Thank you for purchasing the MEAP of *Modern Fortran: Building efficient parallel applications*. I'm excited that you chose to be part of this journey with me. This book is still in early development and I hope that you can help me make it as great as possible.

As you can probably tell from the title, this is a book about parallel programming with modern Fortran. I place emphasis on modern because Fortran has evolved substantially since 1957 when it appeared as the first high-level programming language in history. Despite the explosive growth and mobile- and web-oriented technologies in the last two decades, Fortran remains the only standardized language with a native parallel programming model – the Fortran Coarrays.

This book will teach you Fortran and parallel programming by doing. It will guide you step-by-step through the development of a complete and parallel fluid dynamics solver from scratch. We will start by learning the core features of the language, essential for any Fortran application. We will then dive into Fortran Coarrays and learn how to decompose the computational domain into parallel tiles and exchange data between them. We will dig into measuring and improving performance of parallel apps, and later some object-oriented and functional programming techniques for data abstraction and writing cleaner code. Finally, we will cover modern approaches to interfacing C applications, as well as I/O with common data formats.

I believe this book will be most useful to Fortran novices or existing Fortran programmers who need to harness the power of many parallel CPUs for their application. Whether you use parallelism to speed up your code, or to overcome the memory limitations of a single computer, this book will show you how to do it in practice. In the end, you will come out with a skill set that you can apply to your own application, be it numerical weather prediction, aerodynamics simulation, or machine learning.

Your advice is essential for my writing of this book. Please use the [Author Online Forum](https://forums.manning.com/forums/modern-fortran) to post feedback or ask question. This will give you an opportunity to steer the writing in a certain direction. Whether I left something unclear, or you find a code example with a critical error, let me know! I want to create the best book possible for you and others.

—Milan Curcic
brief contents

PART 1: GETTING STARTED WITH MODERN FORTRAN

1 Introducing Fortran
2 Getting started: Minimal working app
3 Writing reusable code with procedures and modules
4 Fast math with array operators and arithmetic

PART 2: ADVANCED FORTRAN USE

5 Going parallel with Fortran coarrays
6 Using derived types to work with abstract data
7 Overloading operators and generic functions
8 Input and output: Namelists, JSON, and NetCDF

PART 3: THE FINAL STRETCH

9 Advanced parallelism with teams and events
10 Interoperability with C: Exposing your app to the web
11 Easy to use apps with rich CLIs and documentation
12 Publishing your Fortran app

APPENDIXES

A Setting up the Fortran development environment
B From calculus to code
C Glossary
This is a book about Fortran, one of the first high-level programming languages in history. It will teach you the language by guiding you step-by-step through the development of a fully-featured, parallel physics simulation app. Notice the emphasis on parallel. I will introduce the concept of parallel programming early on, and start applying it from first principles. Parallel programming allows you to break down your problem into pieces, and let multiple processors each work on only part of the problem, thus reaching the solution in less time. By the end, you will be able to recognize problems that can be parallelized, and you will be able to use modern Fortran techniques to solve them.

Modern Fortran is not a comprehensive reference manual for every Fortran feature. There are significant parts of the language that I have omitted on purpose. Instead, I focus on the most practical Fortran features that you would use to build a real-world application. As we work on our app chapter by chapter, we will apply modern Fortran features and software design techniques to make our app robust, portable, and easy to use and extend. That said, I stand corrected. This is not just a book about Fortran. This is a book about building robust, parallel software using modern Fortran.
1.1 What is Fortran?

"I don’t know what the language of the year 2000 will look like, but I know it will be called Fortran."

-- Tony Hoare, winner of the 1980 Turing Award

Fortran is a general-purpose, parallel programming language that excels in scientific and engineering applications. Originally called FORTRAN (FORmula TRANslation) in 1957, it has evolved over decades to a robust, mature, and high performance-oriented programming language. Today, Fortran keeps churning under the hood of many systems that we take for granted:

- Numerical weather, ocean, and surf prediction
- Climate science and prediction
- Computational fluid dynamics software used in mechanical and civil engineering
- Aerodynamics solvers for designing cars, airplanes, and spacecraft
- Fast linear algebra libraries used by machine learning frameworks
- Benchmarking the fastest supercomputers in the world (top500.org)

Here’s a specific example. In my work, I deal mostly with the development of numerical models for weather, ocean surface waves, and deep ocean circulation. Speaking about it over the years, I found that most people didn’t really know where weather forecasts come from. The general idea is that a group of meteorologists would gather and together come up with a chart of what the weather will be like tomorrow, in a week, or a month from now. This is only partially true. In reality, we use sophisticated numerical models that crunch a huge amount of numbers on very large computers. In layman terms, these models simulate the atmosphere to create an educated guess of what the weather will be like some time in the future. The results of these simulations are then used by meteorologists to create a meaningful weather map (Figure 1.1). This map shows just a sliver of all data that is produced by the model. The output size of a weather forecast like this counts in hundreds of gigabytes.
Figure 1.1. A forecast of Hurricane Irma on September 10, 2017, computed by an operational weather prediction model written in Fortran. Shading and barbs shows surface wind speed in meters per second, and contours are isolines of sea-level pressure. A typical weather forecast is computed in parallel using hundreds of CPUs. Data provided by the NOAA National Center for Environmental Prediction (NCEP).

The most powerful Fortran applications run in parallel on hundreds or thousands of CPUs. Development of the Fortran language and its libraries have been largely driven by the need to solve extremely large computational problems in physics, engineering, and biomedicine. To access even more computational power than what the most powerful single computer at the time could offer, in the late 20th century we started connecting many computers with high-bandwidth networks, and let them each work on a piece of the problem. The result is the so-called supercomputer, a massive computer that is typically made of thousands of commodity CPUs (Figure 1.2). Supercomputers are similar to modern server farms hosted by Google or Amazon, except that the network infrastructure in supercomputers is designed to maximize bandwidth and minimize latency between the servers themselves, rather than the outside world. As a result, the CPUs in a supercomputer act like one giant processor with distributed-memory access that is almost as fast as local memory access. To this day, Fortran remains the dominant language used for such massive-scale parallel computations.
Figure 1.2. MareNostrum 4 supercomputer at the Barcelona Supercomputing Center. The computer is housed inside the Torre Girona Chapel in Barcelona, Catalonia, Spain. A high-speed network connects each cabinet one to another. With 165,888 Intel Xeon cores, MareNostrum 4 is the fastest supercomputer in Spain, and 16th fastest in the world as of November 2017 (www.top500.org/lists/2017/06/). It is used for many scientific applications, from astro- and materials physics, to climate and atmospheric dust transport prediction, to biomedicine. Image source: www.bsc.es/marenostrum/marenostrum.

