《并行程序设计》
课程大作业

学号：201736721010

姓名：宋雨杭

班级：软件工程 4 班

任课教师：汤德佑

2020/07/14

一、问题描述

在数据库中经常会有 group by 这种操作，需要按某个字段进行分组，并统计各个分组的计数。有时 group by 需要执行的记录条数非常多，传统的串行算法就会很慢很慢，这时就需要探寻一种并行的方法，使得这个操作能使用多个核心、甚至多个节点并行计算，加快数据的获取。

本次大作业将上述在数据库中的 group by 操作，抽象为在给定 N 个字符串中，进行并行化的分组操作的问题。要求求出所有分组的计数，并统计总分组数。

二、求解方法

1. 读写部分：

对比测试了三种读写方式

(1) Scanf()

```
31      freopen(file_name, "r", stdin);
32      for (int i = 0; i < N; ++i)
33      {
34          scanf("%s", all_string[i]);
35      }
```

读写 100M 数据，大约需要 6.6s

```
song@CoolFishS5:~/PC2020/term_project$ time ./1 1 100000000 100M.txt

real    0m6.659s
user    0m6.234s
sys     0m0.424s
```

(2) Fgets()

```
31      freopen(file_name, "r", stdin);
32      for (i = 0; i < N; ++i)
33      {
34          scanf("%s", all_string[i]);
35      }
```

读写 100M 数据，大约需要 2.6s

```
song@CoolFishS5:~/PC2020/term_project$ time ./1 1 100000000 100M.txt

real    0m2.613s
user    0m2.173s
sys     0m0.440s
```

(3) Gets()

读写 100M 数据，大约需要 2.8s

```
song@CoolFishS5:~/PC2020/term_project$ time ./1 1 100000000 100M.txt

real    0m2.804s
user    0m2.403s
sys     0m0.400s
```

对比发现，fgets 更安全而且更快。原以为的 gets()函数居然更慢，而且编译时系统也会提示这个函数非常危险（容易发生内存溢出），不建议使用。

```
song@CoolFishS5:~/PC2020/term_project$ gcc 1.c -fopenmp -o 1
1.c: In function ‘main’:
1.c:40:9: warning: implicit declaration of function ‘gets’; did you mean ‘fgets’? [-Wimplicit-function-declaration]
         gets(all_string[i]);
         ^~~~
         fgets
/tmp/ccNtGq6i.o: In function `main':
1.c:(.text+0xc4): warning: the `gets' function is dangerous and should not be used.
```

因此，我采用 fgets()作为读入的函数，速度最快且最安全。有一点要注意的是 fgets 会将换行符读入，在本次实验中，这一点不影响我们的最终结果正确性。

因为在运算中涉及到位运算，需要将 2 个占用 16 位的 char 解析为占用 32 位的 unsinged int，我添加了一个 char_to_uint(char *x)函数，通过位运算将两个 char 解析为一个 unsigned int。

```
75    inline unsigned int char_to_uint(char *x)
76    {
77        unsigned int re;
78        re = ( ((unsigned int)*x) << 8 | *(x+1) );
79        return re;
80    }
```

同时，为了测试上述转换是否正确，我写了两个函数，通过位运算来打印 char 和 unsinged int 的二进制值。

```
86   void char_to_binary(char x)
87   {
88       int i;
89       for (i = 7; i >= 0; --i)
90       {
91           if (x & (1 << i))
92               printf("1");
93           else
94               printf("0");
95       }
96   }
```

```
99    void unsigned_int_to_binary(unsigned int x)
100   {
101       int i;
102       for (i = 15; i >= 0; --i)
103       {
104           if (x & (1 << i))
105               printf("1");
106           else
107               printf("0");
108       }
109   }
```

```
58   void cal_cnt(int rounds)
59   {
60       int i, j;
61       int offset;
62       unsigned int now;
63       offset = rounds * 2;
64
65       for (i = 0; i < N; ++i)
66       {
67           now = char_to_uint(all_string[i] + offset);
68           printf("%u\n", now);
69           char_to_binary(all_string[i][offset]);
70           char_to_binary(all_string[i][offset + 1]);
71           printf("\n");
72           unsigned_int_to_binary(now);
73           printf("\n");
74       }
75   }
```

可以看到转换完全正确，两个 char 拼接起来的二进制值刚好等于转换后 unsigned int 的二进制值。

```
song@CoolFishS5:~/PC2020/term_project$ ./group_by 1 5 10M.txt
16967
0100001001000111
0100001001000111
17990
0100011001000110
0100011001000110
16705
0100000101000001
0100000101000001
18503
0100100001000111
0100100001000111
17217
0100001101000001
0100001101000001
```

2. 程序结构

一开始我是想的将字符串分段分组后，每 16 位都做一次分组算法，代码如下：

```
void cal_cnt(int round)
{
    int i, j;
    int offset;
    unsigned int now;
    int sum;

    offset = round * 2;

    memset(cnt, 0, (MAX_THREAD + 1) * MAX_CNT);

    for (i = 0; i < N; ++i)
    {
        now = char_to_uint(all_string[I[round][i]] + offset);

        cnt[0][now] += 1;
    }

    for (i = 0; i < MAX_CNT; ++i)
        cnt[1][i] = cnt[0][i];

    sum = 0;
    for (i = 0; i < MAX_CNT; ++i)
    {
        sum += cnt[1][i];
        cnt[1][i] = sum;
    }
```

```
    for (i = 0; i < MAX_CNT; ++i)
        cnt[0][i] = cnt[1][i] - cnt[0][i];


    for (i = 0; i < N; ++i)
    {
        now = char_to_uint(all_string[I[round][i]] + offset);
        I[round + 1][cnt[0][now]++] = I[round][i];
    }
}
```

然后发现这样并不可取，因为这样每轮得到的分组结果都是完全独立的，无法快速的合并。

然后我考虑只在第一次进行分段分组，然后得到了很多小的分组后，对这些小的分组直接进行快速排序。

因为在进行一次分段分组后，分组的数量通常会远远超过总核数，而每个分组间相互独立，刚好符合并行化计算的条件。可以给每个核心分配任务，这样就实现了并行化计算。

3. 串行的程序

读入数据

```
    FILE *fh = fopen(file_name, "r");

    for (i = 0; i < N; ++i)
        fgets(all_string[i], 12, fh);

    fclose(fh);
```

将每个字符串前 32 字节提出来，然后统计累加到 cnt 中

```
    for (i = 0; i < N; ++i)
    {
        now = char_to_uint(all_string[i] + offset);
        cnt[0][now] += 1;
    }
```

然后对其求前缀和

```
    sum = 0;
    for (i = 0; i < MAX_CNT; ++i)
    {
        sum += cnt[1][i];
        cnt[1][i] = sum;
    }
```

然后向上做减法，以求出每个前缀值所在的位置

```
    for (i = 0; i < MAX_CNT; ++i)
        cnt[0][i] = cnt[1][i] - cnt[0][i];
```

将位置下标赋给 I 来保存

```
    for (i = 0; i < N; ++i)
    {
        now = char_to_uint(all_string[i] + offset);
        I[cnt[0][now]++] = i;
    }
```

然后对于每一个分组，都用快速排序算法进行排序。

```
    for (i = 1; i < MAX_CNT; ++i)
    {
        if (cnt[1][i] - cnt[1][i - 1] > 1)
        {
            quick_sort(cnt[1][i - 1], cnt[1][i] - 1);
        }
    }
```

