北京邮电大学软件学院

2019-2020 学年第一学期实验报告

课程名称： 并行计算

项目名称： 城市交通模拟的并行实现

项目完成人：

姓名： 平雅霓 **学号：** 2017211949

指导教师： 卢本捷

日　　期： **2019 年 12 月 17 日**

一、 实验目的

1. 学习同步方式实现的并行计算方法
2. 对城市的交通模型进行建模以及进行计算机模拟
3. 对以上的模拟过程给予并行实现

二、 实验内容

1. 以某种随机的策略生成城市的棋盘状道路模型

（1） 停车场

（2） 道路、车道

2. 以某种随机的策略生成汽车静态分布图
3. 以某种特定的或随机的策略来生成道路管理机制，

（1） 红绿灯的时间比例

（2） 道路限速

4. 以某种特定或随机的方式来生成每辆车的目的地。

（1） 行驶路线。

5. 以某种特定或随机的方式来形成汽车的参数

（1） 速度

（2） 加速度

（3） 变道速度

（4） 刹车时间

6. 以某种特定或随机的方式来生成行驶策略：

（1） 与前车的距离以及前车距离对本车的加速度的影响

（2） 其他

7. 以某种特定或随机的方式来生成交通的高峰期与低谷期。

8. 以某种特定或随机的方式来生成交通管制：

（1） 部分路段

（2） 单行线

（3） 部分车辆

三、 实验环境

1．两台或以上的 windows 或 linux 等。

2．节点的网络互联。

3．VC.net 2015 2017 2019 等或其他

4．MPICH 软件环境

四、 实验要求

1. 以单机或集群的方式实现城市交通过程的模拟。

2. 选定任意一种迭代顺序，对城市交通进行迭代。

   (1) 雅可比方式

   (2) 高斯赛德尔方式

3. 绘图

   (1) 使用 C#语言，进行绘图。

      i. createGraphic

      ii. DrawPoint

(2) Qt 或其他的图形库

4. 以并行的方式实现上述算法。

(1) 多处理器对计算区域进行划分

5. 需要进行同步

(1) 以局部的方式进行

五、 待解决的问题：

1. 红绿灯的调整
2. 道路的扩建选址
3. 潮汐交通的调整
4. 道路限速的调整
5. 交通管制策略

六、 运行结果

如图所示有两个路口，道路上有汽车，图形库使用了 easyX。

并用 MPI_SEND 和 MPI_RECV 函数实现雅可比迭代。

七、 MPICH 实验步骤

方格周围的方格的定义

```
enum  GridDirection{GridDirectionLeft,
                    GridDirectionRight,
                    GridDirectionUp,
                    GridDirectionDown,
                    GridDirectionNone };
```

定义了上下左右四个方向的方格，还有无方向即此方格自己。

Grid 类定义了每个方格的属性和函数

```
class Grid {
protected:
    std::vector<bool> * m_cars;
    int m_gridId;
    int m_numberOfUnits;
    int m_numberOfCars;
    int m_forwardNeighborId;
    int m_reverseNeighborId;
    int m_currentTimeStep;
    GridDirection m_direction;

    void seedCars(int);

public:
    Grid () {};
    Grid ( int, int, int, int, int, GridDirection );
    Grid ( int );

    virtual int GetForwardNeighborId();
    virtual int GetReverseNeighborId();

    virtual bool canAcceptNewCar( GridDirection );
    virtual void insertCar( GridDirection );
    virtual bool canReleaseCarAtEndOfTimeStamp();
    virtual bool releaseFrontCar();

  GridDirection GetDirection();

    virtual void finishTimeStep();
    virtual void increaseTimeStep();

    virtual void printCars();
};
```

int m_gridId 是此方格的 id。

int m_numberOfUnits 是此方格中 unit 的数量。

int m_numberOfCars 是此方格中汽车的数量。

int m_forwardNeighborId 是此方格下一个临近方格的 id。

int m_reverseNeighborId 是此方格之前的一个临近方格的 id。

int m_currentTimeStep 是当前的时间。

GridDirection m_direction 是此方格的方向。

下图为类 Grid 的构造函数

```
Grid::Grid( int numberOfUnits, int numberOfCars, int gridId, int forwardId, int reverseId, GridDirection direction ):
        m_numberOfUnits(numberOfUnits),
        m_numberOfCars(numberOfCars),
        m_gridId (gridId),
        m_forwardNeighborId(forwardId),
        m_reverseNeighborId(reverseId),
        m_direction (direction)
{
    m_currentTimeStep = 0;


    this->seedCars(seed);
}
```

