

SortAlgorithm-程序设计与算法大作业

文件内容说明

文件	说明
SortAlgorithm	项目工程
README.md	说明文档
README.pdf	说明文档PDF
SortAlgorithm.exe	运行程序文件
SortAlgorithmBase10.exe	基数排序使用基数为10的程序文件
仓库地址.txt	仓库地址
算法大作业答辩.pptx	答辩PPT文件
算法耗时测试记录.xlsx	算法耗时测试统计记录
img	算法运行截图和耗时统计分布图
Sort-Java	Java项目代码

使用说明

1.选择参加排序的算法(默认选择排序不参与)

-a <选择的算法> // 写相应的算法代号：选择排序代号1，归并排序代号2，快速排序代号3，归并排序代号4，基数排序代号5

例如全部算法参与排序：

```
E:\sch\课程\程序设计与算法>SortAlgorithm.exe -a 12345
```

2.数组操作

(1) 选择数组长度（默认10000）

-n <数组长度> //n后面写数组长度

例如数组长度为100000：

```
E:\sch\课程\程序设计与算法>SortAlgorithm.exe -n 100000
```

(2) 选择初始数组状态（默认乱序）

-d r //乱序

-d a //升序

-d d //降序

例如选数组初始状态为升序：

```
E:\sch\课程\程序设计与算法>SortAlgorithm.exe -d a
```

3.大数版本

-h //启用大数

4.多线程版本

-t <线程数>

5.启用输出

-o

常规版本

一、实现这五种排序并分析

1. 选择排序

基本思想

每一趟排序待排序序列中选出最小的元素，(先假设第一个元素是最小的，遍历后面的元素，不断记录更小的元素下标，即可得到一趟遍历中最小元素)然后与该未排序序列的第一个元素交换位置，确定该元素的位置，如此重复直到所有元素排序完成。

时间复杂度

一趟排序的比较次数是 $O(n)$ ，重复 $n-1$ 趟，所以时间复杂度是 $O(n^2)$ 。

主要代码

```
template<typename T>
void SelectSort(T* arr, int n)           //选择排序入口
{
    for (int i = 0; i < n; ++i)
    {
        for (int j = i + 1; j < n; ++j)
        {
            if (arr[j] < arr[i])
                std::swap(arr[i], arr[j]);
        }
    }
}
```

2. 归并排序

基本思想

先将最初长度为 n 的待排序序列从中间划分成两个子序列，再递归划分左右两个子序列，形成 n 个长度为1的有序子序列。再将相邻的两个子序列两两合并，形成有序序列，如此重复最后得到长度为 n 的有序序列。

一趟合并的过程：借助一个临时数组，比较待合并的左右两个序列的第一个元素大小，将较小的赋值给临时数组，直到两个子序列的元素都完成合并。

时间复杂度

归并排序的时间复杂度为归并的趟数与每一趟归并的时间的复杂度的乘积。子算法merge合并的时间复杂度为 $O(n)$,趟数为 $\log_2 n$,故算法复杂度为 $O(n\log n)$ 。

主要代码

```
template<typename T>
void MergeSort(T* arr, int n) //归并排序入口
{
    if (n <= 1) return;
    int mid = n / 2;
    MergeSort(arr, mid); // [0, mid - 1]
    MergeSort(arr + mid, n - mid); // [mid, n - 1]
    T* temp = new T[n + 1];
    int p = 0, q = mid;
    for (int i = 0; i < n; ++i)
    {
        if (p >= mid) temp[i] = arr[q++];
        else if (q >= n) temp[i] = arr[p++];
        else temp[i] = arr[q] < arr[p] ? arr[q++] : arr[p++];
    }
    memcpy(arr, temp, n * sizeof(T));
    delete[] temp;
}
```

3. 快速排序

基本思想

从当前待排序序列中选择最后一个元素作为主元，把小于等于主元的所有元素移动到主元前边，大于等于基准元素的所有元素都移动到主元后边，确定主元的最终位置，然后分别对前后两个序列递归上述过程，直到所有元素完成排序。

时间复杂度

快排一趟排序确定一个元素的位置，时间复杂度为 $O(n)$ ，在平均情况下递归趟数为 $\log_2 n$ ，算法复杂度为 $O(n\log n)$ 。在序列已经有序的情况下，需要重复 $n-1$ 趟才能确定所有元素的位置，时间复杂度为 $O(n^2)$ 。

