D-Mason a tutorial

http://www.isislab.it/projects/dmason

July 25, 2012

1 Introduction

We present a framework, D-Mason, that is a distributed version of Mason, a well-known and popular library for writing and running Agent-based simulations. D-Mason introduces the parallelization at framework level so that scientists that use the framework (domain expert but with limited knowledge of distributed programming) can be only minimally aware of such distribution.

In this document, in particular, we provide a step-by-step guide to the process of "parallelization" of the Particle example from Mason by using D-Mason.

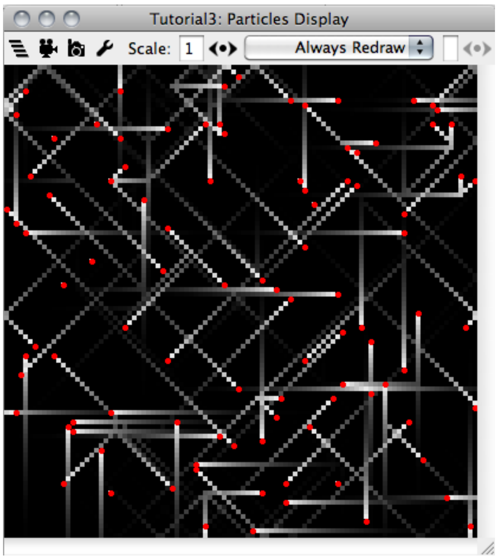

Figure 1: GUIState for `Particle` simulation

2 Acknowledgements

D-Mason contains work by (in alphabetical order): *Michele Carillo, Gennaro Cordasco, Rosario De Chiara, Fabio Fulgido, Ada Mancuso, Dario Mazzeo, Francesco Raia, Vittorio Scarano, Flavio Serrapica, Carmine Spagnuolo, Luca Vicidomini, Mario Vitale.*

3 Structure

The starting point is the package `Particles` is composed by three classes, as it can be found in the original MASON distribution:

- `Particle`: it implements the agent that will be simulated by the application.

- `Particles`: it represents the simulation environment: it allows to run the simulation from the command line without using a GUI.
- `ParticlesWithUI`: it allows to run simulations with a GUI, as depicted in the Figure 1.

Similarly, in D-MASON, there will be the package `DParticles` containing the following classes:

- `RemoteParticle`: it is an abstract class, implementing `RemoteAgent` and containing the remote ID of the agent in the field and its position.
- `DParticle`: it extends `RemoteParticle` and implements the `distributed` agent that will be simulated by the application.
- `DParticles`: it represents the `distributed` simulation environment: it allows to run the simulation from the command line without using a GUI.
- `DParticlesWithUI`: it allows to run simulations with a GUI that is aware of the `distributed` environment.

4 From `Particle` to `DParticle`

The original `Particle` has to implement the `Steppable` interface and, in particular, the method `step()`, containing the agent logic. In the same way `RemoteParticle` is an abstract class that has to implement `RemoteAgent`, that is the D-MASON interface containing the necessary logic for the distributed agent. Finally `DParticle` extends `RemoteParticle` and implements the logic of the agent.

`RemoteAgent` is parameterized with an `Int2D` object-type because, in this simulation, the field has this specific type to indicate locations, and allows programmers to set, for each agent, an unique identifier and a field position. A `Particle` simply contains two integer parameters, `xdir` and `ydir`, for setting the initial direction that the particle will move along.

Listing 1: Class Particle

```
...
  public Particle(int xdir, int ydir) {
   public boolean randomize = false;
   this.xdir = xdir;
   this.ydir = ydir;
  }
...
```

`DParticle` has two constructors: the first is empty and it has been introduced for a future implementation of the method `clone()`, and the second one has as parameter a subclass of the abstract class `DistributedState`.

Listing 2: Class DParticle

```
public class DParticle extends RemoteParticle<Int2D>
{
   public int xdir;  // -1, 0, or 1
   public int ydir;  // -1, 0, or 1

   public DParticle(){ }

   public DParticle(DistributedState state)
   {
     super(state);
   }
```

In order to distribute a MASON simulation it is necessary to change some parts of the agent logic. In the original MASON version each particle, on each step, performs a collision avoidance routine by checking whether the location it is moving to is already occupied by another particle or not.

