
南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

**并行计算技术课程文档**

---

# 双调排序算法证明及结合 Argminmax 的双调排序算法单机并行编程实验报告

---

邵铭恺 2120240747

专业：计算机技术

指导教师：王刚

2025 年 6 月 14 日

# 目录

# 一、 绪论

## (一) 简介

本文档是南开大学 2025 春《并行计算技术》课程的“双调排序算法分析及单机并行编程”实验报告，作者为邵铭恺，学号为 2120240747。文章将针对作业要求，对双调排序算法进行证明，并结合期末选题与双调排序算法使用多种方法完成它的并行化实验。

## (二) 作业要求

1. 双调排序算法证明（50 分）：对序列 $s = a_0, a_1, a_2, a_3, ..., a_{n-1}$，其中 $n$ 为 2 的幂。证明：当 $s$ 满足以下任意一种情况，双调分裂所得到的子序列 $s_1$ 和 $s_2$ 均为双调序列，且 $s_1$ 中元素均小于 $s_2$ 中元素：

a) $s$ 是分界点为中心的升序－降序双调序列。

b) $s$ 是分界点任意的升序－降序双调序列。

2. 结合期末研究报告选题，完成单机并行编程实验。

## (三) 与期末选题的关系

我期末研究报告的选题是 Mindspore 中 Argminmax 算子的并行化研究。开题报告中，讲到该算子包含两种情况，第一种情况是在 topk 大小为 1 时，使用 top1 算法计算，遍历一次即可；另一个算法则是按照轴和偏移量取出对应待操作张量后，将其进行排序，然后直接取前 k 大的元素。由此可见，TopK>1 时，算法的主要计算时间瓶颈在排序的一步上，由此我们可以从排序算法并行化研究入手。Mindspore 中使用的是 C 标准库进行排序，故在本次实验中，结合本次实验的主题，我们使用双调排序及其并行化完成实验，并与 Mindspore 原有实现做性能对比，以此证明我们并行化方法的有效性。

# 二、 双调排序算法证明

证明：若 $s$ 是分界点为中心的升序－降序双调序列，假设分界点为 $k = \frac{n}{2} - 1$，则有：

$$a_0 < a_1 < ... < a_{\frac{n}{2}-1}, a_{\frac{n}{2}-1} > a_{\frac{n}{2}} > a_{\frac{n}{2}+1} > ... > a_n$$

对其进行双调分裂操作，该分裂操作将 $s$ 分为了 $s_1 = \{min(a_0, a_{\frac{n}{2}}), min(a_1, a_{\frac{n}{2}+1}), ...\}$ 和 $s_2 = \{max(a_0, a_{\frac{n}{2}}), max(a_1, a_{\frac{n}{2}+1}), ...\}$，显然任取 $i \in [0, \frac{n}{2})$，有 $s_{1i} < s_{2i}$。

因为 $a_{\frac{n}{2}-1}$ 是 $s_1$ 中的最大值，而 $a_n$ 是 $s_2$ 中的最小值，若 $max(a_{\frac{n}{2}-1}, a_n) = a_n$，则 $s_1$ 中没有任意值大于 $s_2$ 中的任意值。若 $max(a_{\frac{n}{2}-1}, a_n) = a_{\frac{n}{2}-1}$，相当于 $s_1$ 中的最大值被切换成了 $s_2$ 中的最小值，则对 $s_1$ 中的当前最大值（实际是原始的第二大值）和 $s_2$ 中的当前最小值（实际是原始的第二小值）继续进行这一过程，若 $s_2$ 的这个值更大，则说明 $s_1$ 中没有任意值大于 $s_2$ 中的任意值；若直到 0 时，仍旧是 $s_1$ 中的最小值更大，那说明 $s_2$ 中所有的值都不如 $s_1$ 中大，经过交换之后则是 $s_2$ 中的所有值都比 $s_1$ 大。