主要代码

```
template<typename T>
void QuickSort(T* arr, int n) //快速排序入口
{
    if (n <= 1) return;
    int t = (rand() << 15) | rand();
    t %= n;
    std::swap(arr[0], arr[t]);
    int p = 1, q = n; //pq指向最后一个小于等于arr[0]的后一个
    while (p < q)
    {
        while (p < q && arr[p] <= arr[0]) ++p;
        while (q == n || (p < q && arr[q] > arr[0])) --q;
        if (p < q) std::swap(arr[p], arr[q]);
    }
}
```

```

    }
    std::swap(arr[0], arr[p - 1]);
    QuickSort(arr, p - 1);
    QuickSort(arr + p, n - p);
}

```

4. 希尔排序

基本思想

首先确定元素间隔数gap，然后将所有位置相隔gap的元素视为一个子序列，对各个子序列进行排序；然后缩小间隔数，并重新对形成的子序列进行排序，直到gap = 1。

这里初始gap设为n/2，每次缩小一半。子序列内部采用冒泡排序。

时间复杂度

希尔排序的排序趟数为 $\log_2 n$ ；当子序列分的越多时，子序列内的元素就越少，元素比较交换次数就越少，而当子序列的个数减少时，整个序列接近有序，子序列的元素虽然变多但元素之间的比较交换次数没有随之变多。所以一般情况下，认为希尔排序的时间复杂度在 $O(n \log n)$ 与 $O(n^2)$ 之间。

主要代码

```

template<typename T>
void ShellSort(T* arr, int n)                //希尔排序入口
{
    int gap, i, j, flag;
    //外循环以不同的gap值对形成的序列进行排序，直到gap=1
    for(gap = n / 2; gap >= 1; gap = gap / 2)
    {
        //对各个子序列进行冒泡排序
        do{
            flag = 0;
            for(i = 0; i < n - gap; i++)
            {
                j = i + gap;
                if(arr[i] > arr[j])
                {
                    std::swap(arr[i], arr[j]);
                    flag = 1;
                }
            }
        } while (flag);
    }
}

```

5. 基数排序

基本思想

把参加排序的序列中的元素按第1位的值进行排序（最右边一位为第1位），然后再按第2位的值进行排序.....最后按第d位的值进行排序。每一趟排序过程中若有元素的位值相同，则它们之间仍保留前一趟排序的次序。

具体实现借助链表，假设参加排序的序列是d位r进制，不足d位的元素在前面补上0。

时间复杂度

基数排序的趟数是d趟，每趟要把n个元素依次分配到r个分队，再集合到总队，每趟花费的时间为 $O(n+r)$ 。所以基数排序总的时间复杂度为 $O(d(n+r))$ 。

主要代码

```
void Sort::RadixSort(int* arr, int n) //普通基数排序
{
    static const int BASE = 1500;

    int min = arr[0], max = arr[0];
    for (int i = 1; i < n; ++i)
    {
        min = arr[i] < min ? arr[i] : min;
        max = arr[i] > max ? arr[i] : max;
    }
    for (int i = 0; i < n; ++i) arr[i] -= min;
    max -= min;

    auto head = new Node<int>[BASE + 1];
    auto tail = new Node<int>*[BASE + 1];
    auto node = new Node<int>[n + 1];
    for (int i = 1; i < max; i *= BASE)
    {
        memset(head, 0, sizeof(Node<int>) * BASE);
        memset(tail, 0, sizeof(Node<int>*) * BASE);
        for (int j = 0; j < n; ++j)
        {
            int t = arr[j] / i % BASE;
            node[j] = {arr[j], NULL};
            if (head[t].next == NULL) head[t].next = &node[j];
            if (tail[t] != NULL) tail[t]->next = &node[j];
            tail[t] = &node[j];
        }
        int p = 0;
        for (int j = 0; j < BASE; ++j)
        {
            auto t = &head[j];
            while (t = t->next)
            {
                arr[p++] = t->val;
            }
            head[j].next = NULL;
            tail[j] = NULL;
        }
    }
    delete[] head;
```

```
delete[] tail;
delete[] node;

for (int i = 0; i < n; ++i) arr[i] += min;
}
```

二、分析其不同规模的输入下单机性能变化情况

1. 初始数组次序对排序性能影响

分析五种排序算法在初始数组次序分别为乱序，升序，降序情况下的性能变化情况，这里固定数组长度为100000。

实验结果：如下表所示，单位ms

原始次序	乱序	升序	降序
选择排序	77544	7819	119805
归并排序	36	27	28
快速排序	27	11	15
希尔排序	134	2	26
基数排序	9	6	6

性能分析

(1)当原始数组为升序时，五种排序算法的运行时间较乱序均有降低，其中希尔排序性能提升最为明显，选择排序其次。

因为原始序列的次序与排序结果一致，排序算法的交换次数均减少，运行时间也随之减少；其中希尔排序各个子序列已经升序，比较次数也减少，大大减少了排序时间，性能提升最为明显。

