User's Guide to LaBoGrid, a distributed Lattice Boltzmann-based simulation tool

Gérard Dethier

July 2011

Contents

Changelog

0.1 (July 2011)

LaBoGrid version: 0.1

Initial version of this document.

Chapter 1

Introduction

LaBoGrid is a Lattice Boltzmann-based, experimentation-oriented, flow simulation tool. It can be executed in a wide range of environments: from desktop computers to distributed environments such as clusters. LaBoGrid is written in Java and is therefore directly executable on most modern architectures and in heterogeneous clusters (regarding architecture, installed software, etc.).

LaBoGrid is well suited to experimentation: it features methods to easily describe the sequence of high-level simulation elements to execute, without the need to produce additional source code. In addition, if the available simulation elements do not meet the user's requirements, the user can easily develop new simulation elements. LaBoGrid is therefore easy to extend.

LaBoGrid was initially developed in the frame of a research project conducted at the University of Liège (Belgium). The software is essentially based on the theory presented in 2 thesises [1, 2].

This document describes LaBoGrid, its features (see chapter 2) and, very briefly, its architecture (see chapter 3). It also shows how LaBoGrid can be configured (see chapter 4) and executed (see chapter 5) in order to simulate flows. Finally, the way LaBoGrid can be extended is presented (see chapter 6) and some additional tools provided with LaBoGrid are described (see chapter 7).

Chapter 2

Features

This chapter presents the main features of LaBoGrid. These are described in following sub-sections.

2.1 Configuration

LaBoGrid's configuration and the simulations it executes are described in a single configuration file given as input when executing LaBoGrid. See chapter 4 for more details.

2.2 Execution environments

LaBoGrid was designed to be executed on cluster of heterogeneous computers. It is able to balance efficiently the processing load among available computers in order to minimize simulations execution time (see sections 3.2 and 4.5.2). In addition, LaBoGrid features mechanisms that render it tolerant to failures i.e. unexpected interruptions of some processes (see sections 3.3 and 4.5.3).

It can also be executed on a single computer and efficiently make use of any number of available cores.

2.3 Flow simulation

LaBoGrid is able to simulate monophasic 3D flows in potentially complex structures. Lattice Boltzmann methods are used. A D3Q19 lattice (see section 4.3.1) represents the state of the flow which is driven either by pressure conditions, either by a body force applied at each point of the fluid.

Two collision operators (see section 4.4.1) are available : a Single Relaxation Time (SRT) operator[1] and a Multiple Relaxation Times (MRT) operator (for more details about these operators, see the thesis of D. A. Beugre [1]). The MRT operator is more

---

[1] The SRT operator is actually the well known Bhatnagar–Gross–Krook (BGK) operator.

stable numerically than the SRT operator, in particular when simulating turbulent flows. However, it is much more complex to compute (a multiplication by about 15 of execution time was observed for simulations using MRT instead of SRT).

The solid (see section 4.3.2) the fluid is flowing through is represented using a bitmap i.e. a 3D array of same size as lattice indicating, for each point, if it is an obstacle or not.

Pressure conditions (see section 4.4.1) are configured by giving the pressure applied to the input face of the lattice and the pressure applied on the output face of the lattice. These 2 faces are parallel, the flow therefore follows one of the space axises.

The body force (see section 4.4.1) applied at each point of the fluid is given by the components of the force along x, y and z axises.

2.4 Probes

Probes can be setup in order to regularly follow some properties of the flow (see section 4.4.2). Available probes measure the speed and density of fluid at given points or for complete slices of the lattice. New probes may easily be developed in order to extend LaBoGrid's capabilities (see section 6.5).

2.5 Simulations workflow

LaBoGrid is able to execute a sequence of LB flow simulations. The result of each simulation can be stored to disk and/or be reused as starting point for subsequent simulation. The result of a simulation can be reused later to resume the simulation or to be post-processed. Section 4.2 explains how such workflows can be described.

Simulation workflows enable to automatically store the state of a simulation at given points in time or to change some simulation parameters, like used probes, during a simulation. For example, in a simulation lasting 10000 time steps, the state of the flow is required at steps 5000 and 10000, and a probe must be used from steps 5000 to 7000 in order to follow flow speed at some given points of the lattice. The simulation can be desrcribed as a workflow of 3 simulations:

- time steps 0–5000: no probe is used, result is stored onto disk at the end of the simulation;
- time steps 5000–7000: a probe is used to regularly measure flow's speed at given points, result is not stored onto disk;
- time steps 7000–10000: no probe is used, result is stored onto disk at the end of the simulation;

2.6 Post-processing

LaBoGrid itself does not provide post-processing capabilities but includes a tool that allows to convert files resulting from a simulation into portable text files whose content is easily exploitable (see section 7.1).

2.7 Data handling server

LaBoGrid consumes and produces large amounts of data. Solid and/or result files may be required for the execution of a simulation. In addition, the result of a simulation may be stored. Finally, probes produce informations that should also be stored in order to be processed.

When executing simulations on a single machine, all data are read from and written to disk. When executing LaBoGrid on a cluster, input files must be available for each computer of the cluster. In addition, it may be convinient that all result files are centralized on a single computer the experimenter has easy access to.

LaBoGrid therefore provides a “stand alone server” (see section 7.2). This application is executed on a computer that will:

- provide input files to cluster computers,
- retrieve result files from cluster computers,
- log measures received from probes.

Chapter 3

Architecture

The architecture of LaBoGrid is briefly described in this chapter. Indeed, having an insight on LaBoGrid's inner workings is required in order to properly configure, deploy and expand it (see chapters 4 and 6).

3.1 DiMaWo framework

3.1.1 Agent-based programming

LaBoGrid is based on DiMaWo framework which suggests an agent-based programming: an application based on DiMaWo is made of agents sending and receiving messages in an asynchronous way. These agents may be distributed across several computers. Agents may produce logs about their activity. A log level can be set in order to control the verbosity of agents.

A distributed DiMaWo application (like LaBoGrid) consists of one process running per computer, each process executing one or several agents, provided by DiMaWo or the user.

3.1.2 Distributed master/worker model

The commonly used master/worker architectural model implies one master process that manages other worker processes. This model has 2 main drawbacks: the master both acts as a bottleneck and a single point of failure, hence posing scalability and robustness problems.

DiMaWo provides a distributed master/worker architectural model which almost solves above problems. The main idea is that, instead of having one master managing all workers, a tree of master-workers is built, each master-worker managing a subset of available workers. A master-worker is a special worker which executes additional managing code. The root master-worker is the leader and may, as such, execute special code (for example, operations that should be executed by only one worker). This structure is called MN-tree [2]. MN-trees are robust thanks to a master-worker replacement policy: as soon as a master-worker is unexpectedly interrupted, a worker it managed takes its place.

MN-trees have 2 parameters: the maximum number of workers managed by a master-worker and the number of master-worker children per master-worker. These 2 parameters tweak the scalability and the robustness of MN-trees: increasing the number of workers per master-worker increases robustness but may impair scalability and increasing the number of master-worker children per master-worker may increase the efficiency of some operations but increases the load on master-workers.

3.1.3 DiMaWo applications

A DiMaWo application must implement at least 2 base agents: the worker agent and the master agent. The worker agent executes the code of a worker and the master agent executes the code of the leader. These 2 agents may access the services provided by DiMaWo: broadcasting, point-to-point communication, distributed file system, shared map, etc.

