Scientific High Performance Computing

A Comprehensive Introduction using Modern Techniques

Fatih ERTINAZ
fertinaz@gmail.com

April 6, 2021

Contents

Chapter 1

Introduction

This book focuses on the fundamentals of Scientific Computing from two different perspectives; Computer Science and Applied Mathematics. Initially, we provide information about Computer Science related areas such as fundamentals of High-Performance Computing, `Linux`, programming languages and compilers. Then, we delve into mathematical modelling and scientific applications. Finally, we provide an analysis of the current advances in Cloud Computing, and explain its potential impact on `HPC` workloads.

Each chapter is devoted to a specific field and gives detailed explanation about its subject. On the other hand, each of these subject are very broad to cover all of their aspects within a chapter. Therefore, we highly suggest interested readers to refer to other references for a complete study. The main intention in this book is to explain how different components of an HPC focused system should be designed for efficient solutions to computationally demanding problems.

Current chapter contains a summary of each section. Picky readers can have a look at this part to gather a general idea about a certain section if they don't want to read the whole book. However, it is encouraged to follow the order in the book since chapters are connected to each other.

Chapter 2 begins with clarification of the fundamental concept in this book; High-Performance Computing which is also referred as `HPC`. A detailed explanation to the questions like "What is HPC" or "Why/When is HPC needed" can be found in this section. This chapter also introduces the "cluster" concept, and gives information about hardware and software components of HPC clusters.

Chapter 3 is about operating systems, however it specifically focuses on `Linux`. This chapter starts with a historical background, then briefly ex-

plains the evolution of `Linux`. Additionally, some basic information about kernel is given in this chapter. Moreover, chapter continues with the shell environment which is probably the most important tool that an `HPC` user should be comfortable with.

Chapter 4 is an introduction to the basic programming concepts based on commonly used languages like `C`, `C++` and `Python`.

Chapter 5 is devoted to how compilers work. Therefore, this chapter can be thought as a bridge between the two concepts previously discussed; operating systems and programming. Our main focus is on the GCC, however other compiler suites are discussed as well. Other than the compilers, we talk about debuggers and profilers too.

Chapter 2

High Performance Computing

This chapter starts with a clear description of High Performance Computing (HPC), and then explains the scientific aspects of it. Then, we explain how HPC solutions are architected by providing information about key components of typical HPC requirements.

2.1 What is HPC?

HPC is an ambiguous term unless an explicit definition is given. While an enterprise might use to represent a mainframe system that handles millions of financial transactions per second, a digital artist can refer to a workstation that renders gigabytes of videos. A general definition can be "A complex execution that is invoked to calculate a result" (REF HERE!!!). According to this definition, there are two key properties we can emphasize;

- Life-cycle: Unlike services or daemons, we expect HPC applications to end once computations are done.
- Complexity: HPC applications should handle complicated tasks that would normally be either impossible or at least be very hard to complete on a typical computer.

Based on the first item above a very basic example can be a small program that computes the area of a square from a given input. This code can take an input from the user, then executes the algorithm which is required to calculate the result. When it is complete, OS terminates it.

However if an HPC application becomes this simple, then we might have to refer all applications as HPC application which becomes odd. To narrow it down, we consider the second item above and assume that the calculation made by the computer should be based on an complicated algorithm

such as solving turbulent wind flow over a hilly terrain, prediction of cavity bubbles around rotating ship propellers or accretion disks between compact astrophysical objects.

All these examples require implementation of complicated algorithms, but at the end what they do is getting an input and applying the algorithm required to produce a result. Once result is calculated, execution is terminated by the operating system on the platform they run. Such applications can be regarded as High Performance Computing applications.

2.2 What is Scientific HPC?

Since examples above are scientific applications, we can also refer them as Scientific High Performance Computing applications. As a result, we avoid other HPC applications such as Video Analytics or Large DB query handling systems in our definition. Consequently our focus will be on the details of Scientific HPC in the rest of this book.

