中山大学计算机学院本科生实验报告

课程名称：超级计算机组成原理　　任课教师：吴迪

年级	2019	专业（方向）	计科（超级计算）
学号	18324034	姓名	林天皓
开始日期	2021.4.12	完成日期	2021.4.18

一、实验题目

使用 openmp 进行计数排序

二、实验内容

1 实验原理

本次实验的主要内容是使用 openmp 进行并行计算编程。OpenMp 全称 Open Multi-Processing，是一套支持跨平台共享内存方式的多线程并发的编程 API。在本次实验中我们需要使用 openmp 的制导语句来对串行计数程序进行修改。

本次实验中使用串行的方式计数排序的流程如下：

对于数组中的每个数字，通过查找比他小的数字或者与他相同但是坐标靠前的数字有多少个，来直接获得该数字在最终排序好的数组中的位置，并填写。

串行的统计函数如下：

```
void Count_sort_serial(int a[], int n) {
   int i, j, count;
   int* temp = malloc(n*sizeof(int));

   for (i = 0; i < n; i++) {
      count = 0;
      for (j = 0; j < n; j++)
         if (a[j] < a[i])
            count++;
         else if (a[j] == a[i] && j < i)
            count++;
```

```
      temp[count] = a[i];
   }

   memcpy(a, temp, n*sizeof(int));
   free(temp);
}  /* Count_sort_serial */
```

我们需要对串行函数进行优化。

整个测试程序的运行流程如下：

由于为了一次性对多组数据进行测试，因此对 main 函数进行了一些改动。

Setp1：初始化

```
int main(int argc, char* argv[]) {
   int n, thread_count;
   int *a, *copy;
   double start, stop;

   /* please choose terms 'n', and the threads 'thread_count' here. */
   int ns[] = {1e3,1e4,1e5,1e6,2e6,4e6};
   int thread_counts[] = {2,4,8,18,36,54,72};
   /* Allocate storage and generate data for a */
   a = malloc(1e7*sizeof(int));
   Gen_data(a, 1e7);
   /* Allocate storage for copy */
   copy = malloc(1e7*sizeof(int));
```

通过为 a 数组申请空间，并生成随机数填满 a 数组。为了适应各种情况，这里直接先生

成 1e7 个适应后面对 n 不同的情况进行测试

Step2:进行后续四组测试

并行测试部分如下

```
   /* Allocate storage for copy */
   copy = malloc(1e7*sizeof(int));
fprintf(stderr,"start parallel\n");
for(int i=0;i<N_COUNT;i++){
   for(int j=0;j<THREAD_COUNT;j++){
   printf("==== parllar sort start n %d threads %d=====\n",ns[i],thread_counts[j])
;
   fprintf(stderr,"==== parllar sort start n %d threads %d=====\n",ns[i],thread_co
unts[j]);
   /* Parallel count sort */
   memcpy(copy, a, ns[i]*sizeof(int));
```

```
#  ifdef DEBUG
   Print_data(copy, n, "Original: Parallel qsort a");
#  endif
   GET_TIME(start);
   Count_sort_parallel(copy, ns[i], thread_counts[j]);
   GET_TIME(stop);
#  ifdef DEBUG
   Print_data(copy, n, "Sorted: Parallel sort a");
#  endif
   if (!Check_sort(copy, ns[i]))
      printf("Parallel sort failed\n");
   printf("Parallel run time: %e\n\n", stop-start);
   }
}
```

包括重置临时数组 copy，选择 thread_count 与 n 的数量，执行并行排序函数，完成后检查排序是否正确。

其他测试的流程相同，仅仅其中调用的函数不同,此处不在赘述。

除了上文中提到的串行排序以外，测试函数包含其他 3 种测试：

1.使用 openmp 并行排序，无优化

```
void Count_sort_parallel(int a[], int n, int thread_count) {
   int i, j, count;
   int* temp = malloc(n*sizeof(int));
#pragma omp parallel for num_threads(thread_count) private(i,j,count)
   for (i = 0; i < n; i++) {
      count = 0;
      for (j = 0; j < n; j++)
         if (a[j] < a[i])
            count++;
         else if (a[j] == a[i] && j < i)
            count++;
      temp[count] = a[i];
   }
   memcpy(a, temp, n*sizeof(int));
   free(temp);
}
```