由于原序列是双调序列，在进行双调分裂后，假定 $s_1$ 从 $s_2$ 处交换来了 $k$ 个元素，那么显然是后 $k$ 个元素被换掉（因为 $s_1$ 单调递增而 $s_2$ 单调递减），那么 $s_1[\frac{n}{2} - k]$ 此时就是 $s_1$ 的极大值点，且 $s_1$ 仅存在这一个极大值点。故 $s_1$ 是双调序列，同理 $s_2$ 也是双调序列。

若 $s$ 是分界点任意的升序－降序双调序列，假设分界点为 $k$，则有：

$$a_0 < a_1 < ... < a_k, a_{k+1} > a_{k+2} > ... > a_n$$

则 $a_k$ 是原序列的极大值点，考虑 $k$ 在 $s_1$ 的情况：

因为 $a_k$ 是原序列的极大值点，则 $a_k$ 是 $s_1$ 中的最大值，且 $s_2$ 单调递减，则 $a_n$ 是 $s_2$ 中的最小值。因此 $max(a_{\frac{n}{2}-1}, a_n) = a_{\frac{n}{2}-1}$，我们逆序考虑切换动作，从 $a_{\frac{n}{2}-1}$ 切换到 $a_k$。当 $a_k$ 的切换动作发生时，相当于对于剩下的序列而言，又把 $s_1$ 中的最大值和 $s_2$ 中的最小值进行切换了，这时候待切换的序列就回到了上面的这种情况。同理，若 $k$ 在 $s_2$，也满足这种情况。

由于原序列双调，在交换后，$s_1$ 和 $s_2$ 的极值点也只会有一个，仍旧考虑 $k$ 在 $s_1$ 的情况，此情况下 $s_1$ 先增后减，$s_2$ 单调递减。因为 $s_2$ 递减，而序列整体是双调序列，则进行切换操作后 $s_1$ 原有的极大值一定会被换走，新的极大值来源于 $s_2$，而 $s_2$ 单调递减，故 $s_1$ 只存在一个峰，是双调序列。对于 $s_2$，因为它单调递减而且序列整体双调，所以它的极大值一定小于整体最大值，而整体最大值位于 $s_1$，达到整体最大值之前 $s_1$ 是单调递增的，所以 $s_2$ 也只存在一个峰；而 $s_2$ 一开始是单调递减的，当它的值开始小于 $s_1$ 中的值时会开始频繁地数据交换，且整个序列是双调序列，$s_1$ 中没有谷，所以 $s_2$ 中也只有一个谷。所以分裂后的序列也是双调序列。

当 $k$ 在 $s_2$ 时，同理可得上面两条结论。原命题得证。

# 三、 实验设计

本次实验主题为双调排序算法的并行化，实验内容为使用多种并行方法对双调排序算法进行编程实验。实验中，首先对双调排序串行算法使用 C++ 进行实现，包括两种实现方式：

1. 易于理解与实现的递归形式；
2. 易于任务划分和并行化的迭代形式；

实验中完成串行算法的两种实现后，将对双调排序算法进行分析与任务划分。然后分别对两种形式的算法进行多种方式的并行化，包括 SIMD，多线程的多种方式（包含 std::thread，openmp，以及调用其他开源库的线程池与任务队列），对该算法的效率进行实验，将串行算法和 STL 中的 sort 函数（快速排序）作为 baseline，与其进行对比并展示并行化效果，最后根据上述结果开发一个融合方法，来进行双调排序的优化。

实验中将对算法并行化的具体方法，以及这些方法对性能产生的影响进行分析，包括编程策略对执行时间的影响、不同方法带来的额外开销、体系结构问题等方面，来完成一次完整的实验。

