Master's thesis

Simulating dataflow for Hardware/Software Co-Design in the context of SKA

By: Sulaiman Mohammad
Supervisor: Prof.Dr. Martin Quinson
Supervisor: Dr. Frédéric Suter

University of Perpignan Via Domitia
2022

Host organization: École normale supérieure de Rennes
Internship done at IRISA Laboratory rennes

List of Figures

Contents

Chapter 1

Abstract

Dataflow diagram is a model that executes the sequence of applications, where each application will be executed when the application's input data is available. Dataflow is a suitable model to be implemented in data-intensive applications with streaming input and computational-intensive applications. SKA pipeline is an adequate example of those applications. To evaluate the influence of dataflow in the context of SKA a hardware-software co-design is used. Hardware-software co-design is achieved by taking the astronomical algorithms and building the sequence of those algorithms as a pipeline using dataflow followed by evaluating the performance of the pipeline on a specific platform. Optimization process is applied to the pipeline for better performance, followed by modifying the platform for higher optimization. This methodology is more effective by using a simulation of the dataflow model and a simulation of a platform. This work introduces a simulation of the dataflow model with a reasonable level of automation for the DALiuGE framework utilized in the SKA project. The simulation presented in this work implemented the idea of using three channels to represent the edge between two actors of the dataflow diagram, one channel for the state of the actor as an implementation of the data-activated concept. The second channel contains the data that should be transmitted to the successors. The last channel is called Backward-state as a method to simulate streaming input and Producer-consumer streaming.

Chapter 2

Introduction

The Square Kilometer Array (SKA) will be the largest radio telescope ever built. This project is currently under construction in two phases, SKA1 and SKA2, to form ultimately a square kilometer of collecting area in South Africa and Australia. SKA will operate over a wide range of frequencies to eventually provide the highest sensitivity that ever existed with a resolution beyond the Hubble Space Telescope [20]. A million low-frequency antennas and thousands of mid-frequency dishes will generate 7.2 Tb/s and 8.8 Tb/s of data, respectively. This enormous amount of data must be processed in near-real time through different stages, where each stage is a process that performs mathematical calculations and delivers the output to another process, maybe residing on another node to enable parallel computing, which means faster computation. However, multiple software and hardware constraints stand in the way of this strategy.

In the hardware domain, there are two platforms used in SKA. Firstly, the Central Signal Processor (CSP) performs correlation and beam forming operations on the data from antennas on-site. Those operations are done in real-time, eventually providing what is called raw visibility data. Based on the fact that the CSP is on-site equipment, the risks of electromagnetic interference with the observed signal must be taken into account to avoid any distortion of the signal of the sky. The second element is the Science Data Processor (SDP) which represents the computing platform and the astronomical algorithms that perform operations on the raw visibility data to produce the final image. In view of the long-term execution of the SDP with data and computational intensity, energy consumption should be managed while maintaining performance.

In the software domain, signal processing is done as a pipeline, where the algorithms are executed in a consecutive manner with dependencies, which can be represented by a graph such as a workflow or a dataflow. Since SKA is an international project, thus the algorithms that compose the pipeline are collectively designed throughout the world by specialists, regardless of the hardware. Next, the pipeline graph is constructed and executed on a potential platform, so decoupled algorithm designs are a prerequisite. Another constraint relates to the existing astronomical algorithms that are sequential, and optimizing them to execute in parallel requires a very careful procedure. Furthermore, there is a portability issue, for example, FFT data alignment differs across architectures [2]. Noticeably, there is a relation between hardware and software. Co-design can exploit this correlation to minimize the limitations in hardware and software domains by maximizing the interplay between the platform and the algorithms.

To explore the hardware-software co-design space and evaluate the utmost possible design candidates, there are several classical methods, two of which will be discussed here.

- Direct experiment by executing an application on a particular architecture, yet for large-scale applications on a considerable platform, it is labor-intensive in general. In addition to that, the platform design should change in hardware-software co-design, but it is impossible to change a considerable platform repeatedly due to time-cost, expenses, engineering constraints, and difficulties of control experiments for several scenarios on a large-scale platform.

- Simulation is an appealing alternative that solves most of the flaws of the direct experiment, but it also has its own drawbacks. For instance, in an abstract simulation, each scheme has it its own simulation. Since the hardware and software are changeable in co-design, a new simulation for each change is needed, and it is mandatory to explore the space of co-design, but it is cumbersome. On the other hand, a highly automated global simulation that could cover all applications in a specific context is unobtainable.

Consequently, a simulation requires a formalism to define the structure and the properties of what will be simulated; in the context of SKA DALuiGE and DASK are two recognized candidates that design and execute the pipelines. DALuiGE was chosen due to the ease of extracting data that represent the elements of the pipeline; therefore, greater automation could be achieved. Also, not all platforms can be taken into consideration because there are minimum requirements. Accordingly, a cluster of heterogeneous nodes is accommodated to the goal as it is envisioned by the SKA. This work introduces a realizable solution for an automatic simulation of the DALiuGE framework that implements the pipelines in SKA, this simulation can show the performance of any pipeline designed on DALiuGE on any platform.

The co-design lifecycle will be accomplished through simulation as shown in Fig. 2.1; first, platform designers evaluate the execution of the algorithm on it and then modify the platform to reach better performance. That platform design is sent to pipeline designers to optimize the algorithms again based on the new version of the platform.

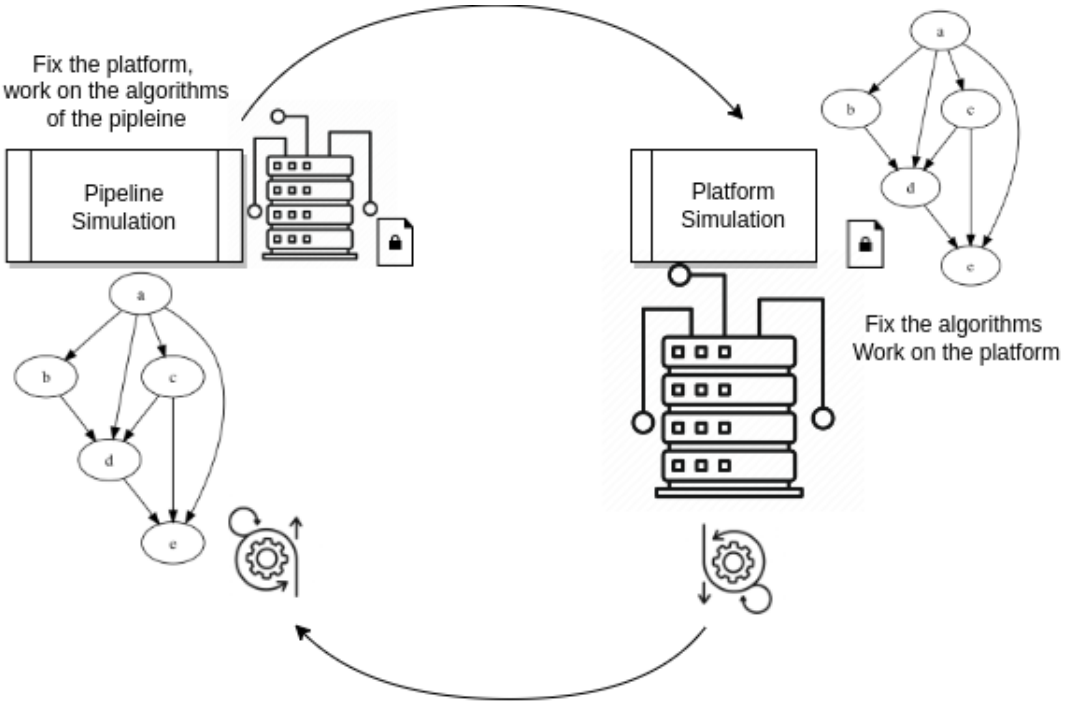

Figure 2.1: Co-design lifecycle

Integration between these two stages requires automatic platform simulation and automatic pipeline simulation for the software, which is the purpose of this work under Dark-Era project that aims to assemble a final version of a co-design tool. Our objectives in this work that help to achieve this tool are the following:

- We propose a method to simulate dataflow, particularly those expressed in the DALiuGE formalism.
- We pursue to obtain the highest possible level of automation that leads to ease of use, which helps in the design loop.
- We test the simulation to verify its accuracy and efficiency of the simulation so that it can be used in the co-design of hardware and software.

Chapter 3

Background

This chapter introduces in the first section introduces the models used to represent pipelines such as workflow and data flow, and the difference between them. The second section presents the frameworks that applied those approaches in the context of SKA.

3.1 Methods to implement a pipeline

3.1.1 Workflow

Scientific applications can consist of many tasks connected as a pipeline, where the output of each task is the input of the successor. Executing a pipeline on a platform requires an expression, called workflow, that describes the sequence of transforming input data to the final output through a collection of computational tasks interconnected together. Workflow can be represented as a graph-directed acyclic graph, where each vertex in the graph is a computational task and the edges denote data or control signals.