Listing 3: Class Particle

```
  ...
 public void step(SimState state) {
  ...
  if (randomize) {
   xdir = tut.random.nextInt(3) - 1;
   ydir = tut.random.nextInt(3) - 1;
   randomize = false;
  }
  ...
  // set my new location
  Int2D newloc = new Int2D(newx,newy);
  tut.particles.setObjectLocation(this,newloc);

  // randomize everyone at that location if need be
  Bag p = tut.particles.getObjectsAtLocation(newloc);
  if (p.numObjs > 1) {
  for(int x=0;x<p.numObjs;x++)
      ((Particle)(p.objs[x])).randomize = true;
  }
 }
}
```

The distributed version is slightly different because it first check if the new location is occupied and, in this case, it randomizes its direction and move to the new location by using the method `setDistributedObjectLocation`.

Listing 4: Class DParticle

```
public void step(SimState state)
{
 DParticles tut = (DParticles)state;
 Int2D location = tut.particles.getObjectLocation(this);
 Bag p = tut.particles.getObjectsAtLocation(location);
 tut.trails.setDistributedObjectLocation(1.0, location,state);
 if (p.numObjs > 1)
 {
   xdir = tut.random.nextInt(3) - 1;
   ydir = tut.random.nextInt(3) - 1;
 }
 int newx = location.x + xdir;
 int newy = location.y + ydir;
 if (newx < 0) { newx++; xdir = -xdir; }
 else if (newx >= tut.trails.getWidth()) {newx--; xdir = -xdir; }
 if (newy < 0) { newy++ ; ydir = -ydir; }
 else if (newy >= tut.trails.getHeight()) {newy--; ydir = -ydir; }
 Int2D newloc = new Int2D(newx,newy);
 tut.particles.setDistributedObjectLocation(newloc, this, state);
 }
}
```

5 From `Particles` to `DParticles`

`Particles` extends the `SimState` class while `DParticles` extends `DistributedState`, parameterized with `Int2D` object-type.

DParticles contains three other variables indicating, respectively, width and height of the field and the way of partitioning the field (that can be one or two dimensional, as shown in Figure 2). `Particles` has just one constructor that has as parameter the random generator seed while `DParticles` constructor has as input an objects array, containing several parameters specific for the distributed simulation (e.g. network address, port, etc . . .).

In `Particles` there are two fields, the first containing the agents, the second one containing the trails. The creation of the fields and the placement of the agents in them are carried out by a simple loop that instantiates new particles with a random position and direction and place them in the proper field.

In order to add particles to the schedule, it is possible to use `scheduleRepeating()`, that allows to schedule agents repeatedly, and to add particles to the field there is `setObjectLocation()`. In `DParticles` there is the method `createDSparseGrid2D` of the class `DSparseGrid2DFactory` for creating a new distributed field. Note that it is necessary to use a factory to choose the kind of field partition. The agent initial position is computed by the method `setAvailableRandomLocation()` and to add particles in the schedule it is necessary to use the method `scheduleOnce()`, because in the next step a certain agent could not stay in the same part of the field, so using `scheduleRepeating()` will not delete the particle from the schedule. Finally there are other three new methods: a `getter` method for returning the subclass of the `DistributedState`, a method for adding an agent with a given position in the field, a method for attaching a portrayal to an agent.

Listing 5: Class Particles

```
public class Particles extends SimState  {
  public DoubleGrid2D trails;
  public SparseGrid2D particles;
  ...
  public Particles(long seed) {
      super(seed);
  }

  public void start() {
   ...
   for(int i=0 ; i<numParticles ; i++) {
     p = new Particle(random.nextInt(3) - 1, random.nextInt(3) - 1);  // random
         direction
     schedule.scheduleRepeating(p);
     ...
     particles.setObjectLocation(p,new Int2D(x,y));  // random location
   }
  }

  public static void main(String[] args) {
    doLoop(Particles.class, args);
    System.exit(0);
  }
}
```

