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**

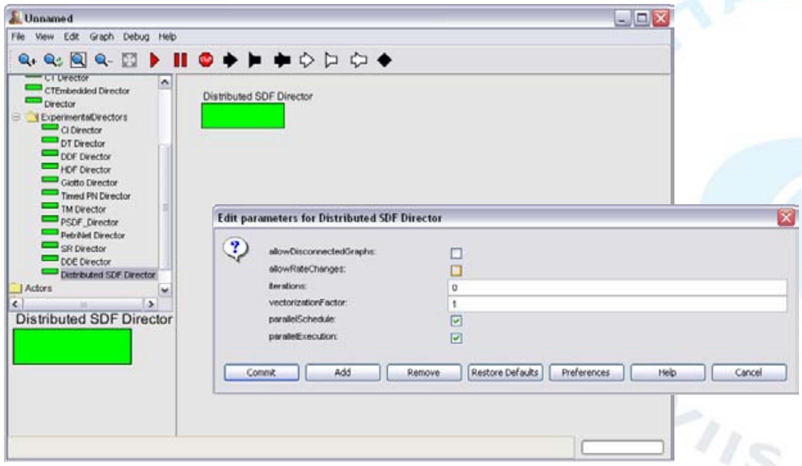
Motivation

- **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?

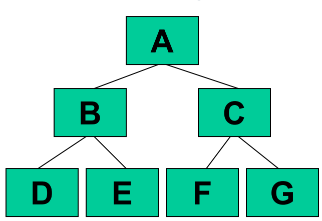
- **Extended version of the existing SDF Domain that performs the simulation in a distributed manner**
  - Smaller simulation times
    - For models with some degree of parallelism
    - Specially those where cost(computation) > >> cost(communication)
  - Allow bigger models (in terms of memory)
- **Exploits 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.



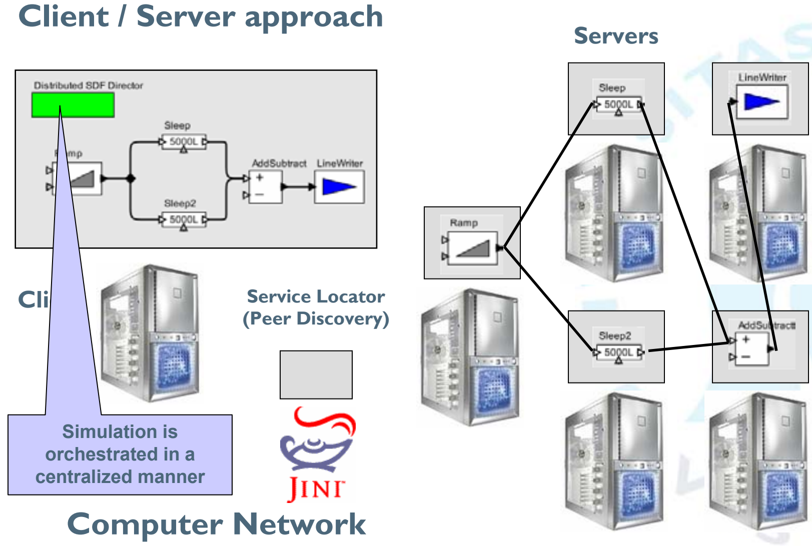
Sequential: (A B D E C F G)

Parallel: ((A) (B C) (D E F G))

