CHAPTER 1

Introduction

This book introduces Owl, a numerical library we have been developing and maintaining for years. We develop Owl for scientific and engineering computing in the OCaml language. It focuses on providing a comprehensive set of high-level numerical functions so that developers can quickly build up any data analytical applications. Over years of intensive development and continuous optimization, Owl has evolved into a powerful software system with competitive performance compared to mainstream numerical libraries. Meanwhile, Owl's overall architecture remains simple and elegant. Its small codebase can be easily managed by a small group of developers.

In this book, we are going to introduce the design and architecture of Owl, from its designers' perspective. The target audience is anyone who is interested in not only how to use mathematical functions in numerical applications but also how they are designed, organized, and implemented from scratch. Some prerequisites are needed though. We assume the readers are familiar with basic syntax of the OCaml language. We recommend [38] as a good reference book on this matter. Also note that this book focuses on introducing core parts of the Owl codebase, such as the implementation and design of various key modules. If you are more interested in how to use the functionalities provided in Owl to solve numerical problems, such as basic mathematical calculation, linear algebra, statistics, signal processing, etc., please refer to our book *OCaml Scientific Computing: Functional Programming in Data Science and Artificial Intelligence* [26].

1.1 Numerical Computing in OCaml

Scientific computing is a rapidly evolving multidisciplinary field that uses advanced computing capabilities to understand and solve complex problems in the real world. It is widely used in various fields in research and industry, for example, simulations in biology and physics, weather forecasting, revenue optimization in finance, etc. One

L. Wang and J. Zhao, *Architecture of Advanced Numerical Analysis Systems*,
https://doi.org/10.1007/978-1-4842-8853-5_1

most recent hot topic in scientific computing is machine learning. Thanks to the recent advances in machine learning and deep neural networks, there is a huge demand on various numerical tools and libraries in order to facilitate both academic researchers and industrial developers to fast prototype and test their new ideas, then develop and deploy analytical applications at a large scale.

Take deep neural networks as an example; Google invests heavily in TensorFlow, while Facebook promotes their PyTorch. Beyond these libraries focusing on one specific numerical task, the interest on general-purpose tools like Python and Julia also grows fast. Python has been one popular choice among developers for fast prototyping analytical applications. One important reason is SciPy and NumPy libraries, tightly integrated with other advanced functionality such as plotting, offer a powerful environment which lets developers write very concise code to finish complicated numerical tasks. As a result, most frameworks provide Python bindings to take advantage of the existing numerical infrastructure in NumPy and SciPy.

On the other hand, back before Owl was developed, the support of basic scientific computing in OCaml was rather fragmented. There had been some initial efforts (e.g., Lacaml, Oml, Pareto, etc.), but their APIs were either too low level to offer satisfying productivity or the designs overly focused on a specific problem domain. Moreover, inconsistent data representation and excessive use of abstract types made it difficult to exchange data across different libraries. Consequently, developers often had to write a significant amount of boilerplate code just to finish rather trivial numerical tasks. There was a severe lack of a general-purpose numerical library in the OCaml ecosystem. However, we believe OCaml is a good candidate for developing such a general-purpose numerical library for two important reasons:

- We can write functional code as concise as that in Python with type-safety.
- OCaml code often has much superior performance compared to dynamic languages such as Python and Julia.

1.2 Architecture

Designing and developing a full-fledged numerical library is a nontrivial task, despite that OCaml has been widely used in system programming such as MirageOS. The key difference between the two is fundamental and interesting: system libraries provide a lean set of APIs to abstract complex and heterogeneous physical hardware, while numerical libraries offer a fat set of functions over a small set of abstract number types.

When the Owl project started in 2016, we were immediately confronted by a series of fundamental questions like: "what should be the basic data types", "what should be the core data structures", "what modules should be designed", etc. In the following development and performance optimization, we also tackled many research and engineering challenges on a wide range of different topics such as software engineering, language design, system and network programming, etc. As a result, Owl is a rather complex library, arguably one of the most complicated numerical software system developed in OCaml. It contains about 269K lines of OCaml code, 142K lines of C code, and more than 6500 functions. We have strived for a modular design to make sure that the system is flexible and extendable.

We present the architecture of Owl briefly as in Figure 1-1. It contains two subsystems. The subsystem on the left part is Owl's numerical subsystem. The modules contained in this subsystem fall into four categories:

- Core modules contain basic data structures, namely, the n-dimensional array, including C-based and OCaml-based implementation, and symbolic representation.
- Classic analytics contains basic mathematical and statistical functions, linear algebra, signal processing, ordinary differential equation, etc., and foreign function interface to other libraries (e.g., *CBLAS* and *LAPACKE*).
- Advanced analytics, such as algorithmic differentiation, optimization, regression, neural network, natural language processing, etc.
- Service composition and deployment to multiple backends such as to the browser, container, virtual machine, and other accelerators via a computation graph.

