实验二

学号 姓名

实验题目:

利用 MPI 进行蒙特卡洛模拟

实验环境(操作系统,编译器,硬件配置等):

操作系统：Ubuntu16.04 （WSL）

编译器：gcc (Ubuntu 5.4.0-6ubuntu1~16.04.9) 5.4.0 20160609

硬件配置：Intel® Core™ i7-8650U CPU @ 1.90GHz

算法设计与分析(写出解题思路和实现步骤)

本次实验有两种思路可行：

1. a) 使用一个规模为 1000000 的数组来存储所有车辆的位置和速度，初始时，所有车辆位置为 0，速度为 0。每次迭代后，通过排序，保证该数组按位置升序排列。

   b) 计算本周期每辆车车速时，从头开始遍历数组，先得到比该车辆位置大的第一辆车，计算两车之间距离，然后按照题目的算法，计算这辆车本周期的速度。

   c) 所有车辆速度计算完毕后，从头开始遍历数组，将每辆车的位置增加当前速度，完成后，进行排序

   d) 在并行情况下，每辆车计算数组的[myid*car_num/size, (myid+1)*car_num/size - 1]的车辆的速度和位置后，在主进程(myid=0)进行排序算法。

   e) 由于排序算法耗时较长，导致该算法在规模较大时，运行时间甚至超过了十分钟，所以该算法不可取。

2. 以空间换时间，不需要进行排序的算法：
   a) 这里使用一个大小为 MAX_VELOCITY * 2000 的桶，代表这条道路，第 i 个位置，存放着位置为 i 的所有车辆的速度构成的链表的表头。另外用一个数组存放每个位置的车辆数目。
   b) 每次计算速度时，从后往前遍历，以节省计算下一个位置的车的时间。同样的，按照算法计算本周期速度即可。
   c) 重新计算车辆的位置是本算法的难点，从后往前对每个位置的车辆处理，使用两个指针 p，q 来进行处理，p，q 初始值均为链表的表头 head：
      i. 如果 p 指向的车辆速度为 0，即没有改变车辆位置，则 q=p，p=p->next，不移动 p，处理下一个车辆
      ii. 记录 p 的下一个节点 p_next，将 p->next 设为 NULL
      iii. 将 p 指向的节点插入到车辆新位置的链表中，如果车辆新位置的链表为空，直接将 road[new_location].head 设置为 p；若非空，则将 p 插入到 road[new_location].head 的下一个位置
      iv. 若 p 本来是原链表的头节点，直接将头节点设置为 p_next，否则，将 q->next 设置为 p_next。
   d) 当并行时，每个进程各有这样一个桶和存放车辆数目的数组 each_num，每个链表处理固定的 car_num/size 辆车，所以还需要另外增加一个大小为 MAX_VELOCITY * 2000 的数组 total_num 存放所有车辆在每个位置的数目，每次迭代完成后，使用 MPI_Reduce，op 设为 MPI_SUM，src 为 each_num，来统计所有进程的车辆在每个位置的数目，存放在 total_num 中，MPI_Reduce 的主进程为 myid=0 的进程，此后，再使用 MPI_Bcast 将

total_num 数组广播到其他进程中。

e) Total_num 数组用于在计算速度时统计下一个位置的车辆的位置，each_num 数组在计算车辆的新位置时更新。

f) 这里在并行时还出现了一个问题，由于此处使用 srand/rand 函数还生成随机数，所以各进程的随机结果大致相同，导致所有进程的车辆结果大致相同，故最终结果相当于车辆数目只有规模/进程数的模拟情况，再每个位置乘以进程数，所以进程数不同时，结果完全不相似。为了避免这种情况，在 srand 后，增加一个迭代次数为 myid*500 的循环，这样以来，每个进程的结果都大致相同了。

```
srand((unsigned int)time(NULL));
for(int i=0;i<myid*500;i++)
    rand();
```

核心代码(写出算法实现的关键部分,如核心的循环等)