A workflow can be constructed using a drag-and-drop interface such as Kepler [1] to express the execution sequence. Another method to describe the workflow can be textually, as with Pegasus [10]. A workflow with a large set of tasks requires software that can orchestrate the execution of the tasks and manage to schedule them on computing resources. This software is called a Workflow Management System (WMS). WMSs are centralized, which means that the tasks of the workflow are governed by this coordinator. With all the success of workflow, it can not fit all the applications. For instance, in a data-intensive application, if an issue occurs with WMS, the communication between the tasks, or in the databases, the execution of the workflow will stop. Furthermore, a failure in one of the tasks will stop the whole workflow, and it can be difficult to diagnose failed tasks in the case of workflow with a million tasks such as SKA applications. Applications with data streaming require a concept different from the workflow. Since the workflow is executed once for a certain input, it needs to be launched again for each new data set. Tasks cannot be rescheduled after pipeline execution began in case certain conditions, such as new data, are met during pipeline execution [4]. This can be a problem for applications with a data stream that needs to be processed in near-real time, as is the case in the SDP. The solution for such applications is the dataflow model.

3.1.2 Dataflow

The dataflow model introduced by Dennis [11] is a model similar to workflow, where each vertex is an individual object called an actor that represents the process that deals with the received data. The dependencies between actors should be fulfilled by using communication channels (edges), as in Fig. 3.1.

In Fig. 3.1 stage 1,2 represent the methods that process the data, and they can be hardware or software. It is noteworthy that hardware has parallel properties such as using FPGA to build the dataflow model, where each actor can be built as a self-contained circuit, but it is unique to each actor operations. Whereas if the actors are software, then the diagram must be executed sequentially [16], or additional work is needed to parallelize it due to the dependencies, by using for example graph partitioning algorithms.

The main difference between workflow and dataflow is that actors in dataflow continually check their input, and once data are available, the actors will be performed and the outputs produced. Accordingly, the dataflow can be implemented for streaming applications, and actors are activated by the data availability in a decentralized manner with no need for a central orchestration tool to organize the actors' executions.

Dataflow is used widely in different domains. For instance, data flow is applied effectively to process huge amounts of data effectively in the Google Cloud. Also, in signal processing applications where the input data usually comes from a stream of discrete samples that are processed using multiple stages as a pipeline. The output size of a stage is not necessarily equal to the input size required for the next stage, as shown in Fig. 3.1 between stage 1 and stage 2. Dataflow can deal with these disparities in the input sizes effectively. SKA's pipeline is a valid example of the significance of utilizing dataflow in signal processing, where the input will be a stream of raw visibility data that consecutive astronomical algorithms will perform mathematical operations on it.

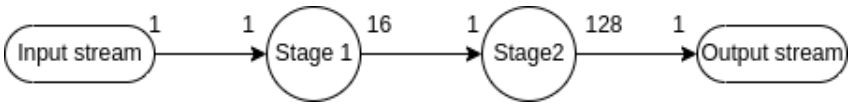

Figure 3.1: Dataflow diagram

The building blocks of a dataflow are actors, edges and tokens as shown in Fig. 3.2. **Actor** is the element that performs operations on the input to produce output $y = f(a, b)$, where f called firing function, and (a, b) are the input.

For $a = [a_1, a_2.....a_n]; n \in \mathbb{N}$ each component of this sequence is an object called **Token**. The tokens in a queue move on a unidirectional link that connects actors called **Edge** and follow the First-In-First-Out (FIFO) method. Attaching a set of edges E to actors V forms the pipeline that represents the sequence of execution as a graph $G = \{V, E\}$. To represent mathematically the relation between actors and edge, let $e \in E$ an edge that connects actors $v_1, v_2 \in V$, then v_1 is the producer actor when it is the source of the tokens on one side of the edge $v_1 = src(e)$, and v_2 is the consumer actor when the actor is on the other side of the edge and consumes the produced tokens $v_2 = snk(e)$. Each actor has two variables, the number of produced tokens by v_1 is $M = prod(v_1)$, and the number of consumed tokens by the actor v_2 is $N = cons(v_2)$.

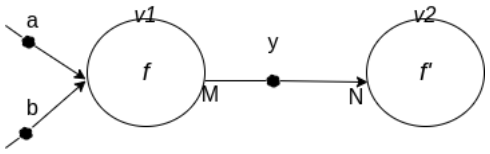

Figure 3.2: Dataflow elements

The execution of the dataflow respects the firing rule that states that actor v is invocated if there is available data in the input and the number of input tokens is equal to or bigger than $cons(v)$ for all incoming edges. Taking v_1, v_2 in Fig. 3.2, if $M < N$ which mean that v_2 will receive fewer tokens than what it is needed to activated it. As a result of that, a buffer should store the tokens produced by v_1 until it is enough to activate v_2, in this scenario v_1 will be executed several times before v_1. If $Mgt; N$ the extra tokens will be stored in the buffer and v_2 will be executed more times than v_2

Adding constraints to the produced/consumed tokens by actors vary the types of Dataflow such as Synchronous Dataflow (SDF) is restricted by the token's production and consumption per actor, and it should be constant for each firing. This constant number is known ahead in SDF. Notice, that Synchronous does not represent the mode of communication with a clock, because sending data is always asynchronous [13]. Synchronous means the prior knowledge of the production and consumption of tokens for all actors. This prior knowledge eases the scheduling and the size of the buffers needed, so it cannot handle dynamic data rates.

If another constraint is added to SDF such as token production equal to consumption equal to unity tokens, SDF then is called Homogeneous Synchronous Dataflow (HSDF) [17]. HDSF is used with the data-activated flow, where the control signal is used to trigger the actor, but that reacquires the existence of a storage actor that receives the data between two actors and uses a control signal also after the needed data is stored completed, to trigger the successor. An example of an application that uses this dataflow is the DALiuGE framework in SKA.

3.2 Graph execution frameworks in the context of SKA

3.2.1 DASK

Dask is a library implemented in Python for parallel computing for big data applications. The block algorithm is used in Dask, the idea of this algorithm is to take a complex problem and break it down into smaller steps. The huge data set will also be divided into small sets, then mathematical operations are performed on each of them. This approach optimizes the execution of big data applications using an out-of-core algorithm. For example, assuming an application has data that cannot be fitted in memory at once, it is better to use the disk as an extension of the main memory. In this way, splitting the data into chunks to be loaded in memory, when it is necessitated. Thereafter, parallel computations are obtained by combining a dynamic and memory-aware task scheduling

combined with the block algorithm [15], where the data is divided into small chunks and distributed into different computational nodes and the task that will perform the computations are scheduled by the memory-aware scheduler.

Dask generates a Directed Acyclic Graph (DAG) to express the arrangement of the execution, then the scheduler will use the produced DAG to schedule the tasks. The main issue here is that the graph is not created by the user; instead, it is derived from the main source code that represents the whole pipeline. Describing the whole pipeline in one code is not efficient, because for a complicated application tracking the automatically generated graph will be very challenging. Another reason is the ineffectiveness of automatic simulation, due to the difficulty of extracting the DAG parameters from the high-level organizational structure of Dask.

3.2.2 DALiuGE

Data Activated Liu Graph Execution Engine (DALiuGE) is a framework that implements dataflow by adding data storage and control flow to the standard dataflow model. This approach is then called the Data-Driven Processing Environment, which stands for the propriety, where each data triggers the following process to define the execution path [2]. Therefore, each task will drive its execution based on the states of the predecessors rather than counting on a central orchestrating entity, in this case, the system is decentralized, allowing for a scalable execution engine [21].

DALiuGE consists of four types of graphs. The first one is the *logical graph template* as shown Fig. 3.3, which is a resource-oblivious graph, where the dependencies between tasks are represented regardless of the resources. Thus, algorithm designers can work collectively on different tasks from anywhere in the world, because the design doesn't depend on the platform, and since SAK is an international project, there is a need for this level of decoupling between designing and implementing the graph. The second graph is the *logical graph* which is the *logical graph template* after adding parameters to it, such as time of execution and the source of data. The graphs in DALiuGE are made up of two components called drops:

- Application-Drop: represents the computational tasks as Docker images, binary executable, Python, shell, and C/C++, allowing the use of existing codes.
- Data-Drop: storage methods such as in-Memory, File, Apache Arrow Plasma, S3, and NGAS.

Application Drops are classified as consumer/producers; those drops are connected through Data Drops that represent the intermediate phase between consumer and producer. Fig. 3.3 shows *CopyApp* consumes the data generated by producer *RandomArrayApp* through *Memory*. The transformation to the third graph is done by a translator responsible for partitioning the graph using multilevel K-way partitioning, forming *Physical Graph Template* as shown in Fig. 3.4, then assigning each Drop to the computation node to form the *Physical Graph*.