这时候所有字符串在自己的分区内都是有序的了。然后扫描一遍进行统计输出即可。

```
fh = fopen("result.txt", "w");

    int final_result = 0;
    char *pres = all_string[I[0]];
    char temp_num[20];
    int pre_pos = 0;
    for (i = 1; i < N; ++i)
    {
        if (strcmp(pres, all_string[I[i]]))
        {
            fputs(strcat(strcat(itoa(i - pre_pos, temp_num), " "), pres
), fh);
            pres = all_string[I[i]];
            pre_pos = i;
            final_result += 1;
        }
    }
    fputs(strcat(strcat(itoa(N - pre_pos, temp_num), " "), pres), fh);
    fputs(itoa(final_result + 1, temp_num), fh);
    fputs("\n", fh);
    fclose(fh);
```

过程中需要手写一个快排函数

```
void swap(int a, int b)
{
    int swap_tmp = I[a];
    I[a] = I[b];
    I[b] = swap_tmp;
}
int partition(int low, int high)
{
    char *base = all_string[I[low]];
    while (low < high)
    {
        while (low < high && strcmp(all_string[I[high]], base) > 0)
        {
            high--;
        }
        swap(low, high);
        while (low < high && strcmp(all_string[I[low]], base) <= 0)
        {
            low++;
        }
        swap(low, high);
    }
    return low;
}

void quick_sort(int low, int high)
{

    if (low < high)
    {
        int nbase = partition(low, high);
        quick_sort(low, nbase - 1);
        quick_sort(nbase + 1, high);
    }
}
```

首先验证串行程序的正确性。

```
song@CoolFishS5:~/PC2020/term_project$ gcc group_by.c -mcmodel=large -o group_by
song@CoolFishS5:~/PC2020/term_project$ time ./group_by 1 10000000 10M.txt

real    0m6.966s
user    0m6.289s
sys     0m0.264s
```

编译后让其对 10M 的数据集进行分组，得到的结果会保存到./result.txt 中。这里

只需要查看其最终得到的分组数即可知道结果正确性。

```
$ tail -n 1 result.txt
```

```
song@CoolFishS5:~/PC2020/term_project$ tail -n 1 result.txt
6356524
```

然后再通过系统自带的工具计算分组

```
$ sort 10M.txt | uniq -c | sort -nr | wc -l
```

```
song@CoolFishS5:~/PC2020/term_project$ sort 10M.txt | uniq -c | sort -nr | wc -l
6356524
```

得到的分组数完全一致，说明串行程序运行是正确的。

下面就要分别对其实现 OpenMP 并行化、MPI 并行化和 OpenMP+MPI 并行化。

4. OpenMP 并行化

将数据按线程数等分，然后每个线程统计自己那部分的前缀出现次数。

```
#pragma omp parallel num_threads(thread_count) default(none) \
    shared(cnt, all_string, N) private(i, now)
    {
        int my_rank = omp_get_thread_num();
        int thread_count = omp_get_num_threads();
        int start = my_rank * N / thread_count;
        int end = (my_rank + 1) * N / thread_count;
        for (i = start; i < end; ++i)
        {
            now = char_to_uint(all_string[i]);
            cnt[my_rank][now] += 1;
        }
    }
```

并行的将上述结果求和

```
#pragma omp parallel for num_threads(thread_count) default(none) \
    shared(thread_count, cnt) private(i, j)
    for (i = 0; i < MAX_CNT; ++i)
        for (j = 0; j < thread_count; ++j)
            cnt[thread_count][i] += cnt[j][i];
```

并行的向上做减法，求出每个值的位置，方便后面分配 I 数组

```
#pragma omp parallel for num_threads(thread_count) default(none) shared
(thread_count, cnt) private(i, j)
    for (i = 0; i < MAX_CNT; ++i)
    {
        for (j = thread_count - 1; j >= 0; --j)
```

```
            cnt[j][i] = cnt[j + 1][i] - cnt[j][i];
    }
```

并行的去分配下标数组

```
#pragma omp parallel num_threads(thread_count) default(none) \
    shared(thread_count, cnt, I, all_string, N) private(i, now)
    {
        int my_rank = omp_get_thread_num();
        int thread_count = omp_get_num_threads();
        int start = my_rank * N / thread_count;
        int end = (my_rank + 1) * N / thread_count;

        for (i = start; i < end; ++i)
        {
            now = char_to_uint(all_string[i]);
            I[cnt[my_rank][now]++] = i;
        }
    }
```

到这一步即完成了前缀的分组。

接着并行的对每个分组进行快速排序

```
#pragma omp parallel for num_threads(thread_count) schedule(dynamic) \
    shared(thread_count, cnt, I) private(i)
    for (i = 1; i < MAX_CNT; ++i)
        if (cnt[thread_count][i] - cnt[thread_count][i - 1] > 1)
            quick_sort(cnt[thread_count][i - 1], cnt[thread_count][i] -
 1);
```

这里一定要设置 schedule(dynamic)动态调度，否则会因为每个分组的任务量不均匀，导致程序运行缓慢。

完成这些改动后就实现了 OpenMP 并行化。

5. MPI 并行化

进行 MPI 并行化时，首先添加 MPI 的初始化函数，并获取当前节点的编号和总结点数。

```
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
```

刚开始我考虑在主节点上将数据按前 32 位的大小划分后，直接将每个节点所要计算的部分传输给该节点。后来发现这样并不可取，因为直接按前 32 位等分的话，一定会出现部分节点分到的数据极多而部分节点极少的这种情况。所以 MPI 并行的时候，也一定需要对其求一个分组的过程，求前缀和，在求前缀和的

过程中划分数据。基于这种考虑，我决定直接广播数据。然后在计算完之后，只需要回收下标数组 I 的数据，即可统计结果。

广播数据：

```
    /*
    Do the distribute
    */
    MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&thread_count, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(all_string[0], 12 * N, MPI_CHAR, 0, MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);
    time_send_done = MPI_Wtime();
```

将快排的部分改为：

```
    for (i = seg[mpi_rank]; i < seg[mpi_rank + 1]; ++i)
        if (cnt[thread_count][i] - cnt[thread_count][i - 1] > 1)
            quick_sort(cnt[thread_count][i - 1], cnt[thread_count][i] -
 1);
```

使得每个节点只用计算自己的那部分。

然后收集结果 I

```
    MPI_Gatherv(I + displs[mpi_rank], recvCount[mpi_rank], MPI_INT,
                &finalI, recvCount, displs, MPI_INT, 0, MPI_COMM_WORLD)
;
    memcpy(I, finalI, sizeof(unsigned int) * MAXN);
```

0 号节点收到所有数据后，扫面一边即可统计出结果。

```
/*
    Label the result
    */
    if (mpi_rank == 0)
        final_result = get_result();

    time_label_data_done = MPI_Wtime();
```

通过 0 号节点将数据结果写入硬盘并输出摘要信息。

```
/*
    Write the result into file
    */
    FILE *fh = fopen("result.txt", "w");
    if (mpi_rank == 0)
    {
        fprintf(fh, "%d\n", final_result);
```

```
        for (i = 0; i < final_result; ++i)
            fprintf(fh, "%d %s", final_label[i], final_string[i]);
    }
    fclose(fh);
    time_write_done = MPI_Wtime();

    /*
    Print the summary
    */
    if (mpi_rank == 0)
        printf("Group By Result: %d\n\n\
Read Time: %f\nSend Data Time: %f\n*Run Time: %f\nRecive Data Time: %f\
n\
Label Time: %f\nWrite Time: %f\n*Total: %f\n\n\
Detailed results have written into file ./result.txt\n",
               final_result,
               time_read_done - time_start,
               time_send_done - time_read_done,
               time_calculate_done - time_send_done,
               time_receive_done - time_calculate_done,
               time_label_data_done - time_receive_done,
               time_write_done - time_label_data_done,
               time_write_done - time_start);
    MPI_Finalize();
```