2.3 HPC Clusters

2.3.1 Hardware: Fabrics, schedulers, storage etc.

2.3.2 Software: Management, security, modules etc.

Chapter 3

Linux Operating Systems and Shell

`Linux` is an operating system (OS) family. There are many different versions called distro, however they all use `Linux Kernel`. Therefore one can say that a `Linux` system is based on three components;

- Kernel space: The part that talks to hardware and handles resource management.
- File system: File organization for efficient use of the system.
- User environment: All the other software and tools that is needed by users.

Linux kernel is developed by Linus Torvalds in 1991. He kept it open source and since then Linux became one of the largest and perhaps the most successful open source projects. Linus is still leading kernel development, however there are thousands of developers contributing it from all around the world.

3.1 Why is Linux Important?

`Linux` systems are widely used by HPC centres. In fact as of August 2019 we can say that it is the only OS type currently being used by `Top-500` clusters. (REF HERE!!!).

Obviously this is not a coincidence, there are a few reasons for it:

- `Linux` systems are free: Cost is always an issue.
- `Linux` systems are configurable: Users, groups and resources are all customizable.
- Most researchers are also working on `Linux`.

- `Linux` systems are more secure - because they're configurable.

3.1.1 A Brief History of Operating Systems

To better understand the importance of Linux, we need to know its evolution. Hence, we should start with some historical background.

In the late 60s, two computer scientists Dennis Ritchie and Ken Thompson developed the first version of `Unix` at AT&T Bell Laboratories. `Unix` was the first major multiuser and multitasking fully compatible OS. Its predecessor `MULTICS` (Multiplexed Information and Computing Service) was developed by GE & MIT and AT&T, but it was a failure. Thompson derived some ideas from `MULTICS` project and merged it with the `C` language developed by Dennis Ritchie. As a result `Unix` was born.

70s were incredible productive in terms of OS development. Since AT&T was unable to sell `Unix` sell as a product due to the contract they made with government, they distributed to universities across the world. This lead many contributions to `Unix`. When Ken Thompson was a visiting professor in California, he continued the development of `Unix` project and made significant extensions to it with his students. Hence, `BSD` was born.

3.2 Linux Internals and Environment

This book is not intended for explaining kernel details, however it is worth mentioning some of the basics.

3.3 Shell Scripting and Common Commands

From a user perspective, perhaps the biggest advantage of using a Linux system would be the flexibility provided by shell.

Chapter 4

Programming Languages

Hope this day will come...

4.1 Basics of Programming

4.1.1 Communication between software and hardware

4.2 Efficient Programming

4.3 Scalable Programming in Distributed Environments

4.4 Code Portability

Chapter 5

Compilers

This section describes details about how a compiler works in a `Linux` environment. Our specific focus will be on the GNU Compiler Collection (GCC), however general idea can be applied to other compilers as well.

GCC is not the only compiler in the computer industry. There are lots of other compilers as well. To name a few:

- `CC`: C compiler. Developed by Denis Ritchie as a part of `Unix` project. It usually comes as the default compiler with `Linux` distributions.
- `Visual Studio`: Microsoft's compiler suite for `Windows` platform.
- `Clang`: C and C++ compiler for Mac.
- `ICC`: Intel Compiler Suite.
- `XL`: IBM compilers.

All these compilers and collection of compilers have pros and cons. Some of them specifically designed for certain platforms and cannot be used on hardware or software other than the ones that they are designed for.

For the purpose of this book `GCC` is the best option. Unlike `ICC` or `XL`, it is not hardware specific. It supports almost every industry standard hardware, therefore readers are not restricted to certain hardware types. Also, it is free to use. It is not completely open source like `Clang`, but interested readers can refer to the `GitHub` repository for publicly available libraries. Most of the `Linux` distribution is shipped with a `GCC` compiler, but it is always an option to compile and build `GCC` from scratch.

For these reasons, reader can always assume the compiler used in this book is `GCC` unless otherwise stated explicitly.

