Lab2-report

- 实验内容

  利用蒙特卡洛算法解决以下问题，并用MPI实现并行化。

  在道路交通规划上，需要对单条道路的拥堵情况进行估计。根据 `Nagel-Schreckenberg` 模型，车辆的运动满足以下规则：

  - 假设当前速度是 v 。
  - 如果前面没车，它在下一秒的速度会提高到 v + 1 ，直到达到规定的最高限速$\mathbf{v_{max}}$。
  - 如果前面有车，距离为d，且 d <= v，那么它在下一秒的速度会降低到 d - 1 。
  - 前三条完成后，司机还会以概率 p 随机减速1个单位，速度不会为负值。
  - 基于以上几点，车辆向前移动v（这里的v已经被更新）个单位。
  - **在本实验中，我取的条件为：$v_{max} = 20$，概率$p = 0.05$，车辆初始速度都为0，初始位置都在道路初始点0。**

- 实验环境

  - 操作系统：Ubuntu 18.04 lts 64bits
  - 编译器
    - mpi 3.2.1 (for mpi)
  - CPU: Inte i5-8250U CPU & 3.40Ghz * 8
  - Memory： 8GB

- 算法设计与分析

  在本次实验中，要求基于 `Nagel-Schreckenberg` 模型对道路车辆的运行情况进行模拟，并且由于规模可能较大，用MPI进行并行化编程实现。在编码过程中，整个实验的算法流程可以分为`Nagel-Schreckenberg` 模型的单次周期模拟、并行化算法MPI实现和算法优化三个部分。

  - **`Nagel-Schreckenberg` 模型的单次周期模拟**

    在实验中，要求在指定时间周期内对指定个数的车辆进行模拟，根据要求的五条规则可知，所有车辆的初始速度和初始位置都为0，从$t = 0$开始的时间一直模拟执行到$t = T$，每个周期内，对每一辆车的情况进行分析从而作出位置和速度的改变。车辆的移动情况是根据其周围环境的车辆位置而决定的（更具体来说是在该辆车前方最近车辆的位置），所以在每个周期内进行模拟时，针对每一辆车，对其周围的所有车辆进行遍历从而找到距离该车前方最近的车的距离，然后根据遍历后的结果，该距离如果为0，说明该辆车在最前方，这样其速度加一，如果找到的距离不为0然后判断距离d和该辆车速度v的大小，如果$d \leq v$，说明车辆速度较快可能发生相撞，所以对v进行限制，降为$d - 1$，同时考虑随机减速的情况（注意速度为0时不能再减速），这样经过以上算法流程车辆的速度进行了更新，然后根据移动速度值的距离即可完成单个周期的模拟。如果是T个周期进行模拟，每一次模拟过程相同。

具体代码实现时，该过程在函数`simulate()`中进行，指定`car_on_road_speed`数组维护更新每一辆车的速度，`car_on_road_pos`数组维护更新每一辆车的位置，两个数组均为全局变量，每个周期进行模拟时，在一个for循环对每一辆车进行搜索前方最近位置，然后针对搜索结果根据规则进行速度和位置的更新。

- **MPI并行实现**

  在用MPI实现并行时，首先要调用MPI函数`MPI_Init(..)`进入MPI环境并完成所有的初始化工作，然后，调用`MPI_Comm_rank`函数获得当前进程在指定通信域中的编号，将自身与其他程序区分。然后要获取指定通信域的进程数，调用`MPI_Comm_size`函数获取指定通信域的进程个数。如果返回的rank为0的话说明是主线程，在我的实现中，主线程任务是负责从命令行接收两个参数参数，分别为车辆数目N和模拟周期时长T，然后通过`MPI_Send`函数将这两个参数发送到其他线程，如果rank不为0，那么该线程就要滴调用`MPI_Recv`函数等待从Id为0的线程接收参数这两个参数。

  各个线程接收到参数之后，根据线程数目$size$进行任务的分配，根据效率最优的原理每一个线程承担的任务量相同均为$n = \frac{N}{size}$， 即每一个线程执行的模拟量为n， 由于在模拟过程中每辆车的更新过程都跟周围车辆相关，所以在此过程中要对线程进行数据的同步，也就是在每一个周期模拟之前需要把线程的情况广播到其他线程，我所采用的是多对多通信方式，调用的MPI接口函数为`MPI_Allgather`，即设定一个全局接收缓冲区，多对多通信接收到的结果保存在这个buff中，然后在本个周期内模拟时就得到了全部的车辆信息，从而进行搜索得到每一辆车的模拟更新。