# 四、 算法实现

本节中对算法的具体实现方法进行描述。

## (一) 串行算法实现

双调排序算法由两个部分组成，一部分是双调分裂，另一部分是双调合并。

本节中对串行算法的不同实现方式进行介绍，并简要对比这些方式的差异。

### 1. 递归形式

递归形式的算法实现较为简单，将双调分裂和双调合并的逻辑分别实现之后，只需要连续进行递归，再递归地进行双调合并就可以。算法的代码如下：

```cpp
void bitonic_merge(vector<int>& arr, int low, int cnt, bool dir) {
    if (cnt > 1) {
        int k = cnt / 2;
        for (int i = low; i < low + k; i++) {
            if (dir == (arr[i] > arr[i + k])) {
                swap(arr[i], arr[i + k]);
            }
        }
        bitonic_merge(arr, low, k, dir);
        bitonic_merge(arr, low + k, k, dir);
    }
}
void bitonic_sort(vector<int>& arr, int low, int cnt, bool dir) {
    if (cnt > 1) {
        int k = cnt / 2;
        bitonic_sort(arr, low, k, true);
        bitonic_sort(arr, low + k, k, false);
        bitonic_merge(arr, low, cnt, dir);
    }
}
void bitonic_sort(vector<int>& arr) {  // 调用时
    padding(arr, arr.size());
    bitonic_sort(arr, 0, arr.size(), true);
}
```

我们容易发现，递归过程中，前一次和后一次递归不存在数据依赖，因此可以将每一次递归分给一个线程，并在一个阶段执行完之后进行线程同步，然后再到下一个阶段继续多线程。

递归方式的双调排序实现简单，但是并行化性能不算好，因为需要在递归调用中频繁创建与销毁线程，开销较大。后续实现均使用与其等价但更好并行化的迭代形式。

使用 OMP 简单对递归形式的算法进行了并行化，几乎没有效果。

### 2. 迭代形式

在迭代方式的实现中，我们将上文提到的递归形式进行了展开，进行自底向上的迭代算法流程:

```cpp
void bitonic_sort_iterative(vector<int>& arr) {
    padding(arr, arr.size());
    int n = arr.size();
    for (int size = 2; size <= n; size <<= 1) {
        for (int stride = size >> 1; stride > 0; stride >>= 1) {
            for (int i = 0; i < n; i++) {
                int j = i ^ stride;
                if (j > i) {
                    bool dir = ((i & size) == 0);
```

```
                    if ((arr[i] > arr[j]) == dir) {
                        swap(arr[i], arr[j]);
                    }
                }
            }
        }
    }
}
```

外层循环中，我们首先对一个无序序列进行双调分裂，控制每次遍历的序列大小按 2 的幂增长；中层循环中，我们则对外层循环的每一次双调分裂做元素的比较/交换，使用 $stride$ 来控制元素的比较/交换间隔，让它按照 2 的幂减小；内层循环中，我们对整个序列进行遍历，使用按位异或运算来计算它需要比较的目标，并且用一个 $j > i$ 来避免重复交换。由于 size 一定是 $2^k$，所以 $i\&size$ 的值只由 $i$ 的第 $k$ 位决定，当 $i$ 位于前一半时，$i\&size = 0$，代表升序；反之代表降序。

由此可见，外部的两个循环分别代表双调排序的两个阶段，而最内层循环则每次都要整体遍历数组，并且数组的前后之间不存在数据依赖。所以可以对内层循环进行并行操作，包括后面提到的几种方法（SIMD、多线程），这比递归算法更易于并行编程。

## (二) 并行算法实现

### 1. SIMD

在迭代算法的最内层循环的遍历过程中，我们可以应用向量指令的思想，在待比较/交换的序列大小大于向量寄存器大小时，可以使用向量指令一次交换多个元素，来加速最内层迭代中的 swap 过程。根据我的电脑实际情况，我使用了 AVX2 指令集进行向量指令编程，实现了内存对齐和不对齐两种情况下的比较交换操作：（以内存不对齐为例）

```
// 单轮SIMD比较-交换，处理8个int
bool dir = ((i & size) == 0);
__m256i va = _mm256_loadu_si256((__m256i*)&arr[i + j]);
__m256i vb = _mm256_loadu_si256((__m256i*)&arr[i + j + stride]);
__m256i cmp = dir ? _mm256_cmpgt_epi32(va, vb) : _mm256_cmpgt_epi32(
    vb, va);
__m256i va_new = _mm256_blendv_epi8(va, vb, cmp);
__m256i vb_new = _mm256_blendv_epi8(vb, va, cmp);
_mm256_storeu_si256((__m256i*)&arr[i + j], va_new);
_mm256_storeu_si256((__m256i*)&arr[i + j + stride], vb_new);
```

