高性能计算导论第五次作业

李晨昊 2017011466

2019-6-1

目录

1 Programming Assignment

第三次的作业中我就已经测量过了性能数据，不过这次换了一下编译器和编译选项，所以还是全部重测了一遍。

既然本次对于性能做出了明确的要求，这里理应采用-Ofast -march=native 优化，让编译器能够自由地用 SIMD 指令来优化浮点运算，不必担心浮点误差对结果的影响。不过经测试性能提升很小，于是我又尝试手写了一些循环展开和 SIMD，发现提升也很小，我认为这可能有下面几方面的原因：

1. 即使我没有写这些东西，编译器也已经优化的比较好了；测试表明，运行速度 gcc -Ofast -march=native ≈ icc -Ofast -march=native ≈ icc -O3 ≈ 手写 SIMD > gcc -O3，不过我对此表示比较怀疑，因为这种优化往往能改变程序的运行结果，编译器应该不能擅自做这么激进的优化
2. SIMD 优化本身对于双精度浮点数的效果就没有对于单精度浮点数大。由于二者数据宽度的差别，理想状态下单精度浮点数的运算效率二倍于双精度浮点数。但是既然题目要求使用双精度浮点数，我对此也没有什么办法
3. 任务本身瓶颈在访存，而非运算。矩阵乘向量过程运算简单但访存需求大，而这种简单浮点运算的速度其实是比访存快得多的。例如在 intel 手册上可以查到，在服务器的 Haswell 架构的 CPU 上，`_mm256_fmadd_pd` 的 latency 为 6，CPI 为 0.5，这比访存大概快了两个数量级

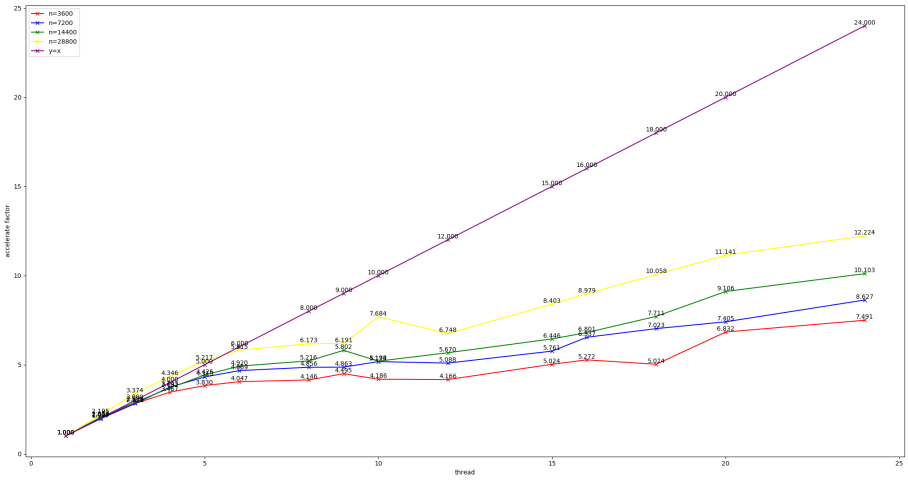
由于最后速度提升并不明显，我在最终版本里没有使用手写的 SIMD，但是还是把它留在了 prod.cpp 中。这代码基本上是我的机器上的 clang 生成的汇编代码反向翻译回 C++ 的，在我的机器上的效果也还算不错，明显地快于 clang 和 gcc 的-O2 生成的代码。但是因为我没有 icc(而且 icc 据说生成的代码在 AMD 平台上效率并不高)，所以没办法在我的平台上和 icc 的-O2 进行比较。

按列划分，不考虑数据分发和收集，耗时结果如下：

p	n=3600	n=7200	n=14400	n=28800
1	0.00854s	0.03399s	0.13942s	0.65621s
2	0.00422s	0.01728s	0.06795s	0.29894s
3	0.00302s	0.01205s	0.04818s	0.19448s
4	0.00247s	0.00908s	0.03776s	0.15100s
5	0.00223s	0.00788s	0.03150s	0.12579s
6	0.00211s	0.00728s	0.02834s	0.11284s
8	0.00206s	0.00700s	0.02673s	0.10631s
9	0.00190s	0.00699s	0.02403s	0.10599s

p	n=3600	n=7200	n=14400	n=28800
10	0.00204s	0.00657s	0.02682s	0.08540s
12	0.00205s	0.00668s	0.02459s	0.09724s
15	0.00170s	0.00590s	0.02163s	0.07809s
16	0.00162s	0.00520s	0.02050s	0.07308s
18	0.00170s	0.00484s	0.01808s	0.06524s
20	0.00125s	0.00459s	0.01531s	0.05890s
24	0.00114s	0.00394s	0.01380s	0.05368s