此部分代码在串行的基础上加入

#pragma omp parallel for num_threads(thread_count)的制导语句，该语句的作用是将下列的

for 循环分配给 thread_count 个进程进行运算。

private(i,j,count) 声明下列 for 循环中的变量 i,j,count 为各个线程的私有变量，否则会被当做共享变量造成计算错误。

2.使用 AVX512 指令集优化 openmp 并行排序

在此基础上，我了解到可以使用 intel 的 avx512 指令集进行优化并行计算，下列函数为在并行排序的基础上改写为使用 AVX512 指令进行计算位置的排序函数。代码如下

```
void Count_sort_parallel_using_avx512(int a[], int n, int thread_count) {
   int i=0, j=0, count;
   int* temp = malloc(n*sizeof(int));
#pragma omp parallel for num_threads(thread_count) private(i,j,count)
//#pragma omp parallel for
   for (i = 0; i < n; i++) {
      __mmask16 res=0;
      __m512i ai,aj;
      count = 0;
      __mmask16 load = _cvtu32_mask16(-1);
      ai= _mm512_maskz_set1_epi32(load,a[i]);
      for(j=0;j+15<i;j+=16){
         aj= _mm512_maskz_expandloadu_epi32(load,a+j);
         res = _mm512_cmpge_epi32_mask(ai,aj);//a[i]>a[j]
         count+=__builtin_popcount(res);//avx imp less equal
      }
      for(;j<i;j++){
         if(a[j]<=a[i]){
            count++;
         }
      }
      for(j=i;j+15<n;j+=16){
         aj= _mm512_maskz_expandloadu_epi32(load,a+j);
         res = _mm512_cmpgt_epi32_mask(ai,aj);//a[i]>=a[j]
         count+=__builtin_popcount(res);//axv imp less than
      }
      for(;j<n;j++){
         if (a[j] < a[i])
            count++;
      }
      temp[count] = a[i];
   }
   memcpy(a, temp, n*sizeof(int));
   free(temp);
```

```
}
```

首先使用

```
__mmask16 res=0;
__m512i ai,aj;
```

声明 avx512 向量，__mmask16 实际上为 unsigned short，用来储存比较的运算结果。

```
__mmask16 load = _cvtu32_mask16(-1);
```

通过_cvtu32_mask16（-1）函数创建一个每一个位上都是 1 的__mmask16 变量。

```
ai= _mm512_maskz_set1_epi32(load,a[i]);
```

将 a[i]复制 32 遍，储存在__m512i 变量中，为下文中一次比较 16 位数字做准备。

```
for(j=0;j+15<i;j+=16){
   aj= _mm512_maskz_expandloadu_epi32(load,a+j);
   res = _mm512_cmpge_epi32_mask(ai,aj);//a[i]>a[j]
   count+=__builtin_popcount(res);//avx imp less equal
}
```

该部分首先将 a[j]到 a[j+16]通过_mm512_maskz_expandloadu_epi32 储存到 aj 向量中，然后使用_mm512_cmpge_epi32_mask 比较 a[i]与 a[j]的大小，将结果储存在 res 中。最后使用__builtin_popcount 函数计算 res 中 1 的个数，即为 a[i]>a[j]的个数。

```
for(;j<i;j++){
   if(a[j]<=a[i]){
      count++;
   }
}
```

该部分是为了剩余部分中不满 16 个数字的部分使用传统方式进行处理。

下面对 i<=j<n 进行处理。原理与上述相同，比较函数需要更改为_mm512_cmpgt_epi32_mask 进行计算得到 a[i]>=a[j]的个数。

```
for(j=i;j+15<n;j+=16){
   aj= _mm512_maskz_expandloadu_epi32(load,a+j);
   res = _mm512_cmpgt_epi32_mask(ai,aj);//a[i]>=a[j]
   count+=__builtin_popcount(res);//axv imp less than
}
for(;j<n;j++){
   if (a[j] < a[i])
      count++;
}
temp[count] = a[i];
```

最后将 a[i]放在排序后数组的对应位置中。

3.调用系统 qsort 进行排序

```
void Library_qsort(int a[], int n) {
   qsort(a, n, sizeof(int), My_compare);
}  /* Library_qsort */
```

直接调用 C 语言 stdlib 中的 qsort 库函数排序。

至此，我们完成了对实验代码的编写，下面运行测试。

三、实验结果

测试环境说明：

本次实验使用的 CPU 为 Intel(R) Xeon(R) Gold 6150 CPU @ 2.70GHz　18 核心 36 线程

编译器版本为 icc version 18.0.1 (gcc version 9.2.0 compatibility)

本次实验中并行测试使用了 7 种线程数量，分别为 2，4，8，18，36，54，72.