1.2 Fortran features

"This is not your parents' Fortran."

-- Damian Rouson

In the context of programming languages, Fortran is:

- **Compiled**: You will write whole programs and pass them to the *compiler* before executing them. This is in contrast to *interpreted* programming languages like Python or Javascript which can be parsed and executed line by line. While this makes writing programs a bit more tedious, it allows the compiler to generate extremely efficient executable code. In typical use cases, Fortran programs are one or two orders of magnitude faster than equivalent Python programs.
What is a compiler?
A computer program that reads source code written in one programming language and translates it to equivalent code in another programming language. In our case, a Fortran compiler will read Fortran source code and generate equivalent assembly code and machine (binary) instructions.

- **Statically-typed**: In Fortran, you will give all variables a type upon declaration, and they will remain of that type until the end of the program:

  ```fortran
  real :: pi  
  pi = 3.141592
  ```

  1. `pi` must be declared before use
  2. `pi` remains a `real` number until the program halts.

You will also need to explicitly declare the variables before their use, which is known as *manifest typing*. Finally, Fortran employs the so-called *strong typing*, which means that the compiler will raise an error if it can notice that a procedure is being invoked with an argument of the wrong type. While static typing helps the compiler to generate efficient programs, manifest and strong typing enforce good programming hygiene and make Fortran a safe language. I find it is easier to write correct Fortran programs than Python or Javascript, which come with many hidden caveats and "gotchas".

- **Multi-paradigm**: You can write Fortran programs in several different paradigms, or styles. These include imperative, procedural, array-oriented, object-oriented, and even functional programming. Some paradigms are more appropriate than others, depending on the problem you are trying to solve. We will explore different paradigms in more detail in later chapters.

- **Parallel**: Fortran is also a *parallel* language. This refers to the capability to split the computational problem between multiple processes that communicate through whatever network lays between them. These processes can be running on the same processing core (known as thread-based parallelism), on different cores that share RAM (shared-memory parallelism), or distributed across the network (distributed-memory parallelism). Computers working together on the same parallel program can be physically located across the room, or even across the world. Fortran 2008 standard introduced *coarrays*, a syntax element that allows you to express parallel algorithms and remote data exchange without any external libraries. A coarray is an entity that allows you to access remote memory in the same way that you would access elements of an array. I show an example of exchanging data between *images* (a Fortran word for parallel processes) in Listing 1.2.

  ```fortran
  integer :: coarray, x, y, z
  coarray = 1
  x = 2
  y = 3
  z = x + y
  ```

```
the program.

```fortran
integer, codimension[*] :: a
integer :: i

a = this_image()

if (this_image() == 1) then
  do i = 1, num_images()
    write(*,*) 'Value on image', i, 'is', a[i]
  end do
end if
```

1. Each image declares a local copy of an integer `a`
2. Each image assigns its number (1, 2, 3, etc.) to `a`
3. Only image 1 will enter this `if`-block
4. Loop from 1 to the total number of images
5. For each remote image, image 1 will get the value of `a` on that image, and print it to screen

The Fortran standard itself does not dictate how the data exchange is implemented in the underlying hardware and operating system — it merely specifies the syntax and the expected behavior. This allows the compiler developers to use the optimal mechanisms available on specific hardware. Given a capable compiler and libraries, the Fortran programmer will be able to write code that will run on conventional CPUs, many-core (hybrid) CPUs like Intel MIC co-processors, or general-purpose GPUs.

- **Mature**: In 2016, we celebrated 60 years since the birth of Fortran. The language has evolved through several iterations of the standard:
  - FORTRAN 66, also known as FORTRAN IV (ANSI, 1966)
  - FORTRAN 77 (ANSI, 1978)
  - Fortran 90 (ISO/IEC, 1991; ANSI, 1992)
  - Fortran 95 (ISO/IEC, 1997)
  - Fortran 2008 (ISO/IEC, 2010)
  - Fortran 2018 (to be published in 2018)

Fortran development and implementation in compilers has been heavily supported by the industry: IBM, CRAY, Intel, NAG, Portland Group/NVIDIA, and others. There have also been significant developments in the open source community, most notably through development of gfortran (gcc.gnu.org/wiki/GFortran), a free Fortran compiler that is part of the GNU Compiler Collection (GCC). Finally, because of its role in the early days of computer science, today we have a vast set of robust and mature libraries that have served as the computational backbone of many applications. With mature compilers and a large and trusted legacy code base, Fortran remains the language of choice for many new software projects for which computational efficiency and parallel execution is key.

- **Easy to learn**: Believe it or not, Fortran is quite easy to learn. This was my
experience and the personal experience of many of my colleagues. This is partly
due to Fortran’s strict typing system, which allows the compiler to keep the
programmer in check, and warn them at compile time when they mess up. While
verbose, the syntax is clean and easy to read. However, like every other
programming language or skill in general, Fortran is difficult to master. This is
one of the reasons I chose to write this book.

1.3 Why learn Fortran?

"There were programs here that had been written five thousand years ago, before
Humankind ever left Earth. The wonder of it - the horror of it, Sura said - was that
unlike the useless wrecks of Canberra’s past, these programs still worked! And via
a million million circuitous threads of inheritance, many of the oldest programs
still ran in the bowels of the Qeng Ho system."

-- Vernor Vinge A Deepness in the Sky

Since the early 1990s, we have seen an explosion of new programming languages and
frameworks, mainly driven by the widespread use of the internet, and later, mobile
devices. C++ took over computer science departments, Java has been revered in the
enterprise, Javascript redefined the modern web, R became the mother tongue of
statisticians, and Python rose up as an all-around great programming language for most
tasks. Where does Fortran fit in all this? Through steady revisions of the language,
Fortran has maintained a solid footing in its niche domain, High Performance
Computing (HPC). Its computational efficiency is still unparalleled, with only C and
C++ coming close. However, unlike C and C++, Fortran has been designed for array-
oriented calculations, and is in my opinion significantly easier to learn and program.
Finally, a strong argument for Fortran is in its native support for parallel programming
that was introduced in the 2008 revision of the standard.

What is High Performance Computing?

High Performance Computing (HPC) is the practice of combining computer resources to solve
computational problems that would otherwise not be possible with a single desktop computer. HPC
systems typically aggregate hundreds or thousands of servers and connect them with fast networks.
Most HPC systems today run some flavor of Linux OS.

Despite being a decades-old technology, Fortran has several attractive features that
make it indispensable, even compared to more recent languages:

- **Array-oriented**: Fortran 90 introduced array-oriented syntax and constructs,
  which greatly simplified operations that operated on arrays element-wise.

  Consider the task of multiplying two 2-dimensional arrays:

