MultiThreading Plugin

SOFA Training days 2013
Montpellier

Federico Spadoni, Hervé Delingette

Asclepios

Add multithreading to SOFA

Why multi-thread ?

- Speed-up computation (in addition to GPU)
- Keep interactive visualisation despite slow computation

Multithreading design:

* Task and Scheduler (*KAAPI, intel TBB*)

Problems:

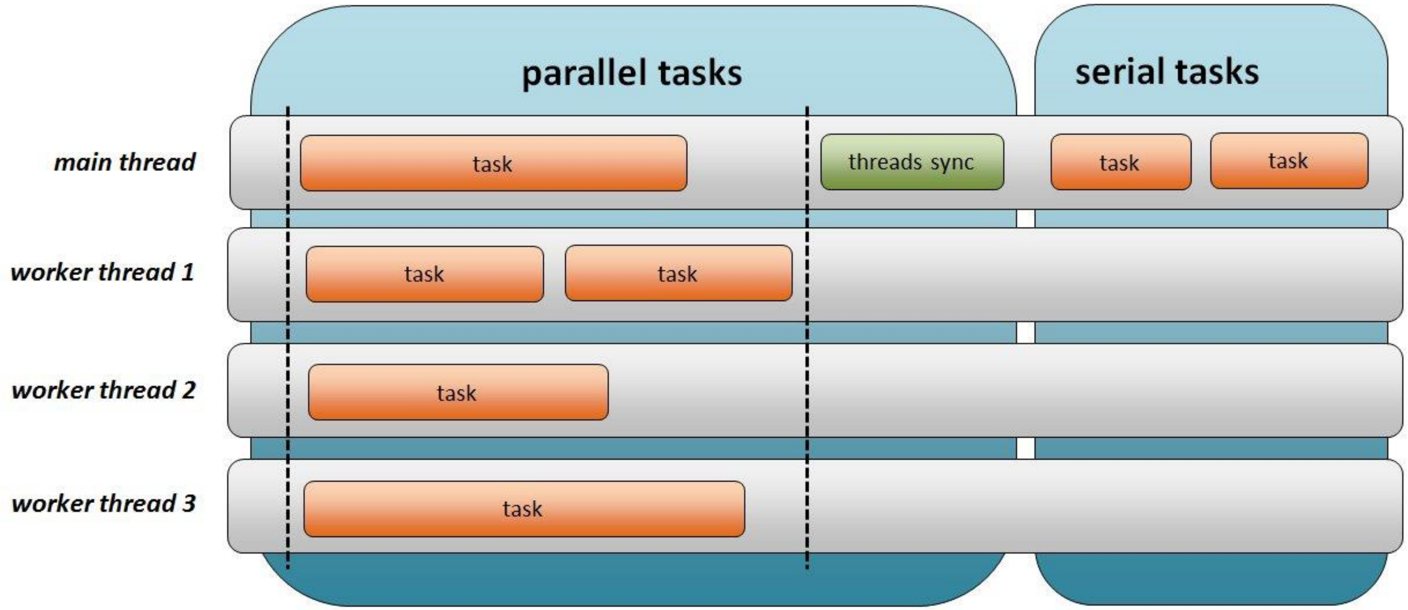
? Shared Data Synchronization

Constraint:

! Do not change Sofa core code

! Time

Task and Scheduler Overview



- the computation work load must be functionally decomposed into tasks.
- the scheduler maps the tasks onto physical threads