Listing 6: Class DParticles

```
public class DParticles extends DistributedState<Int2D> {
public static boolean isToroidal=false;
public DSparseGrid2D particles;
public DDoubleGrid2D trails;
public int gridWidth ;
public int gridHeight ;
public int MODE;

public DParticles(Object[] params)
{
 super((Integer)params[2],(Integer)params[3],(Integer)params[4],
 (Integer)params[7], (Integer)params[8], (String)params[0],
 (String)params[1],(Integer)params[9], isToroidal,
 new DistributedMultiSchedule<Int2D>());
 ip = params[0]+  ;
 port = params[1]+  ;
 this.MODE=(Integer)params[9];
 gridWidth=(Integer)params[5];
 gridHeight=(Integer)params[6];
}

public void start()
{
 super.start();
 try
{
```

```
 trails = DDoubleGrid2DFactory.createDDoubleGrid2D(gridWidth,
  gridHeight, this, super.MAX_DISTANCE, TYPE.pos_i,
  TYPE.pos_j, super.NUMPEERS, MODE,0, false,          );

 particles = DSparseGrid2DFactory.createDSparseGrid2d(gridWidth,
  gridHeight, this,
  super.MAX_DISTANCE, TYPE.pos_i,
  TYPE.pos_j,super.NUMPEERS,MODE,             );

 init_connection();
}catch (DMasonException e) { e.printStackTrace();}
 DParticle p=new DParticle(this);
 while(particles.size() != super.NUMAGENTS)
{
 particles.setAvailableRandomLocation(p);
 p.xdir = random.nextInt(3)-1;
 p.ydir = random.nextInt(3)-1;
 if(particles.setDistributedObjectLocationForPeer(new Int2D
   (p.pos.getX(),p.pos.getY()), p, this))
 {
 schedule.scheduleOnce(schedule.getTime()+1.0,p);
 if(particles.size() != super.NUMAGENTS)
 p=new DParticle(this);
 }
}
 Steppable decreaser = new Steppable()
{
 public void step(SimState state)
{
 trails.multiply(0.9);
}
 static final long serialVersionUID = 6330208160095250478L;
};
 schedule.scheduleRepeating(Schedule.EPOCH,2,decreaser,1);
try
{
 getTrigger().publishToTriggerTopic(                  +particles.cellType+
         );
 } catch (Exception e) {
 e.printStackTrace();
 }
}

public static void main(String[] args)
{
 doLoop(DParticles.class, args);
 System.exit(0);
}
static final long serialVersionUID = 9115981605874680023L;

public DistributedField getField()
{
 return particles;
}

public SimState getState()
{
 return this;
}

public void addToField(RemoteAgent<Int2D> rm,Int2D loc)
{
 particles.setObjectLocation(rm, loc);
}

public boolean setPortrayalForObject(Object o)
 {
 return false;
 }
}
```

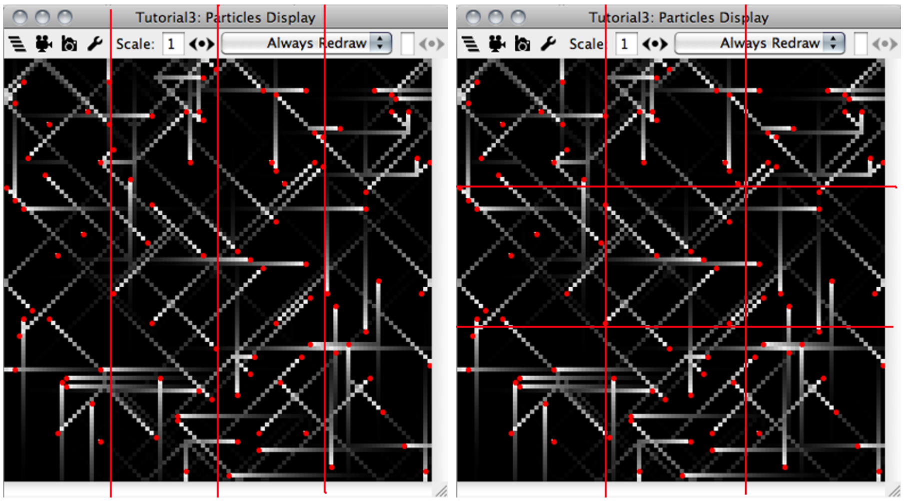

Figure 2: Respectively `HORIZONTAL` and `SQUARE` partition mode.