(2) 当原始数组为降序时，选择排序的时间上升，其余四种排序时间均降低。

这是因为选择排序每次假设待排序序列的第一个元素为最小值，增加了交换次数。而其余四种排序的交换次数均减少，所以运行时间也随之减低。

(3) 从实验结果可知，当数组长度较长时，元素之间的交换次数对运行时间的影响占主导地位，所以原始数组有序（正序或逆序）情况下归并、快排、希尔、和基数排序耗时均减少。

2. 不同规模的输入下性能分析

生成不同长度 $n=1000, n=10000, n=100000, n=1000000$ 的数组，计时，分别统计这五种排序算法在不同数组长度下随机实验10次的平均运行时间，这里以乱序数组为例。

实验结果：如下表所示，单位ms

输入规模	n=1000	n=10000	n=100000	n=1000000
选择排序	7	757	77544	--
归并排序	0	4	36	353
快速排序	0	2	27	283
希尔排序	0	8	134	2264
基数排序	0	0	9	147

性能分析

从实验结果来看，这五种排序耗时从小到大为：基数排序<归并排序和快速排序（一个数量级）<希尔排序<选择排序

- (1) 快速排序和归并排序在不同输入规模下耗时相近，属于同一个数量级。
- (2) 希尔排序的耗时 在归并/快排和选择排序之间，但更接近归并/快排，所以时间复杂度更接近 $O(n\log n)$ 。
- (3) 在输入规模在1000及以下时，五种排序算法用时差别不明显；而输入规模每增大10倍，选择排序耗时增大近似100倍，所以随着输入规模的增大，时间差异越发明显，在输入规模达到百万数量级时，很难运行出结果。

3. 总结

排序算法	平均时间	稳定性	备注
选择排序	$O(n^2)$	不稳定	n小时较好
归并排序	$O(n\log n)$	稳定	n大时较好
快速排序	$O(n\log n)$	不稳定	n大时较好
希尔排序	$O(n\log n)$	不稳定	更接近 $O(n\log n)$
基数排序	$O(d(n+r))$	稳定	d 是位数， r是进制

大数版本

大数排序实现

```
// 详见代码
class HugeInt;
class HugeIntDataProvider;
class Sort;
```

大数的存储方式

以10000为基准，使用int[20]的数组存储一个大数，每一个int存储0~9999的一个数，高位数字表示成10000的n次进位，这样仅仅使用长度为20的int数组即可表示10的100次方这样大的数字。

自定义数据类型，实现其比较方法，此处列举大数对象的定义：

```
class HugeInt
{
public:
    static const int ARR_SIZE = 20;
    static const int BASE = 100000;

    static HugeInt Zero();
    static HugeInt Rand();

    HugeInt();
    //HugeInt(const char* str);
    ~HugeInt();

    bool absEqualTo(const HugeInt&) const;
    bool absLessThan(const HugeInt&) const;
    bool operator==(const HugeInt&) const;
    bool operator!=(const HugeInt&) const;
    bool operator<(const HugeInt&) const;
    bool operator<=(const HugeInt&) const;
    bool operator>(const HugeInt&) const;
    bool operator>=(const HugeInt&) const;
    int getData(int i) const;
    const char* toString();

private:
    int sig; //1: 非负数 0: 负数
    int* data;
    char* str;

    void makeString();
};
```

