D-MASON a tutorial

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

January 9, 2013

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.

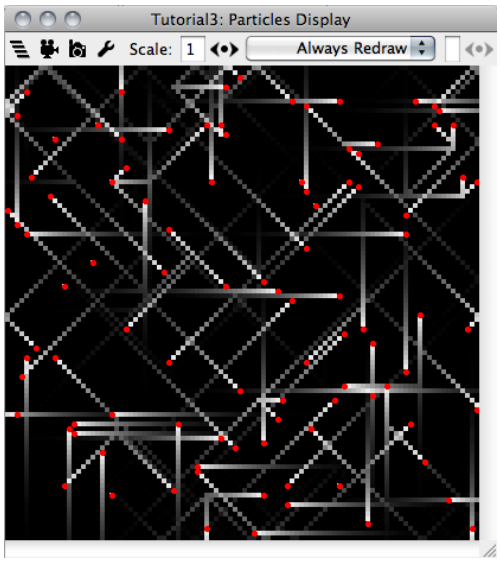

Figure 1: GUIState for `Particle` simulation

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 Fiore 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> {
private 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, "trails");

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

 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("Simulation cell "+particles.cellType+" ready
     ...");
 } 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;
 }
}
```

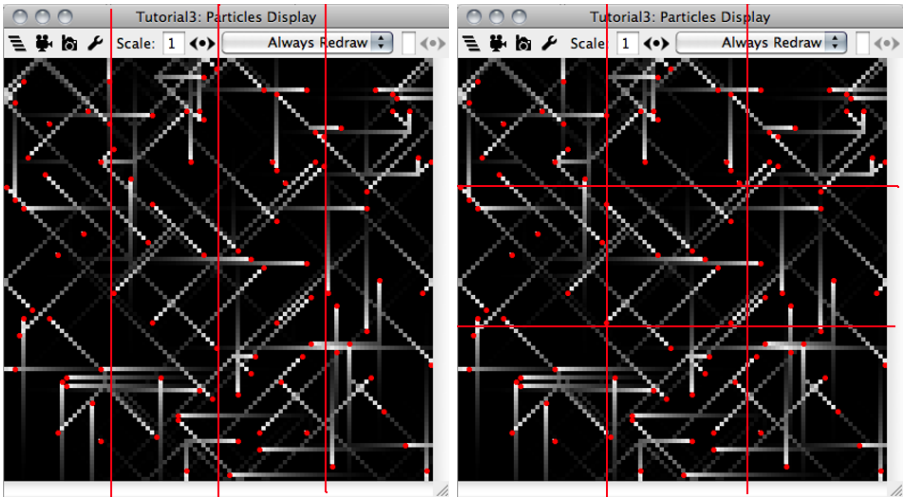

Figure 2: Respectively `HORIZONTAL` and `SQUARE` partition mode.

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 "Peer: <"+name+">"; }
...
}
```

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.

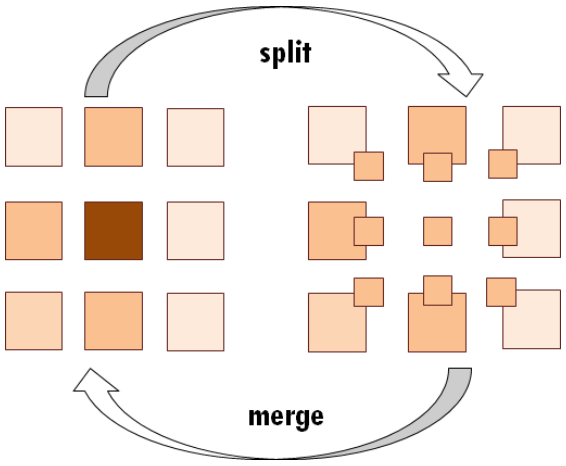

Figure 3: The load balance policy for split and merge: the darker region is overloaded and is split in 9 subregions; when the load decreases the subregions are merged back.