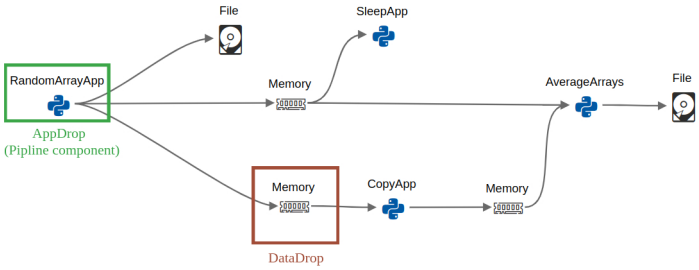

Figure 3.3: DALiuGE Logical graph

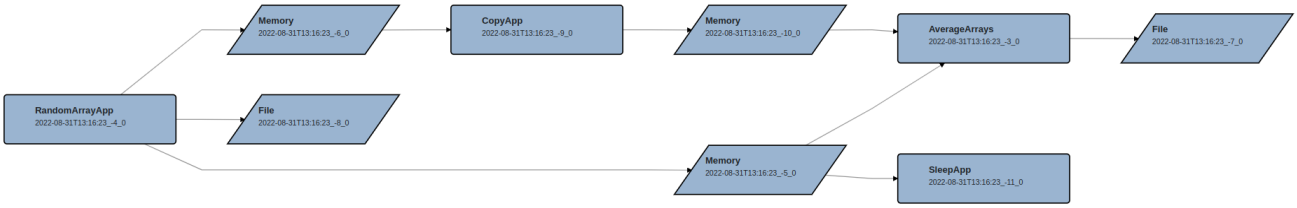

Figure 3.4: DALiuGE Physical graph

Drops in the graph have states, while the drops contain data as payloads, the payload itself is stateless. The state will be changed to *Complete* when a drop finished its functionality, such as termination of the execution for Application Drop and receiving all the data for Data Drop. There are different states that express the drops such as *Running* which means that the drop is still executing, also *Expired* that represents the end of the drop's lifecycle, and the drop will not be accessible after this state. During the execution, drops notify the successors drops after each change in the state to drive the execution sequence.

Once an Application Drop receives *Complete* state from the predecessor Data Drop that will enable data production of the Application Drop if the drop is a producer. If Application Drop is a consumer, then once it receives *Complete* it will consume data from Data Drop by reading the available data in the Data Drop. Data Drop allows reading its content by the successor after receiving *Complete* from all predecessors, unless it is not in the *Expired* state. This was batch-style.

There is another mode, which is streaming mode, that meets the SKA criteria related to the data stream, where the Application Drop has an input stream and the Data Drop, which connect between the producer and the consumer has an output stream. The Data Drop in this mode will notify the consumer after the Application Drop producer writes a block of bytes, so the consumer can read from the Data Drop without the need to wait until the producer finishes execution completely, although when the producer finishes it will also send *Complete* state to the Data Drop which will forward it to the consumer too [21, 9].

Chapter 4

State-of-the-art

4.1 SimGrid

In this work, the simulation is implemented using SimGrid [7], which is a framework that simulates the performance of distributed applications on distributed platforms. SimGrid is a suitable simulator for our objective. The hardware-software co-design must evaluate an application on a particular platform, and SimGrid provides accurate simulation that involves the platform configurations and the system designs. SimGrid has proven its ability to provide accuracy and scalability after using it in different domains. SimGrid is composed of actors that contain all the instructions and the computational functions that SimGrid simulates on the platform. The actors also contain the communications procedures. SimGrid has a file that describes the computation platform that provides information about the resources. The platform file contains all the data about the hosts (computational nodes) and the connection between them, including all specifications of the host, from the amount of FLOPS to the bandwidth of the links between actors.

4.2 WRENCH

WRENCH is an accurate and scalable simulation framework [6] for Workflow Management Systems(WMSs). WRENCH is built on SimGrid which delivers those properties. However, SimGrid's low-level simulation abstractions make it difficult to produce simulations for complex systems. WRENCH eases the development of the simulation while maintaining accuracy and scalability. WRENCH provides the ability to simulate the execution of a workflow on any platform when the workflow is coordinated by a specific WMS, as well as simulate a WMS to evaluate its behavior in multiple instances. WRENCH targets workflows and WMS. Recalling the differences between dataflow and workflow, one of them, in dataflow, there is no need for WMS, thus WRENCH can not be used to simulate dataflow. Since WRENCH has strengths points coming from using SimGrid, it is reasonable to use SimGrid also to simulate dataflow. In SimGrid, it is possible to simulate the workflow using the low-level implementation using SimDAG. Furthermore, SimGrid provides plugins to simulate consumer-producer relations. Whereas, the idea of activating an actor based on the data availability besides the streaming doesn't exist on its own as a high-level implementation.

4.3 Telescope Operations Simulator(TopSim)

TopSim is a workflow simulator in the context of SKA [18]. TopSim's objective is to simulate a workflow of telescope observations and data-archival and test the scheduling techniques of the workflow. TopSim uses ScHeduling Algorithms for DAG-Workflows (SHADOW) that use scheduling heuristics [5]. In this approach, the workflows are described as a JSON file or a Pegasus DAX file, and the platform is also described to simulate the execution of the workflow on the provided platform. This simulator have not proven its accuracy and scalability yet. Since it is a workflow simulation, it did not take into consideration multiple WMS as WRENCH accomplishes. There is also, for example, Step Functions Data Flow Simulator [19] from AWS, but It does not take into account all the characteristics of SDF, and it is a unitary simulation to evaluate only one AWS task each time. As we can see above, an abstract high-level, and directly re-usable simulation for dataflow is not yet accomplished.

Chapter 5

Simulating DALiuGE

In the dataflow model, each actor is triggered and executed by the availability of the data that is expected as input of the actor. Achieving activation of an actor by the data availability using hardware equipment, but hardware design usually stands for a single specific application. To implement this concept in software requires control flow operations and data storage[3] so the data is staged in storage between actors and the actor can be driven by control flow that indicates the availability of data in the storage. This is done in DALiuGE by using the data-activated concept, where the state of the actor is used to create a control flow where the state of the actor activates the successor, besides data-drops that represent the data storage, and the computational actors are represented as App-drops. In DALiuGE there are two types of drops App-drops and data-drops, both can be introduced as actors in the dataflow model. Also, each graph needs a starting point that provides data for the rest of the drops.

Drops of DALiuGE are simulated as shown in Fig. 5.1. A similar approach is used in simulating microservices[8]. Each drop is an actor in the simulation. However, the actor in the simulation does not contain the operations that will be performed on real data. Instead, it comprises the CPUs usage of those operations that produce or consume the data, for example, to simulate an App-drop that performs FFT on data, the amount of the CPU usage that FFT needs in (flops/s), and it should be provided to the simulation. Also, the simulated drop contains all the communication methods of the input and the output. Those methods coordinate the sending and receiving of data based on the dependencies. Same for data-drops, there is no actual data, instead, it holds the size of the data that is involved in the communication between App-drops.

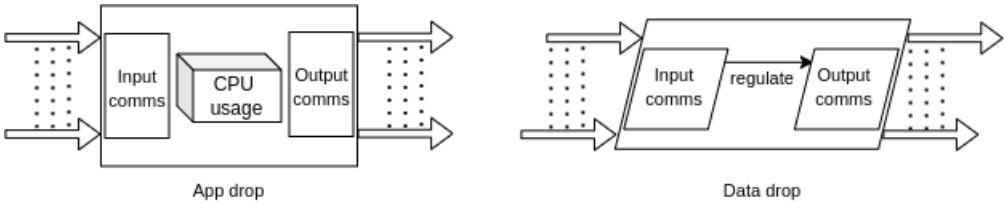

Figure 5.1: Drops representation in the simulation

Dependencies between drops are described by the edges that connect between actors. For a graph shown in Fig. 5.2 described as $G = \{V, E\}$, where $V = \{A, B, C\}$ and $e_i \in E$, such as the edges are $e1 = [A, B]$, $e2 = [A, C]$. Therefore, A has B and C as successors, and each edge that connects the actors contains the data tokens, data unity, that flow between actors. In DALiuGE, the actor will activate the successor using a control message of completion state that indicates that the data are available and can be read. In this manner, when A is completed, it will send its state to B,C to activate them. As mentioned earlier, data storage should be intermediate between actors. A will save its data in data storage, and the completion of the writing process will trigger B and C.

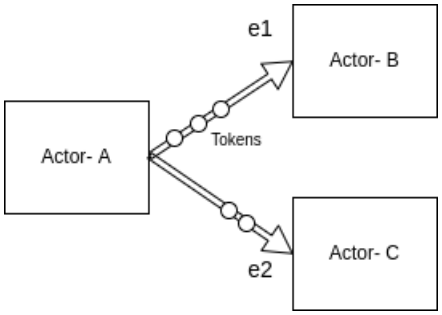

Figure 5.2: Dependencies between actors

To simulate the dependencies and the communication between data-drop and App-drop which is the only case in DALiuGE, where the data-drop always should intermediate between two App-drops. We suggest splitting each edge into three channels $ei = \{e_{i1}, e_{i2}, e_{i3}\}$ as shown in Fig. 5.3. A channel sends the state of the actor to the successor. Another channel that carries the data tokens. The last channel is a backward-state, where the current actor A receives the state of the successor B and C, this channel helps to achieve the streaming mode. The idea of the last channel backward-state is inspired by the signal processing system, where the feedback signal is provided to the streaming input to correct the output.

