分布式方法

N是cpu的核的个数

思考

1. 复杂度问题，实现发现，n平方的排序算法，在1万数据时执行时间为6秒，那么对于500万的数据，执行的时间为$500^2 * 6$秒，超过了17天，如果用32核的cpu，在理想情况下运行时间也超过了12小时，所以我们认为，使用多线程无法在可以容忍的时间内完成n平方级别的排序算法。所以我们只实现归并排序，快速排序，希尔排序。
2. 进程和线程，进程和线程都能利用多核cpu并行，线程之间共享内存，进程之间不共享内存，我们认为进程更符合分布式的精神，所以我们尽量使用进程的方式来实现分布式。

归并排序

想法一， 自上而下的分段归并

在自顶向下的递归归并排序算法中，先把数据分层N段，每一段交给一个进程去计算，最后用一个多路归并得到最后的结果。

```
def msort(A):
    merge_sort(A, 0, len(A), A[:])
    return A

def merge_sort(A, l, r, B):
    if l >= r -1:
        return
    m = l + (r - l) // 2
    merge_sort(A, l, m, B)
    merge_sort(A, m, r, B)
    merge_sort_partition(A, l, m, r, B)

def merge_sort_partition(A, begin, middle, end, B):
    i = begin
    j = middle
    for k in range(begin, end):
        if j >= end or (i < middle and A[i] < A[j]):
            B[k] = A[i]
            i += 1
        else:
            B[k] = A[j]
            j += 1
    for i in range(begin, end):
        A[i] = B[i]

def main(a):
    batch_count = os.cpu_count()
    batch_size = len(a) // (batch_count - 1)
    args = []
    for i in range(batch_count):
        args.append(a[i * batch_size : (i+1) * batch_size])

    with Pool(batch_count) as p:
        r = p.map(msort, args)

    # 多路归并
    idx = [0] * batch_count
    j = 0
    while j < len(a):
        minval = float('inf')
        minidx = 0
        for i in range(batch_count):
            if idx[i] < len(r[i]) and r[i][idx[i]] < minval:
                minval = r[i][idx[i]]
                minidx = i
        a[j] = minval
        idx[minidx] += 1
        j += 1
```

想法二， 利用递归深度

在自顶向下的递归归并排序算法中，记录当前的层数，在某一层，所有的调用都开一个进程，比如第3层，就开8个进程，后续的计算就在各自的进程了。

```
def msort(A):
    with Pool(1) as p:
        merge_sort(A, 0, len(A), A[:], 0, p)

def merge_sort(A, l, r, B, depth, p):
    if l >= r - 1:
        return A
    m = l + (r - l) // 2
    if depth == 3:
        args = [(A[l:m], 0, m - l, B[l:m], depth + 1, None), (A[m:r], 0, r - m,
B[m:r], depth + 1, None)]
        res = p.starmap(merge_sort, args)
        A[l : m] = res[0][:]
        A[m : r] = res[1][:]
    else:
        merge_sort(A, l, m, B, depth + 1, p)
        merge_sort(A, m, r, B, depth + 1, p)

    merge_sort_partition(A, l, m, r, B)
    return A

def merge_sort_partition(A, begin, middle, end, B):
    i = begin
    j = middle
    for k in range(begin, end):
        if j >= end or (i < middle and A[i] < A[j]):
            B[k] = A[i]
            i += 1
        else:
            B[k] = A[j]
            j += 1

    for i in range(begin, end):
        A[i] = B[i]
```

想法三，自下而上的分段归并

在自底向上的迭代归并排序算法中，每一轮中把数组分成N段，每一段开一个进程来计算。

```
def merge_sort(A):
    n = len(A)
    with Pool(12) as p:
        width = 1
        while width < n:
            args = []
            for i in range(0, n, 2 * width):
                begin = i
                middle = min(n, i + width)
                end = min(n, i + 2 * width)
                args.append((A[begin:end], 0, middle - i, end - i))
            r = p.starmap_async(merge_sort_partition, args)
            A[:] = itertools.chain.from_iterable(r.get())
            width *= 2

def merge_sort_partition(A, begin, middle, end):
    B = A[:]
    i = begin
    j = middle
    for k in range(begin, end):
        if j >= end or (i < middle and A[i] < A[j]):
            B[k] = A[i]
            i += 1
        else:
            B[k] = A[j]
            j += 1
    return B
```

实验

在两个cpu上做了实验，一个12核cpu，一个32核cpu，分别实验这三个想法，实验结果如下表：

归并排序	想法一500万	想法一1000万	想法二500万	想法二1000万	想法三500万	想法三1000万
单线程12核cpu	30	64	35	90	37	78
多进程12核cpu	18	39	17	36	29	64
多进程32核cpu	41	83	21	42	27	58

总结

1. 想法二效果最好
2. 实验发现想法一，在数据量比较大的时候，归并的时间过长
3. 想法三，切换进程的次数多于其他实现，降低了效率
4. 对于1000万数据的归并排序，单线程的效率也不算太差
5. 多线程显著提升了效率，12核的cpu提升的更明显
6. 最终的单线程归并是性能不能提高的主要原因，导致更多核的cpu没有更多帮助，所以后面的实验就采用想法二，只使用12核的cpu。

快速排序

使用归并排序的想法二，当深度为3时开新进程。

```
def quick_sort(A):
    n = len(A)
    with Pool(12) as p:
        quick_sort_split(A, 0, n, p, 0)

def quick_sort_split(A, l, r, p : Pool, depth : int):
    if l >= r - 1:
        return A
    j = l + 1
    for i in range(l + 1, r):
        if A[i] < A[l]:
            A[i], A[j] = A[j], A[i]
            j += 1
    A[l], A[j-1] = A[j-1], A[l]

    if depth == 3:
        args = [(A[l : j - 1], 0, j - l - 1, None, depth + 1), (A[j : r], 0, r - j,
None, depth + 1)]
        res = p.starmap(quick_sort_split, args)
        A[l : j - 1] = res[0][:]
        A[j : r] = res[1][:]
        return A
    else:
        quick_sort_split(A, l, j - 1, p, depth + 1)
        quick_sort_split(A, j, r, p, depth + 1)
    return A
```

实现结果如下

快速排序	500万	1000万
单线程	24	50
多线程12	16	43

希尔排序

希尔排序不太容易用进程的方式实现，因为每次都要遍历整个数组，必须共享数据，所以用线程的方式实现，考虑到Python的多线程不能有效利用多核cpu，使用c++来实现希尔排序的分布式实现。

在希尔排序中，使用多线程进行插入排序，未来避免开太多的线程，需要对线程数加一个限制，当gap比较大时，每个线程执行的任务较少，可以不开线程，当gap比较小时，每个线程执行的任务比较大，适合开新线程计算。在实验中设置当`gap < 300`时开新线程。

```
void shell_sort(int arr[], int n)
{
    for (int gap = n / 2; gap > 0; gap /=2)
    {
        vector<thread> ts;
        for (int k = 0; k < gap; k++)
        {
            if (gap < 300)
            {
                ts.push_back(thread(insertSortK, arr, n, gap, k)); // 使用多线程插入排
序
            }
            else
            {
                insertSortK(arr, n, gap, k); // 单线程插入排序
            }
        }
        for (auto &t : ts) {
            t.join();
        }
    }
}
```

希尔排序	500万	1000万	1亿
单线程	0.7秒	1.5秒	22秒
多线程12	0.6秒	1.3秒	20秒