  ```fortran
  do j = 1, jm
    do i = 1, im
      c(i,j) = a(i,j) * b(i,j)
    end do
  end do
  ```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and
other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/modern-fortran
Since Fortran 90, you can simply do:

\[ c = a \times b \]

This is not only more expressive and readable, but also indicates to the compiler that it can choose the most optimal way to perform the operation. Arrays lend themselves very well to CPU architectures and computer memory because they are designed as contiguous sequence of numbers, and in that sense, mirror the physical layout of the memory space. Fortran compilers are capable of generating extremely efficient machine code because of all the assumptions that they can safely make.

- **The only parallel language developed by a standards committee (ISO):** The Fortran standards committee ensures that the development of Fortran goes in the direction that supports its target audience: computational scientists and engineers.
- **Mature libraries for science, engineering, and math:** Fortran started in the 1950s as the programming language for science, engineering, and mathematics. Decades later, we have a rich legacy of robust and trusted libraries for linear algebra, numerical differentiation and integration, and others. These libraries have been used and tested by generations of programmers, to the point that they are guaranteed to be almost bug-free.
- **Growing general-purpose library ecosystem:** In the past decade, Fortran has also seen a growing ecosystem of general-purpose libraries: text parsing and manipulation, I/O libraries for many data formats, working with dates and times, collections and data structures, and so on. One has even built a web framework as a proof of concept ([fortran.io](https://fortran.io)). I think that any programming language is as powerful as its libraries, and the growing number of Fortran libraries make it more useful today than ever before.
- **Still unmatched performance:** Fortran is still about as close to the metal as it gets with high-level programming languages. This is the case both because of its array-oriented design and mature compilers that are getting increasingly better at optimizing code. If you are working on a problem that involves many mathematical operations on large arrays, few other languages get close to Fortran’s performance.

In summary, learn Fortran if you need to implement efficient and parallel numerical operations on large multi-dimensional arrays.

### 1.4 Advantages and disadvantages

Many Fortran features give it both an advantage and a disadvantage. I list some below:

- **Domain-specific language:** Despite being technically a general-purpose language, Fortran is very much a domain-specific language in the sense that it has been designed for science, engineering, and math applications. If your problem involves some arithmetic on large and structured arrays, Fortran will shine. If you want to
write a web browser or low-level device drivers, Fortran is not the right tool for the task.

- **A niche language**: Fortran is extremely important to a relatively small number of people: scientists and engineers in select disciplines. As a consequence, it may often be difficult to find as many tutorials or blogs about Fortran as there are for more mainstream languages. At the time of this writing, there are a bit over 8,000 questions with Fortran tag on Stack Overflow, a popular programming Q&A website. Contrast this with a whopping 800,000 questions with the Python tag.

- **Statically and strongly typed language**: As I mentioned above, this makes Fortran a very safe language to program in, and helps compilers generate efficient executables. On the flip-side, it makes it less flexible and more verbose, and thus not the ideal language for rapid prototyping.

- **Nothing is a pointer**: Unless you explicitly declare it a pointer. Every variable gets its own space in physical memory. In general, you wouldn’t use pointers in Fortran unless you have to. For example, implementing a linked list requires use of pointers by definition. Pointers are also the only way to create a memory leak in Fortran, making it a relatively safe language.

- **Garbage collection**: Fortran has a basic garbage collection model specified by the standard. Any non-pointer variable is automatically freed from memory once it goes out of scope. However, any pointers must be explicitly dereferenced after their use to avoid the possibility of memory leaks. There is thus some responsibility on you as the programmer to keep track of how pointers are used.

The comparison of Fortran to Python that follows will help you better understand its advantages and disadvantages in the general-purpose programming context.

### 1.4.1 Side-by-side comparison with Python

How does modern Fortran compare to a more recent general-purpose programming language? Python has had the most rapidly growing ecosystem in the past few years for data analysis and light number crunching ([stackoverflow.blog/2017/09/14/python-growing-quickly](https://stackoverflow.blog/2017/09/14/python-growing-quickly)). It is used by many Fortran programmers that I know for post-processing of model output and data analysis. In fact, Python is my second favorite programming language (guess which one is my number one). Because of the application domain overlap between Fortran and Python, it is useful to summarize the main differences between these languages. If you are a Python programmer, this summary will give you an idea of what you can and cannot do with Fortran.

**Table 1.1. Comparison between Fortran and Python features. This table lists only those features available by the core implementation of each language.**

<table>
<thead>
<tr>
<th>Language</th>
<th>Fortran</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>First appeared</td>
<td>1957</td>
<td>1991</td>
</tr>
<tr>
<td>Latest iteration</td>
<td>Fortran 2018</td>
<td>3.6.5 (2018)</td>
</tr>
<tr>
<td>International</td>
<td>ISO/IEC</td>
<td>No</td>
</tr>
</tbody>
</table>

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders. [https://forums.manning.com/forums/modern-fortran](https://forums.manning.com/forums/modern-fortran)
<table>
<thead>
<tr>
<th>Standard</th>
<th>Fortran</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation language</td>
<td>C, Fortran, Assembly (compiler dependent)</td>
<td>C</td>
</tr>
<tr>
<td>Compiled vs. interpreted</td>
<td>Compiled</td>
<td>Interpreted</td>
</tr>
<tr>
<td>Typing discipline</td>
<td>Static, strong</td>
<td>Dynamic, strong</td>
</tr>
<tr>
<td>Parallel</td>
<td>Shared and distributed memory</td>
<td>Shared-memory only</td>
</tr>
<tr>
<td>Multidimensional arrays</td>
<td>Yes, up to 15 dimensions</td>
<td>3rd party library only (numpy)</td>
</tr>
<tr>
<td>First array index</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Intrinsic types</td>
<td>character, complex, integer, logical, real</td>
<td>bool, bytearray, bytes, complex, dict, ellipsis, float, frozense, int, list, set, str, tuple</td>
</tr>
<tr>
<td>Integer kinds</td>
<td>1, 2, 4, and 8 bytes, signed only</td>
<td>2, 4, and 8 bytes, signed and unsigned</td>
</tr>
<tr>
<td>Real / float kinds</td>
<td>4, 8, and 16 bytes</td>
<td>4 and 8 bytes</td>
</tr>
<tr>
<td>Constants</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Pointers</td>
<td>Explicit</td>
<td>Implicit</td>
</tr>
<tr>
<td>Classes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Inheritance</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>Limited</td>
<td>Yes</td>
</tr>
<tr>
<td>Generic programming</td>
<td>Limited</td>
<td>Yes</td>
</tr>
<tr>
<td>Pure functions</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Higher-order functions</td>
<td>Limited</td>
<td>Yes</td>
</tr>
<tr>
<td>Anonymous functions</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Metaprogramming</td>
<td>Preprocessor macros only</td>
<td>Yes</td>
</tr>
<tr>
<td>Garbage collection</td>
<td>None</td>
<td>Optional</td>
</tr>
<tr>
<td>Interoperability with other languages</td>
<td>C (limited)</td>
<td>C (limited)</td>
</tr>
<tr>
<td>OS interface</td>
<td>Limited</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Going through Table 1.1, we notice the key differences between Fortran and Python:

- Fortran is developed by an international standard committee. New language features and programming paradigms are more slowly introduced into revisions of the Fortran standard, but the committee ensures that the usefulness of the language
does not decline for its target audience - scientists and engineers.

- Fortran is compiled and statically typed, while Python is interpreted and dynamically typed. This makes Fortran a bit more verbose and slower to write programs in, but makes it easier for the compiler to generate fast binary code. This is thus a blessing and a curse - Fortran is not designed for rapid prototyping, but allows producing robust and efficient programs.