初始化了当前的时间，并在构造函数中创建生成汽车。传入的参数为每格中的单元的数量和每格中汽车的数量、放个的 id，前一个放个的 id 和后一个方格的 id、方向。

下图中 SeedCars( )函数用来生成汽车，default_random_engine 是随机数产生器，在 uniform_int_distribution 的构造函数中，参数说明了随机数的范围，此函数中随机数的范围为 0~规定的车的数量之间。

```
void Grid::seedCars( int seed )
{
    std::default_random_engine generator (seed);
    std::uniform_int_distribution<int> distribution(0, m_numberOfUnits -1);
    m_cars = new std::vector<bool>(m_numberOfUnits);

    int numberOfSeededCars = 0;

    while (numberOfSeededCars < m_numberOfCars)
    {
        int insertIndex = distribution(generator);
        if ((*m_cars)[insertIndex] == true)
        {
            continue;
        }
        else
        {
            (*m_cars)[insertIndex] = true;
            numberOfSeededCars++;
        }
    }
}
```

m_gridId(int gridId)函数

```
Grid::Grid ( int gridId ): m_gridId(gridId)
{
    m_numberOfCars = 0;
    m_numberOfUnits = 1;
    m_direction = GridDirectionNone;
    m_currentTimeStep = 0;

    this->seedCars(seed);
}
```

此函数用来获取 grid 的 ID，并初始化这个方格中的单元和汽车数量，汽车数量初始化为 0。单元数量初始化为 1，方向初始化为无方向，当前时间初始化为 0，并且调用 seedCars()函数在此方格内生成汽车。

```
int Grid::GetForwardNeighborId()
{
    return m_forwardNeighborId;
}

int Grid::GetReverseNeighborId()
{
    return m_reverseNeighborId;
}

GridDirection Grid::GetDirection()
{
  return m_direction;
}
```

GetForwardNeighborID( )函数用来获取下一个方格的 id。

GetReverseNeighborID( )函数用来获取上一个方格的 id。

GetDirection( )函数用来获取下一个方格的方向。

canAcceptNewCar()函数用来判断此方格是否可以接受新的汽车。

```
bool Grid::canAcceptNewCar (GridDirection incomingDirection)
{
    return (*m_cars)[m_numberOfUnits - 1] == false;
}
```

InsertCar()函数用来向此方格中插入新的汽车，其中调用了canAcceptNewCar()函数判断是否可以接受新汽车。

```
void Grid::insertCar(GridDirection incomingDirection)
{
    if (!this->canAcceptNewCar(incomingDirection))
    {
        throw RedundantCarInsert();
    }

    (*m_cars)[m_numberOfUnits - 1] = true;
    m_numberOfCars++;
}
```

I

```
bool Grid::canReleaseCarAtEndOfTimeStamp()
{
    return (*m_cars)[0] == true;
}
```

CanReleaseCarEndOfTimeStamp()函数用来判断是否在时间戳最后清理汽车。

```
bool Grid::releaseFrontCar()
{
    if (this->canReleaseCarAtEndOfTimeStamp())
    {
        (*m_cars)[0] = false;
        m_numberOfCars--;
        return true;
    }
    else
    {
        return false;
    }
}
```

releaseFrontCar()函数用来清除前一辆汽车，其中调用了CanReleaseCarEndOfTimeStamp()函数用来判断是否在时间戳最后清理汽车。