本次实验中测试使用了 6 种计算数量，分别为 1e3, 1e4, 1e5, 1e6, 2e6, 4e6。

编译运行

通过 icc 编译器进行编译并链接-fopenmp 参数，然后运行，并将结果输出到文件中。

```
gpu23 ~/asc21/lth/hpc/hw4 =Σ(((  つ^w^)つ icc hw4_all_test.c -o hw4 -fopenmp && ./hw4 > result2.txt
start parallel
==== parllar sort start n 1000 threads 2=====
==== parllar sort start n 1000 threads 4=====
==== parllar sort start n 1000 threads 8=====
==== parllar sort start n 1000 threads 18=====
==== parllar sort start n 1000 threads 36=====
==== parllar sort start n 1000 threads 54=====
==== parllar sort start n 1000 threads 72=====
==== parllar sort start n 10000 threads 2=====
==== parllar sort start n 10000 threads 4=====
==== parllar sort start n 10000 threads 8=====
==== parllar sort start n 10000 threads 18=====
==== parllar sort start n 10000 threads 36=====
==== parllar sort start n 10000 threads 54=====
==== parllar sort start n 10000 threads 72=====
==== parllar sort start n 100000 threads 2=====
==== parllar sort start n 100000 threads 4=====
```

图 1-编译运行结果（节选）

运行后查看结果文件。

```
gpu23 ~/asc21/lth/hpc/hw4 =Σ((( つ^ω^)つ cat result.txt
==== parllar sort start n 1000 threads 2=====
Parallel run time: 1.078986e-03

==== parllar sort start n 1000 threads 4=====
Parallel run time: 2.300685e-04

==== parllar sort start n 1000 threads 8=====
Parallel run time: 2.610273e-04

==== parllar sort start n 1000 threads 18=====
Parallel run time: 1.014899e-03

==== parllar sort start n 1000 threads 36=====
Parallel run time: 2.092991e-03

==== parllar sort start n 1000 threads 54=====
Parallel run time: 1.127086e-03

==== parllar sort start n 1000 threads 72=====
Parallel run time: 3.733488e-02

==== parllar sort start n 10000 threads 2=====
Parallel run time: 3.390301e-02
```

图 2-查看运行结果（节选）

同样的使用上次实验三中编写的 python 代码进行绘图，此处不再赘述。

运行结果表如下：

串行排序时间如下：

n	1000	10000	100000	1000000	2000000	4000000
串行	3.01E-04	2.78E-02	2.77E+00	2.78E+02	1.11E+03	4.73E+03

Qsort 函数库运行时间如下

n	1000	10000	100000	1000000	2000000	4000000
Qsort	9.00E-05	7.71E-04	9.44E-03	1.13E-01	2.37E-01	5.00E-01

无优化并行排序运行时间如下

线程数量/n	1000	10000	100000	1000000	2000000	4000000
2	1.08E-03	3.39E-02	1.51E+00	1.39E+02	5.56E+02	2.32E+03
4	2.30E-04	1.53E-02	6.97E-01	6.96E+01	2.78E+02	1.15E+03
8	2.61E-04	1.89E-02	3.47E-01	3.48E+01	1.39E+02	5.71E+02
18	1.01E-03	8.73E-03	1.55E-01	1.55E+01	6.19E+01	2.54E+02
36	2.09E-03	3.31E-03	8.95E-02	8.63E+00	3.44E+01	1.47E+02
54	1.13E-03	2.18E-03	2.56E-01	9.47E+00	3.50E+01	1.42E+02
72	3.73E-02	1.63E-03	1.67E-01	9.10E+00	3.55E+01	1.40E+02

