Distributed ParaPy – increasing the computational efficiency of KBE models

Julien Oiknine

Technische Universiteit Delft

ParaPy

TU Delft Delft University of Technology

Challenge the future

DISTRIBUTED PARAPY – INCREASING THE COMPUTATIONAL EFFICIENCY OF KBE MODELS

by

Julien Oiknine

in partial fulfillment of the requirements for the degree of

Master of Science

in Aerospace Engineering

at the Delft University of Technology,

to be defended publicly on Friday September 23, 2022 at 09:30 AM.

Student number:	5152690	
Supervisor:	Dr. ir. G. La Rocca,	TU Delft
Thesis committee:	Dr. ir. M. M. van Paassen,	TU Delft
	Dr. ir. A. H. van Zuijlen,	TU Delft
	Ir. R.E.C. van Dijk,	ParaPy

This thesis is confidential and cannot be made public until September 23, 2025.

An electronic version of this thesis is available at http://repository.tudelft.nl/.
Equivalent number of words: 25581.

TU Delft Delft University of Technology

PREFACE

I would like to thank all the persons without whom the elaboration of this master's thesis would not have been possible.

First, my supervisor and CEO at ParaPy, Reinier van Dijk, who has been there for me every step of the way. He has guided me, helped me when I was feeling stuck, and offered valuable input to direct my research into new directions.

I also want to give a warm thank you to GianFranco La Rocca, my academic supervisor, for his guidance and helpful pointers in the right direction when I needed them.

Then, I would like to thank San Kilkis, a dear friend and colleague who masters the art of Python language like no one and has always answered my questions in the promptest manner as well as helped me solve any problem I might have had with Git. He has been extremely helpful and always there for me when I needed him over the last few months.

A warm thank you as well to my family for supporting me in these challenging times.

And finally, I wanted to acknowledge all the amazing work done by the Open Source community, who give their all to create and develop free and available to all software. My research and programming would not have been possible without them and their selfless dedication.

Julien Oiknine
Villiers-st-Frédéric, September 2022

ABSTRACT

As Knowledge Based Engineering (KBE) is gaining in popularity in the industry, the models developed with the technology grow in complexity. The larger KBE models suffer from long (re)generation times and outgrow the memory resources of standard desktop computer. In the meantime, the number of cores present in computer's processors is steadily increasing, and the cloud revolution has changed the way software is consumed.

Yet, KBE models are not doomed to become slower and heavier as their complexity increase. They often feature inherent parallel region, which could be exploited to decrease their (re)generation time. ParaPy, a leading company in KBE technology, sees this opportunity and research possible solutions to distribute and parallelize their models. To this extent, this works proposes a partitioning strategy for parallelizing KBE models and extends the ParaPy KBE system with high-level parallel programming constructs to express and exploit parallelism present in their models.

Distributed ParaPy, as is called the extended ParaPy system, provides two new programming constructs, Persistent Remote Models (PRMs) and Transactional Remote Models (TRMs), that let developers define child objects in their model that will be instantiated in their own process. The remote children created with PRMs are long-lived and reproduce the behaviour of traditional ParaPy children, while introducing the opportunity to parallelize the evaluation of their slots. PRM can be displayed in ParaPy's graphical user interface, which also automatically parallelize the generation of their geometries. TRMs are short-lived remote children, which are instantiated in the objective to compute a specific set of output slots and destructed once the computation is done. They return a dependency tracked lightweight version of the remote object containing only the values of the computed output slots. They provide the same parallelization opportunity than PRMs and add a solution to control the memory footprint of a model.

In addition to extending the ParaPy KBE system, this work proposes and compares 3 cloud architectures to run Distributed ParaPy model. These architectures are partially implemented such that their respective potential can be assessed. The developed constructs and cloud architectures have been demonstrated on a KBE application from the industry, to evaluate their impact on (re)-generation time and memory footprint.

The tests operations performed on the application showed that the (re)generation time of KBE models can be reduced through parallelization. The speedups obtained depend on the type of operation performed on the model, and it is observed that the parallelization efficiency is higher when running in the cloud than on a desktop computer. Surprisingly, running a parallelized model across several cloud nodes provided better speedups than running on a single node with the same level of parallelization despite the communication overhead between nodes.

LIST OF FIGURES

List of Tables

GLOSSARY

futures Programming constructs used for synchronization in some parallel computing frameworks. 11, 12

NOMENCLATURE

API Application programming Interface. 13, 25–27, 30, 31, 35, 36, 49

CAD Computer Aided Design. 3

CPU Central Processing Unit. 20

GIL Global Interpreter Lock. 6

GUI Graphical User Interface. 1, 13, 14, 17–19, 26, 30, 31, 41, 42, 44–46, 48, 49

IT Information Technology. 6

KBE Knowledge Based Engineering. v, 1–4, 6, 10, 13–15, 17–19, 33, 39, 47, 49–51

MDO Multidisciplinary Design Optimization. 1–4, 13

MP Main Process. 13, 20, 22, 25, 27, 29–32, 41–43, 50

PP Persistent Proxy. 11, 25

PRM Persistent Remote Model. v, 10–13, 17–21, 24–33, 35, 36, 39, 41–43, 45, 46, 49–51

PS Part Studio. 3

RET Rule Evaluation Thread. 27

RH Royal Huisman. 16–18, 32, 33, 49, 50

RM Remote Model. 10, 11, 15, 16, 23–25, 36, 37, 49, 50

RO Remote Object. 11–14, 17, 19–21, 23–32, 35, 37, 39, 41–45, 47–50

SaaS Software-as-a-Service. 14

SKD Software Development Kit. 4, 9, 17, 49

TRM Transactional Remote Model. v, 10, 11, 13, 14, 17, 19, 21, 22, 24, 35, 47–50

VM Virtual Machine. 4, 7, 15, 20, 21, 39, 40, 44

WP Worker Process. 10, 11, 13–17, 19–21, 23–25, 29, 31, 35–37, 39, 41–45, 47, 49, 50

Contents

1

INTRODUCTION

1.1. MOTIVATIONS

Knowledge Based Engineering (KBE) is a technology used to automate and speed up the design process of complex products by capturing and re-using engineering knowledge [**?**]. KBE originated in the aerospace industry where it is used to develop detailed aircraft models common to every discipline involved in the early design stages [**?**]. Over time, KBE gained in popularity and escaped the aerospace niche to find applications in various engineering domains, from the automotive [**? ?**] to the shipbuilding industries [**? ?**].

KBE models can be used inside Multidisciplinary Design Optimization (MDO) frameworks or directly accessed by an end user in the form of a KBE application. In MDO, KBE is used to automatically generate a model for each set of iteration parameters, on which the different discipline analyses will be performed. KBE applications on the other hand define a custom Graphical User Interface (GUI) through which the user can visualize the model, tune its configuration and eventually perform application-specific operations by using the widgets provided by the GUI. Figure 1.1 shows the GUI of a KBE application.

As KBE gain in popularity, the model developed with the technology grow in complexity. This increased complexity affects the performance of larger KBE models, resulting in longer (re)generation times and large memory footprint. As an example, the geometry generation of the model shown in figure 1.1 takes around 40 seconds on a regular desktop computer. For MDO, the generation time of the model will directly impact the overall duration of the optimization. For users working with a KBE application to design a product, the model (re)generation time will impact the GUI's response time upon design changes. Beside hampering the user experience and reducing the time left for design space exploration, delays in an application's GUI response have been proven to negatively affect the quality of a design activity and increase the risk of errors in the final design (such as a faulty design or an unoptimized design) [**?**]. As for the memory footprint of large models, it results in some KBE models outgrowing the memory resources of standard desktop computers.

According to Moore's law, the number of transistors on a microchip (hence their computing power) should double every two years. For over 50 years this law held true, yet the pace has slowed down since the early 2010s, and processor manufacturers have started to focus on multicore processors instead of single core ones to increase their chip's computing power [**? ?**] (see figure 1.2). As a result, parallel computing is becoming one of the main approaches to increasing software performance. Beyond the boundaries of single desktop computers, cloud computing [**?**] can provide

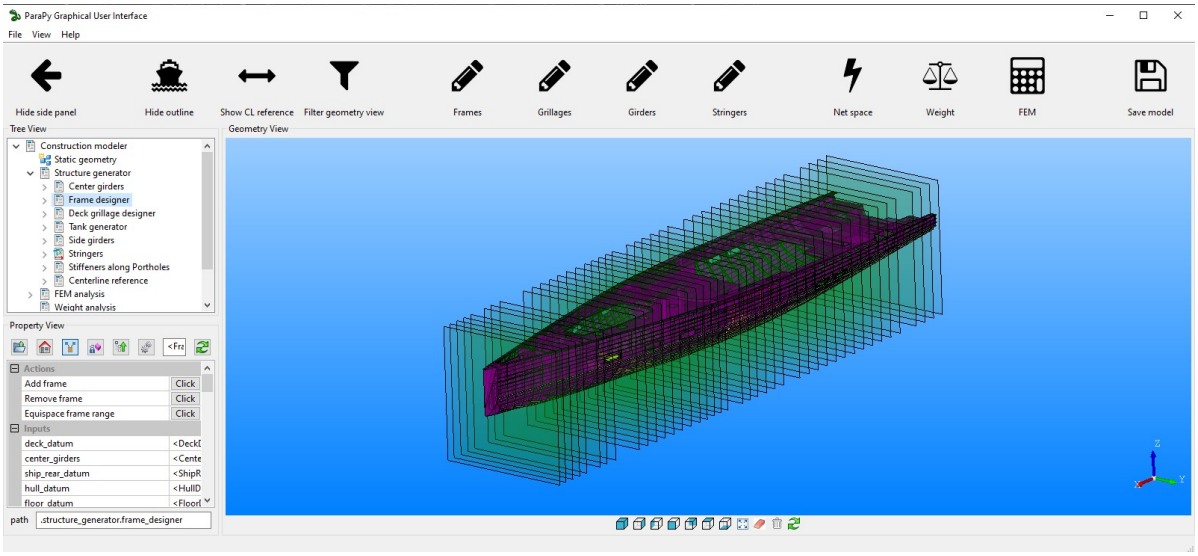


Figure 1.1: Example of KBE application developed with the KBE system ParaPy [**?**]. The model's geometry can be visualized in the center panel. The left panels let the user browse the model components and tune its configuration. Buttons at the top are the widgets that this specific application provides to perform pre-programmed design operations on the model.

flexible and almost infinite computing resources. It has become an interesting alternative to grid computing and dedicated High Performance Computing (HPC) clusters to run parallel computing applications [**? ?**].

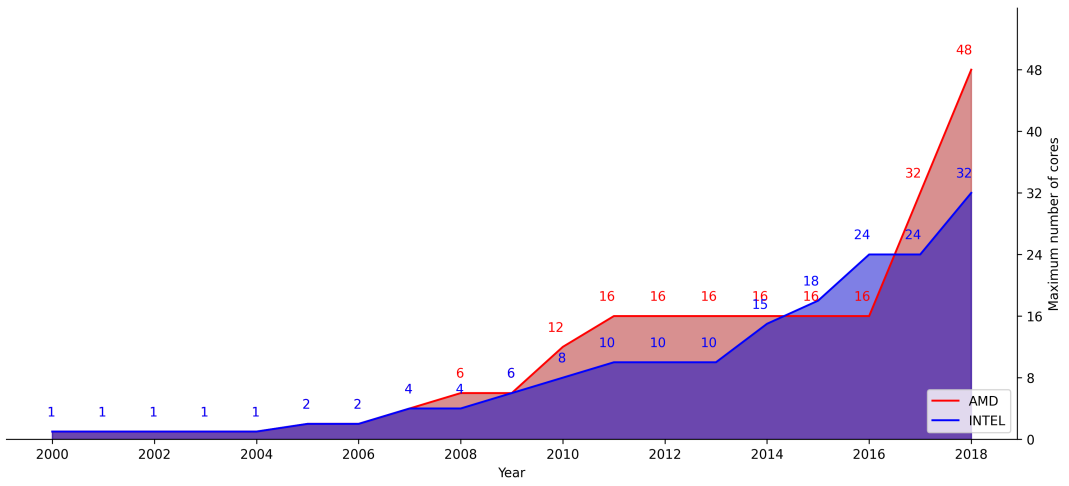


Figure 1.2: Evolution in the maximum number of cores per processor in the past decades for the two industry leaders.

ParaPy [**?**], a software company developing the KBE system of the same name, sees the opportunities offered by multicore computers and cloud computing to increase the computational efficiency of its system. ParaPy is interested in finding a solution to distribute and parallelize its KBE models to reduce their (re)generation times and developing new computing models to control their memory footprint. This master thesis work lies within this initiative and will present the methodology and software development work conducted to improve the computational efficiency of the ParaPy KBE system in these matters.

Parallel computing has already been researched and integrated in the design workflows involving KBE models. In MDO, parallelization is used to run simultaneously discipline-specific analyses

for each iteration, as illustrated by the `ParallelGroup` constructs of the open source MDO framework openMDAO [**?**]. Other works also aimed at parallelizing the generation of meshes [**?**], which is often a required step before performing analyses on KBE models' geometries. Parallelization of the analyses themselves, such as solving flow equations, is also an important research topic [**? ? ?**]. However, the parallelization of the KBE model's generation has received less attention.

1.2. RELATED WORKS

Even if KBE technology is gaining popularity in the industry, it has received little attention from academia and research worlds [**?**]. As a result, there is no real literature about the distribution and parallelization of KBE systems.

An effort to parallelize the generation of KBE models was made by the concurrent KBE system Gendl [**?**]. In this KBE system, the developer can partition a model into child sub-models and run these sub-models in different local or remote server processes. The generation of the distributed sub-models can therefore be executed in parallel. An illustration of this partitioning is shown in figure 1.3. Unfortunately, A single documentation paper was found with some high level instructions to use this feature [**?**]. While this works gives insights on how to partition a KBE system for parallelization, it is difficult to build on it, and it presents some shortcomings. Setting up a distributed model requires the manual creation and configuration of the server running the instances of the sub-models, as they cannot be dynamically created by the system. Furthermore, this work doesn't provide directions for a possible cloud architecture to run distributed KBE models.

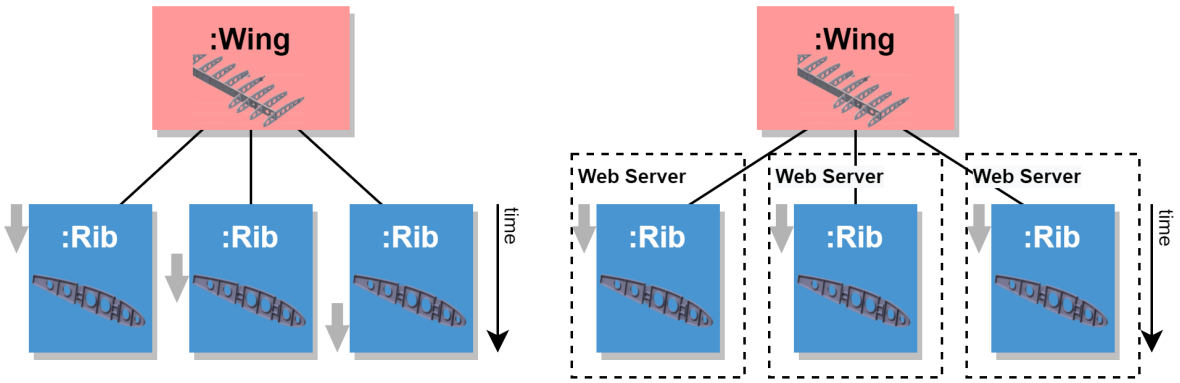


Figure 1.3: Partitioning of a KBE model and parallelization in Gendl.

A similar solution is implemented for Computer Aided Design (CAD) by the commercial and web-based software system OnShape. In OnShape, users can create and modify CAD parts or sketches in a so-called Part Studio (PS). A PS is a container in which the user can create and modify CAD parts, materialized by a tab in the browser [**?**]. Each PS runs in its own process called "container" (which could hint that these so-called containers are actually Docker containers), such that the generation of the CAD part in each PS can take place in parallel. For example, if a PS contains a sketch, and multiple other PS references this sketch and build a 3D geometry based on it, then, when the sketch is modified in the "parent" PS, the geometries in the "dependant" PS are updated in parallel. The partitioning of the CAD model is therefore comparable to the one of a KBE model in Gendl.

Parallel generation of KBE models has also been implemented at a higher level in optimization processes. Some optimization algorithms, such as differential evolution, generate multiple set of design variables at each iteration stage. When the optimization is based on a parametric model, multiple instances of the model can be instantiated and kept alive to be regenerated in parallel for each set of design variables. Such solution is implemented by Brouwer in his master thesis [**?**].

Multiple instances of a Flying-V KBE model are kept alive during the whole optimization process and their geometries are generated in parallel for each set of design variables generated by the differential evolution algorithm. At Boeing, Carrere present a similar architecture to parallelize the generation of a complex parametric propulsion system for multiple sets of design parameters during a MDO process [**?**]. The model instances are running on cloud based Virtual Machines (VMs). These efforts outline the need to reduce the model generation time during optimization processes, however the solutions they propose are ad-hoc and target a specific type of optimization.

1.3. RESEARCH OBJECTIVE AND QUESTIONS

Every KBE system is different and present its own specificities. This statement is particularly true for ParaPy, being implemented in and on top of Python as opposed to the other KBE systems being based for the most part on a Lisp-like language [**?**]. To this extent, this thesis work will target specifically the ParaPy KBE system. Based on section 1.1, the following research objective is formulated:

> **Extend the ParaPy SKD with new programming models and constructs to increase the computational efficiency of KBE models in terms of run time and memory management.**

From this main research objective, several research questions and sub-questions can be derived, which read:

RQ1: Can the (re)generation time of a ParaPy KBE models be reduced through parallelization?

- **a.** What are the main overheads when parallelizing a KBE model?

RQ2: How to distribute a KBE model while preserving its integrity?

- **a.** How to adapt the KBE lazy evaluation, dependency tracking and caching mechanisms to function across multiple processes?
- **b.** Are there conditions to be verified by a KBE model to be parallelized?
- **c.** To what extent can the complexity arising from parallelization can be hidden to the model developer?

RQ3: How to leverage cloud computing in the most efficient manner to run distributed KBE models?

RQ4: Are there opportunities to reduce the memory footprint of a KBE model in a distributed architecture?

1.4. DOCUMENT OUTLINE

Chapter 2 will introduce relevant knowledge and jargon required to understand the content of this work. Chapter 3 will present the methodology followed to answer the posed research objective and questions. Chapter 4 will cover the implementation of Distributed ParaPy, and its verification will be presented in chapter 5. The results will be shown and discussed in chapter 6, and eventually chapter 7 will conclude and propose recommendations for further works.

2

BACKGROUND

2.1. PARALLEL COMPUTING

In this section relevant parallel computing notions related to this thesis work will be introduced

2.1.1. PARALLEL PROGRAMMING MODELS

A parallel programming model provides a set of high level abstractions to express in a clear and simpler way the low level execution of a parallel program. This section will present the main parallel programming models.