  需要注意的是，在线程通信时并没有必要把全部的信息进行广播收集通信，因为对于其他线程而言使用到的数据仅为位置，所以速度值没必要进行线程同步，仅仅同步车辆位置这一数据即可，这样就降低了一半的通信代价。

  所有线程同步的模拟完成后rank为0的线程进行汇总，得到程序的运行时间和车辆分布的最终模拟结果。最后通过`MPI_Finalize`函数从MPI环境中退出。

- **算法时间复杂度分析及优化**

  在模拟时，如果按照以上分析的算法流程，每个时间周期内，每一辆车需要遍历所有数目N的车辆位置，一共在T的时间周期数内模拟N辆车，所以其时间复杂度为$O(N^2T\frac{1}{size})$。 在实践中即时并行化运行速度也很慢，1000辆车模拟1000个周期的运行时间在几十秒，如果是百万级别的车辆数那将无法得到结果。所以为了优化，我对车辆数目进行了哈希映射，考虑到实际结果中车辆的集中密度较大，所以一个位置总是会对应多辆车。在周期T的模拟时间内，车辆运行距离的上界为$T * v_{max}$ 这样建的位置数组大小有上界$O(Tv_{max})$，虽然在对哈希表进行查找时并不能做到$O(1)$时间，但是采用二分查找后整个程序的时间复杂度为$O(log(T)NT\frac{1}{size})$，考虑在实验中$N$的规模大于$T$，所以优化后的算法效率得到极大的提高。

核心代码

- `Nagel-Schreckenberg`模型的单次周期模拟

```
void simulate()
{
    int dis, p;
    int *item;
    for(int i = 0;i < car_num;i++)
    {
        // 二分查找位置数组
        item = (int*)bsearch(&car_on_road_pos[i], search_array, k, sizeof(int),
cmpfunc);
        dis = *(item + 1);

        if(dis == 0 && car_on_road_speed[i] < SPEED_MAX)
            car_on_road_speed[i] ++;
        else if(dis > 0 && dis <= car_on_road_speed[i])
            car_on_road_speed[i] = dis - 1;

        p = rand() % 100;
        if(p < SLOW_P * 100 && car_on_road_speed[i] > 0)
            car_on_road_speed[i] --;

        car_on_road_pos[i] += car_on_road_speed[i];
    }
}
```

- 建立**Hash**表进行算法优化

```
    for(int i = 0;i < run_time;i++)
    {
        ....
        k = 0;
        // 统计，建立hash表
        for(int j = 0;j < car_num * size;j++)
            search_buff[all_on_road_pos[j]] ++;
        // 根据hash表映射，建立位置数组用于搜索
        for(int j = 0;j < run_time * SPEED_MAX;j++)
            if(search_buff[j] > 0)
                search_array[k++] = j;
        search_array[k] = 0;
    }
```

- **MPI**执行部分

```
    if(rank == 0)
    {
        // Get arguments from command line.
        if(argc != 3)
        {
            printf("Error arguments!\n");
            return -1;
        }
        car_num = atoi(argv[1]) / size;
        run_time = atoi(argv[2]);
        // Send car number.
        wtime = MPI_Wtime();
        for(int j = 1; j < size; j++)
        {
            MPI_Send(&car_num, 1, MPI_INT, j, 1, MPI_COMM_WORLD);
            MPI_Send(&run_time, 1, MPI_INT, j, 1, MPI_COMM_WORLD);
        }
    }
    else
    {
        MPI_Recv(&car_num, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
        MPI_Recv(&run_time, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
    }
    ......
    for(int i = 0;i < run_time;i++)
    {
        simulate();
        memset(search_buff, 0, run_time * SPEED_MAX *  sizeof(int));
        MPI_Allgather(car_on_road_pos, car_num, MPI_INT, all_on_road_pos, car_num,
MPI_INT, MPI_COMM_WORLD);
        ....
    }
```

