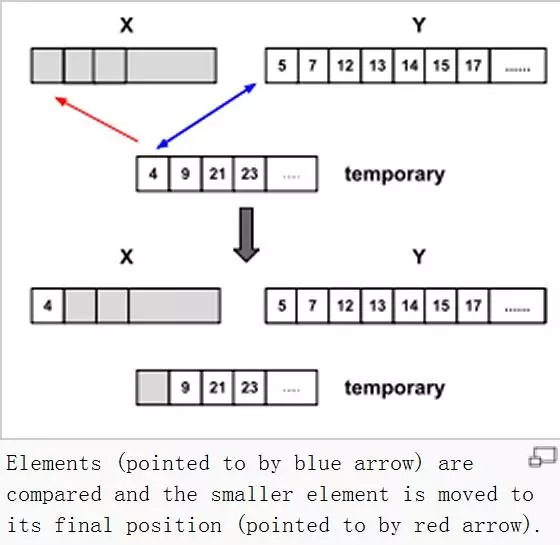
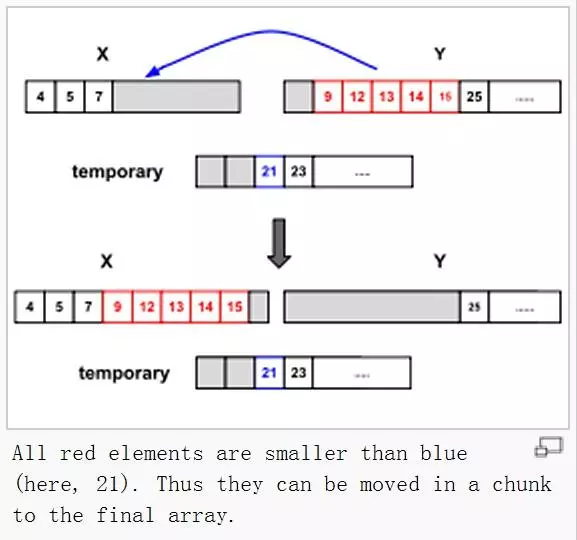


### 归并运行

归并2个相邻的运行需要临时存储空闲，临时存储空间的大小是2个运行中较小的运行的大小. Timsort算法先将较小的运行复制到这个临时存储空间，然后用原先存储这2个运行的空间来存储合并后的运行。



简单的合并算法是用简单插入算法，依次从左到右或从右到左比较，然后合并2个运行。为了提高效率，Timsort用二分插入算法（二进制合并排序）。先用二分查找算法/折半查找算法（binary search）找到插入的位置，然后在插入。

例如，我们要将A和B这2个run合并，且A是较小的run。因为A和B已经分别是排好序的，二分查找会找到乙的第一个元素在一个中何处插入。同样，A的最后一个元素找到在乙的何处插入，找到以后，B在这个元素之后的元素就不需要比较了。这种查找可能在随机数中效率不会很高，但是在其他情况下有很高的效率。

# 串行算法分析与实现

## 二分插入排序的设计

### 二分插入排序的原理

由于在直接插入排序过程中，待插入数据左边的序列总是有序的，针对有序序列，就可以用二分法去插入数据了，也就是二分入排序法。适用于数据量比较大的情况。

二分插入排序的算法思想是：

* 计算 0 ~ i-1 的中间点，用 i 索引处的元素与中间值进行比较，如果 i 索引处的元素大，说明要插入的这个元素应该在中间值和刚加入i索引之间，反之，就是在刚开始的位置 到中间值的位置，这样很简单的完成了折半；
* 在相应的半个范围里面找插入的位置时，不断的用上一步缩小范围，不停的折半，范围依次缩小为 1/2 1/4 1/8 .......快速的确定出第 i 个元素要插在什么地方；
* 确定位置之后，将整个序列后移，并将元素插入到相应位置。

### 二分插入排序的实现

这里由于电脑CPU限制，我们决定在n<100的时候使用二分插入排序来处理。其中插入排序的代码实现如下：

1. **void** insert\_sort(**int** a[], **int** n) {
2. **int** i, j, temp, low, high, m;
3. **for** (i = 1; i < n; ++i) {
4. temp = a[i];
5. low = 0;
6. high = i - 1;
7. **while** (low <= high) {
8. m = (low + high) / 2;
9. **if** (temp < a[m]) {
10. high = m - 1;
11. } **else** {
12. low = m + 1;
13. }
14. }
15. **for** (j = i - 1; j > high; j--) {
16. a[j + 1] = a[j];
17. }
18. a[high + 1] = temp;
19. }//for
21. **for** (i = 0; i < n; ++i) {
22. printf("%d ", a[i]);
23. }
24. }