SHARED MEMORY MODEL

In the shared memory model, a program execution is composed of multiple concurrently running threads within the same process. If the program is run on a multicore machine, threads can run in parallel. The communication between threads is implicit, as they have access to the same memory space and therefore the same data. Locking mechanisms have to be implemented to avoid multiple thread accessing the same data at the same time. The advantage of this model is the absence of communication overhead between the simultaneously running pieces of the program. Its main disadvantage is that a program parallelized using a shared memory model is bound to a single machine and cannot be scaled over a cluster of computers.

MESSAGE PASSING MODEL

In this model, the different pieces of a program running concurrently have their own private memory space and communicate together through messages. There are two types of messages:

- Asynchronous messages: once the message is sent, the process initiating the communication can continue to perform operations directly, without waiting for a reply. The communication is said to be non-blocking.
- Synchronous messages: these messages expect a reply from the recipient. The sending process will therefore wait until it receives the reply, the communication is said to be blocking.

Higher level programming abstractions exists on top of the message passing model.

TASK PARALLELISM

This model defines how a program is partitioned to introduce parallelism. In the task parallelism model, a program is partitioned in a set of different tasks that can be run simultaneously. Tasks have

inputs and outputs. This model can be used on top of the shared memory or the message passing models.

ACTOR MODEL

In the actor model, a parallel program is viewed as a set of autonomous objects called *actors*, communicating together through message passing [**?**]. Autonomous means that each actor possesses its own thread of execution and state, the latter being private. When an actor receives a message, it executes the method specified by the message arguments. The original actor definition stated that actors should only communicate through asynchronous messages, however more developed communication patterns such as synchronous messages (request/reply) can be implemented [**?**]. Notorious actor implementations are the programming language Erlang [**?**] and the distributed programming frameworks Akka [**?**] and Orlean [**?**] which implement the actor model for Java and .Net applications respectively.

2.1.2. PARALLEL COMPUTING IN PYTHON

As ParaPy KBE system is implemented in and on-top of Python, this paragraph will introduce the main characteristics of Python regarding parallel computing. A notable feature of Python is the Global Interpreter Lock (GIL). The GIL is a locking mechanism which ensures that a single thread can control Python's interpreter at a time. The functioning of the GIL won't be detailed here, but one of its consequence is that parallelism is not possible in Python using threads. The preferred way to introduce parallelism in Python is to distribute the algorithm execution into multiple processes.

2.1.3. AMDAHL'S LAW

For a given application, the speedup achieved by introducing parallelization does not only depend on the number of processors used. Amdahl [**?**] stated that this speedup is limited by the non-parallelizable portion of the code, also called the sequential portion of the code. Hill & Marty [**?**] provide a synthetic expression of the Amdahl's equation: given a computer program with an infinitely parallelizable fraction of code f (and thus a serial fraction of code $1-f$), then the maximum speedup s_N achieved by using N processors is given by:

$$s_N = \frac{1}{(1-f) + \frac{f}{N}}$$

The consequence of this law is that no matter the number of available computing resources, the speedup obtained from parallelization will always be bound by the serial portion of the code. It is an important rule to have in mind when considering the result of a parallelization.

2.1.4. NESTED PARALLELIZATION AND OVERSUBSCRIPTION

Nested parallelism happens when a parallel region of a parallelized program triggers another layer of parallelization as illustrated in figure 2.1. Nested parallelism can result in oversubscription (more threads and/or processes working in parallel than the number of processors of the machine the program is running on) and eventually reduce and even invert the gain in computation time obtained from parallelization. In ParaPy, the geometry kernel already uses parallelization to improve the computational performance of some geometry operations. Adding a new layer of parallelization in ParaPy could therefore result in oversubscription.

2.2. CLOUD COMPUTING

Cloud computing is the use of Information Technology (IT) resources (such as servers, data storage, software applications) over the internet in an on-demand and pay-as-you-go fashion. It provides an often advantageous alternative to owning and maintaining these IT resources on premise. A

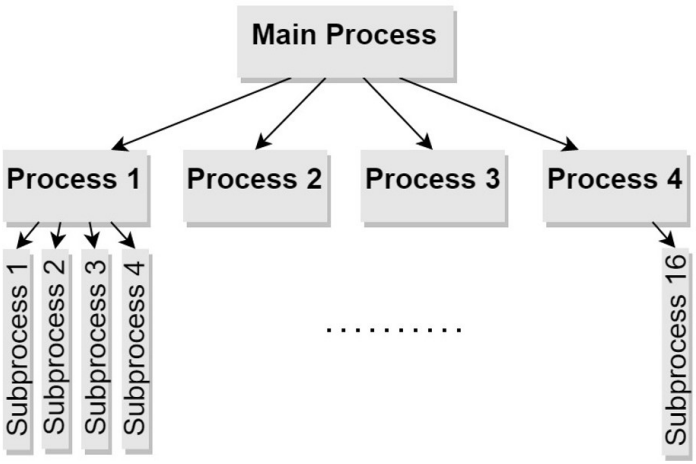


Figure 2.1: Visualization of nested parallelism.

large part of this work will focus on cloud computing and therefore an introduction to its related technologies is necessary.

2.2.1. CONTAINERIZATION

Containerization is the packaging of an application's code together with the required libraries, dependencies and configuration files to run the code in isolated user spaces, called containers. Containerization allows developers to run an application consistently in different infrastructure. Containers are more lightweight than traditional virtualization technologies such as Virtual Machines (VMs) since they share the host machine's operating system, which is the reason containerization is the preferred way to run applications in the cloud. Many solutions exist to create and run containerized application but the most popular platform is Docker [**?**]. As Docker will be used in this thesis, its containerization process will be shortly introduced.

There are two important components in Docker: Docker images and Docker containers. A Docker image is the file containing the containerized application's code and dependencies. It acts as a read only template for creating Docker containers. To this extent, Docker containers can be described as running instances of Docker images. Docker images are composed of layers, each layer originates from the previous layer and adds new components to the image, such as a new dependency or codebase. The layering of docker images allow for high reusability and saving in disk space. For example, if two images include the same first layers then these layers will be stored only once by the docker engine.

2.2.2. ORCHESTRATION

Orchestration is the use of programming technology to manage the deployment and management of applications' workload on a cloud infrastructure. While this first definition can seem vague to a novice reader, an orchestration tool can be described in the context of this thesis as a system controlling the lifecycle of the containers composing a distributed cloud application. In this work, the open-source Kubernetes [**?**] orchestration system is used. The following paragraph will provide a simplified view of the Kubernetes' components used in this work.

Kubernetes manages the lifecycle of pods on a cluster. A cluster is a collection of physical machines or VMs managed called nodes and a Pod can be seen as Kubernetes wrapper around a container. Pods can be directly created by a user, or indirectly created by another workload resource. For example, if a Kubernetes cluster is used to run a web application accessible from the internet,

the cluster can be configured so that Kubernetes scales the number of pods to adjust to the incoming traffic: with increasing traffic Kubernetes will add more pods to cope with demand, and discard pods if the traffic decreases.

3

PROPOSED METHODOLOGY

This chapter will present the methodology followed in this work to answer the thesis objective and questions. The first step is to analyze the inner functioning of ParaPy models to identify where inherent parallelism is located and propose a partitioning strategy and a programming model to exploit it. In the meantime a solution to control and reduce the memory footprint of parallelized model will also be defined. Once the strategy to improve the computational efficiency of ParaPy models is set, development work is carried out to create high level programming constructs that will be made available to model developers, thus allowing them to use these constructs in order to express the parallelism present in their own models. Then, three different cloud architectures are proposed and developed to run distributed models. Each architecture is expected to provide better performance results than the previous one, to the cost of making more assumptions regarding the solving of technical challenges induced by the architectural choices. This overall development work will result in an extension of the ParaPy SKD named *Distributed ParaPy*. Finally, the improvements in computational efficiency of Distributed ParaPy compared to the standard implementation are demonstrated on a use case from the industry, on a multicore desktop machine and in the Cloud for the three proposed architectures.

The run time improvements resulting from parallelization will be assessed by computing the **speedup** obtained by using the developed programming construct when running specific test-cases on the model. In parallel computing, the speedup s_N is defined as the ratio of the run time of the serial algorithm to the run time of the parallel algorithm, using N processors. Let T_S be the run time of the serial algorithm and T_N the run time of the parallel algorithm on N processors, then

$$s_N = \frac{T_S}{T_N}$$

Another relevant metric is the parallelization efficiency η, defined as the ratio between the speedup and the number of processors:

$$\eta = \frac{s_N}{N}$$

The assessment method for the memory efficiency is more ad-hoc and will be introduced in section 3.4.2

Section 3.1 and 3.2 will present the chosen partitioning strategy and programming models. Section 3.3 will introduce the three different architectures to gradually exploit the resources offered by

```
class Aircraft(Base):

    ...

    @Part
    def wing(self):
        return Wing(airfoil=self.airfoil,
                    span=self.wing_span)

    @Part
    def tailplane(self):
        return Tail(airfoil=self.airfoil,
                    span=self.tail_span)

    @Part
    def engine(self):
        return Engine(position=self.wing.engine_position)
```

Figure 3.1: Example of children defining parallel regions in a KBE model.

cloud computing to run distributed ParaPy application and their corresponding assumptions. Eventually, section 3.4 will present the ParaPy KBE application that will serve as a use case, and which specific test-cases will be performed to assess the performance improvements.

3.1. PARTITIONING STRATEGY

Similarly to Gendl [**?**] this work proposes to partition ParaPy KBE models across parent-child relations to parallelize the evaluation of slots in sibling children objects. Indeed, ParaPy models often features inherent parallel regions encompassed inside sibling children of a parent object, if the children do not present interdependencies (a slot of one child depending on the slot of another child). Such configuration is frequent in KBE models, and is often implemented by repetitive structural elements in models representing the geometric structure of a product, as illustrated by the wing model of figure 1.3. The other advantage of partitioning the models across children is that on top of often defining parallel regions they also make the identification of such parallel regions relatively easy to model developers, especially when the children are defined in Sequences. In order to identify these parallel regions, the model developer only has to verify that the defined inputs of the children do not depend on slots from the others. An example is given in figure 3.1. Once instantiated, the `wing` and the `tailplane` children will define regions with no interdependencies where parallelism could be exploited. This is not the case for the `wing` and `engine` children.

3.2. PROGRAMMING MODELS

In order to exploit the parallel regions introduced above two constructs have been developed, namely Persistent Remote Model (PRM) and Transactional Remote Model (TRM). These two constructs will sometime be referred to by the umbrella term Remote Model (RM) in the rest of this document. Note that the Model of the RMs naming refers to KBE models and not programming model. PRMs and TRMs provide two different parallel programming models, respectively actor based and task-parallel models, to parallelize the evaluation of slots in non-dependent sibling children. The reasons behind these choices will be detailed in subsequent sections. RMs are children of a ParaPy object that live in a different process than their parent. They are declared with a customized `@Part` decorator, and their definition must follow the same strict syntax than Part slots. When a slot defining a RM is accessed, the ParaPy class instance that should be returned according to the slot definition is actually instantiated in a different process, called Worker Process (WP). As explained in section

2.1.2, multiprocessing is the only way to execute parallel code in Python. The instance is called Remote Object (RO). A proxy object acting as a handle on the RO is returned by the slot in the main process. Internally, communication between proxies and their corresponding RO happens through message passing.

The ParaPy syntax to implicitly define Sequences (with the `quantify` keyword) is compatible with RMs. If the keyword is used in a RM definition, the Sequence is said to be a remote Sequence, and its members will be treated as independent RMs, meaning that their respective ROs will be living in different WPs. Remote Sequences are therefore very handy to parallelize a ParaPy model.

The respective programming models of PRMs and TRMs will be further detailed in section 3.2.1.

3.2.1. PERSISTENT REMOTE MODELS

PRMs are the RMs whose behaviour resembles the most the one of traditional child objects. As their name suggests, the RO associated with a PRM is long-lived, has a state as any object and the operations it executes are dictated by the messages it receives from its proxy. To this extent the RO can be defined as an actor. The proxies of PRMs are called Persistent Proxy (PP) and from the user perspective they behave almost identically as regular ParaPy objects: they feature the same slots than their associated remote object, which can be accessed using the dot notation. Behind the scenes, the slot call is transferred to the remote object which evaluates the slot and send back the returned value. The computational effort is therefore delegated to the WP where the remote object resides. The value is then cached by the proxy object in the main process. Intermediate slots of the remote object that get evaluated during the requested slot evaluation are cached by the remote object but not transferred to the proxy. If the returned value is a ParaPy object, which is the case for every Part slot but can also be the case for Attribute or Input slots, a custom serialization is sent to the proxy, which instantiates a proxy object in the main process bound to the newly created remote object.

PARALLELIZATION API

Using the dot notation to retrieve a slot value on a proxy will block the calling process execution until the slot is evaluated by the RO and the value sent back to the proxy. To evaluate slots from PRMs asynchronously, proxy objects have an additional method called `eval_async()` which takes as argument the name of the slot to evaluate. When called, it triggers the evaluation of the slot by the RO and returns directly, such that the calling process can continue performing operations.

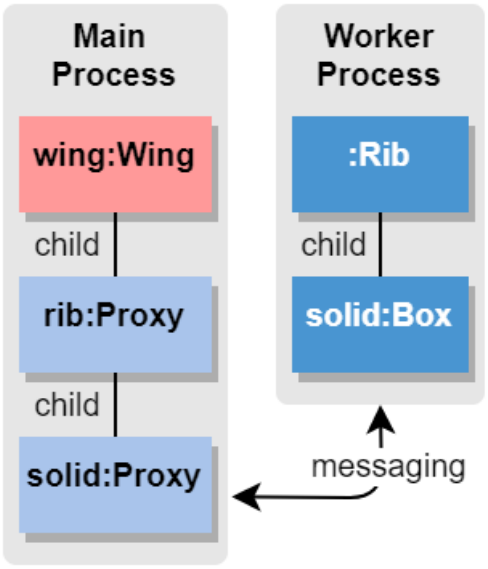
An alternative implementation was envisioned using futures. Futures are programming constructs first introduced by Baker and Hewitt [**?**] to represent the result of an asynchronous operation. Future objects are directly returned by asynchronous function calls. The future object itself does not hold the value returned by the call, but can be described as a reference to this value. Typically, futures implementations include a blocking method to retrieve the value. In Python, the value can be retrieved using the `result()` method of `Future` instances or by using the `await` syntax. The main drawback of this approach is the refactoring effort required to make it compatible with existing ParaPy models and the rest of the ParaPy ecosystem, as values and not future objects are expected when accessing a slot using the dot notation. This is the reason why the implementation with the blocking dot notation and `eval_async()` method was selected.

DEPENDENCY TRACKING AND CACHING

As introduced above, only the accessed slots of a proxy are cached in the main process. Intermediate slots of the RO on which the slot of interest depends are cached by the RO in the WP. The slots of the RO are dependency tracked as any ParaPy object.

```python
class Rib(Base):

    position = Input()

    @Part
    def solid(self):
        return Box(1, 1, 1,
                   position=self.position)


class Wing(Base):

    oml = Input()
    position = Input()

    @Part(remote=True)
    def rib(self):
        """PRM definition"""
        return Rib(position=self.position)


if __name__ == "__main__":

    # the runtime is initialized
    start_distributed()

    rib = Wing().rib
    solid = rib.solid
```

(a) A PRM definition.



(b) View of the objects and processes of interest.

Figure 3.2: A PRM definition (left) and the representation of the created objects (right). The `rib` proxy and its corresponding RO are respectively called root proxy and root RO.

```python
class Foo(Base):

    @Part(remote=True)
    def boxes(self):
        return Box(1, 1, 1,
            quantify=2)


if __name__ == "__main__":

    # the runtime is initialized
    start_distributed()

    foo = Foo()
    foo.boxes[0].eval_async("volume")
    foo.boxes[1].eval_async("volume")
    vol1 = foo.boxes[0].volume
    vol2 = foo.boxes[2].volume
```

```python
class Foo(Base):

    @Part(remote=True)
    def boxes(self):
        return Box(1, 1, 1,
                   quantify=2)


if __name__ == "__main__":

    # the runtime is initialized
    start_distributed()

    foo = Foo()
    future1 = foo.boxes[0].volume
    future2 = foo.boxes[1].volume
    vol1 = await future1
    vol2 = await future2
```

Figure 3.3: Parallel evaluation of two PRMs using the current implementation (left) and the alternative implementation using futures (right). In both implementations, lines 15 and 16 trigger the evaluation of the `volume` slot of each ROs, which are then evaluated in parallel and retrieved lines 17 and 18.

To preserve the model integrity, the dependency tracking mechanism of ParaPy has been extended to operate in a distributed environment such that slot dependencies can span across processes. When an Input of a PRM gets invalidated in the main process, the information is dispatched to the RO which performs the invalidation and informs back its associated proxy of the slots to invalidate.

PRMS AND GUI

ParaPy features a GUI to display and interact with KBE models. GUIs are particularly important in parametric studies since they provide near instant feedback to the user over design parameters changes. They offer a convenient way to explore the design space and help stirring the creative thinking of engineers. To this extent, PRMs have been implemented in such a way that they are fully compatible with ParaPy's GUI: they can be visualized and inspected as any regular ParaPy object.

In addition to being able to visualize and inspect PRMs as any regular objects, the GUI has been refactored to automatically take advantage of PRMs present in the model to evaluate their geometries in parallel when displaying them. A GUI centered KBE application can therefore benefit from parallelization speedups solely by declaring PRMs (if applicable), without the need to use the parallelization API inside the KBE model definition.

3.2.2. TRANSACTIONAL REMOTE MODELS

PRMs are interesting when several design iterations will be performed involving the RO. However KBE models can also be used in a more linear fashion, especially in MDO. In some use cases, keeping the ROs alive in the WP once the slots of interest have been evaluated and accessed in the MP is not useful as no other iterations will be performed with different inputs, or memory space needs to be reclaimed to perform other evaluations in the model. For these use case, TRMs have been introduced. The main concept of TRM is that their RO is instantiated to compute a specific subset of its slots called *outputs* and once they are evaluated and fetched back in the MP, the RO is deleted from the WP. In that way, the evaluation of the RO and its outputs is similar to the one of a task. The proxy object of TRMs provides a dependency tracked reduced view of the RO, constituted of only its Inputs and outputs slots and the dependency relations between them. If an output of the proxy gets invalidated, re-accessing it with the dot notation will trigger the creation of a new RO in the WP to compute its value. A TRM definition is shown in figure 3.4. The only difference with a PRM definition is the `outputs` argument of the `@Part` decorator. The output names must correspond to Attribute slots of the RO. The following subsections will provide more details about the behaviour of TRMs.

PARALLELIZATION API

The evaluation of a remote object's outputs can be triggered asynchronously using the proxy's `eval_async()` method. This method is different from the PRM's one since it doesn't take any argument. Indeed, the developer does not choose which outputs get evaluated: they all get evaluated. If the RO has already been deleted due to a previous fetch of its outputs AND the proxy has invalidated outputs, a new RO will be instantiated to evaluate the invalidated outputs.