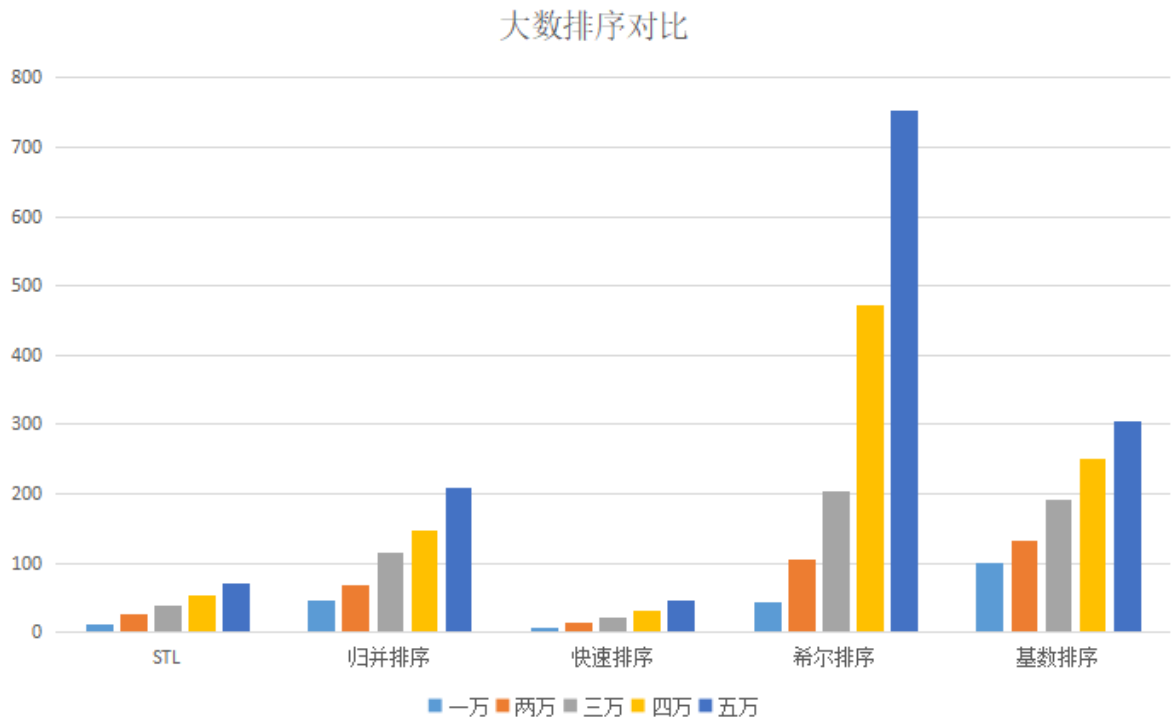
对大数排序的性能分析

各算法在一万到五万数据量下，排序耗时统计（单位ms）：

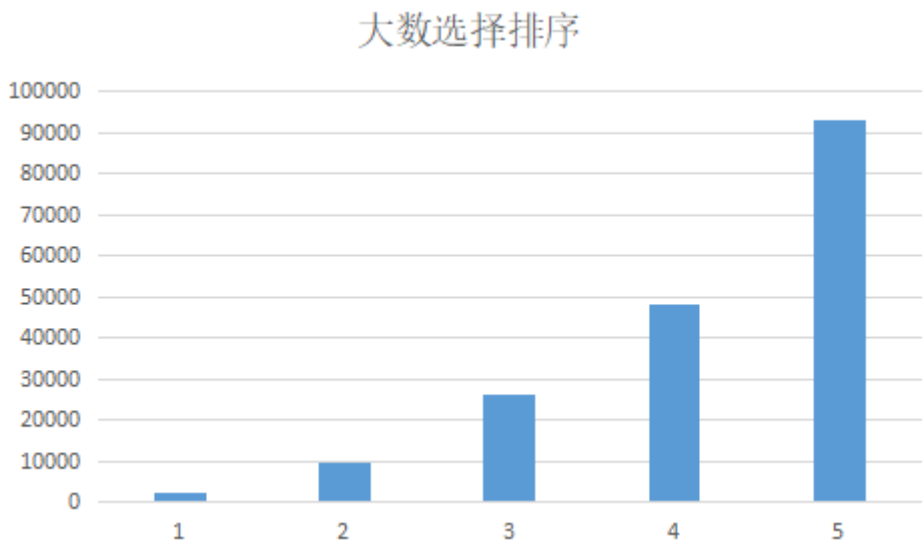
	STL	归并排序	快速排序	希尔排序	基数排序	选择排序
--	-----	------	------	------	------	------

	STL	归并排序	快速排序	希尔排序	基数排序	选择排序
一万	12	45	7	43	101	2312
两万	26	68	15	105	132	9714
三万	38	116	21	203	191	26351
四万	54	146	31	471	250	48497
五万	70	208	46	754	304	93056