加速比曲线如下：

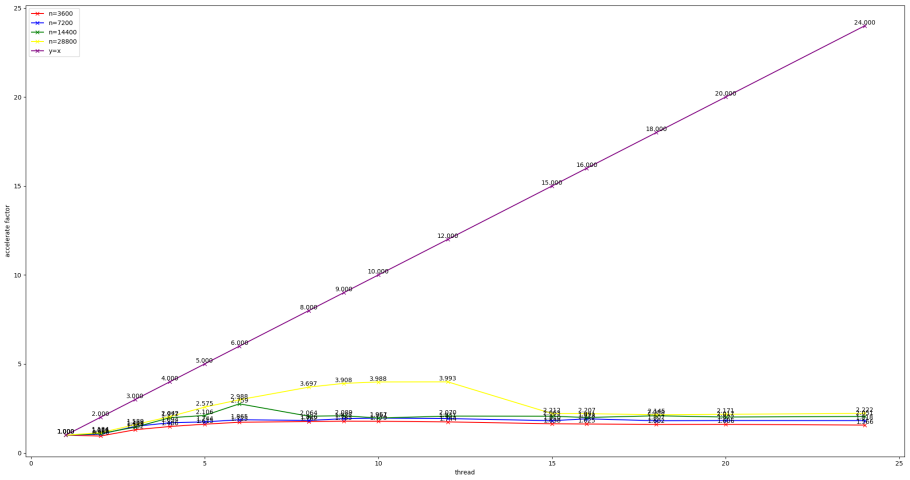


考虑数据分发和收集，耗时结果如下：

p	n=3600	n=7200	n=14400	n=28800
1	0.04101s	0.16158s	0.65047s	2.68258s
2	0.04319s	0.15372s	0.59899s	2.36608s
3	0.03149s	0.10811s	0.44896s	1.68811s
4	0.02759s	0.09536s	0.32899s	1.31387s
5	0.02540s	0.09263s	0.30892s	1.04187s
6	0.02378s	0.08665s	0.23575s	0.89775s
8	0.02327s	0.08849s	0.31522s	0.72554s
9	0.02297s	0.08393s	0.31131s	0.68641s
10	0.02305s	0.08256s	0.33165s	0.67264s

p	n=3600	n=7200	n=14400	n=28800
12	0.02351s	0.08366s	0.31419s	0.67175s
15	0.02503s	0.08926s	0.31526s	1.21293s
16	0.02523s	0.08426s	0.32665s	1.21559s
18	0.02560s	0.08943s	0.31148s	1.25053s
20	0.02554s	0.08869s	0.32243s	1.23589s
24	0.02619s	0.08890s	0.31714s	1.20727s

加速比曲线如下：



对于每个 n，计算误差二范数的计算量都是一样的 (而且都很小)，比较不同线程数下的执行速度没什么价值，直接用默认的线程数即可：

n=3600	n=7200	n=14400	n=28800
0.00744s	0.00416s	0.00393s	0.00478s

这些数据都是多次测量取得的最小值，而测量过程中也发现时间的波动很大。我认为原因在于数据量太小，以至于计算耗时远小于开启线程和同步线程的额外开销。事实上，我把它们替换回了串行版本后，再测试一次，结果为：

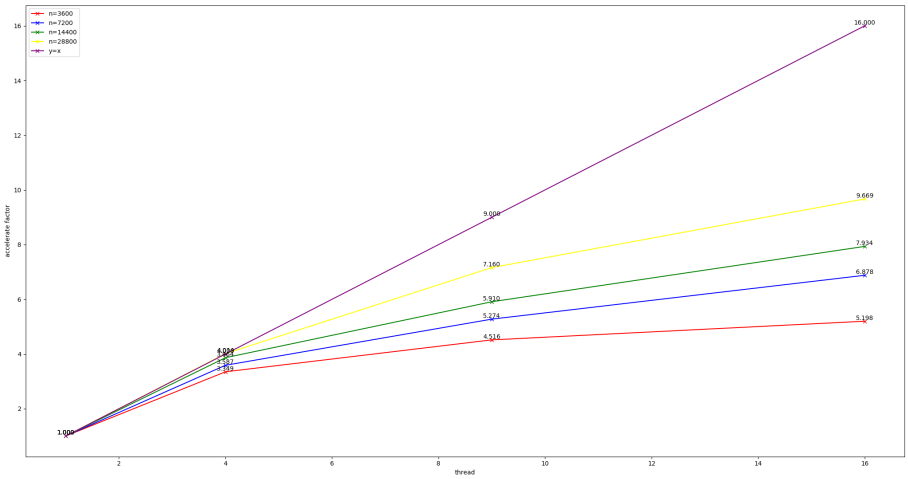
n=3600	n=7200	n=14400	n=28800
0.00000s	0.00001s	0.00003s	0.00006s

这算是比较正常的结果，而上面的测试结果里的波动也就可以解释了：几乎所有的时间都属于 $T_{overhead}$。对于数据规模如此小的计算，没有使用并行的意义。

按子矩阵划分，不考虑数据分发和收集，耗时结果如下：

p	n=3600	n=7200	n=14400	n=28800
1	0.00998s	0.03824s	0.16265s	0.71438s
4	0.00298s	0.01066s	0.04209s	0.17797s
9	0.00221s	0.00725s	0.02752s	0.09978s
16	0.00192s	0.00556s	0.02050s	0.07388s