DEPENDENCY TRACKING

The dependencies between the inputs and outputs of the RO are traced in the remote object and returned when the outputs are fetched. This means that when an Input of proxy is invalidated in the main process, only its dependent outputs get invalidated, and the other output slots can still be accessed without re-instantiating a new RO.

```
1  class Wing(Base):
2      ...
3
4      @Part(remote=True,
5            outputs=["weight", "shape"])
6      def ribs(self):
7          Rib(quantify=3,
8              oml=self.oml
9              offset=self.offset)
10
11
12 if __name__ == "__main__":
13     distributed.start()
14
15     wing = Wing()
16     for rib in wing.ribs:
17         rib.eval_async()
18     for rib in wing.ribs:
19         rib.weight
20         rib.shape
```

Figure 3.4: TRMs declaration and use.

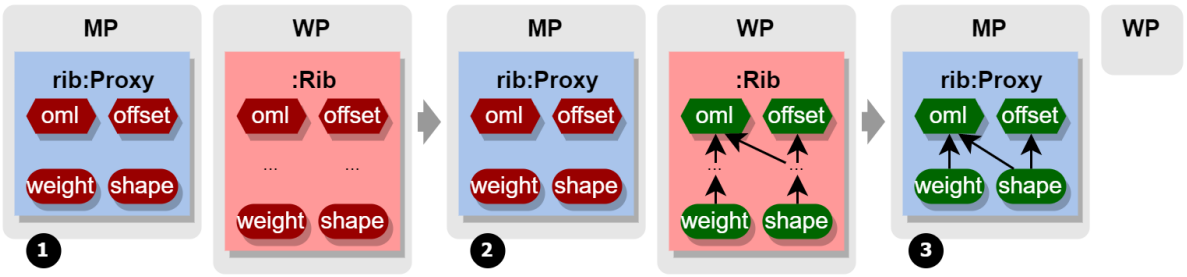


Figure 3.5: Illustration of the proxy's and RO's flow when using TRMs. Black arrows read "depends on". Sharp angled slots represent Input slots, and rounded ones Attributes. Red are invalidated slots and green slots that are evaluated.

EXPLAINING WITH AN EXAMPLE

As TRMs provide a new and unfamiliar behaviour to KBE developers, an example of its use will be detailed. When a TRM is accessed for the first time in the MP, its proxy object is returned and a first RO is instantiated in a WP. This corresponds to line 16 of figure 3.4 and stage 1 of figure 3.5. The `eval_async()` method can then be called on the proxy object to evaluate its output slots asynchronously (line 17 and stage 2). Then the outputs can be fetched using the dot notation. Note the all the outputs and the dependencies between outputs and Inputs are fetched and the RO is deleted the first time the dot notation is used on an output slot (line 19 and stage 3). The outputs value that are not accessed are cached in the proxy. Therefore, line 20 , the returned `shape` value comes from the cache.

3.3. CLOUD

In the last decade, cloud computing has contributed to the rise of Software-as-a-Service (SaaS) [**?**]. SaaS is a licensing model for software developers in which the software is hosted on a cloud platform and made accessible through web technologies to the customer, instead of being installed and run on the customer's computer. KBE applications also follow this trend, as it makes it easier to share and use them. ParaPy offers its own SaaS platform, the ParaPy cloud, to run ParaPy KBE applications in the cloud and access them through a web-based GUI. ParaPy cloud is constituted of a cluster, where the applications are run, and an interface to manage users and applications called

ParaPy Hub [**?**]. A reflection is therefore required on how to leverage cloud technologies to run distributed ParaPy applications in the most efficient manner. To this extent, this work proposes three cloud architectures to run distributed ParaPy applications, discusses their advantages and technical challenges and eventually compares their performances. These architectures are grouped in two different approaches to apprehend cloud computing: the single-user and multi-user approaches.

3.3.1. THE SINGLE-USER APPROACH

In this approach, cloud computing is solely considered as a way to access a "bigger machine" - a machine with more computing power than a regular desktop - to run distributed ParaPy applications. The corresponding architecture is very basic: the distributed application is run on a single (large) VM in the cluster. The VM's size is chosen as a trade off between computational power and cost by the user. The more cores the VM will have, the more the application can be parallelized (increased number of WPs) but also the more the VM will cost. The chosen VM is added to the cluster when an instance of the application is started and removed after use. The main advantage of this architecture is that all the WPs of the distributed application will run on the same machine, thus the communication overhead is minimum. This approach also has shortcomings. Acquiring a new VM usually takes time (around 5 minutes), and this overhead will be present each time a user wants to run an application. This architecture also inhibits one of the most important feature of cloud computing: elasticity. The number of RMs present in an application can vary while the application is running, and therefore so can the number of WPs. However, the size of the VM cannot, which can result in under or overused machine.

3.3.2. THE MULTI-USER APPROACH

The objective of this approach is to leverage the cloud in a more efficient manner by sharing infrastructure between different distributed applications running in the ParaPy cloud. Before continuing, a distinction will be made between two terms that will be used in the next discussion: applications and application instances. A KBE application is the application definition (viz. the application code), for example an application "Aircraft Designer" for designing aircrafts. An application instance is a running instance of a particular application, and multiple instances of the same application can be running simultaneously within ParaPy's cloud.

Concretely, sharing infrastructure means that WPs from distinct application instances could be allocated on any node in the cluster, as illustrated by figure 3.6.

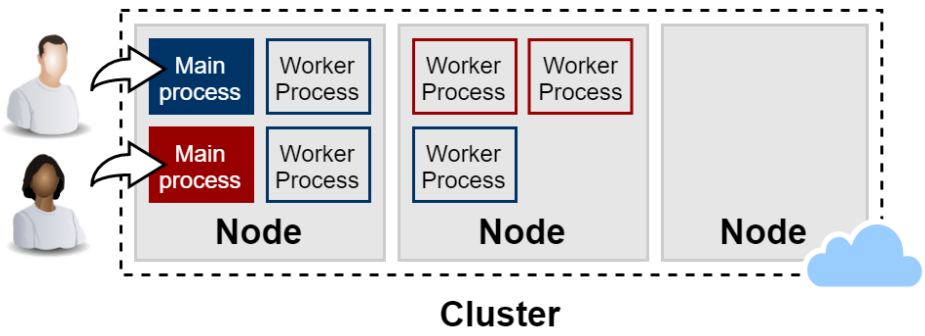


Figure 3.6: The multi-user approach. Colours represent different application instances.

This approach would solve the node (VM) creation overhead and elasticity issues of the single-user approach in the following manners:

- Node creation overhead: as applications are distributed over multiple nodes, the cluster can

be constituted of similar, smaller sized nodes. The cluster can be configured to always have spare resources (nodes are added if the cluster starts to become full, and removed as the utilization decreases) to directly accommodate new application instances. As any application can be instantiated on the spare resources, this mitigates the cost of always maintaining such spare resources.

- Elasticity: the cluster can be scaled up and down to automatically adapt for the total workload. This allows to save up costs for applications for which the number of RM varies during their lifetime.

The foreseen downside of this approach is the communication overhead. In contrast with the precedent architecture, messages will transit through the cluster's network instead of being passed between processes running on the same machine.

The complete cluster infrastructure and configuration to run multiple distributed ParaPy applications and scale the cluster to always maintain spare resources as described above is out of scope for this thesis and won't be implemented. This work will focus on evaluating the parallel speedups that would be achieved on such an infrastructure to assess its potential. For this matter, two cloud architectures will be proposed to spawn WPs from a distributed ParaPy application across a shared cluster. When evaluating each architecture, the assumption will be made that the cluster always has enough resources (nodes) to accommodate for the number of WPs the application will create.

TWO ARCHITECTURES FOR THE MULTI-USER APPROACH

This approach introduces some challenges. ParaPy applications are packaged into Docker images in order to be run in the cloud. A consequence from the RM's implementation, which will be explained in chapter 4, is that WPs created by an application instance must run inside a container of the application's image. Each time a WP of an application is allocated to a new node, the image of this specific application has to be pulled to the node, which could result in an important overhead during the WP creation. However, the largest layer in applications' images is the base ParaPy layer, common to every application images. For example, the image of the use-case RH application weights 2.6GB, in which the ParaPy base layer counts for 2.2GB, thus the application specific layer is only 400MB thin. Docker is smart enough to only pull the layers that are different between images. The assumption is made that the overhead derived from image pulling can be resolved or optimized, and that the solving of this challenge is out of scope for the current work.

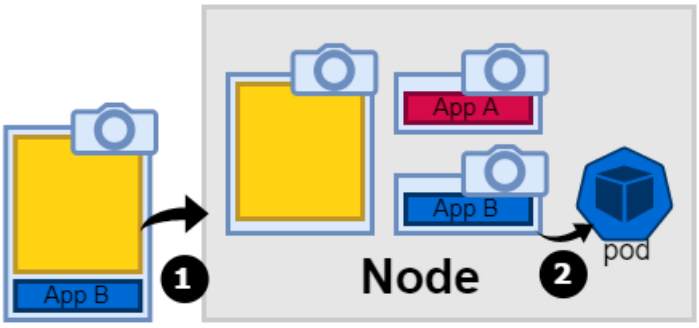


Figure 3.7: Pulling a Docker image on a node and creating a Pod from it. The yellow layer represents the common layer to all the applications images, the red and blue ones represent the application specific layers. During the tests, pulling the image (stage 1) is assumed to be instantaneous.

From this assumption, a first architecture can be proposed, called B1. In this architecture, a Kubernetes Pod is created for each WP. As Kubernetes Pods are wrappers around Docker containers (see section 2.2.2), this means that each WP is running in its own container.

In architecture B1, a Kubernetes Pod (and thus a container) has to be created for each WP. Even if containers are considered as lightweight processes, their creation time introduces an overhead when spawning new WPs. Ideally, there should be no creation overhead when spawning WPs, besides the creation time of the Python process itself. Therefore, a last architecture is implemented, B2, where a single Pod is present on all the nodes, and this Pod can contain multiple WPs. The additional assumption for this architecture is that a solution can be found to remove the Pod creation overhead when spawning new WPs of a distributed ParaPy application on a shared cluster. Some suggestions can already be made on how to achieve this:

- A cluster is dedicated to a specific application. This means that only instances of this specific application will run on the cluster. This would be plausible if popular KBE applications are developed that have a sizable user base. In this case, WPs from any of the instances could run in the same Pod, as they all rely on the same Docker image.
- The images from the different applications are optimized such that the application specific layer is very thin, and each node runs a Pod (and thus a container) for each application's image ready to spawn new WPs.
- WPs do not have to run from within a Docker container that is an instance of their application's image. This could be achieved for generic ROs, that are common to every ParaPy applications, such as geometry objects from the ParaPy library.

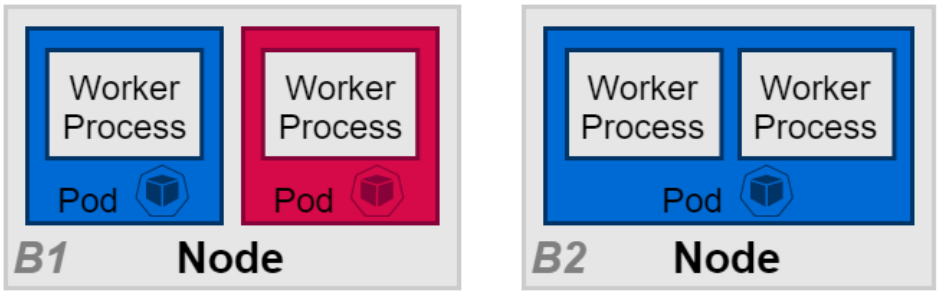


Figure 3.8: Architecture B1 (left) and B2 (right).

3.4. ASSESSMENT OF DISTRIBUTED PARAPY

To assess the performance improvements introduced by the use of the extended Distributed ParaPy SKD, different tests have to be run on real KBE applications from the industry. In this work, both PRMs and TRMs will be demonstrated on the same application, the RH hull design application developed jointly by ParaPy and engineers from the Royal Huisman shipyard. This application is representative of KBE applications developed with the ParaPy SKD. One of its advantages is that it presents a wide range of utilization, making it possible to define tests for both PRMs and TRMs. Furthermore, the hull structure of a sail yacht has similar features to the one of an aircraft.

3.4.1. THE ROYAL HUISMAN APPLICATION

This section will provide a high level view of the RH application's components and design workflow, as well as the elements of the hull structure KBE model it is based on. Then, the tests defined to assess PRMs and TRMs will be presented.

The RH application generates the structure of a hull based on a shape retrieved from an external file. The user can visualize the structure and tune its configuration through a set of tools provided in the GUI. Once the user is satisfied with a configuration, Finite Element Method (FEM), weight and

collision analyses can be performed on the hull structure to assess its characteristics and quality. The custom GUI of the application is shown in figure 1.1. The application is also often used in a more linear fashion by the RH engineers, directly generating the hull structure from a predefined configuration to perform the analyses. These two utilization workflow are summarized in figure 3.9.

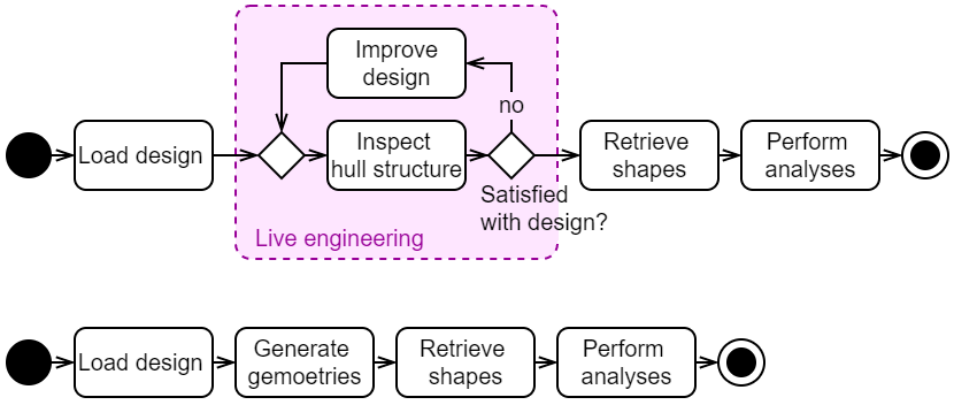


Figure 3.9: The two different workflows defining the utilization of the RH application: iterative (top) and linear (bottom).

The application is based on a KBE model of the hull structure, named the *construction modeler*. This model generates the geometries of all the structural elements of the hull. The structure can be broken down in three main groups of elements; the frames, beams and stringers. The construction modeler present multiple parallel regions. The main ones are the frames and the stringers. This work focus on the frames, as their generation is the most time expensive of the two. The frames are transverse structural elements of the hull similar to ribs in a wing or the fuselage's frames of an aircraft. The hull is constituted of 42 frames, which are instances of the `Frame` class. They can be visualized in figure 3.10.

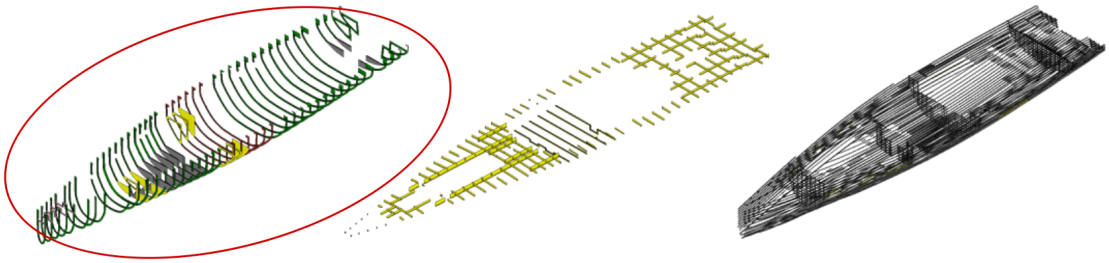


Figure 3.10: Anatomy of the hull structure as generated by the construction modeler. From left to right: frames, beams and stringers.

3.4.2. TEST CAMPAIGN

As introduced above, the application can be used in two different workflows, which will be referred from now on as the *iterative* and *linear* workflows.

In the iterative workflow, the user will visualize the frames through the GUI and iteratively change the frames' configuration until a satisfactory design is obtained. When the user is satisfied with a design, he can move on to perform the different analyses. PRMs are particularly suited for this workflow, due to its iterative nature and the fact that the parallel objects have to be displayed by the GUI.

In the linear workflow, the complete hull geometry is generated from a predefined configuration and directly used for subsequent analyses. The analyses only require the shape representation of the hull elements in order to be performed. The geometries of the different elements of the hull model are rather simple and therefore their generation cannot be described as memory heavy. For example, once their geometry is generated the complete set of frame objects weights 261MiB, which is low compared to the memory standards of current desktop computers (around 16GiB). Despite not being representative of the memory greedy KBE models, the frames can still serve to demonstrate the saving up in memory space induced by the use of TRMs, by defining as output the shape representation of a frame's geometry. This way, the proxy objects will only contain the shape representation of the frame, while the RO containing the "heavy" geometry objects required to produce the shape is destroyed.

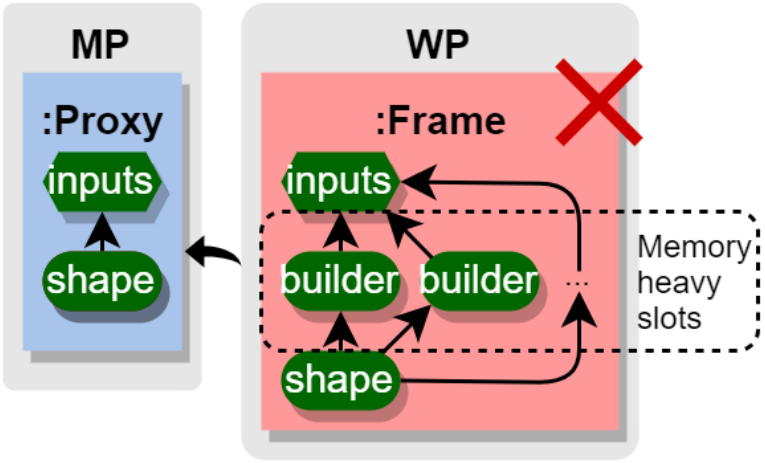


Figure 3.11: Simplified representation of how TRMs are used to parallelize the frames in the linear workflow to save up memory space. The shape representation required by the subsequent analyses is defined as *output*, and the unnecessary intermediate slots counting for most of the total size are deleted with the RO.

The test campaign will therefore be divided in two stages: tests for the PRMs and test for the TRMs. The next sections will detail each stage.

TESTS OF PRMS

This stage comprises two tests and will assess the effect of using PRMs for the frames on a typical design session through the GUI. These tests will be performed on different environment which will be detailed in section 3.4.2 and 3.4.2.

