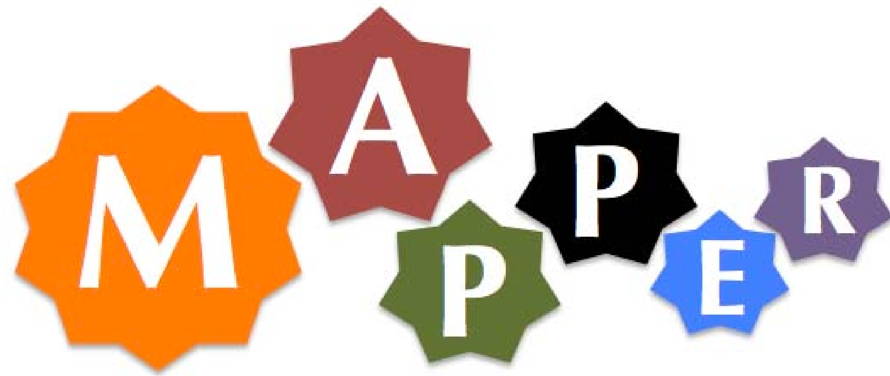


MUSCLE tutorial

10.02.2012

Joris Borgdorff

The MAPPER project receives funding from the EC's Seventh Framework Programme (FP7/2007-2013) under grant agreement n RI-261507.

Overview

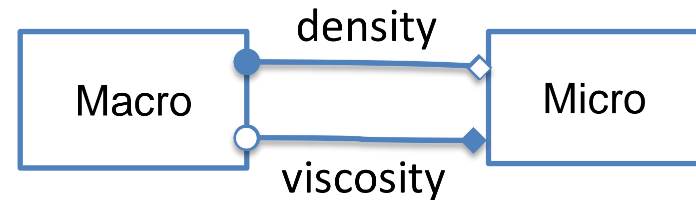
- Introduction
- Architecture
- Code examples
  - Submodel in Java
  - Submodel in C++
  - MUSCLE configuration (CxA)
- MTO
- Practical

Introduction

- Multiscale Coupling Library and Environment (MUSCLE)
- Designed to implement and execute:
  - tightly coupled multiscale models
  - on heterogeneous hardware and clusters
- Implemented in Java
  - supports C, C++ and Fortran
  - uses the JADE library for communication
    - based on message passing

Use case

- Take a multiscale model with two submodels:
  - Macro-model
  - Micro-model
- Micro is called at each iteration of Macro.
  - Macro sends density to Micro
  - Macro receives viscosity from Micro



MUSCLE architecture

Multiscale model

Macro

Micro

connect

connect

MUSCLE

Plumber

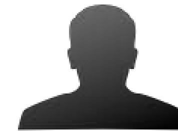
JADE

Main container

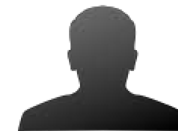
Container

Infrastructure

TCP/IP



Model and implement



Install and execute

Basic MUSCLE types

- CAController
  - implementation of a submodel or a mapper
  - unaware of other submodels
- Conduit
  - implementation of a coupling
  - already provided
  - specified by two portals in two CAControllers: an entrance (sending end) and an exit (receiving end)
    - uni-directional
- filtering: convert data or alter the flow of data of a conduit

CAController

- Submodels and mappers extend CAController
- Implement at least:
  - getScale()
    - scale of the submodels
  - addPortals()
    - define all ports: entrances and exits
  - execute()
    - run submodel

```
public class Macro extends CAController {

    public Scale getScale() { … }

    protected void addPortals() { … }

    protected void execute() { … }

}
```

CAController

- getScale() gives the scale of a submodel to MUSCLE
- Temporal and spatial scales
- Uses an SI unit library
  - Provided in third-party jar files
- In future versions, this will be replaced by MML specs

```
public class Macro extends CAController {

public Scale getScale() {
    DecimalMeasure<Duration> dt =
        DecimalMeasure.valueOf(
            BigDecimal.ONE, SI.SECOND);
    DecimalMeasure<Length> dx =
        DecimalMeasure.valueOf(
            BigDecimal.ONE, SI.METER);
    return new Scale(dt,dx);
}
```