表 1-无优化并行排序运行时间

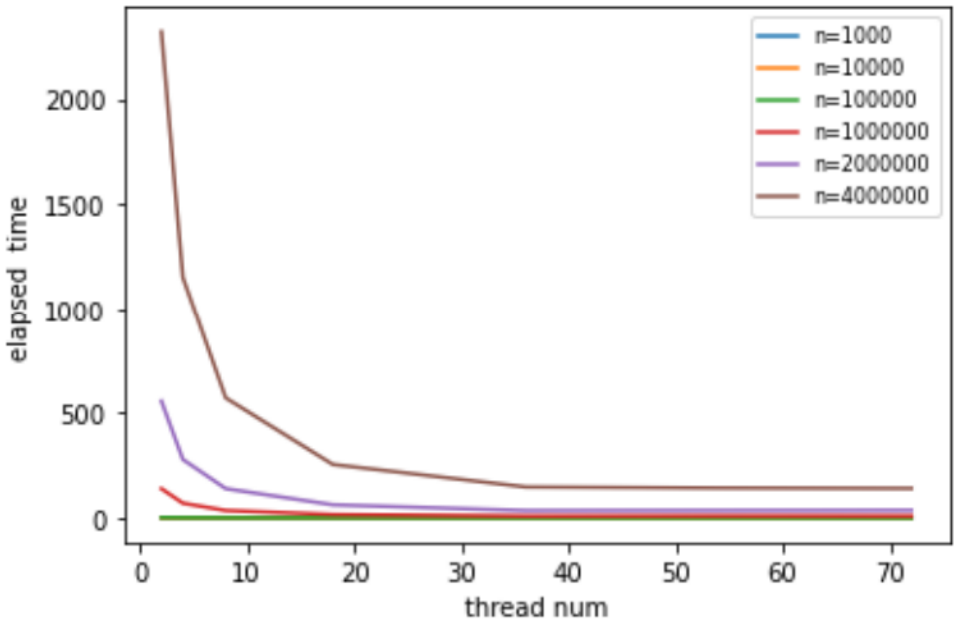

图 3 – 运算数量与运行时间关系

分析：n 越小，线程数量越小，运行时间越小

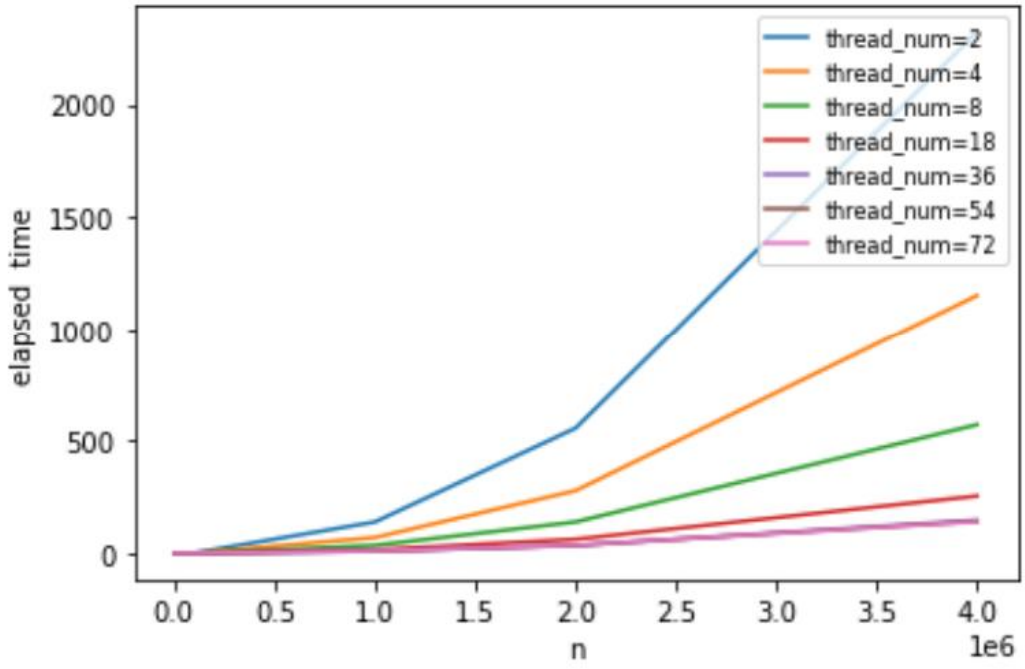

图 4 – 线程数量与运行时间关系

分析：运算数量和所用时间成近似二次关系，线程数量越少，所用时间越多。

无优化并行排序的加速比如下

线程数量/n	1000	10000	100000	1000000	2000000	4000000
2	2.79E-01	8.20E-01	1.84E+00	2.00E+00	2.00E+00	2.04E+00
4	1.31E+00	1.82E+00	3.98E+00	3.99E+00	3.99E+00	4.12E+00
8	1.15E+00	1.47E+00	7.98E+00	7.99E+00	7.98E+00	8.30E+00
18	2.97E-01	3.18E+00	1.79E+01	1.79E+01	1.79E+01	1.86E+01
36	1.44E-01	8.41E+00	3.10E+01	3.22E+01	3.23E+01	3.21E+01
54	2.67E-01	1.27E+01	1.08E+01	2.93E+01	3.17E+01	3.33E+01
72	8.06E-03	1.71E+01	1.66E+01	3.05E+01	3.13E+01	3.39E+01