## 归并排序

归并排序，是创建在归并操作上的一种有效的排序算法。算法是采用分治法（Divide and Conquer）的一个非常典型的应用，且各层分治递归可以同时进行。归并排序思路简单，速度仅次于快速排序，为稳定排序算法，一般用于对总体无序，但是各子项相对有序的数列。

### 归并排序算法思想

归并排序是用分治思想，分治模式在每一层递归上有三个步骤：

* 分解（Divide）：将n个元素分成个含n/2个元素的子序列。
* 解决（Conquer）：用合并排序法对两个子序列递归的排序。
* 合并（Combine）：合并两个已排序的子序列已得到排序结果。

### 实现逻辑

1. **迭代法**
   1. 申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列
   2. 设定两个指针，最初位置分别为两个已经排序序列的起始位置
   3. 比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置
   4. 重复步骤(c)直到某一指针到达序列尾
   5. 将另一序列剩下的所有元素直接复制到合并序列尾
2. **递归法**
   1. 将序列每相邻两个数字进行归并操作，形成floor(n/2)个序列，排序后每个序列包含两个元素
   2. 将上述序列再次归并，形成floor(n/4)个序列，每个序列包含四个元素
   3. 重复步骤(b)，直到所有元素排序完毕

### 算法实现

以下是非并行化地mergesort代码实现

1. **void** naive\_merge(**int** array[], **int** start, **int** mid, **int** end) {
2. **int** LeftLen = mid - start + 1;
3. **int** RightLen = end - mid;
4. **int** \*Left = (**int** \*) malloc(**sizeof**(**int**) \* LeftLen);
5. **int** \*Right = (**int** \*) malloc(**sizeof**(**int**) \* RightLen);
6. **int** i = 0, j = 0, k = 0, f = 0;
7. **for** (i = 0, k = start; i < LeftLen; i++, k++) {
8. Left[i] = array[k];
9. }
10. **for** (i = 0, k = mid + 1; i < RightLen; i++, k++) {
11. Right[i] = array[k];
12. }
13. **for** (k = start, i = 0, j = 0; i < LeftLen & j < RightLen; k++) {
14. **if** (Left[i] <= Right[j]) {
15. array[k] = Left[i];
16. i++;
17. }
18. **if** (Left[i] > Right[j]) {
19. array[k] = Right[j];
20. j++;
21. }
22. }
23. **if** (i < LeftLen) {
24. **for** (f = i; f < LeftLen; f++, k++) {
25. array[k] = Left[f];
26. }
27. }
28. **if** (j < RightLen) {
29. **for** (f = j; f < RightLen; f++, k++) {
30. array[k] = Right[f];
31. }
32. }
33. }
35. **void** naive\_MergeSort(**int** array[], **int** start, **int** end) {
36. **if** (start < end) {
37. **int** mid = (start + end) / 2;
38. naive\_MergeSort(array, start, mid);
39. naive\_MergeSort(array, mid + 1, end);
40. naive\_merge(array, start, mid, end);
41. }
42. }

# 并行算法设计与实现

## 二分插入排序

这一部分和上面情况一样，此时无需进行更改。

## 并行化的归并排序

### 并行化归并排序的原理

1. 通常我们使用的是二分治的归并排序，即将数据每次一分为二，之后通过递归的方式将这两个数据块内部变的有序，之后再通过一个O(n)的Merge算法就可以做到排序。
2. 首先获得要创建的进程/线程数量k，最好是个。这样方便数据的划分。 之后将n个数据等分为k份的时候，将这些任务分配给不同的core，进行并行化排序。我们可以借助pthread库来实现这个想法。

### 线程数据数据结构定义

我们归并的时候划分出两个子线程需要对其数据进行单独的封装处理。实现部分定义如下：

1. **typedef** **struct** ThreadData {
2. **int** start, end;//待排序的范围
3. **int** \*array;//数组指针
4. **int** length;//数组的长度
5. } ThreadData;