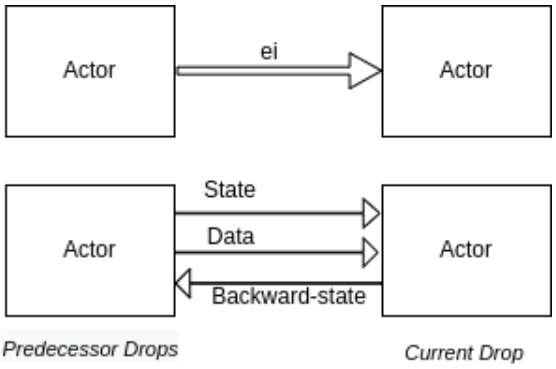

Figure 5.3: Dependency and the communication between actors

The concept of communication is between two App-drops through data-drop demonstrated in Fig. 5.4. All drops start with the state *Initialized*. The App-drop has the state *Completed* when the execution is done, while the data-drop has the state *Finished* after it receives all the data from the producer. There is a data-drop between each consumer and producer, for example, *(DataDrop-A)* intermediates *(AppDrop-Prod-A)* which is the producer and the consumer *(AppDrop-Prod-consA-prodB)* which is the producer also for the next drops.

The first App-drop *(AppDrop-Prod-A)* starts executing and producing data. Once it is terminated, it will change its state variable to *Completed*, and send it via the channel of state to the data-drop *(DataDrop-A)* in non-blocking communication mode, which means that actors will proceed and will not wait until the message is retrieved by the receiver. Subsequently, the produced data is transmitted over the data channel in non-blocking mode as well. Consequently, the *(AppDrop-Prod-A)* can be re-executed at the same time while the data is still being transmitted from the first stage, which reduces the idle time. *(DataDrop-A)* will receive the state of the predecessors, if it refers to the completion of the producer, then the *(DataDrop-A)* actor begins receiving data. In the framework that we use here for the simulation, its communication protocol itself also includes the time needed to write the data into memory. Otherwise, the time of I/O should be included in the data-drop actor as a parameter in the simulation, which depends on the size of the data. After finalizing the data received, the data-drop state becomes *Finished* and it is sent to the *(AppDrop-Prod-consA-prodB)* consumer notifying it that the data are available. The consumer can start receiving the data from data-drop and that mimics the reading data procedure from data-drop in DALiuGE, also the time of reading is included in the communication protocol. The moment that *(DataDrop-A)* accomplishes a stage, it can receive new data from the producer drop *(AppDrop-Prod-A)*. This can be achieved by utilizing the backward-state channel.

The data-drop *(DataDrop-A)* will dispatch a message of its state backwards to the producer drop, declaring that the previous stage is finished. Consequently, the predecessor sends the recent data. The same strategy propagates through the graph, where each drop executes, send/receive data, and eventually dispatches backward the state to notify the predecessors that the actor is available again as shown in the rest of the Fig. 5.4.

Throughout this process, the dependencies must be respected, as shown in Fig. 5.5. For graph $G = \{V, E\}$, where $V = \{A, B, C\}$ and $e_1 = \{A, C\}$, $e_2 = \{B, C\}$ which means A, B are inputs of C. Thus, when C receives state then it will wait in blocking mode, meaning that the drop will not be activated (execution for App-drop or sending data until data-drop) until all the inputs are received to satisfy the dependencies. Noticing, which is applied on all three channels of e_i.

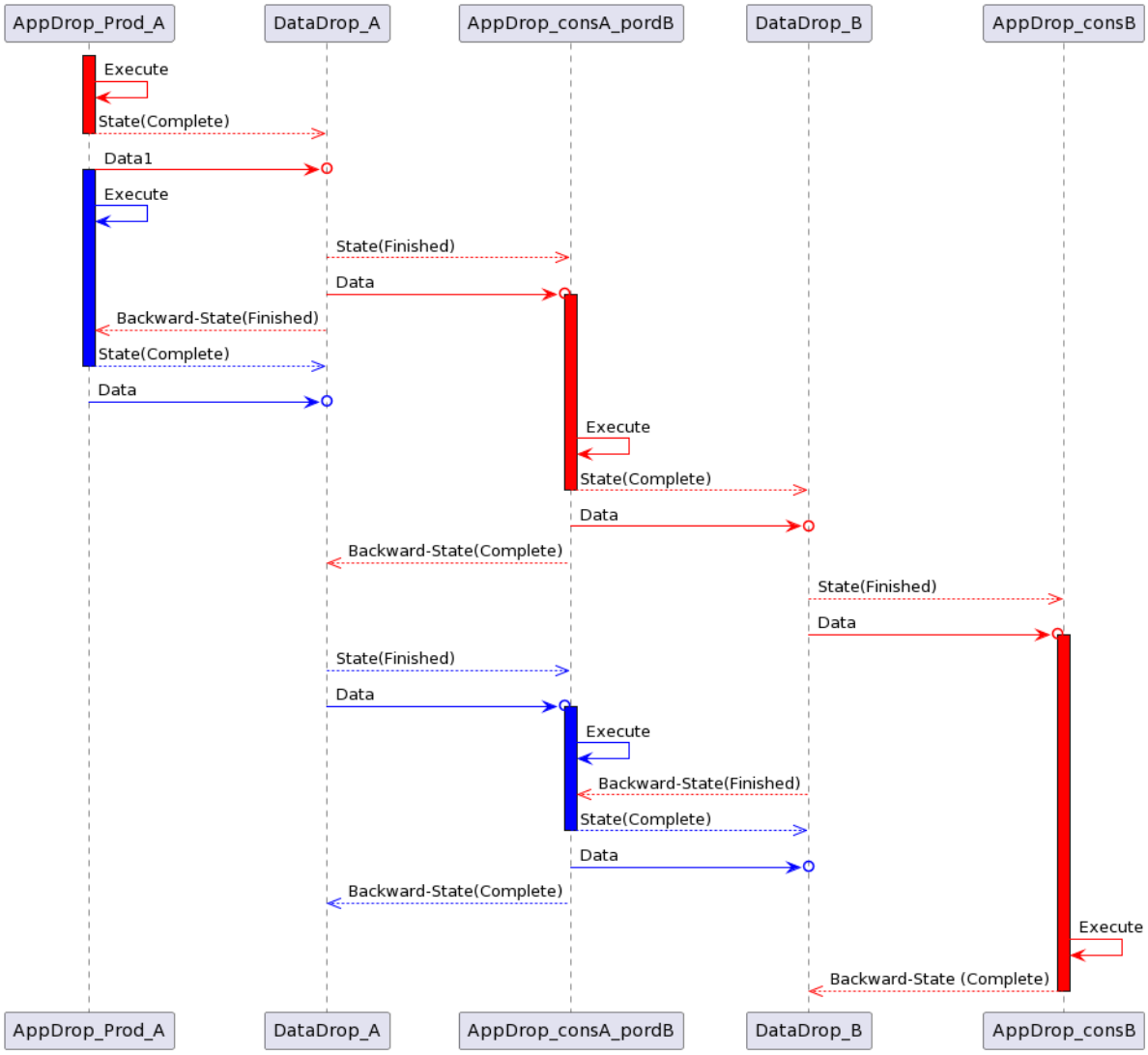

Figure 5.4: Drops communication

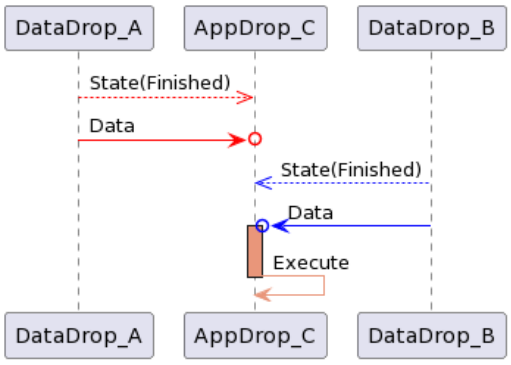

Figure 5.5: Drops communication with dependencies

Simulating the streaming mode between two actors can be achieved using the same technique of backward-state as shown in Fig. 5.6. The data-drop will receive the data tokens from the producer App-drop. The transmission process begins by setting the state of data-drop to *Data-token* and sending it to the consumer as a notification of a streaming mode. The data is then sent one token after another in the cyclic mode. The cyclic operations are based on backward-state messaging from the consumer to confirm receiving tokens and inform the data-drop if the number of tokens sent is sufficient. Once the consumer receives all the necessary tokens, it will update its state to *Tokens-done* and send it to data-drop. The consumer in this example should be executed twice to consume all the tokens of the producer, where the data-drop will send all the 4 tokens that are received from the producer, and the consumer needs 2 tokens each time.