SOFA

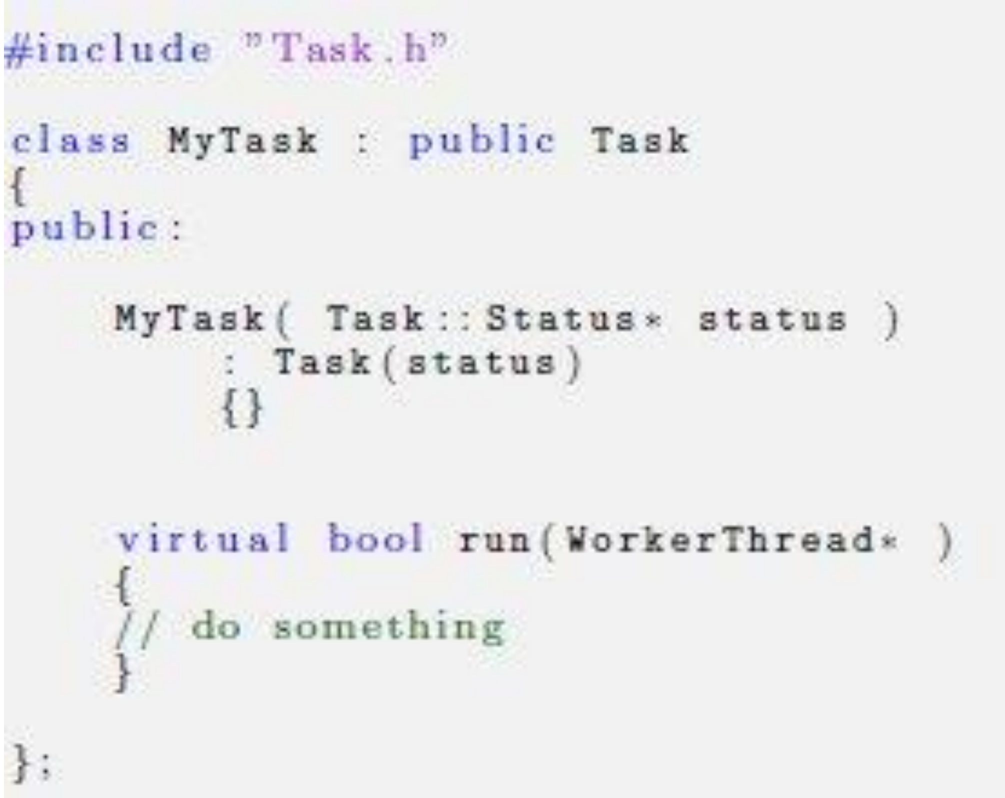
Task

- An independent functional block that can be executed in parallel
- Faster than a thread to create and destroy for the operating system
- It can itself create new tasks
- do not have data dependencies between other tasks executed concurrently

Task Scheduler (boost::thread)

- creates *n-1* worker threads on a system with *n* logical cores, and takes care of their synchronization (idling, running).
- each worker thread manages its own queue of tasks.
  - every time one task finishes the worker thread pops the next one from the top of its queue
  - when a task is created it is pushed directly on to the top of the queue
- the scheduler steals the bottom half of the queue of a busy thread and gives it to a starving thread

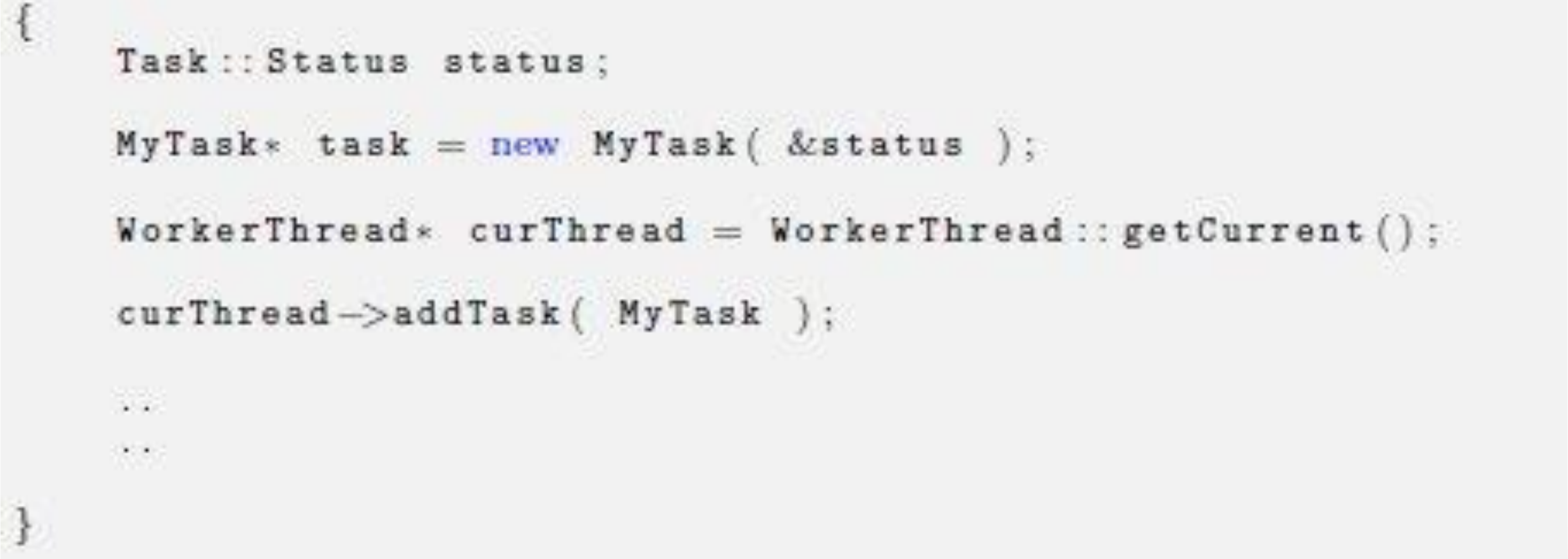
Task implementation



```
#include "Task.h"

class MyTask : public Task
{
public:

    MyTask( Task::Status* status )
        : Task(status)
        {}


    virtual bool run(WorkerThread* )
    {
    // do something
    }

};
```