CAController

- Define ports
  - first as field
  - instantiate in addPortals()
- Specify message datatype
  - in field
  - when instantiating
- In Java, datatype may be any class

```
public class Macro extends CAController {

ConduitEntrance<double[], double[]>
    densityEntrance;
ConduitEntrance<double[], double[]>
    viscosityExit;

protected void addPortals() {
    this.densityEntrance =
      this.addEntrance(”densityOut”,
          1, double[].class);
    this.viscosityExit =
      this.addExit(”viscosityIn”,
          1, double[].class);
}
```

CAController

- execute submodel in execute()
- for a simple example, do everything in execute;
- otherwise, use functions or classes
- willStop() is governed by a parameter called max_timesteps

```
public class Macro extends CAController {

protected void execute() {
    // f init
    double[] density = new double[5];
    for(int i = 0; i < density.length; i++) {
        density[i] = i;
    }

    while(!this.willStop();){
        // Oi
        densityEntrance.send(density);

        // S
        modifyDensity(density,
                      viscosityExit.receive());
    }
    // Of
}
```

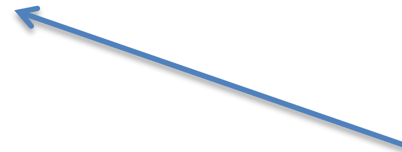
CAController

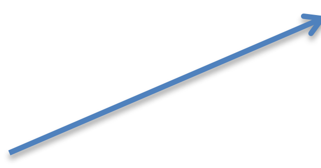
- Submodels and mappers extend CAController
- Implement at least:
  - getScale()
    - scale of the submodels
  - addPortals()
    - define all ports: entrances and exits
  - execute()

```
public class Macro extends CAController {
public Scale getScale() { …;
   return new Scale(dt, dx) }

ConduitEntrance<…> densityEntrance;
protected void addPortals() {
  densityEntrance = addEntrance(…);…}

protected void execute() { … }
   f_init
   while {
     O_i; S
   }
   O_f
}
```

CAController

```
public class Micro extends CAController {

protected void execute() { … }
    while (!willStop()) {
        f_init: densityExit.receive();
        while (!finished) {
          O_i;
          S
        }
        O_f: viscosityEntrance.send();
    }
}
}
```





```
public class Macro extends CAController {

protected void execute() { … }

    f_init
    while (!willStop()) {
      Oi: densityEntrance.send(density);
      S: viscosityExit.receive();
    }
    O_f

}
}
```

CAController

- Good practice: give each submodel its own Java package
- Good practice: keep I/O operations together in a defined set of functions
- Currently: own responsibility to prevent deadlocks

Fortran, C, and C++

- Java Native Interface (JNI) can compile Java with native executables
- MUSCLE provides facilitating classes and functions
  - Initialize conduits in Java
  - Pass them on to C/C++

MUSCLE+JNI

- Java native modifier on function
- Generate C header with javah
- Write C, C++ or Fortran
- Write a Makefile to compile the code
- import the code in Java

```
public class CMacro extends CAController {

private native void callNative(JNIConduitEntrance e);

public Scale getScale() { ... etc ... }

private JNIConduitEntrance<double[],double[]> density;
public void addPortals() {
 density= addJNIEntrance(”densityOut", 1, double[].class);
}

private static boolean libraryLoaded = false;
protected void execute() {
 if (!libraryLoaded)
  System.loadLibrary(”cmacro”);
 }

 callNative(density);
}
}
```

MUSCLE+JNI

- Java native modifier on function
- Generate C header with javah
- Write C, C++ or Fortran
- Write a Makefile to compile the code
- import the code in Java

```
JNIEXPORT void JNICALL
 Java_examples_simplecpp_CMacro_callNative
 (JNIEnv* env, jobject obj, jobject entranceJref)
{
try {
 KernelController kernel(env, obj);
 ConduitEntranceArray<jdouble>* densityEntrance
  = new ConduitEntranceArray<jdouble>(env, entranceJref);

 JNIArray<jdouble> density(kernel.getEnv(), 5);

 for(int time = 0; !kernel.willStop(); time ++) {

  // process data
  for(int i = 0; i < density.size(); i++) {
   density[i] = i;
  }

  // dump to our portals
  densityEntrance->send(density);
 }
} catch(...) {
 std::cerr<<"error"<<std::endl;
}
}
```