- Parallelism on both shared and distributed memory computers is native to Fortran. Shared memory parallelism is available in Python using the `multiprocessing` module, however, distributed-memory parallelism is possible only with a third party library that interfaces a message passing protocol written in another language.

- Fortran is array-oriented. Arrays are also where Fortran performs best, as they map well to the layout of elements in memory. In contrast, Python wants little to do with arrays except in special cases. The array-oriented programming model came about from the need of scientists and engineers to apply same arithmetic operations to a large number of elements, and do it fast. The need for blazing fast array operations drove the development of the SIMD (Single Instruction Multiple Data) computing architecture and vector computers in the 1970s, which dominated the supercomputer space through late 1990s. Similarly, GPUs (Graphics Processing Units) were developed with the goal to rotate and translate a large number of small matrices at once. Originally pushed by the video game industry, GPUs are coming back as an important player in general-purpose HPC applications.

- Fortran offers a minimal set of intrinsic types, and most of them are numerical. The standard library lacks common collections and data structures such as lists, dictionaries, and queues. However, it is relatively straightforward to implement these with core Fortran features, as we will learn later in this book. Because of limited types and data structures out-of-the-box, Fortran is not the ideal language for complex business and web applications that operate on unstructured user data in real time. Nevertheless, thanks to the object-oriented features introduced in Fortran 2003 and 2008, several libraries with general-purpose, reusable data structures are now available.

- While Fortran has had a powerful object-oriented programming model since Fortran 2003, it still has limited capability in terms of generic (procedures accepting arguments of any type) and functional programming. For example, while you can pass a function as an argument to another function, it is still not possible to create and return a function object programmatically. Fortran also has an advantage in terms of declaring pure functions, which allows the compiler to execute them in the most efficient way it can find. Inclusion of more advanced programming paradigms into the Fortran standard has been limited to ensure that program performance remains close to that of machine instructions or assembly code.

In summary, it is difficult to use Fortran to write device drivers, graphical video games, or a web browser. However, if you need to solve a large numerical problem that can be
distributed across multiple computers, Fortran is the ideal implementation language.

### 1.5 The parallel Fortran mental model

Let me take a few minutes to illustrate the kind of problem where Fortran really shines.

---

**Summer ends on old Ralph’s farm**

Farmer Ralph has two sons and two daughters, and a big farm. It’s the end of the summer and about time to cut the grass and make hay for the cattle to eat. But the pasture is big and old Ralph is weak. His children, however, are young and strong. If they all work hard and as a team, they could get it done in a day. They agree to split the work between themselves in four equal parts. Each of Ralph’s children grabs a scythe and a fork, and head to their part of the pasture. They work hard, cutting grass row by row. Every hour or so, they meet at the edges to sharpen the tools and chat about how it’s going. The work is going well and almost all of the grass is cut by mid-afternoon. Near the end of the day, they collect the hay into bales and take them to the barn. Old Ralph is happy that he has strong and hard-working children, but even more so that they make such a great team! Working together, they completed work that would take four times as long if only one of them was working.

Now you must be thinking, what the heck does old Ralph’s farm have to do with parallel Fortran programming? More than meets the eye, I can tell you! Old Ralph and his big pasture are an analogy to a slow computer and a big compute problem. Just like Ralph asked his sons and daughters to help him cut the grass, in a typical parallel problem we will divide the computational domain, or input data, into equal pieces and distribute them between CPUs. Recall that his children cut the grass row-by-row — some of the most efficient and expressive Fortran code are the whole-array operations and arithmetic. Periodically, they met at the edges to sharpen the tools and have a chat. In many real-world apps, you will instruct the parallel processes to exchange data between each other, and this is true for all the parallel examples that I will guide you through in this book. Finally, each parallel process will asynchronously write its data to disk. I illustrate this pattern on Figure 1.3.

---
Much like farmer Ralph, Fortran is old. This is by no means a bad thing! It is a mature, robust, and dependable language that isn’t going anywhere. While it does carry some quirks of an old programming language, it has been improved decade over decade by generations of computer scientists and programmers, and has been battle-tested in countless applications where performance is critical. The ease of parallel programming with Fortran is key for high-performance apps, which is why I chose to make it the focus of this book.

1.6 What will you learn in this book?

This book will teach you how to write modern, efficient, and parallel Fortran programs. Working through each chapter, we will build from scratch a fully-functional, parallel, fluid dynamics solver with a specific application to tsunami prediction. If you work through the book, you will come out with three distinct skill sets:

- You will be fluent with most modern Fortran features. This is a unique and desired skill in a robust and niche market that is HPC.
- You will be able to recognize problems that are parallel in nature. You will think parallel-first, and parallel solutions to problems will seem intuitive. In contrast, a serial solution to a parallel problem will become just an edge-case scenario.
- You will get a grasp on good software design, including design patterns, unit and regression testing, documenting the code, and sharing your project with the online community.
community. You will also be able to adapt existing Fortran libraries in your project and contribute back. This will not only make your project useful to others, but can open doors in terms of career and learning opportunities. It did for me!

In this book, I assume that you have at least some programming experience, and understand basic concepts like variables, loops, and branches. Ideally, you already have coded basic scripts in Python or MATLAB. Since our running example is centered around solving a system of partial differential equations, it is helpful if you have some knowledge of calculus and linear algebra. We will also be working a lot in the terminal, so some experience with Linux or UNIX-like shell is expected. Given the topic of the book, I expect that this book will be ideal for:

- Undergraduate and graduate students in physical science, engineering, or applied math, especially with focus on fluid dynamics
- Instructors and researchers in the above fields
- Meteorologists, oceanographers, and other fluid dynamicists working in the industry
- Serial Fortran programmers who want to step up their parallel game
- HPC system administrators

If you fit in one of the above categories, you may already know that Fortran’s main selling point is its ease of programming efficient and parallel programs for large supercomputers. This has kept it as the dominant HPC language of physical sciences and engineering. While this book will teach you Fortran from the ground up, I will also take the unconventional approach and teach it in the context of parallel programming from the get go. Rather than gaining just another technical skill as an afterthought, you will learn how to think parallel. You will recognize ways in which the workload and memory can be distributed to arrive at the solution more efficiently. With parallel thinking, you will come out with two critical advantages:

1. You will be able to solve problems in less time.
2. You will be able to solve problems that can’t fit into a single computer.

The first is a definite nice-to-have, but the second is a deal-breaker. Some problems simply can’t be solved without parallel programming. Next section will give you a gentle introduction and an example of parallel programming.

## 1.7 Think parallel!

"For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution."

-- Gene Amdahl, (computer architect) in 1967

Parallel programming is only becoming more important with time. The rate of semiconductor density increase, as described by Moore’s law, while still positive,
