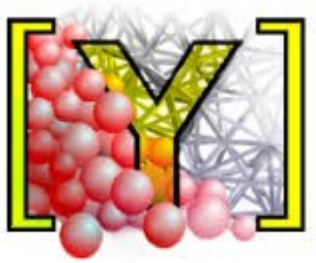
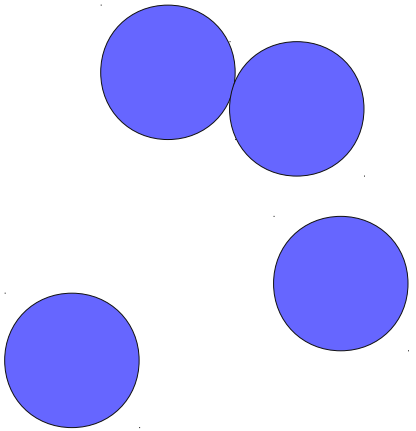
Yade-DEM-MPI

Introduction

Bruno Chareyre, Grenoble INP, 3SR



Yade-DEM



DEM Cycle:

0- If triggered: broad contact detection

1- for each interaction compute interaction forces f_{ij} ...

$$f_{ij} \mathrel{+}= \Delta t \;\; df_{ij}/dt$$

2- … and accumulate forces on the particles

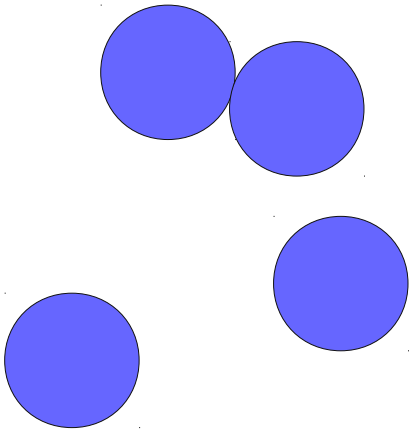
$$F_i = \Sigma_k f_{ik}$$

3- for each particle update velocity + position:

$$v^{n+1} = v^n + \Delta t * F/m$$

$$x^{n+1} = x^n + \Delta t * v^{n+1}$$

Yade-DEM



Programming techniques:

C++ with python wrapping (boost::python)

Boost::serialization

OpenMP

and much more

boost::python

Example

Consider this piece of C++ code that we want to use in python:

```
vector<int> myRange(int n)
{
   vector<int> list;
   for (int k=0; k<n; n++) list.push_back(k);
   return list;
}
```

boost::python

Example

Consider this piece of C++ code that we want to use in python:

```
vector<int> myRange(int n)
{
   vector<int> list;
   for (int k=0; k<n; n++) list.push_back(k);
   return list;
}
```

It is enough to append:

```
#include <boost/python.hpp>
BOOST_PYTHON_MODULE(myModule)
{
    boost::python::def("myRange", myRange);
}
```

boost::python

Example

```
vector<int> myRange(int n) {
   vector<int> list;
   for (int k=0; k<n; n++) list.push_back(k);
   return list;
}

#include <boost/python.hpp>
BOOST_PYTHON_MODULE(myModule) {
    boost::python::def("myRange", myRange);
}
```

Compilation produces a dynamic library which python can import as a module:

```
>>> from myModule import *
>>> x=myRange(10)
>>> print x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

YADE_CLASS macro

Without python wrapping the class declaration of “Sphere” would be:

```
// Geometry of spherical particle
class Sphere: public Shape{
»       public:
»       »       // Radius [m]
»       »       Real radius;
»       »       // constructor
»       »       Sphere (): radius(NaN) {createIndex();}
};
```

Yade is imposing a different form in which declaration, initialization, wrapping and documentation are simultaneous:

```
class Sphere: public Shape{
»       YADE_CLASS_BASE_DOC_ATTRS_CTOR(Sphere,Shape,"Geometry of spherical particle.",
»       »       ((Real,radius,NaN,,"Radius [m]")),
»       »       createIndex(); /*ctor*/
»       );
};
```

YADE_CLASS macro

Functions as well (and much more):

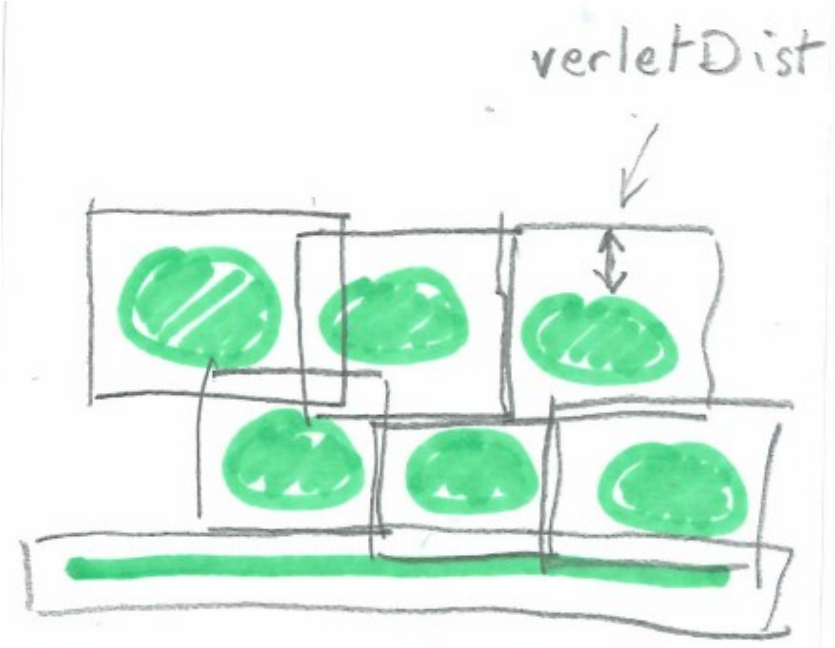
```
class Sphere: public Shape{
»	Real newFunction(const char* path);