各个版本的大数排序耗时分布如下图，数据量从一万到五万。



选择排序由于耗时太久，单独进行展示，数据量从一万到五万。



分布式排序

分布式排序实现

```
// 详见代码
class SortAlgorithm;
```

分布式排序算法分析

对于小规模数据量的数据进行排序和运算，其需要的内存容量，核心性能，和运算时间；单机的性能就可以满足。

对于超大规模量的数据进行排序和运算，往往单机的性能无法满足其需要；需要多台机器分别对数据的某一部分进行排序，最后进行数据的合并排序来完成大体量数据的排序运算。

真实环境中的分布式运算即将任务进行切片，分发给不同的核心进行计算；可以在单机中使用多线程来模拟分布式排序运算；最终都需要将多个线程运算的结果进行汇总排序。

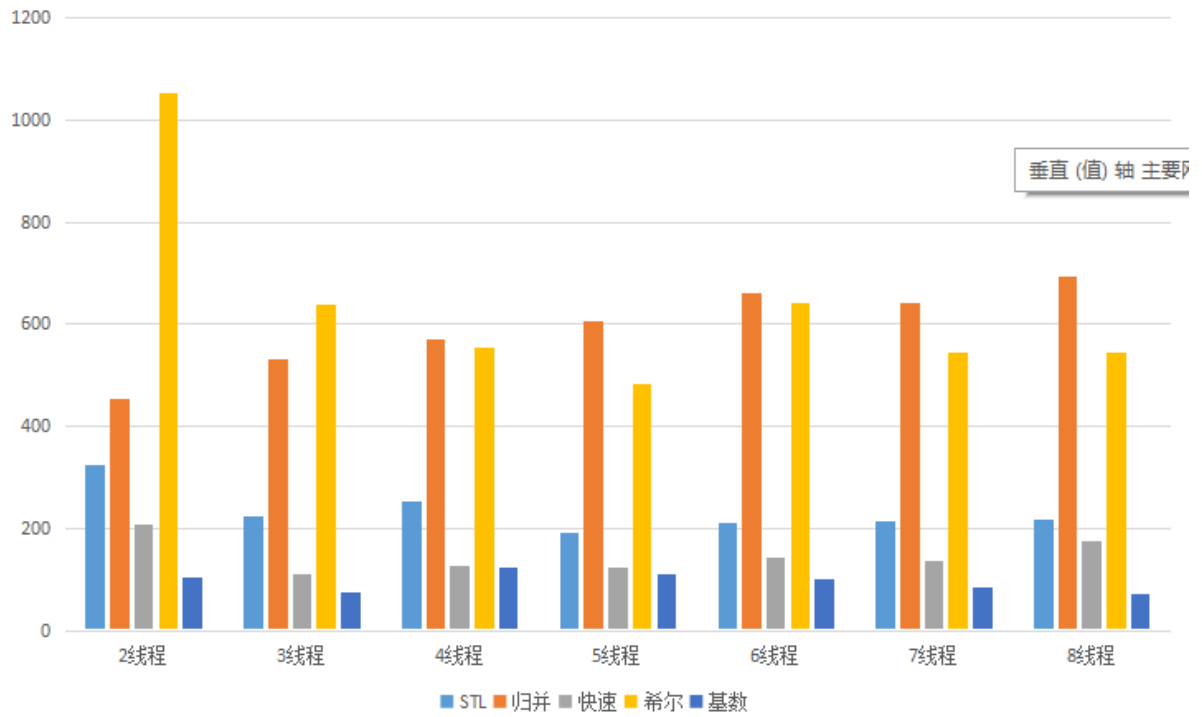
分布式算法耗时

数据量为一百万时，耗时统计（单位ms）：

	STL	归并	快速	希尔	基数
2线程	323	455	208	1054	103
3线程	223	532	111	639	75
4线程	254	570	127	554	125
5线程	193	607	122	482	112
6线程	211	661	144	643	100
7线程	214	643	135	544	84
8线程	218	692	176	546	73