5.1 What is a Compiler?

Following description is from the book "An Introduction to GCC" by Brian Gough; *Compiler is a software which converts a program from the textual source code, in a programming language such as C or C++, into machine code, the sequence of 1's and 0's used to control the central processing unit (CPU) of the computer* ([1], p. 7).

In the same reference ([1], p. 7), following source code is given:

```
#include <stdio.h>
int main (void)
{
    printf ("Hello, world!\n");
    return 0;
}
```

This is a very simple `C` code, and its expected behaviour is printing "Hello, world!" to the screen. Let's assume that above source code is saved in a file called `hello.c`. If we run the command below in a terminal, this code gets compiled and an output executable is generated. When we run this executable file, we get "Hello, world!" printed to the screen:

```
$ gcc -Wall hello.c -o hello
$ ./hello
Hello, world!
```

By running the `gcc` command, we use `GCC`. Although `GCC` is a collection of compilers, `gcc` command can be used to compile codes written in `C` language.

In the command above, we set the flag `-Wall` which is one of the many options of `gcc` and it means "print all warnings occurred during compilation". It should always be enabled for a cleaner code. Then we specify name of our source code and provide an output name after the `-o` option. Note that this is an optional flag. We could have used the following command as well:

```
$ gcc -Wall hello.c
```

which spits out the following files:

```
$ ls -l
-rw-r--r--  1 ertinaz  staff    87 28 Aug 16:20 hello.c
-rwxr-xr-x  1 ertinaz  staff  8432 28 Aug 16:54 a.out
$ ./a.out
Hello, world!
```

When an output name is not given compiler produces a file called `a.out` by default. It is same as the previous executable file, however providing an

output name is much more convenient than the default naming.

We will get into details about what goes on behind the scenes in the following sections, but it is apparent that understanding the mechanism behind compilation is a key factor for improving your skills to be able to use resources more efficiently..

5.1.1 Why is it important to know how compilers work?

Compiler is a very crucial aspect of your environment since it is the tool that converts your source code to the binary executed by the machine. Therefore one can make a nicely written code perform even better using the correct compiler options. On the contrary choosing inconvenient compiler flags would result in a poorly performing output.

5.2 How do compilers work?

Compilers are sophisticated tools. On the surface there are five major processes when one compiles a programme:

- Pre-process: Merge all source code, e.g. include header files, add macros etc.
- Convert to assembly
- Create object files
- Link shared libraries
- Generate binary executable

While this process depends on the type of compilation...

5.2.1 Optimization using Compilers

Compilers consist of three components:

- Front-end: Modules that applies pre-processor.
- Back-end: Part that applies conversion from assembly to executable binary.
- Optimizer: Simply the performance enhancer. This is where the magic happens.

To prove my point please check out the following example. This source code is from ([1], p. 51):

```c
#include <stdio.h>

double powern (double d, unsigned n)
{
  double x = 1.0;
  unsigned j;

  for (j = 1; j <= n; j++)
    x *= d;

  return x;
}

int main (void)
{
  unsigned i;

  double sum = 0.0;
  for (i = 1; i <= 100000000; i++)
    sum += powern (i, i % 5);

  printf ("sum = %g\n", sum);

  return 0;
}
```

Details can be read in the reference. In short this code computes the $n-th$ power of a floating point number by repeated multiplication. Below you'll find execution times of this code where two different optimization levels are used:

```
$ gcc -Wall -O0 powern.c -lm -o powern-O0.x
$ time ./powern-O0.x
sum = 4e+38

real 0m0.869s
user 0m0.858s
sys  0m0.004s
```

In this attempt, we enforce no-optimization by using `-O0` flag. This is equivalent of not specifying a `-Olevel` option. When we execute the binary compiled this way, we complete execution in 0.858 seconds of CPU run-time. However, as you can see below just changing optimization level from `-O0` to `-O3` and introducing a loop unrolling based optimization, we can reduce run-time to 0.150 seconds which is almost 17.5% of the previous run-time.

```
$ gcc -Wall -O3 -funroll-loops powern.c -lm \
    -o powern-O3.x
$ time ./powern-O3.x
```

```
sum = 4e+38

real  0m0.156s
user 0m0.150s
sys   0m0.003s
```

5.3 Debugging, Profiling and Useful Linux Tools

Explain GDB - Valgrind - GProf etc.