- Override the *run()* function
- Pass a *task::Status* pointer to the constructor

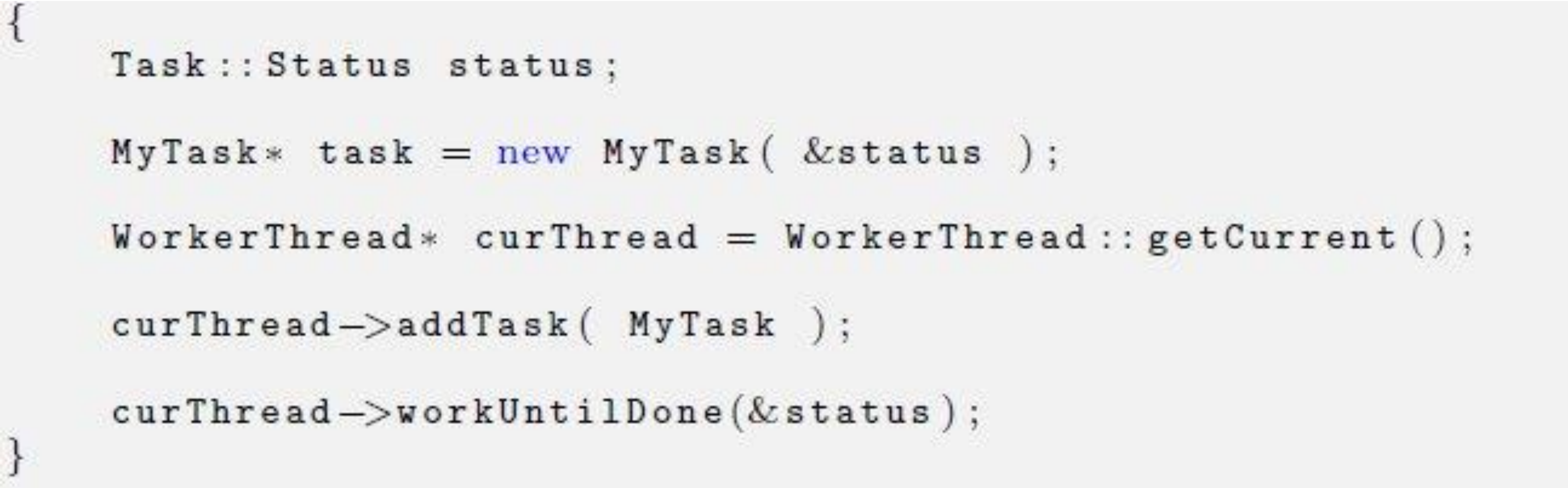
Task creation and queuing



```
{
      Task::Status status;

      MyTask* task = new MyTask( &status );

      WorkerThread* curThread = WorkerThread::getCurrent();

      curThread->addTask( MyTask );

      ..
      ..
}
```

- The *addTask()* push *MyTask* in the queue of the current thread
- the current thread keeps executing the code and it can create and queue other tasks.
- Take care of the *MyTask* destruction