limited. Traditionally we went past this limit by placing more processing cores on a single die. Even the processors in most smartphones today are multicore. Beyond the shared-memory computer, we have connected many machines using sophisticated networks, and made them talk to each other to solve huge computational problems. As I mentioned earlier, the weather forecast that you saw this morning on your favorite TV channel or news website was computed on hundreds or thousands of parallel processors. Due to the practical limits of Moore’s law and the current tendency toward many-core architectures, there is a sense of urgency to teach programming parallel-first.

**What is Moore's law?**
Gordon Moore, the cofounder of Intel, noticed in 1965 that the number of transistors in a CPU was doubling each year. He later revised this trend to a doubling every two years, but nevertheless, this kind of rate of increase is exponential. This trend is closely related to a continuous decrease in cost of computers. For example, a computer you buy today for $1000 is about twice as powerful as a computer you could buy for the same amount two years ago.

Similarly, when you buy a new smartphone, the OS and the apps seem smooth and fast. What happens two years later? As the apps update and get new features, they demand increasingly more CPU cycles and memory. As the hardware in your phone stays the same, eventually the apps slow down to a creep.

All parallel problems fall into two categories:

1. **Embarrassingly parallel**: Here, by "embarrassingly" we actually mean "embarrassingly easy" - it’s a good thing! These are problems that can be distributed across processors with little to no effort (Figure 1.4, left). In general, any function $f(x)$ that operates element-wise on an array $x$ without need for communication or synchronization between elements is embarrassingly parallel. Because the domain decomposition of embarrassingly parallel problems is trivial, modern compilers are capable of auto-parallelizing such code in most cases. Real world examples include graphics rendering, serving static websites, or processing a large number of independent data records.

2. **Non-embarrassingly parallel**: Any parallel problem in which there is inter-dependency between processing elements, which requires communication and synchronization (Figure 1.4, right). Most partial differential equations solvers are non-embarrassingly parallel. The relative amount of communication versus computation dictates how well a parallel problem will scale. The objective for most physical solvers is thus to minimize communication and maximize computation. Real world examples include modeling fluid flows, molecular dynamics, or any other physical process that can be described by partial differential equations. This class of parallel problems is more difficult, and in my opinion, more interesting!

*Figure 1.4. A schematic of an embarrassingly parallel problem (left) and a non-embarrassingly parallel problem (right). In both cases, the CPUs receive input $(x_1, x_2)$ and process it to produce output $(y_1, y_2)$. In an embarrassingly parallel problem, $x_1$ and $x_2$ can be processed independently.*
of each other. Furthermore, both input and output data are local in memory to each CPU, indicated by solid arrows. In a non-embarrassingly parallel problem, input data is not always local in memory to each CPU and has to be distributed through the network, indicated by dashed arrows. In addition there may be data inter-dependency between CPUs during the computation step, which requires synchronization (horizontal dashed arrow).

Why is it called embarrassingly parallel?
It refers to overabundance, as in embarrassment of riches. It’s the kind of problem that you want to have. The term is attributed to Cleve Moler, inventor of MATLAB and one of the authors of EISPACK and LINPACK, Fortran libraries for numerical computing. LINPACK is still used to benchmark the fastest supercomputers in the world.

Because our application domain deals mainly with non-embarrassingly parallel problems, we will focus on how to implement parallel data exchange between processors in a clean, expressive, and minimal way. This will involve both distributing the input data among processors (downward dashed arrows in Figure 1.4), and exchanging the data between them whenever there is inter-dependency (horizontal arrow in Figure 1.4).

Parallel Fortran programming in the past has been done either using the OpenMP directives for shared-memory computers only, or with the Message Passing Interface (MPI) for both shared and distributed memory computers. Differences between shared-memory (SM) and distributed-memory (DM) systems are illustrated on Figure 1.5. The main advantage of SM systems is very low latency in communication between processes. However, there is a limit to the number of processing cores that can exist in a SM system. Since OpenMP was designed for SM parallel programming exclusively, we will focus on MPI for our specific example below.

Figure 1.5. Shared-memory (left) versus distributed-memory (right) system. In a shared-memory system, processors (orange) have access to common memory (RAM, purple). In a distributed-memory system, each processor has their own memory, and exchange data through a network,
indicated by dashed lines. The distributed-memory system is most commonly composed of multicore shared-memory systems.

### OpenMP versus MPI

OpenMP is a set of directives that allow the programmer to indicate to the compiler the sections of the code that are to be parallelized. OpenMP is implemented by most Fortran compilers and does not require external libraries. However, OpenMP is limited to shared-memory machines.

Message Passing Interface (MPI) is a standardized specification for portable message passing (read: data copy) between arbitrary remote processes. This means that MPI can be used for multi-threading on a single core, multicore processing on a shared-memory machine, or distributed-memory programming across networks. MPI implementations typically provide interfaces for C, C++, and Fortran. MPI is often described as the assembly language of parallel programming, illustrating the fact that most MPI operations are low-level.

#### 1.7.1 Copying an array from one processor to another

In most scientific and engineering parallel applications, there is data dependency between computational processes. Typically, a 2-d array is decomposed into tiles like a chess board, and the workload of each tile is assigned to a processor. Each tile has its own data in memory that is local to its processor. To illustrate the simplest case of parallel programming in a real-world scenario, let’s take the following meteorological situation for example. Suppose that the data consists of two variables, wind and air temperature. Wind is blowing from one tile with lower temperature (cold tile) toward another tile with higher temperature (warm tile). If we were to solve how the temperature evolves in time, the warm tile would need to know what temperature is coming in with the wind from the cold tile. Because this is not known a priori (remember that the data is local to each tile), we need to copy the data from the cold tile into the memory that belongs to the warm tile. On the lowest level, this is done by explicitly copying the data from one processor to another. When the copy is finished, the processors can continue with the remaining computations. Copying an array from one process to another is the most common operation done in parallel
programming (Figure 1.6).

**Figure 1.6.** An illustration of a remote array copy between two CPUs. The numbers inside the boxes indicate initial array values. Our goal is to copy values of array from CPU 1 to CPU 2.

Since we’re barely starting, let’s focus on getting just this one operation done. Our goal is to do the following:

1. Initialize array on each process - [1, 2, 3, 4, 5] on CPU 1 and all zeros on CPU 2.
2. Copy values of array from CPU 1 to CPU 2.
3. Print the new values of array on CPU 2. These should be [1, 2, 3, 4, 5].

I will show you two examples of how to solve this problem. One is the traditional approach using an external library like MPI. Unless you’re a somewhat experienced Fortran programmer, don’t try to grok this example. I merely want to demonstrate how complicated and verbose this approach is. Then, I will show you the solution using the new Fortran Coarray approach. In contrast to MPI, with coarrays you can use an array indexing-like syntax to perform remote data exchange between parallel processes.

**MPI: The traditional way of parallel programming**

MPI has been often described as the assembly of parallel programming, and indeed, that was developers' original intention! The main vision of MPI was to be implemented by compiler developers to enable native parallel programming languages. However, over the past three decades, application developers were much faster at adopting MPI directly in their programs, and MPI has become, for better or for worse, a de facto standard tool for parallel programming in Fortran, C, and C++. As a result, most HPC applications today still rely on low-level MPI calls.

Below is a Fortran program that sends data from one process to another using MPI:

```fortran
program array_copy_mpi
use mpi
implicit none
integer :: ierr, nproc, procsze, request
integer, dimension(mpi_status_size) :: stat
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders. https://forums.manning.com/forums/modern-fortran
integer, dimension(5) :: array
integer, parameter :: sender = 0, receiver = 1

call mpi_init(ierr)  
call mpi_comm_rank(mpi_comm_world, nproc, ierr)  
call mpi_comm_size(mpi_comm_world, procsyze, ierr)  

if (procsyze /= 2) then  
call mpi_finalize(ierr)  
stop 'Error: This program must be run on 2 parallel processes'  
end if  

if (nproc == sender) then  
array = [1, 2, 3, 4, 5]  
else  
array = 0  
end if  

write(*,'(a,i1,a,5(4x,i2))')'array on proc ', nproc, &  
' before copy:', array  

call mpi_barrier(mpi_comm_world, ierr)  

if (nproc == sender) then  
call mpi_isend(array, size(array), mpi_int, receiver, 1, &  
mpi_comm_world, request, ierr)  
else if (nproc == receiver) then  
call mpi_irecv(array, size(array), mpi_int, sender, 1, &  
mpi_comm_world, request, ierr)  
call mpi_wait(request, stat, ierr)  
end if  

write(*,'(a,i1,a,5(4x,i2))')'array on proc ', nproc, &  
' after copy: ', array  

call mpi_finalize(ierr)  

end program array_copy_mpi  

1 Access MPI subroutines and mpi_comm_world global variable from a module  
2 Initialize MPI  
3 Which processor number (nproc) am I?  
4 How many total processes are there?  
5 Shut down MPI and stop the program if we are not running on 2 processors  
6 Initialize array on sending process  
7 Initialize array on receiving process  
8 Print text to screen with specific formatting  
9 Wait here for both processes  
10 Sender posts a non-blocking send  
11 Receiver posts a non-blocking receive  
12 Receiver waits for the message  
13 Finalize MPI at the end of the program
Running the program on 2 processors outputs the following:

<table>
<thead>
<tr>
<th>Listing 1.4. Output of array_copy_mpi program.</th>
</tr>
</thead>
<tbody>
<tr>
<td>array on proc 0 before copy: 1 2 3 4 5</td>
</tr>
<tr>
<td>array on proc 1 before copy: 0 0 0 0 0</td>
</tr>
<tr>
<td>array on proc 0 after copy: 1 2 3 4 5</td>
</tr>
<tr>
<td>array on proc 1 after copy: 1 2 3 4 5</td>
</tr>
</tbody>
</table>

The above output confirms that our program did what we wanted - copy the array values from process 0 to process 1.

Compiling and running this example

Don't worry about building and running this example yourself for the time being. In the next chapter, you will set up the complete compute environment for working with examples in this book, including this one.

Enter Fortran Coarrays

Coarray Fortran (CAF) is the native Fortran model for parallel programming. Originally developed by Robert Numrich and John Reid in the 1990s as an extension for the Cray Fortran compiler, CAF has been introduced into the standard starting with Fortran 2008 revision. Coarrays are very much like arrays, as the name implies, except that their elements are distributed along the axis of parallel processes (think cores or threads). As such, they provide an intuitive way to send and receive data between remote processes.