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);

    }
```

8.3 Master GUI and load balancing

To activate the load balancing mechanism the field must be square (same width and height) and its width (height) must be divisible by $3 \times sqrt(\#regions)$. For example when your field is divided in 16 regions then the width (height) must be divisible by $3 \times \sqrt{16} = 12$. The number of regions must be ≥ 9 because of the way the split/merge mechanism works. The next step is just to tick the checkbox in Figure 4.

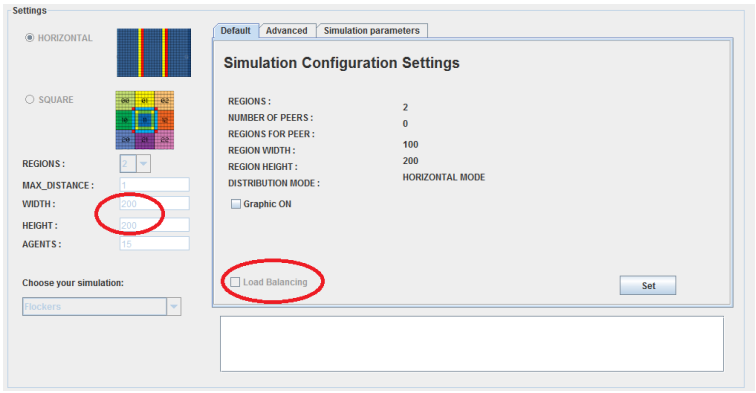

Figure 4: Master GUI.

9 System Management

In this section it is explained the System Management functionalities, in detail:

Peer Auto Reconnection Automatic reconnection when the communication server (CS) is restarted after a failure;

Restart Simulation Restart simulation without restart the CS (eg launch another simulation).

Jar Deploy Deploy simulation/update jar to workers.

While the Peer Auto Reconnection is a mechanism that has been implemented at the framework level to enable the Simulation Reset requires to modify the ActiveMQ configuration file.

Windows Under Windows the file to edit is `wrapper.conf` located in:
`%ACTIVEMQ_BASE%\bin\win32` or `%ACTIVEMQ_BASE%\bin\win64`
In this file it is necessary to uncomment the following line:

Listing 10: 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 11: 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 12: 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 Under Linux the file to edit is *activemq* located in:
`%ACTIVEMQ_HOME%/bin/`
It is necessary to uncomment this line

Listing 13: File `activemq`

```
...
#
#ACTIVEMQ_SUNJMX_START="-Dcom.sun.management.jmxremote.port=11099 "
#ACTIVEMQ_SUNJMX_START="$ACTIVEMQ_SUNJMX_START -Dcom.sun.management.jmxremote.
    password.file=${ACTIVEMQ_CONF}/jmx.password"
#ACTIVEMQ_SUNJMX_START="$ACTIVEMQ_SUNJMX_START -Dcom.sun.management.jmxremote.
    access.file=${ACTIVEMQ_CONF}/jmx.access"
#ACTIVEMQ_SUNJMX_START="$ACTIVEMQ_SUNJMX_START -Dcom.sun.management.jmxremote.ssl
    =false"
ACTIVEMQ_SUNJMX_START="$ACTIVEMQ_SUNJMX_START -Dcom.sun.management.jmxremote"
```

We obtain this

Listing 14: File `activemq`