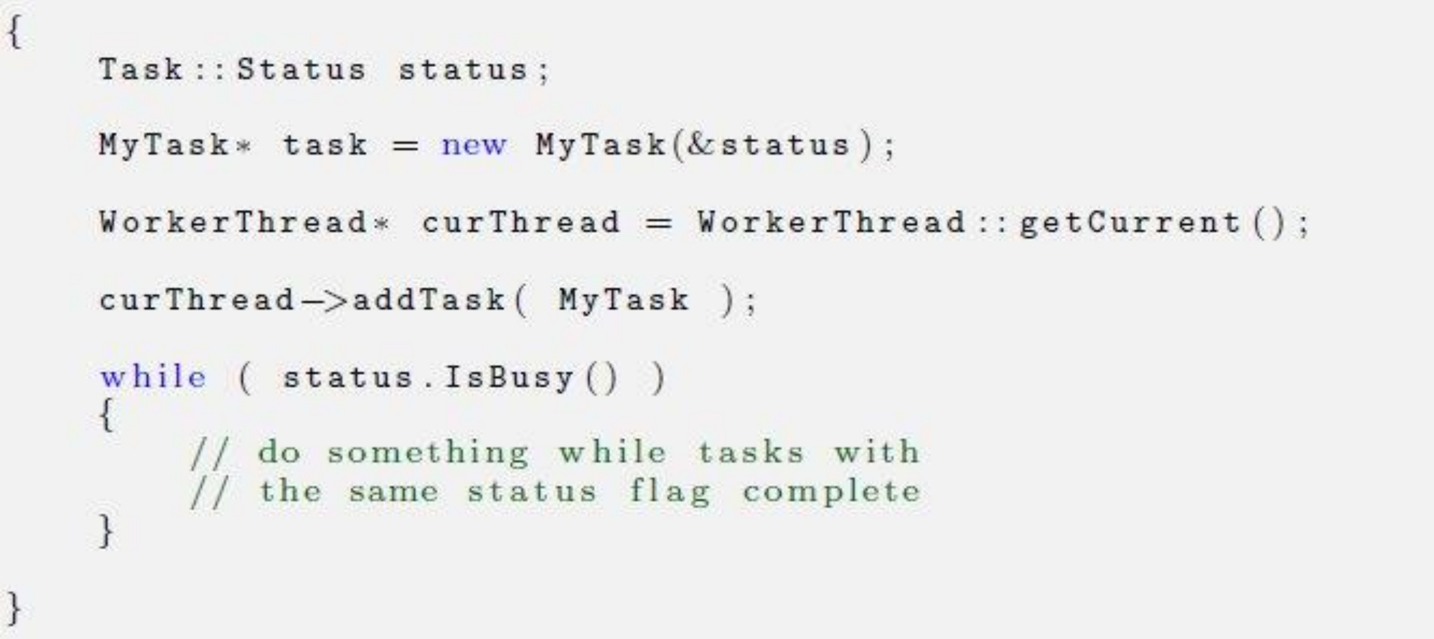
Task synchronization (blocking)



```
{
    Task::Status status;

    MyTask* task = new MyTask( &status );

    WorkerThread* curThread = WorkerThread::getCurrent();

    curThread->addTask( MyTask );

    curThread->workUntilDone(&status);
}
```

- *workUntilDone()* stops the execution untill all the tasks with the same *status* completed
- the current thread starts to execute tasks from its queue or steals from other thread queue.

Task synchronization (not blocking 1)