What follows is the coarray implementation of our array copy example:

<table>
<thead>
<tr>
<th>Listing 1.5. Copying an array from one process to another using coarrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>program array_copy_caf</td>
</tr>
<tr>
<td>implicit none</td>
</tr>
<tr>
<td>integer, dimension(5), codimension[*] :: array</td>
</tr>
<tr>
<td>integer, parameter :: sender = 1, receiver = 2</td>
</tr>
<tr>
<td>if (num_images() /= 2) then</td>
</tr>
<tr>
<td>stop 'Error: This program must be run on 2 parallel processes'</td>
</tr>
<tr>
<td>end if</td>
</tr>
<tr>
<td>if (this_image() == sender) then</td>
</tr>
<tr>
<td>array = [1, 2, 3, 4, 5]</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>array = 0</td>
</tr>
<tr>
<td>end if</td>
</tr>
<tr>
<td>write(*,'(a,i2,a,5(4x,i2))')'array on proc ', this_image(),&amp;</td>
</tr>
<tr>
<td>' before copy:', array</td>
</tr>
<tr>
<td>sync all</td>
</tr>
</tbody>
</table>
if (this_image() == receiver) array(:) = array(:)[1] 

write(*,'(a,i1,a,5(4x,i2))')'array on proc ', this_image(),&
  ' after copy: ', array

end program array_copy_caf

1 Declare an integer coarray
2 Throw an error if we are not running on 2 processes
3 Coarray image indices start at 1
4 Wait here for all images; equivalent to mpi_barrier()
5 Non-blocking copy from sending image to receiving image

The output of the program is the same as in the MPI variant:

<table>
<thead>
<tr>
<th>Listing 1.6. Output of array_copy_caf program.</th>
</tr>
</thead>
<tbody>
<tr>
<td>array on proc 1 before copy: 1 2 3 4 5</td>
</tr>
<tr>
<td>array on proc 2 before copy: 0 0 0 0 0</td>
</tr>
<tr>
<td>array on proc 1 after copy: 1 2 3 4 5</td>
</tr>
<tr>
<td>array on proc 2 after copy: 1 2 3 4 5</td>
</tr>
</tbody>
</table>

These two programs are thus semantically the same. Let’s look at the key differences:

- The number of lines of code (LOC) dropped from 30 in the MPI examples to 17 in the coarray example. This is almost a factor of 2 decrease. However, if we look specifically for MPI-related boilerplate code, we can count 15 lines of such code. Compare this to 2 lines of coarray-related code! As debugging time is roughly proportional to the LOC, we see how Coarray Fortran will be much more cost-effective for development of parallel Fortran applications.

- The core of the data copy in MPI example is quite verbose for such a simple operation:

```
if (nproc == 0) then
  call mpi_isend(array, size(array), mpi_int, receiver, 1,&
    mpi_comm_world, request, ierr)
else if (nproc == 1) then
  call mpi_irecv(array, size(array), mpi_int, sender, 1,&
    mpi_comm_world, request, ierr)
  call mpi_wait(request, stat, ierr)
end if
```

compared to the intuitive array-indexing and assignment syntax of coarrays:

```
array(:)[2] = array(:)[1]
```
• Finally, MPI needs to be initialized and finalized using `mpi_init()` and `mpi_finalize()` subroutines. Coarray Fortran needs no such code. This one is minor, but a welcome improvement!

**Parallel process indexing**

Did you notice that our parallel processes were indexed 0 and 1 in the MPI example and 1 and 2 in the coarray example? This is because MPI is implemented in C, in which array indices begin at 0. In contrast, coarray images start at 1 by default.

As we saw in this example, both MPI and CAF can be effectively used to exchange data between parallel processes. However, MPI code is low-level and verbose, and would soon become tedious and error-prone as the complexity of our app increases. In contrast, CAF offers an intuitive indexing syntax that is analogous to the familiar operations with arrays. Furthermore, with MPI, you tell the compiler *what to do*; with CAF, you tell the compiler *what you want*, and let it decide what’s the best way to do it. This approach relieves a big deal of responsibility off your shoulders, and lets you focus on your application. I hope that this convinces you that Fortran coarrays are the way to go for expressive and intuitive implementation of data exchange between parallel processes.

### 1.8 A Partitioned Global Address Space language

Fortran is also a Partitioned Global Address Space (PGAS) language. In a nutshell, PGAS abstracts the distributed-memory space and allows you to:

1. View the memory layout as a shared-memory space: This will give you a tremendous boost in productivity and ease of programming when designing parallel algorithms. When performing data exchange, you won’t need to translate or transform array indices from one image to another. In other words, the memory spaces that belong to remote images will *appear* as they are local, and you will be able to express your algorithm in such way.

2. Exploit the locality of reference: In simpler words, design and code your parallel algorithms without foresight about whether a subsection of memory is local to the current image or not. If it is, the compiler will use that information to its advantage. If it is not, the most efficient data exchange pattern available will be performed.

For example, with Fortran Coarrays, PGAS allows you to use one image to initiate a data exchange pattern between two remote images:

**Listing 1.9. From image 1, initiate a remote copy of `array` from image 8 to image 7.**

```fortran
if (this_image() == 1) array(:)[7] = array(:)[8]
```

In this snippet, the `if`-statement ensures that the assignment executes only on image 1. However, the indices inside the square brackets refer to images 7 and 8! What this means is that the image 1 will asynchronously request an array copy from image 8 to
The power of PGAS is that, from the programmer’s point of view, the indices inside the square brackets can be treated just like any other array elements that are local in memory. However, in practice, these images could be mapped to different cores on the same shared-memory computer, or across the server room and connected via the local interconnect, or even across the world and connected through the internet!

Other notable PGAS languages are Chapel (chapel-lang.org) and Unified Parallel C (upc-lang.org).

1.9 **Running example: A parallel tsunami simulator**

I believe that most learning happens by doing rather than reading, especially if immersed in a longer-term project. Lessons in this book are thus framed within the context of developing your own, fully featured, parallel app.

1.9.1 **Why tsunami simulator?**

A *tsunami* is a series of long water waves that are triggered by a displacement of a large body of water. This typically occurs due to earthquakes, underwater volcanoes, or landslides. Once generated, a tsunami propagates radially outward and grows in height and steepness as it enters shallow water. I think a tsunami simulator is a good running example for this book because tsunamis are:

- **Fun**: Speaking strictly as a scientist here! Tsunami is a process that is fun to watch and play with in a numerical sandbox.
- **Dangerous**: Tsunamis pose a great threat to low-lying and heavily populated coastal areas. There is thus a great need to understand and predict them better.
- **Simple math**: Can be simulated using a minimal set of equations - the so-called shallow water equations. This is important so that we don’t get bogged down by the math and focus on implementation instead.
- **Parallelizable**: A physical process that is suitable for teaching parallel programming, especially considering that it is a non-embarrassingly parallel problem. To get it to work in parallel, we will need to carefully design data exchange patterns between images.