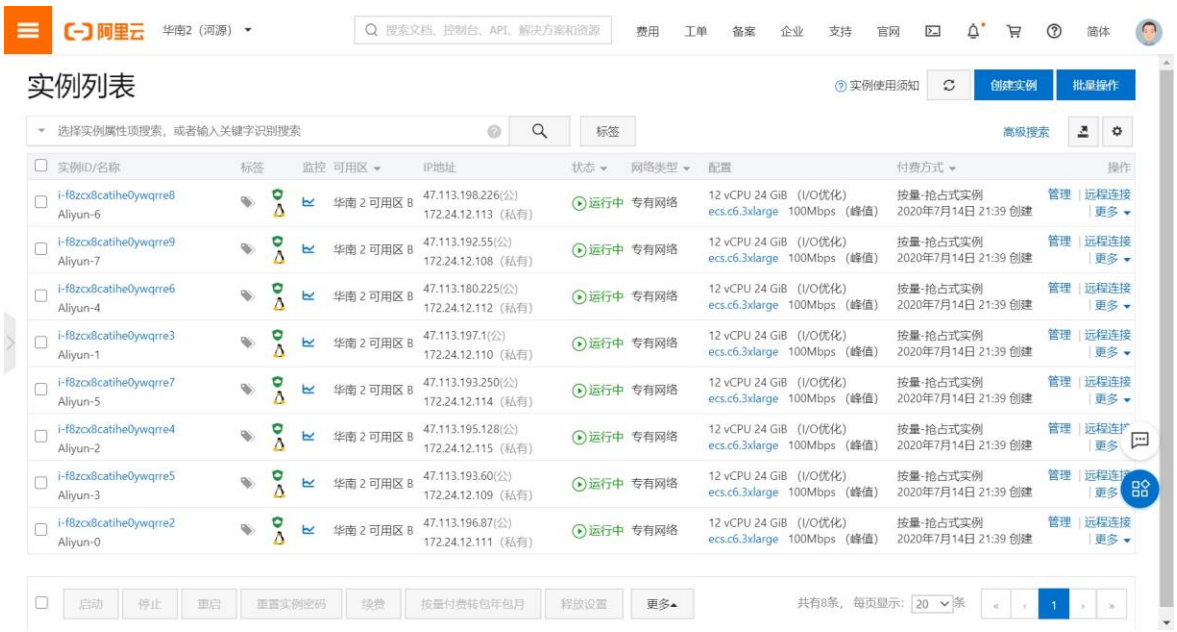
6. OpenMP + MPI 并行化

同时保留上面 OpenMP 和 MPI 部分对代码的修改即可。

三、实验结果截图及性能分析

在阿里云上搭建 MPI 集群并进行测试。

首先创建 8 台 12 核 24G 的计算型实例，并将其置于同一交换机下。这样服务器两两之间能使用 4Gbps 的内网带宽，能够保障 MPI 实验的顺利进行。



然后连接上这 8 台服务器，向其 hosts 文件中追加 8 台服务器的内网 IP 地址，方便后面用主机名替代 IP 直接连接，易于识别。

```
echo 'localhost 127.0.0.1
172.24.12.111 Aliyun-0
172.24.12.110 Aliyun-1
172.24.12.115 Aliyun-2
172.24.12.109 Aliyun-3
172.24.12.112 Aliyun-4
172.24.12.114 Aliyun-5
172.24.12.113 Aliyun-6
172.24.12.108 Aliyun-7
' >> /etc/hosts
```

然后向每个节点中添加用户 mpiuser。因为在创建服务器的时候已经生成了公钥，所以可以直接把 root 用户的公钥拷贝到该用户下。

```
# adduser mpiuser
# rsync --archive --chown=mpiuser:mpiuser ~/.ssh /home/mpiuser
```

然后将私钥上传到 Aliyun-0 上，重置权限，并将其添加到 ssh-agent 中，使其能够免密登录到其它子节点。

```
$ eval `ssh-agent`
```

```
$ chmod 600 /home/mpiuser/.ssh/id_rsa

$ ssh-add ~/.ssh/id_rsa
```

```
mpiuser@Aliyun-0:~$ chmod 600 /home/mpiuser/.ssh/id_rsa
mpiuser@Aliyun-0:~$ ssh-add ~/.ssh/id_rsa
Identity added: /home/mpiuser/.ssh/id_rsa (/home/mpiuser/.ssh/id_rsa)
```

尝试一下免密登录其它节点，能够正常登录，说明所有机器的公钥都正常准备了。

```
mpiuser@Aliyun-0:~$ ssh mpiuser@Aliyun-5
The authenticity of host 'aliyun-5 (172.24.12.114)' can't be established.
ECDSA key fingerprint is SHA256:gFGBLtBO4+kJDTgx3gVqVzBNRicFrn9l1FiulSfV1go.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'aliyun-5,172.24.12.114' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-106-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

 * "If you've been waiting for the perfect Kubernetes dev solution for
   macOS, the wait is over. Learn how to install Microk8s on macOS."

   https://www.techrepublic.com/article/how-to-install-microk8s-on-macos/

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.


Welcome to Alibaba Cloud Elastic Compute Service !

To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

mpiuser@Aliyun-5:~$
```

然后需要向主节点安装 NFS-Server，MPI 用这个方式与子节点交换数据。

```
$ sudo apt install nfs-kernel-server
```

```
mpiuser@Aliyun-0:~$ sudo apt install nfs-kernel-server
[sudo] password for mpiuser:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer
  linux-headers-4.15.0-55 linux-headers-4.15.0-55-generic linux-image
  linux-modules-extra-4.15.0-55-generic
Use 'sudo apt autoremove' to remove them.
The following additional packages will be installed:
  keyutils libevent-2.1-6 libnfsidmap2 libtirpc1 nfs-common rpcbind
```

然后创建一个 cloud 文件夹，并将其添加到 exports 中

```
$ mkdir cloud

$ echo '/home/mpiuser/cloud
```

```
*(rw,sync,no_root_squash,no_subtree_check)' |sudo tee -a
/etc/exports
$ sudo exportfs -a
```

```
mpiuser@Aliyun-0:~$ mkdir cloud
mpiuser@Aliyun-0:~$ echo '/home/mpiuser/cloud *(rw,sync,no_root_squash,no_subtree_check)' |sudo tee -a /etc/exports
/home/mpiuser/cloud *(rw,sync,no_root_squash,no_subtree_check)
mpiuser@Aliyun-0:~$ sudo exportfs -a
```

最后重启一下 nfs-kernel-server restart 服务

```
$ sudo systemctl restart nfs-kernel-server
```

在其它 7 个子结点上，安装 nfs-common

```
$ sudo apt-get install -y nfs-common
```

```
mpiuser@Aliyun-7:~$ sudo apt-get install -y nfs-common
[sudo] password for mpiuser:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and ar
  linux-headers-4.15.0-55 linux-headers-4.15.0-55-generic
  linux-modules-extra-4.15.0-55-generic
Use 'sudo apt autoremove' to remove them.
The following additional packages will be installed:
  keyutils libevent-2.1-6 libnfsidmap2 libtirpc1 rpcbind
```

