The GCF Computational Platform for DT Models: Proof of Concept

Steffen Fürst (Global Climate Forum)

May 17, 2020

1 The Platform

This text is supplementary material for computer code (included in the text) giving a proof of concept for the GCF computational platform for DT models. The platform offers a graph dynamical system framework for ABM simulations. The code and the present text are available at `https://globalclimateforum.org/GCF-Platform-for-DT-Models`. The proof-of-concept code is written in R, more elaborate versions with other languages will follow. Previous work on the concept can be found at `https://globalclimateforum.org/2018/09/26/`.

The agents on the platform can be of different types $\delta_a \in \Delta_A$, where Δ_A is the set of all agent types. For each agent type δ_a we have a different set of state variables $x_{\delta_a,1}, \ldots, x_{\delta_a,n}$, where each $x_{\delta_a,i}$ is element of a (potentially) different set of possible states $X_{\delta_a,i}$. So the state space for an agent of type δ_a is $\Theta_{\delta_a} = X_{\delta_a,1} \times \ldots \times X_{\delta_a,n}$.

The agents can be set into relation via a directed multi-graph, whereby the agents are represented by vertices. The edges in this graph can also be associated with states. We construct the state space for the edges in the same way as for the vertices/agents.

The overall state space of the system at time t is then $\Theta_t = \prod_{\delta_a \in \Delta_A} \Theta_{\delta_a}^{n_{\delta_a,t}} \times \prod_{\delta_e \in \Delta_E} \Theta_{\delta_e}^{n_{\delta_e,t}}$ where Δ_E is the set of all edge types, $n_{\delta_a,t}$ is the number of agents of type δ_a and $n_{\delta_e,t}$ is the number of edges of type δ_e in the system.

Let Ψ be the set of all agents. Each agent has a transition function (or a set of transition functions) that construct a set of agents and edges, so we have:

$$\begin{aligned} t_\psi \quad : \quad & \Theta_t \to \Theta_{\psi,t} \\ & \text{where } \Theta_{\psi,t} = \prod_{\delta_a \in \Delta_A} \Theta_{\delta_a}^{n_{\phi,\delta_a,t}} \times \prod_{\delta_e \in \Delta_E} \Theta_{\delta_e}^{n_{\phi,\delta_e,t}} \end{aligned} \tag{1}$$

where $n_{\phi,\delta,t}$ is the number of agents/edges that the agent ϕ has constructed in her transition function. We can then combine individual transition functions to a comprehensive transition function by concatenating the constructed elements.

$$t = (t_{\psi_1}, \ldots, t_{\psi_{n_t}}) : \Theta_t \to \Theta_{t+1} \tag{2}$$

So, in contrast to the usual object-oriented ABM frameworks, the state of objects is not modified by the transition function, but completely recreated by the transition function.

In the code presented here as proof of concept, the model state Θ is implemented in the form of an associative array with the types δ_a and δ_e as keys and the corresponding states of the type Θ_{δ_a} and Θ_{δ_e} respectively in the form of tables as values.

In addition to the model state, we usually need model parameters for a simulation and want to observe certain results, which however have no influence on the simulation itself. In this framework, the model parameters can be of any structure P and the observed results can be of any structure O (but in the example implementation both are tables only).

The framework expects that the following functions are defined:

- `initState`: $P \to \Theta_0$
- `initObserver`: $P \to O$
- `observe`: $(O, \Theta_t) \to O$
- and a set of transition functions.

1.1 Parallelism

The system described above should be distributed among different processing elements (PE) in such a way that each PE only receives a subset of the overall system state. Therefore, a transition function is always connected to a network formed by the edges of a type $\delta_e \in \delta_E$. For a single agent ψ only the state of agents $\hat{\psi}_i$ is available for which an edge $(\hat{\psi}_i, \psi)$ exists in the associated network of the transition function. Even a single agent does not know the complete graph, but only the edges where it is located at the tail or head.

Agents are assigned to processing elements (PE). Each PE does only know some part of the overall model state, namely:

- The state of the agents on this PE
- The edges of the graph, where an agent is on tail or head position
- While a transition function is called: The state of agents on the tail position of the network associated with the transition function

To calculate a transistion function, the following steps are taken:

- First we must ensure that for all edges of the network associated with the transition function, the state of agents on the tail position is known on the PE of the agents on the head position.
- Then the transition function can be called. We construct the new model state by gathering the returned lists of agents and edges.
- For the returned graph by agents on PE x we check which agents on the head or tail position are on another PE y, and send those edges to this PE.
- Since PE y may not know the PE of the agent on the other edge position (e.g. agent 1 on PE 1 knows agent 2 on PE 2 and agent 3 on PE 3 and adds an edge between these two. PE 2 may not know PE of agent 3), PE x sent this information also in form of a (agentID, PE) list.