To simulate tsunamis, we will write a solver for the shallow water system of equations.

1.9.2 **Shallow water equations**

Shallow water equations (SWE) are a simple system equation derived from Navier-Stokes equations. They are also known as the Saint Venant equations, after the french engineer and mathematician A. J. C. Barre de Saint-Venant, who derived the 1-d form from first principles in pursuit of his interest in hydraulic engineering and open-channel flows. SWE are powerful because they can reproduce many observed motions in the atmosphere and the ocean:

- Large-scale weather such as cyclones and anticyclones
Western boundary currents such as Gulf Stream in the Atlantic and Kuroshio current in the Pacific
Long gravity waves such as tsunami and tidal bores
Watershed from rainfall and snow melt over land
Wind-generated (surf) waves
Ripples in a pond

SWE system consists of only a few terms:

**Figure 1.7. Shallow water equations.** Top equation is the momentum (velocity) conservation law, and the bottom is mass (water level) conservation law. $u$ is the 2-d velocity vector, $g$ is the gravitational acceleration, $h$ is the water elevation, $H$ is the unperturbed water depth, and $t$ is time. The "nabla" symbol (upside-down triangle) is a vector differentiation operator.

\[
\begin{align*}
\text{velocity tendency} & \quad \frac{\partial u}{\partial t} + u \cdot \nabla u = -g \nabla h \\
\text{water height tendency} & \quad \frac{\partial h}{\partial t} = -\nabla \cdot (u(H + h))
\end{align*}
\]

What is the physical interpretation of the above system? The top equation states that where there is slope along the water surface, water will accelerate and move from region of higher level to region of lower level due to pressure gradient. The advection term is non-linear and causes chaotic behavior in fluids, known as turbulence. The bottom equation states that if there is an area where velocity is converging (coming together), there will be an increase in water level because water has to go somewhere - this is why we call it conservation of mass. Similarly, if the velocity is diverging (moving apart), there will be a decrease in water level in response.

**Comfortable with math?**

If you’re experienced with calculus and partial differential equations, great! There is more for you in Appendix B. Otherwise, don’t worry! This book won’t dwell on math much more than this, and will focus instead on implementation and Fortran programming.

Shallow water equations are dear to me because I first learned Fortran programming by modeling these equations in my undergraduate meteorology program at the University of Belgrade. Despite my Fortran code looking (and working) much differently now than back then, I still find this system of equations an ideal case for teaching parallel Fortran programming. I hope you enjoy the process as much as I do!
1.9.3  What we want our app to do

Let’s decide on some requirements for the features of our Tsunami Simulator:

- **Parallel**: The model can scale to hundreds of processors with nothing but pure Fortran code. This is not only important for speeding up the program and reducing run-time, but also for enabling very large simulations that otherwise would not fit into the memory of a single computer. With almost all modern laptops having at least 2 processing cores, most readers should be able to enjoy the fruits of their (parallel programming) labor.

- **Extensible**: Physics terms can be easily formulated and added to the solver. This is important for the general usability of the model. If we can design our computational kernel in form of reusable classes and functions, new mathematical terms can be easily added as functional, parallel operators, following the approach by Damian Rouson
  (www.lanl.gov/conferences/salishan/salishan2014/rouson.pdf). We could code our equations from Figure 1.7 as:
   - Momentum balance:
     \[ \frac{du}{dt} = -u \cdot \text{dot} \cdot (\text{grad} \cdot u) - g \cdot (\text{grad} \cdot h) \]
   - Mass balance:
     \[ \frac{dh}{dt} = -\text{div} \cdot (u \cdot (h + h_{\text{mean}})) \]

In the above snippets, parallel decomposition and data exchange would be implemented inside the operators .dot., .grad., and .div., which correspond to dot product, gradient, and divergence operators, respectively. This way, the technical implementation is encapsulated inside these functions, and on high-level we would be able to code our equations much like we would write them on a blackboard.

- **General**: Can be run in idealized experiments, for example with flat bottom and periodic boundary conditions, as well as on realistic domains with ocean bathymetry from input data.

- **Easy to use**: The model can be configured via command line parameters, like common Linux tools:

  $ \text{tsunami} --\text{amplitude}=3 --\text{duration}=12 --\text{output}=\text{netcdf}$

- **Software library**: Provides a reusable set of classes and functions that can be used to build other parallel models.

- **Useful code documentation**: All software should be useful, and no software user should have to guess what did the original author of the program intended. We will write our app in such a way that great code documentation can be auto-generated. We often call this self-documenting code.

- **Discoverable online**: Writing a program just for yourself is great for learning and discovery. However, software becomes really useful once you can share it with others who can use it to solve their problems. I will teach you how to put your app...
out there in the wild, make it easy to discover, and make it attractive to other contributors. Other people fixing my bugs and implementing features from my todo list? Yes, please!

If you haven’t already, I encourage you to go ahead and check out the code for the running example from Github:

git clone https://github.com/modern-fortran/tsunami

Once you got it, take a look around, explore, peek inside the source files. It’s the project that we will build together. We start the next chapter by setting up the development environment so you can compile and run the minimal working version of our app.

1.10 Summary

In this chapter you learned that Fortran is:

- One of the first high-level programming languages in history.
- Still the dominant technology for many applications in science and engineering.
- The only standardized language with a native model for parallel programming.
- In the concrete example of array copy from one parallel process to another, you learned that Fortran Coarrays are ideal for clean and expressive implementation of parallel algorithms.
- Robust, efficient, and easy to program.

Fortran is not a language for everybody. It is definitely not a systems programming language, nor a web development language. Programming a graphical video game or a web browser in Fortran is possible, but extremely difficult. However if you are working on computationally intensive problems in science or engineering, it may be exactly what you need. Modern Fortran will take you on a journey through core Fortran features from a parallel-first perspective. Where applicable, you will also apply object-oriented and functional techniques, as well as adapt existing Fortran libraries into your application. By working through this book chapter by chapter, you will gain the experience of developing a fully-featured parallel app from scratch. If it’s your first software project, I hope it excites your inner software developer and inspires you to go make something on your own.