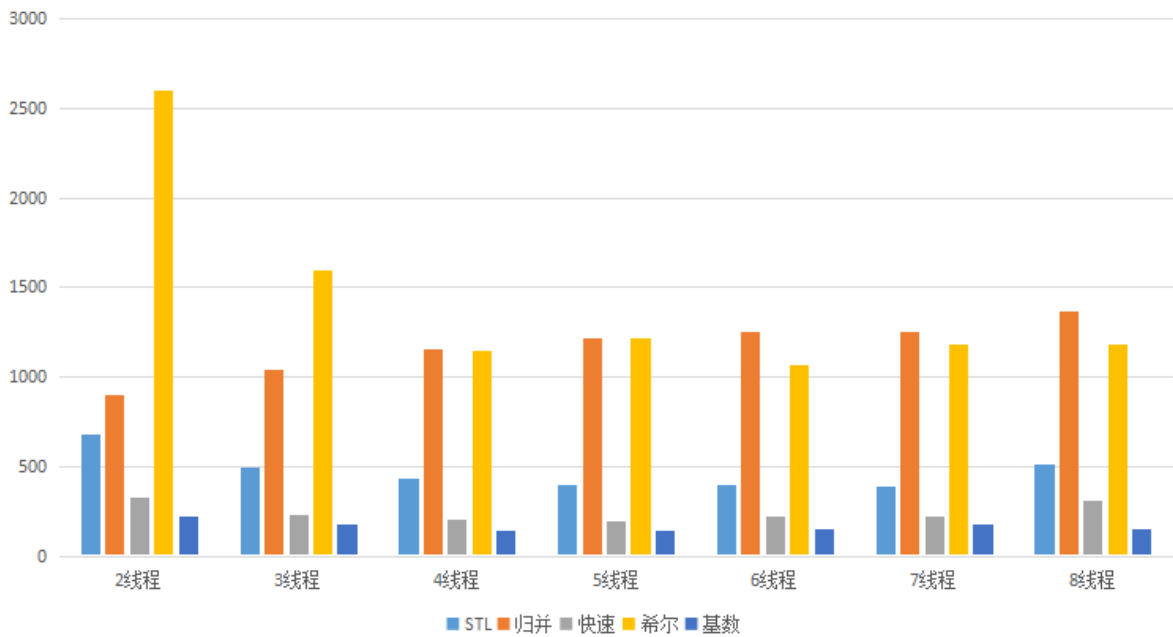
源数据为一百万条时，各算法耗时分布：

分布式排序一百万



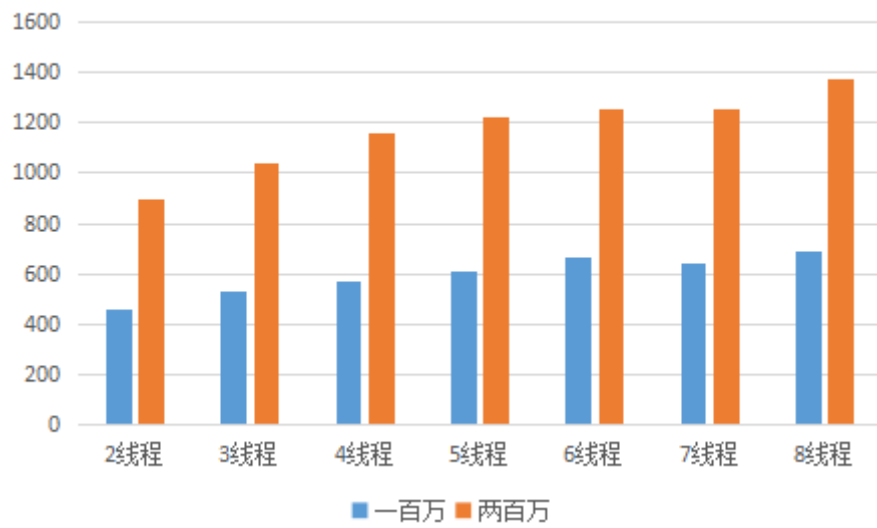
源数据为两百万条时，各算法耗时分布：

分布式排序两百万

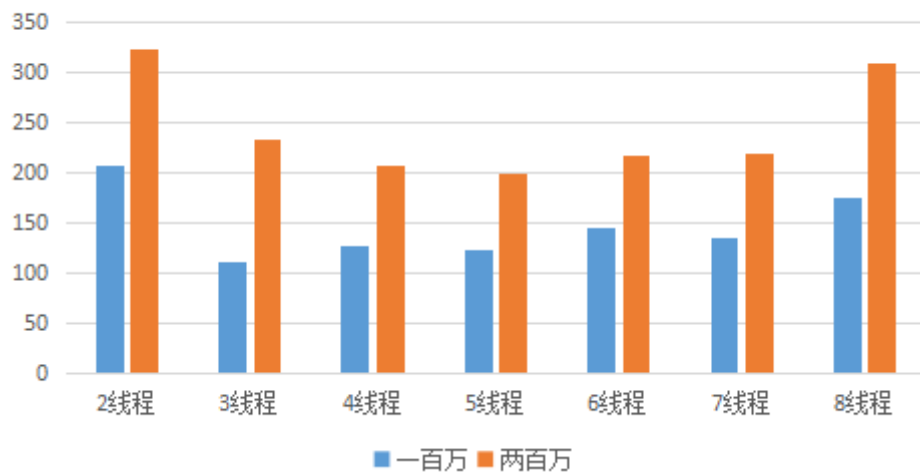


各个版本排序方法分别在不同数据量的情况下，排序耗时分布如图：

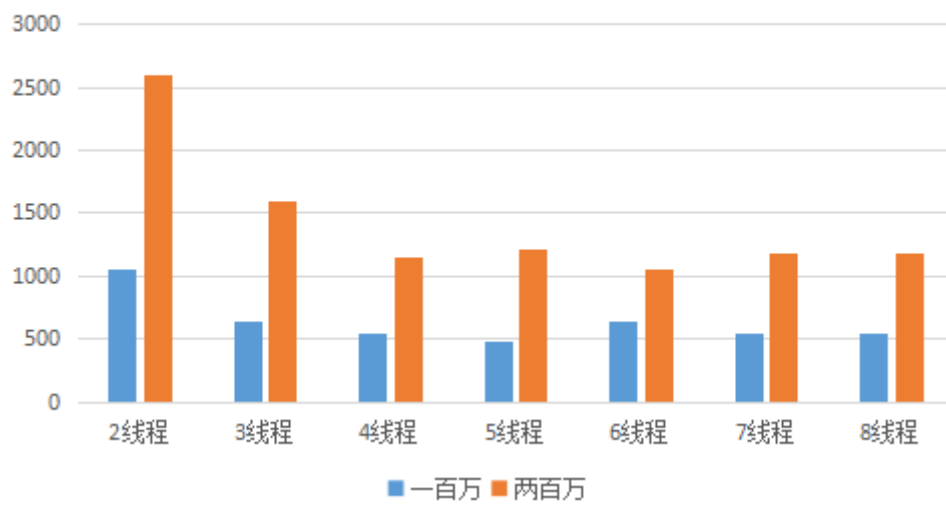
归并排序数据量对比



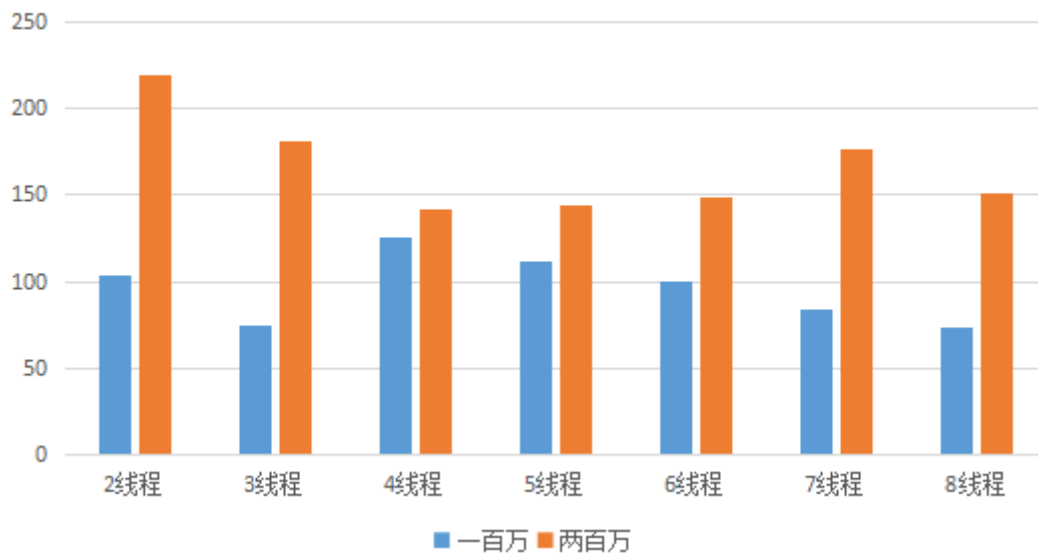
快速排序数据量对比



希尔排序数据量对比



基数排序数据量对比



选择排序由于耗时太长，一百万4线程耗时高达949394ms，没有参加本次排序耗时对比。

测试环境：CPU核心数为4，内存16G

Java语言实现常规和分布式排序（简易版）

常规版排序算法实现

```
// 详见代码
public class MergeSort;
public class QuickSort;
public class RadixSort;
public class SelectionSort;
public class ShellSort;
```

直接在IDE中运行调试，其中常规排序算法在IDE调试控制台打印结果：

```
选择排序：
9, 12, 45, 65, 78, 98, 100, 132,
归并排序：
9, 12, 45, 65, 78, 98, 100, 132,
快速排序：
9, 12, 45, 65, 78, 98, 100, 132,
希尔排序：
9, 12, 45, 65, 78, 98, 100, 132,
基数排序：
9, 12, 45, 65, 78, 98, 100, 132,
Process finished with exit code 0
```

分布式排序实现

```
public class DateOperator; // 生产数据和汇总后写入文件
public class DistributedSort; // 多线程调用排序算法进行排序
```

