高性能计算导论：hw5

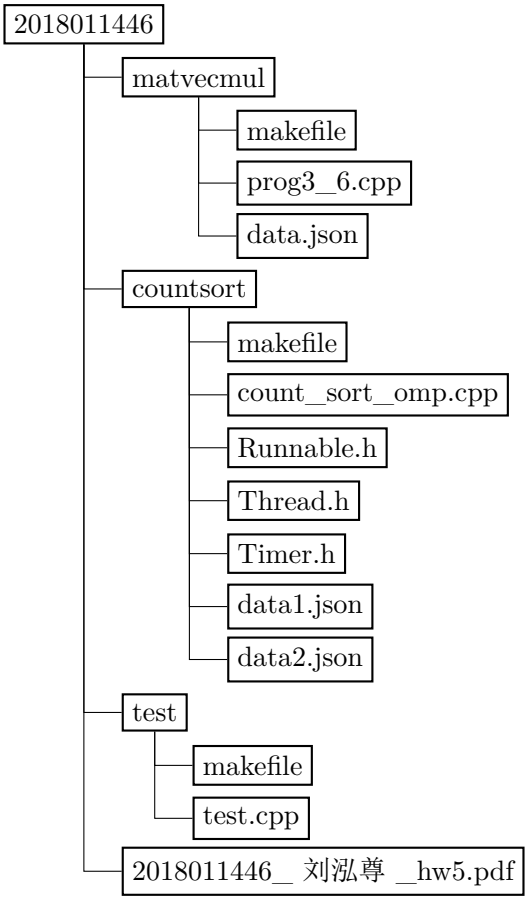
刘泓尊　2018011446　计 84

2020 年 4 月 19 日

目录

本次作业已上传至集群**./2018011446/hw5** 文件夹下。

1 File Structure



2 Exercise 5.4

&&	1(true)
||	0(false)
&	111...111
|	0
^	0

3 Exercise 5.5

a 计算完 4 次加法后，寄存器得到结果 1008，四舍五入后存入内存为 101×10^1, **输出 1010.0**

b 线程 0 得到结果 4.0，线程 2 寄存器得到结果 1003，存入内存后为 100×10^1, 规约后寄存器得到 1004, 再次四舍五入存入内存为 100×10^1, **输出 1000.0**

4 Programming Assignment 1

4.1 运行方式

./matvectmul 文件夹下运行“make run [p=...] [n=...] [t=..]”可以运行程序，并对不同的进程数 p, 线程数 t 和数据规模 n 进行设置。程序会输出二范数误差，总时间、通信时间、计算矩阵乘法时间和计算二范数时间，并给出相应的加速比和并行效率。正常的运行结果如下:

运行结果

```
srun -n 25 -l matvectmul 16000 4
00: error(2 norm):                          0.00000000002357028461
00: time (Serial):                          0.281075s
00: time (Total: multiply+communication): 8.792906s
00: time (Matrix-vector Multiply):          0.014261s
00: time (Scatter&Reduce):                  8.778645s
00: time (Calculate 2-norm):                0.000142s
00: Speedup ratio(with MPI_Scatter/MPI_Reduce): 0.0320
00: Speedup ratio(without MPI_Scatter/MPI_Reduce): 19.7092
00: Efficiency(with MPI_Scatter/MPI_Reduce): 0.0013
00: Efficiency(without MPI_Scatter/MPI_Reduce): 0.7884
```

4.2 测试结果

下一页展示了不同 n, p, th 值的运行时间 (总时间、通信时间、计算时间)、加速比 (总时间、计算时间)、并行效率 (总时间、计算时间) 统计图如下。(因为数据过多，我在报告中只展示折线图，您可以在 data.json 中查看实际的统计数据, 其中数据是三个维度的字典，维度信息依次为: 进程数 p，线程数 t，数据规模 n)。($p >= 25$ 时会跨机器运行，运行时间大大增加，与较小进程数不具备可比性。)

由于 MPI 通信部分无法并行，所以不同线程数的通信时间是相同的，这一点从统计图中也可以看出来。

在只考虑计算时间的并行效率上，在进程数固定时，线程数增多则效率下降，这符合我们的预期。进程数增多也会导致并行效率的下降。在总并行效率的表现上，进程内线程数的增多并不能显著增加并行效率，因为 MPI 的通信时间占了 95% 以上的比重。

从图中可以看到，提高此程序效率的关键在于 MPI 通信，而不是计算部分。对于 MPI 通信时间的改进，我也进行了多种尝试，我发现通信的 90% 都用在 matrix 的 Scatter 上，因为 Scatter 函数是线性进行数据分配的，而且矩阵规模巨大，所以 Scatter 的耗时很长。我将 Scatter 替换为了基于 MPI_ISend() 和 MPI_IRecv() 的点到点非阻塞通信，通过“非阻塞通信 + 并行分发”的方法，我将数据分发时间缩短到了原来的一半！但是对于 $n > 10000$ 的规模，点到点通信总是会引起程序崩溃。保险起见，我最终还是使用了 Scatter. 但这一尝试至少为我们提供了一种缩短通信时间的思路！我也期待助教老师能为我提供更多的解决方案！