然后将主节点 Aliyun-0 的 cloud 目录挂载到本机。

```
$ mkdir cloud
$ sudo mount -t nfs Aliyun-0:/home/mpiuser/cloud ~/cloud
$ df -h
```

```
mpiuser@Aliyun-7:~$ mkdir cloud
mpiuser@Aliyun-7:~$ sudo mount -t nfs Aliyun-0:/home/mpiuser/cloud ~/cloud
mpiuser@Aliyun-7:~$ df -h
Filesystem                    Size  Used Avail Use% Mounted on
udev                           12G     0   12G   0% /dev
tmpfs                         2.3G  4.1M  2.3G   1% /run
/dev/vda1                      20G  4.0G   15G  22% /
tmpfs                          12G     0   12G   0% /dev/shm
tmpfs                         5.0M     0  5.0M   0% /run/lock
tmpfs                          12G     0   12G   0% /sys/fs/cgroup
tmpfs                         2.3G     0  2.3G   0% /run/user/1000
Aliyun-0:/home/mpiuser/cloud   20G  3.9G   15G  21% /home/mpiuser/cloud
```

看到最后一行信息可知挂载成功。现在子节点就能简单的和主节点交换数据了。

然后在每个节点上安装 mpich（它是 openmpi 的一种版本，可以直接安装，避免了 openmpi 手动编译安装的麻烦）

```
$ sudo apt install -y mpich
```

```
mpiuser@Aliyun-7:~$  sudo apt install -y mpich
[sudo] password for mpiuser:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
  linux-headers-4.15.0-55 linux-headers-4.15.0-55-generic linux-image-4.15.0-55-generic linux-modules-4.15.0-55-generic
  linux-modules-extra-4.15.0-55-generic
Use 'sudo apt autoremove' to remove them.
The following additional packages will be installed:
  gfortran gfortran-7 hwloc-nox libcr-dev libcr0 libgfortran-7-dev libgfortran4 libhwloc-plugins libhwloc5 libmpich-dev
  libmpich12 libpciaccess0 ocl-icd-libopencl1
```

然后创建一个 host_file，用于指定运行的节点。

```
echo 'Aliyun-0

Aliyun-1

Aliyun-2

Aliyun-3

Aliyun-4

Aliyun-5

Aliyun-6

Aliyun-7

' >> host_file
```

编译运行示例程序。

```
$ mpicc -o mpi_hello_world mpi_hello_world.c

$ mpirun -n 8 --hostfile host_file ./mpi_hello_world
```

刚开始遇到的问题：

```
mpiuser@Aliyun-0:~/cloud$ mpirun -n 8 --hostfile host_file ./mpi_hello_world
Host key verification failed.
Host key verification failed.
```

显示 Host key verification failed. 但是显示的密钥出错的数量又小于总节点数。排查后发现，是因为部分从未在主节点上登陆过，所以没有信任这些节点的公钥。只需要在主节点上手动登陆一下每个子节点就好。

解决后成功运行，输出如下

```
mpiuser@Aliyun-0:~/cloud$ mpicc -o mpi_hello_world mpi_hello_world.c
mpiuser@Aliyun-0:~/cloud$ mpirun -n 8 --hostfile host_file ./mpi_hello_world
Hello world from processor Aliyun-1, rank 1 out of 8 processors
Hello world from processor Aliyun-5, rank 5 out of 8 processors
Hello world from processor Aliyun-7, rank 7 out of 8 processors
Hello world from processor Aliyun-2, rank 2 out of 8 processors
Hello world from processor Aliyun-3, rank 3 out of 8 processors
Hello world from processor Aliyun-6, rank 6 out of 8 processors
Hello world from processor Aliyun-4, rank 4 out of 8 processors
Hello world from processor Aliyun-0, rank 0 out of 8 processors
```

```
$ mpicc -fopenmp -mcmodel=large -o group_by_mpi_opm
group_by_mpi_opm.c
$ mpirun -n 4 --hostfile host_file ./group_by_mpi_opm 1 10000000
10M.txt
```

1. 天河一号

（1）天河二号 4 线程情况时各数据集的测试结果

运行截图：

```
[scut_gqchen_1 ~/201736721010 06:41:19] # yhrun -N 1 -p free group_by_mpi_opm 1 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.667971
Send Data Time: 0.000034
*Run Time: 5.436423
Recive Data Time: 0.287106
Label Time: 0.698405
Write Time: 2.220578
*Total: 8.642546

Detailed results have written into file ./result.txt

[scut_gqchen_1 ~/201736721010 06:41:31] # yhrun -N 4 -p free group_by_mpi_opm 4 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.644725
Send Data Time: 0.118899
*Run Time: 0.429434
Recive Data Time: 0.284678
Label Time: 0.657855
Write Time: 2.058509
*Total: 3.549375

Detailed results have written into file ./result.txt
```

运行截图 图 1

```
[scut_gqchen_1 ~/201736721010 06:41:38] # yhrun -N 1 -p free group_by_mpi_opm 1 100000000 100M.txt
^Cyhrun: interrupt (one more within 1 sec to abort)
yhrun: task 0: running
^Cyhrun: interrupt (one more within 1 sec to abort)
yhrun: task 0: running
^Cyhrun: interrupt (one more within 1 sec to abort)
yhrun: task 0: running
^Cyhrun: sending Ctrl-C to job 15217360.0
yhrun: Job step aborted: Waiting up to 2 seconds for job step to finish.
slurmd[cn9883]: *** STEP 15217360.0 KILLED AT 2020-07-15T07:11:35 WITH SIGNAL 9 ***
```

运行截图 图 2

统计结果如下：

不同数据集的测试结果				
数据集	10M		100M	
运行方式	串行	并行	串行	并行
运行时间	8.642546	3.549375	-	-
加速比	-	2.434949	-	-
效率	-	0.608737	-	-
截图编号	图1		图2	

因为天河二号平台运行速度实在太慢，没能测出 100M 的运行时间。

（2）天河二号 单节点 MPI 并行性能测试：

由于天河二号的运行速度过于缓慢，我只使用 10M 的数据集进行了测试。

运行截图如下：

```
[scut_gqchen_1 ~/201736721010 06:21:17] # yhrun -N 1 -p free group_by_mpi_opm 1 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.649917
Send Data Time: 0.000033
*Run Time: 5.405896
Recive Data Time: 0.279190
Label Time: 0.698996
Write Time: 2.065019
*Total: 8.449134

Detailed results have written into file ./result.txt
[scut_gqchen_1 ~/201736721010 06:21:27] # yhrun -N 2 -p free group_by_mpi_opm 1 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.690930
Send Data Time: 0.068484
*Run Time: 2.626106
Recive Data Time: 0.293409
Label Time: 0.482612
Write Time: 1.662147
*Total: 5.132758

Detailed results have written into file ./result.txt
[scut_gqchen_1 ~/201736721010 06:21:34] # yhrun -N 4 -p free group_by_mpi_opm 1 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.643908
Send Data Time: 0.119146
*Run Time: 1.416817
Recive Data Time: 0.269485
Label Time: 0.693311
Write Time: 2.062259
*Total: 4.561018

Detailed results have written into file ./result.txt
```

图 3

```
[scut_gqchen_1 ~/201736721010 06:21:40] # yhrun -N 8 -p free group_by_mpi_opm 1 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.654610
Send Data Time: 0.146368
*Run Time: 0.762814
Recive Data Time: 0.269964
Label Time: 0.504070
Write Time: 1.695424
*Total: 3.378640

Detailed results have written into file ./result.txt
[scut_gqchen_1 ~/201736721010 06:21:45] # yhrun -N 16 -p free group_by_mpi_opm 1 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.685098
Send Data Time: 0.196324
*Run Time: 0.489278
Recive Data Time: 0.293848
Label Time: 0.516804
Write Time: 1.717892
*Total: 3.214146

Detailed results have written into file ./result.txt
```