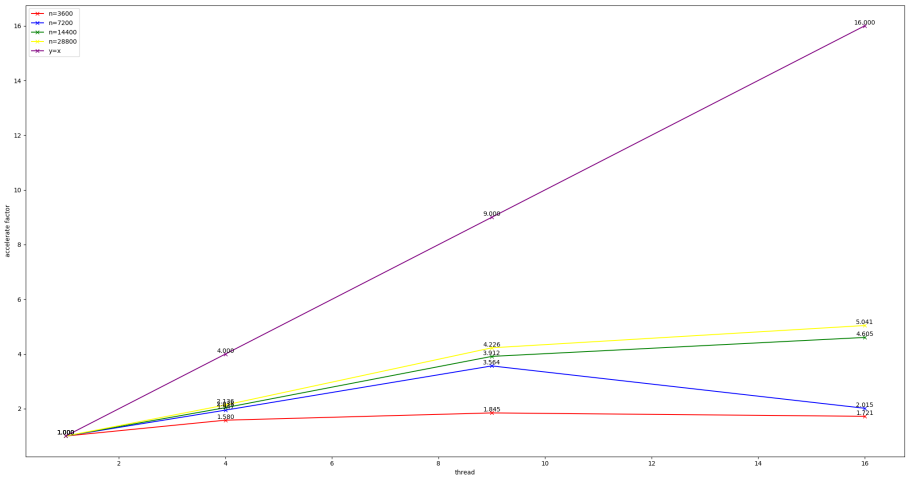
加速比曲线如下：



考虑数据分发和收集，耗时结果如下：

p	n=3600	n=7200	n=14400	n=28800
1	0.04069s	0.16151s	0.64825s	2.66666s
4	0.02576s	0.08296s	0.31809s	1.24863s
9	0.02205s	0.04532s	0.16570s	0.63102s
16	0.02364s	0.08014s	0.14077s	0.52902s

加速比曲线如下：



由于 n 没有变化，这里就不重复列出计算二范数的耗时了。

2 Programming Assignment 4.5

题目要求大概让我们实现一个线程池。至于那个给我们的主函数，我没有管它。

线程池的大致实现思路是，开启多个线程，它们一直都在等待任务而不退出，这样在创建任务时就避免了创建线程的开销。当然开销肯定还是存在的，这是线程间通信的开销，比创建线程要小的多。

“等待任务” 的具体实现方法是使用条件变量，维护一个任务队列，当队列为空时需要等待，当加入新任务时可唤醒一个线程。此外，线程池还需要一个退出的逻辑，因此等待条件中还需要检查是否退出，如果已经退出，且任务队列为空，则本线程的执行结束。

关键代码如下：

```
static void *task(void *arg) {
  ThreadPool *self = (ThreadPool *)arg;
  while (true) {
    self->mu.lock(); // 管程开始
    // Mesa 管程，用 while
    while (self->q.empty() && !self->no_more) {
      self->cv.wait(&self->mu);
    }
    if (self->q.empty()) {  // 证明 self->no_more 成立，可以退出
```

```
        self->mu.unlock();
        break;
      }
      T t = self->q.front();
      self->q.pop();
      self->mu.unlock(); // 管程结束
      t(); // 管程外执行任务
    }
    return nullptr;
  }
```

题目中要求的：“each time it generates a new block of tasks, it awakens a thread with a condition signal”，对应于代码：

```
  void push(T t) {
    mu.lock();
    q.push(t);
    cv.notify_one();
    mu.unlock();
  }
```

题目中要求的：“When the main thread completes generating tasks, it sets a global variable indicating that there will be no more tasks, and awakens all the threads with a condition broadcast”，对应于代码：

```
  ~ThreadPool() {
    no_more = true;
    cv.notify_all();
    ...
  }
```

我对锁和条件变量等做了一点封装，这样自己用起来舒服一些。当然 stl 其实已经封装好了 pthread，不过为了练习的目的，显然我们不应该使用使用 stl 的版本。不过这个 no_more 变量，我使用了 std::atomic<bool>，至于为什么不应该用普通 bool 或者 volatile bool，我在上次报告中说的比较清楚了。也许它们生成的汇编是一模一样的，但是我们期望的语义是 atomic，不是 volatile。比较可惜 pthread 没有提供原子变量，因此这里是不得不用到 stl 的。

关于链表，直接用一个锁和一个 std::list 来实现。这里没有维护链表的有序性，因为题目中并没有说需要维护，它只是一个普通的链表而已。对于插入操作，直接使用了 `push_back` 将它插入到链表头。如果真的想要一个课件中的那种维护了有序性和元素唯一性的链表，毫无疑

问应该使用平衡树，例如 std::set 来实现，使用链表是完全没有道理的。

3 Programming Assignment 5.3

3.1 普通版本

这个小标题里回答了题目提出的几个问题，并且按照最直接的想法并行化了计数排序。

1. 如果我们试图并行化 i 的 for 循环 (外层循环)，哪些变量应该是私有的，哪些变量应该是共享的?

私有：i，j，count

共享：a，n，temp

2. 如果我们使用前面定义的作用域来并行化 i 的 for 循环，是否存在循环依赖？请解释你的回答。

不存在。注意到循环体内没有对于共享可变变量的读操作，因此不存在循环依赖。

3. 我们是否能够并行化对 memcpy 的调用？我们是否能够修改代码，使得这部分代码可以并行化?