```
...
#
ACTIVEMQ_SUNJMX_START="-Dcom.sun.management.jmxremote.port=1616 "
ACTIVEMQ_SUNJMX_START="$ACTIVEMQ_SUNJMX_START -Dcom.sun.management.jmxremote.
    authenticate=false"
#ACTIVEMQ_SUNJMX_START="$ACTIVEMQ_SUNJMX_START -Dcom.sun.management.jmxremote.
    password.file=${ACTIVEMQ_CONF}/jmx.password"
#ACTIVEMQ_SUNJMX_START="$ACTIVEMQ_SUNJMX_START -Dcom.sun.management.jmxremote.
    access.file=${ACTIVEMQ_CONF}/jmx.access"
ACTIVEMQ_SUNJMX_START="$ACTIVEMQ_SUNJMX_START -Dcom.sun.management.jmxremote.ssl=
    false"
#ACTIVEMQ_SUNJMX_START="$ACTIVEMQ_SUNJMX_START -Dcom.sun.management.jmxremote"
```

9.1 How to use Master GUI to restart simulation

To restart simulation there is reset button (in a the red circle in 5). When a simulation is stopped by clicking the button the parameters can be edit again and the simulation can be re-initialized.

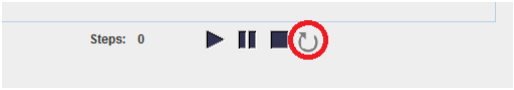

Figure 5: Simulation reset button.

9.2 Jar Deploy

9.2.1 Simulation Deploy

This feature allows the user to quickly deploy a simulation on all the workers by using jar file. Following we will describe the steps needed to generate a simulation jar file:

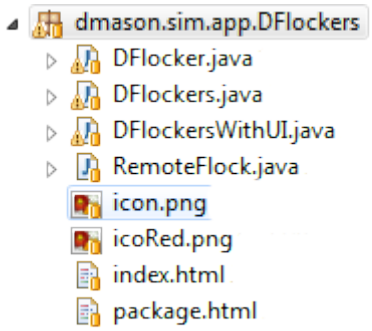

Figure 6: An example: DFlockers simulation package.

1. select the simulation package, right click on the package and choose export JAR file
2. select the main class (i.e. DFlockers in our example)

By using the MasterUI (see Figure 7) application it is possible to deploy the created file to each worker and the Master.

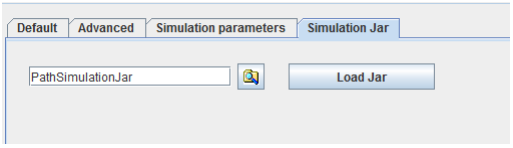

Figure 7: Simulation deploy in MasterUI.

First of all the user has to choose the jar. Using the Simulation Jar tab the user can load the simulation jar file. After this operation the simulation is available for the Master and can be chosen from the "Choose your simulation" combobox. When a simulation is run by the Master, each worker will be able to download and run the corresponding simulation jar.

9.2.2 Worker update

This feature allows the user to update the worker application.

Two different update methods have been provided:

Manual Update. The user can start a manual update (which update all the active workers) using the menu item System→Update worker in the MasterUI application (see Figure 8). The update panel will appear (Figure 9). In order to update the active workers, the user need to load the new application (a jar file). The update starts when the user clicks the "Update Worker" button. A progress bar will show the evolution of the update process.

Automatic Update. This procedure is still required since the manual update affects only active workers. Then when a new worker wake up it is important to check whether the workers need to be

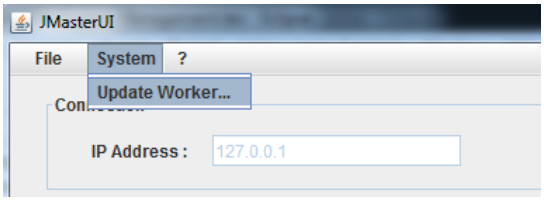

Figure 8: Manual update in MasterUI.

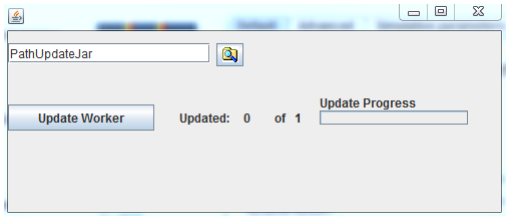

Figure 9: Manual update panel.

updated. The MasterUI is able to detect this problem (i.e. a worker not up to date) and automatically starts the update process.
The automatic update requires that the last worker jar file is available for the MasterUI application. When this file is not available a warning message appears in the MasterUI application (see Figure 10).

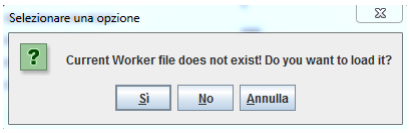

Figure 10: Jar update not found.

When the MasterUI finds a worker which requires to be updated, a message will appear to inform that an updating process is going to start (see Figure 11).

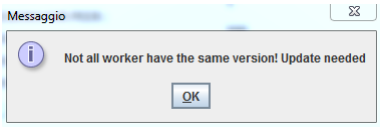

Figure 11: Starting update process.

10 The Global Inspector

Taking inspiration from the MASONinspector, we wanted to create a mechanism to let the user to inspect a distributed simulation. This is just a first a step toward a complete support of the mechanism.

An example. Think about a simulation regarding inhabitants of a city. Suppose the simulation class exposes a method called `getPopulation()` returning an `int` representing the number of alive citizens.

Now assume that our city is split in a 2×2 grid. If any worker calls the `getPopulation()` on its region of the city, that call will return the number of alive citizen living in that region of the city. In this example, if we want to know how many citizens are alive in the entire city, we need to sum the number of citizens alive in each region.

This is the goal if the Global Inspector.

10.1 Enabling the Global Inspector

The Global Inspector (GI) is embedded into the Global Viewer. It allows the user to track specific statistics about the simulation being run, and also to view agent's positions.

In order to GI to work, we need build an *inspectable* simulation, whose only goal will be to retrieve information from the workers, reduce it and show it in the model inspector.

Being the simulation class called `DParticles`, the GI will look for an inspectable class called `DParticles`*`Insp`*.

For each of the properties that the GI will display, the inspectable class will have:

- a getter method (e.g. `getNumAgents()`);
- a reducer method (e.g. `reduceNumAgents()`);
- possibly a variable (e.g. `numAgents`)

This example may not have much sense, since the number of particles in this simulation never changes, but it is simple enough to get started. Following the details the class `DParticlesInsp`:

Listing 15: Class DParticlesInsp