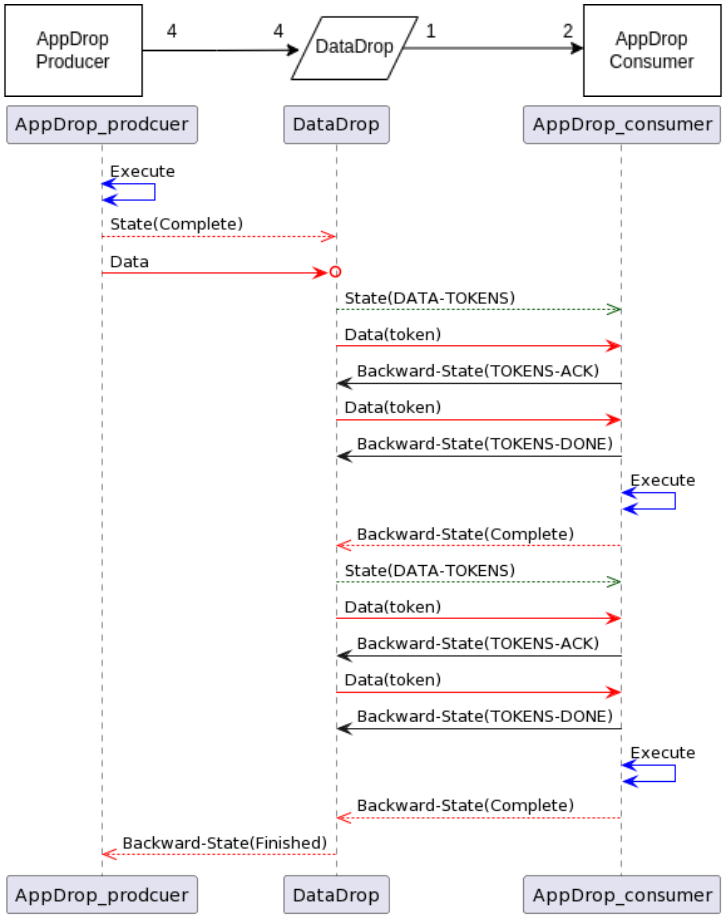

Figure 5.6: Streaming communication

5.1 Implementation details

The graphs in DALiuGE are constructed in a web interface called EAGLE [12]. The graph is assembled using a collection of Apps and data-drops that are provided in palettes. It is

possible to build new App-drops based on the need and add them to EAGLE. The graph is constructed by dragging the drops to the workspace to eventually create *logical graph* as shown in Fig. 3.3. The web interface provides a connection to DALiuGE translator that transforms the logical graph to a *physical graph template* as shown in Fig. 3.4, by unrolling the logical graph, which means if the graph contains a loop, then all the drops in the loop will be repeated as the number of the iterations. The next phase of the translator is deploying the physical graph template by transferring it to a *physical graph*, which is generated based on the available resources that will execute the graph. All three interpretations of the graph can be serialized to JavaScript Object Notation (JSON) files. Eagle web interface provides the *logical graph* and *physical graph template* to be downloaded using DALiuGE API, or we can obtain all graphs using DALiuGE shell instructions to translate the *logical graph* into all the rest graphs.

DALiuGE engine that executes the graph will parse the parameters of the graph from each phase of the translation because each of them contains unique information to be used. Similar parsing is needed to collect all the data from the graphs to simulate graph deployment. The graphs are arguments of the simulation, then RapidJSON [14] which is a fast and light parser for C++ is used to parse the JSON files of the graph to be utilized in the simulation.

The collected data from the JSON files are stored in classes members. There is a class for App-drop, and another for Data-drop, both are inherited from the Drop class, this analogy mimics the hierarchy of the classes in DALiuGE. These variables are then assigned to objects of those classes and used in the simulation.

The approach we introduced previously is implemented in SimGridn, where all drops are actors in the simulation. There are three types of actors, one for the master, and it is assigned to an App-drop or data-drop based on the constructed graph; the simulation distinguishes which drop is the master by examining the input of the drop, if the drop has no input then it is a master. There are also workers for the rest of App-drops and data-drops, the corresponding drop objects are assigned to them. The edges between actors are established using mailboxes, which are rendez-vous points accessible to any actor. A sender puts a message in the mailbox; another actor will be able to pull the message by accessing the same name of the mailbox. Since each edge is divided into three channels of communication, as described previously, three mailboxes should be engaged to depict one dependency as shown in Fig. 5.7. The names of the mailboxes are the manner to access them, so the names are constructed from the identifiers of the two actors of the dependency, for more details see Appendix A.3. In such a manner, the two actors who want to communicate on a specific channel will access the same mailbox to put and retrieve messages.

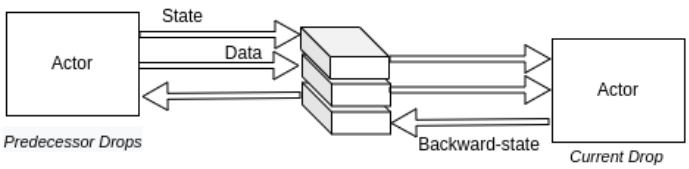

Figure 5.7: Mailboxes that correspond to channels between actors

Chapter 6

Evaluation and results

The simulation (available online [19]) will take a graph described in DALiuGE and parse all its parameter before starting the execution. For a simulation on a single node, it is enough to provide the simulation only the physical graph as JSON file and then the parser will extract all the necessary information to achieve the simulation. For a simulation on multiple nodes, a JSON physical graph template should be presented to collect data that shows how the graph is partitioned and assigned to a specific node according to DALiuGE translator. We will see that this simulation achieves a reasonable level of automation. In the next sections, we introduce some small benchmarks to validate the simulation performance.

6.1 Evaluation of data-activated approach

Considering the graph shown in Fig. 6.1 that constructed on EAGLE, three App-drops should be executed before the data-drop (memory, dynamic allocation) can send data to the last App-drops to activate it, then data-drop (file) that saves data permanently in a file. The simulation result on a single node confirmed that the time dependencies are respected and the first three Python applications are executed on a one node following Round-robin scheduling. Then by sending the states via the state channels and the data via data channels, the memory can send the data to the App that calculates the average of all the input data and then writes the final answer in a file.

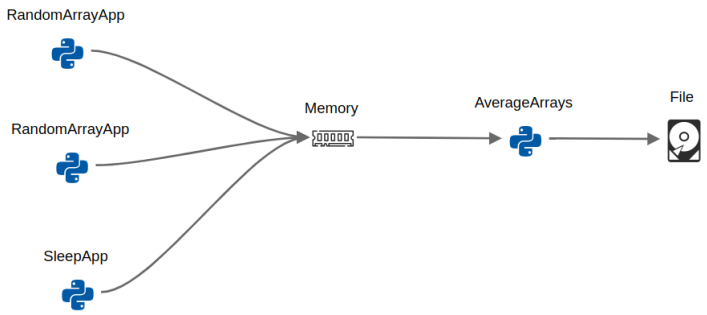

Figure 6.1: Dependencies benchmark

To compare the accuracy of the simulation, the same graph is executed on DALiuGE, and in the simulation. We consider multiple data sizes of the input drops, For the first three App-drops, the memory drop will contain all the data of the generated input. The time of execution will increase with the data size, especially for the last App-drop that needs to process all the data of the input drops. As shown in Fig. 6.2, the maximum relative error between the simulation and DALiuGE is 1.7% for this experiment, this error comes from the time of communication and also the times of messages between the mailboxes.

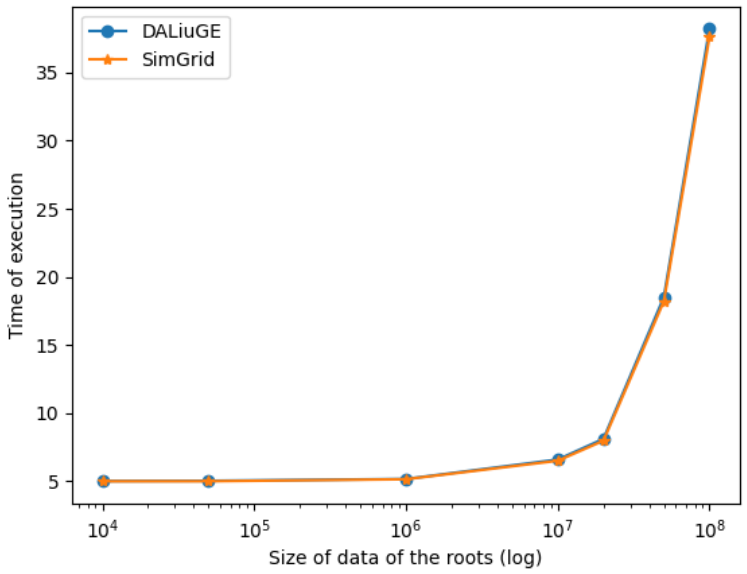

Figure 6.2: Comparison between simulation and DALiuGE execution time

6.2 Evaluation streaming input approach

In this benchmark, we need to test the concept of the third channel, the backward-state, when the input sends multiple chunks as time passes. DALiuGE doesn't support this feature distinctly. However, it is possible to create a logical graph that can be replicated as many as the number of the input when it is translated to a physical graph, as shown in Fig. 6.3 where the input (App-drop RandomArray) generates an array that is scattered to 4 branches which can be represented as 4 chunks of inputs and the rest of the graph should process those 4 chunks. On the other hand, the simulation supports simulation of this streaming input approach. The simulation follows the excepted behavior and uses the backward-state channels to manage execution termination to deal with new input data.