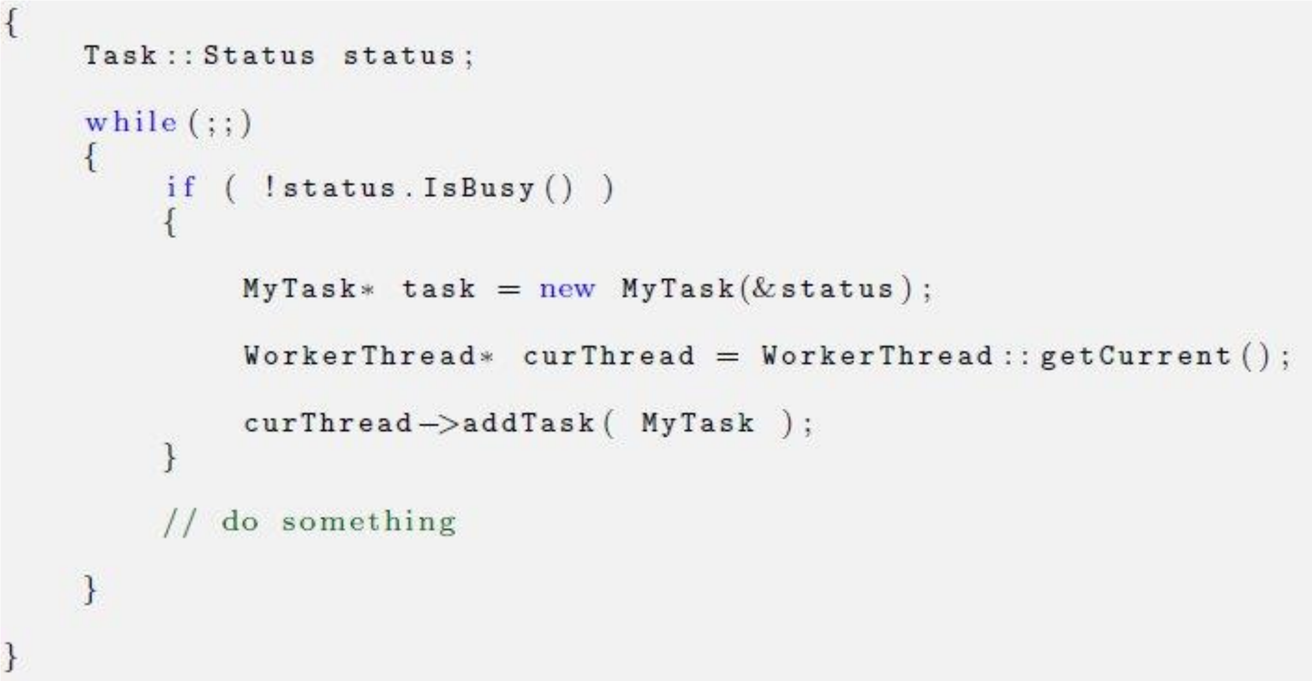


```
{
    Task::Status status;

    MyTask* task = new MyTask(&status);

    WorkerThread* curThread = WorkerThread::getCurrent();

    curThread->addTask( MyTask );

    while ( status.IsBusy() )
    {
        // do something while tasks with
        // the same status flag complete
    }

}
```

- It executes some code inside the *while* loop concurrently with the tasks already queued until all the tasks with the same *status* completed *(status.IsBusy() == false)*

SOFA

Task synchronization (not blocking 2)



```
{
    Task::Status status;

    while(;;)
    {
        if ( !status.IsBusy() )
        {

            MyTask* task = new MyTask(&status);

            WorkerThread* curThread = WorkerThread::getCurrent();

            curThread->addTask( MyTask );
        }

        // do something

    }
}
```

- The *if* statement block is executed when *MyTask* is not running
- The *while* loop queues continuously *MyTask* as soon as it completed

Creating tasks

- Component level:

  running two or more components in parallel

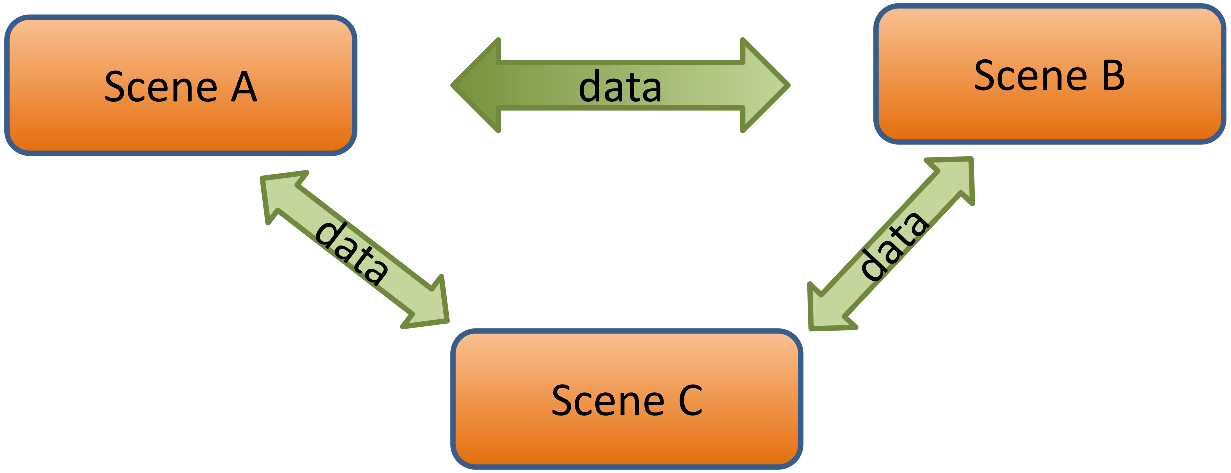
  AnimationLoopParallel runs all the *BaseAnimationLoop::step()* function found in the child nodes in parallel