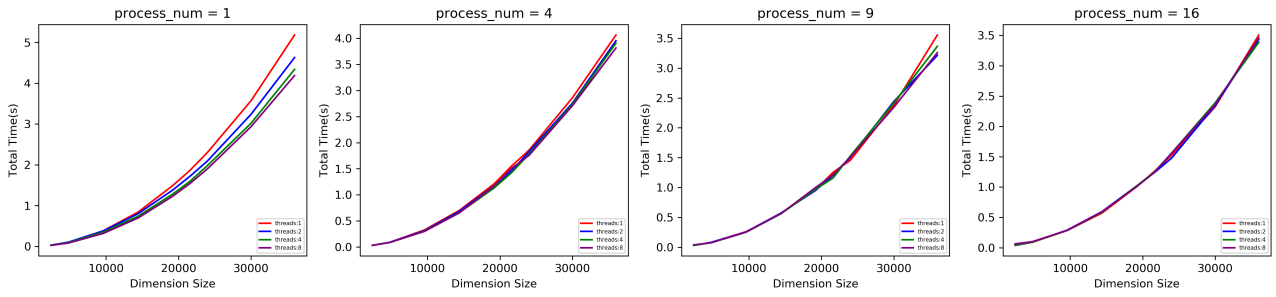


图 1: Total Time(s)

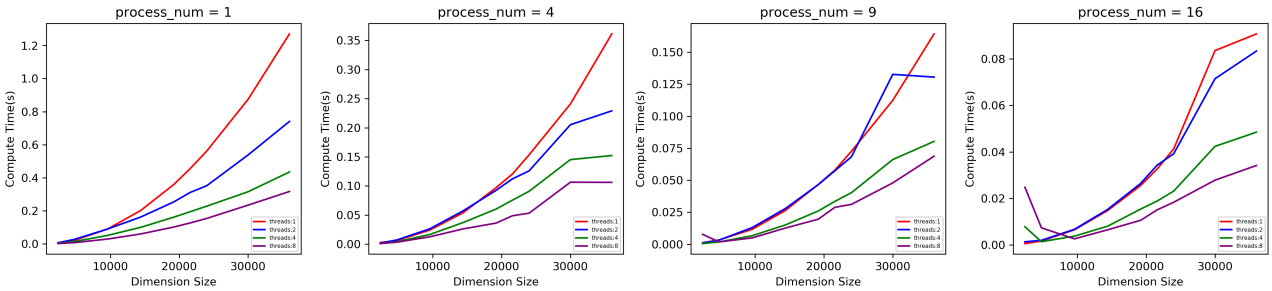


图 2: Compute Time(s)

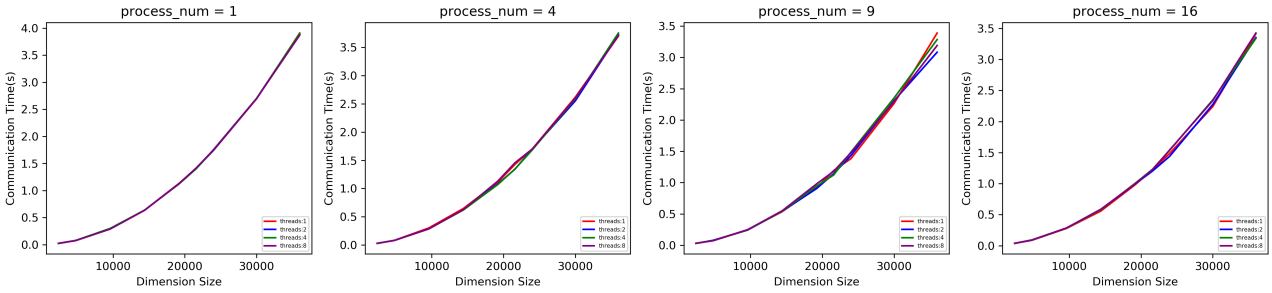


图 3: Communication Time(s)

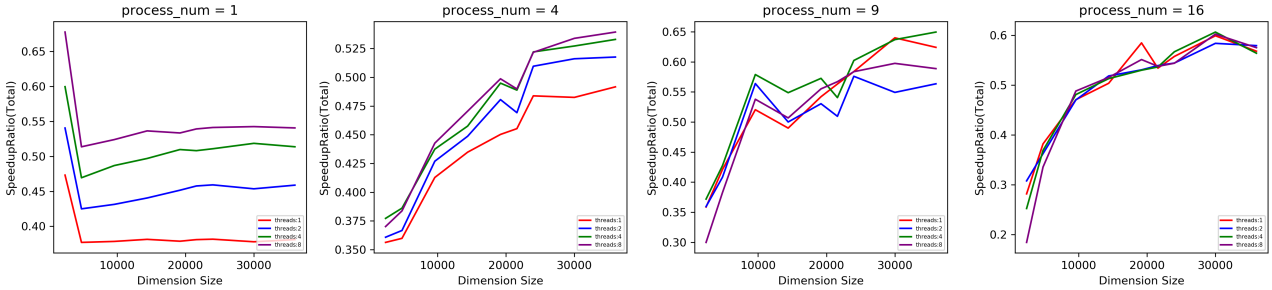


图 4: SpeedupRatio(Total)