### 代码实现

以下是并行化地mergesort代码实现

1. //方便pthread\_create调用，和mergesort之间做一个接口
2. **void** p\_mergeSort(ThreadData \*data) {
3. mergeSort(data->array, data->length, data->start, data->end);
4. }
6. // 归并排序中的合并算法
7. **void** merge(**int** array[], **int** length, **int** left, **int** m, **int** right) {
8. **int** \*temp = (**int** \*) malloc(**sizeof**(**int**) \* length);
9. **int** i, j, k;
10. i = left;
11. j = m + 1;
12. k = 0;
14. **for** (; i <= m && j <= right;) {
15. **if** (array[i] < array[j]) {
16. temp[k++] = array[i++];
17. } **else** {
18. temp[k++] = array[j++];
19. }
20. }
21. **while** (i <= m) {
22. temp[k++] = array[i++];
23. }
24. **while** (j <= m) {
25. temp[k++] = array[j++];
26. }
27. k = 0;
28. **for** (i = left; i <= right; i++) {
29. array[i] = temp[k++];
30. }
31. free(temp);
32. }
34. // 归并排序
35. **void** mergeSort(**int** array[], **int** length, **int** start, **int** end) {
36. **if** (start < end) {
37. **int** i;
38. pthread\_t tid1, tid2;
39. ThreadData data\_left, data\_right;
40. i = (end + start) / 2;
42. data\_left.array = array;
43. data\_left.length = length;
44. data\_left.start = start;
45. data\_left.end = i;
47. data\_right.array = array;
48. data\_right.length = length;
49. data\_right.start = i + 1;
50. data\_right.end = end;
52. // 启动2个线程分别对左右两部分进行mergeSort
53. pthread\_create(&tid1, NULL, (**void** \*) p\_mergeSort, (**void** \*) &data\_left);
54. pthread\_create(&tid2, NULL, (**void** \*) p\_mergeSort, (**void** \*) &data\_right);
55. pthread\_join(tid1, NULL);
56. pthread\_join(tid2, NULL);
57. // join等待完成后，再merge
58. merge(array, length, start, i, end);
59. }
60. }

# 性能比较与分析

对串行算法和并行算法进行比较，并得出结论。

## 串行算法性能分析：

根据信息学理论，在平均情况下，比较排序不会比更快。由于Timsort算法利用了现实中大多数数据中会有一些排好序的区，所以Timsort会比对于随机数没有可用利用的排好序的区，Timsort时间复杂度会是。下表是Timsort与其他比较排序算法时间复杂度（time complexity）的比较。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Timsort | Mergesort | Quicksort | Insertionsort | Selectionsort |
| 最好情况 |  |  |  |  |  |
| 平均情况 |  |  |  |  |  |
| 最差情况 |  |  |  |  |  |

## Timsort的优势：

是一个稳定的自适应的迭代mergesort，在部分排序的数组上运行时，需要远远少于的比较，同时在随机数组上运行时性能堪比传统的mergesort。像所有合适的合并类型一样，这种类型是稳定的，并运行时间（最坏情况）。在最坏的情况下，这种类型需要临时存储空间来存放n / 2个对象引用; 在最好的情况下，它只需要很小的恒定空间。

大体是说，Timsort是稳定的算法，当前排序的数组中已经有排序好的数，它的时间复杂度会小于。与其他合并排序一样，Timesrot是稳定的排序算法，最坏时间复杂在最坏情况下，Timsort算法需要的临时空间是n / 2，在最好情况下，它只需要一个很小的临时存储空间

这里我们并没有使用快速排序作为Timsort中代替mergesort的方法，而是采用了mergesort。其中一个很大的原因就是mergesort稳定，而quicksort不稳定。这里给出关于快速排序和归并排序的一个对比：

1) 快速排序和归并排序都是基于比较的排序，在基于二分治的情况下，他们的时间复杂度下界均为。

2) 他们都是使用动态规划方法进行编写的排序算法，但是快速排序和归并排序的分治有着一些区别。归并排序在并行化之后，仍然需要的存储空间，而快速排序并行化之后，需要额外的O(n)存储空间，性质发生改变，所以选了归并排序进行并行化。因为在相同的空间复杂度之下，归并排序更加稳定，所以我们使用归并排序。