可以，分段并行 memcpy 即可。但是这样很有可能并不能达到较好的效果，因为 memcpy 本身非常快，很有可能耗时无法弥补开启线程等操作的耗时。

4. 编写一个包含对计数排序程序并行化实现的 C 程序。

具体实现见代码。

这里对于内层循环做了一点优化，用两个循环代替了原来的一个：

```
u32 cnt = 0;
for (u32 j = 0; j < i; ++j) {
  cnt += a[j] <= a[i];
}
for (u32 j = i; j < n; ++j) {
  cnt += a[j] < a[i];
}
```

容易看出这和题目中给出的循环是等价的，它的好处在于大大减少了循环体内所需的判断。不过，其实这种优化应该算是比较平凡的，我很遗憾编译器没有自己看出来，还是需要人的智慧的帮助。

同时注意到这个循环可以很好的用 SIMD 指令来实现。但是，是否能够进行 SIMD 优化，以

及优化能到什么程度，其实都完全取决于编译器的行为：我在我的机器 (AMD R7-2700) 上分别使用 gcc 8.3.0 和 clang 8.0.0，在服务器上分别使用了 gcc 4.8.5 和 icc 18.0.0(**注：icc 编译出来的带 openmp 代码似乎只能在 bootstraper 上运行，在 cn0xx 上运行提示缺少动态链接库 libiomp5.so**)，对于 $n = 10^5$ 记录单线程版本耗时，记录如下：

	-O2	-O3 -march=native
AMD R7-2700, gcc 8.3.0	5.52603s	0.87211s
AMD R7-2700, clang 8.0.0	0.78120s	0.62353s
Intel Xeon E5-2680 v3, gcc 4.8.5	6.26367s	3.89370s
Intel Xeon E5-2680 v3, icc 18.0.0	1.60733s	1.12694s

不得不说 clang 的 SIMD 优化还是相当值得信赖的，符合我对它一贯的认知。$O(n^2)$ 能过 $n = 10^5$(过：指耗时小于 1s)，还是挺不错的。

然而，继续研究这几个编译器之间的差别意义不大，而循环展开/向量化这些本来应该是编译器自动完成的机械化的工作，如果让人来做意义也不太大 (当然，的确在很多真实的应用场景中这种手工的优化仍然是有必要的，但是如果这里为了一个非常低效的算法去优化常数，实在没什么价值)，因此我也没有继续优化下去了。

顺便说一下，希望贵课的服务器能早日更新编译器，一个现代的编译器对于代码速度的提升可能远远超过程序员的绞尽脑汁想出来的一些奇技淫巧。

5. 与串行化的计数排序程序相比，并行化计数排序的性能如何？与串行化的库函数 qsort 相比，并行化的计数排序性能如何？

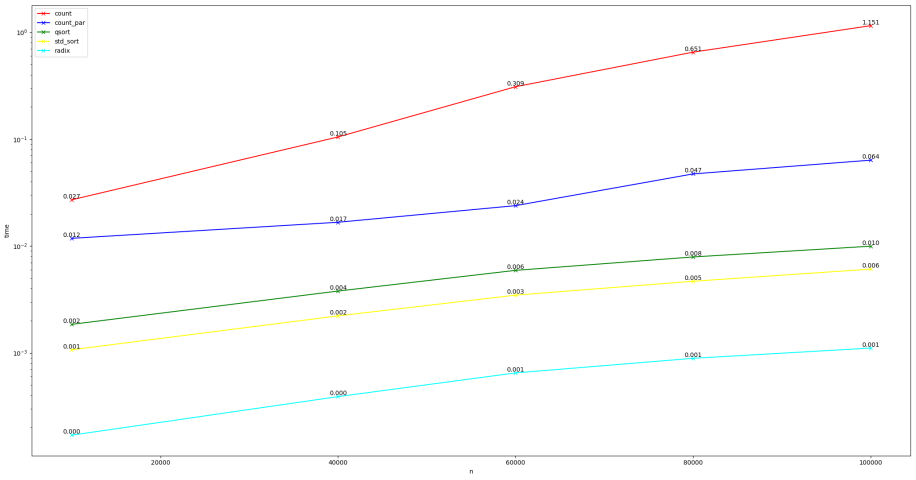
与串行化的计数排序程序相比，并行化计数排序的性能的提升很显著。

然而，这点常数的减小相比于它 $O(n^2)$ 的复杂度来说，实在没什么价值，因此它显著地慢于 $O(n \log n)$qsort 和 std::sort。而相比于真正高效的排序算法：$O(n \log U)$ 的基数排序来讲，它的速度更加显得可笑了。

测试结果如下：

	$n = 2 * 10^4$	$n = 4 * 10^4$	$n = 6 * 10^4$	$n = 8 * 10^4$	$n = 10^5$
count	0.02705s	0.10475s	0.30914s	0.65088s	1.15128s
count_par($th = 24$)	0.01180s	0.01668s	0.02386s	0.04726s	0.06355s
qsort	0.00185s	0.00379s	0.00592s	0.00790s	0.00996s
std::sort	0.00107s	0.00222s	0.00347s	0.00468s	0.00608s
radix	0.00017s	0.00039s	0.00065s	0.00089s	0.00111s