表 2-无优化排序的加速比

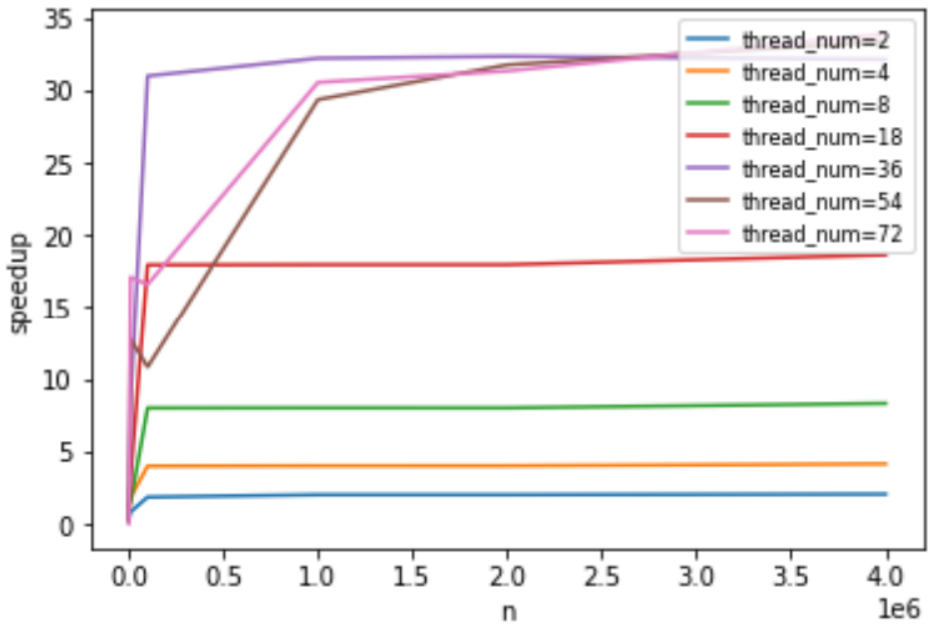

图 5－ 线程数量与加速比关系

分析:

由图可得当线程数量没有超过 CPU 线程数量时候，线程数越多，加速比越大。

对于同一线程数，n 越大，加速比越大。

对于同一线程数量，当 n 足够大时，加速比约等于线程数时不再增加。

当线程数量大于 CPU 线程数量时，加速比也不会超过 CPU 线程数量。

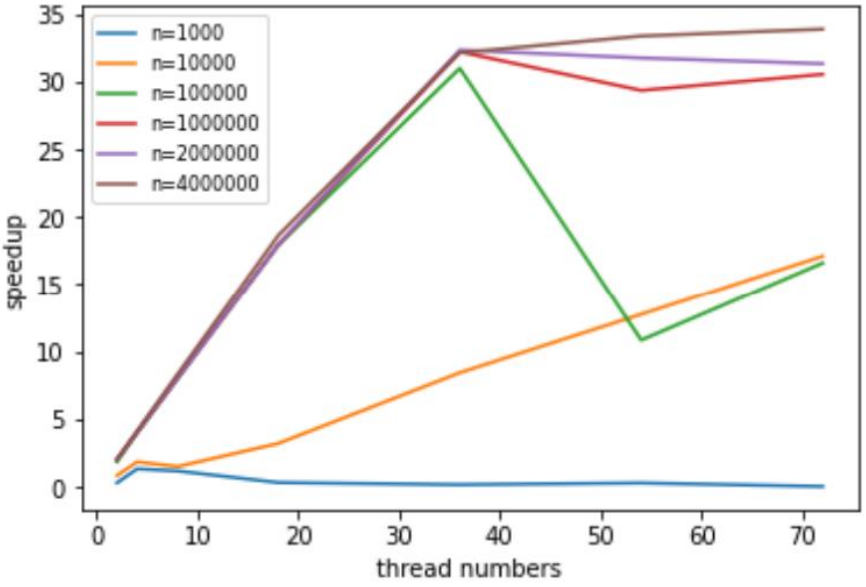

图 6－ 线程数量与加速比关系

分析：由图可见当 n 一定时，加速比随着线程数先增加后减少，对于每一个 n 都有一个最优的线程数量，这个数量随着 n 的增加而增加。超过这个线程数量以后，因为过多线程调用的开销，会导致加速比下降。不过在实验中发现对于非 CPU 线程数的整数倍且 n 较小时（图中绿色线），加速比下降得更多。