使用多线程模拟进行分布式排序

多线程处理十五万条数据，分10个线程，共150万条数据

使用CountDownLatch，限定在所有线程执行结束后再进行汇总和写入文件

排序后的结果存储在distributedSort.txt中。

```
distributedSort.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
0 1 4 4 7 8 10 11 12 16 19 19 19 20 20 22 24 25 25 25
26 28 29 29 30 31 36 36 39 42 42 42 43 45 45 46 48 49 51 51
52 52 53 53 54 55 56 58 59 60 63 63 63 64 64 65 66 70 70 72
72 73 74 75 75 76 77 77 80 80 80 80 81 82 85 88 90 91 92 95
96 100 100 101 102 102 105 105 105 107 107 107 108 110 111 111 112 112 116 116
118 120 122 122 125 125 126 126 126 128 129 129 132 134 135 135 136 141 142 145
147 148 152 153 157 158 160 162 162 163 163 163 164 165 166 166 168 169 175 176
178 178 182 186 186 187 188 189 189 190 192 193 193 194 195 195 196 197 198 198
199 205 205 206 206 208 208 209 209 210 212 215 216 218 218 220 220 222 225 226
226 227 230 231 233 233 234 235 242 242 243 243 244 246 250 251 252 252 253 253
256 259 260 260 262 263 263 263 267 272 278 282 282 283 286 286 288 289 289
289 289 290 291 293 293 296 296 300 306 307 308 309 310 311 311 311 312 313 315
315 319 322 323 324 324 326 327 327 328 329 331 338 340 340 341 341 342 344 345
347 347 347 348 352 353 354 354 355 356 357 358 359 359 359 359 360 361 362 364
366 367 368 369 370 374 374 375 375 377 378 379 380 380 381 381 381 381 382 382
382 384 384 386 386 388 388 390 390 390 391 392 394 394 396 396 396 396 397 399
399 399 400 400 404 404 405 405 410 413 415 418 419 419 421 422 423 424 424 428
431 431 431 434 435 435 436 442 444 445 448 448 448 450 452 455 455 458 459 460
463 468 470 472 476 478 480 480 484 484 489 489 494 500 501 501 503 506 507 509
509 513 513 514 514 517 518 520 521 524 524 524 525 525 526 526 526 527 532 532
534 536 537 540 540 542 542 547 548 548 548 549 552 553 555 556 557 558 565 567
569 570 572 573 574 575 576 579 579 583 583 584 586 586 588 588 588 589 590 591
591 594 594 595 595 597 598 598 601 603 604 604 605 606 610 612 613 614 616 617
617 617 621 621 624 626 626 627 628 631 632 635 640 642 642 643 646 649 650 651
652 652 653 654 656 657 658 659 659 660 663 666 666 666 667 668 671 673 673 673
674 675 676 677 677 678 679 679 679 680 681 681 682 684 686 686 687 688 689 692
694 694 695 697 697 699 701 704 704 704 707 708 709 709 709 712 712 713 713 714
715 717 717 721 724 727 730 733 738 738 739 739 741 742 742 744 745 746 749 752
753 753 753 755 756 758 760 760 760 761 767 767 770 772 777 779 779 787 789 789
790 791 792 793 793 794 795 795 798 799 800 801 803 804 807 808 808 811 811 812
814 815 815 816 821 822 822 823 823 824 824 827 828 829 829 830 836 836 837 837
838 841 843 845 846 847 848 849 851 851 854 855 864 864 864 864 866 866 867 869
871 872 873 874 875 875 875 876 877 878 879 880 881 881 883 883 884 884 885 885
888 888 891 892 895 896 898 898 900 900 904 907 909 914 915 915 917 917 919 921
922 924 924 926 930 931 931 938 940 940 941 942 943 943 946 947 951 951 954 955
955 957 957 957 958 961 961 963 964 965 965 965 966 966 967 968 972 972 973 974
974 976 977 981 982 983 983 983 984 985 986 986 987 990 990 993 996 996 997 998
999 1001 1001 1004 1004 1004 1006 1007 1010 1012 1013 1015 1016 1017 1019 1020 1022 1023 1026 1027
1028 1028 1030 1031 1033 1033 1034 1035 1036 1036 1038 1039 1040 1043 1044 1045 1047 1047 1047
1050 1052 1053 1054 1054 1061 1062 1062 1065 1067 1067 1067 1069 1071 1072 1072 1075 1078 1080
1082 1083 1083 1090 1090 1090 1092 1096 1097 1099 1100 1101 1103 1105 1105 1106 1106 1111 1113 1113
1115 1118 1119 1120 1125 1127 1132 1132 1132 1133 1134 1134 1136 1137 1141 1142 1144 1144 1144 1147
1148 1153 1162 1163 1165 1166 1167 1168 1169 1170 1171 1173 1174 1174 1178 1180 1180 1185 1185 1186
1187 1187 1188 1192 1192 1195 1196 1197 1198 1201 1203 1207 1207 1210 1211 1213 1215 1216 1217 1220
1221 1221 1224 1224 1226 1227 1228 1231 1234 1234 1235 1240 1241 1241 1242 1242 1245 1249 1251 1253 1254
1256 1259 1259 1264 1267 1268 1269 1269 1271 1272 1273 1275 1277 1277 1277 1278 1279 1279 1279 1279
1279 1281 1282 1283 1284 1284 1285 1286 1287 1288 1289 1289 1291 1293 1295 1296 1296 1297 1298 1299
1299 1299 1301 1303 1304 1304 1307 1308 1309 1310 1310 1312 1312 1313 1314 1315 1315 1316 1319 1320
1321 1321 1323 1325 1326 1326 1328 1328 1331 1332 1333 1335 1335 1339 1340 
```