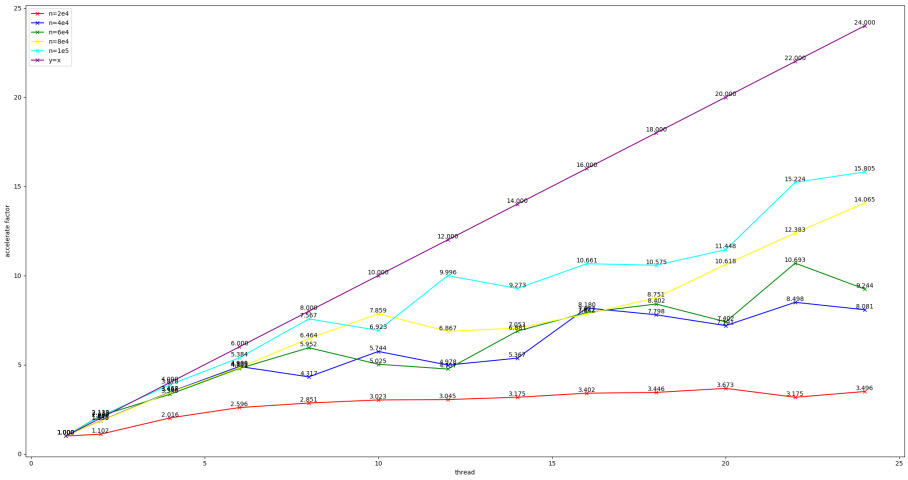
曲线图如下，注意时间是对数坐标的：



单独测试计数排序，而选择不同的线程数，测试结果如下：

	$n = 2 * 10^4$	$n = 4 * 10^4$	$n = 6 * 10^4$	$n = 8 * 10^4$	$n = 10^5$
$th = 1$	0.02905s	0.10707s	0.27759s	0.62348s	1.14682s
$th = 2$	0.02637s	0.05793s	0.12984s	0.33981s	0.54269s
$th = 4$	0.01441s	0.03087s	0.08311s	0.18087s	0.29576s
$th = 6$	0.01119s	0.02185s	0.05794s	0.12896s	0.21299s
$th = 8$	0.01019s	0.02480s	0.04664s	0.09645s	0.15156s
$th = 10$	0.00961s	0.01864s	0.05524s	0.07933s	0.16566s
$th = 12$	0.00954s	0.02151s	0.05835s	0.09079s	0.11473s
$th = 14$	0.00915s	0.01995s	0.04034s	0.08840s	0.12367s
$th = 16$	0.00854s	0.01309s	0.03502s	0.07951s	0.10757s
$th = 18$	0.00843s	0.01373s	0.03304s	0.07125s	0.10845s
$th = 20$	0.00791s	0.01489s	0.03750s	0.05872s	0.10018s
$th = 22$	0.00915s	0.01260s	0.02596s	0.05035s	0.07533s
$th = 24$	0.00831s	0.01325s	0.03003s	0.04433s	0.07256s

加速比曲线图如下：



可见 n 较大时，并行效率还是比较优秀的。不过再次申明，本身这个算法是非常低效的，因此这个效果也没什么价值。如果能 (高效地) 并行化基数排序，那么意义将大得多，不过一眼看上去，很明显难度也大得多。

3.2 一个更好的版本

llx 同学在微信群中提出可以让每个线程负责排序一段，然后将所有排好序的序列归并起来。其实这个方法并不是很不平凡，我之前之所以没有这么做，只是因为我认为这样没有意义：既然是让每个线程负责排序一段，为什么每一段要用这个低效的算法？完全可以每个线程都使用基数排序或者快速排序，再把结果归并起来，效率一定会更高。这样做的唯一意义在于，它强行保留了题目提供的这个低效的算法，表面上看起来更加符合题目的要求，然而实际想一下，凭什么说这个算法还算是计数排序？很明显归并才是它的灵魂所在。

好的，上面抱怨完了。简单讲下我的实现思路：为灵活性我使用了 pthread 来实现这个算法。每个线程先是负责排序自己的部分，然后开始树形规约，规约代码如下：

```
pthread_barrier_wait(&calc);
for (u32 i = 0; tid + (1 << i) < th && !(tid >> i & 1); ++i) {
  u32 right_tid = tid + (1 << (i + 1));
  u32 right = (right_tid >= th) ? n : each * right_tid;
  merge(a_off, a_off + (each << i), a + right, aux_off);
  pthread_barrier_wait(&reduce[i]);
}
```

设置路障个数的代码如下：

```
  u32 reduce_idx = 0;
  for (u32 i = th; i != 1; ++reduce_idx, i = (i + 1) >> 1) {
    pthread_barrier_init(&detail::reduce[reduce_idx], nullptr, i >> 1);
  }
```

如果助教还记得的话，会发现这代码和我上次提交的代码中的利用树形规约来加速直方图计算的代码基本是一样的，只是规约的地方需要额外处理一下边界情况。

顺便说一句，这个归并函数是我以前雕琢了很久得出的一个版本：

```
void merge(i32 *first, i32 *mid, i32 *last, i32 *aux) {
  i32 *pos1 = first, *pos2 = mid, *pos = aux;
  while (pos1 != mid && pos2 != last) {
    i32 a = *pos1, b = *pos2;
    i32 cmp = b < a;
    *pos++ = cmp ? b : a;
    pos1 += cmp ^ 1, pos2 += cmp;
  }
  memcpy(pos, pos1, sizeof(i32) * (mid - pos1)), pos += mid - pos1;
  memcpy(first, aux, sizeof(i32) * (pos - aux));
}
```