运行截图 图 4

天河二号 单节点上MPI性能测试					
进程数	**1**	**2**	**4**	**8**	**16**
运行时间(s)	8.449134	5.132758	4.561018	3.37864	3.214146
加速比	-	1.646	1.852	2.501	2.629
效率	-	0.823	0.463	0.313	0.164
截图编号	图3 图4				

天河二号上的运行结果极其不稳定，同一条件下用时忽快忽慢，会受平台资源的和其它程序的干扰极大，难以得出确切结论。因此这里不做过多分析，只展示结果。后面对阿里云上的测试结果进行详细分析。

（3）天河二号 OpenMP 并行性能测试：

运行截图如下

```
[scut_gqchen_1 ~/201736721010 06:24:07] # yhrun -N 1 -p free group_by_mpi_opm 1 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.662866
Send Data Time: 0.000025
*Run Time: 5.034468
Recive Data Time: 0.292897
Label Time: 0.474304
Write Time: 1.668456
*Total: 7.470150

Detailed results have written into file ./result.txt
[scut_gqchen_1 ~/201736721010 06:25:14] # yhrun -N 1 -p free group_by_mpi_opm 2 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.647186
Send Data Time: 0.000027
*Run Time: 2.660770
Recive Data Time: 0.277333
Label Time: 0.485102
Write Time: 1.752374
*Total: 5.175606

Detailed results have written into file ./result.txt
[scut_gqchen_1 ~/201736721010 06:25:21] # yhrun -N 1 -p free group_by_mpi_opm 4 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.676076
Send Data Time: 0.000025
*Run Time: 1.470159
Recive Data Time: 0.317588
Label Time: 0.487699
Write Time: 1.776005
*Total: 4.051476

Detailed results have written into file ./result.txt
```

图 5

```
[scut_gqchen_1 ~/201736721010 06:25:26] # yhrun -N 1 -p free group_by_mpi_opm 8 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.669559
Send Data Time: 0.000026
*Run Time: 0.855918
Recive Data Time: 0.299996
Label Time: 0.486920
Write Time: 1.797617
*Total: 3.440477

Detailed results have written into file ./result.txt
[scut_gqchen_1 ~/201736721010 06:25:32] # yhrun -N 1 -p free group_by_mpi_opm 16 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.660211
Send Data Time: 0.000026
*Run Time: 0.486769
Recive Data Time: 0.328786
Label Time: 0.488478
Write Time: 1.726916
*Total: 3.030975

Detailed results have written into file ./result.txt
```

图 6

统计结果如下：

天河二号 OpenMP并行性能测试					
进程数	**1**	**2**	**4**	**8**	**16**
运行时间(s)	7.47015	5.175606	4.051476	3.440477	3.030975
加速比	-	1.443	1.844	2.171	2.465
效率	-	0.722	0.461	0.271	0.154
截图编号	图5 图6				

天河二号上的运行结果极其不稳定，这里不做过多分析，只展示结果。后面对阿里云上的测试结果进行详细分析。

（4）MPI+OpenMP 性能测试

运行截图如下：

```
[scut_gqchen_1 ~/201736721010 07:13:41] # yhrun -N 1 -p free group_by_mpi_opm 1 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.659561
Send Data Time: 0.000025
*Run Time: 5.105668
Recive Data Time: 0.285040
Label Time: 0.482241
Write Time: 1.671534
*Total: 7.544508

Detailed results have written into file ./result.txt
[scut_gqchen_1 ~/201736721010 07:13:51] # yhrun -N 1 -p free group_by_mpi_opm 2 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.654746
Send Data Time: 0.000032
*Run Time: 2.656007
Recive Data Time: 0.296261
Label Time: 0.642166
Write Time: 2.083684
*Total: 5.678150

Detailed results have written into file ./result.txt
```

图 7

```
[scut_gqchen_1 ~/201736721010 07:13:59] # yhrun -N 1 -p free group_by_mpi_opm 4 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.659911
Send Data Time: 0.000036
*Run Time: 1.489122
Recive Data Time: 0.298763
Label Time: 0.514980
Write Time: 1.784121
*Total: 4.087022

Detailed results have written into file ./result.txt
[scut_gqchen_1 ~/201736721010 07:14:28] # yhrun -N 1 -p free group_by_mpi_opm 8 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.661343
Send Data Time: 0.000026
*Run Time: 0.877397
Recive Data Time: 0.310556
Label Time: 0.487067
Write Time: 1.764787
*Total: 3.439833

Detailed results have written into file ./result.txt
```

图 8

```
[scut_gqchen_1 ~/201736721010 07:17:12] # yhrun -N 2 -p free group_by_mpi_opm 1 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.670920
Send Data Time: 0.070170
*Run Time: 2.642477
Recive Data Time: 0.289758
Label Time: 0.481200
Write Time: 1.667775
*Total: 5.151380

Detailed results have written into file ./result.txt
[scut_gqchen_1 ~/201736721010 07:17:28] # yhrun -N 2 -p free group_by_mpi_opm 2 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.687904
Send Data Time: 0.068687
*Run Time: 1.432902
Recive Data Time: 0.286459
Label Time: 0.495566
Write Time: 1.731232
*Total: 4.014846

Detailed results have written into file ./result.txt
```

图 9

```
[scut_gqchen_1 ~/201736721010 07:18:37] # yhrun -N 2 -p free group_by_mpi_opm 4 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.682072
Send Data Time: 0.069634
*Run Time: 0.752036
Recive Data Time: 0.311100
Label Time: 0.490476
Write Time: 1.749804
*Total: 3.373050

Detailed results have written into file ./result.txt
[scut_gqchen_1 ~/201736721010 07:19:03] # yhrun -N 2 -p free group_by_mpi_opm 8 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.700898
Send Data Time: 0.072331
*Run Time: 0.438340
Recive Data Time: 0.338083
Label Time: 0.499486
Write Time: 1.769770
*Total: 3.118010

Detailed results have written into file ./result.txt
```

图 10

```
[scut_gqchen_1 ~/201736721010 07:21:23] # yhrun -N 4 -p free group_by_mpi_opm 1 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.659893
Send Data Time: 0.118340
*Run Time: 1.309681
Recive Data Time: 0.267973
Label Time: 0.469343
Write Time: 1.635399
*Total: 3.800736

Detailed results have written into file ./result.txt
[scut_gqchen_1 ~/201736721010 07:21:33] # yhrun -N 4 -p free group_by_mpi_opm 2 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.503040
Send Data Time: 0.117071
*Run Time: 0.710897
Recive Data Time: 0.276015
Label Time: 0.469498
Write Time: 1.658778
*Total: 3.232259

Detailed results have written into file ./result.txt
```

图 11

```
[scut_gqchen_1 ~/201736721010 07:23:34] # yhrun -N 4 -p free group_by_mpi_opm 4 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.680504
Send Data Time: 0.118929
*Run Time: 0.415025
Recive Data Time: 0.273865
Label Time: 0.487352
Write Time: 1.714333
*Total: 3.009504

Detailed results have written into file ./result.txt
[scut_gqchen_1 ~/201736721010 07:23:40] # yhrun -N 4 -p free group_by_mpi_opm 8 10000000 10M.txt
Group By Result: 6356524

Read Time: 0.652817
Send Data Time: 0.131923
*Run Time: 0.278838
Recive Data Time: 0.307157
Label Time: 0.490229
Write Time: 1.725911
*Total: 2.934058

Detailed results have written into file ./result.txt
```