In the context of LaBoGrid, worker agents execute simulation code. The master agent only triggers the simulations described in the configuration file, which is available for all workers (it must be given as input file).

3.2 Load balancing

LaBoGrid minimizes simulation execution time when executed on a heterogeneous cluster by optimizing computational load distribution and by minimizing communications. This is achieved by partitioning the lattice of a simulation into “sublattices” and distributing them among computers using the Modified Tree Walking Algorithm (MTWA) [2].

In order to work, MTWA needs a weight to be associated to each computer running a worker. This weight is measured by running benchmarks consisting of small simulations.

3.3 Fault Tolerance

In order to be tolerant to the unexpected interruption of a worker’s process, a scalable state replication scheme is used: each worker sends a copy of the state of the sublattices it is running a simulation on to several other workers which store the state on their computer’s disk. This way, in case a worker’s process is interrupted, the state of the sublattices it “hosted” may be retrieved from other workers and the simulation be restarted from last saved state.

The workers a particular worker sends its state to, called “replication nighbors”, are selected among the workers managed by the same master-worker as the worker. Consequently, the number of replication neighbors is at most the number of workers managed by a master-worker (one of the parameters of MN-trees).

Chapter 4

Configuration

This chapter describes the XML configuration file taken as input by LaBoGrid by documenting the XML schema file describing it. This file, `labogrid-conf-schema.xsd`, is contained by folder `src/main/schema/` of LaBoGrid's source tree.

4.1 XML configuration file

A LaBoGrid configuration file is composed of 4 parts:

1. a description of the simulations' workflow (see section 2.5) LaBoGrid must execute (see section 4.2);
2. simuation parameters (solid file, lattice size, etc.) that can be used for several simulations (see section 4.3), at least one set of parameters must be provided;
3. "processing chains" describing the chains of operations (streaming, collision, etc.) to apply at each time step of a simulation; a same chain can be used for several simulations (see section 4.4), at least one chain must be given;
4. parameters for LaBoGrid's middleware part (load balancing, fault tolerance, etc. (see section 4.5).

Following XML schema snippet describes a complete LaBoGrid configuration file:

```
<xs:element name="LaBoGridConfiguration">
  <xs:complexType>
    <xs:sequence minOccurs="1" maxOccurs="1">
      <xs:element name="Experiment" type="ExperimentType" />
    <xs:sequence minOccurs="1" maxOccurs="unbounded">
      <xs:element name="LBConfiguration"
        type="LBConfType" />
    </xs:sequence>
    <xs:sequence minOccurs="1" maxOccurs="unbounded">
```

```
      <xs:element name="ProcessingChain"
        type="ProcChainType" />
    </xs:sequence>
    <xs:element name="LaBoGridMiddleware"
      type="LBGMiddlewareType" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Following table shows which part each element describes:

`<Experiment>`	simulations' workflow
`<LBConfiguration>`	a simuation's parameters
`<ProcessingChain>`	a chain of operations
`<LaBoGridMiddleware>`	LaBoGrid's middleware parameters

4.2 Experiment

The `<Experiment>` element is composed of a sequence of simulations sequences. Each sequence represents "chained simulations": the result of a simulation of the sequence is always used as starting point by next simulation in the sequence, if it exists.

4.2.1 Input and output

For every simulation, an "input" and an "output" can be given. These are essentially ways of respectively retrieving and providing files: to be executed, a simulation needs to retrieve input files (solid or result files) and at the end of a simulation, the result of the simulation may be stored to be reused (in order to resume simulation or for post-processing).

An input/output has 2 parameters: a "client" class and the parameters of the client. The client implements a method to retrieve/provide files. Following XML schema snippet provides the definition of an element describing an input or an output. `clientClass` attribute is the class of the client and `parameters` attribute is the parameters of the client. The parsing of the parameters depends on the client but they are generally separated by spaces.

```
<xs:complexType name="InOutType">
  <xs:attribute name="clientClass" type="xs:string"
    use="required"/>
  <xs:attribute name="parameters" type="xs:string"
    use="required"/>
</xs:complexType>
```

LaBoGrid provides two types of clients: a local client and a remote client. The local client reads/writes files from/to disk. The remote client requests/submits files from/to a stand alone server (see section 2.7).

Local input client is implemented by following class:

```
laboGrid.ioClients.local.LocalInputClient
```

It takes 1 argument: the path to the folder containing input files.

Local output client is implemented by following class:

```
laboGrid.ioClients.local.LocalOutputClient
```

It takes 1 argument: the path to the folder the output files must be stored into.

Remote input client is implemented by following class:

```
laboGrid.ioClients.standalone.StandAloneInputClient
```

It takes 3 argument: the path to the folder containing input files on the computer running the stand alone server, the host name of the computer running stand alone server and the port the stand alone server accepts connections on.

Remote output client is implemented by following class:

```
laboGrid.ioClients.standalone.StandAloneOutputClient
```

It takes 4 argument: the maximum size in bytes of the file chunks sent to the stand alone server (increasing this parameter should accelerate transfers but increase memory usage), the path to the folder containing input files on the computer running the stand alone server, the host name of the computer running stand alone server and the port the stand alone server accepts connections on.

4.2.2 Simulations

The first simulation of a sequence has following parameters:

- an “input” i.e. a way of retriving input files (solid or result files);
- a “starting iteration” i.e. the first time step that will be executed; if this value is greater than 0, the result of a previous simulation, which must be available, is loaded from input;
- an “iterations count” i.e. the number of time steps to execute;
- the identifier of a processing chain;
- the identifier of a set of simulation parameters.
- an “output” i.e. a way of providing output files (result files);

The input must be given for first simulation of a sequence. For next simulations, it may be omitted because they reuse the result of previous simulation. However, it may be interesting to provide an input if the result of previous simulation was stored through its output. In this case, if a severe failure of cluster computers leads to the result of previous simulation to be unavailable despite fault tolerance mechanism (see section 3.3), it can still be retrieved from the input.

The starting iteration must be given for first simulation. It is omitted for next simulations of the sequence because this value depends on the starting iteration of previous simulation and the number of iterations it executes. For example, if first simulation's starting iteration is 0 and iterations count is 10, the starting iteration of next simulation is necessarily 10.

The iterations count, identifier of a processing chain and identifier of a set of simulation parameters must be given for every simulation.

The output is always optional.

Following XML schema snippet describes the elements `<Simulation>`, which represents the first simulation of a sequence, and `<NextSimulation>`, which represents next simulations. `startingIteration` attribute is the starting iteration, `iterationsCount` attribute is the iterations count, `processingChain` attribute is the identifier of a processing chain and `lbConfiguration` attribute is the identifier of a set of simulation parameters.