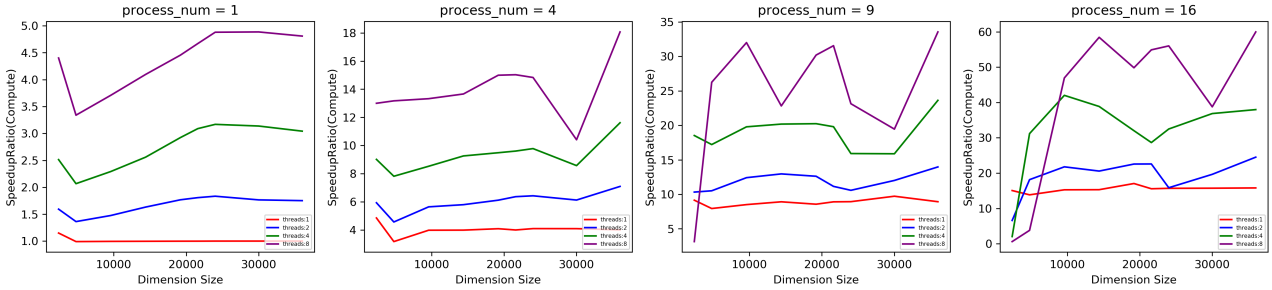


图 5: SpeedupRatio(Compute)

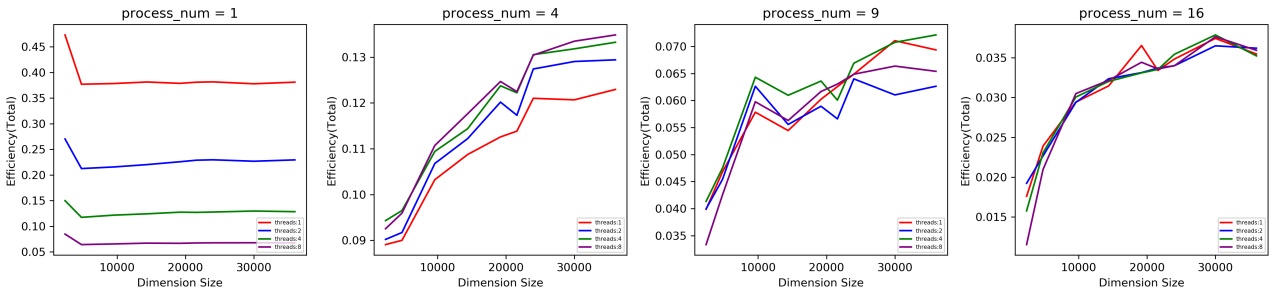


图 6: Efficiency(Total)

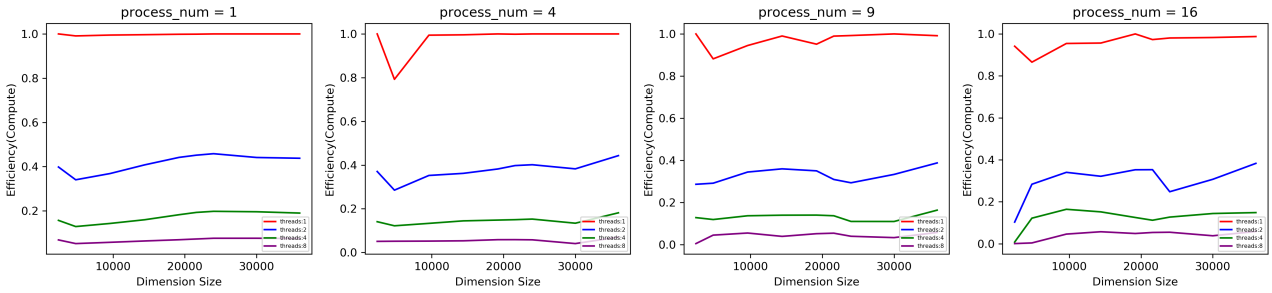


图 7: Efficiency(Compute)

4.3 和作业 3 中的比较

本次作业比作业 3 添加了 OpenMP 的多线程编程，具体做法是在每个线程中派生出若干个新线程，同时计算并行效率时应注意除以 $p \times th.$ 和作业三中的结果比较，主要是计算时间有了明显加速，平均加速比从 0.4 上升到了 0.6。但是程序热点依然在于 MPI 通信，占掉了 95% 以上的时间，并且通信部分使用 MPI_Scatter() 是无法并行的，但是计算时间有了明显加速，和线程个数呈正相关。这进一步印证了“线程”调度开销远小于“进程”通信开销的结论，因此，在存在较多数据共享的场合，使用“多线程编程”能达到更高的效率；但是在可靠性上，多进程因为数据隔离，实际上更加安全，并且适用于多机并行。实际应用应该综合两者的优点进行设计。此外我发现，由于核心数量的限制，在 p 较大时，增大线程数无法起到加速效果，因为此时多线程的任务实质上是在一个核心上进行的，这反而会拖慢效率。