图 12

统计数据如下：

表4 MPI+OpenMP性能测试												
进程数	1				2				4			
线程数	1	2	4	8	1	2	4	8	1	2	4	8
运行时间	7.545	5.678	4.087	3.440	5.151	4.015	3.373	3.118	3.801	3.232	3.010	2.934
加速比	1.000	0.753	1.846	2.193	1.465	1.879	2.237	2.420	1.985	2.334	2.507	2.571
效率	1.000	0.376	0.461	0.274	0.732	0.470	0.280	0.151	0.496	0.292	0.157	0.080
截图	图7		图8		图9		图10		图11		图12	

2. 使用阿里云服务器运行

通过前述步骤的操作，我使用 8 台 12 核 24G 的服务器搭建了 MPI 的运算集群。下面是在该集群下的运行结果。

编译：

```
mpicc -fopenmp -mcmodel=large -O2 -o group_by_mpi_omp group_by_mpi_omp.c
```

运行：

```
mpirun -n 1 --hostfile host_file group_by_mpi_opm 1 100000000 100M.txt
mpirun -n 1 --hostfile host_file group_by_mpi_opm 2 100000000 100M.txt
mpirun -n 1 --hostfile host_file group_by_mpi_opm 4 100000000
```

100M.txt

…

mpirun -n 4 --hostfile host_file group_by_mpi_opm 16 100000000 100M.txt

部分运行截图：

```
mpiuser@Aliyun-0:~/cloud$ mpirun -n 1 --hostfile host_file ./group_by_mpi_omp 1 100000000 100M.txt
Group By Result: 45197730

Read Time: 2.885028
Send Data Time: 0.000007
*Run Time: 81.314921
Recive Data Time: 0.337923
Label Time: 4.153908
Write Time: 14.019086
*Total: 99.825845

Detailed results have written into file ./result.txt
mpiuser@Aliyun-0:~/cloud$ mpirun -n 1 --hostfile host_file ./group_by_mpi_omp 2 100000000 100M.txt
Group By Result: 45197730

Read Time: 2.889424
Send Data Time: 0.000006
*Run Time: 41.703847
Recive Data Time: 0.337369
Label Time: 4.156218
Write Time: 14.793962
*Total: 60.991403
```

```
mpiuser@Aliyun-0:~/cloud$ mpirun -n 1 --hostfile host_file ./group_by_mpi_omp 4 100000000 100M.txt
Group By Result: 45197730

Read Time: 2.882027
Send Data Time: 0.000007
*Run Time: 21.676424
Recive Data Time: 0.337043
Label Time: 4.144243
Write Time: 13.828691
*Total: 39.986409

Detailed results have written into file ./result.txt
mpiuser@Aliyun-0:~/cloud$ mpirun -n 1 --hostfile host_file ./group_by_mpi_omp 8 100000000 100M.txt
Group By Result: 45197730

Read Time: 2.888243
Send Data Time: 0.000006
*Run Time: 13.819442
Recive Data Time: 0.337711
Label Time: 4.154403
Write Time: 14.874668
*Total: 33.186231

Detailed results have written into file ./result.txt
mpiuser@Aliyun-0:~/cloud$ mpirun -n 1 --hostfile host_file ./group_by_mpi_omp 16 100000000 100M.txt
Group By Result: 45197730

Read Time: 2.897397
Send Data Time: 0.000008
*Run Time: 12.248173
Recive Data Time: 0.342267
Label Time: 4.338523
Write Time: 14.863460
*Total: 31.792431

Detailed results have written into file ./result.txt
```

```
mpiuser@Aliyun-0:~/cloud$ mpirun -n 2 --hostfile host_file ./group_by_mpi_omp 1 100000000 100M.txt
Group By Result: 45197730

Read Time: 2.879087
Send Data Time: 1.684158
*Run Time: 42.221769
Recive Data Time: 0.531578
Label Time: 4.171665
Write Time: 13.961320
*Total: 62.570490

Detailed results have written into file ./result.txt
mpiuser@Aliyun-0:~/cloud$ mpirun -n 2 --hostfile host_file ./group_by_mpi_omp 2 100000000 100M.txt
Group By Result: 45197730

Read Time: 2.878402
Send Data Time: 1.762377
*Run Time: 22.222852
Recive Data Time: 0.497706
Label Time: 4.178654
Write Time: 14.501803
*Total: 43.163393

Detailed results have written into file ./result.txt
```

```
mpiuser@Aliyun-0:~/cloud$ mpirun -n 2 --hostfile host_file ./group_by_mpi_omp 4 100000000 100M.txt
Group By Result: 45197730

Read Time: 2.879356
Send Data Time: 1.907986
*Run Time: 12.094116
Recive Data Time: 0.480137
Label Time: 4.249056
Write Time: 15.165597
*Total: 33.896893

Detailed results have written into file ./result.txt
mpiuser@Aliyun-0:~/cloud$ mpirun -n 2 --hostfile host_file ./group_by_mpi_omp 8 100000000 100M.txt
Group By Result: 45197730

Read Time: 2.889514
Send Data Time: 1.687854
*Run Time: 7.829722
Recive Data Time: 0.481561
Label Time: 4.275726
Write Time: 14.354742
*Total: 28.629605

Detailed results have written into file ./result.txt
mpiuser@Aliyun-0:~/cloud$ mpirun -n 2 --hostfile host_file ./group_by_mpi_omp 16 100000000 100M.txt
Group By Result: 45197730

Read Time: 2.886628
Send Data Time: 2.121851
*Run Time: 6.613459
Recive Data Time: 0.476085
Label Time: 4.167609
Write Time: 15.061313
*Total: 28.440317

Detailed results have written into file ./result.txt
```

```
mpiuser@Aliyun-0:~/cloud$ mpirun -n 4 --hostfile host_file ./group_by_mpi_omp 1 100000000 100M.txt
Group By Result: 45197730

Read Time: 2.889935
Send Data Time: 3.583466
*Run Time: 22.338070
Recive Data Time: 0.347528
Label Time: 4.248518
Write Time: 14.384703
*Total: 44.902286

Detailed results have written into file ./result.txt
mpiuser@Aliyun-0:~/cloud$ mpirun -n 4 --hostfile host_file ./group_by_mpi_omp 2 100000000 100M.txt
Group By Result: 45197730

Read Time: 2.887078
Send Data Time: 4.246910
*Run Time: 12.021274
Recive Data Time: 0.354665
Label Time: 4.290489
Write Time: 14.683032
*Total: 35.596370

Detailed results have written into file ./result.txt
mpiuser@Aliyun-0:~/cloud$ mpirun -n 4 --hostfile host_file ./group_by_mpi_omp 4 100000000 100M.txt
Group By Result: 45197730

Read Time: 2.879812
Send Data Time: 3.597649
*Run Time: 6.728951
Recive Data Time: 0.345314
Label Time: 4.187494
Write Time: 14.751227
*Total: 29.610635

Detailed results have written into file ./result.txt
```

```
Detailed results have written into file ./result.txt
mpiuser@Aliyun-0:~/cloud$ mpirun -n 4 --hostfile host_file ./group_by_mpi_omp 8 100000000 100M.txt
Group By Result: 45197730

Read Time: 2.885393
Send Data Time: 3.557592
*Run Time: 4.144656
Recive Data Time: 0.351008
Label Time: 4.243136
Write Time: 15.008207
*Total: 27.304600

Detailed results have written into file ./result.txt
mpiuser@Aliyun-0:~/cloud$ mpirun -n 4 --hostfile host_file ./group_by_mpi_omp 16 100000000 100M.txt
Group By Result: 45197730

Read Time: 2.886463
Send Data Time: 3.825218
*Run Time: 4.016561
Recive Data Time: 0.350900
Label Time: 4.238907
Write Time: 14.387682
*Total: 26.819268

Detailed results have written into file ./result.txt
```