Explain ldd - nm - strings etc.

5.4 Tools for Automatic Compilation

Explain Make - CMake - Bazel etc. here.

Chapter 6

Mathematical Models

Hope this day will come...

6.1 Development of Mathematical Models in Science and Engineering

6.2 Computational Linear Algebra

6.3 Numerical Solutions of ODEs and PDEs

Chapter 7

Scientific Libraries and APIs

Hope this day will come...

7.1 What is a library, API, framework etc.?

7.1.1 Numerical Recipes and GSL

7.1.2 MPI: Message Passing Interface

Message passing interface, widely known as `MPI`, is a programming interface for distributed memory workloads. One can basically send and receive bytes (messages) between processors in the `MPI` environment. Almost every major simulation packages takes advantage of `MPI` for running large scale simulations over a network of distributed processors.

7.1.3 CUDA

7.1.4 PETSc and Trillinos

7.2 Why is it important to use libraries?

7.3 How do libraries talk to executables?

Chapter 8

Scientific Applications

Hope this day will come...

8.1 Computational Fluid Dynamics

8.1.1 Solution of Turbulent Flows using OpenFOAM

8.1.2 Weather Modelling with WRF

8.2 Computational Molecular Dynamics

8.2.1 Gromacs and LAMMPS

8.3 Computational Quantum Chemistry

8.3.1 VASP and Quantum Espresso

8.4 Machine Learning

8.4.1 Next Day Forecasting

8.4.2 Deep Learning

Chapter 9

Cloud Computing and HPC

Even though cloud computing is not very new, one can argue that it has gained significant growth in the last 5 years. Currently most of the vendors are providing `IaaS` and `PaaS` solutions, and enterprises are transforming their business strategies to cloud native environments.

HPC and Scientific HPC are no exception to this situation.

9.1 HPC as a Service

Major cloud infrastucture providers offer HPC services as well.

9.1.1 Amazon Web Services

9.1.2 IBM Cloud

9.1.3 Google Cloud

9.1.4 Microsoft Azure

9.1.5 OpenStack

Unlike the previous vendors `OpenStack` is a Platform-as-a-Service (PaaS) offering. To simply put, one can transform their HPC infrastructure to a cloud service using `OpenStack`.

9.2 Virtulization and Containerization

Virtual machines and containers are similar technologies desgined for similar purposes. The basic idea behind their development is the abstraction of runtime environment from the host platform. For instance, one can create a virtual machine that uses `Linux` on top of a `Windows` machine and can run `Linux` application in that VM. This technology has been used very efficiently

by data centers in the last 2 decades. However, there is one disadvantage of VMs; they are a complete mimic of a host system and because of that they consume lots of resources.

Containerization solves this platform. Unlike VMs, containers are not a complete OS and therefore they are much less light-weight compared to VMS.

9.2.1 Docker Containers

9.2.2 Singularity

`Singularity` is one of the most exciting projects in the HPC community. It is a containerization application which does not required priviliged user permissions. Therefore, it is more secure than `Docker` and used widely by HPC centers.

9.3 Orchestration and Scheduling

Orchestration of the container deployments is a crucial task of the containerization management.

9.3.1 Kubernetes

`Kubernetes` became the de-facto standard of the container orchestrators.

9.4 Storage and Networking as a Software

9.5 Performance and Comparison to Traditional HPC

9.5.1 InfiniBand and RDMA

Chapter 10

Final Remarks

Hope this day will come...

Bibliography

[1] Brian Gough. *An Introduction to GCC*. Network Theory Limited, 2004.