1.2 Implementation

The following is an implementation of this approach, in which the parallelism is simulated by creating a separate list for each simulated PE. The lists are stored in the variable `modelStateDistributed`. The parameters of the transition functions only include a subset of `modelStateDistributed[[i]]`, this ensures that the agent PE does not access the "memory" of another PE.

We start with the helper function `mapAgentsToPEs`, that takes a non-distributed model state and creates a distributed version.

```
mapAgentsToPEs <- function(modelState, agentTypes, numPEs) {
  edgeTypes <- modelState %>% names %>% discard(. %in% agentTypes)

  ## Store which agent was moved to which PE
  PElist <- tibble(ID = character(),
                   PE = integer())

  ## Prepare the distributed data structures (the indices are the different PEs)
  modelStateDistributed <- vector("list", numPEs)
  PElists <- vector("list", numPEs)
```

```
  for (i in 1:numPEs) {
    PElists[[i]] <- tibble(ID = character(),
                           PE = integer())
  }

  ## Distribute the agents
  for (a in agentTypes) {
    ## We group the agents by their rowid, so that each PE gets the
    ## same number (plus/minus one) of agents
    modelState[[a]] %<>% rowid_to_column("PE") %>%
      mutate(PE = ((PE - 1) %% numPEs) + 1)
    PElist %<>% bind_rows(modelState[[a]]) %>% select(ID, PE)
    perPE <- modelState[[a]] %>% split(.$PE)

    for (i in 1:numPEs) {
      modelStateDistributed[[i]][[a]] <- perPE[[i]]
      PElists[[i]] %<>% bind_rows(modelStateDistributed[[i]][[a]]) %>%
        select(ID, PE)
    }
  }

  ## Distribute the edges
  for (e in edgeTypes) {
    if (nrow(modelState[[e]]) > 0) {
      for (i in 1:numPEs) {
        ## Find for each PE the edges where an agent of this PE is involved ..
        fID <- modelState[[e]] %>% filter(fromID %in% PElists[[i]]$ID)
        tID <- modelState[[e]] %>% filter(toID %in% PElists[[i]]$ID)
        ## ... and add those edges to the distributed state
        modelStateDistributed[[i]][[e]] <- bind_rows(fID, tID) %>% unique
        ## Also add the PEs of the agents on the other edge pos to the PElists
        PElists[[i]] %<>% bind_rows(fID %>% transmute(ID = toID) %>%
        ↪  left_join(PElist, by = "ID")) %>% unique
        PElists[[i]] %<>% bind_rows(tID %>% transmute(ID = fromID) %>%
        ↪  left_join(PElist, by = "ID")) %>% unique
      }
    }
  }

  list(modelStateDistributed, PElists)
}
```

The `enhanceAgentStates` function ensures that the state of agents at the tail position of the network associated with the next transition function is available on the PEs of the agents at the head position of the edges. This function is called before the transition function itself, which then gets the returned extendend model state.

```
enhanceAgentStates <- function(modelStateDistributed, PElists, agentTypes,
↪  edgeType) {
  modelStateEnhanced <- modelStateDistributed

  numPEs <- modelStateDistributed %>% length

  for (i in 1:numPEs) {
    ## We allow the creation of Networks while the simulation is running
    if (! is.null(modelStateDistributed[[i]][[edgeType]]) &&
        nrow(modelStateDistributed[[i]][[edgeType]]) > 0) {
      ## agentPEmap is a tibble(fromID, toID, PE of toID agent)
      agentPEmap <- modelStateDistributed[[i]][[edgeType]] %>%
        select(fromID, toID) %>%
        left_join(PElists[[i]], by = c("toID" = "ID")) %>%
        unique
```

```
      for (a in agentTypes) {
        ## iterate over the other PEs
        for (j in (1:numPEs %>% discard(. == i))) {
          ## find the agents that must be send to other PEs ...
          agentsIDtoSend <- agentPEmap %>% filter(PE == j) %>% select(fromID)
          agentsToSend <- agentsIDtoSend %>%
            inner_join(modelStateDistributed[[i]][[a]], by = c("fromID" = "ID"))
            ↪  %>%
            unique %>%
            rename(ID = fromID)
          ## ... and add them to the state of this other PE
          modelStateEnhanced[[j]][[a]] %<>% bind_rows(agentsToSend)
        }
      }
    }
  }

  modelStateEnhanced
}
```