其核心在于循环内部的判断可以用条件传送指令来实现，有些人似乎觉得只要用了三元运算符就一定比 if 高效，这种想法是错误的，譬如说写：

```
  *pos++ = *pos2 < *pos1 ? *pos2++ : *pos1++;
```

和写

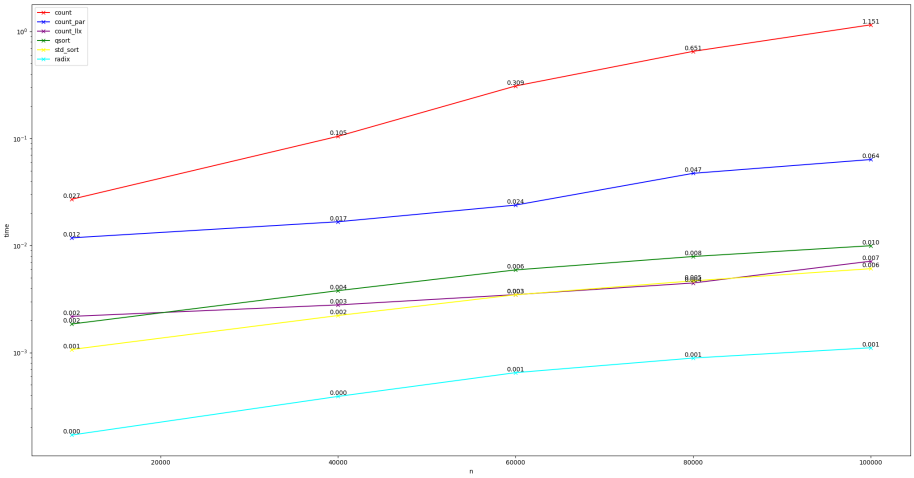
```
  if (*pos2 < *pos1) {
    *pos++ = *pos2++;
  } else {
    *pos++ = *pos1++;
  }
```

经测试在最新版的 gcc 和 clang 上生成的汇编都是一模一样的，都使用了分支指令，而我的版本则真的会被用条件传送指令来实现。当然这三种写法实际上都是等价的，只是目前的编译器还都没有能力看出来而已。

测试结果如下，其它方法的数据都与上一节的相同：

	$n = 2 * 10^4$	$n = 4 * 10^4$	$n = 6 * 10^4$	$n = 8 * 10^4$	$n = 10^5$
count	0.02705s	0.10475s	0.30914s	0.65088s	1.15128s
count_par($th = 24$)	0.01180s	0.01668s	0.02386s	0.04726s	0.06355s
count_llx($th = 24$)	0.00218s	0.00279s	0.00348s	0.00447s	0.00714s
qsort	0.00185s	0.00379s	0.00592s	0.00790s	0.00996s
std::sort	0.00107s	0.00222s	0.00347s	0.00468s	0.00608s
radix	0.00017s	0.00039s	0.00065s	0.00089s	0.00111s

曲线图如下，时间仍是对数坐标的：



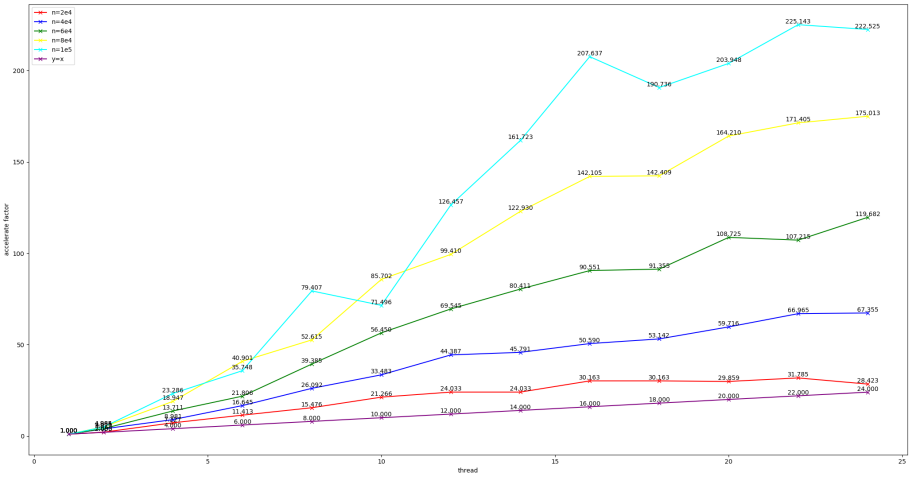
使用这个方法，选择不同的线程数，测试结果如下：

	$n = 2 * 10^4$	$n = 4 * 10^4$	$n = 6 * 10^4$	$n = 8 * 10^4$	$n = 10^5$
$th = 1$	0.02956s	0.11585s	0.30878s	0.66505s	1.14823s
$th = 2$	0.01412s	0.02988s	0.07663s	0.13395s	0.23511s
$th = 4$	0.00409s	0.01290s	0.02252s	0.03510s	0.04931s
$th = 6$	0.00259s	0.00696s	0.01416s	0.01626s	0.03212s
$th = 8$	0.00191s	0.00444s	0.00784s	0.01264s	0.01446s
$th = 10$	0.00139s	0.00346s	0.00547s	0.00776s	0.01606s
$th = 12$	0.00123s	0.00261s	0.00444s	0.00669s	0.00908s
$th = 14$	0.00123s	0.00253s	0.00384s	0.00541s	0.00710s
$th = 16$	0.00098s	0.00229s	0.00341s	0.00468s	0.00553s
$th = 18$	0.00098s	0.00218s	0.00338s	0.00467s	0.00602s