$n \backslash p$	4	16	25
2000	0.013/0.4523	0.013/0.4618	0.168/0.0268
8000	0.184/0.3611	0.190/0.3601	1.274/0.0311
12000	0.422/0.3514	0.437/0.3506	2.234/0.0314
16000	0.781/0.3514	0.768/0.3571	5.006/0.0315
24000	2.556/0.3412	2.548/0.3496	8.891/0.0281

表 1: 作业 3 中 Total time(s) & Speedup Ratio(考虑数据分发)

我还发现，在不同线程数上，并行效率与数据规模存在相关性，比如在 $p = 4, t = 16$ 时，规模 $n = 20000$ 的情况获得了远超其他规模的并行效率。这说明为了达到更高的效率，应对不同的数据规模设计不同的线程数和进程数。

5 Programming Assignment 2

5.1 运行方式

“make”编译成功后，执行“make run [p=...] [n=...] [c=...] [max=...]”可以运行程序，其中 p 代表线程数, n 代表数据规模，c 代表块大小 (chunk_szie)，max 代表生成的数据范围 (鉴于 $O(n+k)$ 计数排序的要求，推荐为 $max < 5000$)。最终程序会输出各种排序的运行时间、加速比及并行效率。一个可能的运行结果如下:

```
thread_num: 4, num: 50000, chunk_size:32, max_num:50
serial count:           0.321601s
parallel count:         0.311788s
parallel count+merge:   0.044520s
parallel count O(n+k):  0.020629s
qsort() time:           0.003546s

Speedup Ratio(count):   1.031474
Speedup Ratio(count+merge): 7.223718
Speedup Ratio(count O(n+k)): 15.589802

Efficiency(count):      0.257868
Efficiency(count+merge): 1.805929
Efficiency(count O(n+k)): 3.897450

parallel count <static>: 0.293447s
parallel count <dynamic>: 0.256954s
parallel count <guided>: 0.361447s

Speedup Ratio <static>: 1.095942
Speedup Ratio <dynamic>: 1.251590
Speedup Ratio <guided>: 0.889759

Efficiency <static>:    0.273986
Efficiency <dynamic>:   0.312897
Efficiency <guided>:    0.222440
```

5.2 哪些变量是私有的，哪些变量是共享的

private	i, j, count
shared	a, n, temp

5.3 外循环是否存在循环依赖

不存在循环依赖。a 和 n 只有**读**操作，对 temp 只有**写**操作，并且不同线程不会访问 temp 的同一元素，因此不存在。

5.4 是否能并行化对 memcpy 的调用

修改区间划分之后可以并行化 memcpy，因为 memcpy 的内部实现也是基于循环的，并且没有循环依赖。下面我给出实现代码 (测试代码放在./test 文件夹下):

并行化 memcpy

```
    int interval = n / thread_count + 1, i;
#   pragma omp parallel for num_threads(thread_count)
    for(i = 0; i < n; i += interval){
        memcpy(a+i, temp+i, min(n-i, interval) * sizeof(int));
    }
```

下面我给出实测的不同规模 n 和线程数 $threads$ 的加速比数据，注意横坐标是对数的:

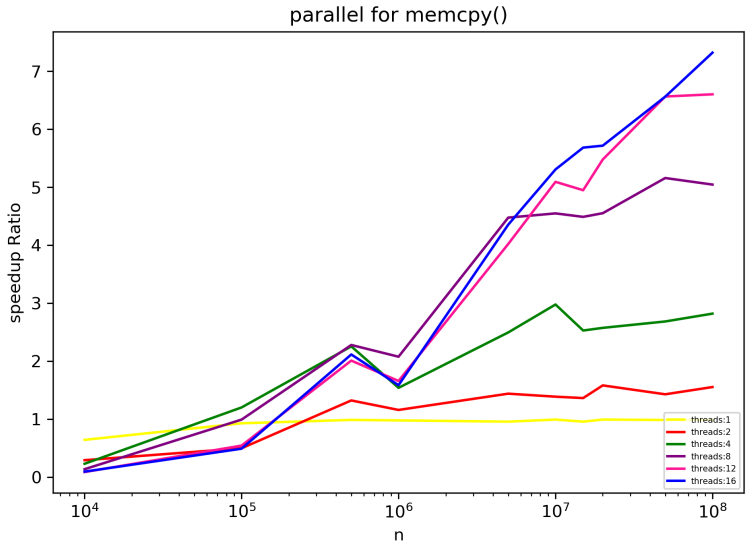


图 8: Speedup Ratio of OpenMP parallel for memcpy()

从数据中可以看到，openMP 对于 $n > 10^5$ 的数据规模会有比较明显的加速。在编写程序时可以采用加速后的 memcpy()，但是因为题给计数排序是 $O(n^2)$ 的，所以这一改进对整体性能优化不明显。

