The Distributed-SDF Domain

Daniel Lázaro Cuadrado
Anders Peter Ravn, Peter Koch
Aalborg University
Aalborg, Denmark

Overview

- Motivation
- What is the Distributed-SDF domain?
- How to use it?
- Calculation of the parallel schedule
- Client / Server Approach
- The Server
- The Client
- Software packages
- Further Work
- Conclusions
Motivation

- Ptolemy simulations are performed in one machine
  - Sequentially or threaded (but sharing the same CPU)
- Memory limitations
  - Locally installed memory
  - JVM
- Why SDF?
  - Dataflow is a good candidate formalism for distribution
  - Allows for static scheduling
  - One of the most popular

What is the Distributed-SDF Domain?

- Extended version of the existing SDF Domain that performs the simulation in a distributed manner
  - Smaller simulation times
    - For models with some degree of parallelism
    - Specially those where cost(computation) > >> cost(communication)
    - Allow bigger models (in terms of memory)
  - Exploits the degree of parallelism many models expose in their topology
  - It is transparent to the user
  - It requires a distributed platform to perform the simulation
  - Keep the existing architecture untouched and only extending it
How to use it?

Calculation of the parallel schedule

- To take advantage of the inherent parallelism of the model we need to generate a parallel schedule to determine which actors can be executed in parallel.
- Performs a topological sort of the graph that can be constructed with the data dependencies among the actors.
  - The existing SDF Scheduler produces schedules in a deep-first fashion.

\[
t(A) + \ldots + t(G) > t(A) + \max(t(B),t(C)) + \max(t(D),t(E),t(F),t(G)) + \text{toh}
\]

Time overhead = communication + initialization \ll\ll simulation time
The Server

- Prepares for discovery of a service locator
  - Loads various settings as for example:
    - Unicast locators (predefined location where to search for a service locator)
    - The service class \(\rightarrow\) DistributedActorWrapper.class (class that provides the distributed service)
  - Discovers a Service Locator
    - Unicast (specific location is known)
    - Multicast (no location is known)
    - Both
  - Creates and exports the Service
    - Exports an instance of a proxy class based on the service implementation to the Service Locator
      - This proxy allows to make RMI calls to the implementation
  - Stays alive
    - Maintains the registration lease with the service locator.
    - The registration of the service has to be renewed.
Server (Discovery and Service Registration)

The Client

- Prepare for discovery of a Service Locator
  - ClientServerInteractionManager encapsulates the Jini functionality
  - Calculates the number of servers required (number of actors)
  - Loads various settings as for example unicast locators
- Discover a Service Locator
  - Either unicast, multicast or both
- Looking up Services
  - ServiceDiscoveryManager, LookupCache → DistributedActorWrapper
- Filtering Services
  - Makes sure that the gathered services are alive
    - It can happen that a service has died and it is still registered if the lease has not expired.
    - Checks if there is a sufficient number of services to perform the simulation
- Map Actors onto Services
  - Creates a mapping that assigns a server for every actor
- Calls to the Services (RMI)
Service Lookup

Service Locator

Service Proxy

Server

Client

Look up Services

Client / Server interaction
Java RMI

Service Interface

ptolemy: distributed: common: DistributedActor

ptolemy: actor: Executable

RemoteDistributedActor

DistributedActorWrapper

java::rmi::Remote

java::rmi::RemoteException
Server and Service

MoML description of a pre-initialized actor

Class loaded from local storage

Message passing (Distributed Receivers)

- Receivers are created at every connected input port to hold tokens for every connection

- Two new types of distributed receivers have been created
  - DistributedSDFReceiver extends SDFReceiver with an unique ID in order to identify Receivers when distributed
  - The DistributedReceiver forwards tokens to remote services
The Service

```
inputport_i (ID_a, ... , ID_n)
outputport_j (service_y (ID_x, ... , ID_n), ... , service_m (ID_u, ... , ID_j))
```

```
fire()
getAddress()
initialize()
iterate()
loadMoML()
prefire()
preeinitialize()
put()
setConnections()
setPortTypes()
stop()
stopFire()
terminate()
wrapup()
```