	$n = 2 * 10^4$	$n = 4 * 10^4$	$n = 6 * 10^4$	$n = 8 * 10^4$	$n = 10^5$
$th = 20$	0.00099s	0.00194s	0.00284s	0.00405s	0.00563s
$th = 22$	0.00093s	0.00173s	0.00288s	0.00388s	0.00510s
$th = 24$	0.00104s	0.00172s	0.00258s	0.00380s	0.00516s

注：这里测出来的 $th = 24$ 时似乎比上表中快了不少，这可能是因为每次测量时间的访存和分支规律都是一样的，所以运行次数变多时效率有一定提升。

加速比曲线图如下：



并行效率很轻松地就超过了 1。这很容易从算法复杂度角度分析出来。排序阶段的复杂度为 $O(\frac{n^2}{p^2})$；假设并行的线程数不超过核心数，则归并阶段的复杂度为

$$\frac{1}{p}n + \frac{2}{p}n + ... + \frac{2^{\lceil \log_2 p \rceil}}{p}n = O(n)$$

因此总复杂度为 $O(\frac{n^2}{p^2} + n)$。在 p 较小的时候可以达到接近平方级别的加速比。不过，这种加速是否真的是我们平常讨论的“加速”仍然存疑，本质上这样属于换了一种算法来实现。

如果 p 继续增大，超过核心数，可以预料一定范围内耗时仍会下降，因为 p 越大这个算法中计数排序的部分越少，归并排序的部分越多。

4 Programming

我依然没有管那个 main.c。本来 gen.c 中提供的代码是生成单精度浮点数的，但是根据助教的要求，还是使用了双精度浮点数。这挺可惜的，因为单精度浮点数的效率确实明显高于双精度浮点数。

运行了几次发现，主要的时间瓶颈在数据生成上，rand() 函数的效率非常低。于是我使用了Xorshift算法生成随机数。

4.1 openmp 版本

在代码中使用了 runtime 调度，这样比较方便测试不同的调度方式的速度差别。输入数据为 $n = 48000 = 24 * 2000$，$th = 24$，选择这个 n 主要是为了分块的数据更好看一些。

4.1.1 static 调度

块大小	1	10	20	30	40	50	60	70	80	90
耗时	0.274s	0.287s	0.284s	0.286s	0.285s	0.286s	0.288s	0.286s	0.283s	0.287s

块大小	100	200	300	400	500	600	700	800	900	1000
耗时	0.286s	0.285s	0.287s	0.283s	0.286s	0.290s	0.281s	0.290s	0.294s	0.286s

块大小	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
耗时	0.281s	0.290s	0.287s	0.294s	0.300s	0.301s	0.304s	0.308s	0.304s	0.285s

可以观察到一定的周期现象。理论分析如下：假定每个块的工作量基本相同，考虑到块大小不是很小，虽然有随机数 b_i 的因素存在，但是工作量的方差也不大，这一条件是近似成立的。在此基础上，所有线程总共需进行 $\lceil \frac{n}{blk*th} \rceil$ 轮计算，假设最后一轮计算中没有发生总剩余数目小于一个块的大小的现象，则总耗时正比于 $\lceil \frac{n}{blk*th} \rceil * blk$。从中可以预测，当 $blk * th$ 接近而小于一个能整除 n 的数时，耗时会较长，从表格中 900，1900 等处的数据可以基本验证，但是实际上它们之间的差距没有公式预言的这么大，我认为这可能是由以下原因导致的：

1. 每个块的工作量的差距不可忽视
2. 线程调度和同步的开销不可忽视
3. 线程之间的计算并非完全独立，例如同一个物理核心超线程出来的两个逻辑核心共享一部分计算单元，尤其是在密集的浮点运算下，它们之间的干扰不可忽视

4.1.2 dynamic 调度

块大小	1	10	20	30	40	50	60	70	80	90
耗时	0.174s	0.161s	0.160s	0.160s	0.160s	0.160s	0.161s	0.162s	0.161s	0.161s

块大小	100	200	300	400	500	600	700	800	900	1000
耗时	0.161s	0.161s	0.172s	0.164s	0.180s	0.194s	0.190s	0.168s	0.175s	0.182s

块大小	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
耗时	0.183s	0.198s	0.212s	0.220s	0.227s	0.236s	0.249s	0.261s	0.273s	0.283s

可见 dynamic 调度性能普遍比 static 调度好一些，这证明线程间的工作分配确实是比较不均匀的。而对于 $blk = 2000$ 的情形，二者的速度几乎一样，证明 dynamic 调度的额外开销不是很大。综合考虑调度开销和线程间的工作分配，在 $blk = 100$ 附近达到的效果最好。