The tests consist of measuring the duration when performing two operations representative of a design session, with parallelized and serial frames, and compute the **speedup**. The tests will be run for multiple configurations that will be detailed in section 3.4.2 and 3.4.2. For each configuration, the tests will be run 3 times and the duration averaged over these 3 runs before computing the speedup. These operations and why they are representative of a design session are listed below:

OP1 **Frames creation and visualization:** this is the first logical action performed by a user when starting a design session. It is triggered when the user "double click" on the Part slot defining the frames for the first time. This action will successively create and initialize the WPs and ROs, and generate the frames' geometries and display them. When PRMs are used to define the frames, the generation of the frames' geometries will automatically be parallelized by the GUI. This operation will induce several initialization overheads that will be detailed in section 4.2.

OP2 **Position update:** this operation consists of changing the position of all the already displayed frames. This operation is proposed in the application in the form of a button to "equi-space" the frames. Clicking it changes the positions of the frames such that the space between them is the same and regenerate their geometries. This operation will be performed after OP1, to emulate a typical design session. The objective of this test is to evaluate the speedup obtained for a design change resulting in the regeneration of all the frames' geometries once the ROs have been initialized.

The tests are summarized in table 3.1. Note on the prerequisite of T1: evaluated inputs on master side means that the values that will be returned by the child rules defined in the frames Part slot definition are already evaluated, but does not mean that these values are already sent to the ROs (see section 4.2.2).

Test name	Objective	Prerequisite	Steps	Performance Indicator
T1	Evaluate the speedup when performing operation **OP1**	1. Application is running 2. The inputs of the frames are evaluated on MP side	1. Time **OP1** with serial frames 2. Time **OP1** with PRM parallelized frames	Speedup
T2	Evaluate the speedup when performing operation **OP2**	1. **OP1** has been performed	1. Time **OP1** with serial frames 2. Time **OP1** with PRM parallelized frames	Speedup

Table 3.1: Tests to be performed to assess the increase in performance brought by PRMs.

LOCAL TEST ENVIRONMENT FOR PRM TESTS

Tests T1 and T2 will in a first place be run on a multicore laptop, for different number of cores. For each number of cores, the maximum number of WPs used to parallelize the frames will set to the same value as the number of cores, to avoid over subscription. The laptop has an AMD CPU with 8 physical cores and a maximum frequency of 2.90GHz per core. The number of cores is controlled using Windows' *msconfig* utility which allows to reduce the number of cores accessible by the OS, and thus emulates a machine with less cores but with the same computing power per core.

CLOUD ENVIRONMENT FOR PRM TESTS

Tests T1 and T2 will be run in the cloud for the 3 architectures presented in section 3.3, on the Microsoft Azure cloud platform. Azure proposes different VM families and the Dasv5 family [**?**] has been chosen to run all the series of tests. These VMs utilize AMD's 3rd Generation EPYC 7763v processors that can achieve a boosted maximum frequency of 3.5GHz. The reason for this choice are the following:

- The VM family should be targeted toward CPU intensive applications.
- The VM family should offer VMs with number of cores varying from 2 to 32 (4 to 96 multi-threaded cores [**?**]).
- The VM family should be available in European datacenters.

For the single-user approach, the two tests will be run for a number of WPs ranging from 2 to 42. Each run will be run on a VM with (if possible) the same number of cores as the number of WPs.

The multithreaded cores present on the Dasv5 family are counted as a half core.

Number of WPs	2	4	8	16	32	42
VM	D4as_v5	D8as_v5	D16as_v5	D32as_v5	D64as_v5	D96as_v5

Table 3.2: VMs used to assess the single-user approach.

For the multi-user approach (architectures B1 and B2) the two tests will be run on a cluster of 8 nodes of 4 cores each (**D8as_v5** VM), so a total of 32 cores. The tests will be run for a number of WPs ranging from 2 to 32. The size of the cluster will remain the same for every number of WPs.

For each architecture, the cluster will be configured in such a way that the assumptions from section 3.3.2 are verified. As a reminder, this means that for architecture B1 the application Docker image will be pre-pulled on each node before the tests are run, to avoid the image pulling overhead. For architecture B2, a specific Pod will be running on each node ready to spawn new WPs before the application is started.

TEST OF TRMS

TRMs are currently still at the prototyping phase and therefore will undergo less extensive tests than the PRMs. Actually, this stage of the test campaign consists of a single test, that was already partially introduced above. The frames will be parallelized using TRMs, with their shape representation defined as output. The metric of interest here is the ratio r_{mem} between the memory size of the set of frames once their shapes have been evaluated with the non-distributed application and the size of the frames' proxies once the shapes have been retrieved using TRMs. The size of the frames will be measured by computing the difference between the memory size of the application's process before the shapes evaluation and after. Let S_{local} be the size of the set of frames when evaluated locally and S_{proxy} the size of the proxies once the shape has been fetched, then

$$r_{mem} = \frac{S_{local}}{S_{proxy}}$$

The size of the empty WPs won't be included when computing the size of the proxies, as they are supposed to be configurable such that they are killed once no more ROs are left (not implemented yet).

As the only metric of interest for this test is the memory ratio which does not depend on the environment the test is run in (due to the low memory requirements of the frames), this test will be run on a regular laptop computer.

Test name	Objective	Prerequisite	Steps	Performance Indicator
T3	Assess the saving in memory consumption when evaluating the frames shapes with TRMs	1. Application is running 2. The inputs of the frames are evaluated on MP side	1. Get mem. size of process 2. Evaluate the shapes of the serial frames 3. Get mem. size of application's process 3. Restart app 4. Get mem. size of process 5. Evaluate the shapes of the TRMs frames & fetch them 6. Get mem. size of application's process	r_{mem}

Table 3.3: Test to assess the memory improvement from using TRMs.

4

Distributed ParaPy Implementation

4.1. Architecture

A distributed ParaPy application is divided in different processes: the main process where the application's script is run, a message broker, a scheduler and several WPs. An overview of these components is given in figure 4.1. To ensure a proper functioning of a distributed ParaPy application, the developer must enclose the application script in the `if __name__ == "__main__":` guard and before anything else initialize the runtime environment by calling the `distributed.start()` function. This function call will start the broker and scheduler processes.

```
1  class Aircraft(Base):
2      ...
3
4
5  if __name__ == "__main__":
6      from parapy import distributed
7
8      distributed.start()
9
10     obj = Aircraft()
```

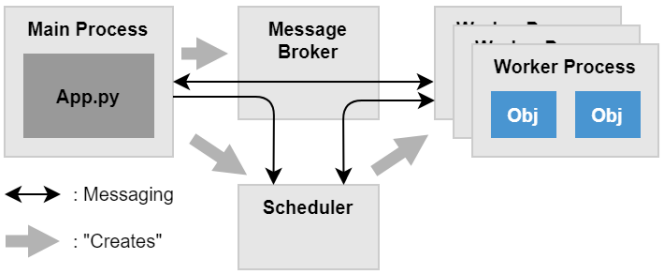


Figure 4.1: Typical setup of a distributed ParaPy application (left) and overview of the processes involved during the application lifetime and their interactions (right).

4.1.1. Message Broker

All messages in Distributed ParaPy transit through the message broker. The message broker was developed for this work based on the ZeroMQ messaging framework [**?**]. User friendliness, performance and the quantity of communication patterns available [**?**] are the reason why the ZeroMQ framework was chosen. The developed message broker acts as a message queue and enables the different communicating entities to send messages only by specifying the name of the recipient, without having to know the recipient's address. The main advantage of developing a custom message broker is the ability to create messaging patterns adapted to the application being developed.

4.1.2. WPs and Scheduler

WPs were introduced earlier, and will be further detailed in this section. A WP can contain multiple ROs, both persistent and transactional. The developer can define pool-like groups of WPs, called WP types and assign RMs to a specific WP type. The available WP types can be defined us-

ing the `distributed.Config` class. An instance of this class must be passed as argument to the `distributed.start()` function. A WP type can define the following attributes:

- **name**: the name of the WP type. This is how a RM declaration will reference this type. Each type must have a unique name.
- **cpu**: the CPU request of this type. Float value corresponding to the minimum number of CPU cores that must be reserved for each WP of this type. This attribute is only relevant when running on a cluster.
- **memory**: the memory request for each WP of this type. Again, this attribute is only relevant when running in a cluster.
- **max**: the maximum number of WP of this type. When the maximum number is reached, no more WP are created and new remote objects assigned to this type will be instantiated in an already existing WP.

The scheduler is responsible for creating the WPs and allocating the remote objects on the right WP. If the application is running in a cluster, the scheduler is also responsible for placing the WP on the right node. This section will only discuss the behaviour of the scheduler when the application is running on a single machine. When a RM is accessed for the first time, a message is sent to the scheduler process with the metadata required to create the RO, including type information about the object and the name of the WP type in which the RO should be instantiated. The object's type information consists of the fully qualified class name of the object (e.g. `'app.src.rib.Rib'`) such that the WP knows how to instantiate the object[1]. The scheduler will first look in the metadata for the WP type, and check if the maximum number of WPs of this type is reached. Then:

- If the maximum is reached, it will schedule the object on the less crowded WP of this type.
- If not, it will create a new WP and schedule the object on it.

The scheduler keeps track of the number of objects living in the WPs and uses this information to determine the less crowded WP of each type. The number of remote objects in a WP is decremented when the object associated with a TRM is fetched and garbage collected.

The initialization of a new WP is a time consuming task. This is due to the substantial import time of the required packages, such as ParaPy core, the geometry kernel library, etc. This overhead is around 2 seconds on a laptop with a 2.9GHz processor. Furthermore the import time of the ROs class and its dependencies can also incur an extra overhead. To avoid adding up the WP initialization overheads when accessing multiple RMs consecutively, the RM creation is asynchronous. The root proxy object is directly returned while the creation instruction is sent to the scheduler. Subsequent calls on the proxy (such as `eval_asyn()` or dot notation calls) will be queued until the RO is instantiated. The chronology of a PRM creation is presented in figure 4.2. In this sequence diagram, a PRM is accessed, and then one of its slots' value is retrieved using the dot notation on its proxy.

[1] An alternative solution would have been to serialize the class definition (at least until the parent ParaPy class, which is most probably not serializable due to C++ extensions, and reconstruct the full class definition in the WP) and send it instead of the fully qualified class name of the object. The main advantage would be that the application source code would not have to be present in the environment where the WP is running (interesting when the WP is susceptible to run on a different machine, typically when an application is distributed on a cluster. In this case WPs could run in any Docker container with the ParaPy package installed). The main reason this implementation was not chosen is that the classes defined in a model can rely on compiled packages which would not be serializable making this solution less robust.

Instead of calling a slot, another RM could have been initialized in parallel (not represented due to the complexity of the resulting sequence diagram).

```
1  class Rib(Base):
2
3      @Attribute
4      def weight(self):
5          value = ...
6          return value
7
8
9  class Wing(Base):
10
11     @Part(remote=True)
12     def rib(self):
13         return Rib()
14
15
16 if __name__ == "__main__
17
18     distributed.start()
19
20     rib = Wing().rib
21     weight = rib.weight
```

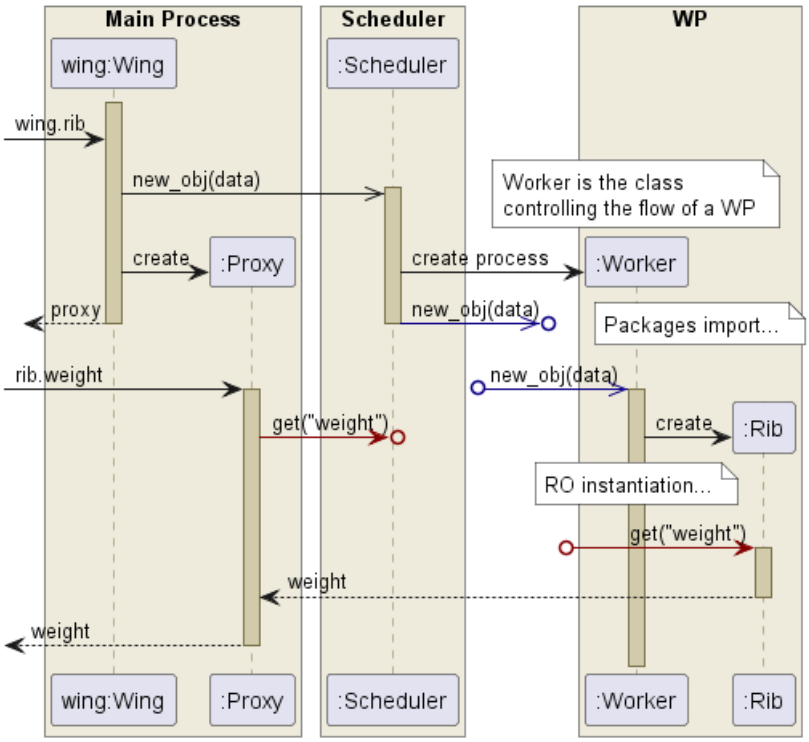


Figure 4.2: The sequence diagram (right) corresponds to line 20 and 21 of the code snippet (left). Arrows ending with a "o" indicate that a message is queued by the message broker since the recipient is not ready to receive messages yet. Arrows starting with a "o" indicate dequeued messages. Colours help relating queued and dequeued messages. Yellow boxes represent processes boundaries. The WP is actually created with the "create process" arrow, even if the size of the box suggests that it is present from the beginning.

4.2. PERSISTENT REMOTE MODELS

This section will lay out the inner functioning of PRMs. The objective is to decompose the operations taking place when interacting with PRMs into low level actions to be able later to finely breakdown and analyse the parallel run time of the developed constructs and provide relevant recommendations for further work.

PRMs are controlled through the API provided by proxy objects. Part of this API was already introduced earlier such as the dot notation to access a slot or the `eval_async()` method. Table 4.1 summarizes the most important components of the API that will be detailed in this section. These components are grouped by the entity consuming them.

An overview of the actions occurring during the different API calls is given by the activity diagrams of figure 4.3 and 4.4.

4.2.1. PROXY CREATION

PPs are instances of dynamically created ParaPy classes, they are created based on the class of their RO such that they feature the same slots. A custom serialization mechanism is added to any ParaPy objects living in a WP. When a ParaPy object living in a WP is serialized, only its fully qualified class name and optional slots value are actually serialized (the optional slots are drawables' shape slot, see 2). When the object is deserialized in the MP, a mechanism will first look if a proxy class has

Consumer	API component
Developer	Get slot value: dot notation. E.g. `value = proxy.slot` Evaluate slot asynchronously: `eval_async(name: str) -> None` Set slot value: python attribute setting syntax. E.g. `proxy.slot = 2`
GUI	Evaluate children shapes: `eval_drawables() -> None` Construct child tree: `construct_tree(names: List[str]) -> None`
ParaPy internals	Invalidate inputs: `invalidate_inputs(names: List[str]) -> None`

Table 4.1: API of PRMs. The consumer indicates the entity for which the API endpoint was originally developed. Endpoints usage is not limited to the consumer specified in this table, to the exception of ParaPy internals endpoints which should not be called by the developer nor the GUI.

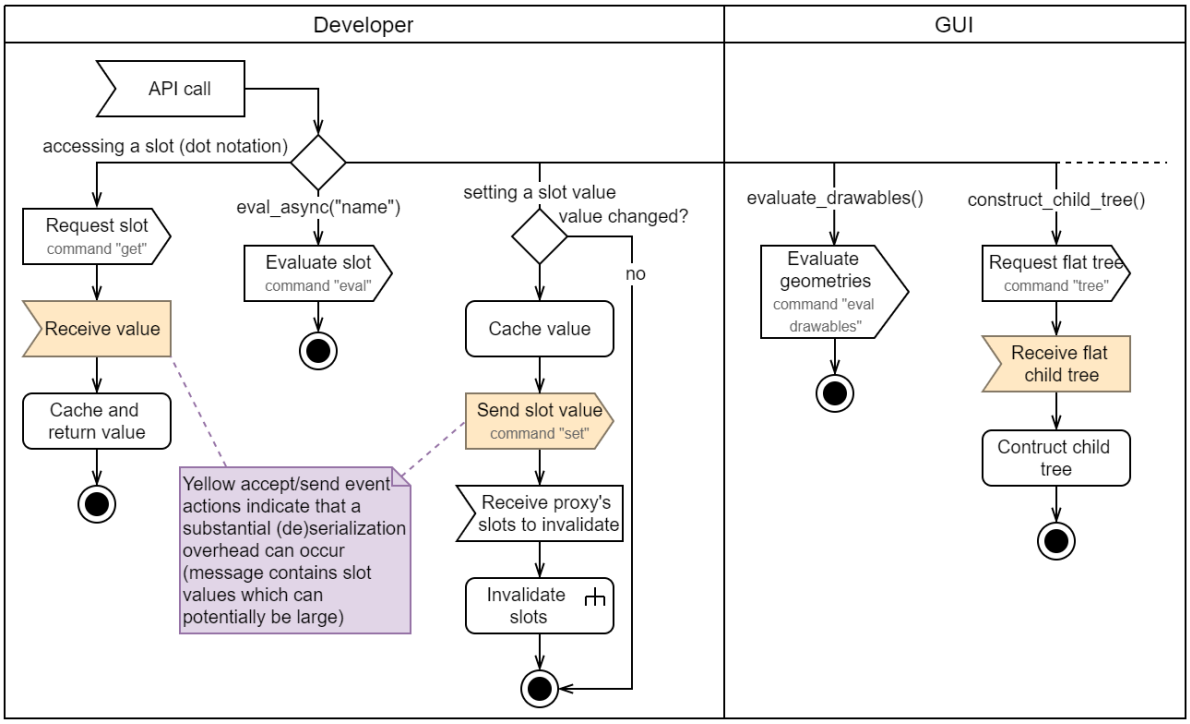


Figure 4.3: Activity diagram showing the actions taking place for each API call on a proxy object. The input invalidation is not shown here and will have its own diagram.

already been created for the RO's class. If yes, an instance from this proxy class is returned. If this is the first time that a RO of this class is serialized, a new proxy class is created for this class and cached.

4.2.2. INPUT EVALUATION

Chapter 3 introduced the API calls to evaluate slots asynchronously and retrieve their value, but it lacked important information about how this is achieved. The first question a reader might have is *"when are the inputs of a PRM evaluated and sent to the RO?"* Two different strategies for evaluating the inputs of a PRM have been implemented, called respectively lazy and eager inputs evaluation.