t(A) + … + t(G) > t(A) + max(t(B),t(C)) + max(t(D),t(E),t(F),t(G)) + toh

Time overhead = communication + initialization <<< simulation time

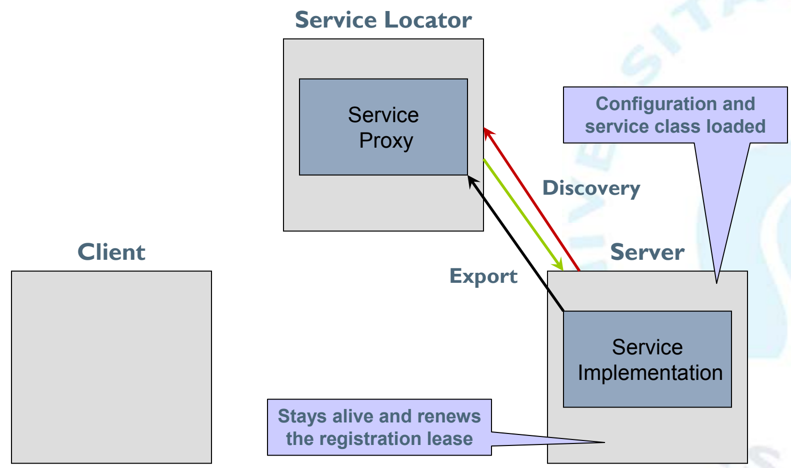
Client / Server approach



The Server

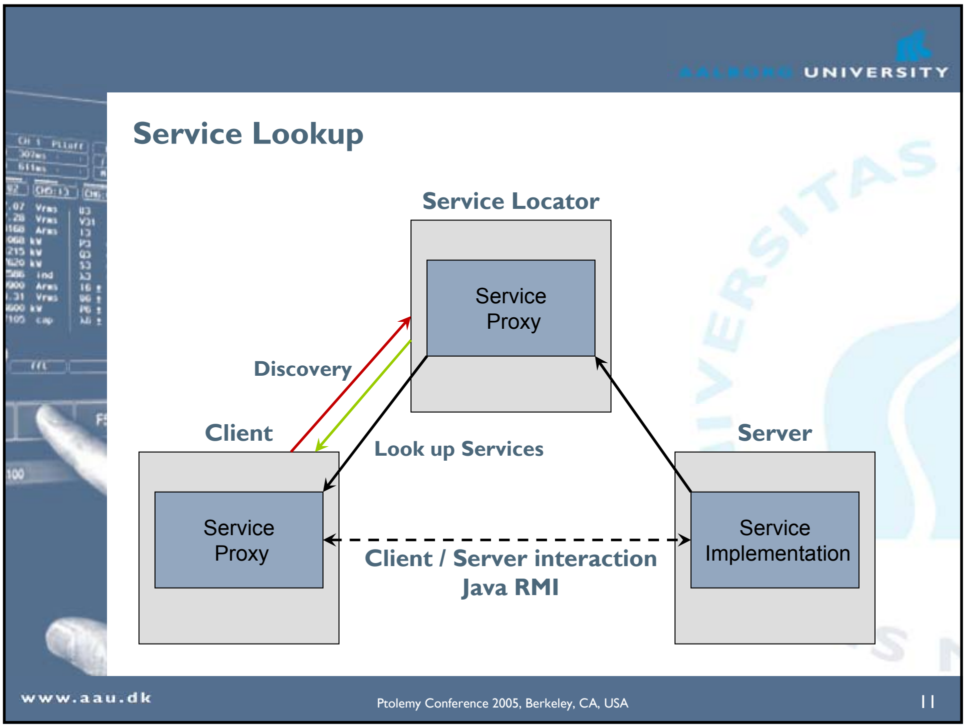
- **Prepares for discovery of a service locator**
  - Loads various settings as for example:
    - Unicast locators (predefined location where to search for a service locator)
    - The service class → DistributedActorWrapper.class (class that provides the distributed service)
- **Discovers a Service 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.

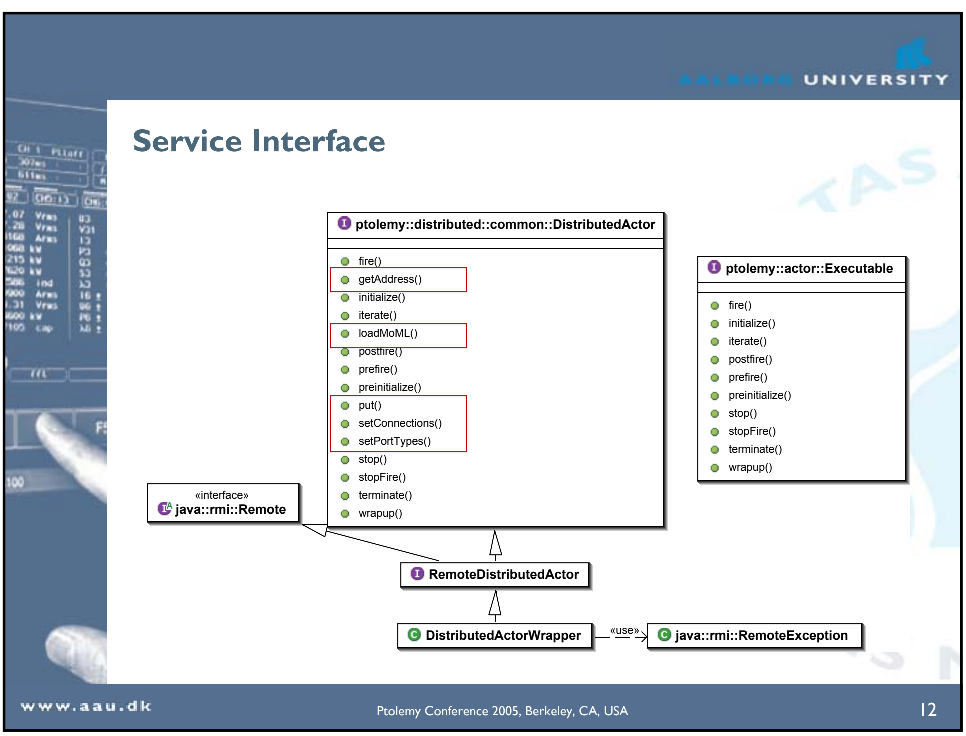
Server (Discovery and Service Registration)