Configuration

- Coupling topology and parameters are given in a Ruby CxA file
- use any Ruby syntax
- Retrieve parameter:
  - CxADescription.ONLY. getIntProperty(”max_timesteps");

```
# configure cxa properties
cxa = Cxa.LAST

cxa.env["max_timesteps"] = 2

# declare kernels
cxa.add_kernel(‘macro’,’eu.mapper.Macro')
cxa.add_kernel(‘micro’,'eu.mapper.Micro')

# configure connection scheme
cs = cxa.cs

cs.attach(‘macro' => ‘micro') {
tie(’densityOut', 'densityIn')
}
cs.attach(‘micro' => ‘macro') {
tie(’viscosityOut', ’viscosityIn')
}
```

Filtering

- Data sent over a coupling can be filtered
- Drop messages, multiply or average them, or change data or datatype
- Specify in config wiht a single parameter after underscore

```
cs.attach(‘macro' => ‘micro') {
    tie(’densityOut', 'densityIn’,
    Conduit.new(
"muscle.core.conduit.AutomaticConduit",
    ["muscle.core.conduit.filter.MultiplyFilterDouble_0.5”,
    “muscle.core.conduit.filter.DropFilter_2”]
    )
}
```

Run application

- List available kernels:
  - $ muscle --cxa_file src/cxa/example.cxa.rb
- Start multiple submodels per machine
  - $ muscle --cxa_file src/cxa/example.cxa.rb --main plumber w r
- Start multiple simulations on multiple machines
  - grass2$ muscle --cxa_file src/cxa/example.cxa.rb --localport 51234 --localhost grass2 --main plumber w
  - grass1$ muscle --cxa_file src/cxa/example.cxa.rb --mainhost grass2 --mainport 51234 r

MUSCLE architecture

Multiscale model

Macro

Micro

connect

connect

MUSCLE

Plumber

JADE

Main container

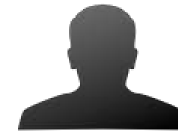
Container

Infrastructure

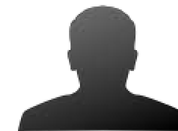
TCP/IP

grass1

grass2



Model and implement



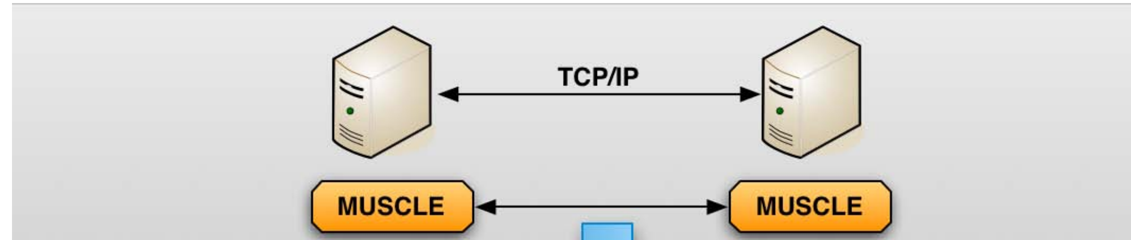
Install and execute

Run application

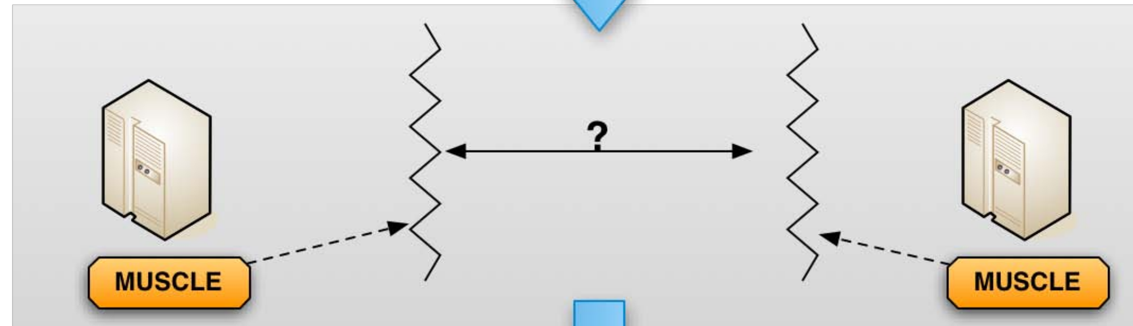
- Start multiple simulations on multiple clusters
  - grass1$ muscle --intercluster --cxa_file src/cxa/example.cxa.rb --main plumber w
  - mavrino$ muscle --intercluster --cxa_file src/cxa/example.cxa.rb --mainhost 150.254.173.215 --mainport 51234 r