每个规模的初始化：

```
        have_car[0] = car_num[i];
        int CarNum = car_num[i]/size;
        road[0].count = CarNum;
        Car *tail;
        for(int j=0;j<CarNum;j++) {
            Car *total_num=(Car*)malloc(sizeof(Car));
            total_num->next = NULL;
            total_num->velocity = 0;
            if(j==0) {
                tail = total_num;
                road[0].head = total_num;
            }
            else {
                tail->next = total_num;
                tail = total_num;
            }
        }
```

核心循环：

```
        for(int j=0;j<cycles[i];j++) {
            calculateNextV(myid);
            moveCars();
            MPI_Reduce(have_car, total_num, FAREST_DISTANCE, MPI_INT, MPI_SUM,
0, MPI_COMM_WORLD);
            MPI_Bcast(total_num, FAREST_DISTANCE, MPI_INT, 0, MPI_COMM_WORLD);
            int count=0,count2=0;
            for(int loc=0;loc<FAREST_DISTANCE;loc++) {
                count+=total_num[loc];
                count2+=have_car[loc];
```

```
            }
        }
```

calculateNextV 函数：

```
    srand((unsigned int)time(NULL));
    for(int i=0;i<myid*500;i++)
        rand();
    int j;
    for(j=FAREST_DISTANCE-1;j>=0;j--)
        if(total_num[j]>0) break;
    for(int i=j;i>=0;i--) {
        if(road[i].count==0) {
            if(total_num[i]>0) j=i;
            continue;
        }
        Car *head = road[i].head;
        Car *p;
        if(i==j) {
            //no cars in the front.
            for(p=head;p!=NULL;p=p->next) {
                if(p->velocity < MAXV) p->velocity++;
                if(rand() <= P * RAND_MAX)  {
                    p->velocity--;
                }
            }
        }
        else {
            int distance = j-i;
            for(p=head;p!=NULL;p=p->next) {
                if(distance<=p->velocity) {
                    p->velocity = (distance-1<0)?0:(distance-1);
                }
                else if(p->velocity < MAXV) p->velocity++;
                if(p->velocity>0 && rand() <= P * RAND_MAX)  {
                    p->velocity--;
                }
            }
        }
        j = i;
    }
```

moveCars 函数：

```
for(int i=FAREST_DISTANCE-1;i>=0;i--) {
        have_car[i] = 0;
        if(road[i].count==0) {
            continue;
        }
        Car *p=road[i].head;
        Car *q=p;
        while(p!=NULL) {
            if(p->velocity==0) {
                //not moving
                q = p;
                p = p->next;
                have_car[i]++;
            }
            else {
                road[i].count--;
                if(p==road[i].head) {
                    road[i].head = p->next;
                }
                else {
                    q->next = p->next;
                }
                //insert p into target list.
                int new_loc = i+p->velocity;
                if(new_loc>=FAREST_DISTANCE) {
                    printf("out of range!\n");
                    exit(1);
                }
                else {
                    Car* next_p = p->next;
                    if(road[new_loc].head!=NULL)
                    {
                        Car* temp=road[new_loc].head->next;
                        road[new_loc].head->next = p;
                        p->next = temp;
                    }
                    else {
                        road[new_loc].head = p;
                        p->next = NULL;
                    }
                    p = next_p;
                    road[new_loc].count++;
                    have_car[new_loc]++;
                }
```

```
            }
        }
    }
```

实验结果:

实验结果按如下格式排列

以下时间单位均为 ms

运行时间

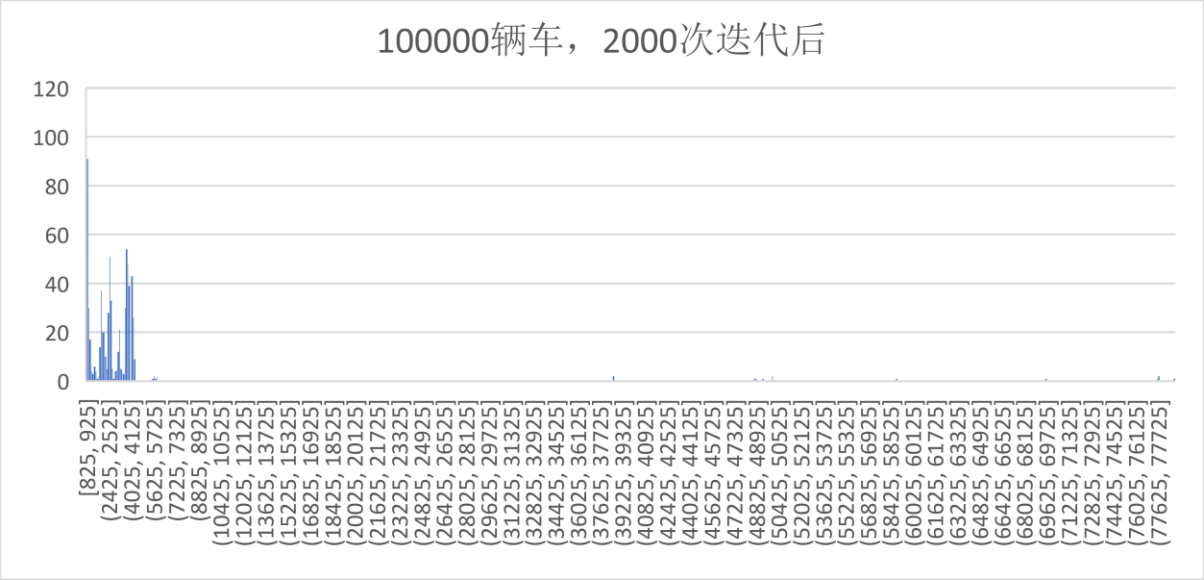
规模\线程数	1	2	4	8
(100000,2000)	8188.001871	7172.800064	7507.314205	9228.831768
(500000,500)	40723.033905	20831.704140	11428.278446	8348.297596
(1000000,300)	58394.065142	31578.602076	17279.705048	10829.584599

加速比

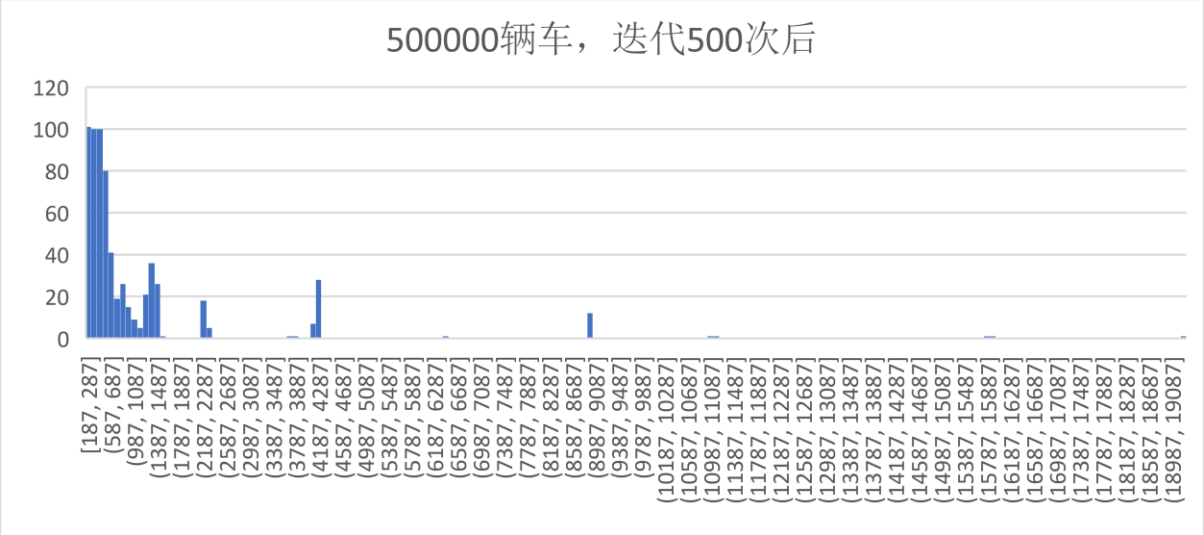
规模\线程数	1	2	4	8
(100000,2000)	1	1.141535	1.09067	0.88722
(500000,500)	1	1.954858	3.563357	4.878005
(1000000,300)	1	1.849166	3.379344	5.392087