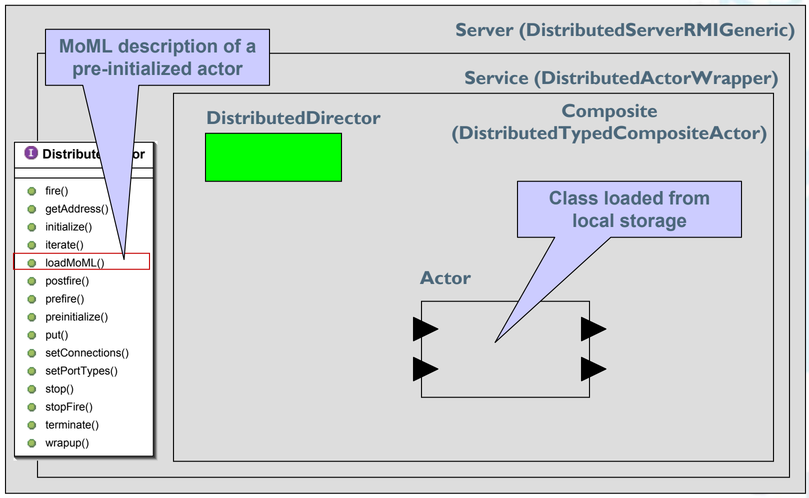
The Client

- **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)**

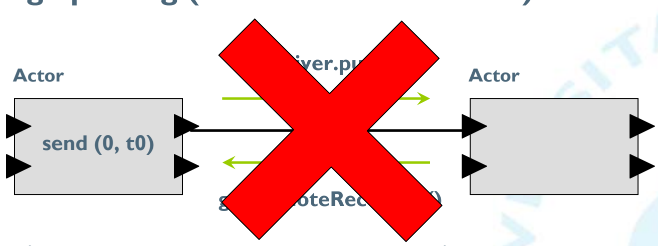

UNIVERSITY
Service Lookup
Service Locator
Service
Proxy
Discovery
Client
Look up Services
Server
Service
Proxy
Client / Server interaction
Java RMI
Service
Implementation
www.aau.dk
Ptolemy Conference 2005, Berkeley, CA, USA
11


UNIVERSITY
Service Interface
ptolemy::distributed::common::DistributedActor
fire()
getAddress()
initialize()
iterate()
loadMoML()
postfire()
prefire()
preinitialize()
put()
setConnections()
setPortTypes()
stop()
stopFire()
terminate()
wrapup()
ptolemy::actor::Executable
fire()
initialize()
iterate()
postfire()
prefire()
preinitialize()
stop()
stopFire()
terminate()
wrapup()
«interface»
java::rmi::Remote
RemoteDistributedActor
DistributedActorWrapper
«use»
java::rmi::RemoteException
www.aau.dk
Ptolemy Conference 2005, Berkeley, CA, USA
12