`distributeNetworkState` takes the result of a transition function, and sends the edges to the PEs of the agents on the head or tail position.

```
distributeNetworkState <- function(modelStateDistributed, PElists, edgeTypes) {
  modelStateEnhanced <- modelStateDistributed

  numPEs <- modelStateDistributed %>% length
  ## create new empty lists
  newPElists <- PElists %>% map(~ .x[0,])

  for (i in 1:numPEs) {
    for (e in edgeTypes) {
      ## We allow the creation of Networks while the simulation is running
      if (! is.null(modelStateDistributed[[i]][[e]]) &&
          nrow(modelStateDistributed[[i]][[e]]) > 0) {
        ## Create two version of the edge tibble, ~matchFrom~ is extended with
        ## the PEs of the tail, ~matchTo~ with the PEs of the head agents
        matchFrom <- inner_join(modelStateDistributed[[i]][[e]],
                                PElists[[i]], by = c("fromID" = "ID"))
        matchTo <- inner_join(modelStateDistributed[[i]][[e]],
                              PElists[[i]], by = c("toID" = "ID"))
        ## sendNetworkTable is a tibble: fromID toID PE
        sendNetworkTable <- bind_rows(matchFrom, matchTo)
        ## sendPElist is a tibble: ID, PE, toPE
        sendPElist <- bind_rows(sendNetworkTable %>% select(ID = fromID,
                                                            toPE = PE),
                                sendNetworkTable %>% select(ID = toID,
                                                            toPE = PE)) %>%
          inner_join(PElists[[i]], by = "ID")

        ## Use the send... tibbles to distribute the edges to the PEs
        ## and also to send the (agentID, PE) information to those PEs
        for (j in 1:numPEs) {
          if (i != j) {
            modelStateEnhanced[[j]][[e]] %<>%
              bind_rows(sendNetworkTable %>% filter(PE == j) %>% select(-PE))
          }
          newPElists[[j]] %<>%
            bind_rows(sendPElist %>% filter(toPE == j) %>% select(-toPE)) %>%
            unique
        }
      }
    }
  }
```

```
  list(modelStateEnhanced, newPElists)
}
```

`simulate` combines all these pieces. `transform` 1 takes one agent type and iterates over the agents of this type, calling for each agent one of the transition functions. Inside of `transform` we use the `limitState` 2 function to ensure, that only the state of agents in the tail position of the associated network is accessable.

`consolidate` 3 gets the collected output of `transform` from all agents in the system, so `newModelState` is a list (with the agent types as keys) of lists (with the agents as keys) of tables (the tables returned by a transition function), and reduces this to a single table per agent type, our new state Θ_{t+1}.

```
simulate <- function(agentTypes, modelParams,
                     stateInitFunc, observerInitFunc, observerFunc,
                     tFuncs) {
  ## the edgeTypes are the associated network of the transition functions
  edgeTypes <- tFuncs %>% map(~ .x[[2]])

  numPEs <- modelParams$numPEs

  transform <- function(type, agents, modelState, tFunc, edgeType) { # 1
    limitState <- function(agent) { # 2
      incoming <- modelState[[edgeType]] %>% filter(toID == agent$ID)

      ## reduce the agent states to the agents on the networks tail position
      agentTables <- agentTypes %>%
        map(~ modelState[[.x]] %>% filter(ID %in% incoming$fromID)) %>%
        setNames(agentTypes)

      ## reduce the network to edges where the agent is on tail or head position
      edgeTables <- edgeTypes %>%
        map(~ modelState[[.x]] %>%
              (function(x) { if(is.null(x)  nrow(x) == 0) NULL else
                             { x %>% filter(agent$ID == fromID |
                                            agent$ID == toID)}})) %>%
        setNames(edgeTypes)

      c(agentTables, edgeTables)
    }

    agents %>% transpose %>% map(~ tFunc(as_tibble(.), type, limitState(.x)))
  }

  consolidate <- function(newModelState) { # 3
    flattenLists <- newModelState %>% flatten

    c(agentTypes, edgeTypes) %>%
      map(~ { flattenLists %>% map(.x) %>% bind_rows }) %>%
      setNames(c(agentTypes, edgeTypes))
  }

  ## create the initial distributed model state and PElist
  modelStateFull <- stateInitFunc(modelParams)
  distributed <- mapAgentsToPEs(modelStateFull, agentTypes, numPEs)
  modelStateDistributed <- distributed[[1]]
  PElist <- distributed[[2]]

  observerState <- observerInitFunc(modelParams)

  ## run the simulation
  for (. in 1:modelParams$numSteps) {
```

```
    for (s in tFuncs) {
      enhancedModelStateD <- enhanceAgentStates(modelStateDistributed, PElist,
      ↪  agentTypes, s[[2]])
      for (i in 1:numPEs) {
        modelStateDistributed[[i]] <- agentTypes %>%
          map(~ transform(.x, modelStateDistributed[[i]][[.x]],
          ↪  enhancedModelStateD[[i]], s[[1]], s[[2]])) %>%
           consolidate()
      }
      d <- modelStateDistributed %>% distributeNetworkState(PElist, edgeTypes)
      modelStateDistributed <- d[[1]]
      PElist <- d[[2]]
    }
    observerState <- observerFunc(observerState, modelStateDistributed)
  }

  observerState
}
```