无优化并行排序的加速效率如下

线程数量/n	1000	10000	100000	1000000	2000000	4000000
2	1.39E-01	4.10E-01	9.18E-01	9.99E-01	9.99E-01	1.02E+00
4	3.27E-01	4.56E-01	9.94E-01	9.98E-01	9.98E-01	1.03E+00
8	1.44E-01	1.84E-01	9.97E-01	9.99E-01	9.98E-01	1.04E+00
18	1.65E-02	1.77E-01	9.96E-01	9.97E-01	9.97E-01	1.03E+00
36	3.99E-03	2.34E-01	8.60E-01	8.94E-01	8.98E-01	8.92E-01
54	4.95E-03	2.36E-01	2.00E-01	5.43E-01	5.88E-01	6.18E-01
72	1.12E-04	2.37E-01	2.30E-01	4.24E-01	4.35E-01	4.70E-01

表 3 -无优化并行排序的加速效率

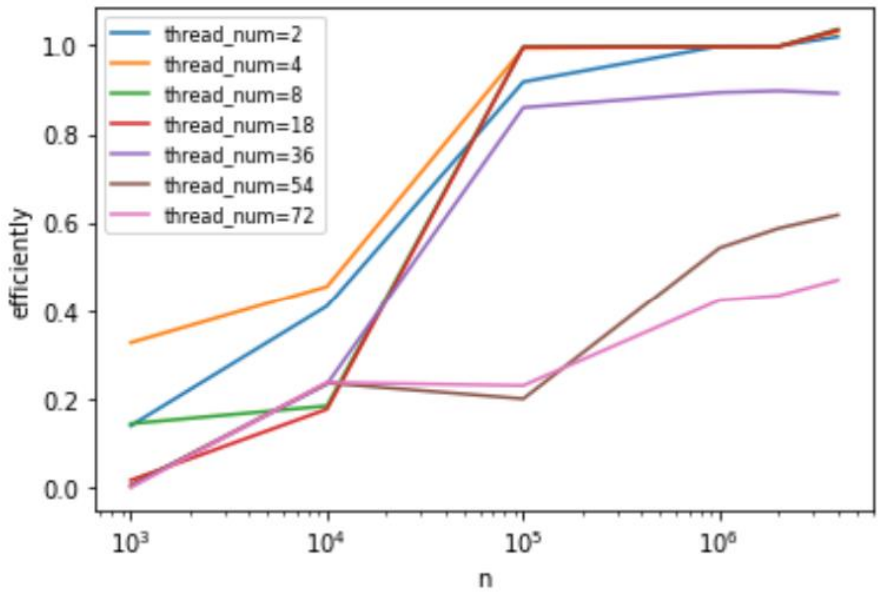

图 7 – 线程数量与效率关系

由图可见，当线程数越小时，效率越高。当线程数量小于 CPU 线程数量时，n 足够大时效率近似等于 1.当线程数量大于 CPU 线程数量时，最大效率随着线程数增多而降低。