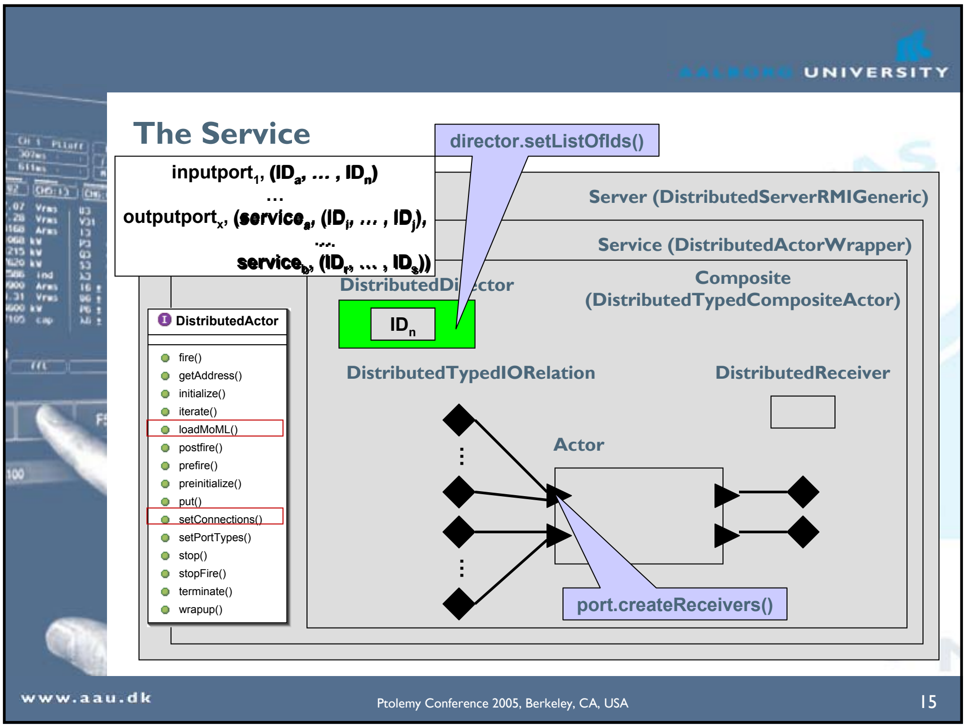
Server and Service

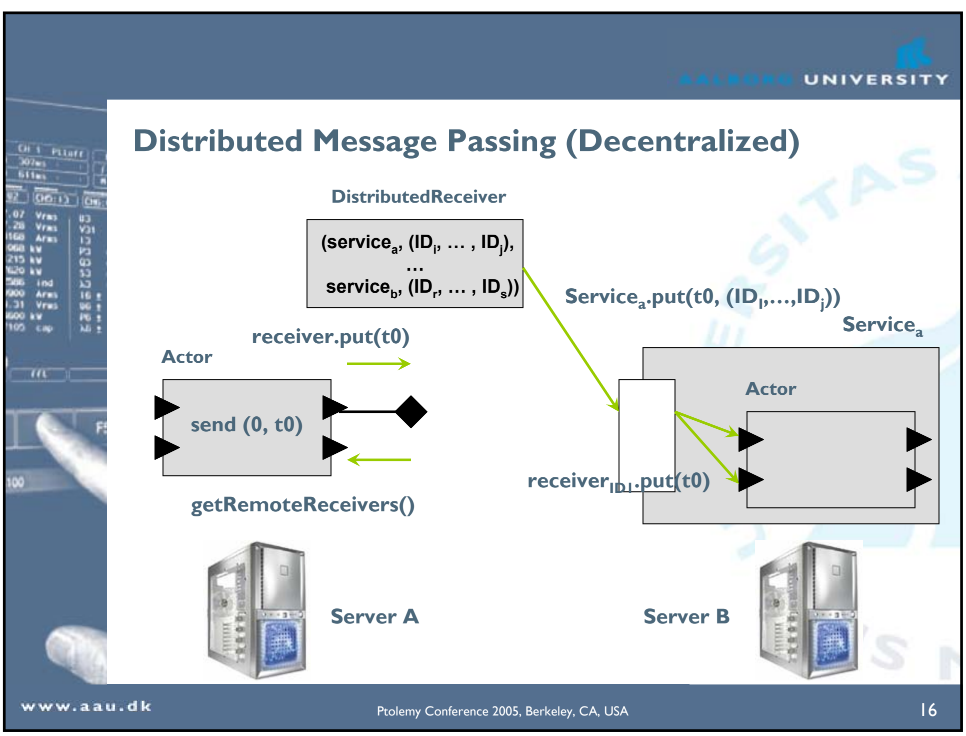


Message passing (Distributed Receivers)