- in functions:

  Parallelizing some floating point computation inside a *for loop*

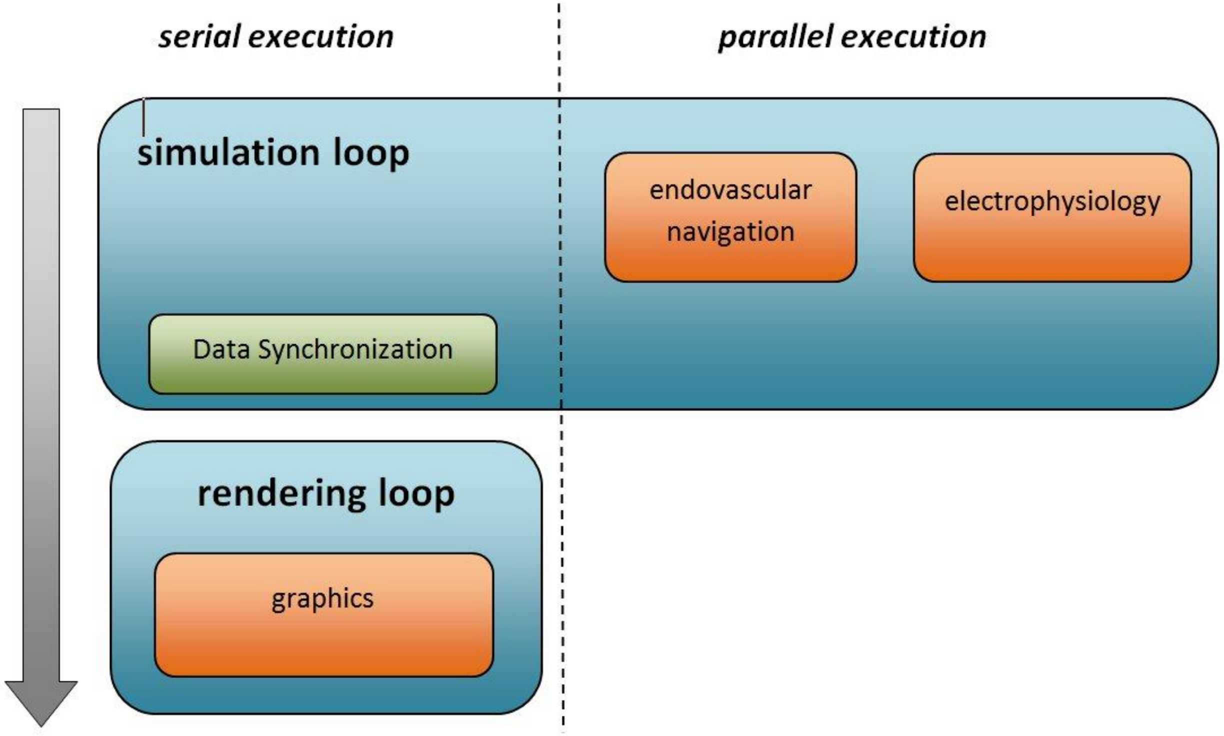
  BeamLinearMapping_mt in *apply() applyJ() applyJT()*

AnimationLoopParallel and *DataExchange*
(component level parallelism)



```
<node “root”>
    <AnimationLoopParallel>
            Scene A
            Scene B
            Scene C
    <DataExchange source=“@scenepath” dest=“@scenepath” />
<node />
```