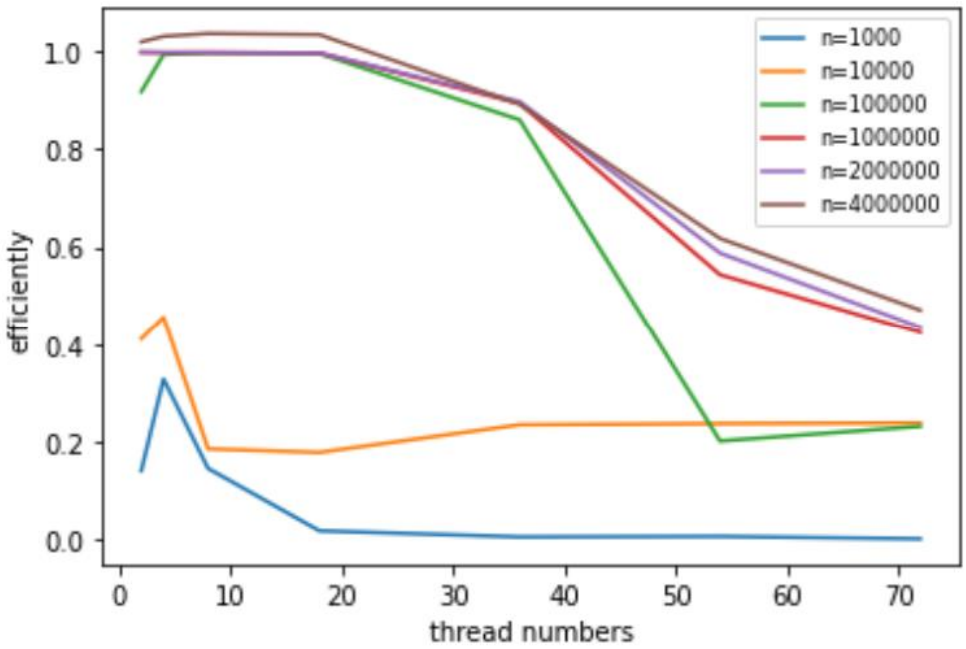

图 8 – 运算数量与效率关系

分析：n 越大，效率越高。线程数量越多，效率越低。当线程数量超过 CPU 线程数量时候效率大幅度下降。

AVX512 优化并行排序运行时间如下

线程数量/n	1000	10000	100000	1000000	2000000	4000000
2	3.17E-04	2.95E-02	1.29E+00	1.19E+02	4.76E+02	2.01E+03
4	1.68E-04	2.26E-02	5.87E-01	5.93E+01	2.37E+02	1.00E+03
8	1.12E-04	4.41E-03	2.94E-01	2.97E+01	1.19E+02	4.98E+02
18	1.18E-04	1.04E-02	1.31E-01	1.32E+01	5.30E+01	2.28E+02
36	1.98E-04	5.76E-03	1.32E-01	6.89E+00	2.72E+01	1.08E+02
54	2.38E-02	1.54E-03	1.75E-01	7.49E+00	2.96E+01	1.12E+02
72	1.86E-02	1.72E-03	1.90E-01	7.39E+00	2.86E+01	1.10E+02

表 4 -AVX512 优化并行排序的时间

对比无优化 AVX512 减少的时间比例

线程数量/n	1000	10000	100000	1000000	2000000	4000000
2	7.06E-01	1.29E-01	1.43E-01	1.46E-01	1.44E-01	1.33E-01
4	2.69E-01	-4.80E-01	1.57E-01	1.47E-01	1.46E-01	1.25E-01
8	5.71E-01	7.66E-01	1.53E-01	1.46E-01	1.46E-01	1.28E-01
18	8.84E-01	-1.87E-01	1.52E-01	1.45E-01	1.44E-01	1.01E-01
36	9.05E-01	-7.40E-01	-4.78E-01	2.02E-01	2.10E-01	2.69E-01
54	-2.01E+01	2.97E-01	3.18E-01	2.09E-01	1.55E-01	2.09E-01
72	5.01E-01	-5.34E-02	-1.35E-01	1.88E-01	1.95E-01	2.11E-01

表 5 – AVX512 优化并行排序相对减少时间比例

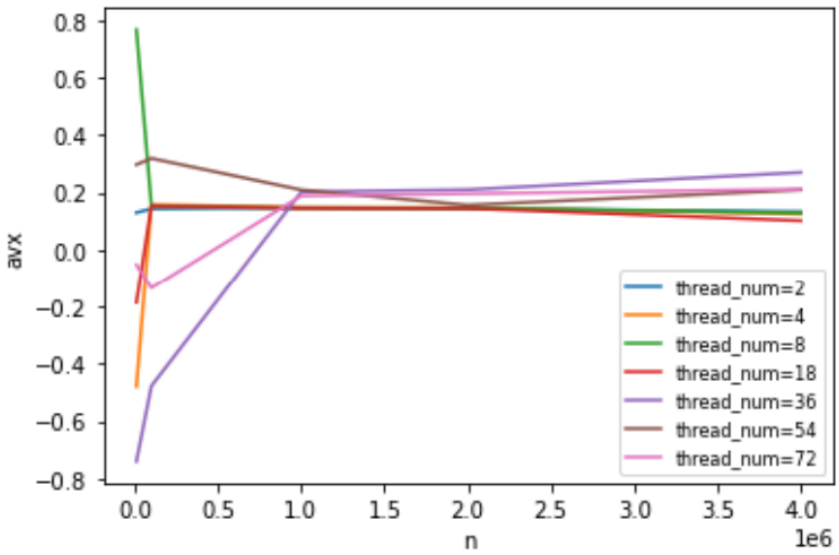

图 9 – 线程数量与 AVX512 优化的时间效率关系

分析：在线程数较大且 n 较大时候，使用 AVX512 优化几乎可以达到稳定的 20%的时间减少，而在 n 较小时候，AVX 的加速不稳定，且有可能出现减速现象。

实验感想

本次实验是使用 openmp 进行并行计数排序的一次实验，openmp 使用共享内存的方式进行并行计算，相比较与 MPI 并行编程，所编写的代码更简单，更容易并行化，也不需要 MPI 一样强大的网络带宽，缺点是不能共享内存的部分需要每个线程都复制一份，因此消耗内存较大，例如在本次实验中使用 72 线程时经过计算，使用的内存达到了 4GB，同时共享内存也为跨节点并行带来了困难。