5.5 程序说明

为了更全面地了解各种优化的效果，我分别实现了“串行 $O(n^2)$ 计数排序”，“并行 $O(n^2)$ 计数排序”,“并行 $O(n+k)$ 计数排序”，“四分区间最终归并的计数排序”共 4 个版本，并进行了性能测试. 编译选项选择“-O3 -march=native”

5.5.1 串行 $O(n^2)$ 计数排序的优化

我发现在作业要求中给出的计数排序是 $O(n^2)$ 的，并且在循环内的判断是十分低效的，$if-else$ 分支预测错误会浪费很多时间。因此我将原程序修改成了如下代码，在 $n = 100000$ 的规模下，优化后的版本 (1.34s) 比原始版本 (28.34s) 快了 20 倍以上。具体思路是使用两段循环代替了之前的分支判断。

Count_sort_fast()

```
for(i = 0; i < n; i++){
    count = 0;
    for(j = 0; j < i; j++){//减少分支预测
        count += (a[j] <= a[i]);
    }
    for(j = i; j < n; j++){
        count += (a[j] < a[i]);
    }
    temp[count] = a[i];
}
```

5.5.2 $O(n^2)$ 版本的并行化

之后我对改进后的计数排序进行了并行化，正如前面几个小题分析的那样，外层循环可以并行化，因此复杂度可以从 $O(n^2)$ 下降到 $O(n^2/th)$，其中 th 是线程个数. openMP 的并行化十分简单，代码如下.

Count_sort_para()

```
#  pragma omp parallel for num_threads(thread_num) private(i, j, count)
    for(i = 0; i < n; i++){
        count = 0;
        for(j = 0; j < i; j++){
            count += (a[j] <= a[i]);
        }
        for(j = i; j < n; j++){
            count += (a[j] < a[i]);
        }
        temp[count] = a[i];
    }
```

5.5.3 $O(n+k)$ 版本的并行化

实际上计数排序有复杂度 $O(n+k)$ 的实现，其中 k 是数据范围。当 $k < nlogn$ 的时候，计数排序的理论时间复杂度是小于基于比较的排序的。简单分析可以看到，因为 $O(n+k)$ 的计数排序涉及到对每个数字频率的统计，而不是直接算出它们的位置，之后还需要计算“前缀和”，所以存在着较强的依赖。并行化的方式需要特殊设计，而不是像 $O(n^2)$ 的并行那样直接。

我的思路是，将数据均分给各个线程之后，首先在各个线程内部设置 local_count[] 数组，用于统计本线程负责数据的频率统计，然后再对各个线程的 local_count[] 进行规约得到 count[]. 之后在主线程内计算前缀和，之后各个线程再负责计算每个元素的位置。

如果线程数为 th, 串行版本的计数排序复杂度为 $O(2n+k)$, 则并行后的奇数排序复杂度为 $O(2n/th+2k)$, 但是考虑到线程调度的开销以及本地数组 local_count[] 创建销毁的开销，实际上并行版本的常数比较大。实测发现，4 个线程版本的速度仅仅是串行版本的 1.5 倍左右，而且这种优化只适用于 k 比较小的情况。当然，因为复杂度是线性的，依然比原始 $O(n^2)$ 的版本快了很多！

Count_sort_onk()

```
    int length = n / thread_num + 1;
#   pragma omp parallel for num_threads(thread_num)
     for(int t = 0; t < thread_num; t++){//统计频率，O(n)
         int* local_count = new int[k+1];
         memset(local_count, 0, (k+1)*sizeof(int));
         int begin = t * length;
         int end = std::min(begin + length, n);
         for(int i = begin; i < end; i++){//各线程统计线程内数组
             local_count[a[i]] ++;
         }
         for(int i = 0; i < k; i++){//规约
#          pragma omp critical
             count[i] += local_count[i];
         }
         delete[] local_count;
     }
     int i, j, index = 0, temp, loc;
     for(i = 1; i < k; i++){
         count[i] += count[i-1];
     }
#   pragma omp parallel for num_threads(thread_num) private(temp, loc)
     for(i = n-1; i >= 0; i--){//统计结果O(n)，可以简单并行
         temp = a[i];
#       pragma omp critical
         loc = --count[temp];
         result[loc] = temp;
     }
```

5.5.4 最后 4 路归并的计数排序

实际上，归并排序是十分修改成并行的。为了使得程序达到更好的表现，我将数据划分为 4 部分，交给 4 个线程做并行计数排序，之后进行 4 路归并得到最终的结果。在底层排序采用 $O(n^2)$ 的计数排序的情况下，复杂度优化到了

$$O(\frac{(n/4)^2}{th}+\frac{n}{2}+n)=O(\frac{n^2}{16th}+\frac{3n}{2})$$

归并部分采用线性归并，代码如下:

mergeThread

```
if(can_sort){//4个线程分别进行计数排序
    Count_sort_para(a + left, right - left);
    finish_count ++;
    return;
}
int* b = new int[right - left];//如果不需要排序，对两个子数组进行线性归并
```

```
int mid = (left + right) >> 1;
for(int p = left, q = mid, i = 0; p < mid || q < right; ){
    b[i++] = (q >= right || (p < mid && a[p] <= a[q])) ? a[p++] : a[q++];
}
memcpy(a + left, b, (right - left) * sizeof(int));
finish_count ++;//线程完成数++
```