2 A Model Example

2.1 Model description

We have a simple but volatile market with n buyers, m sellers, and two goods called x and y. Each seller sells both commodities, buyers buy both commodities from a single seller, but at each step the buyer randomly selects the seller from a fixed subset of all sellers. We are only interested in relative prices, so the price for good x is set to 1, the price for good y is p.

At each step, each buyer has a fixed budget B for buying the commodities. The quantities are choosen using a Cobb-Douglas utility function:

$$\begin{aligned} \max_{x,y} \quad & u(x,y) = x^{\alpha} \cdot y^{1-\alpha} \\ \text{s.t.:} \quad & x + y \cdot p \leq B \end{aligned} \tag{3}$$

The solution of this optimization is:

$$\begin{aligned} x &= B \cdot \alpha \\ y &= \frac{B \cdot (1-\alpha)}{p} \end{aligned} \tag{4}$$

For illustrative reasons, we neglect production except for the assumption that x and y are joint products, so that the seller tries to find a price where $x = y$. In the case of a single buyer, the solution is $p = \frac{1-\alpha}{\alpha}$. But in our case, the sellers are facing different buyers with different preferences α and a different budget B. So they start with a random price and adjust that price at each time step:

$$\begin{aligned} p_t &= \frac{d_y}{d_x} \cdot p_{t-1} \\ \text{where} \quad d_x &= \sum_{b \in \text{buyers}} x_b, \quad d_y = \sum_{b \in \text{buyers}} y_b \end{aligned} \tag{5}$$

It's easy to see that in the single buyer case, this will lead to $p = \frac{1-\alpha}{\alpha}$ after a single step.

2.2 Implementation

We have two networks, one is fixed and describes the sellers known to a single buyer. Since the direction of an edge determines the flow of information and the buyer needs the price information to calculate his demand, it is a network from sellers to buyers. We call this network `KNOWN_SELLER`.

The second network is constructed at each step in the function `calcDemand`, this network is called `BOUGHT_FROM` and describes from which seller a buyer bought something, so it will go from the buyers to the sellers.

As the `BOUGHT_FROM` networks will be constructed in the simulation itself, we do not need to worry about it in the `stateInitFunc`.

2.3 Init Functions

```
initAgents <- function(modelParams) {
  nb <- modelParams$numBuyer
  ns <- modelParams$numSeller
  list(
    "BUYER" = tibble(
      alpha = runif(nb),
      B = runif(nb, max = 100) %>% ceiling,
      x = 0,
      y = 0,
      ID = paste0("BUYER", 1:nb)
    ),
    "SELLER" = tibble(
      p = runif(ns, min = 0.5, max = 1.5),
      ID = paste0("SELLER", 1:ns)
    )
  )
}

initNetwork <- function(modelParams, modelState) {
  sampleSellers <- function() {
    sample_n(modelState$SELLER, modelParams$knownSellers) %>% pull(ID)
  }

  list("KNOWN_SELLERS" =
         modelState$BUYER$ID %>%
         map(~ tibble(fromID = sampleSellers(),
                      toID = .x)) %>%
         bind_rows)
}

initState <- function(modelParams) {
  agentState <- initAgents(modelParams)
  c(agentState, initNetwork(modelParams, agentState))
}

initObserver <- function(modelParams) {
  tibble(
    sum_x = numeric(),
    sum_y = numeric()
  )
}
```

The `BOUGHT_FROM` network will be available after calcDemand, because every buyer returns an edge to the seller from whom he bought the goods. The `KNOWN_SELLER` network is constant, but must be recreated in this proof of concept through each transition function. However, this is easily done by making each agent return all edges where itself is at the head. The sum of these edges then results in the `KNOWN_SELLER` network.