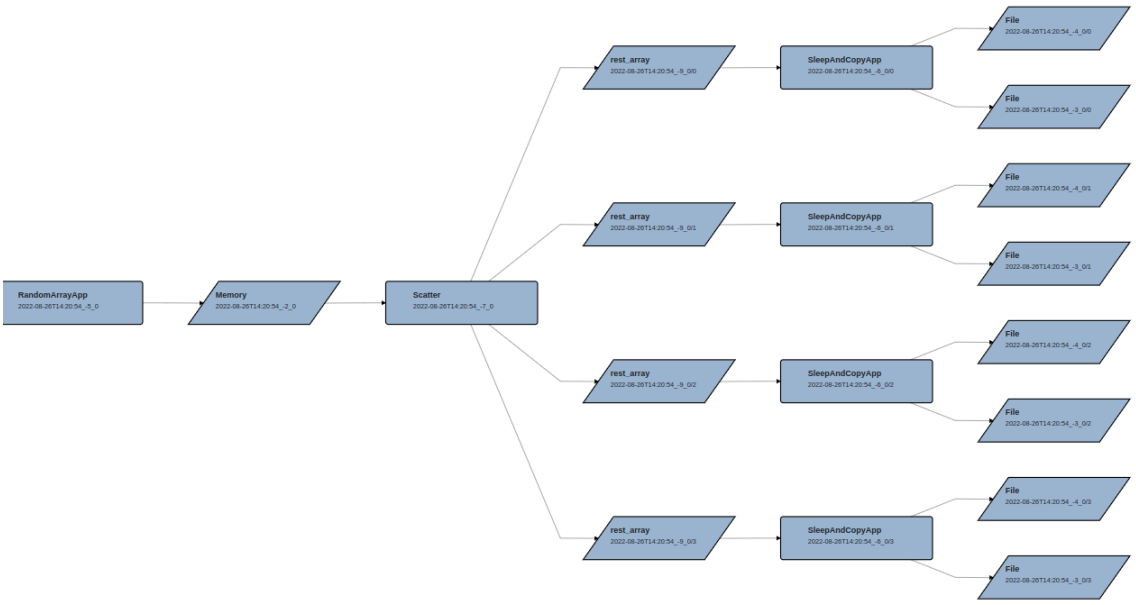

Figure 6.3: Streaming input benchmark

In the Fig. 6.4, we can see that the simulation was close enough to the DALiuGE execution time with relative error equal to 0.8% on the execution on a single node.

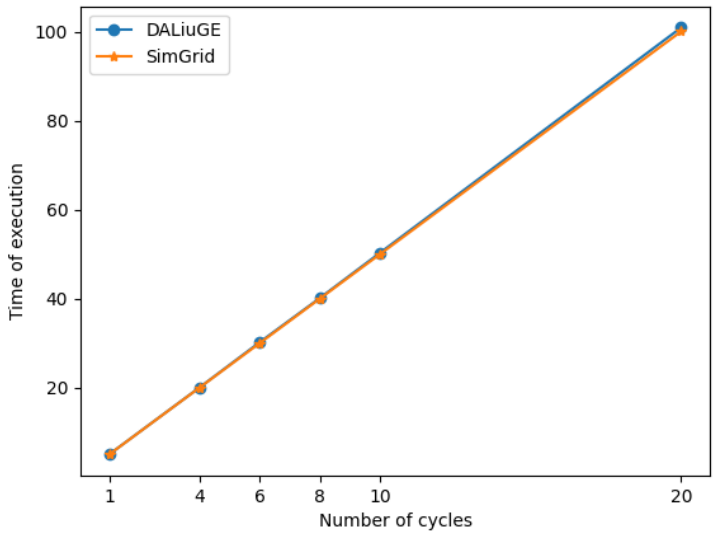

Figure 6.4: Streaming input benchmark simulation vs DALiuGE

6.3 Evaluation producer-consumer streaming

The producer-consumer streaming mode in DALiuGE is not implemented as is in SDF. In the streaming mode of DALiuGE a consumer reads the data after each certain amount of bytes of data was written by the producer in the intermediate data-drop, no need to wait until the data are fully written in the data-drop. This concept is HSDF where the producer and the consumer produce and consume a unity data token, and all drops of DALiuGE produce and consume a unity data token. Particular applications require this continuous data input, but it will not decrease the time of execution. HSDF is a special case of general SDF, taking Fig. 6.5 the token produced and consumed will be one for each drop, but the graph of Fig. 6.5 cannot be implemented in DALiuGE. Our simulation adapts to the general SDF that also includes the HSDF of DALiuGE. Thus, the graph in Fig. 6.5 can be simulated. The memory drop will receive 100 tokens from the input. The *process A* needs 50 tokens to be executed, so the memory will send 50 tokens one by one to *process A*. For all the input tokens then *process A* will be executed twice and also the *process B* will be executed twice as well. The simulation gives the result shown in Fig. 6.6.

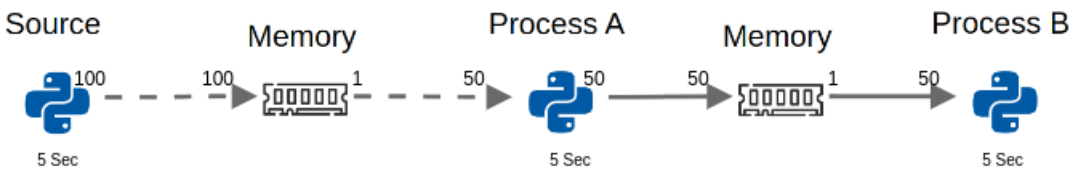

Figure 6.5: Producer-Consumer streaming with different data tokens

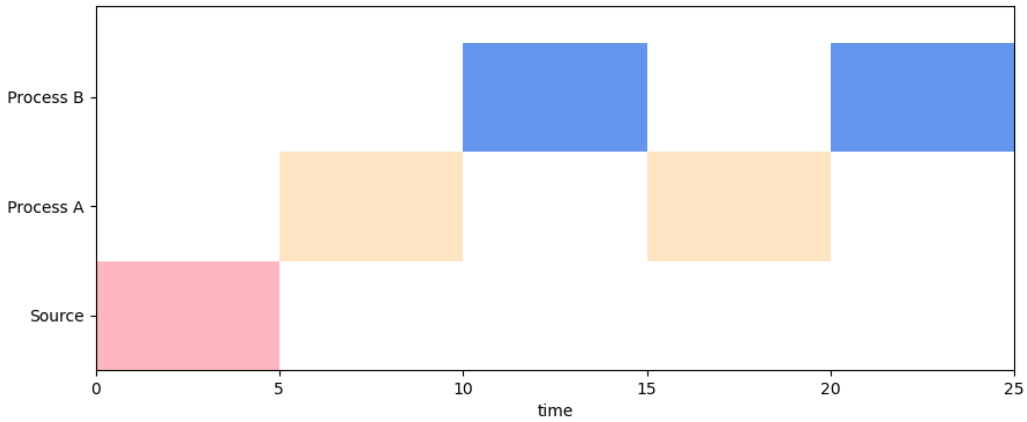

Figure 6.6: Producer-Consumer streaming simulation

6.4 Evaluation execution on multiple nodes

DALiuGE partitions the graph to be distributed on the number of nodes in such a way that each actor, including the data-app, can be executed in parallel with respecting the dependencies. Fig. 6.7 shows three roots applications and one application to produce the final data. Executing this graph on two nodes should start one root on the first node and the rest on the second node using Round-robin scheduling, and the processing application will wait until all the roots are terminated to respect the dependencies. The simulation gives the order of execution as shown in Fig. 6.8 which takes 18 seconds instead of 23 seconds on a single node. Fig. 6.8 is only the result of the simulation that used the DALiuGE partitioned graph; it can give an image of how DALiuGE will be executed because DALiuGE will assign real resources to nodes 1,2.