实际运行中又发现，写文件的时间居然比其它所有操作还要慢

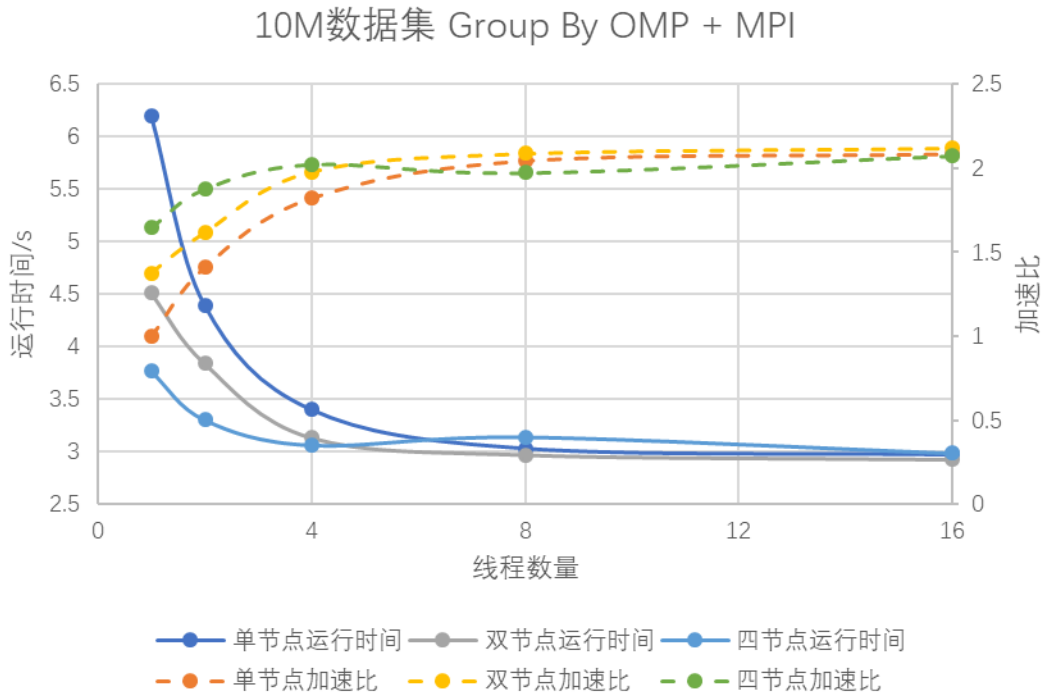
```
mpiuser@Aliyun-0:~/cloud$ sh run.sh
Group By Result: 45197730

Read Time: 3.029251
Send Data Time: 5.403176
*Run Time: 2.913431
Recive Data Time: 0.357061
Write Time: 16.902266
*Total: 25.575933

Detailed results have written into file ./result.txt
```

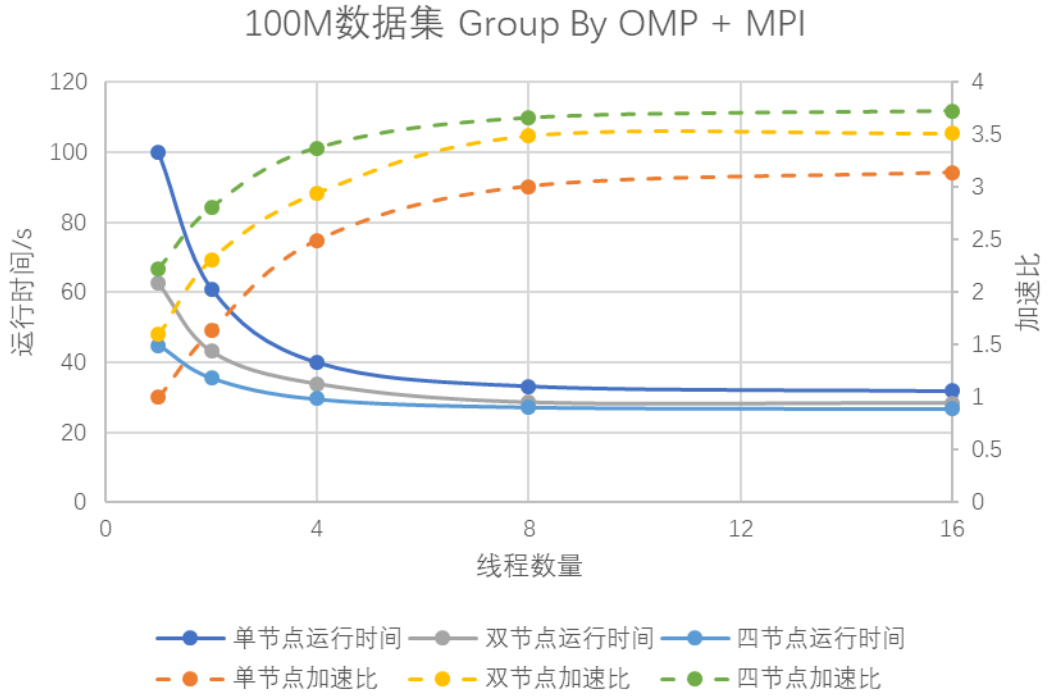
统计结果如下：

10M数据集 Group By OMP + MPI					
线程数	1	2	4	8	16
单节点运行时间	6.200619	4.398182	3.404369	3.034428	2.975052
单节点加速比	1	1.409814	1.821371	2.043423	2.084205
单节点效率	1	0.704907	0.455343	0.255428	0.130263
双节点运行时间	4.510849	3.84158	3.133346	2.969745	2.925219
双节点加速比	1.374601	1.61408	1.978913	2.08793	2.119711
双节点效率	0.687301	0.40352	0.247364	0.130496	0.066241
四节点运行时间	3.767277	3.30567	3.064484	3.142124	2.98915
四节点加速比	1.645915	1.875753	2.023381	1.973385	2.074375
四节点效率	0.411479	0.234469	0.126461	0.061668	0.032412

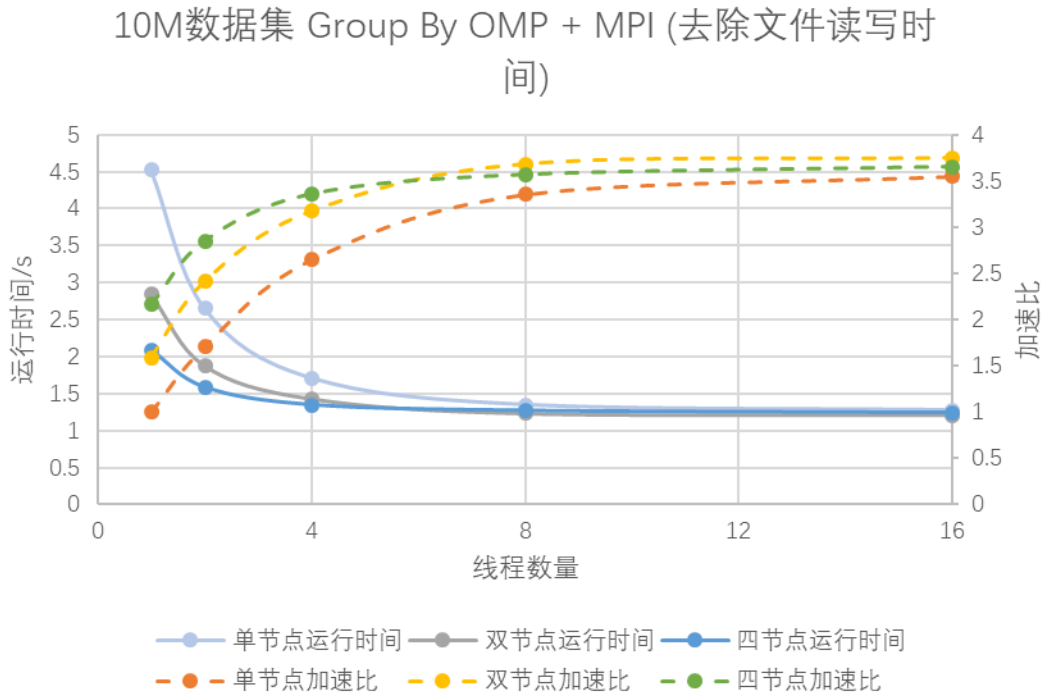
100M数据集 Group By OMP + MPI					
线程数	1	2	4	8	16
单节点运行时间	99.82585	60.9914	39.98641	33.18623	31.79243
单节点加速比	1	1.63672	2.496494	3.00805	3.139925
单节点效率	1	0.81836	0.624124	0.376006	0.196245
双节点运行时间	62.57049	43.16339	33.89689	28.62961	28.44032
双节点加速比	1.595414	2.312743	2.944985	3.486805	3.510012
双节点效率	0.797707	0.578186	0.368123	0.217925	0.109688
四节点运行时间	44.90229	35.59637	29.61064	27.3046	26.81927
四节点加速比	2.22318	2.804383	3.371283	3.656008	3.722169
四节点效率	0.555795	0.350548	0.210705	0.11425	0.058159