»	YADE_CLASS_BASE_DOC_ATTRS_CTOR_PY(Sphere,Shape,"Geometry of spherical particle.",
»	»	((Real,radius,NaN,,"Radius [m]")),
»	»	createIndex(); /*ctor*/,
»	»	.def(newFunction, &Sphere::newFunction, boost::python::arg("folder")="./",
		"Write into a file. This is a cross-ref to :yref:`Body`")
»	);
};
```

Result:

Python wrapping is a (mandatory) part of the class declaration, it appears in all header *.hpp files

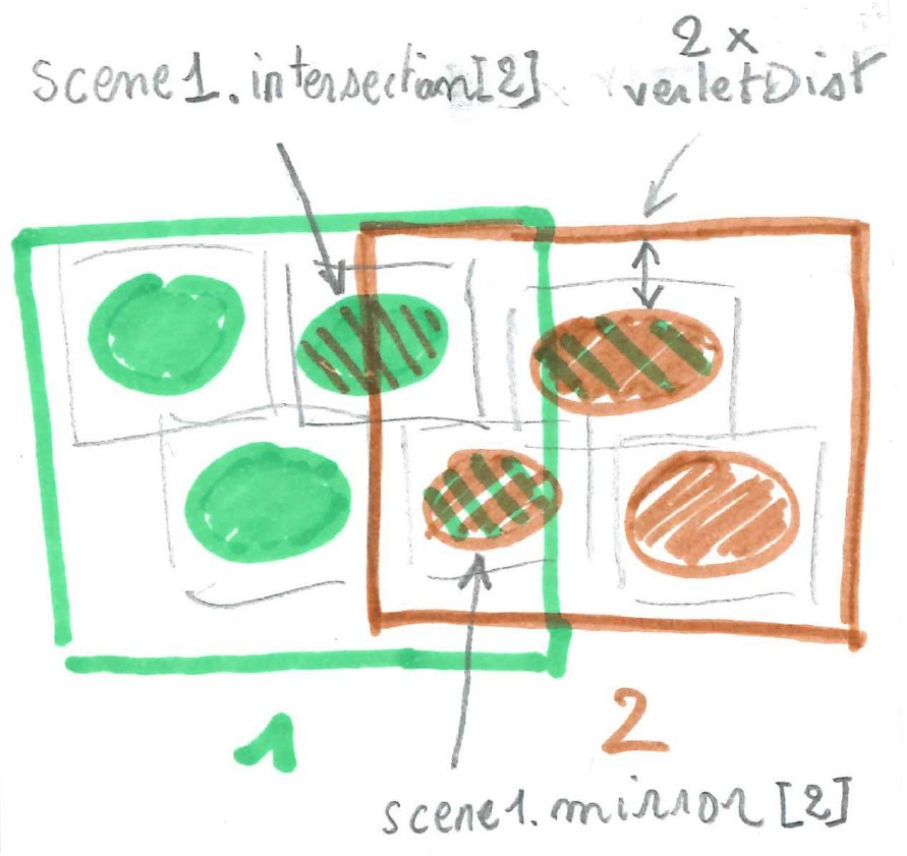
Yade-MPI: implemented logic



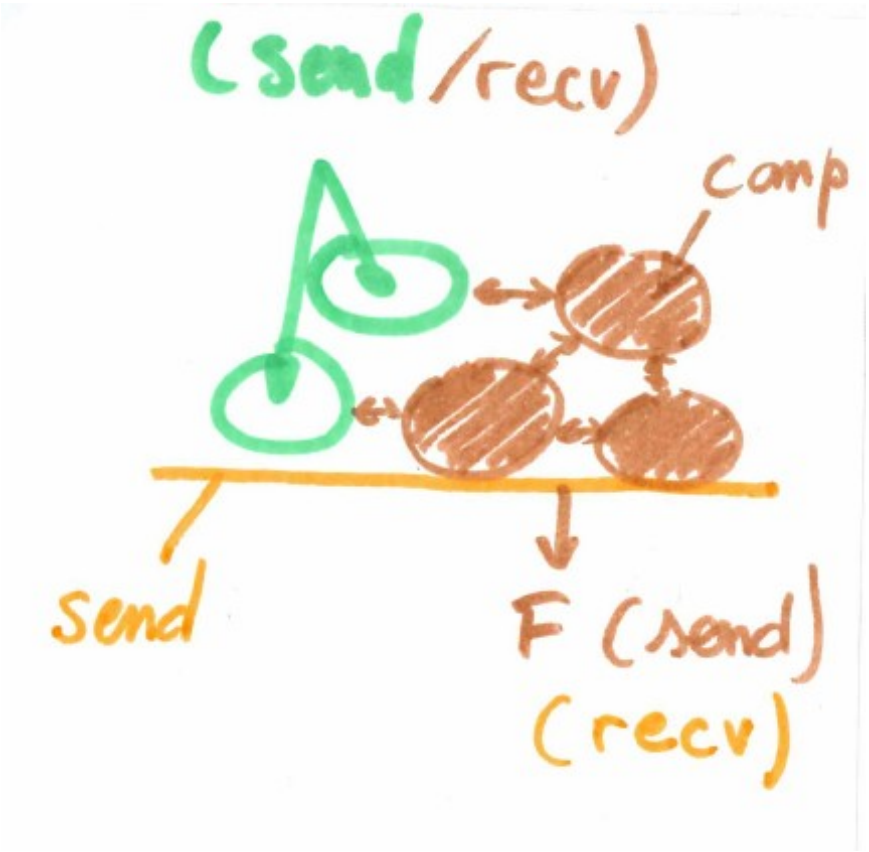
Yade-MPI: implemented logic

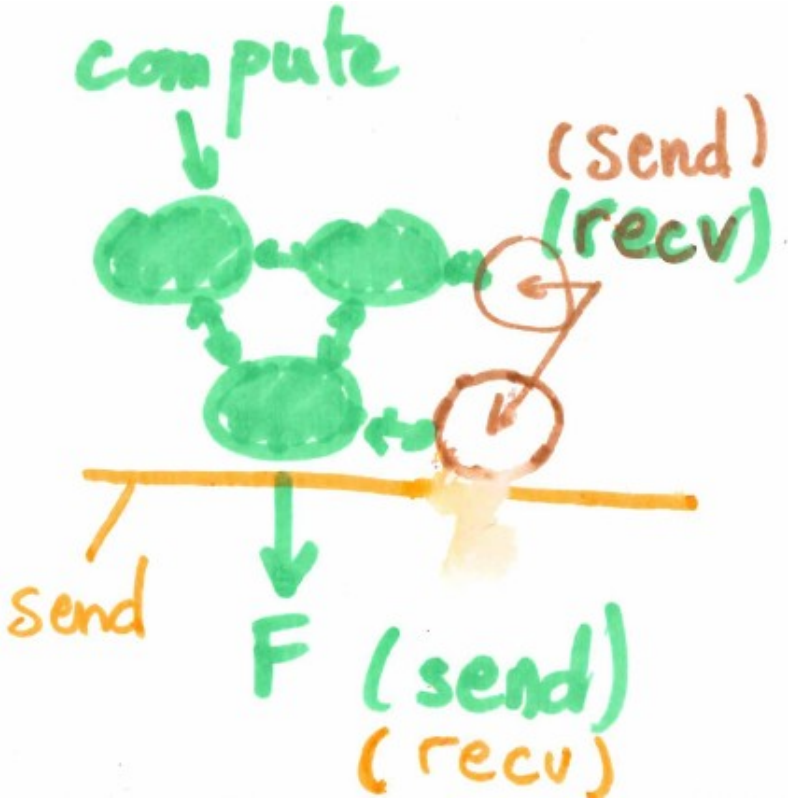
O.subD.intersections[k]: bodies in “O” intersecting bound of another (“k”) subdomain

O.subD.mirrorIntersections[k]: bodies in “k” intersecting bound of “O” (the list is received from “k”)



Yade-MPI: implemented logic





Yade-MPI: communications
(git:65806456f)

Initial split:
Boost::serialization + basic mpi4py comm (python pickling)