首先将其加载到向量寄存器中，然后根据 dir 参数判定比较方向，生成一个掩码寄存器，然后将掩码寄存器作用到开始的两个向量上，使其交换需要交换的数据，最后使用存储指令将向量寄存器中的内容存储回内存。在对齐版本中，将对应的 load 和 store 指令改为对齐版本（_mm256_load_si256 和 _mm256_store_si256）即可。

在算法实际的执行过程中，我们需要对迭代算法做一些调整，由于使用了 AVX2，只有在步长是 8 的整数倍的时候才能使用向量指令，所以需要将步长为 8 以下的元素退化到普通算法进行比较。此外，在内存对齐版本中，我们需要确保输入数据被分配为内存对齐的数组。

### 2. 多线程（OMP）

使用 OMP 对递归算法进行了并行化：

```
void bitonic_sort_task(vector<int>& arr, int low, int cnt, bool dir,
    int threshold) {
    if (cnt > 1) {
        int k = cnt / 2;
        if (cnt > threshold) {
            #pragma omp task shared(arr)
            bitonic_sort_task(arr, low, k, true, threshold);
            #pragma omp task shared(arr)
            bitonic_sort_task(arr, low + k, k, false, threshold);
            #pragma omp taskwait
        } else {
            bitonic_sort_task(arr, low, k, true, threshold);
            bitonic_sort_task(arr, low + k, k, false, threshold);
        }
        bitonic_merge(arr, low, cnt, dir);
    }
}
```

为避免太多次创建线程，设置一个阈值，在阈值以下时单线程处理。递归算法并行化时候需要在每一轮递归结束时等待任务完成，该算法效果较差。

使用 OMP 对迭代算法进行优化：

```
void bitonic_sort_iterative_omp(vector<int>& arr) {
    padding(arr, arr.size());
    int n = arr.size();
    for (int size = 2; size <= n; size <<= 1) {
        for (int stride = size >> 1; stride > 0; stride >>= 1) {
            #pragma omp parallel for schedule(static)  // 只需要在这
                里添加omp指示即可
            for (int i = 0; i < n; i++) {
                // 对应处理...
            }
        }
    }
}
```

只需要在最内层循环上执行 omp 指令即可。本任务中，迭代的循环过程中工作量较为均匀，适合使用 static 方式来处理任务。测试时，迭代算法的并行化算法性能更好。

### 3. 多线程（自主编程）

如果不使用 omp，使用自主编程的方式实现多线程自由度更高，并且能够达到更细粒度的调度。在本问题中，由于每一次内层迭代中的任务数量一定，所以我们可以按照线程数计算出每一个线程需要得到的任务，并且均匀地分配给它们，在每一轮循环结束之后等待他们同步即可完成