Before detailing further each strategy, lets first start with a reminder of how inputs are evaluated in ParaPy. Evaluating a slot of a ParaPy object can lead to one or more of the object's inputs to be evaluated, if the slot depends on them. For non-remote child objects, the default behaviour is to check if a child rule has been defined for this Input slot in the Part definition. Child rules in

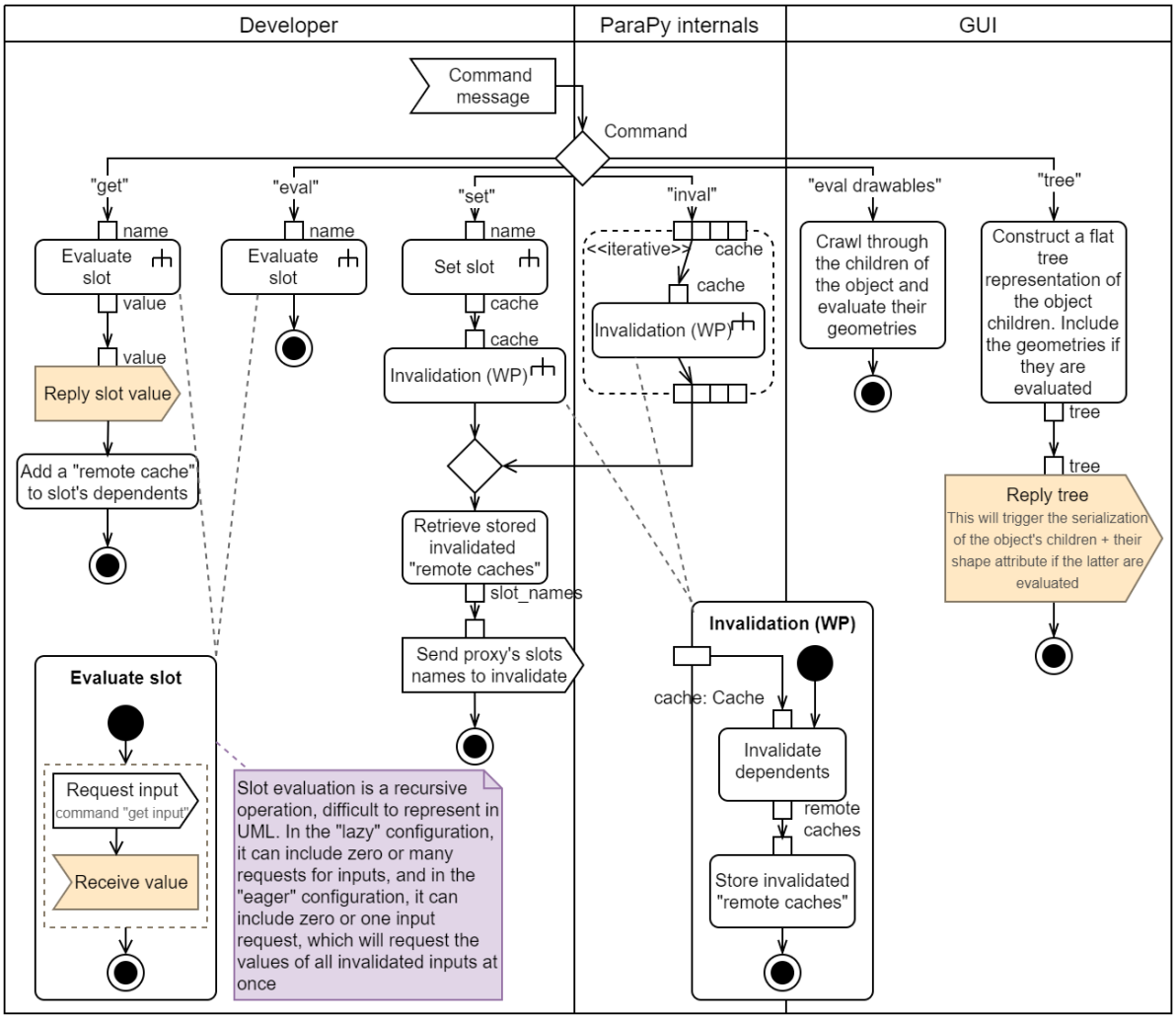


Figure 4.4: Activity diagram showing the control flow of a RO. Each command message received by the RO is the result from an API call on its associated proxy object.

ParaPy are the statements where a child Input slots values are defined as function of their parents' slots. For example, in figure 3.2a, the statement line 8 (`position=self.position`) in the `solid` slot definition is a child rule. If a child rule has been defined for this Input, it gets evaluated and the Input is set to the returned value. If no, then the default value of the Input is used (other methods of passing Input values from parent to child exists in ParaPy but are rarely used and haven't be implemented for PRMs). As dependencies are discovered and tracked at runtime in ParaPy, it is not possible to know in advance when accessing a slot from an object if it might depend on one of the object's Inputs.

As its name suggests, the lazy inputs evaluation strategy mimics the behaviour of traditional ParaPy's Part slots. The name of the inputs for which a child rule is defined are transferred to the RO at creation time. The evaluation of the child rule is requested by the RO when a remote slot evaluation requires the value of such an input. This means that ROs are capable of initiating a request in the direction of the MP, which implies that the MP should be listening for incoming requests from the RO for the whole duration of the remote slot evaluation (and therefore be blocked). This behaviour is incompatible with the parallelization objective of PRMs. For this reason, a specific thread is created for each PRM that listens for input evaluation requests from the RO. This thread is called

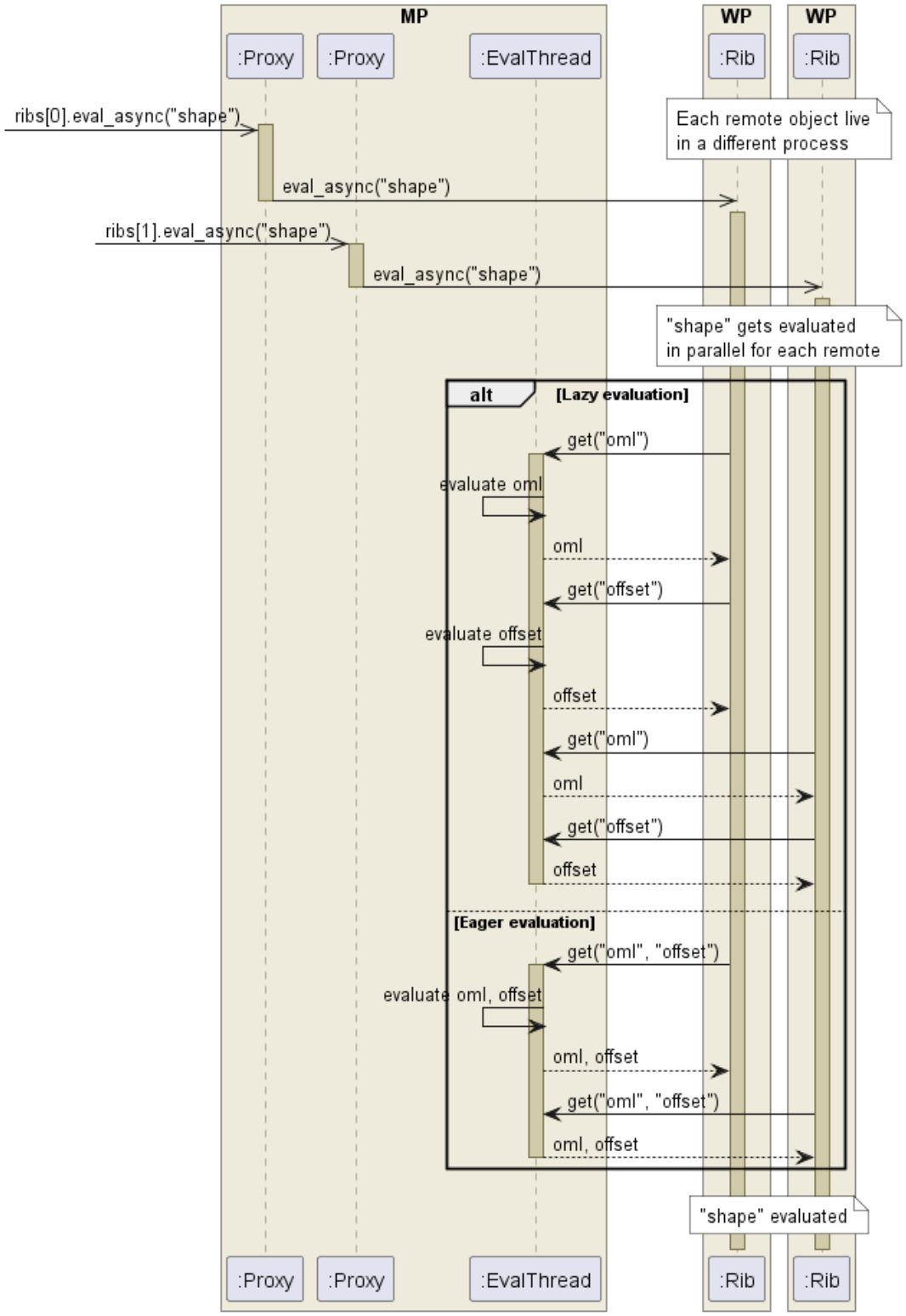


Figure 4.5: Sequence diagram of the lazy and eager input evaluations. The sequence corresponds to line 40 of figure 4.6, if the `ribs` PRM was defined with lazy or eager evaluation (for practical reason only two ROs are shown).

the Rule Evaluation Thread (RET). Internally, ParaPy uses thread local objects to track dependencies between slots which makes the caching and dependency tracking mechanisms thread safe.

The advantage of extending the lazy inputs evaluation of Part slots to PRMs is that only the required inputs get evaluated and requested by the RO. However, the consequence is that a request/reply cycle is performed for each requested input. In many use-cases, the slots of interest in PRMs will depend on every child rule defined inputs. For such use-cases, the eager input evaluation strategy was developed. If this strategy is chosen, each time an input of a RO is accessed the evaluation of all the child rules is requested at once by the RO (only if the corresponding RO's input is invalidated, to avoid sending unnecessary input values). With this architecture, a single request/reply cycle is required to evaluate the inputs. Lazy and eager input evaluation are represented in the sequence diagram from figure 4.5.

```
1  class Rib(Base):
2
3      oml = Input()
4      offset = Input()
5
6      @Attribute
7      def shape(self):
8          """Depends on oml and offset"""
9          return ...
10
11
12 class Wing(Base):
13
14     # wing outer mold line
15     oml = Input(...)
16     # offsets of the ribs
17     offsets = Input([...])
18
19     @Part(remote=True,
20           config=RemoteConfig(eager=True,
21                               worker="id"))
22     def ribs(self):
23         return Rib(quantify=4,
24                    oml=self.oml,
25                    offset=self.offsets[
26                        child.index
27                    ])
28
29
30 if __name__ == "__main__":
31
32     wrk = WorkerType(name="id",
33                      max=2, cpu=1.0)
34     config = Config(workers=[wrk])
35
36     distributed.start(config)
37
38     ribs = Wing().ribs
39     for rib in ribs:
40         rib.eval_async("shape")
```

Figure 4.6: Sequence of PRMs definition and parallel evaluation of a slot.

The sending of inputs to ROs feature some optimization for PRMs defined inside a Sequence. When defining a child rule for a Sequence, the value returned by the child rule can either be distinct for each member of the Sequence (it is said that the child rule is *child-specific*) or the same for all the members (*child-agnostic* rule). ParaPy's `child` object is used to define child-specific rules, as shown line 26 of figure 4.6. In this example, the `oml` child rule's value is the same for the 4 members of the Sequence, while the `offset` value is distinct for each. If multiple members of a Sequence are allocated to the same WP, a caching mechanism is implemented on WP side to only request once the value of a child-agnostic rule. This behaviour is illustrated in figure 4.7.

Another optimization targets the serialization of child-agnostic rules' value. Serialization is time-consuming and to avoid repeating this redundant operation for child-agnostic rules the serialized value returned by the rule is cached in the MP when a first RO requests it is and reused for all the members. If the value is invalidated the cached serialization is removed and will be recomputed for the next RO to request it.

It is important to note that the current implementation does not allow to pass ParaPy objects as Input to PRMs. PRMs Inputs (those who are defined through child rules) should be non-ParaPy serializable objects. If a developer really needs to pass a ParaPy object as Input to a PRM,

a custom serialization should be implemented for this particular object, by using either Python's `__getstate__()` and `__setstate__()` or `__reduce__()` *dunder* methods.

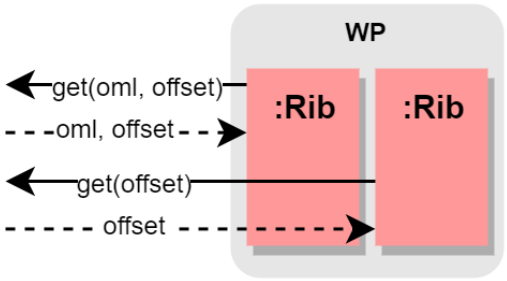


Figure 4.7: Pseudo sequence diagram showing the messaging optimization for inputs defined by child-agnostic rules (here depicted with the eager input evaluation configuration). The first RO to evaluate its `oml` input will request the value, and the second one will use the cached value (as long as it hasn't been invalidated in the meantime).

4.2.3. GUI INTEGRATION

ParaPy's GUI was extended to automatically exploit parallelization opportunities offered by PRMs.

GUI FUNCTIONING

The GUI displays *drawable* objects. Drawable objects are ParaPy objects that define a geometry representation, which is stored inside a slot called `TopoDS_Shape`. This slot will be referred as *shape* in the rest of this work.

The display of a ParaPy object by the GUI can be broken down in 3 steps. The first step performed by the GUI is to scan through the object and its children recursively to discover drawable children (step A). If the children are not instantiated yet or have been invalidated, this operation will instantiate them and it is said that the child tree of the root object is being constructed. Once the drawable children are gathered, their shape will be evaluated (step B). The last step is the tessellation of the shapes to transform them into displayable objects (step C). The main time-consuming step is the shapes' evaluation, since the generation of the geometries often involve complex computation. However, it is not the only time-consuming step. Scanning through the children can also be lengthy if some children are not instantiated. For example if the size of a Sequence depends on a slot whose evaluation involves complex computation, scanning through the members of the Sequence will require the evaluation of said slot and incur a delay. These 3 steps are summarized and illustrated in figure 4.8.

GUI PARALLELIZATION

A straightforward parallelization implementation would be to let the GUI process with the scanning step (A) and use the asynchronous slot evaluation API on the shape slot of all the collected drawable children that are also proxies. Then accessing the shape in step B would only fetch the already evaluated shape from the RO. However, such an implementation would be highly inefficient. First, in the case where the children are not instantiated the ROs' tree construction would not be parallelized for PRMs. Furthermore building the proxies' tree would require instantiating each child proxy one by one using the dot notation (`rootproxy.child1`, then `rootproxy.child1.child2`, etc...), when the full child tree could be instantiated in a single request/reply cycle. This is the reason why the API features methods dedicated to the GUI, to optimize the parallelization and display of PRMs.

In order to understand how the GUI parallelizes the display of PRMs, an example will be followed step by step. Let's suppose one wants to display a ParaPy object with two PRMs. On MP side, the GUI will start by scanning through the object's children recursively to find drawable objects (*get*

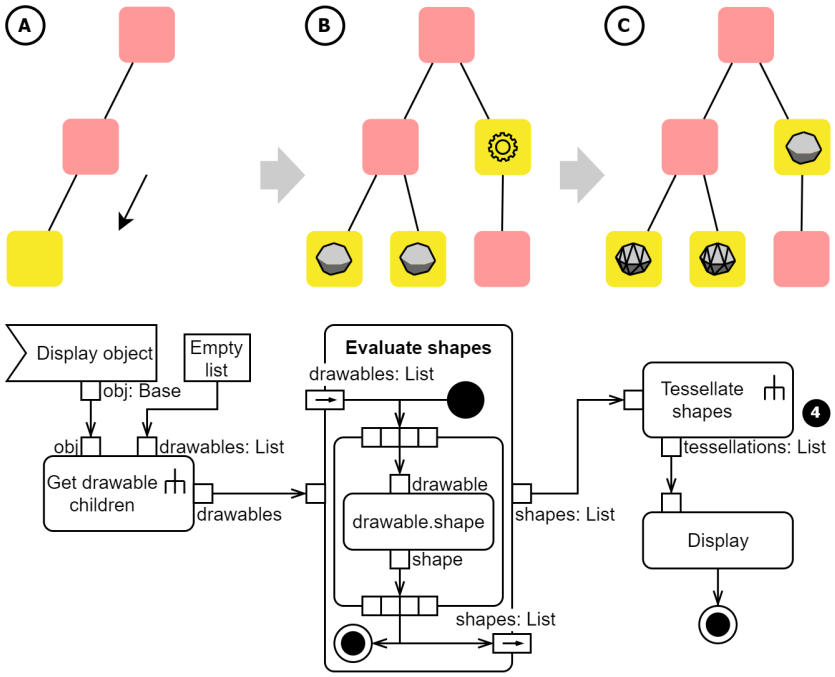


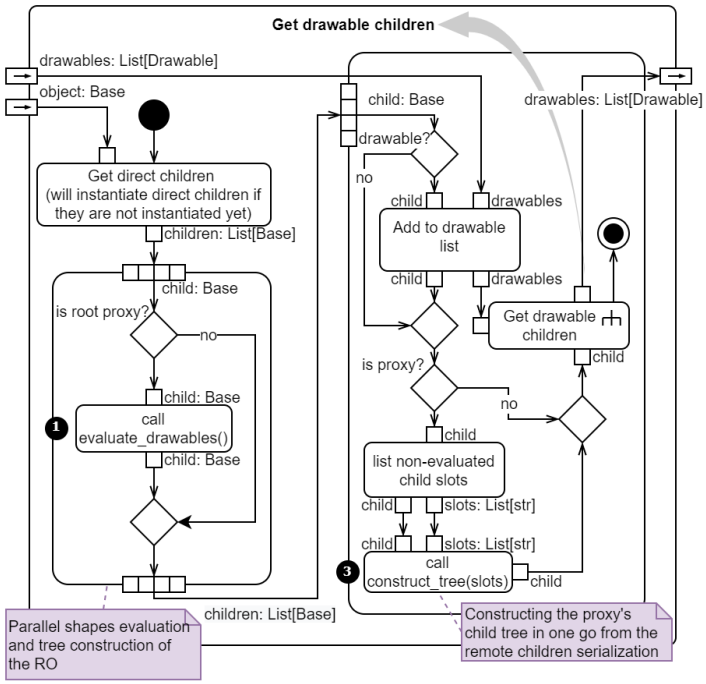
Figure 4.8: Activity diagram and schema of the three main steps happening during the display of an object by the GUI. The first activity scans through the object's children to find drawable objects (represented in yellow). Here the upper schema illustrates the tree being constructed, but the children could also already be instantiated (typically if the object has already been displayed and is being re-displayed after that a change in the configuration invalidated the model geometry). Then the shapes of drawable children are evaluated, and finally the shapes are tessellated in order to be displayed.

drawable children activity, see figure 4.9a). The first two children will be the root proxy objects. The GUI will therefore call the `evaluate_drawables()` method on each proxy (stage 1 of figure 4.9a and 4.9b). This API call is asynchronous, it triggers the scanning for drawables children of the RO and the evaluation of their shape in the WP. Therefore the child tree construction (if the RO's children are not instantiated yet) and the shape evaluation will be parallelized for the two PRMs. Then the GUI will check if the proxies have some non-evaluated direct child slots. If this is the case, it will call the blocking `construct_tree()` method on the proxy (stage 3). This is an optimization method. Its role is to reduce to the maximum the messaging between proxies and ROs. This methods request the flat tree representation of the RO's children and constructs the full child tree of the proxy from it. The flat tree representation contains the serialization of the RO children (which are de-serialized into proxy objects in the MP as detailed in section 4.2.1) and their parent/child relations. The serialization of drawable children will also contain their shape attribute which will be cached by their proxy [2]. Thus, the recursive calls of the *get drawable children* activity on the proxies' children won't require any more communication, since all the children are already instantiated in the MP. During the second display step (B) the GUI will access the drawable proxies' shapes through the dot notation, and the cached value will be returned. The last step is the tessellation of the shapes, which is not parallelized yet (stage 4).