```
389                     #distribute work
390                     for worker in range(1,numThreads):
391                         sceneAsString=yade.Omega().sceneToString()
392                         comm.send(sceneAsString, dest=worker, tag=_SCENE_)
```

Intersections: basic mpi4py alone

```
424                 for worker in range(1,numThreads):#FIXME: we actually don't need so much data since at
425                     #states= [[id,O.bodies[id].state,O.bodies[id].shape] for id in intersections[wo
426                     #wprint("sending states to "+str(worker)+" ("+str(len(subD.intersections[worke
427                     comm.send(subD.intersections[worker], dest=worker, tag=_MIRROR_INTERSECTIONS_)
```

Yade-MPI: communications

(git:65806456f)

Updated states (critical)
→ 3 optional methods depending on mpy.OPTIMIZE_COM and mpy.USE_CPP_MPI

```
301 »       »       if not OPTIMIZE_COM:
302 »       »       »       comm.send(genUpdatedStates(O.subD.intersections[k]), dest=k, tag=_ID_STATE
303 »       »       else:
304 »       »       »       if not USE_CPP_MPI:
305 »       »       »       »       reqs.append(comm.isend(subD.getStateValues(k), dest=k, tag=_ID_STA
306 »       »       »       else:
307 »       »       »       »       O.subD.mpiSendStates(k)
```

Yade-MPI: communications
(git:65806456f)

Updated states (critical)

Ultimate hack: move forming message + comm. to C++

```
73      void mpiSendStates(/*vector<double> message,*/ unsigned otherSubdomain){
74          std::vector<double> vals = getStateValues(otherSubdomain);
75          MPI_Send(&vals.front(), vals.size(), MPI_DOUBLE, otherSubdomain, 177, MPI_
76      }
77
78      void mpiRecvStates(unsigned otherSubdomain){
79          if (mirrorIntersections.size() <= otherSubdomain) LOG_ERROR("inconsistent
80          if (stateBuffer.size() <= otherSubdomain) stateBuffer.resize(otherSubdomai
81          const vector<Body::id_t>& b_ids = mirrorIntersections[otherSubdomain];
82          unsigned nb = b_ids.size()*13;
83          vector<Real>& vals = stateBuffer[otherSubdomain];
84          vals.resize(nb);
85          int recv_count;
86          MPI_Status recv_status;
87          MPI_Recv(&vals.front(), nb, MPI_DOUBLE, otherSubdomain, 177, MPI_COMM_WORL
```

Yade-MPI: test scripts
(git:65806456f)

trunk/examples/mpi/testMPI_2D.py
- gravity deposition
- runs with/without mpiexec
- (np-1) subdomains in 2D
- only one collision detection (very small timestep)

trunk/examples/mpi/testMPI_2D_withMerge.py
- collision detection when needed
- needs merge/re-split
- runs until final equilibrium (and beyhond...)
- save merged scene at the end for validation

Yade-MPI: questions

Scaling on many cores?

profiling?

An alternative to c++/mpi hack? (what type of message passing?)

Why is force on floor different?

What is the max size given that master has the whole scene?

Best balance in terms of collision detection vs. virtual interactions?

Spawn a pool of workers to enable interactive run?

How to merge interactions? (needs a clear logic)

How to merge/re-split efficiently?

Is there a smarter way for global update, without merging (obviously there is but it needs design and implementation)?

How to organize load balancing peer-to-peer?

Collective communications for initialization?

An interactive mode to manipulate scenes with fake “rank” in non-mpi run (for algorithmic experiments)?