下面是 StoplightGrid 类，此类用来设置红绿灯的信息。

```
class StoplightGrid : public Grid {
protected:
    int m_gridId;
private:
    int m_bottomGridId;
    int m_rightGridId;
    int m_topGridId;
    int m_leftGridId;
    bool m_bottomGoingCar;
    bool m_rightGoingCar;

    void setDirection();
```

```
public:
    StoplightGrid(){};
    StoplightGrid ( int, int, int, int, int );

    bool canAcceptNewCar( GridDirection );
    void insertCar ();
    bool releaseFrontCar();
    void increaseTimeStep();
    int GetForwardNeighborId();
    int GetReverseNeighborId();
    int GetOffForwardNeighborId();
    void printCars();
};
```

int m_gridId 是方格的 id。

int m_bottomGridId 是当前方格下方的方格的 id。

int m_rightGridId 是当前方格右边的方格的 id。

int m_topGridId 是当前方格上方的方格的 id。

int m_leftGridId 是当前方格左方的方格的 id。

bool m_bottomGoingCar

bool m_rightGoingCar

下面为此类的构造函数，初始化类中的属性。

```
StoplightGrid::StoplightGrid ( int gridId, int rightGridId, int bottomGridId, int topGridId, int leftGridId ):
     m_rightGridId(rightGridId), m_bottomGridId(bottomGridId), m_leftGridId(leftGridId),m_topGridId(topGridId)
{
    m_gridId = 0;
    m_numberOfCars = 0;
    m_numberOfUnits = 1;
    m_direction = GridDirectionNone;
    m_currentTimeStep = 0;
    Grid::seedCars(m_gridId);
    this->setDirection();
}
```

GetForwardNeighborId()函数用来获得当前方格下一个的方格 id。

```
int StoplightGrid::GetForwardNeighborId()
{
    if (m_direction == GridDirectionRight)
    {
        return m_rightGridId;
    }
    else
    {
        return m_bottomGridId;
    }
}
```

SetDirection()函数用来设置方向。

```
void StoplightGrid::setDirection()
{
    if (m_currentTimeStep % 2 == 0)
    {
        m_direction = GridDirectionDown;
    }
    else
    {
    m_direction = GridDirectionRight;
    }
}
```

下面解释 main 函数的实现

```
int main(int argc, char **argv)
{
  const size_t sz = 10;
  long buffer, result;
  MPI_Status status;
  int comm_sz; // total procs
  int my_rank; // my id

  int NUMBER_OF_UNITS_PER_GRID = atoi(argv[1]);
  int NUMBER_OF_CARS_PER_GRID = atoi(argv[2]);
  int timesteps = atoi(argv[3]);
  int loggingRank = 1;

  MPI_Init(NULL, NULL);
  // get # of procs from communicator
  MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
  // get my id from the communicator
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

在 main 函数的开始定义了一些要用到的变量，MPI_Init( )函数来初始化进程，然后 MPI_Comm_size( )函数用来获取进程数量，并存在 comm_sz 变量中，使用 MPI_Comm_rank( )函数获取当前进程号。此处还声明了一个 MPI_Status 状态。

```
if (my_rank == 0)
{

  StoplightGrid stopLightGrid(0, 3,4, 2, 1);

  int grid1Ready = 0;
  int grid2Ready = 0;
  int grid3Ready = 0;
  int grid4Ready = 0;
  MPI_Request request1;
  MPI_Request request2;
  MPI_Request request3;
  MPI_Request request4;