MUSCLE transport overlay

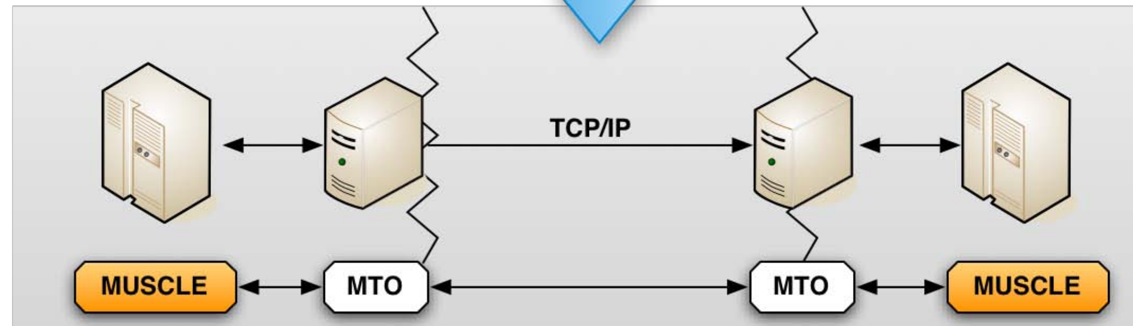
Single cluster or no firewall



Multi-cluster with firewall

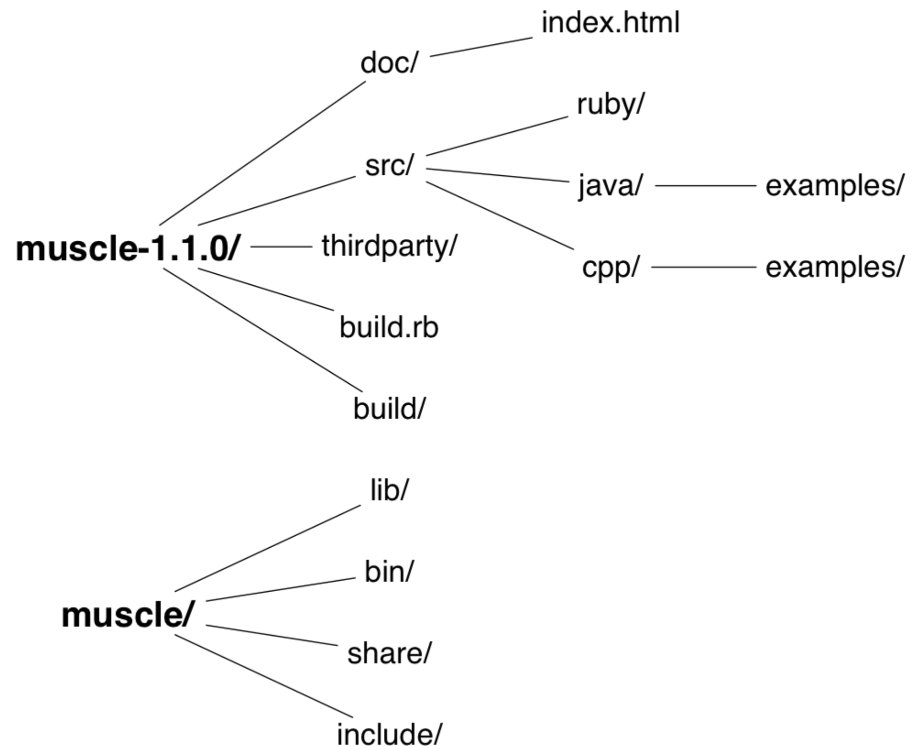


Multi-cluster with MUSCLE transport overlay (MTO)



Try it out!

- Get the code yourself:
  - http://apps.man.poznan.pl/trac/muscle
  - With installation manual
  - Includes java dependencies
  - Take a look in muscle/src/java/examples



Joris Borgdorff

University of Amsterdam

j.borgdorff@uva.nl