The seller does nothing else but return its state. Otherwise the model state after calling the function `calcDemand` would be without any seller.

```
calcDemand <- function(agent, type, modelState) {
  seller <- modelState$SELLER %>% sample_n(1)
  switch(type,
```

```
          "BUYER" = {
            agent$x <- agent$B * agent$alpha
            agent$y <- agent$B * (1 - agent$alpha) / seller$p
            list("BUYER" = agent,
                 "BOUGHT_FROM" = tibble(fromID = agent$ID, toID = seller$ID),
                 "KNOWN_SELLERS" = modelState$KNOWN_SELLERS %>% filter(toID ==
                 ↪  agent$ID))
          },
          "SELLER" = {
            list("SELLER" = agent)
          })
}
```

In the transition function `calcPrice`, the network `KNOWN_SELLER` can be treated as in `calcDemand`, and the network `BOUGHT_FROM` can be ignored, since it is rebuilt in the next `calcDemand` call.

```
calcPrice <- function(agent, type, modelState) {
  quant <- modelState$BUYER %>% select(x, y) %>% summarise_all(sum)
  switch(type,
          "BUYER" =
            list("BUYER" = agent,
                 "KNOWN_SELLERS" = modelState$KNOWN_SELLERS %>% filter(toID ==
                 ↪  agent$ID)
                ),
          "SELLER" = {
            agent$p <- if (quant$x > 0) { quant$y / quant$x * agent$p } else {
            ↪  agent$p }
            list("SELLER" = agent)
          }
          )
}
```

The framework does the parallelization mostly automatically, it is not necessary to keep the parallelization in mind while writing the transition functions or the initialization of the model state remains the same.

The observer on a PE, however, can only observe the part of the model state which is exactly on this PE, which must be considered by the modeller. The framework to be developed, however, will provide functions that support him, similar to the following sumD function.

```
observeDistributed <- function(observerState, modelStateDistributed) {
  sumD <- function(type, var) {
    modelStateDistributed %>% map(type) %>% map(var) %>% unlist %>% sum
  }

  observerState %>% add_row(
                  sum_x = sumD("BUYER", "x"),
                  sum_y = sumD("BUYER", "y")
                )
}
```

Finally, here is a run of this example model. The result of the simulated function is a table with the total demand for x and y for each time step. We plot a time series of the difference of this demand, which in equilibrium should be exactly 0.

```
runExampleModel <- function() {
  simulate(agentTypes = c("BUYER", "SELLER"),
           modelParams = tibble(numBuyer = 20, numSeller = 5, knownSellers = 2,
                                numSteps = 20, numPEs = 5),
           stateInitFunc = initState,
           observerInitFunc = initObserver,
           observerFunc = observeDistributed,
           tFuncs = list(c(calcDemand, "KNOWN_SELLERS"),
```

```
                            c(calcPrice, "BOUGHT_FROM"))
            ) %>%
    rowid_to_column("step") %>%
    ggplot(aes(x = step, y = sum_x - sum_y)) + geom_line()
}

runExampleModel()
```

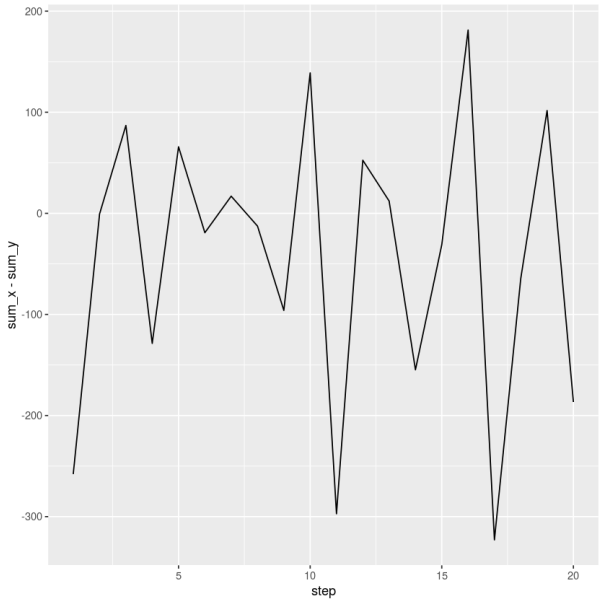

Figure 1: A volatile market with joint production