  //std::cout <<"Stop light waiting for 4 MpiTagBeginTimeStep"<<endl;
    MPI_Irecv(&result, 1, MPI_LONG, 1, MpiTagBeginTimeStep, MPI_COMM_WORLD, &request1);
    MPI_Irecv(&result, 1, MPI_LONG, 2, MpiTagBeginTimeStep, MPI_COMM_WORLD, &request2);
    MPI_Irecv(&result, 1, MPI_LONG, 3, MpiTagBeginTimeStep, MPI_COMM_WORLD,  &request3);
    MPI_Irecv(&result, 1, MPI_LONG, 4, MpiTagBeginTimeStep, MPI_COMM_WORLD,  &request4);

    while (!grid1Ready || !grid2Ready || !grid3Ready || !grid4Ready)
    {
      MPI_Test(&request1, &grid1Ready, NULL);
      MPI_Test(&request2, &grid2Ready, NULL);
      MPI_Test(&request3, &grid3Ready, NULL);
      MPI_Test(&request4, &grid4Ready, NULL);
    }
```

当是主进程的时候创建红绿灯，并且创建请求，接受所有进程的请求，MPI_TEST 用来检测进程的完成状态。第一个参数为 request 非阻塞通信对象(句柄) ，第二个参数为 flag 操作是否完成标志(逻辑型) 。第三个参数为 status 返回的状态 (状态类型)。

下图中的 t1 记录了一个程序运行的开始时间，之后使用 MPI_Send( )函数，其第一个参数是 buf 发送缓冲区的起始地址(可选类型) ，第二个

参数为 count 将发送的数据的个数(非负整数)，第三个参数为 datatype 发送数据的数据类型(句柄)，第四个参数为 dest 目的进程标识号(整型)，第五个参数为 tag 消息标志(整型)，第六个参数为 comm 通信域(句柄)，使用此函数向进程 1，2，3，4 发送缓冲区的数据。这也是实现雅可比迭代的一部分。

```
//start time
auto t1 = std::chrono::high_resolution_clock::now();

for(int j = 0; j < timesteps; j++)
{


  MPI_Send(&buffer, 1, MPI_LONG, 1, MpiTagBeginTimeStep, MPI_COMM_WORLD);
  MPI_Send(&buffer, 1, MPI_LONG, 2, MpiTagBeginTimeStep, MPI_COMM_WORLD);
  MPI_Send(&buffer, 1, MPI_LONG, 3, MpiTagBeginTimeStep, MPI_COMM_WORLD);
  MPI_Send(&buffer, 1, MPI_LONG, 4, MpiTagBeginTimeStep, MPI_COMM_WORLD);

  if (loggingRank == my_rank)
  {
    std::cout << std::endl <<  "***beginning timestep: " << j << " on stoplight***" <<std::endl << std::endl;
    stopLightGrid.printCars();
  }
```

下方的语句开始接收请求。MPI_IRECV 调用返回的时刻，MPI_SEND 的执行必须等到 MPI_IRECV 返回后才可以开始。

```
********** Recieve Open spot request ***************
  int recievedRequestFromLeft = 0;
  int recievedRequestFromTop = 0;

  MPI_Request requestTop;
  MPI_Request requestLeft;


  MPI_Irecv(&result, 1, MPI_LONG, 2, MpiTagRequestOpenSpot, MPI_COMM_WORLD, &requestTop);
  MPI_Irecv(&result, 1, MPI_LONG, 1, MpiTagRequestOpenSpot, MPI_COMM_WORLD, &requestLeft);
```

这一部分代码表示，当此当前方格可以接收新的汽车时，判断接收从上方来的车辆还是左边来的车辆。

```
long canTakeTopCar = 0;
long canTakeLeftCar = 0;
while (!recievedRequestFromLeft || !recievedRequestFromTop)
{
  if (!recievedRequestFromTop)
  {
    MPI_Test(&requestTop, &recievedRequestFromTop, &status);
    if (recievedRequestFromTop)
    {
      if (stopLightGrid.canAcceptNewCar(GridDirection::GridDirectionDown))
      {
        canTakeTopCar = 1;
      }
      if (loggingRank == my_rank)
        std::cout << "Recieved request from top for space sending: " << canTakeTopCar << std::endl;
      MPI_Send(&canTakeTopCar, 1, MPI_LONG, 2, MpiTagHasOpenSpot, MPI_COMM_WORLD);
    }
  }
  if (!recievedRequestFromLeft)
  {
    MPI_Test(&requestLeft, &recievedRequestFromLeft, &status);
    if (recievedRequestFromLeft)
    {
      if (stopLightGrid.canAcceptNewCar(GridDirection::GridDirectionRight))
      {
        canTakeLeftCar = 1;
      }
      if (loggingRank == my_rank)
        std::cout << "Recieved request from Left for space sending: " << canTakeLeftCar << std::endl;
      MPI_Send(&canTakeLeftCar, 1, MPI_LONG, 1, MpiTagHasOpenSpot, MPI_COMM_WORLD);
    }
  }
}
```

下方表示接收新的汽车并将它插入到红绿灯处的方格里。