```
public class DParticlesInsp extends InspectableState {
  protected int numAgents;

  public int getNumAgents() {
    return numAgents;
  }

  public void reduceNumAgents(Object[] shards) {
    int sum = 0;
    int n = shards.length;
    for (int i = 0; i < n; i++)
      sum += (Integer)shards[i];
    numAgents = sum;
  }
}
```

The reduce method takes an `Object[]` as parameter and the length of this array equals the number of regions in which the field is split and, obviously, each entry of the array contains information about one region (in no particular order).

A reduce method may have or not a return value. Remember that the global inspector reads the simulation status using getter methods so what the reduce method needs has to set a variable that will be read by the getter (in this case, `reduceNumAgents()`) set the variable `numAgents` that will be read by `getNumAgents()`.

11 Load balancing for Horizontal partition

11.1 Introduction

The problems for the load balancing for horizontal partition are the same of the Grid.

11.2 The balancing mechanism

This load balancing extends or contracts the regions to preserve the equity of agents between the regions. A region more overloaded contracts its limits and gives the agents in the loosing part to the neighbors. A region alternates the balancing with the neighbors, so in a step balances with the right neighbor and in the next step balances with the left neighbor.

11.3 How to use

The load balancing for horizontal partition is simple to use. The field $rows$ must be 1 and the check button $Load\ balancing$ must be selected.

12 Thin mode

12.1 Introduction

D-MASON allocates by default the entire field for every region. For example if the field has $width = 100$ and $height = 100$ and there are two regions, each region allocates a field with $width = 100$ and $height = 100$. This is a limit because a part of field is allocated but not used. For this reason we implemented the Thin mode that allocates the portion of field really used. Note that there is a backword compatibility with previous simulations' implementation.

12.2 Difference with standard fields

In D-MASON, for example `DSparseGrid2D` as described previously, there is the method `createDSparseGrid2D` of the class `DSparseGrid2DFactory` for creating a new distributed field. For the `Thin mode` there is the method the method `createDSparseGrid2DThin` of the class `DSparseGrid2DFactory` that returns an instance of `DSparseGrid2DThin`. The process is the same with the other fields. Another difference is the invocation of the methods of the object returned from the factory: all the methods, invocated on the object, that are present in the MASON superclass must be invocated with `Thin` suffix without any additional parameter. For example `setObjectLocation` becames `setObjectLocationThin`. The last difference is on the the grid field not sparse. To get the value of a location x, y in standard mode we have `field[x][y]`, in `Thin mode` is present a method `getThin(x,y)` that must be invocated on the object returned from the factory. To set a value z in the location x, y in standard mode we have `field[x][y]=z`, in `Thin mode` is present a method `setThin(x,y,z)` that must be invocated on the object returned from the factory.