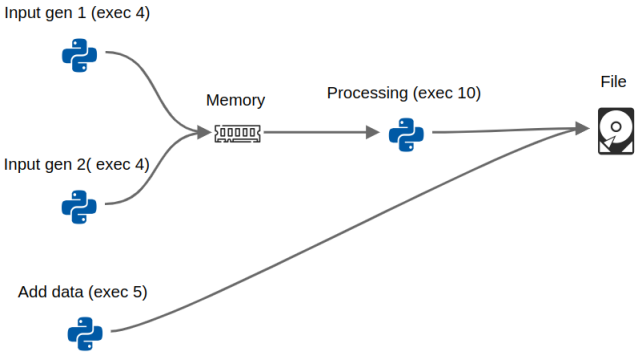

Figure 6.7: Multiple nodes benchmark

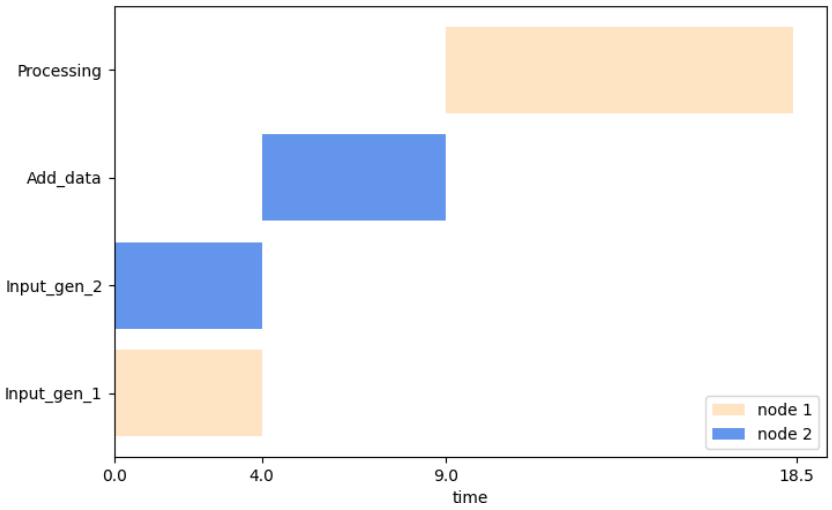

Figure 6.8: Execution on multiple nodes

6.5 Generalization

We have introduced two benchmarks; their approaches are not fully supported in DALiuGE. The streaming input benchmark was done by using a trick of scatter. In DALiuGE is not possible to design a graph that has inputs to provide new data over time. This concept is very important in data streaming applications, such as SKA, where the input is a sequence of data (chunks) that the graph should be executed for each chunk. The simulation of this work simulates this concept for any number of input tokens over time. The simulation gives a picture of the execution of the graph for a number of inputs over a fixed time, such as N chunks/s.

The second approach is producer-consumer streaming, where an actor produces more or less data than the successor actor, and this is a very common situation in signal processing applications. Execution of the actors in DALiuGE can be done once because DALiuGE is a hybrid concept between workflow and datafolow, not standard dataflow. While the simulation covers more of the dataflow. The producer-consumer streaming benchmark demonstrates the possibility of simulating one of the most important features of the dataflow. The simulation in this case needs more information to simulate this concept, such as the number of tokens consumed by the drop, as shown in Fig. 5.6. Therefore, there is a need to add this information to the JSON file of DALiuGE because DALiuGE works with uniform-produced tokens and consumed tokens. Adding information to JSON files reduces the automation of the simulation, and it is very difficult if the graph contains a large number of drops. For this reason, further betterment is needed.

6.6 Areas of improvement

Firstly, we have to avoid manually adding the parameters that represent the number of consumed tokens for each drop. Since this concept will be for general simulation not only for DALiuGE then a standard JSON file should be designed. The new JSON file requires further instructions to parse that JSON file and assign its data to the drops variables. Another crucial improvement that is very necessary relates to the execution time of the App-drop. In this work, the execution time of App-drop was measured manually and inserted into the graph parameter to have it after in the JSON file to assign it to the drop variable. For example, taking the App-drop *RandomArayAPP* in Fig. 6.1 it is a Python application, the time of execution is the time of running the script and while doing the benchmark on one node that time was measured then using it as a parameter in EAGLE. This reduces the level of automation. The solution is to write a program to run each App-drop in the graph and measure the time, then insert the time into the JSON file.

Chapter 7

Conclusion

Hardware-software co-design is a concept that aims to incorporate both domains in an attempt to optimize the performance, cost, and energy with respect to the constraints of the design. Hardware-software co-design requires a simulation that eases the process instead of using classical methods such as real experiments or analytical models. This work showed the possibility of simulation dataflow using the data-activated approach used in DALiuGE. Using the state of each actor besides the data storage to activate the successor can represent the availability of the data that active the actor as it is in the dataflow model. Using backward-state has proven its ability to simulate the streaming mode, whether in the case of streaming input or even in the case of producer-consumer streaming with a different range of consumed data tokens. Furthermore, SimGrid showed its ability as a framework to implement the simulation of the dataflow. Additional work is needed in this field to improve the simulation and cover more frameworks such as DASK for the context of SKA, and universal design that describes a dataflow model for more general simulation.