许多问题下，dynamic 调度并不能和 block+cyclic 的 static 调度拉开差距，但是这个问题下差距比较明显，我认为这是因为在这个问题中 block+cyclic 的 static 调度与只有 block 的 static 调度没有本质的区别，每个线程的工作量仍然是随机生成的一系列整数，只是改变了一下具体负责哪一行而已。而很多其他问题中，工作量可能关于循环变量单调递增或递减，这样的 block+cyclic 的 static 调度就可以起到一定的负载均衡作用。

4.1.3 guided 调度

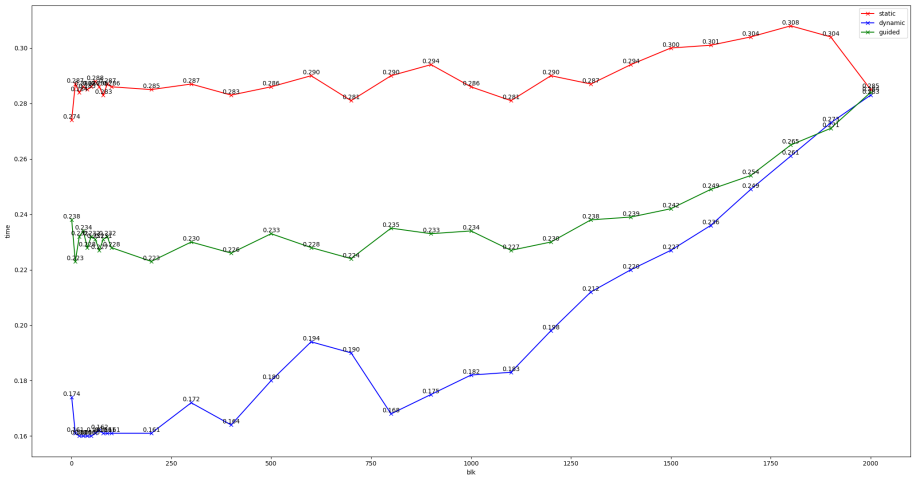
块大小	1	10	20	30	40	50	60	70	80	90
耗时	0.238s	0.223s	0.232s	0.234s	0.228s	0.232s	0.231s	0.227s	0.231s	0.232s

块大小	100	200	300	400	500	600	700	800	900	1000
耗时	0.228s	0.223s	0.230s	0.226s	0.233s	0.228s	0.224s	0.235s	0.233s	0.234s

块大小	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
耗时	0.227s	0.230s	0.238s	0.239s	0.242s	0.249s	0.254s	0.265s	0.271s	0.284s

可见 guided 调度速度普遍在 dynamic 调度和 static 调度之间，这也是符合对它的期望的。

三种调度方式的耗时曲线图如下：



4.2 pthread 版本

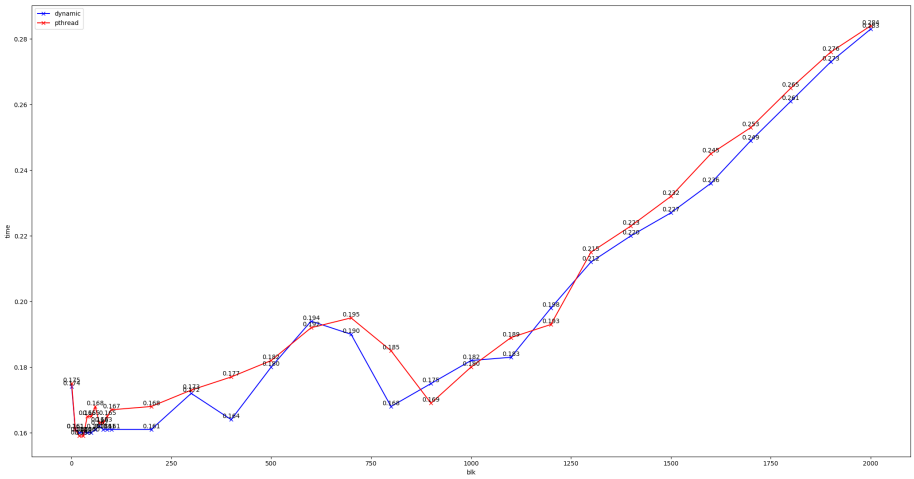
(我实现的)pthread 版本从任务分配的角度来看相当于 openmp 里的 dynamic 调度版本，也可以调节分块的大小，耗时与分块大小的关系如下：

块大小	1	10	20	30	40	50	60	70	80	90
耗时	0.175s	0.161s	0.159s	0.159s	0.165s	0.165s	0.168s	0.163s	0.163s	0.165s

块大小	100	200	300	400	500	600	700	800	900	1000
耗时	0.167s	0.168s	0.173s	0.177s	0.182s	0.192s	0.195s	0.185s	0.169s	0.180s

块大小	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
耗时	0.189s	0.193s	0.215s	0.223s	0.232s	0.245s	0.253s	0.265s	0.276s	0.284s

因为它和 openmp 的 dynamic 调度非常相似，所以这里把它和 openmp 的 dynamic 调度的耗时结果汇成了一张图：



可见二者的耗时也非常相似。