多线程调度。我们可以使用 pthread 和 std::thread 等方式，在每一次迭代时手动创建线程，并让主线程等待所有线程执行完毕之后再开始下一次迭代。此外，为了减小在数据量小时频繁创建和销毁线程带来的额外开销，可以为多线程添加一个阈值，在当前任务规模大于这个阈值的时候才使用多线程调度：

```
...
int num_threads = std::thread::hardware_concurrency();
for (int size = 2; size <= n; size <<= 1) {
    for (int stride = size >> 1; stride > 0; stride >>= 1) {
        if (n < threshold) {
            // 单线程处理任务....
        } else {
            int batch = (n + num_threads - 1) / num_threads;
            std::vector<pthread_t> threads(num_threads);
            std::vector<pthread_args> args(num_threads);
            for (int t = 0; t < num_threads; t++) {
                args[t] = pthread_args{&arr, t * batch, std::min(n, (
                    t + 1) * batch), size, stride, n};
                pthread_create(&threads[t], nullptr,
                    bitonic_thread_func, &args[t]);  // 线程函数中处理
                    实际计算
            }
            for (int t = 0; t < num_threads; t++) {
                pthread_join(threads[t], nullptr);
            }
        }
    }
}
```

进一步地，如果使用多线程，我们其实也没有必要非要每次循环都创建和销毁线程，可以维护一个线程池，在每一次循环到来时为线程池中的线程分配任务，然后等待任务结束后再进行下一步操作。这里使用开源的BS::thread_pool实现线程池，通过 detach_task 来给线程池中的线程分配任务，然后通过 wait 函数等待任务执行完成：

```
....
for (int size = 2; size <= n; size <<= 1) {
    for (int stride = size >> 1; stride > 0; stride >>= 1) {
        int batch = (n + num_threads - 1) / num_threads;  // 向上取
            整，负载更均衡
        for (int t = 0; t < num_threads; ++t) {
            int start = t * batch;
            int end = min(n, (t + 1) * batch);
            pool.detach_task([&, start, end, size, stride]() {
                ....
            });
        }
```

```
12          pool.wait();
13  ....
```

此种方式能够降低线程创建与维护的开销，能够得到更好性能。

#### 4. 多线程与 SIMD 的融合方法

根据以上的研究，我们可以使用多线程方法与 SIMD 融合，完成一个综合性能很强的方法。我在这里使用 AVX2 和 BS::thread_pool 协同工作，最终算法的性能在数据量较大时可以优于 C++ 标准库中提供的快速排序。

## 五、 实验结果

本节中将对以上算法实现的性能数据进行呈现，并简要分析和对比它们之间的性能差距。本次实验中，我们让 n 按照二的幂次方的数量增长，然后测量多个数据规模下的，执行时间长短，使用多次测量，取平均的方式来得到一个稳定的性能。算法对比的 baseline 有三个，一个是上面提到的递归算法，另一个是迭代算法，另一个是 STD 标准库中提供的快速排序函数，也就是 sort 这个函数，通过优化，我们应该可以让我们的算法性能优于 sort 函数。