- **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

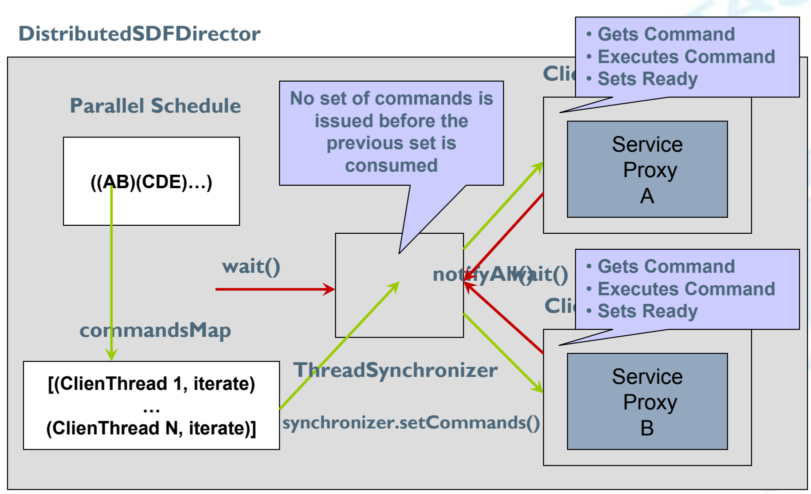

AALBORG UNIVERSITY
The Service
director.setListOfIds()
inputport1, (IDa, … , IDn)
…
outputportx, (servicea, (IDi, … , IDj),
…
serviceb, (IDr, … , IDs))
Server (DistributedServerRMIGeneric)
Service (DistributedActorWrapper)
DistributedDirector
Composite
(DistributedTypedCompositeActor)
IDn
DistributedActor
fire()
getAddress()
initialize()
iterate()
loadMoML()
postfire()
prefire()
preinitialize()
put()
setConnections()
setPortTypes()
stop()
stopFire()
terminate()
wrapup()
DistributedTypedIORelation
DistributedReceiver
Actor
port.createReceivers()
www.aau.dk
Ptolemy Conference 2005, Berkeley, CA, USA
15


AALBORG UNIVERSITY
Distributed Message Passing (Decentralized)
DistributedReceiver
(servicea, (IDi, … , IDj),
…
serviceb, (IDr, … , IDs))
Servicea.put(t0, (IDi,…,IDj))
Servicea
receiver.put(t0)
Actor
send (0, t0)
Actor
receiverID1.put(t0)
getRemoteReceivers()
Server A
Server B
www.aau.dk
Ptolemy Conference 2005, Berkeley, CA, USA
16

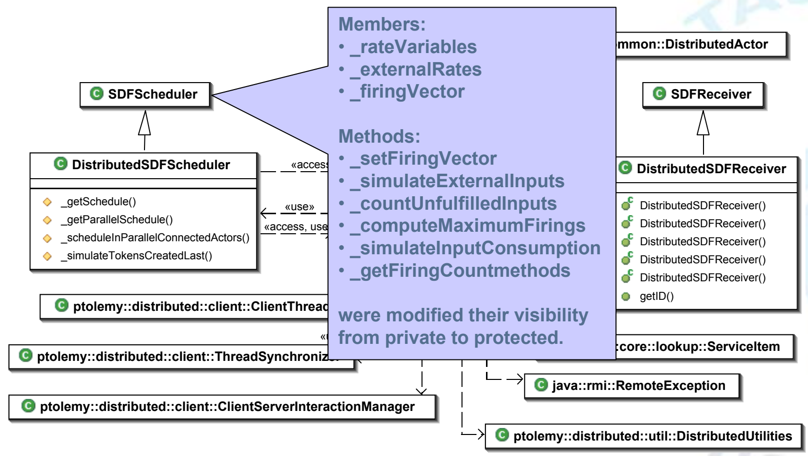
Issuing commands in parallel and synchronization (Client)

- 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

Issuing commands in parallel and synchronization



Client's Software Architecture



Software Packages

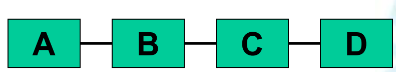
- **ptolemy.distributed.actor**
  - DistributedDirector, DistributedReceiver, DistributedTypedCompositeActor, DistributedTypedIORelation
- **ptolemy.distributed.actor.lib**
  - Library of distributed actors
- **ptolemy.distributed.client**
  - ClientServerInteractionManager, ClientThread, ThreadSynchronizer
- **ptolemy.distributed.common**
  - DistributedActor Interface
- **ptolemy.distributed.config**
  - Jini config files
- **ptolemy.distributed.rmi (Server classes)**
  - DistributedActorWrapper, DistributedServerRMIGeneric, 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



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