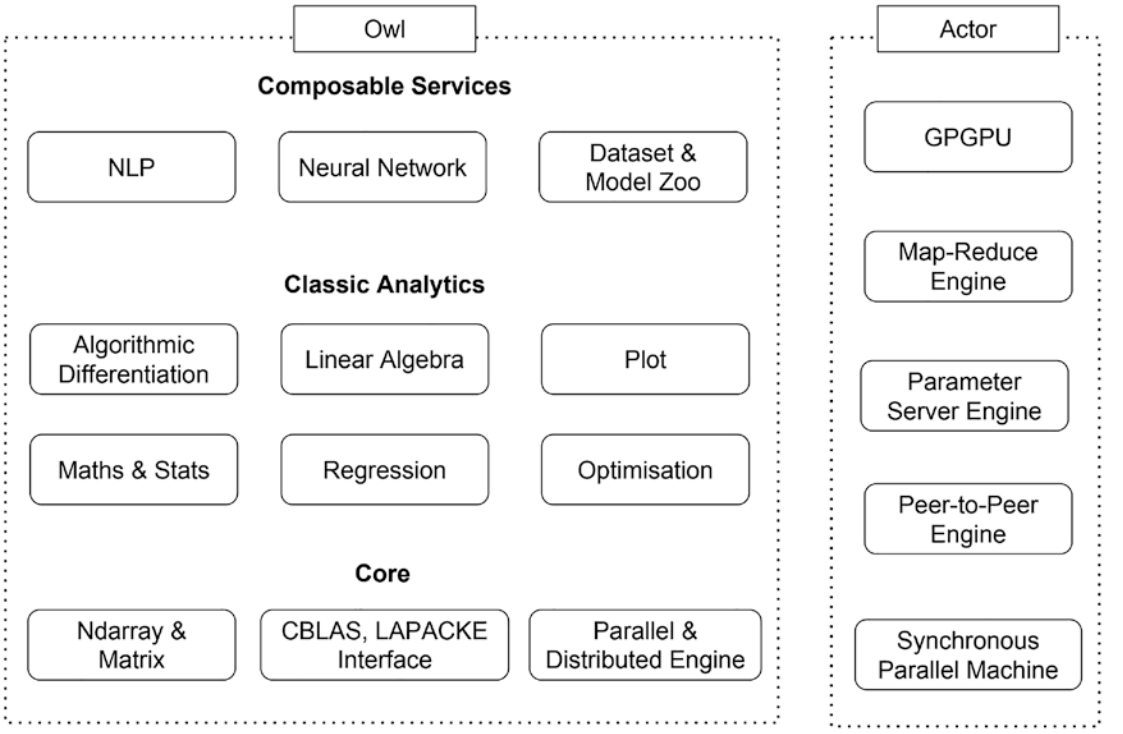

Figure 1-1. *The whole system can be divided into two subsystems. The subsystem on the left deals with numerical computation, while the one on the right handles the related tasks in a distributed and parallel computing context including synchronization, scheduling, etc*

In the rest of this chapter, we give a brief introduction about these various components in Owl and set a road map for this book.

Basic Computing and Analytics with Owl

In a numerical library or software such as NumPy, MATLAB, TensorFlow, and Owl, the *N*-dimensional array (ndarray) is the most fundamental data type for building scientific computing applications. In most such applications, solely using scalar values is insufficient. Both matrices and vectors are special cases of ndarray. Owl provides the Ndarray module, the usability and performance of which are essential to the whole architecture in Owl. In Chapter 2, we will introduce this module, including how it is designed and its performance optimization.

Owl supports a wide variety of classic numerical analytics methods, including basic mathematical functions, statistics, linear algebra, ordinary differential equation, and signal processing. The functions in each field are included in a corresponding module. Their design is similar to that of Ndarray module, which is mainly interfacing to existing tools in C code, such as OpenBLAS. The Ndarray module partly relies on these functions, especially the ones that operate on scalars. For example, Ndarray provides a sin function. What it does is actually calling the scalar version sin function in the Maths module and mapping them on all its elements. Since this book mainly focuses on the architectural design of Owl, we will not introduce in detail how to apply Owl in these fields; instead, we will briefly give some examples in Appendix A.

Advanced Design in Owl

We have introduced the various classic analytics fields that are supported in Owl. What makes Owl more powerful are a series of advanced analytics. At the heart of them lies algorithmic differentiation (AD), which will be introduced in Chapter 3. Performing differentiation is so crucial to modern numerical applications that it is utilized almost everywhere. Moreover, it is proven that AD provides both efficient and accurate differentiation computations that are not affected by numerical errors.

One of the prominent analytics that benefit from the power of AD is optimization (Chapter 4), which is about finding minima or maxima of a function. Though optimization methods do not necessarily require computing differentiation, it is often a key element in efficient optimization methods. One most commonly used optimization method is gradient descent, which iteratively applies optimization on a complex object function by small steps until it is minimized by some standard. It is exactly how another type of advanced application, regression, works. Regression covers a variety of methods, including linear regression, logistic regression, etc. Each can be applied to different scenarios and can be solved by various methods, but they share a similar solution method, namely, iterative optimization.