```
while (!recievedRequestFromLeft || !recievedRequestFromTop)
{
  if (!recievedRequestFromTop)
  {
    MPI_Test(&request1, &recievedRequestFromTop, NULL);
    if (recievedRequestFromTop)
    {
      if (loggingRank == my_rank)
        std::cout << "Recieved " << numberOfCarsFromTop << " cars from top for space" << std::endl;

      if (numberOfCarsFromTop == 1)
      {
        stopLightGrid.insertCar();
      }
    }
  }
```

下方的代码表示当方格 1，2，3，4 进入下一个时间戳的条件。

```
while (!grid1Ready || !grid2Ready || !grid3Ready || !grid4Ready)
{
  if (!grid1Ready)
  {
    MPI_Test(&requestGrid1, &grid1Ready, &status);
    if (grid1Ready)
    {
      if (loggingRank == my_rank)
        std::cout << "Grid 1 Ready for next timestep" << std::endl;
    }
  }

  if (!grid2Ready)
  {
    MPI_Test(&requestGrid2, &grid2Ready, &status);
    if (grid2Ready)
    {
      if (loggingRank == my_rank)
        std::cout << "Grid 2 Ready for next timestep" << std::endl;
    }
  }

  if (!grid3Ready)
  {
    MPI_Test(&requestGrid3, &grid3Ready, &status);
    if (grid3Ready)
    {
      if (loggingRank == my_rank)
        std::cout << "Grid 3 Ready for next timestep" << std::endl;
    }
  }

  if (!grid4Ready)
  {
    MPI_Test(&requestGrid4, &grid4Ready, &status);
    if (grid4Ready)
    {
      if (loggingRank == my_rank)
        std::cout << "Grid 4 Ready for next timestep" << std::endl;
    }
  }
}
```

下方的代码用来判断接收汽车数量的请求，使用 MPI_Test()函数检测进程是否完成，如果 numCarsRecieved 为 1，则向当前的方格中添加汽车。

```
while (!recievedOpenSpotRequest || !recievedNumCarsRequest)
{
  //std::cout << recievedOpenSpotRequest<<recievedNumCarsRequest<< std::endl;
  if (!recievedNumCarsRequest)
  {
    MPI_Test(&request2, &recievedNumCarsRequest, &status);
    if (recievedNumCarsRequest)
    {
      if (loggingRank == my_rank)
        std::cout << "Recieved " << numCarsRecieved << " cars" << std::endl;
      if (numCarsRecieved == 1)
      {
        grid.insertCar(GridDirection::GridDirectionNone);
      }
      break;
    }
  }
  if (!recievedOpenSpotRequest)
  {
    MPI_Test(&request1, &recievedOpenSpotRequest, &status);
    if (recievedOpenSpotRequest)
    {
      if (loggingRank == my_rank)
        std::cout << "Recieved open spot request - sending "<< numOfCarsToSend << " cars" << std::endl;
      MPI_Send(&numOfCarsToSend, 1, MPI_LONG, reverseId, MpiTagHasOpenSpot, MPI_COMM_WORLD);
    }
  }
}
```

八、 调试心得

通过本次实验，我了解了雅可比迭代，学习了同步方式实现并行计算方法，对城市的交通模型进行建模以及进行计算机模拟，并使用 easyX 图形库对以上的模拟过程给予并行实现，是一次十分有趣的尝试，使我对并行计算的思想的了解更加深刻。