归并部分的调度如下，其中 mergeThread() 是我基于 Pthread 封装的线程类，这也算是 openMP 和 Pthread 混合编程的一个尝试。显然，Pthread 更适用于细粒度地进行线程调度，赋予了开发人员更大的自由度，但是同时也增加了繁琐性。将 openMP 和 Pthread 混合使用，可以综合两者的优点，达到更好的效果。

Count_sort_para_merge()

```
int m2 = n >> 1;//区间划分
int m1 = m2 >> 1;
int m3 = (m2 + n)>>1;
//先分配给4个线程进行计数排序
mergeThread* t1 = new mergeThread(a, 0, m1, 1);
mergeThread* t2 = new mergeThread(a, m1, m2, 1);
mergeThread* t3 = new mergeThread(a, m2, m3, 1);
mergeThread* t4 = new mergeThread(a, m3, n, 1);
finish_count = 0;
t1->start(); t2->start(); t3->start(); t4->start();
t1->join(); t2->join(); t3->join(); t4->join();
while(finish_count < 4)//busy-wait
    ;
if(finish_count == 4){
    finish_count = 0;
    t1->set(0, m2, 0);//重新设置排序区间，reuse t1 and t2
    t2->set(m2, n, 0);
    t1->start(); t2->start(); t1->join(); t2->join();//两个线程进行2路归并
}
while(finish_count < 2)//busy-wait
    ;
if(finish_count == 2){
finish_count = 0;
    t1->set(0, n, 0);//reuse t1, 负责最后的2路归并
    t1->start(); t1->join();
}
while(finish_count < 1)//busy-wait
    ;
```

5.6 性能测试

下一页展示了我对上面 4 种版本的计数排序的性能进行了测试，整数范围 0 – 49，各版本**计算用时**、**加速比**、**并行效率**如下 (原始数据位于./data1.json 下，数据量过大，这里只给出统计图):

从数据中可以看到，qsort() 的效率最高, 但是实际上，因为本题中整数范围 0-49，所以快速排序中很多的区间划分不需要做任何的数据移动操作，使得 qsort 获得了很大的性能改进。但是对于完全随机的 0-65535 的数据，qsort 的性能产生了明显下降, 速度会小于 $O(n+k)$ 的计数排序。

在 $O(n^2)$ 的版本中，两者使用相同的算法，在线程数比较小时 (比如 threads=2)，并行版本的速度并没有比串行版本快，线程的调度开销比较大。此外，因为使用了“-O3 -march=native”编译选项，串行版本的 CountSort() 比“-O2”选项又提升了 3 倍左右，所以在线程数较少的时候，线程调度占了很大比重的时间，导致并行效率不高。在线程数 >4 之后，加速比开始超过 1，并且数据规模越大，加速效果越明显。这是因为计算时间在总时间中比重上升，使得线程调度时间相对而言影响减少了。此外，线程数的增加会导致并行效率的下降，这一现象是符合预期的，因为调度开销会增多。

此外可以看到，CountSort+4 路归并的版本获得了更高的效率，尽管算法依然是 $O(n^2)$ 的，但是因为分成 4 段进行排序，所以这种加速更加显著。加速比始终超过了 1，并且并行效率也是随着数据规模不断上升的。我们也可以看到，Pthread+OpenMP 不仅为开发人员提供了灵活性，还带来了更大的效率提升。

对于 $O(n+k)$ 复杂度的 CountSort 而言，讨论其加速比似乎没有意义，因为复杂度已经和 $O(n^2)$ 不在一个量级，“加速比”只能再一次印证线性复杂度的高效！但是我也发现了一个有趣的现象，随着进程数的上升，$O(n+k)$ 的计数排序的并行效率有了显著的下降。分析之后我认为有两个原因：1. 我们的数据规模对于 $O(n+k)$ 的算法没有充分体现其并行效率，因为计算时间占比大大下降了，线程的调度时间占了上风，增大数据规模后，计数排序的性能又有了提升。2.$O(n+k)$ 版本的 CountSort 存在很多不完全并行的情况，诸如存在临界区、子线程需要创建 local_count[] 数组等，这极大地增大了算法的常数。