Appendix A

Appendix

A.1 DALiuGE Drop class

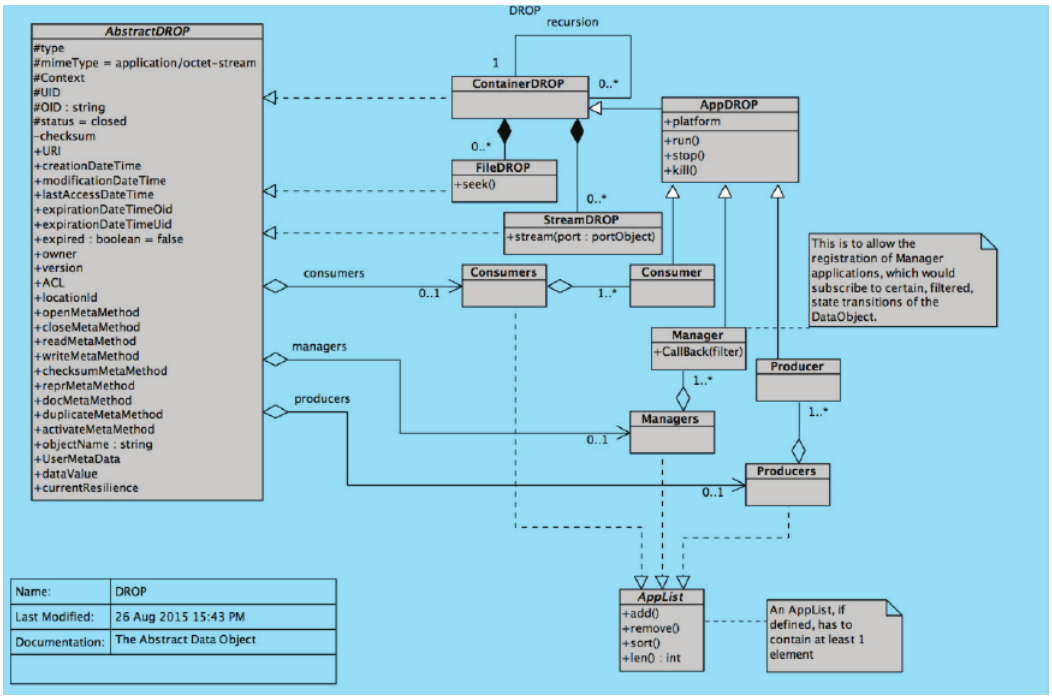

Figure A.1: Drop class diagram in DALiuGE

A.2 Drops Class diagram in simulation

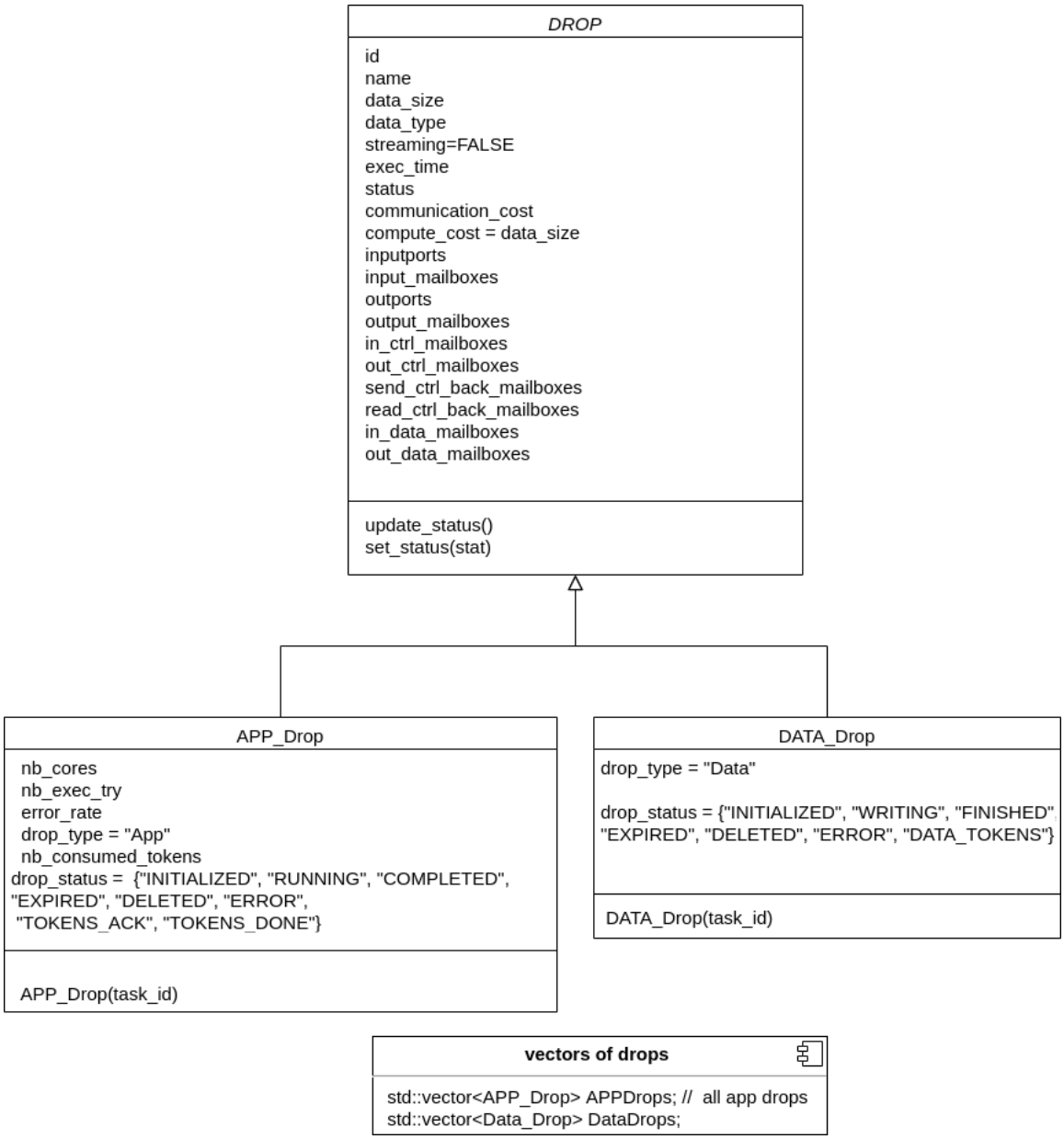

Figure A.2: Drop class diagram in the simulation

A.3 Determining mailboxes' names

Fig A.2 shows the variable that will store the parsed JSON files of DALiuGE. We can recognize two members "inputports" and "input-mailboxes". To show the difference, consider a Drops B with Id 1 has $\{A1, A2\}$ with id $2, 3$ respectively as input drops

"predecessor", where $\{A1, A2\}$ are App-drops and they are the input of B which is data-drop. So the member 'inputports' of B stores the Id of $A1$, $A2$.

When $A1$ needs to send to B, then the name of the mailbox should be shared name between the predecessor and the current drop and stored in both drops. To achieve that the name of the mailbox is formed using the "Id" of drops. For the state channel, the mailbox is shared between the output of the predecessor and the input of the current actor. Therefore, we can set the stored name of output-mailbox of $A1$ to be the id of $B - A1$ so *1-2* and the name of the input-mailbox of B to be the id $B - A1$ so the name of the mailbox is *1-2*. Thus $A1$ will put a message in the output mailbox *1-2* and B read from the input-mailbox *1-2* which is the same mailbox where the predecessor put its message.

The names of the mailboxes should be formed automatically as string variables and independently in each drop object, which can be done by following these formulas:

$$\begin{cases} App - output - mailbox = \underline{(output - data - drop - id)} + \textbf{(current-App-Drop id)} \\ App - input - mailbox = (current - App - Dropid) + (input - data - drop - id) \\ Data - output - mailbox = (output - App - task - id) + (currentData - Dropid) \\ Data - input - mailbox = \underline{(current - Data - Dropid)} + \textbf{(input-App-task-id)} \end{cases} \tag{A.1}$$

When the parser of JSON file is parsing the parameter of drop B which is DataDrop and because it has $A1$ as input app drop, then following the last formula, the name of input mailbox is $'1'+'2'$ forming *1-2* as input-mailbox. if the parser is in app drop $A1$ then the first formula is followed producing *1-2* as output-mailbox, which means the same mailbox to exchange messages on.

The second channel, the data channel, is constructed by following the exact same method, only keyword *'data'* is added to the previous formulas. For example data-input-mailbox of B from $A1$ is *1-2-data*. Similarly, the third channel, the backward-state, B will send messages backwards to $A1$ using a mailbox called *1-2-backward* and $A1$ receives that backward message on the same mailbox.

Bibliography

[1] Ilkay Altintas et al. "Kepler: An Extensible System for Design and Execution of Scientific Workflows". In: *Proceedings of the International Conference on Scientific and Statistical Database Management, SSDBM* 16 (July 2004), pp. 423–424. DOI: 10.1109/SSDBM.2004.44.

[2] Rodrigo Tobar Andreas Wicenec Dave Pallot and Chen Wu. "DROP Computing: Data Driven Pipeline Processing for the SKA". In: (2017), p. 10.

[3] Arvind and R.S. Nikhil. "Executing a program on the MIT tagged-token dataflow architecture". In: *IEEE Transactions on Computers* 39.3 (1990), pp. 300–318. DOI: 10.1109/12.48862.

[4] Rosa Badia, Eduard Ayguade, and Jesus Labarta. "Workflows for Science: A Challenge When Facing the Convergence of HPC and Big Data". In: *Supercomput. Front. Innov.: Int. J.* 4.1 (Mar. 2017), pp. 27–47. ISSN: 2409-6008. DOI: 10.14529/jsfi170102. URL: https://doi.org/10.14529/jsfi170102.

[5] Ryan Bunney, Andreas Wicenec, and Mark Reynolds. "SHADOW: A workflow scheduling algorithm reference and testing framework". In: (Jan. 2020), pp. 148–155. DOI: 10.25080/Majora-342d178e-014.

[6] Henri Casanova et al. "Developing accurate and scalable simulators of production workflow management systems with WRENCH". In: *Future Generation Computer Systems* 112 (2020), pp. 162–175. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2020.05.030. URL: https://www.sciencedirect.com/science/article/pii/S0167739X19317431.

[7] Henri Casanova et al. "Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms". In: *Journal of Parallel and Distributed Computing* 74.10 (June 2014), pp. 2899–2917. URL: http://hal.inria.fr/hal-01017319.

[8] Anne-Cécile Orgerie Clément Courageux-Sudan and Martin Quinson. "Automated performance prediction of microservice applications using simulation". In: *International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (2021), p. 9.

[9] *DALiuGE documentation*. URL: https://daliuge.readthedocs.io.

[10] Ewa Deelman et al. "Pegasus, a workflow management system for science automation". In: *Future Generation Computer Systems* 46 (2015), pp. 17–35. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2014.10.008. URL: https://www.sciencedirect.com/science/article/pii/S0167739X14002015.

[11] Jack B. Dennis. “First Version of a Data Flow Procedure Language”. In: *Programming Symposium, Proceedings Colloque sur la Programmation* (1974), pp. 362–376.

[12] *EAGLE Website*. URL: https://eagle.icrar.org.

[13] E. A. Lee and S. A. Seshia. “Introduction to Embedded Systems - A Cyber-Physical Systems Approach”. In: MIT Press, 2017. Chap. Concurrent Models of Computation, pp. 147–162.

[14] *RapidJSON github*. URL: https://github.com/Tencent/rapidjson/.

[15] Matthew Rocklin. “Dask: Parallel computation with blocked algorithms and task scheduling”. In: 130 (2015).

[16] Patrick R. Schaumont. “A Practical Introduction to Hardware/Software Codesign”. In: Springer New York, NY, 2010. Chap. Data Flow Modeling and Implementation, pp. 33–69.

[17] Praveen K. Murthy Shuvra s. Battacharyya and Edward A. Lee. “Software synthesis from dataflow graphs”. In: Kluwer Academic Publishers, 1996. Chap. Synchronous dataflow, pp. 37, 57.

[18] *Telescope Operations Simulator github*. URL: https://github.com/top-sim/topsim#running-your-first-simulation.

[19] *The simulation repository*. URL: https://github.com/SulaimanMohammad/dataflow_simulation.

[20] *The SKA Project Website*. URL: https://www.skatelescope.org/.

[21] C. Wu et al. “DALiuGE: A graph execution framework for harnessing the astronomical data deluge”. In: *Astronomy and Computing* 20 (2017), pp. 1–15. ISSN: 2213-1337. DOI: https://doi.org/10.1016/j.ascom.2017.03.007. URL: https://www.sciencedirect.com/science/article/pii/S2213133716301214.