### (一) 实验数据

在实验过程中，我们对双调排序算法实现的多种策略的性能数据进行了记录，数据如图1所示。

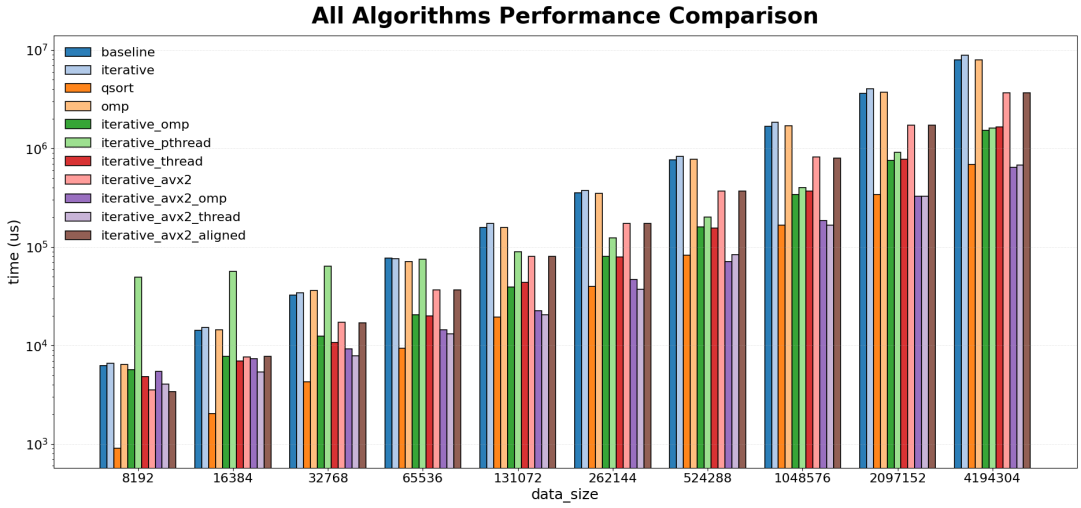


图 1: 所有算法之间的性能数据对比（无编译优化）

图1中，横坐标为问题规模（待排序数组大小），不同颜色的柱状图为不同方法（baseline 为递归方法，iterative 为迭代法，qsort 为 std::sort 快速排序，omp 为递归方法的 omp 多线程，其他的为迭代方法的对应并行化策略方法），纵坐标为执行时间，单位为微秒，后文将对该数据进行分析。

此外，测量了 O2 优化之后的算法性能，数据如图2所示：

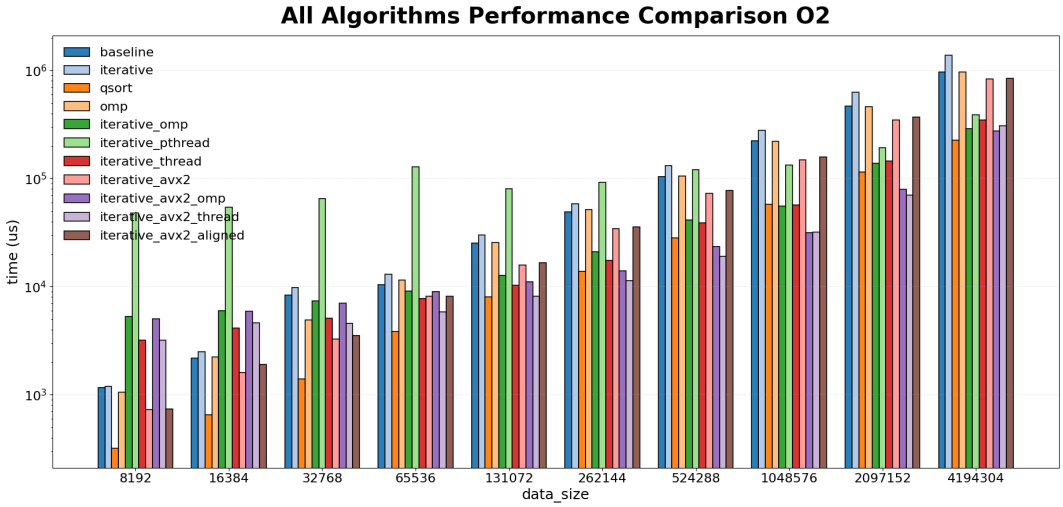


图 2: 所有算法之间的性能数据对比（O2 优化）

## (二) 性能分析

### 1. 串行算法实现方式带来的性能差距

实验数据中，串行算法的数据一共包含三组，分别为 baseline、iterative 和 qsort。观察实验数据可知，串行算法的两种实现中，如果使用默认的编译选项，那么递归算法的性能略优于迭代算法，两种算法的性能均不如快速排序算法。

### 2. SIMD 编程方法带来的性能差距

使用 SIMD 编程的算法（*avx2*）性能比串行算法的性能好很多，基本上是串行算法执行时间的一半，这能够证明能够证明我们 SIMD 方法的有效性。

在对 SIMD 方法进行实验中,对齐和非对齐两种方式(iterative_avx2 和 iterative_avx2_aligned)的性能差距并不明显，或许是因为双调排序本身就是以二的幂次方的大小的数组为基准的一个排序算法，默认内存可能就是对齐的。在测定出来性能数据后，我去观察了对于不同数组大小，初始化数组的内存对齐情况，观察得出在数据量较大（大于 65536）时，初始数组均被初始化为对齐情况，所以二者最后性能差距不大。

