Demo: Game Of Life

2019 年 3 月 Version 1.0

1 介绍

1.1 简介

本教程将介绍一系列示例演示构建 Agent Based Modeling of HPC 模型的几种方法。 其目的是展示平台工具包中包含的一系列功能，并提供了如何构建以不同方式使用这些功能的模型的工作示例 demo。 讨论一些功能与平台基础架构有关的内容，例如：如何设置和更改模型参数，获取和记录模拟输出，实现不同形式的随机数生成等。特别是如何便于构建一个 ABM 模型，在此模型中，agent 根据某些结构约束（包括空间布置和网络连接）进行交互。

1.2 要求

本教程适用于任何想要了解平台详细信息并熟练掌握 ABM 模型的人员。对于对 HPC 环境不太熟悉的人来说，主要适用于了解“基于 Agent 建模（ABM）”基础知识、并且熟悉 X10 语言语法的人员。

2 Demo: Game Of Life

2.1 简介

Game Of Life 是生命游戏，元胞自动机（Cellular Automaton/Automata）中最著名的一组规则（该规则的想法可以追溯到冯·诺依曼，别名“生命游戏”）。每个细胞死或活的状态由它周围的八个细胞所决定。

1、“人口过少”：任何活细胞如果活邻居少于 2 个，则死掉。

2、“正常”：任何活细胞如果活邻居为 2 个或 3 个，则继续活。

3、“人口过多”：任何活细胞如果活邻居大于 3 个，则死掉。

4、“繁殖”：任何死细胞如果活邻居正好是 3 个，则活过来。

2.2 构建

1 Step 1 Agent

在 Agent.x10 文件中加入实体的所有属性，已经存在的是位置和状态信息，在 AddCode 后加入。

```
import x10.regionarray.Array;
import x10.util.Random;

public class Agent {
    private var location:Point(2);//Agent位置信息
    private var state:Int;//Agent状态信息

    //AddCode
    private var neighbors:Array[Int];//生命游戏——邻居

    public def this()
    {
        location=[((new Random().nextDouble()*Grid.Grid_XMax) as Int)+1, ((new Ran
        state=0 as Int; //生命游戏——Agent存活 0存活 1死亡
        neighbors=new Array[Int](8,0 as Int);//生命游戏——Agent邻居
    }
    public def this(po:Point(2),s:Int)
    {
        location=po;
        state=s;
        neighbors=new Array[Int](8,0 as Int);
    }
    public def getLocation():Point(2)
    {
        return location;
    }
```

在本例中，按照模型需要，加入了位置获取、状态获取、邻居获取等。

2 Step 2 Grid

在 Grid.x10 文件中进行网格设置，在本例中，网格初始设置为横向和纵向均为 10 格，共计 100 格。

```
public class Grid {

    public static val Grid_XMax:Int=10 as Int;//网格横向
    public static val Grid_YMax:Int=10 as Int;//网格纵向

}
```

在本例中，按照模型需要，加入了位置获取、状态获取、邻居获取等。

3 Step 3 Event

在 EventSequence.x10 文件中进行仿真事件设计，本例中设置了四个事件，分别是生命游戏初始化（init()）、获取 agent 的邻居信息（AgentNeighborState()）、生命游戏按规则演进（Interactive()）、agent 信息显示（Display()）四个事件，同时按照调度步骤写入对应步中。

```
public def step1():void
{
    init();
    AgentNeighborState();
}
public def step2():void
{
    Interactive();
    AgentNeighborState();
}
public def step3():void
{
    Display();
}
```

4 Step 4 Model

在 Model.x10 文件中进行调度设计，在指定的时间运行指定的事件。

```
    public var LiveCount:Long;
    public val AgentList:DistArray[Agent](2);
    public val Reg:Region(2);
    private val runner:ScheduleRunner;

    public def this()
    {
        LiveCount=0 as Long;
        Reg = Region.make((0..(Grid.Grid_XMax-1)),(0..(Grid.Grid_YMax-1)));
        val D=Dist.makeBlockBlock(Reg);
        AgentList=DistArray.make[Agent](D,(p:Point(2))=>null);
        runner = new ScheduleRunner();
    }

    public def initSchedule():void
    {

        val eventsum=new GameOfLifeEventSequence();
        runner.scheduleStopEvent(20,new ScheduleStopFunctor(runner));
        runner.scheduleEvent(0, new ScheduleMethodFunctor(eventsum,1n));
        runner.scheduleEvent(1, 5, new ScheduleMethodFunctor(eventsum,2n));
        runner.scheduleEvent(2, 5, new ScheduleMethodFunctor(eventsum,3n));

    }

    public def getRunner(): ScheduleRunner{
        return runner;
    }
```

5 Step 5 DataCollection

数据收集可以发生在任意指定事件中，也可在任意时刻进行，但此时需要自定义数据收集事件。

1、类实例化

```
public class GameOfLifeEventSequence {
    public var LiveCount:Long;
    public var LiveCountBorn:Long;
    public var LiveCountDead:Long;
    public val AgentList:DistArray[Agent](2);
    public val Reg:Region(2);
    public val DC:DataCollection;
```

2、类实现

```
DC.RecordWrite("  "+"The number of changed Agent (after interactive) is "+ LiveCount +" at Plac
```

6 Step 6 Run

在 Run.x10 文件中的 main 函数中进行并统计时间,在 Settin.x10 文件中设置相关参数。

```
public static def main(Rail[String]):void
{
    var Step:Int=0 as Int;
    val Run=new GameOfLifeModel();
    var Uptime:Long = -System.nanoTime();

    Run.initSchedule();//调度初始化
    Run.getRunner().run();

    Uptime += System.nanoTime();
    Uptime /= 1000000;//将时间由纳秒转化为毫秒
    Console.OUT.println("\n\r"+"Game of life is over.\n"+"  "+"Total time consuming is "+"Tot
}
```