分析与总结

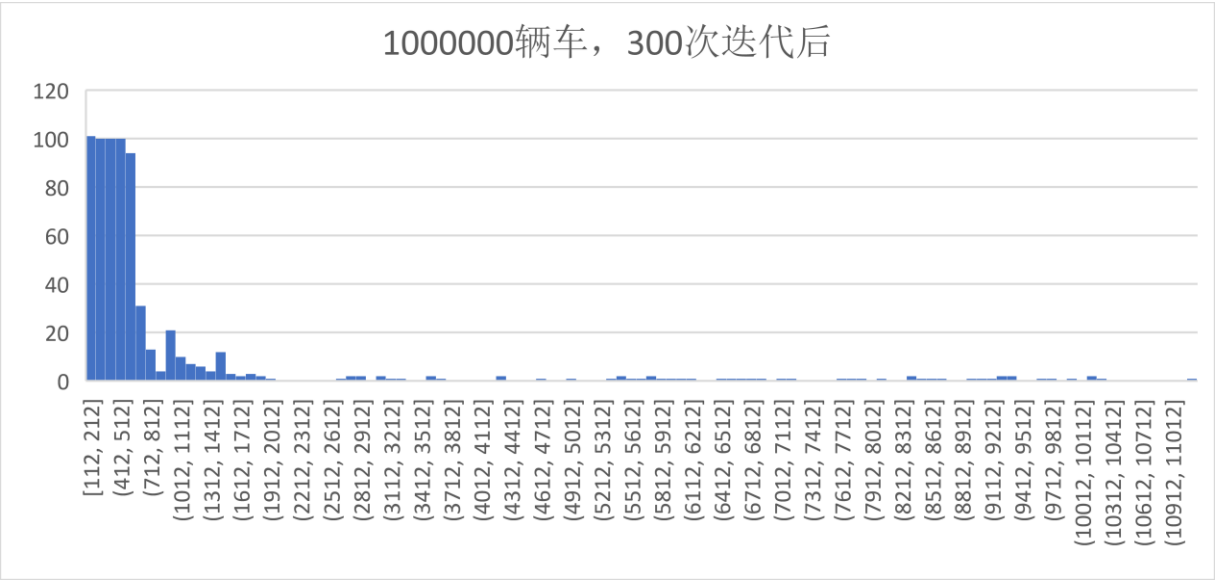
1. 经过测试，使用 1，2，4，8 进程得到的模拟结果总体趋势相同，可见多进程实现是成功的。
2. 加速比分析：而从用时和加速比上来看，理想情况下，1、2、4、8 进程的加速比应该分别为 1、2、4、8，但是由于通信成本较高，导致了实际上达不到那样的加速比。在车辆数目为 100000，迭代周期为 2000 时，8 进程加速比甚至小于 1，这时由于车辆数目较小时，计算开销较小，而每次迭代都需要一次 MPI_Reduce 和 MPI_Bcast 操作，进程数为 8 时这样的开销尤其大，当迭代次数较大时，通信产生的开销已经超过了计算节省的时间，所以导致了加速比小于 1 的情况。
3. 车辆模拟情况分析：
    a) 100000 辆车，迭代 2000 次：



b) 500000 辆车，迭代 500 次：



c) 1000000 辆车，迭代 300 次：



可见，落在最后的车堵得越多，前面的车分布比较稀疏。

备注(可选)
这里已经把所有的模拟结果一并附上
编号为：100000 辆车的结果为 0_{进程数}.txt，500000 辆车的结果为 1_{进程数}.txt，1000000 辆车的结果为 2_{进程数}.txt（这些文件在 Ubuntu 下生成，所以在 Windows 环境下可能不会正确显示换行符）