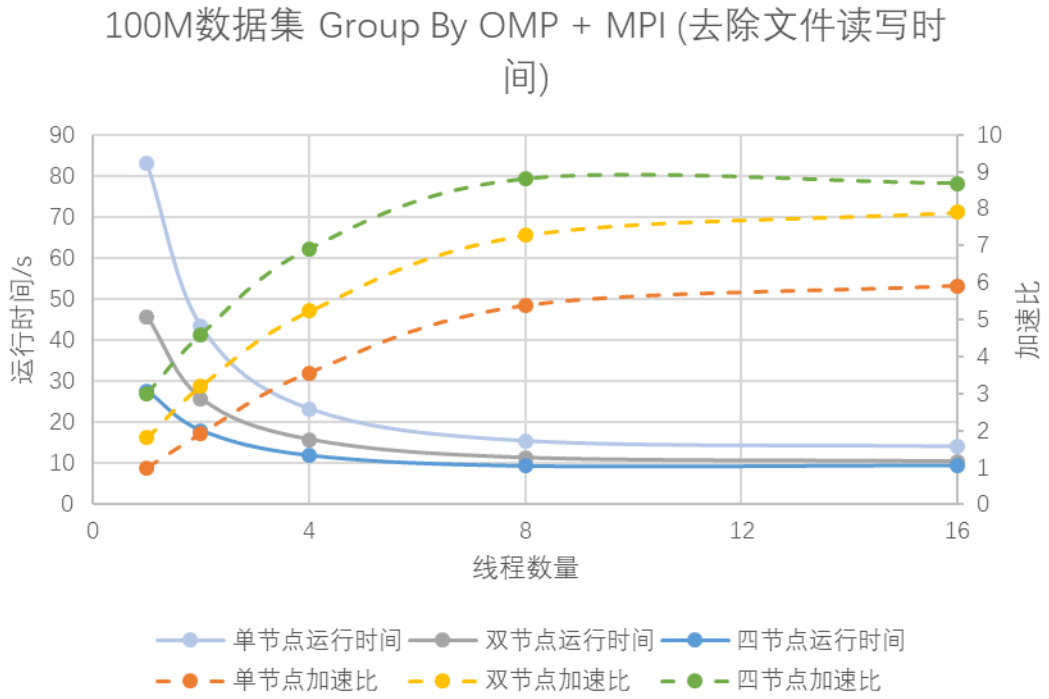
整理得到统计图：





然后我发现有大量的运行时间用于文件读写，这部分是完全无法并行化的，而这些时间会影响解读整个统计图。因此我将 100M 的统计数去除掉文件读写时间，重新绘制了统计图。





分析：

对于两个数据集（10M 条数据、100M 条数据），都是节点数、每个节点内进程数越多，加速比就越大。由于读写文件的部分不可能并行化，所以将其从统计数据中去除，只看剩余部分运行时间，更能直观的看出并行程序的效果。我使用的服务器的物理核心只有 6 个，准确说是 6 核 12 线程。所以可以看到在 8 线程到

16 线程时，加速比几乎不变。100M 的数据集去除文件读写时间后，其加速比与核心数的增长变化非常接近理论值。

如果纵向对比，比较同一线程数下 MPI 的加速比，会发现它的提升并不是太大。主要是因为 MPI 需要传输数据，并且节点越多，传输的数据量也越大。大量时间消耗在了数据的传输上，就显得速度提升没那么明显了。但是如果从上面的截图中，单看计算所需要的时间，不算 MPI 的传输时间的话，速度提升还是非常明显的，几乎接近理论值。

四、课程总结

刚拿到大作业题目的时候我还是一头雾水的，反复观看老师的讲解视频后，思路才逐渐清晰起来。总体的解决方法是先通过前缀进行一轮分组，得到若干个相互独立的分组。这时就能用并行计算的方法对每个小分组单独运算，得到分组结果。

OpenMP 的并行化看似简单，但也暗藏玄机。在 OpenMP 并行化时，如果某个函数中有全局变量，且在并行 for 循环中调用了该函数，那么即使指定这个全局变量为私有变量，也不生效，并且不会报任何错误。解决这个问题后，我又发现运行时间不太稳定，通过观察 htop 中的 CPU 占用，发现总有少量核心运算结束时间较晚，我推测是默认的调度方式导致了任务分配不均匀，将其改为动态调度后问题解决。

在进行 MPI 并行化的时候，我考虑了多种方法来传输数据。第一种是在主节点上将数据按前 32 位的大小划分后，直接将每个节点所要计算的部分传输给该节点。后来发现这样并不可取，因为直接按前 32 位等分的话，一定会出现部分节点分到的数据极多而部分节点极少的这种情况。所以 MPI 并行的时候，也一定需要对其求一个分组的过程，求前缀和，在求前缀和的过程中划分数据。基于这种考虑，我决定直接广播数据。然后在计算完之后，只需要回收下标数组 I 的数据，即可统计结果。

完成了上述的程序设计，我又发现天河二号的实验平台非常不稳定，不仅运行速度慢，磁盘 IO 低，内网带宽不足，还会受到其他同学运行程序的干扰，测试效果非常不理想。于是我使用 8 台 12 核 24G 的阿里云的计算型服务器，按照

网上的文档搭建了 MPI 并行计算的平台。搭建过程中也遇到过一些小问题，但最终都能顺利解决。因为使用的服务器配置较高，磁盘使用的是有更高吞吐量的ESSD，内网带宽也有足足 4Gbps，我在这个环境下非常顺利的对程序运行效果进行了测量。

通过这门课，我终于打开了并行程序设计的大门。我认为这门课带给我的不仅是并行程序的设计方法，更是一种思维方式。它与我以往接触到的串行程序有很大的不同。这种思维方式通过一些巧妙的转换、冗余等方法，让程序能并行的运行在多个核、多个计算节点上。这种思维对我现有的思维方式是一种很重要的补充。当然，我也通过这门课切身体验了并行程序的编码过程，了解了串行程序并行化的一般方法。每次实验课中，我看到原本要运行很长时间的程序，在并行的化后能在短短几秒算出结果，我就能体会到并行程序的重要意义。