```
<xs:complexType name="SimulationType">
  <xs:sequence minOccurs="0" maxOccurs="1">
    <xs:sequence minOccurs="1" maxOccurs="1">
      <xs:element name="Input" type="InOutType"/>
    </xs:sequence>
    <xs:sequence minOccurs="0" maxOccurs="1">
      <xs:element name="Output" type="InOutType"/>
    </xs:sequence>
  </xs:sequence>
  <xs:attribute name="startingIteration"
    type="xs:nonNegativeInteger" use="required"/>
  <xs:attribute name="iterationsCount"
    type="xs:nonNegativeInteger" use="required"/>
  <xs:attribute name="processingChain"
    type="xs:string" use="required"/>
  <xs:attribute name="lbConfiguration"
    type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="NextSimulationType">
  <xs:sequence minOccurs="0" maxOccurs="1">
    <xs:sequence minOccurs="0" maxOccurs="1">
      <xs:element name="Input" type="InOutType"/>
    </xs:sequence>
    <xs:sequence minOccurs="0" maxOccurs="1">
      <xs:element name="Output" type="InOutType"/>
    </xs:sequence>
  </xs:sequence>
  <xs:attribute name="iterationsCount"
    type="xs:nonNegativeInteger" use="required"/>
  <xs:attribute name="processingChain"
    type="xs:string" use="required"/>
  <xs:attribute name="lbConfiguration"
```

```
    type="xs:string" use="required"/>
</xs:complexType>
```

The `<Experiment>` element is described by following XML schema snippet:

```
<xs:complexType name="ExperimentType">
  <xs:sequence minOccurs="1" maxOccurs="unbounded">
    <xs:element name="SimulationSequence"
      type="SimulationSequenceType"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="SimulationSequenceType">
  <xs:sequence minOccurs="1" maxOccurs="1">
    <xs:element name="Simulation"
      type="SimulationType"/>
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:element name="NextSimulation"
        type="NextSimulationType"/>
    </xs:sequence>
  </xs:sequence>
</xs:complexType>
```

4.2.3 Example

Below XML snippet gives an example of LaBoGrid experiment composed of 3 simulations, the first 2 being chained. We suppose a stand alone server is running on computer "Resource0" and accepts connections on TCP port 50200. Input files are taken from folder `src/main/sim-test/` and output files are stored into folder `src/main/sim-test/out/` for all simulations (see parameters of clients). Given below experiment description, LaBoGrid will:

1. execute iterations 0–10 of a simulation using parameters "conf0" and processing chain "proc0", input files are retrieved from stand alone server;
2. execute iterations 10–20 of a simulation using parameters "conf0" and processing chain "proc1", result of previous simulation is used as starting point, result files are sent to stand alone server;
3. execute iterations 20–100 of a simulation using parameters "conf0" and processing chain "proc1", input files are retrieved from stand alone server; the input files are the result files of simulation 2.

```
<Experiment>
  <SimulationSequence>
    <Simulation
      iterationsCount="10"
      lbConfiguration="conf0"
      processingChain="proc0"
      startingIteration="0">
      <Input
        clientClass="laboGrid.ioClients.standalone.
StandAloneInputClient"
        parameters="src/main/sim-test/
          Resource0 50200" />
    </Simulation>
    <NextSimulation
      iterationsCount="10" lbConfiguration="conf0"
      processingChain="proc1">
      <Output
        clientClass="laboGrid.ioClients.standalone.
StandAloneOutputClient"
        parameters="524288 src/main/sim-test/out/
          Resource0 50200" />
    </NextSimulation>
  </SimulationSequence>
  <SimulationSequence>
    <Simulation
      iterationsCount="80"
      lbConfiguration="conf0"
      processingChain="proc1"
      startingIteration="20">
      <Input
        clientClass="laboGrid.ioClients.standalone.
StandAloneInputClient"
        parameters="src/main/sim-test/
          Resource0 50200" />
      <Output
        clientClass="laboGrid.ioClients.standalone.
StandAloneOutputClient"
        parameters="524288 src/main/sim-test/out/
          Resource0 50200" />
    </Simulation>
  </SimulationSequence>
</Experiment>
```

4.3 LB Configurations

LaBoGrid's configuration file can contain one or several `<LBConfiguration>` elements (see section 4.1), an element giving a set of simulation parameters. Each element has a unique identifier used by a simulation's description to link to it (see section 4.2.2) and contains following simulation parameters:

1. lattice type (dimensions, number of velocities, etc.) and size;

2. solid type (dimensions, bitmap, etc.), the name of the solid file (file containing solid's descriptions), solid file's content type (binary, text, compressed text);
3. a way of partitioning the lattice into sublattices (see section 3.2).

An element is associated to each item above. Element `<Lattice>` describes item 1, element `<Solid>` describes item 2 and element `<SubLattices>` describes item 3. A `<LBConfiguration>` element is composed of one of each of these elements as described in following XML schema snippet (the `id` attribute is the identifier associated to the set of simulation parameters):

```
<xs:complexType name="LBConfType">
  <xs:sequence minOccurs="1" maxOccurs="1">
    <xs:element name="Lattice" type="LatticeType"/>
    <xs:element name="Solid" type="SolidType"/>
    <xs:element name="SubLattices" type="SubLatticesType"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:string" use="required"/>
</xs:complexType>
```

4.3.1 Lattice

`<Lattice>` element provides the class of the lattice (its "type") and its size. It is described by following XML schema snippet, where `size` attribute is the size of the lattice and `class` attribute its class.

```
<xs:complexType name="LatticeType">
  <xs:attribute name="size" type="xs:string" use="required"/>
  <xs:attribute name="class" type="xs:string" use="required"/>
</xs:complexType>
```

The size of the lattice must be given in the format `(x,y,z)` where `x`, `y` and `z` are positive integers. LaBoGrid provides following lattice class:

`laboGrid.lb.lattice.d3.q19.D3Q19DefaultLattice`

This class represents a 3 dimensions lattice with 19 velocities (a D3Q19 lattice).

4.3.2 Solid

`<Solid>` element provides the class of the solid (its "type"), the solid file's name and the content type of the solid file. It is described by following XML schema snippet, where `fileId` attribute is the file name of solid file, `type` attribute the solid file content type and `class` attribute the class of the solid.

```
<xs:complexType name="SolidType">
  <xs:attribute name="fileId" type="xs:string" use="required"/>
  <xs:attribute name="type" type="SolidFileType" use="required"/>
  <xs:attribute name="class" type="xs:string" use="required"/>
</xs:complexType>
```

`fileId` is the path to the solid file relative to the folder containing input files given as parameter to input client (see section 4.2.1). `type` is the content type of solid file and `class` is the class of the solid to use.

3 content types are supported by LaBoGrid: `bin`, `ascii` and `compressed-ascii`.

`bin` content type is a serialized solid object. Note that in this case, solid class is overriden by the class of serialized object.

`ascii` content type is a text file containing the size and the content of a 3D matrix, each element of the matrix being a number. If a number is not equal to 0, then the associated position in the lattice is an obstacle. Otherwise, it is not. Following pseudocode illustrates how such a file can be produced:

```
"Initialize matrix A of size (x, y, z)" ;
"Print x to file" ;
"Print y to file" ;
"Print z to file" ;
for(int k = 0; k < z; ++k) {
  for(int j = 0; j < y; ++j) {
    for(int i = 0; i < x; ++i) {
      "Print A(i,j,k) to file" ;
    }
  }
}
```

`compressed-ascii` is the same as `ascii` but the file is compressed using GZIP. In other words, if file `x.mat` contains an `ascii` solid, the file `x.mat.gz` obtained by compressing file `x.mat` with GZIP has content type `compressed-ascii`.

LaBoGrid provides one solid class:

`laboGrid.lb.solid.d3.D3SolidBitmap`

It represents 3D solids using a bitmap i.e. a 3D array of booleans.

4.3.3 Sublattices

`<SubLattices>` element gives the minimum number of sublattices to generate and the method used to partition the lattice. The method to use is given by the class name of the partitioner, called "generator", to use. Following XML schema snippet describes `<SubLattices>` element; `minSubLatticesCount` attribute gives the minimum number of sublattices to generate and `generatorClass` attribute gives the class of the generator to use:

```
<xs:complexType name="SubLatticesType">
  <xs:attribute name="minSubLatticesCount" type="xs:integer"
    use="required"/>
  <xs:attribute name="generatorClass" type="xs:string"
    use="required"/>
</xs:complexType>
```

LaBoGrid provides 3 generators for 3D lattices: a cubes generator, a cuboids generator and a slices generator. They respectively have following classes:

```
laboGrid.graphs.model.d3.D3CubesGenerator
laboGrid.graphs.model.d3.D3CuboidsGenerator
laboGrid.graphs.model.d3.D3SliceGenerator
```

Cubes generator tries to produce only cubic partitions. When this is not possible, some partitions may not be cubes. The cuboids generator produces cuboids that are all about the same size. Cuboids are, if possible, as close as possible to cubes. The slices generator produces slices of variable thickness in function of the requested number of sublattices. Note that with this generator, the number of sublattices cannot be greater than the greatest dimension of the lattice.

The cuboids generator should be preferred in most cases: it produces sublattices of almost the same size, which a desirable behavior regarding used load balancing method (see section 3.2), and produces sublattices mostly minimizing the amount of data to exchange between sublattices during simulation.

The number of produced sublattices is the higher bound on the number of cores that can be used to execute a simulation. For example, if only 1 sublattice is produced and simulation is executed on a computer with 4 cores, only 1 core will be executing the simulation on the single produced sublattice. If the number of produced sublattices is equal to 4, then all cores will be used and the simulation's execution time will be reduced.

4.3.4 Example

Following parameters describe a simulation on a D3Q19 lattice of size $(44, 44, 44)$. Solid is read from file "MU44.mat.gz" (file contains compressed text representation of solid). Finally, lattice is partitioned into 9 sublattices using cuboids generator.

```
<LBConfiguration id="conf0">
  <Lattice class="laboGrid.lb.lattice.d3.q19.
D3Q19DefaultLattice" size="(44,44,44)" />
  <Solid class="laboGrid.lb.solid.d3.D3SolidBitmap"
    fileId="MU44.mat.gz" type="compressed-ascii" />
  <SubLattices generatorClass="laboGrid.graphs.model.d3.
D3CuboidsGenerator" minSubLatticesCount="9" />
</LBConfiguration>
```

4.4 Processing chains

LaBoGrid's configuration file can contain one or several `<ProcessingChain>` elements (see section 4.1), an element giving a chain of operations to execute each time step of a simulation. Each element has a unique identifier used by a simulation's description to link to it (see section 4.2.2) and contains a sequence of "chain elements". A chain element is either a logger description or an operator description.

Following XML schema snippet describes a `<ProcessingChain>` element; `id` attribute is the identifier of the processing chain, `<Operator>` element describes an operator and `<Logger>` element describes a logger:

```
<xs:complexType name="ProcChainType">
  <xs:sequence minOccurs="0" maxOccurs="unbounded">
    <xs:group ref="ProcChainElement"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:string"/>
</xs:complexType>

<xs:group name="ProcChainElement">
  <xs:choice>
    <xs:element name="Operator" type="OperatorType"/>
    <xs:element name="Logger" type="LoggerType"/>
  </xs:choice>
</xs:group>
```

4.4.1 Operator

An operator is executed each time step and generally modifies the state of the lattice. Examples of operators are streaming operator, collision operator, pressure conditions operator, etc. An operator is described by the class implementing it and parameters used by this class.

Following XML schema snippet describes `<Operator>` element; `class` attribute is the class implementing the operator and `parameters` attribute the parameters to pass to the operator:

```
<xs:complexType name="OperatorType">
  <xs:attribute name="class" type="xs:string" use="required"/>
  <xs:attribute name="parameters" type="xs:string" use="required"/>
</xs:complexType>
```

LaBoGrid provides operators described below.

Border filler

This operator waits for incoming densities coming from other sublattices. Incoming densities are inserted into sublattice upon reception. The operator terminates its execution when all incoming densities have been received.

Class	`laboGrid.procChain.operators.BorderFiller`
Parameters	

In-place streamer

This operator executes in-place streaming. In-place streaming must be implemented by any lattice class.

Class	`laboGrid.procChain.operators.InPlaceStream`
Parameters	

Border sender

This operator extracts outgoing densities from sublattice and stores them into output buffers. The content of these buffers is sent to other sublattices. Data sending is handled by another thread and does not pause the execution of the simulation.

Class	`laboGrid.procChain.operators.NonBlockingBorderSender`
Parameters	

Operator waiting for borders to be sent

This operator pauses the simulation's execution until all outgoing densities have been sent. This is necessary before simulation goes to next time step if outgoing densities are sent using above border sender, otherwise output buffers' content may be overwritten before it was successfully sent.

Class	`laboGrid.procChain.operators.WaitBorderSent`
Parameters	

SRT collision operator for D3Q19 lattices

This operator applies the SRT collision operator on a D3Q19 lattice.

Class	`laboGrid.procChain.operators.d3.q19.D3Q19SRTCollisionOperator`
Parameters	1. density of the fluid; 2. x component of body force to apply at each point of the fluid; 3. y component of body force to apply at each point of the fluid; 4. z component of body force to apply at each point of the fluid; 5. block size, this parameter allows to enhance data locallity. A block size of 50 should give decent results on most computers.

SRT collision operator with Smagorinsky’s turbulent viscosity model for D3Q19 lattices

This operator applies the SRT collision operator combined with Smagorinsky’s turbulent viscosity model on a D3Q19 lattice.

Class	`laboGrid.procChain.operators.d3.q19.D3Q19SRTSmagoCollisionOperator`
Parameters	1. density of the fluid; 2. *x* component of body force to apply at each point of the fluid; 3. *y* component of body force to apply at each point of the fluid; 4. *z* component of body force to apply at each point of the fluid; 5. Smagorinsky subgrid constant; 6. block size, this parameter allows to enhance data locallity. A block size of 50 should give decent results on most computers.

MRT collision operator for D3Q19 lattices

This operator applies the MRT collision operator on a D3Q19 lattice.

Class	`laboGrid.procChain.operators.d3.q19.D3Q19MRTCollisionOperator`
Parameters	1. density of the fluid; 2. *x* component of body force to apply at each point of the fluid; 3. *y* component of body force to apply at each point of the fluid; 4. *z* component of body force to apply at each point of the fluid; 5. block size, this parameter allows to enhance data locallity. A block size of 50 should give decent results on most computers.

MRT collision operator with Smagorinsky’s turbulent viscosity model for D3Q19 lattices

This operator applies the MRT collision operator combined with Smagorinsky’s turbulent viscosity model on a D3Q19 lattice.

Class	`laboGrid.procChain.operators.d3.q19.D3Q19MRTSmagoCollisionOperator`
Parameters	1. density of the fluid; 2. x component of body force to apply at each point of the fluid; 3. y component of body force to apply at each point of the fluid; 4. z component of body force to apply at each point of the fluid; 5. Smagorinsky subgrid constant; 6. block size, this parameter allows to enhance data locallity. A block size of 50 should give decent results on most computers.

Pressure conditions operator for D3Q19 lattices

This operator applies pressure conditions on two opposit faces of a D3Q19 lattice.

Class	`laboGrid.procChain.operators.d3.q19.D3Q19PressureOperator`
Parameters	1. flow direction: 0 – flow along x-axis, 1 – flow along y-axis, 2 – flow along z-axis; 2. input pressure (ρ_{in}): pressure applied on the input face (generally a number slightly greater than 1); 3. output pressure (ρ_{out}): pressure applied on the output face (generally a number slightly lower than 1).

4.4.2 Logger

A logger is a probe (see section 2.4) that is not necessarily executed each time step. A logger is described by following parameters:

- an identifier used to generate recognizable file names;
- a “rate” throttling the frequence of executions of the logger, a rate of x means the logger is executed when iteration number is divisible by x;
- the class implementing the logger and the parameters used by this class;
- a “client” class and parameters used by this class.

The concept of client is similar to the clients used in the context of simulation inputs and outputs (see section 4.2.1): a client is a way of logging the data produced by corresponding probe. LaBoGrid provides 2 clients: a client logging data into a local file and a client sending data to the standalone server which writes data into a local file.

Following XML schema snippet describes `<Logger>` element; `id` attribute is the identifier of the probe, `rate` attribute is the rate (see above), `loggerClass` is the name of the class implementing the probe, `loggerParameters` are the parameters taken by the probe, `clientClass` is the name of the class implementing a client and `clientParameters` are the parameters taken by the client:

```
<xs:complexType name="LoggerType">
  <xs:attribute name="id" type="xs:string"
    use="required"/>
  <xs:attribute name="rate" type="xs:nonNegativeInteger"
    use="required"/>
  <xs:attribute name="loggerClass" type="xs:string"
    use="required"/>
  <xs:attribute name="loggerParameters" type="xs:string"
    use="required"/>
  <xs:attribute name="clientClass" type="xs:string"
    use="required"/>
  <xs:attribute name="clientParameters" type="xs:string"
    use="required"/>
</xs:complexType>
```

LaBoGrid provides 3 loggers and 2 clients. These are described below.

Iteration logger

This logger simply logs the current iteration of the simulation.

Class	`laboGrid.procChain.loggers.IterationLogger`
Parameters	

Points logger

This logger logs the state of the flow (local density, speed) at given points.

Class	`laboGrid.procChain.loggers.d3.D3PointsLogger`
Parameters	A space-separated list of points given in (x,y,z) format (without spaces).

Slice logger

This logger logs the state of the flow (local density, speed) for a given slice of the lattice.

Class	`laboGrid.procChain.loggers.d3.D3SliceLogger`
Parameters	1. slice type (can be XY, YZ or XZ); 2. slice position.

Local file client

This client logs data to a local file.

Class	`laboGrid.procChain.loggers.client.LocalFileLogClient`
Parameters	1. path to a folder that will contain log file. If the folder does not exist, it is created.

Stand alone client

This client sends data to log to the stand alone server.

Class	`laboGrid.procChain.loggers.client.StandAloneLogClient`
Parameters	1. host name of the computer executing a stand alone server; 2. port the stand alone server accepts connections on.

4.4.3 Example

Following example implies that the following operations are executed at each simulation time step (we suppose a stand alone server is running on host "ail3" and is accepting connections on port 50200):

1. outgoing densities are extracted from lattice and sent to other sublattices;
2. streaming is applied;
3. incoming densities are received and inserted into sublattice;
4. pressure conditions are applied (flow is parallel to x-axis, $\rho_{in} = 1.001$ and $\rho_{out} = 0.999$);
5. collision operator is applied (fluid has a density of 1, no body force is applied and block size is 20);
6. simulation thread is paused until the content of output buffers is sent;
7. every iteration, current iteration number is sent for logging to the stand alone server;
8. every 3 iterations, flow state at points $(0,0,0)$, $(10,10,10)$ and $(22,22,22)$ is sent to stand alone server for logging;
9. every 3 iterations, flow state of xy slice at position $z = 22$ is sent to stand alone server for logging.

```
<ProcessingChain id="proc0">
  <Operator class="laboGrid.procChain.operators.
    NonBlockingBorderSender" parameters="" />
  <Operator class="laboGrid.procChain.operators.InPlaceStream"
    parameters="" />
  <Operator class="laboGrid.procChain.operators.BorderFiller"
    parameters="" />
  <Operator class="laboGrid.procChain.operators.d3.q19.
    D3Q19PressureOperator" parameters="0 1.001 0.999" />
  <Operator class="laboGrid.procChain.operators.d3.q19.
    D3Q19MRTCollisionOperator" parameters="1 0 0 0 20" />
  <Operator class="laboGrid.procChain.operators.WaitBorderSent"
    parameters="" />
  <Logger id="iter0" rate="1"
    loggerClass="laboGrid.procChain.loggers.IterationLogger"
    loggerParameters=""
    clientClass="laboGrid.procChain.loggers.client.
StandAloneLogClient"
    clientParameters="Resource0 50200" />
  <Logger id="points0" rate="3"
    loggerClass="laboGrid.procChain.loggers.d3.D3PointsLogger"
    loggerParameters="(0,0,0) (10,10,10) (22,22,22)"
    clientClass="laboGrid.procChain.loggers.client.
StandAloneLogClient"
    clientParameters="Resource0 50200" />
  <Logger id="slices0" rate="3"
    loggerClass="laboGrid.procChain.loggers.d3.D3SliceLogger"
    loggerParameters="XY 22"
    clientClass="laboGrid.procChain.loggers.client.
StandAloneLogClient"
    clientParameters="Resource0 50200" />
</ProcessingChain>
```

4.5 Middleware

The `<LaBoGridMiddleware>` element contains parameters related to the middleware part of LaBoGrid. There are 3 categories of parameters:

- stabilization: in order to execute simulations on a set of computers, this set must be "stable" i.e. no computer should join or leave it – the stabilizer is a component of LaBoGrid which detects the stability of the set of available computers;
- load balancing: parameters related to load balancing features of LaBoGrid (see section 3.2);
- fault tolerance: parameters related to fault tolerance features of LaBoGrid (see section 3.3).

Following XML schema snippet describes `<LaBoGridMiddleware>` element; child elements of `<LaBoGridMiddleware>` correspond to above categories.

```
<xs:complexType name="LBGMiddlewareType">
  <xs:sequence minOccurs="1" maxOccurs="1">
    <xs:element name="Stabilizer" type="StabilizerType" />
    <xs:element name="LoadBalancing" type="LoadBalancingType" />
    <xs:element name="FaultTolerance" type="FaultToleranceType" />
  </xs:sequence>
</xs:complexType>
```

4.5.1 Stabilization

The stabilizer is executed by the leader (see section 3.1.2). It signals a stable set of computers if no computer joined or leaved the set during a given amount of time, called "stabilization time-out". The `<Stabilizer>` element has a single attribute: the stabilization time-out. Following XML schema snippet describes `<Stabilizer>` element.