### 3. 多线程实现方式带来的性能差距

对递归算法使用多线程时，没有特别明显的优化效果（omp 和串行算法对比）。

对迭代算法使用多线程时，不同编程策略会带来一些性能上的影响，使用 omp 的方式性能较为固定，只能使用系统内部为我们提供的优化方法，与其他方法对比差距不明显；使用 pthread 方式实现的话，在数据量小的时候会对线程频繁进行创建和释放操作，占用系统资源，所以在数据量较小的时候，多线程算法性能较差，但是当数据量较大时，性能与其他算法基本持平。如果使用线程池方式实现的话，在数据量较小的时候，性能是最好的。在数据量中等时，该方法性能略优于其他两个方法，在数据较大时，三者性能差距不大。

三种实现方法各有各的优点，omp 的优点主要在于实现比较简单，只需要添加少量的代码就可以实现效果不错的并行化，但是自由度相比于其他两种方法略差一些。pthread 的优点主要在于自由度较高，方法较为底层，但是实现起来是三者之中最麻烦的。如果使用线程池的方式来

实现的话，那么我们可以使用 std::thread 或者网上的开源线程池库来实现，这样只需要加入一小部分代码，我们就可以实现一个相对优良的优化效果。

#### 4. O2 优化

使用 O2 优化之后，我们基本上所有算法的性能都呈现了提升趋势，但是有一些算法之间的性能差距发生了改变。例如 pthread 算法在数据规模小时看起来更慢了；串行递归算法比迭代算法快的更多了；在数据规模中等时融合方法比快速排序更有效，但数据规模较大时又是快速排序更快了，等。我们需要在实际的应用场景下分析对应的算法性能，这样才能设计实际有效的方法。

#### 5. 最终算法

在几种方法联合使用的场景下，算法的性能相比单独使用某种方法编程都有优化，比串行算法优化一个数量级，在数据量较大时，算法的执行时间是单独使用多线程方法的 1/3，单独使用 simd 方法时的 1/6，略微快于快速排序。数据量中等时，它与快速排序的性能相当。在数据量较小时还是快速排序最快，融合算法相比于本实验中实现的其他所有双调排序的并行化算法而言，在中等数据规模以上都具备着优势，但是在数据规模较小的时候，只应用 SIMD 的性能是最快的。

最终将几种编程方式运用到同一个算法中时，具备一定的优化效果，后续可以使用其他的算法编程策略来进一步对算法的性能进行优化，比如 cuda 或者 mpi 多进程，以及在算子项目中需要使用到的外存排序编程思想，来优化多级的排序算法性能，进一步让我们的算法能够在生产实践当中被使用。

## 六、 总结与展望

本次实验中，我们首先对双调排序算法的正确性进行了证明，然后我们使用了多种编程方法，包括多线程和向量指令的方式来对双调排序算法进行并行的优化。最终，我们取得了比较不错的优化效果，在数据规模较大时，它的算法性能可以优于 C++: std 里面的快速排序算法，这为算子项目中的多线程向量指令优化编程打下了坚实的基础。后续我们在算子项目中和期末研究报告主要侧重的点，会更偏向于外部排序算法的并行优化，这次实验相当于为外部排序算法做了一个内部排序的子问题，具备非常重要的研究和应用价值。

## 七、 AI 使用说明

本次实验中使用 AI 进行了辅助工作，其中，双调排序串行算法的两种思路是我自主思考并参考现有资料而得出的，并行化算法的思路和算法是我参考网上的资料并自主思考的，报告是我自主撰写的。本次实验使用 AI 辅助了测试代码的完成、测试数据图像的生成（python 脚本处理 csv 文件）、串行和并行算法的 debug 过程。对于 AI，我主要的使用思路是，使用 AI 辅助验证 idea 的正确性，来加速实验的过程。