

Sofa Paralellization

The SOFA team

2007

1 Athapascan Interface Overview

Athapascan is an interface to describe a data flow graph, composed by a set of tasks accessing data according to a given pattern. All the data to be inserted in the Data Flow must be of the type `Shared<TYPE>` where `TYPE` is the type of information we want to represent in the graph. Tasks are created using a Fork primitive. Each time we create a new task we must specify the access mode for each parameter.

With this information Athapascan is able to link the tasks respecting the data access order. Here is an example where we create a `Shared` variable, that is accessed by two tasks, one writing a value and another one that reads it.

```
class Task1{
public:
void operator()(Shared_w<int> x){
x.write(2);
}
};
class Task2{
public:
void operator()(Shared_rw<int> x){
int & x1=x.access();
x1++;
}
};

class Task3{
public:
void operator()(Shared_r<int> x){
cout<<x.read()<<endl;
}
};

void main(){

Shared<int> var1; //Create Vari
Fork<Task1>()(var1); //Insert task 1 in the Graph
```

```

Fork<Task2>()(var1);//insert task 2 in the Graph
Fork<Task2>()(var1);//insert task 2 in the Graph
Fork<Task3>()(var1);//insert task 3 in the Graph
Sync(); //Execute the graph
}

```

All the tasks inserted in a graph must be implemented as Functors. A Functor is a class that overloads the operator(). When using Functors we are sure that none of the computation depends on class attributes, and all the needed data is passed as parameter. The remote execution of this kind of task becomes easier, as we only need to transfer the Functors arguments.

A **Shared_r** represents a read only data. It must be accessed using the read() function, that returns a **const** reference to the data. A read/write argument is represented by a Shared_rw, it is accessed by the access() function, that returns a pointer to the data which can be modified. A write only argument, Shared_w, can only be written through the **write(newValue)** function, it ensures that there will be no reading in the object, as it doesn't returns a pointer to the data.

There is also a cumulative write type, that allows to make parallel accumulative operations. It's not fully functional, but will be used on addForce Operations.

2 Using Athapascan inside Sofa

To create an Athapascan Graph in Sofa we need to specify the tasks we want to insert in the graph and execute in parallel, and some Shared data, that are accessed by those tasks.

2.1 Shared Data

To have Athapascan Shared types in Sofa, we have created new types to be used by multivectors, a VecCoordShared and a VecCoordDeriv. They are implemented as follows:

```

typedef Shared<VecCoord> VecCoordShared;
typedef Shared<VecDeriv> VecDerivShared;

```

For compatibility reasons we conserved all the VecCoord and VecDeriv variables and created equivalent variables of type VecCoordShared and VecDerivShared. For example for a MechanicalObject we have **xSh** attribute that is used when executing the Athapascan implementation, and a **x** attribute that is used by the standard implementation. By the same way there are getX and getXSh object functions. There is also a getXfromSh and getVfromSh that can be used to access directly the data inside a Shared. It only makes sense to access getXfromSh before and after the graph execution, as during the graph execution we have no guarantees on the consistency of this information.

Shared versions of **Data** and **DataPtr** were created as **DataShared** **DataPtrShared**. They must be used when we have a Data that must to be considered for the data dependency graph. Also it is useful to avoid large Data to be passed as value instead of reference. A task argument that is not a Shared is always passed by copy, while a Shared type is always passed as reference. A copy of a Shared occurs only for network data transfer on remote calls.