Depending on applications, the computation generated from AD can be quite complex. They often form a directed acyclic graph, which can be referred to as a computation graph. Based on the computation graph, we can perform various optimizations to improve the execution speed, reduce memory usage, etc. A computation graph plays a critical role in our system, and we introduce it in Chapter 6.

Finally, neural networks, as complex as their architectures can be, are in essence also an extension of regression and therefore are also trained by iterative optimization. We cover this topic in Chapter 5. With the popularity of machine learning and neural networks, this series of advanced analytics, especially the core technique AD, has become increasingly essential in modern numerical analysis library stacks. In this book, we will briefly introduce these topics and their architecture in design in Owl.

Hardware and Deployment

In the next part, we discuss the execution backends Owl executes on in Chapter 7. The first important backend we consider is hardware accelerators such as GPUs. Scientific computing often involves intensive computations, and a GPU is important to accelerate these computations by performing parallel computation on its massive cores. Besides, there are growingly more types of hardware accelerators. For example, Google develops the Tensor Processing Unit (TPU), specifically for neural network machine learning. It is highly optimized for large batches in using TensorFlow. To improve performance of a numerical library such as Owl, it is necessary to support multiple hardware platforms.

One idea is to "freeride" existing libraries that already support various hardware platforms. We believe that a computation graph is a suitable intermediate representation (IR) to achieve interoperability between different libraries. Along this line, we develop a prototype symbolic layer system by using which the users can define a computation in Owl and then turn it into an ONNX structure, which can be executed on many different platforms such as TensorFlow. By using the symbolic layer, we show the system workflow and how powerful features of Owl, such as algorithmic differentiation, can be used in TensorFlow.

Besides aiming for maximal performance on accelerators, sometimes a numerical library also targets to run in a wide range of environments, for example, to execute on web browsers based on JavaScript for ease of access to many end users. Besides, with the trending of edge computing and the Internet of Things, it is also crucial for numerical libraries to support running on computation resource–limited devices. One approach is to construct *unikernel*, a specialized minimal virtual machine image that only contains necessary libraries and modules to support an application. In this book, we also introduce Owl's support for execution in this environment.

Research on Owl

In the last part of this book, we introduce two components in Owl: *Zoo*, for service composition and deployment, and *Actor*, for distributed computing. The focus of these two chapters is to present two pieces of research based on Owl.

In Chapter 9, we introduce the Zoo subsystem. It was originally developed to share OCaml scripts. It is known that we can use OCaml as a scripting language as Python (at certain performance cost because the code is compiled into bytecode). Even though compiling into native code for production use is recommended, scripting is still useful and convenient, especially for light deployment and fast prototyping. In fact, the performance penalty in most Owl scripts is almost unnoticeable because the heaviest numerical computation part is still offloaded to Owl which runs native code. While designing Owl, our goal is always to make the whole ecosystem open, flexible, and extensible. Programmers can make their own "small" scripts and share them with others conveniently, so they do not have to wait for such functions to be implemented in Owl's master branch or submit something "heavy" to OPAM. Based on these basic functionalities, we extend the Zoo system to address the computation service composition and deployment issues.

Next, we discuss the topic of distributed computing. The design of distributed and parallel computing module essentially differentiates Owl from other mainstream numerical libraries. For most libraries, the capability of distributed and parallel computing is often implemented as a third-party library, and the users have to deal with low-level message passing interfaces. However, Owl achieves such capability through its Actor subsystem. Distributed computing includes techniques that combine multiple machines through a network, sharing data and coordinating progresses. With the fast-growing number of data and processing power they require, distributed computing has been playing a significant role in current smart applications in various fields. Its application is extremely prevalent in various fields, such as providing large computing power jointly, fault-tolerant databases, file system, web services, and managing large-scale networks, etc.

In Chapter 10, we give a brief bird's-eye view of this topic. Specifically, we introduce an OCaml-based distributed computing engine, Actor. It has implemented three mainstream programming paradigms: parameter server, map-reduce, and peer-to-peer. Orthogonal to these paradigms, Actor also implements all four types of synchronization

barriers. We introduce four different types of synchronization methods or “barriers” that are commonly used in current systems. We further elaborate how these barriers are designed and provide illustrations from the theoretical perspective. Finally, we use evaluations to show the performance trade-offs in using different barriers.

1.3 Summary

In this chapter, we introduced the theme of this book, which centers on the design of Owl, a numerical library we have been developing. We briefly discussed why we build Owl based on the OCaml language. And then we set a road map for the whole book, which can be categorized into four parts: basic building block, advanced analytics, execution in various environments, and research topics based on Owl. We hope you will enjoy the journey ahead!