```
<xs:complexType name="StabilizerType">
  <xs:attribute name="timeout" type="xs:nonNegativeInteger" />
</xs:complexType>
```

4.5.2 Load balancing

LaBoGrid is able to minimize execution time by optimizing the way sublattices are distributed among computers (see section 3.2). If the set of used computers is heterogeneous regarding computational power (number of operations per time unit), it must be evaluated in order to be taken into account when computing sublattices distribution: a "power model" must be built.

The computational power of a computer is evaluated using benchmarks. Load balancing parameters include benchmarks parameters: the number of iterations of a benchmark and the size of the lattice. These parameters are optional if no power model needs to be built (e.g. when all computers have same computational power).

Following XML schema snippet describes `<LoadBalancing>` element, optional `<Benchmark>` element describes benchmark parameters and `buildPowerModel` attribute enables power model construction if true; `<Benchmark>` element must be given if `buildPowerModel` is true:

```
<xs:complexType name="LoadBalancingType">
  <xs:sequence minOccurs="0" maxOccurs="1">
    <xs:element name="Benchmark" type="BenchmarkType" />
  </xs:sequence>
  <xs:attribute name="buildPowerModel" type="boolean" />
</xs:complexType>
```

Following XML schema snippet describes `<Benchmark>` element, `iterations` attribute gives the number of iterations of benchmarks and `refSizes` gives the lattice sizes to use when running benchmarks, the sizes is a list of space-separated vectors given in the format (x, y, z):

```
<xs:complexType name="BenchmarkType">
  <xs:attribute name="iterations" type="xs:string"
    use="required" />
  <xs:attribute name="refSizes" type="xs:string" use="required" />
</xs:complexType>
```

4.5.3 Fault tolerance

LaBoGrid implements a fault tolerance mechanism based on the distributed replication of state files containing flow's state (see section 3.3). Fault tolerance mechanism has following parameters:

- the "backup rate": a backup rate of x means state files are produced and replicated when current iteration is divisible by x;
- the number of replication neighbors i.e. the number of replicas of state files;
- the maximum size of file chunks to send;
- state files content type: can be uncompressed, compressed (solid and flow state are compressed) or mixed (solid is compressed, flow state is not);

Following XML schema snippet describes `<FaultTolerance>` element, `backupRate` attribute gives the backup rate, `neighborsCount` gives the number of replication neighbors, `chunkSize` gives maximum chunk size, `compressFiles` gives state files content type (raw means uncompressed data) and `replicationEnabled` enables fault tolerance mechanism if set to true:

```
<xs:complexType name="FaultToleranceType">
  <xs:attribute name="backupRate" type="xs:nonNegativeInteger"
    use="required" />
  <xs:attribute name="neighborsCount" type="xs:nonNegativeInteger"
    use="required" />
  <xs:attribute name="chunkSize" type="xs:nonNegativeInteger"
    use="required" />
  <xs:attribute name="compressFiles" type="StateFileContentType"
    use="required" />
  <xs:attribute name="replicationEnabled" type="boolean" use="required"/>
</xs:complexType>

<xs:simpleType name="StateFileContentType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="compress"></xs:enumeration>
    <xs:enumeration value="mixed"></xs:enumeration>
    <xs:enumeration value="raw"></xs:enumeration>
  </xs:restriction>
</xs:simpleType>
```

4.5.4 Example

Below example has following effect on LaBoGrid:

- a stabilization time out of 5 seconds is used;
- a power model is built using 2 benchmarks per computer: using a lattice of $(22,22,22)$ and 500 time steps, and using a lattice of $(44,44,44)$ and 200 time steps;
- fault tolerance mechanism is enabled: state files are replicated every 100 time steps on 1 replication neighbor per computer, uncompressed data are written in state files and files are transmitted in chunks of 512 kilobytes.