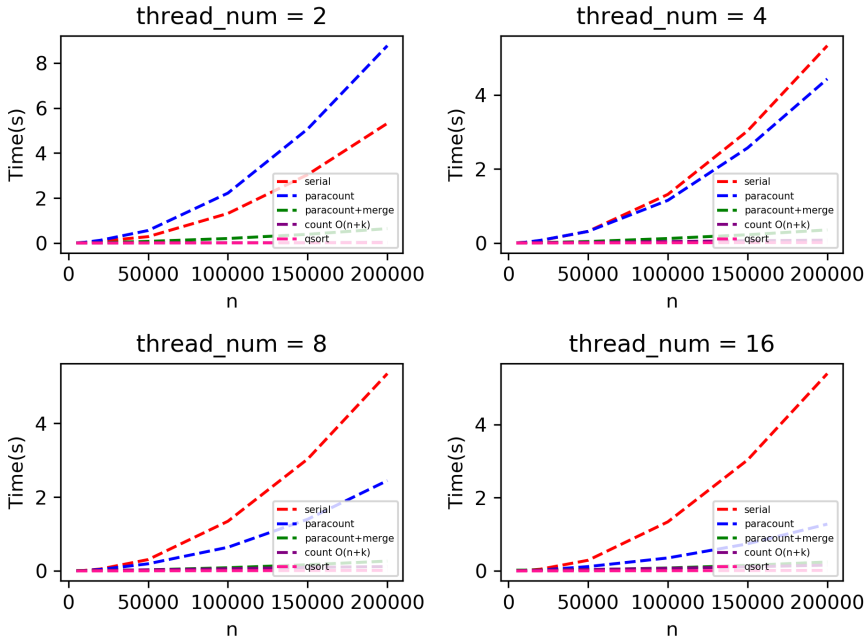


图 9: Time(s)

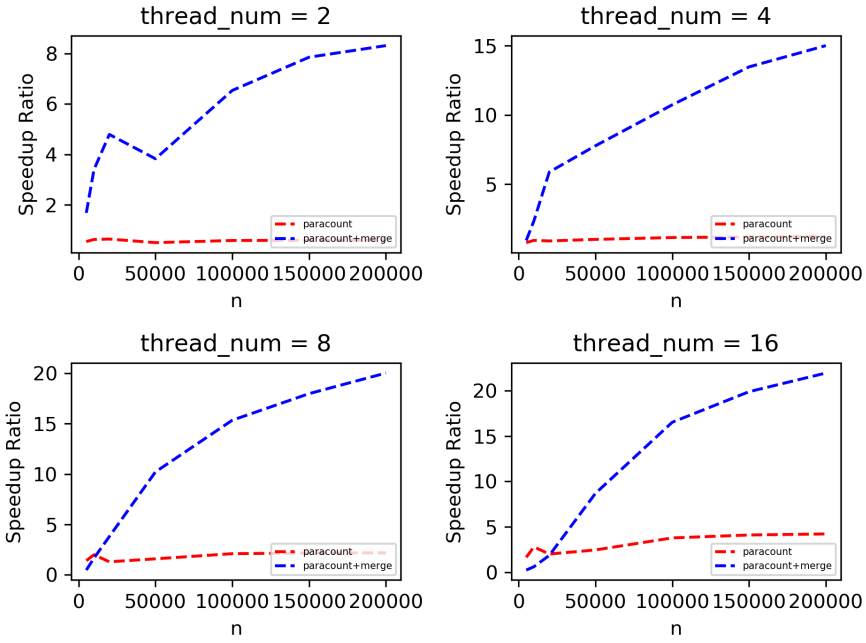


图 10: SpeedupRatio

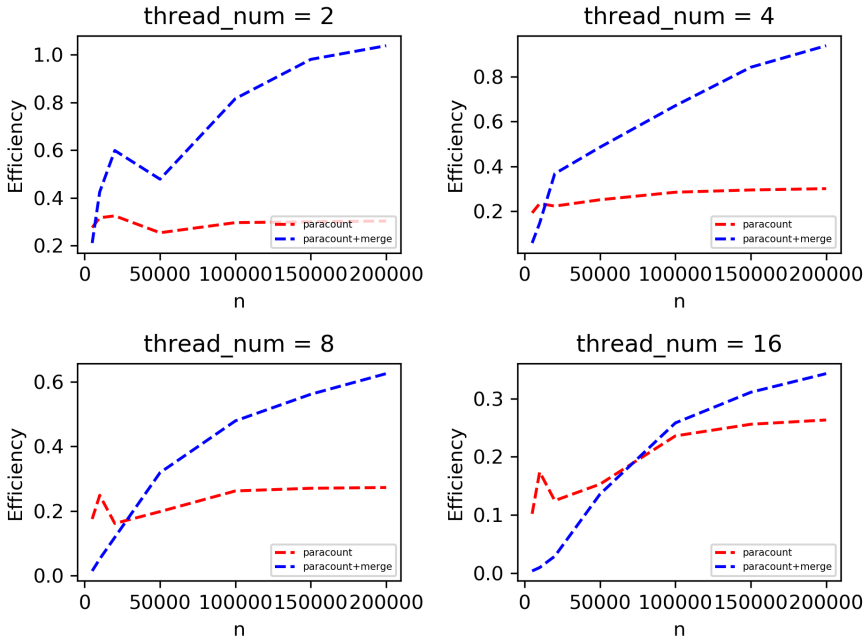


图 11: Efficiency

5.7 不同调度策略和块大小的对比

我对不同调度策略和 chunk_size 的运行效率进行了测试 (只对比了 $O(n^2)$ 的串行、并行版本)，对于不同的线程数 t 和 $n = 100000$ 的数据规模，得到的结果如下 (原始数据位于./data2.json 下，数据量过大，这里只给出统计图).