在使用 openmp 之外，还进一步尝试了使用 AVX512 指令进行加速运算。对于长度为 16 整数倍的运算，使用 AVX512 大大减少了指令数量。

```
ai= _mm512_maskz_set1_epi32(load,a[i]);
for(j=0;j+15<i;j+=16){
    aj= _mm512_maskz_expandloadu_epi32(load,a+j);
    res = _mm512_cmpge_epi32_mask(ai,aj);//a[i]>a[j]
    count+=__builtin_popcount(res);//avx imp less equal
}
```

图 10 – 调用 AVX512 函数

通过查阅 intel 文档，只需要一个时钟周期就可以完成对 16 个个 32 位整数的比较操作。比使用传统 32 位指令的速度得到的成倍的提升。

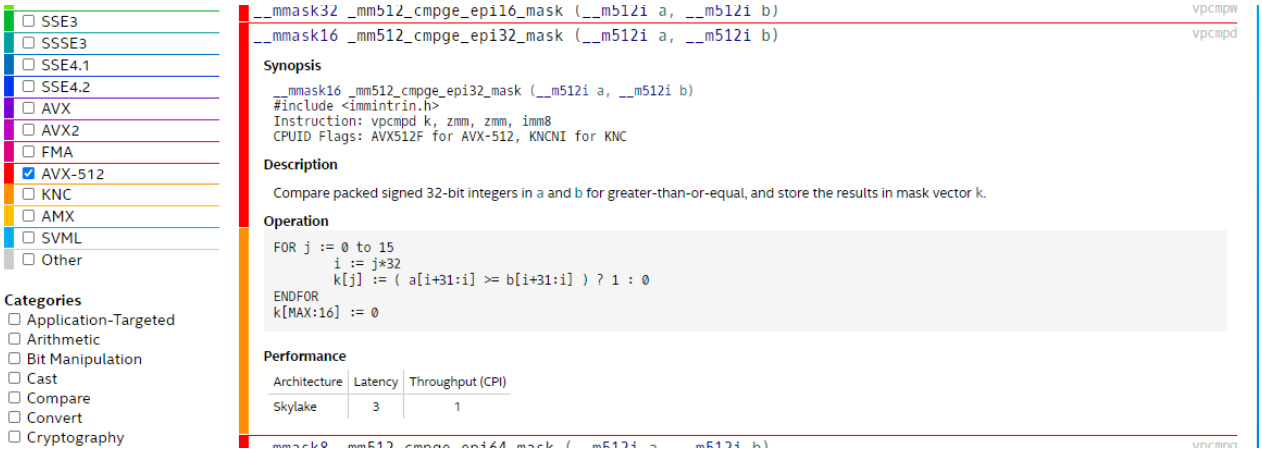

图 11 – intel 说明书

同时我们也可以通过 godbolt 查看对应生成的一系列 v 开头的向量指令。即为向量化指令。

```
..B1.21:                        # Preds ..B1.21 ..B1.20
        movzx     eax, WORD PTR [-684+rbp]                      #19.14
        mov       edx, DWORD PTR [-604+rbp]                     #19.14
        movsxd    rdx, edx                                      #19.14
        imul      rdx, rdx, 4                                   #19.14
        add       rdx, QWORD PTR [-528+rbp]                     #19.14
        kmovw     k1, eax                                       #19.14
        vpexpandd zmm0{k1}{z}, ZMMWORD PTR [rdx]                #19.14
        vmovups   ZMMWORD PTR [-176+rbp], zmm0                  #19.14
        vmovups   zmm0, ZMMWORD PTR [-176+rbp]                  #19.14
        vmovups   ZMMWORD PTR [-336+rbp], zmm0                  #19.10
        vmovups   zmm0, ZMMWORD PTR [-400+rbp]                  #20.16
        vmovups   zmm1, ZMMWORD PTR [-336+rbp]                  #20.16
        vpcmpd    k0, zmm0, zmm1, 5                             #20.16
        kmovw     eax, k0                                       #20.16
        mov       WORD PTR [-682+rbp], ax                       #20.16
        movzx     eax, WORD PTR [-682+rbp]                      #20.16
        mov       WORD PTR [-688+rbp], ax                       #20.10
```

图 12－ 翻译得到的 simd 指令

最终经过实测，在数据量较大的时候，使用 SIMD 指令可以为程序带来约 20%的性能提升，是一种并行化的优化方法。