```
<LaBoGridMiddleware>
  <Stabilizer timeout="5000"/>
  <LoadBalancing buildPowerModel="true">
    <Benchmark
      iterations="500 200"
      refSizes="(22,22,22) (44,44,44)" />
  </LoadBalancing>
  <FaultTolerance
    replicationEnabled="true"
    backupRate="100"
    chunkSize="524288"
    compressFiles="raw"
    neighborsCount="1" />
</LaBoGridMiddleware>
```

4.6 Full configuration file

Below the complete XML schema description of a configuration file whose snippets have been presented in previous sections.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <!-- Root element -->

  <xs:element name="LaBoGridConfiguration">
    <xs:complexType>
      <xs:sequence minOccurs="1" maxOccurs="1">
        <xs:element name="Experiment" type="ExperimentType" />
      <xs:sequence minOccurs="1" maxOccurs="unbounded">
        <xs:element name="LBConfiguration"
          type="LBConfType" />
      </xs:sequence>
      <xs:sequence minOccurs="1" maxOccurs="unbounded">
        <xs:element name="ProcessingChain"
          type="ProcChainType" />
      </xs:sequence>
      <xs:element name="LaBoGridMiddleware"
        type="LBGMiddlewareType" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>


  <!-- Experiment element -->

  <xs:complexType name="ExperimentType">
    <xs:sequence minOccurs="1" maxOccurs="unbounded">
      <xs:element name="SimulationSequence"
        type="SimulationSequenceType"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="SimulationSequenceType">
    <xs:sequence minOccurs="1" maxOccurs="1">
      <xs:element name="Simulation"
        type="SimulationType"/>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element name="NextSimulation"
          type="NextSimulationType"/>
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="SimulationType">
    <xs:sequence minOccurs="0" maxOccurs="1">
      <xs:sequence minOccurs="1" maxOccurs="1">
        <xs:element name="Input" type="InOutType"/>
      </xs:sequence>
      <xs:sequence minOccurs="0" maxOccurs="1">
        <xs:element name="Output" type="InOutType"/>
      </xs:sequence>
    </xs:sequence>
    <xs:attribute name="startingIteration"
      type="xs:nonNegativeInteger" use="required"/>
    <xs:attribute name="iterationsCount"
```

```
    type="xs:nonNegativeInteger" use="required"/>
  <xs:attribute name="processingChain"
    type="xs:string" use="required"/>
  <xs:attribute name="lbConfiguration"
    type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="NextSimulationType">
  <xs:sequence minOccurs="0" maxOccurs="1">
    <xs:sequence minOccurs="0" maxOccurs="1">
      <xs:element name="Input" type="InOutType"/>
    </xs:sequence>
    <xs:sequence minOccurs="0" maxOccurs="1">
      <xs:element name="Output" type="InOutType"/>
    </xs:sequence>
  </xs:sequence>
  <xs:attribute name="iterationsCount"
    type="xs:nonNegativeInteger" use="required"/>
  <xs:attribute name="processingChain"
    type="xs:string" use="required"/>
  <xs:attribute name="lbConfiguration"
    type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="InOutType">
  <xs:attribute name="clientClass" type="xs:string"
    use="required"/>
  <xs:attribute name="parameters" type="xs:string"
    use="required"/>
</xs:complexType>


<!-- LBConfiguration element -->

<xs:complexType name="LBConfType">
  <xs:sequence minOccurs="1" maxOccurs="1">
    <xs:element name="Lattice" type="LatticeType"/>
    <xs:element name="Solid" type="SolidType"/>
    <xs:element name="SubLattices" type="SubLatticesType"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="LatticeType">
  <xs:attribute name="size" type="xs:string" use="required"/>
  <xs:attribute name="class" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="SolidType">
  <xs:attribute name="fileId" type="xs:string"
```

```
    use="required"/>
  <xs:attribute name="type" type="SolidFileType"
    use="required"/>
  <xs:attribute name="class" type="xs:string"
    use="required"/>
</xs:complexType>

<xs:complexType name="SubLatticesType">
  <xs:attribute name="minSubLatticesCount" type="xs:integer"
    use="required"/>
  <xs:attribute name="generatorClass" type="xs:string"
    use="required"/>
</xs:complexType>


<!-- ProcessingChain element -->

<xs:complexType name="ProcChainType">
  <xs:sequence minOccurs="0" maxOccurs="unbounded">
    <xs:group ref="ProcChainElement"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:string"/>
</xs:complexType>

<xs:group name="ProcChainElement">
  <xs:choice>
    <xs:element name="Operator" type="OperatorType"/>
    <xs:element name="Logger" type="LoggerType"/>
  </xs:choice>
</xs:group>

<xs:complexType name="OperatorType">
  <xs:attribute name="class" type="xs:string" use="required"/>
  <xs:attribute name="parameters" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="LoggerType">
  <xs:attribute name="id" type="xs:string"
    use="required"/>
  <xs:attribute name="rate" type="xs:nonNegativeInteger"
    use="required"/>
  <xs:attribute name="loggerClass" type="xs:string"
    use="required"/>
  <xs:attribute name="loggerParameters" type="xs:string"
    use="required"/>
  <xs:attribute name="clientClass" type="xs:string"
    use="required"/>
  <xs:attribute name="clientParameters" type="xs:string"
    use="required"/>
</xs:complexType>
```

```
  <!-- LaBoGridMiddleware element -->

  <xs:complexType name="LBGMiddlewareType">
    <xs:sequence minOccurs="1" maxOccurs="1">
      <xs:element name="Stabilizer" type="StabilizerType" />
      <xs:element name="LoadBalancing" type="LoadBalancingType" />
      <xs:element name="FaultTolerance" type="FaultToleranceType" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="StabilizerType">
    <xs:attribute name="timeout" type="xs:nonNegativeInteger" />
  </xs:complexType>

  <xs:complexType name="LoadBalancingType">
    <xs:sequence minOccurs="0" maxOccurs="1">
      <xs:element name="Benchmark" type="BenchmarkType" />
    </xs:sequence>
    <xs:attribute name="buildPowerModel" type="boolean" />
  </xs:complexType>

  <xs:complexType name="FaultToleranceType">
    <xs:attribute name="backupRate" type="xs:nonNegativeInteger"
      use="required" />
    <xs:attribute name="neighborsCount" type="xs:nonNegativeInteger"
      use="required" />
    <xs:attribute name="chunkSize" type="xs:nonNegativeInteger"
      use="required" />
    <xs:attribute name="compressFiles" type="StateFileContentType"
      use="required" />
    <xs:attribute name="replicationEnabled" type="boolean"
      use="required"/>
  </xs:complexType>

  <xs:simpleType name="StateFileContentType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="compress" />
      <xs:enumeration value="mixed" />
      <xs:enumeration value="raw" />
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

4.6.1 Example

Below the complete configuration file whose snippets have been presented in previous sections. We suppose that the XML schema file `labogrid-conf-schema.xsd` is located in the same folder as the configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<LaBoGridConfiguration
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="labogrid-conf-schema.xsd">

  <Experiment>
    <SimulationSequence>
      <Simulation
        iterationsCount="10"
        lbConfiguration="conf0"
        processingChain="proc0"
        startingIteration="0">
        <Input
          clientClass="laboGrid.ioClients.standalone.
  StandAloneInputClient"
          parameters="src/main/sim-test/
            Resource0 50200" />
      </Simulation>
      <NextSimulation
        iterationsCount="10" lbConfiguration="conf0"
        processingChain="proc1">
        <Output
          clientClass="laboGrid.ioClients.standalone.
  StandAloneOutputClient"
          parameters="524288 src/main/sim-test/out/
            Resource0 50200" />
      </NextSimulation>
    </SimulationSequence>
    <SimulationSequence>
      <Simulation
        iterationsCount="80"
        lbConfiguration="conf0"
        processingChain="proc1"
        startingIteration="20">
        <Input
          clientClass="laboGrid.ioClients.standalone.
  StandAloneInputClient"
          parameters="src/main/sim-test/
            Resource0 50200" />
        <Output
          clientClass="laboGrid.ioClients.standalone.
  StandAloneOutputClient"
          parameters="524288 src/main/sim-test/out/
```

```
            Resource0 50200" />
      </Simulation>
    </SimulationSequence>
  </Experiment>

  <LBConfiguration id="conf0">
    <Lattice class="laboGrid.lb.lattice.d3.q19.
D3Q19DefaultLattice" size="(44,44,44)" />
    <Solid class="laboGrid.lb.solid.d3.D3SolidBitmap"
      fileId="MU44.mat.gz" type="compressed-ascii" />
    <SubLattices generatorClass="laboGrid.graphs.model.d3.
D3CuboidsGenerator" minSubLatticesCount="9" />
  </LBConfiguration>

  <ProcessingChain id="proc0">
    <Operator class="laboGrid.procChain.operators.
      NonBlockingBorderSender" parameters="" />
    <Operator class="laboGrid.procChain.operators.InPlaceStream"
      parameters="" />
    <Operator class="laboGrid.procChain.operators.BorderFiller"
      parameters="" />
    <Operator class="laboGrid.procChain.operators.d3.q19.
      D3Q19PressureOperator" parameters="0 1.001 0.999" />
    <Operator class="laboGrid.procChain.operators.d3.q19.
      D3Q19MRTCollisionOperator" parameters="1 0 0 0 20" />
    <Operator class="laboGrid.procChain.operators.WaitBorderSent"
      parameters="" />
    <Logger id="iter0" rate="1"
      loggerClass="laboGrid.procChain.loggers.IterationLogger"
      loggerParameters=""
      clientClass="laboGrid.procChain.loggers.client.
StandAloneLogClient"
      clientParameters="Resource0 50200" />
    <Logger id="points0" rate="3"
      loggerClass="laboGrid.procChain.loggers.d3.D3PointsLogger"
      loggerParameters="(0,0,0) (10,10,10) (22,22,22)"
      clientClass="laboGrid.procChain.loggers.client.
StandAloneLogClient"
      clientParameters="Resource0 50200" />
    <Logger id="slices0" rate="3"
      loggerClass="laboGrid.procChain.loggers.d3.D3SliceLogger"
      loggerParameters="XY 22"
      clientClass="laboGrid.procChain.loggers.client.
StandAloneLogClient"
      clientParameters="Resource0 50200" />
  </ProcessingChain>

  <LaBoGridMiddleware>
    <Stabilizer timeout="5000"/>
    <LoadBalancing buildPowerModel="true">
```

```
      <Benchmark
        iterations="500 200"
        refSizes="(22,22,22) (44,44,44)" />
    </LoadBalancing>
    <FaultTolerance
      replicationEnabled="true"
      backupRate="100"
      chunkSize="524288"
      compressFiles="raw"
      neighborsCount="1" />
  </LaBoGridMiddleware>

</LaBoGridConfiguration>
```

Chapter 5

Execution

5.1 Environment

LaBoGrid may be deployed in a variety of execution environments (desktop computers, clusters, supercomputers, etc.). The only rule is: 1 worker per computer, even if the computer has several cores (the worker will detect and use them). The main reason for this is that the resources waste caused by the execution of several JVMs on the same computer is avoided.