- 实验结果

  针对实验要求中的三组参数测试时间结果 如下：

  - 运行时间

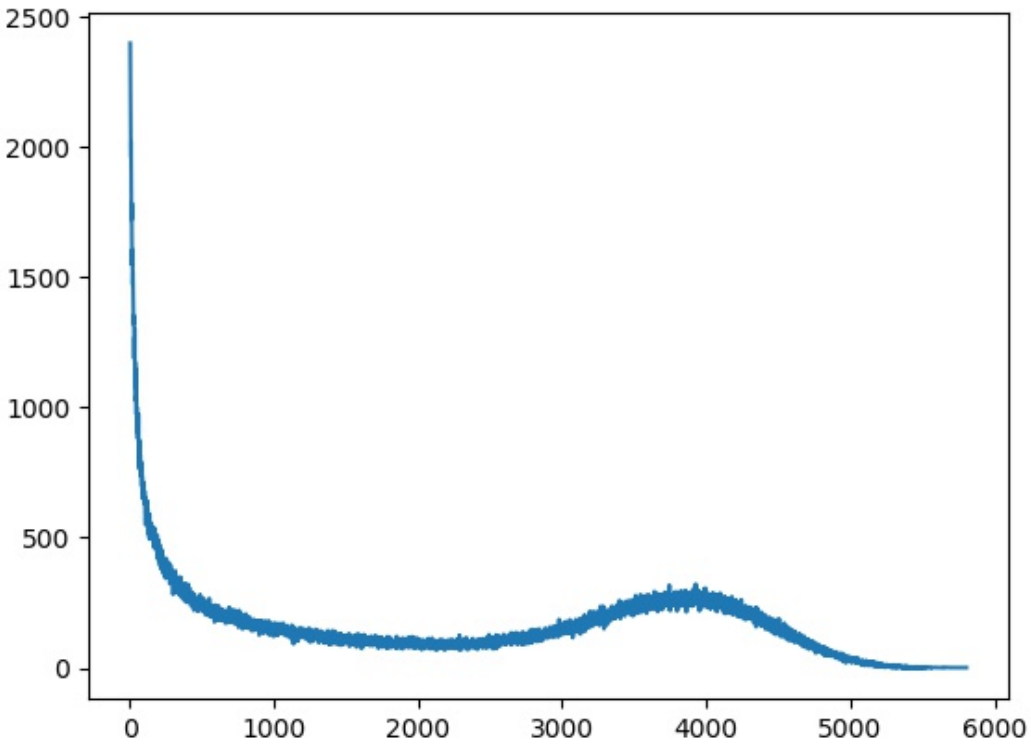
规模\线程数	1	2	4	8
N=10w，T=2000	21.755336s	15.767236s	9.642927s	7.383271s
N=50w，T=500	28.507921s	21.209424s	13.594710s	9.707623s
N=100w，T=300	32.206940s	21.308450s	14.144541s	10.192306s

  - 加速比

规模\线程数	1	2	4	8
N=10w，T=2000	1	1.38	2.26	2.95
N=50w，T=500	1	1.34	2.10	2.94
N=100w，T=300	1	1.51	2.28	3.16

- 车辆分布（以**100w**规模，执行**300**个周期后的分布为例进行分析）

  将车辆位置输出之后绘图可以得到如下的分布曲线。



- 结果分析
  - 运行时间

    从实验结果来看，三个规模下的运行时间大致相近，并且每一个规模下多线程的并行化都起到了明显的加速效果，且线程数越多加速比越大，但是并不是与线程数成线性关系，这是因为随着线程数的增加，通信开销也会变大。

  - 车辆道路分布

    上图的曲线中，x轴指代的是道路位置，y轴指代的是位置坐标的车辆数。由于模型中限制了车辆的加速条件并且设定了随机减速的概率，所以大多数车辆的前进距离较小，在道路坐标较小的位置车辆分布较多，随着道路距离的增加，车辆数量减少，并且由于模拟过程随机性，在某一距离处的车辆数目出现波动的情况。

- 总结和收获

  此次实验，通过MPI模拟了`Nagel-Schreckenberg`模型，并且在线程通信时使用了多对多的通信方式，通过实验可以看到并行化之后可以提高程序效率，并且在实验过程中实现并行时要针对具体的算法实现进行并行优化和搜索优化，从而进一步提高时间性能。