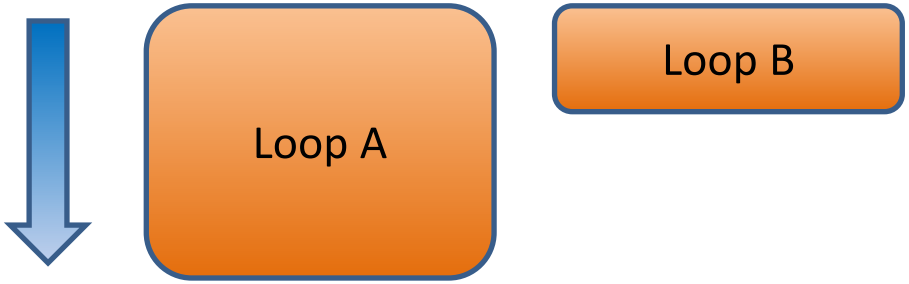
Combining Multi-threading & GPU for RF Ablation Simulation



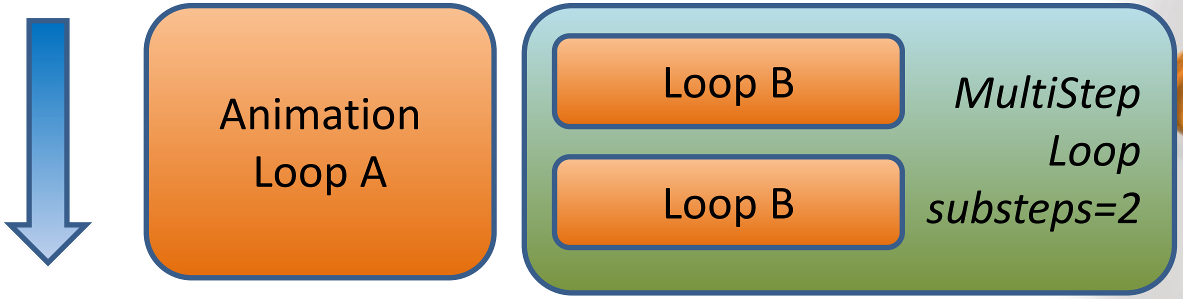
SOFA

AnimationLoopParallel Tips

each animation loop should be manually balanced to get almost the same duration:



replace the AnimationLoop of the fastests scenes with a *MultiStepAnimationLoop* and increase the number of *substep* untill to get a decrease of *fps.* Keep the time steps consistent



AnimationLoopParallel Limitations

- No Collision interaction between objects in different scenes

- No mouse interaction with the objects in the scenes

BeamLinearMapping_mt (*for* loop parallelization)

- It inherits from *BeamLinearMapping* and overrides three virtual functions that contain a *for* loop: *apply(), applyJ() and applyJT()*

- Creates 3 task for each function overridden *applyTask, applyJTask and applyJTTask* and keeps locally in the task some shared data

- It adds the *granularity* attribute: the number of iterations of the *for* loop assigned and executed for each task.

Parallelize for loop

```
for ( i=0; i<N, ++i ) {  some_code[i] }
```

- Define a *forTask* class with two indices members *first* and *last*
- Pass and initialize the *first* and *last* member in the constructor:

```
forTask::forTask( int first,int last, task::status* )
```

- Copy the loop in the run function using the *first* and *last* as range values:

```
 forTask::run() { for ( i=first; i<last, ++i )  {some_code[i] }  }
```

- Allocate the forTask(s) and add to the queue of the current thread

```
for ( i=0; i<N; i += grainSize )
        currentThread->addTask( new forTask( i, i+grainsize, &status) )
```

- Wait for all the tasks completion

waitUntilDone(status);

Create a Parallel component

- Find the computationally intensive work with a performance profiler tool (vs analyzer, ..)
- decompose it into tasks, taking in account the parallelism level:
  - independent components that can run concurrently.
  - computationally intensive loop inside a function
- Check the share data and keep it local in the task.
- For big shared data arrays avoid or minimize the *false sharing* problem (keep more than 64bytes memory distance between the data acces from different cores)

Future work

- Add a *Parallel_For* function to parallelize automatically *for loop* tasks:

  void Parallel_For(const forTask, const GrainSize&)*

- memory pool allocator for fast run-time tasks creation and destruction