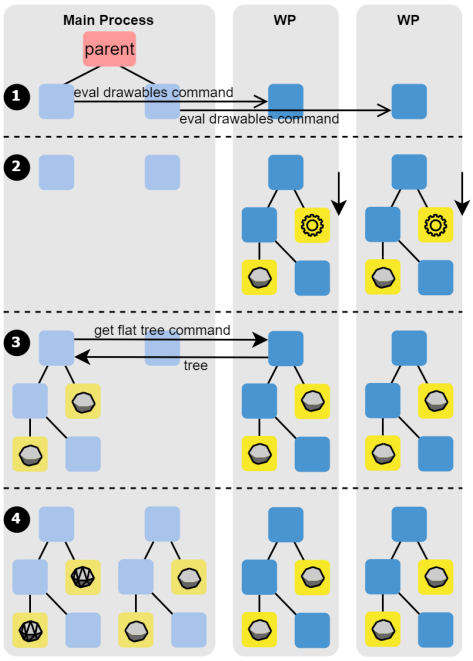
In the case where the proxy's children are already instantiated, stage 3 (tree construction) is omitted. This situation usually happens when an object has already been displayed, but a change to the model configuration has invalidated the shapes of the ROs. The consequence is that during the next GUI step (B), accessing the shape slot through the dot notation will result in a request to the RO as the shape value is not cached by the proxy yet (but already evaluated and cached by the

[2]This shape Attribute is the optional slot's value that a ParaPy object's serialization can contain

RO).



(a)



(b) Light blue boxes represent proxy objects living in the MP. Darker blue boxes represent the ROs. Yellow boxes are drawable objects.

Figure 4.9: Activity diagram of the scanning step (left) and pseudo sequence diagram representing the display of two sibling PRMs (right).

4.3. DISTRIBUTING THE RH APPLICATION

Despite the efforts in making the transition between a traditional ParaPy model to a distributed model seem seamless to the developer, a few changes had to be made to the model codebase to parallelize the frames. The changes mainly targeted the Inputs of the `Frame` class which defines the frames. Most of these Inputs expect ParaPy objects, and as introduced earlier ParaPy objects cannot currently be passed as Inputs for PRMs without defining a custom serialization. These problematic Inputs can be broken down in two categories: those for which a custom serialization is easily implementable and those who require a minor refactoring of the model architecture.

Developing a custom serialization for the first category was trivial. Most of them were simple geometry objects, and the only slot of interest for the `Frame` class was their shape representation (`TopoDS_Shape` slot). In this case, the custom serialization consisted of serializing the shape and recreating a new geometry object from this shape at deserialization time.

For the Inputs that could not be serialized, the model architecture had to be modified. This will be explained through an example. One of the elements of the hull structure is the fuel tank. The frames need a reference to the tank object in order to generate their geometries. In the application, the tank is passed to the frames as Input, however defining a custom serialization for the `Tank` class that would fit the needs of the frames was deemed difficult, and an easier solution was implemented. Instead of passing the tank object to the frames as Input, the `Frame` class is refactored to instantiate its own tank object. As `Tank` instances are not too heavy, having one for each frame instance instead of a single one in the model does not affect too much the application performance. It is important to note that this solution had to be found in a time when the author didn't have much knowledge about the functioning and architecture of the RH model, and that retrospectively some

```
def create(type_, shape):
    return type_(TopoDS_Shape=shape)


def reduce(self: Face_):
    shape = self.TopoDS_Shape
    return create, (Face_, shape)

Face_.__reduce__ = reduce
```

Figure 4.10: Custom serialization for the `Face_` class. When a `Face_` object gets serialized, the `__reduce__()` method is called, which returns a reference to a function to call at deserialization time with some parameters to pass. At deserialization time the `create()` method will be called which will instantiate a new `Face_` object with the same shape representation. This kind of simple serialization does not work for more complex geometry objects but was enough for most of the frames inputs.

more elegant and efficient solutions could have been implemented. This refactoring is illustrated in figure 4.11.

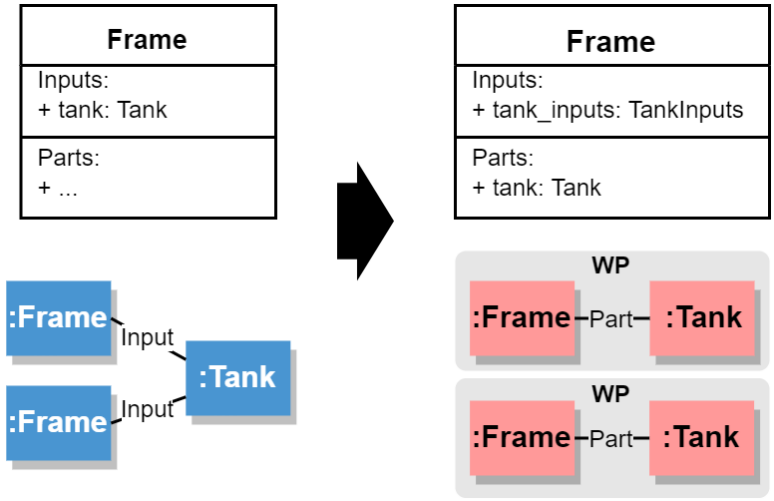


Figure 4.11: Example of the refactoring required to distribute the frames.

From now on, the original KBE model of the hull will be called model A, and the model refactored to allow the parallelization of the frames will be called model B. Even if the code structure has been modified, the embedded engineering rules have been preserved between both models and the resulting frames' geometry remains unchanged. The refactoring of the model incurred a penalty in run time for the operations targeted by T1 and T2. This penalty is summarized in table 4.2. The run time penalty won't be included in the computation of the speedups. The reason is that this penalty could be reduced with a smarter refactoring of the hull model if time allowed.

Operation	Run time - model A (s)	Run time - model B (s)	Runtime penalty
Frames creation and visualization	14.97	17.01	14%
Position update	13.67	14.240	4%

Table 4.2: Run time penalty incurred by the refactoring of the hull model to comply with PRMs requirements.

Eventually, the frames definition in the hull model is shown in figure 4.12. The frames are defined in a Sequence (`quantify=42`). Some comments can be made regarding the child-rules. The

only child-specific rule is the frame's position, all the other rules are child-agnostic. The serialized size of the position's value is 743B, and the total size of the child-agnostic rule is 12.2MB. The size ratio between the child-specific rule and the child-agnostic rules is of 6.1e−5, which can be considered negligible. This information will be relevant when analysing the results.

```
@Part
def frames(self):
    return Frame(quantify=42,
                 position=self.frame_positions[child.index],
                 deck_datum=self.deck_datum,
                 center_girder_datum=self.center_girder_datum,
                 floor_datum=self.floor_datum,
                 ship_rear_datum=self.ship_rear_datum,
                 hull_datum=self.hull_datum,
                 in_plane_v_dim=self.in_plane_v_dim,
                 cut_outs=self.cut_outs,
                 tank_datums=self.tank_datums)
```

Figure 4.12: The frames' definition in the refactored hull model (here declared as a regular ParaPy Part).

5

VERIFICATION OF DISTRIBUTED PARAPY

As this thesis is strongly oriented on software development, the verification methods chosen will differ significantly from the ones usually encountered in other thesis works.

The verification of PRMs and TRMs will be presented in section 5.1. Section 5.3 will present the verification of developed cloud architectures.

5.1. VERIFICATION OF PRMS AND TRMS

In software engineering, unit testing is the preferred method to verify a program. Unit testing is the action of checking the behaviour of small sections of a software to detect bugs at an early stage and avoid their propagation to the rest of the software [**?**]. Unit tests have been developed in parallel with the main software to ensure that PRMs and TRMs behave as expected. The testing framework used is the Python package `pytest` [**?**]. These unit tests are run regularly in order to detect breaking changes in the codebase.

Figure 5.1 shows a first series of test targeting the user APIs of PRMs. The code starts with the definition of the ParaPy class that will be used to conduct the tests. This class defines a PRM named `box` (line 5 - 7) that returns a ParaPy `Box` object. The first test `test_get()` is very simple: it checks that we can evaluate slots of ROs using the dot notation. It also checks that the mechanism for evaluating child rules of PRMs works (line 19). The second test, `test_async_evaluation()`, checks the proper functioning of the asynchronous evaluation of remote slots. First, the evaluation of a remote slots is triggered using the PRM' parallelization API (`eval_async()`). The status of the width Input of the root object is then checked (checking the status means checking if the slot's value is evaluated and cached). If width is evaluated (status is `True`), it means that the RO has evaluated its volume slot and requested the evaluation of the child rule. The next test `test_invalidate()` checks that the invalidation still works across PRMs. The last `test test_proxy()` checks that ParaPy objects are not directly serialized but instead return a new proxy object associated with them. Indeed, the `bbox` Part slot of a `Box` object returns an instance of the `BBox` ParaPy class. The `BBox` object gets instantiated in the WP, and a proxy object is returned in the main process. Similar unit tests have been developed for TRMs.

5.2. ROS AND WPS ALLOCATION

The ROs allocation to the right WP can be verified through logging. During the program execution, relevant events are logged in the system console, with a convenient formatting to track which pro-

```
class Foo(Base):

    width = Input(2)

    @Part(remote=True)
    def box(self):
        return Box(self.width, 1, 1)

    @Attribute
    def volume(self):
        return self.box.volume


def test_get(self, initialize):
    """Check that the remote evaluation works, as well
    as the child rules evaluation."""
    foo = Foo()
    assert foo.box.volume is 2
    assert foo.get_slot_status("width") is True

def test_async_evaluation(self, initialize):
    """Check that the asynchronous evaluation works."""
    foo = Foo()
    foo.box.eval_async("volume")
    time.sleep(4)
    assert foo.get_slot_status("width") is True

def test_invalidate(self, initialize):
    """Check that distributed invalidation works"""
    foo = Foo()
    foo.volume
    assert foo.get_slot_status("volume") is True
    foo.width = 1
    assert foo.get_slot_status("volume") is False

def test_proxy(self, initialize):
    """Check the serialization of ParaPy objects"""
    foo = Foo()
    assert isinstance(foo.box.bbox, Proxy)
```

Figure 5.1: Example of tests verifying the proper functioning of the user's API.

cess emitted the message. A simple model is developed with two RMs Sequences of 4 members assigned to different WP types (`worker_1` and `worker_2`) and run on a single machine. The model definition and the logs are shown in figure 5.2 and 5.3. On the left of the logs is the id of the WP that emitted the message (Worker-n). There is a total 4 WPs, which is consistent with the WP types definitions. The 4 `Cylinder` objects are distributed over WPs 2 and 3 and the 4 Box objects over WPs 0 and 1, which is also consistent with the RMs declarations.

5.3. VERIFICATION OF THE CLOUD ARCHITECTURES

This section will only cover the verification of the proper functioning of cloud architecture B1. The WP allocation of cluster architecture B1 is verified with the Kubernetes command line interface (kubectl) during one of the test runs, by running a command which lists the pods present in the cluster and the nodes they are running on. The relevant pieces of information of the run are:

- A PRM Sequence of 42 members is evaluated.

```python
class Foo(Base):

    @Part(remote=True,
          config=RemoteConfig(worker="worker_1"))
    def boxes(self):
        return Box(1, 1, 1, quantify=4)

    @Part(remote=True,
          outputs=["volume"]
          config=RemoteConfig(worker="worker_2"))
    def cyls(self):
        return Cylinder(1, 1, quantify=4)


if __name__ == "__main__":
    worker_1 = WorkerType(name="worker_1", max=2)
    worker_2 = WorkerType(name="worker_2", max=2)
    distributed.start(config=Config(workers=[worker_1, worker_2])
    foo = Foo()
    foo.boxes, foo.cyls
```

Figure 5.2: Defining a model with multiple RMs allocated to different WP types.

```
Worker-3  , MainThread    : new Cylinder transactional object
Worker-2  , MainThread    : new Cylinder transactional object
Worker-0  , MainThread    : new Box persistent object
Worker-3  , MainThread    : new Cylinder transactional object
Worker-1  , MainThread    : new Box persistent object
Worker-2  , MainThread    : new Cylinder transactional object
Worker-0  , MainThread    : new Box persistent object
Worker-1  , MainThread    : new Box persistent object
```

Figure 5.3: Logs once the ROs are instantiated.

- The Sequence is bound to a WP type with a maximum number of 16 workers. The **cpu** request of this WP type is set to 1 cpu.
- The cluster is composed of 8 nodes with 4 cpus each.

With this configuration 16 WP should be created (each one inside its own pod) and Kubernetes should allocate 2 pods per nodes. The output from the kubectl command is summarized in table 5.1. One can see that the pods are distributed as expected on the cluster's nodes.

Node ID	vmss001	vmss002	vmss003	vmss004	vmss005	vmss006	vmss007	vmss008
Pod ID	pod-12, pod-3	pod-10, pod-4	pod-14, pod-6	pod-2, pod-11	pod-15, pod-7	pod-0, pod-8	pod-5, pod-13	pod-9, pod-1

Table 5.1: Pods and their allocated nodes.

6

Results and Discussion

This chapter will present the results obtained from the test campaign. Section 6.1 will present the speedups obtained for tests T1 and T2

6.1. PRMs

6.1.1. Multicore laptop

Distributed ParaPy has been developed as a solution to parallelize KBE models both in the Cloud and on regular desktop computers. The results from the test conducted on a multicore laptop are shown in figure 6.1. The difference in speedup for both tests is clearly observable. As expected, the overheads foreseen in section 3.4 results in a lower speedup for T1. For this specific operation, the speedup is almost negligible on a 2 core laptop (1.1 times speedup), and still fairly low for 4 and 8 cores laptops (1.4 and 1.8). The parallelization efficiency ranges from 0.55 for 1 core to 0.2 for 8 cores, which is considered low. However, as explained in the methodology section, the operation performed during test T1 is an initialization operation that will be performed once during the design session. Furthermore, solutions can be implemented to reduce the WPs and ROs creation overhead, as will be detailed in section 7.2.

The operation performed in test T2 shows slightly greater speedups, but the parallel efficiency is still considered low (0.55, 0.425, 0.325). Yet the 1.7 and 2.6 times speedups obtained on a 4 and 8 cores computer are sufficient to have a significant impact on the user's experience. Concretely, a 2.6 time speedup consists of a 5.6 seconds delay instead of 14.5 seconds when performing operations on the hull geometry, which is a noticeable improvement for the user. Eventually, even is the parallelization efficiency is relatively low, these speedups are appreciable on a laptop machine since they make use of otherwise idle cores of the processor.

6.1.2. Cloud Architectures

Single User Approach

The single-user approach is expected to provide the highest speedups from the three proposed Cloud architecture, due to the fact that all the WPs of the application are running on the same VM, thus it is exempt of network communication overhead (as opposed to architectures B1 and B2) and container creation overhead (unlike architecture B1). When running the tests using this architecture, unexpected results were observed. For each VM size, the tests operations have to be run using non parallelized frames and PRM parallelized frames in order to compute the speedup. The run times measured for the non-parallelized frames are expected to decrease slightly with larger ma-

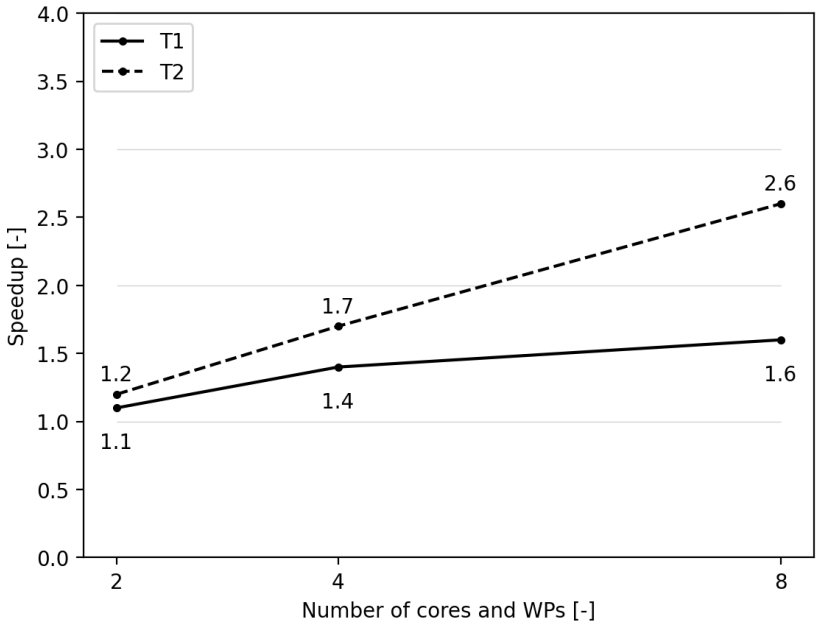


Figure 6.1: Speedups obtained for T1 and T2 on a multicore laptop computer

chines, as ParaPy's geometry kernel internally uses parallelization to improve the performance of some of its operations. However, the opposite effect was observed: the run time of both operations increased with larger VMs. The measured run times are reported in figure 6.2. There is a dramatic increase in run time starting off with the D16as VM. No concrete explanations were found for this behaviour, and therefore the speedups for the single-user approach were computed by using for T_S the shortest run time; obtained with the D8as VM.

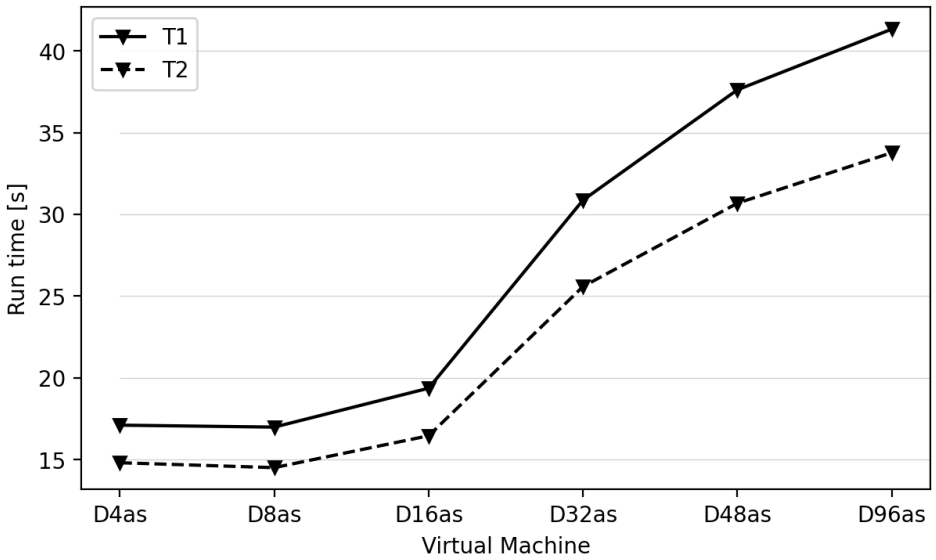


Figure 6.2: Run times of the serial frames creation/visualization and position update operations when performed on the set of VMs selected to assess the single user approach.