从数据中可以看到，dynamic 调度的加速比和并行效率是最高的，static 次之，guided 最差。这说明线程间的负载其实是不太均匀的。对于不同的 chunksize，2 个线程时，在块大小为 150 左右时，并行效率达到最高。8 或 16 个线程时，块大小为 100 时并行效率达到最高。实际上，块大小的变化对并行效率的影响并不大，整体上曲线都十分“水平”。最优块大小和线程数之间存在相关性，在实际工作中，我们需要对不同的线程数设置不同的块大小，以达到最优速度。最终我选择 $chunk\_size = 100$。

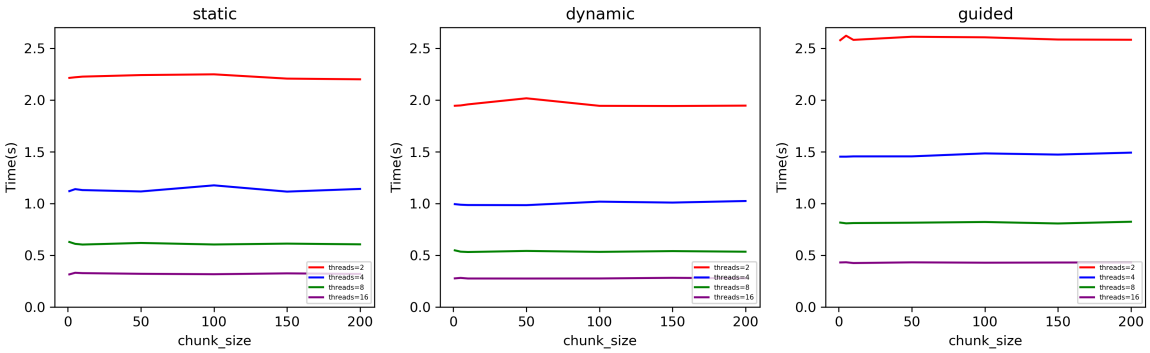


图 12: Time(s)

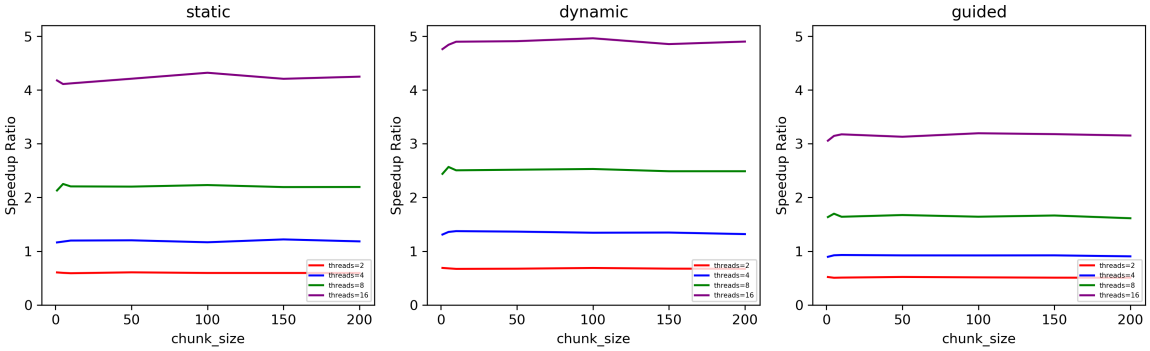


图 13: Speedup Ratio

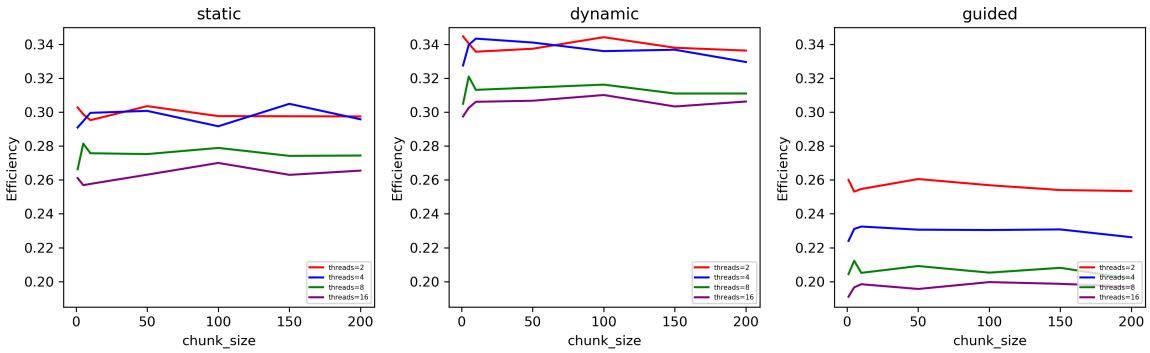


图 14: Efficiency