6 From `ParticlesWithUI` to `DParticlesWithUI`

There are few differences between original `ParticlesWithUI` and the its distributed version, `DParticlesWithUI`. They both extend the class `GUIState`, responsible of instantiating all graphics elements; `DParticlesWithUI` has a constructor for passing to `DParticles` the objects array and it has to store in a `String` the `region` identifier, in order to show which region it is simulating (e.g. `0-0` means the upper-left part of the grid partitioned field).

Listing 7: Class ParticlesWithUI

```
public class ParticlesWithUI extends GUIState {
...
public static void main(String[] args) {
ParticlesWithUI t = new ParticlesWithUI();
t.createController();
}
public ParticlesWithUI() {
super(new Tutorial3(System.currentTimeMillis()));
}
public ParticlesWithUI(SimState state){
super(state);
}
...
}
```

Listing 8: Class `DParticlesWithUI`

```
public class DParticlesWithUI extends GUIState {
...
public static String name;
...
public DParticlesWithUI(Object[] args) {
super(new DParticles(args));
name = String.valueOf(
args[7])+  +(String.valueOf(args[8]));
}
public static String getName() { return          +name+   ; }
...
}
```

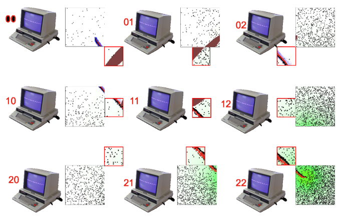

Figure 3: Example of load balancing for grid partition mode.

7 Load balancing for Grid partition

8 Introduction

As described before D-MASON uses a space partitioning approach where the fields are subdivided in regions assigned to workers; this approach allows to limit the communication among the workers. Indeed, since each agent interacts only within a small area around it, the communication is limited to local messages (messages between workers, managing neighboring spaces, etc.).

The problem with this approach is that agents can migrate between regions and consequently the association between workers and agents changes during the simulation. Moreover, load balancing is not guaranteed and needs to be addressed by the application.

8.1 The balancing mechanism

The balancing mechanism is quite simple: when a region is overloaded it decides to split itself in smaller regions, dividing consequently the amount of agents in each of these regions.

Two important factors have an impact on the efficiency of this decision: when to split and how to split.

Each region during the simulation compares the number r of agents it is simulating with the average number a of agents simulated by its neighbors. When these $r > k_s \times a$ then the load balancing kicks in splitting the overloaded region in 9 subregions as depicted in 3: 8 will go to neighbors of the overloaded region and it will be in charge just for the central one.

As along as the inequality stands the region stays split; whenever $r < k_m \times a$ the subregions are called back to its original region.

It is worth noting that the both the split and the merge are computational expensive operations so deciding the factors k_s and k_m is a key point in the performances of the whole system.

8.2 Parameters for load balancing

In this section it is explained how to set the parameters for the load balancing in `DParticles`. The factor k_s is the variable `thresholdSplit` and specifies while k_m is the simulation parameter `thresholdMerge`.

Listing 9: Class `DParticles`

```
public DParticles(Object[] params)
    {
      ...
      gridWidth = (Integer)params[5];
      gridHeight = (Integer)params[6];
      ((DistributedMultiSchedule)schedule).setThresholdMerge(1);
        ((DistributedMultiSchedule)schedule).setThresholdSplit(5);

    }
```