The speedups obtained with the single user approach are reported in figure 6.3. Test T1 provides slightly higher speedups than when run on a laptop computer, with 1.7 and 1.8 speedups for 4 and 8 cores, compared to the 1.4 and 1.6 obtained on the laptop computer. As enacted by Am-

dahl's law, the speedup curves attain a plateau with increasing number of processors (and thus, in our case, numbers of WPs). The value of this plateau is dictated by the serial portion of the operation being run. However, there is an unexpected speedup stall observed for 42 WPs, which is not predicted by Amdahl's law. Test T2 on the other hand does not seem to be too affected by Amdahl's law for the assessed range of WPs. The slope steepness decreases with increasing number of WPs, but the plateau is not reached. In order to understand the speedups obtained and propose relevant recommendations, a breakdown of the different contributions to the parallel run time is required.

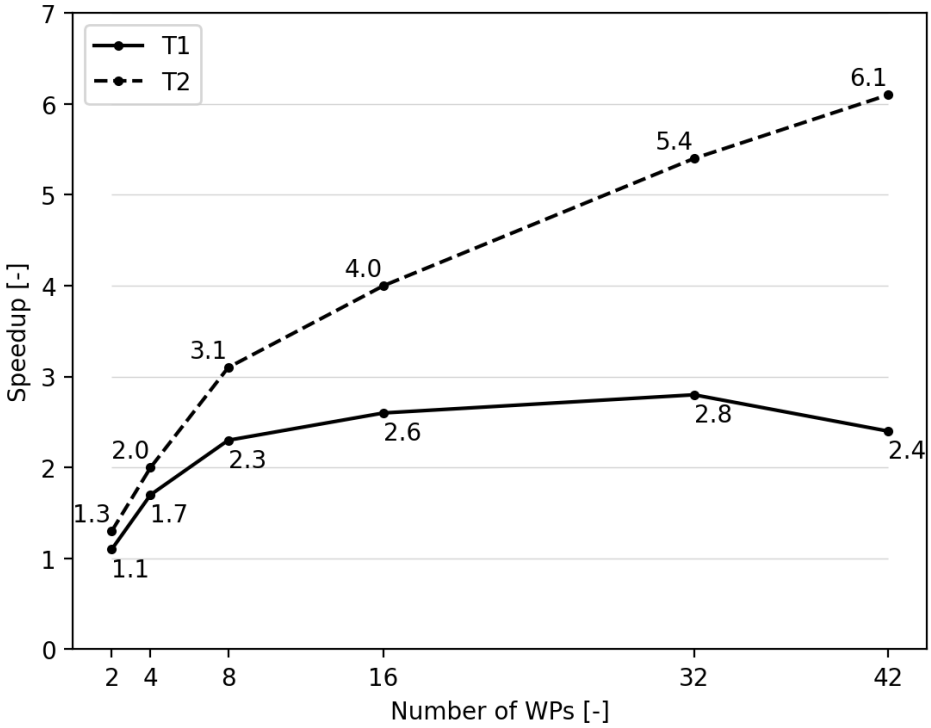


Figure 6.3: Speedups obtained with the single-user approach.

The proposed breakdown is mostly similar for both tests, but present some differences. To understand the breakdown, the sequence of actions performed by the GUI and PRMs already introduced in section 4.2 will be recalled for each test.

- T1: when the user "double-click" on the frame slots, the frames PRMs are asynchronously created. This triggers the WPs creation and ROs instantiation (see figure 4.2). The GUI will then directly call the `eval_drawables()` asynchronous method on all the frames root proxies. This call will be queued by the message broker until the ROs are ready. Once the RO are instantiated, they start scanning through their children and evaluating their drawable children's shapes (stage 1 of figure 4.9b). At some stage, this action will trigger the request of the inputs values by the ROs (the eager input evaluation method was used during the tests, so all the inputs values will be requested at once). In the meantime, the GUI has called the blocking `construct_tree()` method on the first frame and waits for the child tree serialization, which will also contain the serialized shapes values. Once the parallel evaluation of the ROs drawables' shapes is completed, the proxies tree construction can be performed (stage 3 of figure 4.9b). Then the GUI will access the shapes already cached by the proxies (see note 2). The last step is the shapes' serialization in the MP.

- T1: when the user clicks on the GUI's button to change the frames position, a first function

will be called that changes the position input of each of the frames one by one. This triggers the invalidation of each frame's geometry (and the destruction of part of its tree, since the frames' tree is dynamic). This first invalidation stage happens serially for all the frames. This invalidation happens serially for all the frames. Then, the same steps as for T1 are performed by the GUI to re-display the frames. The only differences are: 1. the WPs and ROs are already initialized. 2. Part of the frames proxy tree is already constructed on MP side (but some parts of it has been destructed by the invalidation. Therefore, after the parallel re-evaluation of the ROss shapes, the GUI will first reconstruct the missing parts of the proxies' child tree, then request the shapes of the non-destructed drawables' proxies. This flow is summarized in figure 6.4.

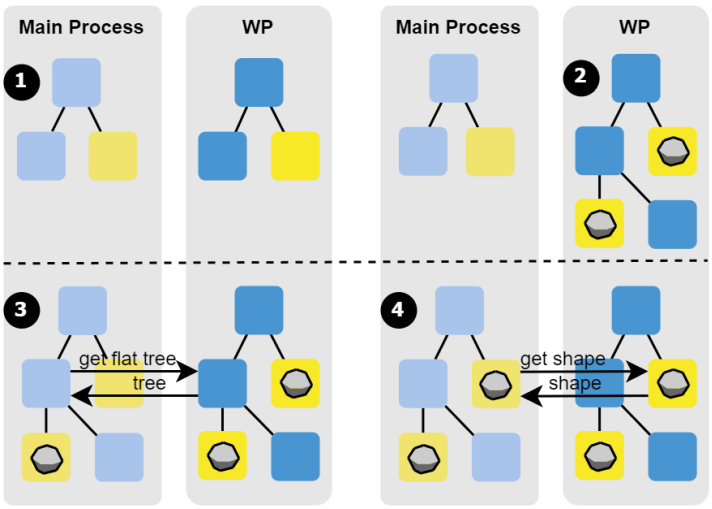


Figure 6.4: Simplified flow of actions happening during test T2, represented here for a single PRM. Previous to the position's invalidation, the proxy and RO were in the same state as in stage 4 of figure 4.9b. Changing the frame's position results in the destruction of parts of the tree, and invalidation of the shapes both in the ROs and proxies (stage 1 here). The ROs' geometries are then re-evaluated in parallel for the new position (stage 2). The GUI then reconstructs the parts of the tree that are invalidated (stage 3). This happens serially for all the frames. Multiple portion of a single frame's tree can be destructed and therefore the reconstruction of a frame's tree can result in multiple request/reply cycle. Then, the invalidated shapes of the drawables' proxies are requested (stage 4). Finally, the shapes are tessellated in the MP (not represented here).

The breakdown for T1 consists of the following time contributions:

C1 Duration of the WPs creation and instantiation of the ROs. Also called WPs initialization.

C2 Duration of the input requests by the ROs. The input request is triggered by a specific method to emulate the "natural" request procedure of the ROs. This duration is further broken down in three sub-contributions:

- The total serialization time of the values returned by the child rules (for all the frames) in the MP.
- The average deserialization time of the inputs' values in the WPs. This duration is measured for each WP and averaged since it happens in parallel for all the WPs.
- Communication time (how long it takes to send the inputs to all the ROs).

C3 Duration of the parallel evaluation of the frames geometries (once the inputs are cached by the ROs following the previous step). This duration should therefore monitor the "pure paral-

lel" part of the test.

C4 Total duration of the tree construction for all the frames. As the tree construction happens serially for each frame, the tree construction time of each frame is added up. This duration is further broken down in the following sub-contributions:

- Creation of the flat tree by the RO.
- Serialization of the flat tree in the WP (includes the serialization of the drawables' children shapes).
- Communication time.
- Deserialization of the flat tree (includes deserialization of the shapes).
- Composition of the proxy's child tree from the received flat tree.

These sub-contributions are measured for each frame and added up.

C5 Request of the drawables' proxies shapes. This will only happen for test T2. Broken down in the following sub-contributions:

- Serialization of the shapes in the WPs.
- Communication of the shapes.
- Deserialization of the shapes.

Once again, the sub-contributions are measured for each frame and added-up.

C6 Tessellation of the shapes.

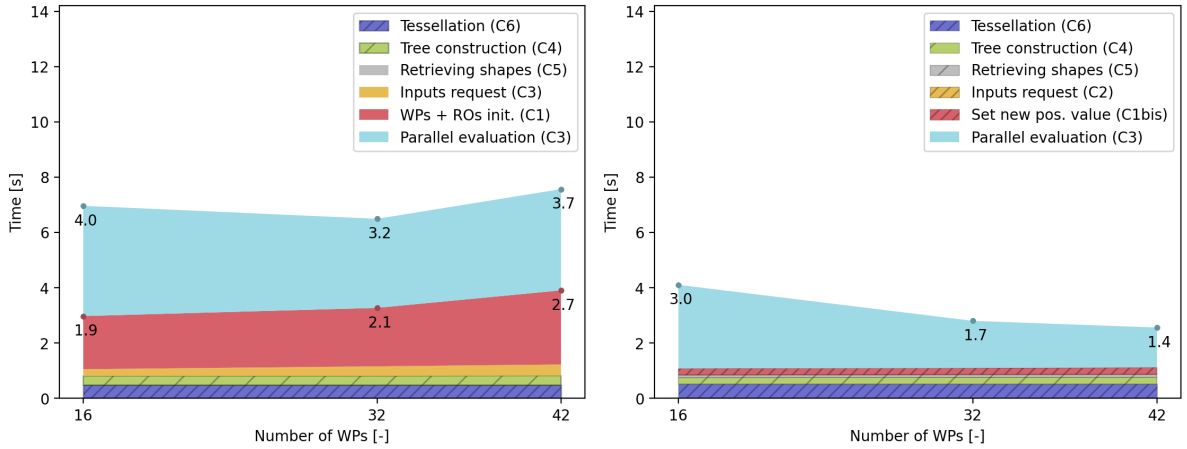
The run time breakdown for T2 is the same as for T1, to the only difference that the first contribution is the duration of changing the frames positions' value in the MP and the subsequent ensuing invalidations:

C1 (bis) Duration of the WPs creation and initialization of the ROs.

C2 Duration of the request of the inputs by the ROs. Here, only the position input will be requested, since it is the only invalidated input.

C3 Subsequent contributions are the same as for T1.

The inner breakdown of the input requests contribution **C2** is shown in figure 6.6. Focusing on T1, interesting observations can be made. First, the serialization time on MP side is constant. This is expected: the input values returned by the child-agnostic rules are serialized once and then cached. The child-specific rules are serialized for each RO, and the number of ROs is constant. Therefore, the same quantity of data is serialized no matter the number of WPs. The averaged deserialization time seems constant independently of the number of WPs. This can be explained: in each WP, the child-agnostic inputs are deserialized once, since their values are shared between ROs. However, the child-specific inputs are deserialized for each RO, thus the less ROs there are in a WP, the less deserialization should happen and therefore the deserialization time should decrease with increasing number of WPs. In our case, the size of the child-specific inputs is negligible compared to the size of the child-agnostic ones (see section 4.3). Accordingly, their deserialization time is also neg-



(a) **OP1** (b) **OP2**

Figure 6.5: Breaking down the contributions to the overall parallel run time of operations **OP1** and **OP2**. Hatched contributions represent operations that are not parallelized in the current implementation but that could be in further work.

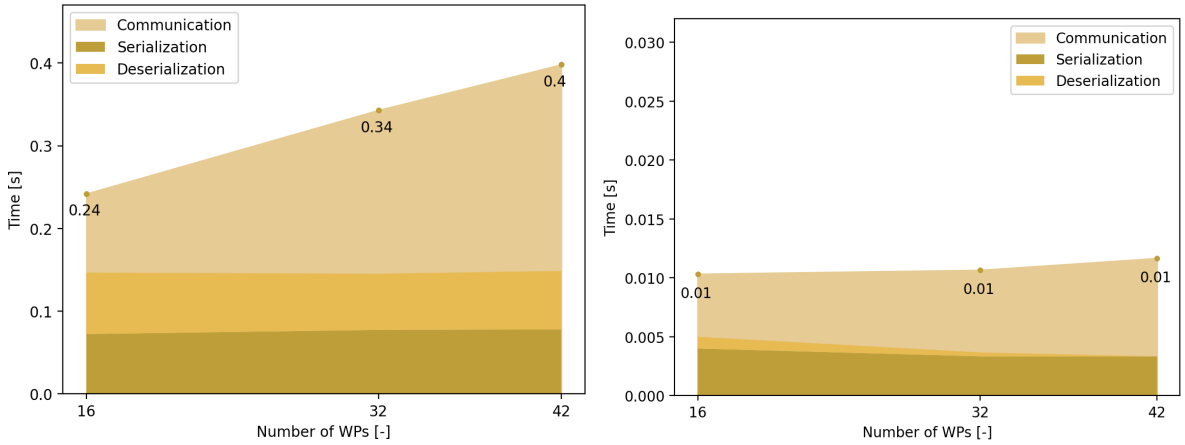
ligible which explain why the deserialization time is constant. On the other hand, the decrease in deserialization time can be observed for test T2 in figure 6.6b. In test T2, only the child-specific position input is requested and therefore its serialization time is not eclipsed by the large child-agnostic inputs. The last observation is the communication time. For test T1 it increases with the number of WPs. This is the result from the child-agnostic rules' optimization described in section 4.2.2. The value returned by these rules is only sent once per WP, so the amount of communication increases with the number of WPs.

The breakdowns of the tree construction and retrieval of the shapes are shown in figure 6.7. The tree construction is initiated by the GUI and happens serially for all the frames. Therefore, the total duration for building all the frames' tree is not affected by parallelization. This is also true for the shapes retrieval in test T2. One observation is that the communication time for the tree construction is greater for T2 than for T1, which can seem counterintuitive, since in T1 the whole tree is sent and contains all the shapes of the frame, while in T2 only the destructed parts of the tree are sent and therefore a subset of the frame's shapes. The reason is that for T1 the `construct_tree()` method is called once on the root proxy, resulting on a single request/reply cycle per frame, while for T2 this method is called all the child proxies of a frame that have invalidated Part slots.

6.1.3. MULTI-USER APPROACH

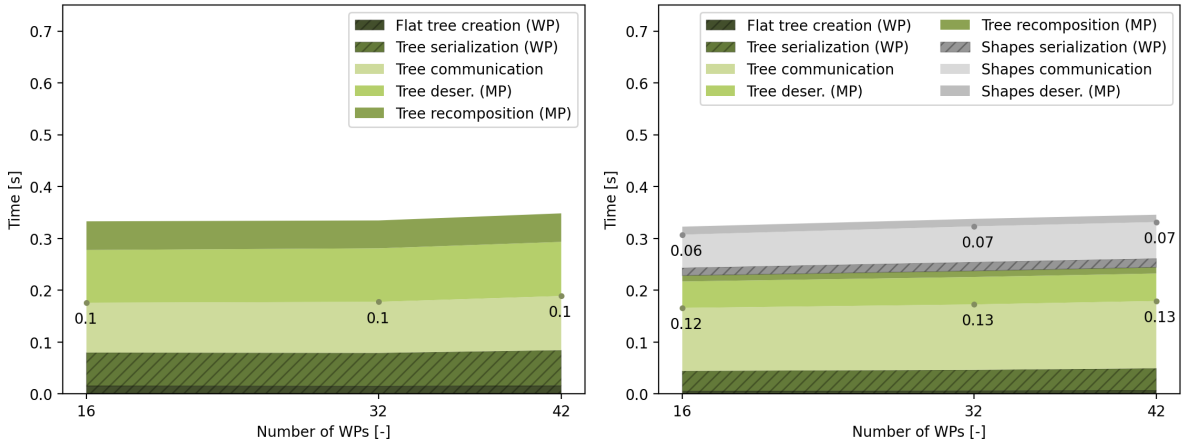
The speedups obtained for the two architectures of the multi-user approach are shown in figure 6.9. An unanticipated observation is that the speedups for both architectures are greater that the ones obtained when running on a single VM. This will be explained by looking at the run time breakdowns. T1 speedups are lower for architecture B1 which confirms that the docker container creation introduces an overhead when creating new WPs. T2 speedups are almost identical for both architectures as expected since both architectures only differ for the WP creation process. A notable feat is the increase in parallelization efficiency when comparing to the same tests run on the laptop computer. For T2, the parallelization efficiency is respectively of 0.675 and 0.53 for 4 and 8 cores against to 0.425 and 0.325 on a laptop.

Only the breakdowns of architecture B1's parallel run time will be presented and discussed. The high-level breakdown is given in figure 6.10. The parallel evaluation of the ROs's geometries (in blue) is significantly faster for architecture B1 compared to the single user approach for both tests. The



(a) **OP1** (b) **OP2**

Figure 6.6: Breaking down the durations in the RO's inputs request. Note: due to the large difference between the duration of this operation for **OP1** and **OP2**, the scales were no matched.



(a) **OP1** (b) **OP2**

Figure 6.7: Breakdown of the GUI's process of retrieving the proxies's trees and the shape representation of the ROs.

only supposition for an explanation is oversubscription caused by the geometry kernel parallelization. The oversubscription factor will always be greater for the single user approach. For example, with 32 WPs the oversubscription factor will be of 32 for the single user approach since all the processes are running on the same 32 core machine, while for architecture B1 the oversubscription factor will be of 4 for each node.

Figure 6.11 and 6.12 present the same general aspect than the ones obtained from the single user approach. The serialization times are similar, and only the communication contribution is greater for architecture B1 as expected.

Figure 6.8 present the speedups computed by using the ideal parallel contribution **C3** for the parallel run time, instead of the complete parallel run time. The light grey line represent the linear evolution that should follow the curves, given that the operation performed is completely parallelized and doesn't feature any serial computation. As one can see, the speedup curves do not evolve linearly and deviate from the linear path. This figure illustrates that there is a probable oversubscription due to the geometry kernel's parallelization that slows down the frames geometry evaluation.