Distributed Message Passing (Decentralized)

```
send (0, t0)
generic.put(t0, (IDI,...,ID j))
receiver.put(t0, (IDI,...,ID j))
receiver.put(t0)
```

```
(Service,put(t0, (IDI,...,ID j)))
Service.put(t0, (IDI,...,ID j))
```

```
DistributedActor
DistributedDirector
Composite (DistributedTypedCompositeActor)
DistributedTypedIORelation
DistributedReceiver
```

```
ateg.generic.put(t0, (IDI,...,ID j))
```

```
actor1.put(t0, (IDI,...,ID j))
receiver.generic.put(t0, (IDI,...,ID j))
receiver.put(t0, (IDI,...,ID j))
```

```
Server A Server B
```

```
getRemoteReceivers()
```

```
director.setListOfIds()
```

```
port.createReceivers()
```

```
director.setListOfIds()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```

```
port.createReceivers()
```
Issuing commands in parallel and synchronization (Client)

- In order to allow parallel execution a Thread (ClientThread) is created to handle calls to different servers in parallel.
- These threads prevent the main thread of execution to be blocked by the remote calls to the remote services.
- A synchronization mechanism to issue and access commands in parallel is provided by ThreadSynchronizer.
- Centralized.

DistributedSDFDirector

Parallel Schedule

((AB)(CDE)...) 

No set of commands is issued before the previous set is consumed

[(ClientThread 1, iterate) 
...
(ClientThread N, iterate)]

- Gets Command
- Executes Command
- Sets Ready

Service Proxy A

Service Proxy B

ThreadSynchronizer

synchronizer.setCommands() 

notifyAll() 

wait() 

wait() 

COMMANDS Map
Client’s Software Architecture

Members:
- _rateVariables
- _externalRates
- _firingVector

Methods:
- _setFiringVector
- _simulateExternalInputs
- _countUnfulfilledInputs
- _computeMaximumFirings
- _simulateInputConsumption
- _getFiringCount

were modified their visibility from private to protected.

Software Packages

- ptolemy.distributed.actor
  - DistributedDirector, DistributedReceiver, DistributedTypedCompositeActor, DistributedTypedIORelation
- ptolemy.distributed.actor.lib
  - Library of distributed actors
- ptolemy.distributed.client
  - ClientServerInteractionManager, ClientThread, ThreadSynchronizer
- ptolemy.distributed.common
  - DistributedActor Interface
- ptolemy.distributed.config
  - Jini config files
- ptolemy.distributed.rmi (Server classes)
  - DistributedActorWrapper, DistributedServerRMI_Generic, RemoteDistributedActor
- ptolemy.distributed.util
  - DistributedUtilities
- ptolemy.domains.sdf.kernel
  - DistributedSDFDirector, Scheduler & Receiver
Further Work

- **Optimization of the initialization phase**
  - Reduce to one single call for each actor
  - Perform initialization in parallel

- **Security + Robustness**
  - Jini -> Jxta
  - Implement distributed versions of other domains
  - Allow for remote loading of classes as opposed to local loading

- **Pipelining**

Pipelining

- **To increase parallelism**
  - Can be applied to models without loops

![Diagram](https://example.com/diagram.png)

Buffering Phase:

- $((A))$
- $((A B))$
- $((A B C))$

Fully Parallel:

- $((A B C D))$
Conclusions

• The Distributed-SDF domain automates distributed simulation of SDF models.
• It does not modify the existing architecture, just extends it
• Implements common features that can be reutilized to make distributed versions of other domains
• Allows to speedup simulations (specially for models where the computation cost > communication cost)
• Allows for larger models by distributing the memory load