## 串行和并行算法性能分析比较

### 测试代码

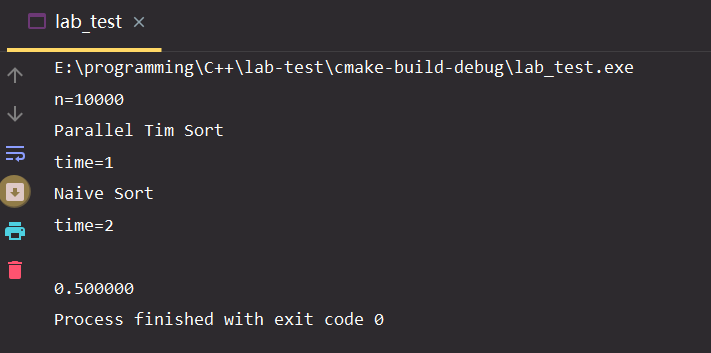
1. **int** main() {
2. pthread\_t tid;
3. **int** ret;
4. **int** n = 100000;
5. printf("n=%d\n", n);
6. **int** \*array = generateRandomArray(n, 0, 100);
7. **int** a[n];
8. **for** (**int** i = 0; i < n; i++) a[i] = array[i];
10. printf("Parallel Tim Sort\n");
12. ThreadData data;
13. data.array = array;
14. data.start = 0;
15. data.end = n - 1;
16. data.length = n;

19. **time\_t** start, end;
20. **if** (n <= 100) {
21. insert\_sort(array, n);
22. end = time(NULL);
23. } **else** {
24. start = time(NULL);//or time(&start);
26. ret = pthread\_create(&tid, NULL, (**void** \*) p\_mergeSort, (**void** \*) &data);
27. **if** (ret) {
28. printf("Thread Create Error\n");
29. exit(0);
30. }
32. pthread\_join(tid, NULL);
33. end = time(NULL);
34. }
36. **long** **long** t1 = end - start;
37. printf("time=%lld\n", t1);
39. //    for (int i = 0; i < n; i++) {
40. //        printf("%d ", array[i]);
41. //    }
42. printf("\n");
43. printf("Naive Sort\n");

46. start = time(NULL);//or time(&start);
47. mergeSort(a, n, 0, n - 1);
48. end = time(NULL);
49. **long** **long** t2 = end - start;
50. printf("time=%lld\n", t2);
52. //    for (int i = 0; i < n; i++) {
53. //        printf("%d ", a[i]);
54. //    }
55. printf("\n");
56. printf("%lf", (**double**) t1 / (t2 + 1e-7));
57. free(array);
58. **return** 0;
60. }

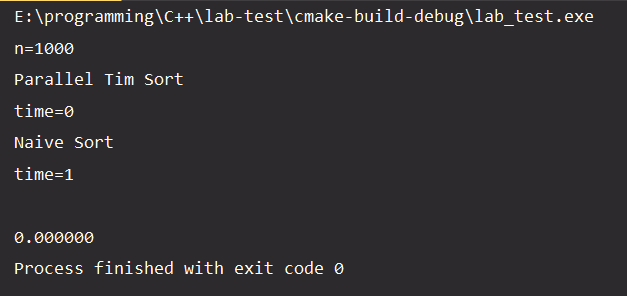
### 测试结果如下

1. 测试1：取规模n=10000得到测试结果如下。时间比为2

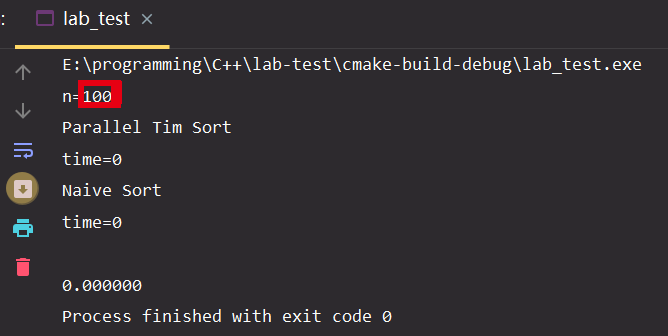


1. 测试2：取规模n=1000得到测试结果如下。

可以看到tim sort的并行化版本时间已经极快。



1. 测试3：取规模n=100得到测试结果如下



可以看出两者排序都极快。在这里为了防止除零报错，加上了小项1e-7防止出错。