Therefore:

- Executing LaBoGrid on a cluster implies that 1 JVM executing a worker is instantiated per computer.
- Executing LaBoGrid on a multi- or single-core computer implies the execution of 1 JVM executing a worker on this computer.

5.2 Launchers

DiMaWo (see section 3.1) provides helper classes that can be used to execute workers (i.e. classes providing a static "main" method). These classes are called "launchers".

DiMaWo provides 2 worker launchers useful in the context of LaBoGrid: a bootstrap worker launcher (first executed worker) and a normal worker launcher (for all subsequent workers). These 2 launchers have respectively following class names:

```
dimawo.exec.GenericBootstrapLauncher
dimawo.exec.GenericLauncher
```

They execute a particular application by instantiating specific classes implementing workers and master (see section 3.1.3) using a factory class provided by the user. This factory may take parameters.

5.2.1 Simulations

LaBoGrid provides following factory class for the execution of flow simulations:

`laboGrid.LaBoGridFactory`

The factory takes 2 arguments:

1. a configuration file (see chapter 4),
2. (optional) a power model file i.e. a file containing the result of previous benchmarks for a list of resources; the content of this file may be used by LaBoGrid when a power model is required for load balancing (see section 4.5.2).

The bootstrap launcher requires at least 6 arguments:

1. factory class name,
2. port number the worker accepts connections on,
3. working directory (the directory that will contain temporary and log files),
4. maximum number of workers managed by a master-worker (master-worker included, see section 3.1.2),
5. number of master-worker children per master-worker (see section 3.1.2),
6. verbosity level (messages produced by agents with a verbosity level lower than given level are not printed, see section 3.1.1),

Following arguments are passed to given factory.

The normal launcher requires at least 8 arguments:

1. factory class name,
2. port number the worker accepts connections on,
3. working directory (the directory that will contain temporary and log files),
4. host name of a computer already executing a worker (generally the computer executing bootstrap worker),
5. port number the already-running worker is accepting connections on,
6. maximum number of workers managed by a master-worker (master-worker included, see section 3.1.2),
7. number of master-worker children per master-worker (see section 3.1.2),
8. verbosity level (messages with a verbosity level lower than given level are not printed),

Following arguments are passed to given factory.

5.2.2 Benchmarks

Second optional argument of LaBoGrid's factory is a power model file. If it describes well the computational power of the computers that execute LaBoGrid, there is no need to execute benchmarks again. This is interesting when simulations are mostly executed on the same set of computers or subsets of the same set of computers: in this case, benchmarks are executed once for all and their result re-used several times.

LaBoGrid provides a DiMaWo application which executes all the benchmarks needed regarding a given configuration file on the computers the application is executed by. Its factory class is:

```
laboGrid.ClusterBenchmarkFactory
```

It takes 3 arguments:

1. a LaBoGrid configuration file,
2. the number of computers executing the application (i.e. the number of workers),
3. an output file name (the name of the power model file to produce).

Note that the content of a power model file is in a human readable format that may be edited by hand.

Chapter 6

Extension

LaBoGrid's configuration file, described in chapter 4, contains references to classes whose implementation is provided by LaBoGrid. However, most of them can be replaced by classes provided by the user. For example, one may want to use its own collision operator. In this case, it can implement its own collision operator and link to it in a processing chain (see section 4.4).

This chapter presents the base classes and interfaces LaBoGrid provides and that can be subclassed or implemented by the user to extend LaBoGrid's capabilities. More details are available in source code.

In order to be used, classes provided by the user must be available in JVM's classpath.

6.1 Lattice

`Lattice` element (see section 4.3.1) provides the name of a class implementing a lattice. This class must be a subclass of

```
laboGrid.lb.lattice.Lattice
```

6.2 Solid

`Solid` element (see section 4.3.2) provides the name of a class implementing a solid. This class must be a subclass of

```
laboGrid.lb.solid.Solid
```

6.3 Input and output

`Input`/`Output` element (see section 4.2.1) provides the name of a class implementing an input/output. This class must be a subclass of

`laboGrid.ioClients.InputClient`
or
`laboGrid.ioClients.OutputClient`

6.4 Processing chain operator

`Operator` element (see section 4.4.1) provides the name of a class implementing the operator. This class must be a subclass of

`laboGrid.procChain.operators.LBOperator`

6.5 Processing chain logger

`Logger` element (see section 4.4.2) provides the name of a class implementing the logger. This class must be a subclass of

`laboGrid.procChain.loggers.LBLogger`

`Logger` element also provides the name of a class implementing a client used by the logger. A class implementing a client must be a subclass of

`laboGrid.procChain.loggers.client.LogClient`

Chapter 7

Additional tools

7.1 Result files converter

Result files produced by LaBoGrid contain binary data (serialized objects). These data may be compressed. This format is not suitable if the content of these files must be processed using tools other than Java. Therefore, LaBoGrid provides a tool that converts the content of its binary result files into “portable” text files. These files contain macro variables such as density and speed evaluated at each point of the lattice.

The conversion tool produces several output files: 1 per slice of the lattice (the user must choose if the tool will produce slices that are perpendicular to x-, y- or z-axis). Each file has name “αSlice_β.txt” where α is the slice type (xy, yz or xz) and β the position of the slice.

Each file contains M lines, M being the number of sites in each slice (all slices contain the same number of sites). A line has following format

$$s_x\ s_y\ s_z\ \frac{d}{3}\ \sigma$$

where s_i is the speed along i-axis, d is the local density, σ is a flag indicating if the site is an obstacle ($\sigma = 1$) or fluid ($\sigma = 0$).

For example, let a lattice have size (a, b, c):

- with xy slice type, c “xySlice_i.txt” files are produced with $0 \leq i < c$,
- with xz slice type, b “xzSlice_i.txt” files are produced with $0 \leq i < b$,
- with yz slice type, a “yzSlice_i.txt” files are produced with $0 \leq i < a$.

Following pseudo-code illustrates the way data from converted files should be read for post-processing in the case xy slices where requested (other slice types are handled in a similar way). `xSize`, `ySize` and `zSize` are the size of the lattice along each axis.

```
for(int z = 0; z < zSize; ++z) {
    "Open file xySlice_z.txt" ;
    for(int y = 0; y < ySize; ++y) {
        for(int x = 0; x < xSize; ++x) {
            "Parse next line of file (data associated to (x,y,z)" ;
            "Exploit parsed data" ;
        }
    }
    "Close file" ;
}
```

7.2 Stand alone server

A Stand Alone Server (SAS) is instantiated on a computer in order to send input files to workers and receive result files or log data from them (see sections 4.2.1 and 4.4.2).

Following class provides the “main” method running a SAS when invoked:

`laboGrid.standalone.StandAloneDistributedAgent`

A SAS takes 3 arguments:

1. the host name of the computer hosting it,
2. the port the SAS should accept connections on,
3. the path to a folder where logs and temporary files are stored (created if it does not exist).

Bibliography

[1] D. A. Beugre. *Étude de l'écoulement d'un fluide dans des géometries complexes rencontrées en Génie Chimique par la méthode de Boltzmann sur réseau*. PhD thesis, Université de Liège, 2010.

[2] G. Dethier. *Design and Implementation of a Distributed Lattice Boltzmann-based Fluid Flow Simulation Tool*. PhD thesis, Université de Liège, 2011.