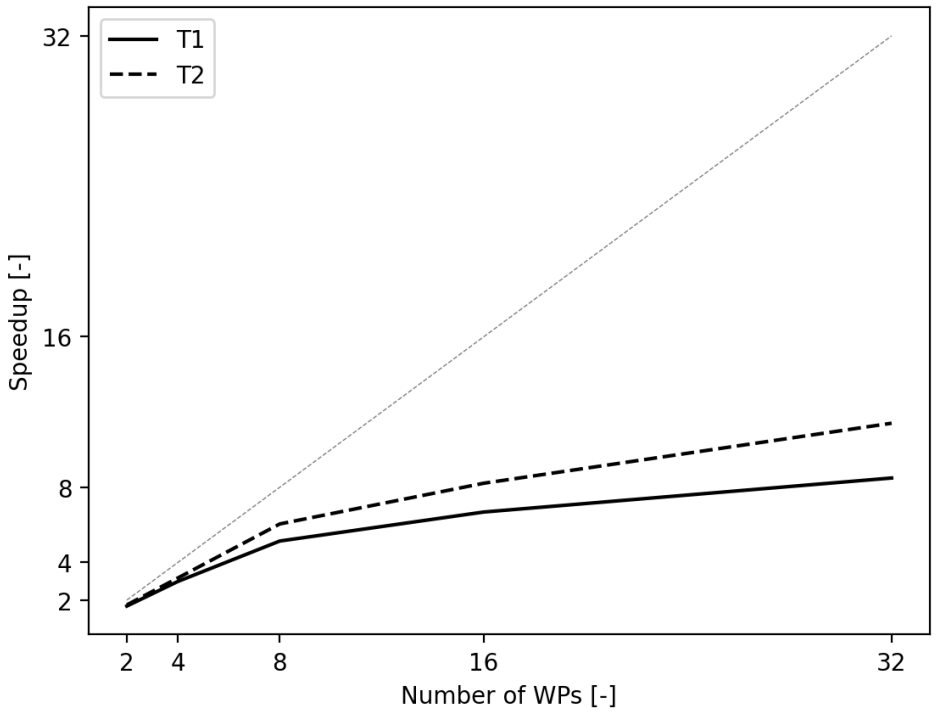


Figure 6.8: Speedups computed when isolating the parallel evaluation contribution of the run time, from architecture B1's run time breakdowns

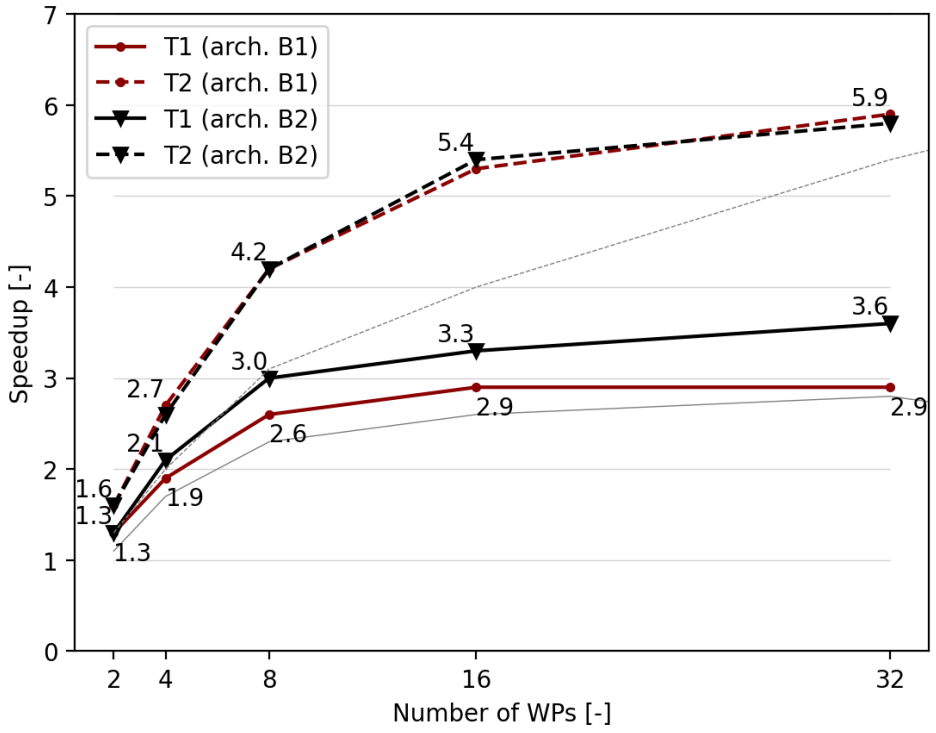
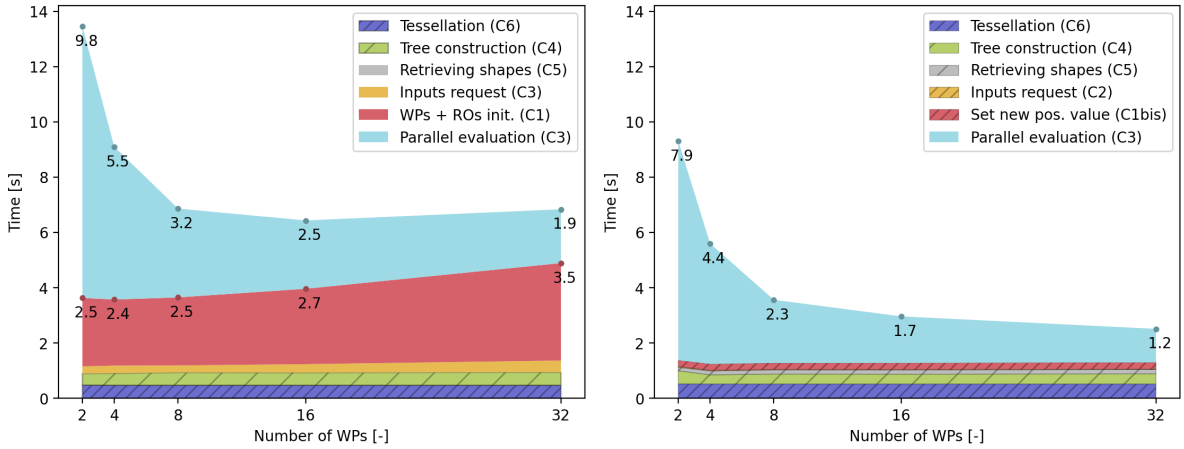


Figure 6.9: Speedups obtained with architectures B1 and B2 of the multi-user approach. Speedups from the single-user approach are shown in light grey for comparison.

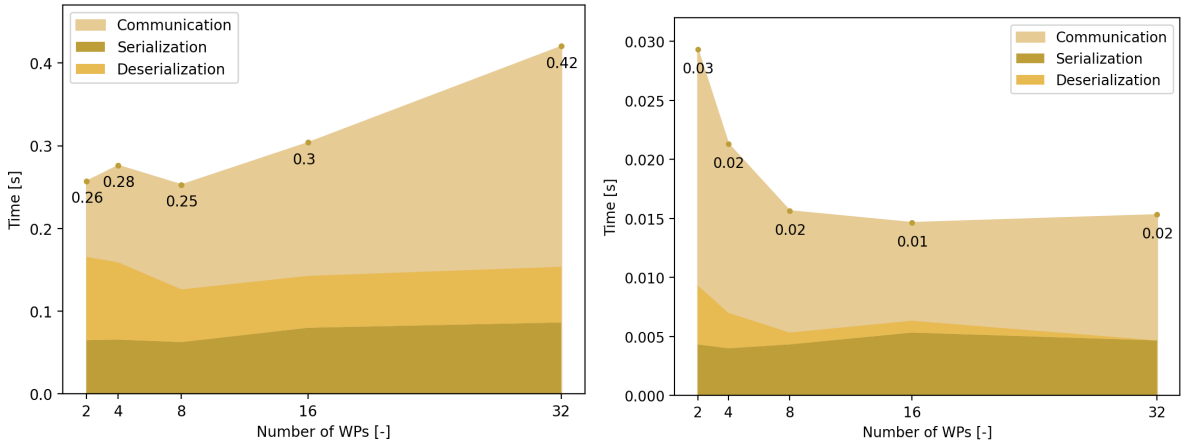
6.1.4. PARALLEL TESSELLATION AND SERIALIZATION OF THE SHAPES

In a last optimization effort to improve the speedups, a PRM and GUI implementation has been developed where the tessellation of the shapes and their serialization happens in parallel in the



(a) **OP1** (b) **OP2**

Figure 6.10: Breaking down the contributions to the overall run time of the parallel operations **OP1** and **OP2**.



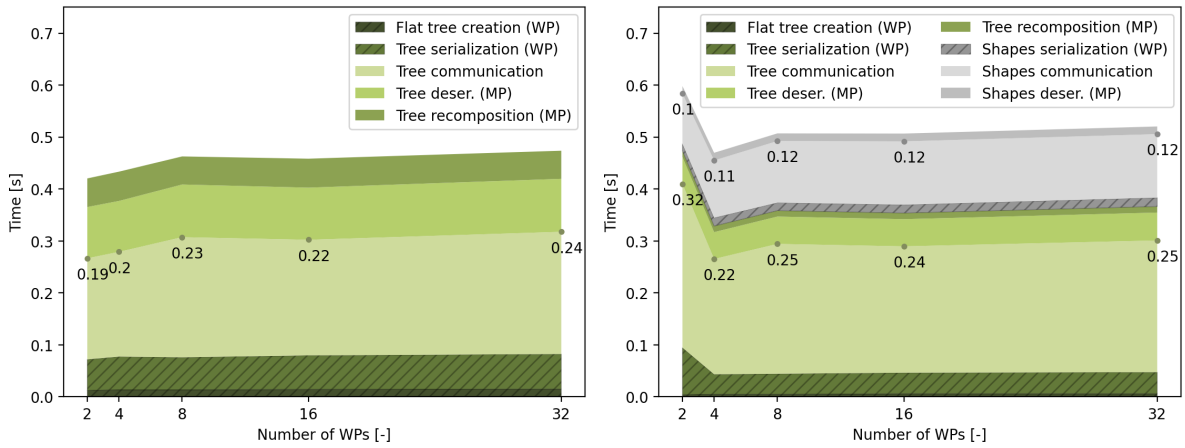
(a) **OP1** (b) **OP2**

Figure 6.11: Breaking down the durations in the RO's inputs request. Note: due to the large difference between the duration of this contribution for **OP2** and **OP2**, the scales were no matched.

WPs. This implementation reduces the tessellation time and serializations times of the tree and shapes. The speedups are shown in figure 6.13.

6.2. TRMs

The results from running **T3** on a laptop computer are reported in table 6.1. Despite the primitiveness of this test, the results demonstrate the potential of TRMs for reducing the memory footprint of KBE models. The memory ration ratio between the frames size once their shapes have been evaluated (and fetched in the TRM case) if of **6.2**. For this specific use-case, this only represents a saving of 223.1MiB, but it is easily imaginable that such constructs could be successfully used on larger use cases and eventually solve memory issues in KBE models by providing dependency tracked reduced representations of child objects. Furthermore, it is envisioned that the caching programming model of TRMs, where only the inputs and outputs are cached and dependencies traced between them could be adapted to local children as a way to only save memory for use cases where parallelization is not relevant.



(a) **OP1** (b) **OP2**

Figure 6.12: Breakdown of the GUI's process of retrieving the proxies's trees and the shape representation of the ROs.

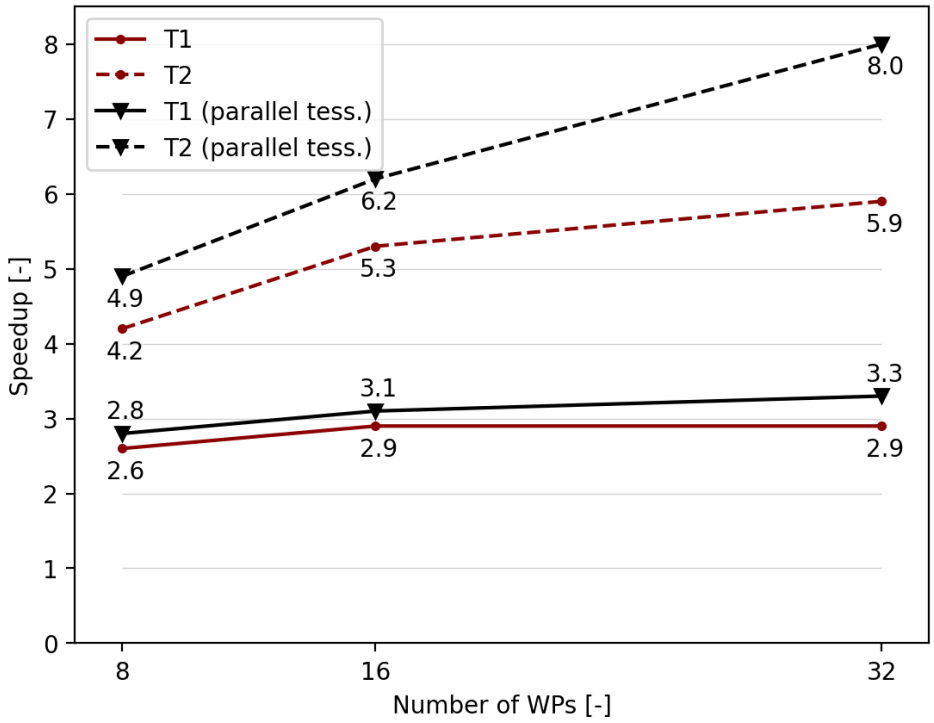


Figure 6.13: Comparison of the speedups obtained when parallelizing the tessellation and serialization of the ROs' shapes with the speedups from the previous parallel implementation, using cloud architecture B1.

Frames	Serial	TRMs
Process's size before evaluation (MiB)	751.49	750.9
Process's size after evaluation (MiB)	1017.8	793.83
Frames size (MiB)	266	42.9
Δ_{mem} (MiB)		223.1
Average size/frame	6.34	1.02
r_{mem}		6.2

Table 6.1: Memory

7

Conclusion and Recommendations

7.1. Conclusion

This work proposed an extension to the ParaPy SKD by introducing two new constructs made available to KBE model developers to distribute and parallelize the execution of their models. The choice was made based on observation of recurring patterns in KBE models architectures to partition the models across parent-child relations, which often define parallel regions in the models. Two constructs were introduced, with different programming models and objectives. PRMs aim at increasing the performance of KBE models and applications by decreasing their (re)generation times. They are fitted for KBE applications where the user performs iterative design, and particularly optimized for fast and efficient display of the model's geometry by the GUI. TRMs are more at a prototype stage and aim at proposing a solution to control the memory footprint of a model when parallelizing it.

The implementation choices of RMs enforce some conditions on ParaPy KBE models to be parallelized. The first condition is that the model should present parallel regions materialized by sibling children. Furthermore, no interdependencies are permitted between RMs. This is a strong condition to parallelize KBE models, which could be removed in future works. Furthermore, the Inputs of both RMs should be non-ParaPy serializable objects, or define a custom serialization method. All these criteria answer question **RQ2:.b.**. Furthermore, the refactoring work described in section 4.3 to accommodate the RH application to RMs shows that existing application can be adapted to parallelization with moderate efforts. For developers creating new applications, the clear API and programming models proposed by both RMs is expected to make their use relatively easy. Furthermore, the behaviour of PRMs, which emulates the one of regular ParaPy children and their complete integration with the GUI should make their adoption by developers swift and effortless (research question **RQ2:.c.**).

RQ1: was answered by tests on PRMs. The results of these tests showed that the partitioning strategy and programming model proposed by PRMs can indeed reduce the (re)generation time of KBE models. The speedups obtained on a laptop computer were relatively low, with a maximum speedup of 2.6 on a 8 core machine. However, such speedup can already improve the user experience when interacting on a model. The speedups obtained on the different proposed cloud architecture were more conclusive, and the computed speedup efficiency was superior. The last optimization effort, where the tessellation of the shapes was performed in parallel, showed speedups up to 8 times when using 32 WPs. The main overhead observed is the initialization of the WPs and ROs. Surprisingly, the communication overhead was relatively low, and its impact only felt for higher

number of WPs.

The comparison of the 3 proposed cloud architectures provides significant insight on the most efficient manner to leverage cloud computing for distributed models (**RQ3:**). The so-called single-user approach, which presents drawbacks in terms of setup time (claiming the node), cost (no shared resources) and scalability eventually gave against all odds the lowest speedups. The suggested reason is oversubscription, but this hypothesis should be confirmed in further works. Cloud architectures B1 and B2 only showed difference in terms of speedup during the initial worker creation, due to B1's container creation overhead. However, the difference in speedups are for the creation operation are relatively low (maximum 17%), and other design update operations are not impacted. It can be concluded that it is worth to further work in the goal of resolving the assumptions made for architecture B1 as it seems to provide the lowest development effort to achievable gain ratio.

Eventually, the prototype proposed with TRMs demonstrates the potential of these constructs to control and reduce the memory footprint of KBE models, and partially answers **RQ4:**. Further tests need to be conducted to fully assess their ability to solve the memory issues encountered by large KBE models.

7.2. RECOMMENDATIONS

As this work paved the way for distributed ParaPy, many recommendations can be made.

A first recommendation can be made to facilitate the integration of PRMs with already existing KBE models. When adapting the RH use case for distribution, most of the effort targeted the Inputs. A solution should be found to pass ParaPy objects as Inputs. A suggestion would be to create in inverse proxy mechanism, with a proxy object in the WP for the object living in the MP. The main issue foreseen with this implementation is the high communication overhead it could incur, since ROs could be required to perform multiple request/reply cycle before accessing the slot of interest of the input object.

A second recommendation is the pre-creation of place holder WPs when an application is started. This would reduce the WP initialization time when accessing RMs for the first time, especially if the WPs are configured to automatically import the required libraries to instantiate the ROs when started.

Eventually, PRMs could evolve into independent remote child object controllable by another user. This architecture would allow synchronous collaboration in large KBEs models, as represented in figure 7.1.

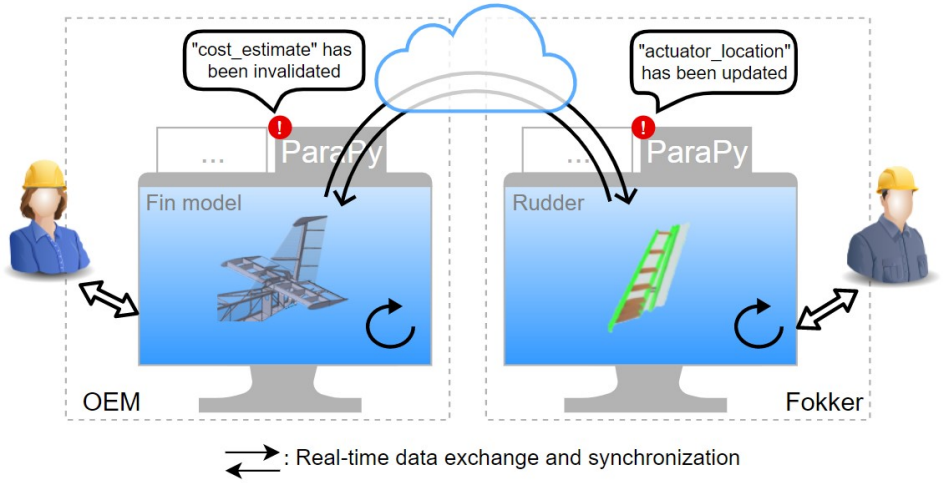


Figure 7.1: Evolution of PRMs into independent sub-models to introduce real-time collaboration in KBE systems.