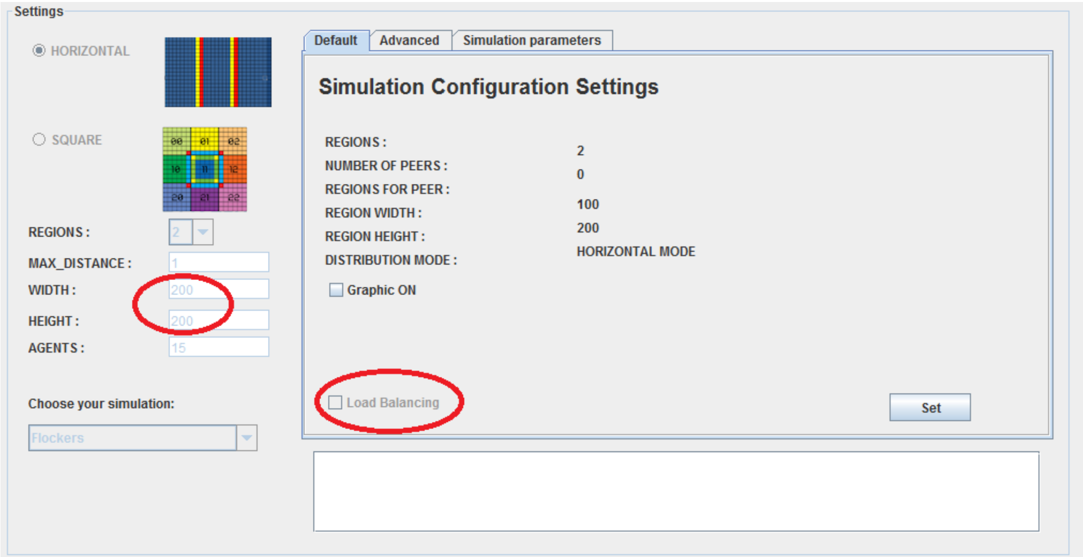

Figure 4: Master GUI.

8.3 How to use Master's GUI with load balancing

For load balancing the textbox WIDTH and the textbox HEIGHT must contain the same value and this value must be divided by $3 * sqrt(REGIONS)$. For example if the regions are 16 the value of the textbox must be divided by $3 * 4 = 12$. For enable the load balancing there is a checkbox to set, see 4. Remember that the regions must be at least 9 because of type of load balancing.

9 System Management

9.1 Introduction

In this section it is explained the system management functionalities, in detail:

- Peer auto reconnection
  - Automatic reconnection when the comunication server(CS) is restarted after a failure
- Restart simulation
  - Restart simulation without restart the CS (eg launch another simulation)

9.2 Peer auto reconnection

This feature requires no change.

9.3 Restart simulation

To enable the functionality of simulation reset it is necessary to edit the ActiveMQ configuration file.

- Windows
  The file to edit is "wrapper.conf" located in:

Listing 10: Class DParticles

```
%ACTIVEMQ_BASE%\bin\win32 or %ACTIVEMQ_BASE%\bin\win64
```

It is necessary to uncomment this line

Listing 11: File wrapper.conf

```
    ...
    # Uncomment to enable jmx
#wrapper.java.additional.n=-Dcom.sun.management.jmxremote.port=1616
#wrapper.java.additional.n=-Dcom.sun.management.jmxremote.authenticate=false
#wrapper.java.additional.n=-Dcom.sun.management.jmxremote.ssl=false
```

and then replace the "n" with the next number from the previous section

Listing 12: File wrapper.conf

```
...
wrapper.java.additional.10=-Dactivemq.conf=%ACTIVEMQ_CONF%
wrapper.java.additional.11=-Dactivemq.data=%ACTIVEMQ_DATA%
```

In this case we obtain this

Listing 13: File wrapper.conf

```
...
# Uncomment to enable remote jmx
wrapper.java.additional.12=-Dcom.sun.management.jmxremote.port=1616
wrapper.java.additional.13=-Dcom.sun.management.jmxremote.authenticate=false
wrapper.java.additional.14=-Dcom.sun.management.jmxremote.ssl=false
```

- Linux
  The file to edit is "activemq" located in:

Listing 14: File wrapper.conf

```
%ACTIVEMQ_HOME%/bin/
```

It is necessary to unccomment this line

Listing 15: File activemq

```
...
#
# ACTIVEMQ_SUNJMX_START=
# ACTIVEMQ_SUNJMX_START=

# ACTIVEMQ_SUNJMX_START=

# ACTIVEMQ_SUNJMX_START=

ACTIVEMQ_SUNJMX_START=
```

We obtain this

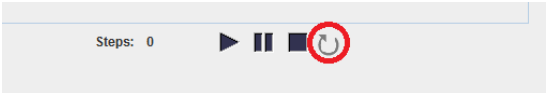

Figure 5: Simulation reset button.

Listing 16: File `activemq`

```
...
#
ACTIVEMQ_SUNJMX_START=\
```

9.4 How to use Master's GUI for restart simulation

For restart simulation there is reset button, highlighted by the red circle, see 5. When the button is clicked the current simulation stops and the parameters can be edit again and the simulation can be changed or not.