Title

Maria Streater
West Texas A&M University

May 2017

Signature page

Abstract

In this paper the advantages of parallel computing... are presented using a parallel... . The objective of this work is to explore and to analyze.... The results demonstrate...

Contents

List of Tables

List of Figures

1 Introduction

1.1 Objective

1. Examine ...
2. Analyze and compare processing time on a various number of processors.

1.2 Chapter Outline

2 Background/Concepts

2.1 Linear Equations

2.1.1 Conservation Laws and Differential Equations

2.1.2 Hyperbolicity of Linear Systems

2.2 Advection Equations and Hyperbolic Systems

2.3 Lax Wendroff Method

2.4 Lax Friedrichs Method

2.5 Stability

2.6 Characteristic tracing and Interpolation

2.6.1 Characteristic Variables

3 Linear Acoustics on a Network Domain

3.1 Example of a network problem

3.2 Domain Decomposition Algorithm

4 Parallel Computing

4.1 Serial versus Parallel

Traditional serial programming runs a problem or job on a single computer having a single central processing unit (CPU). The CPU breaks the problem or jobs into a discrete series of instructions which are executed sequentially. Only one instruction may execute at any moment in time with this type of computing.

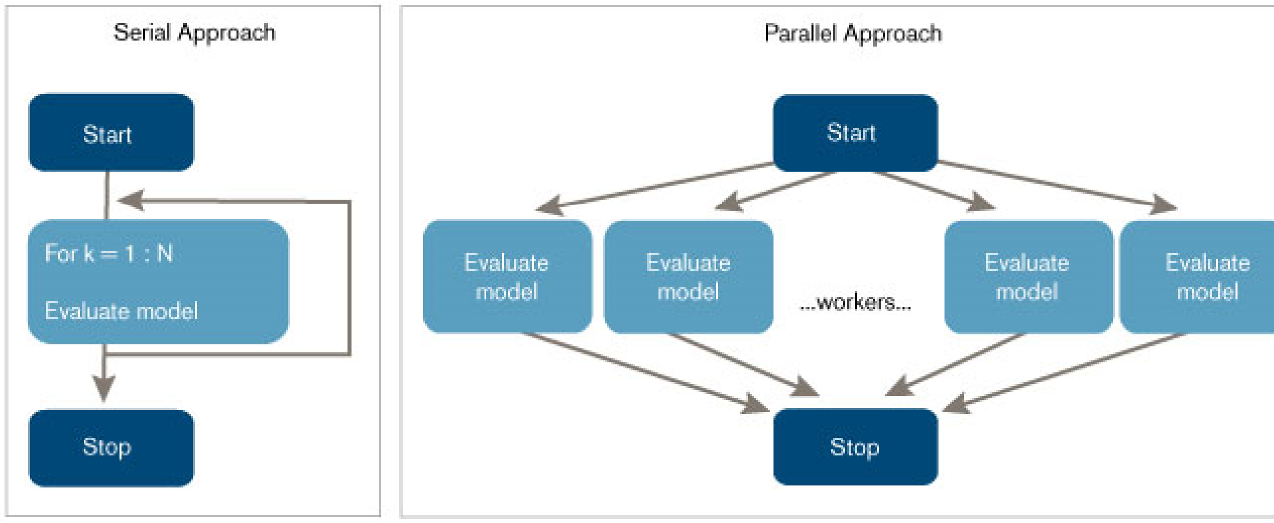

Figure 1: Serial versus Parallel Processing

From the above figure it is clear that in a serial approach, all processing happens one after the other. The next job is not triggered until the previous job has finished. Speed improvement on serial processes is limited by an inability to increase transfer speeds of operands to and from the data buffers and the functional units.[1] Where as the execution of a parallel program tasks are simultaneous. Work load is split up on multiple CPUs in order to obtain more rapid results. The idea is that the process of solving a problem can usually be divided into several smaller tasks, which may be carried out concurrently with some communication.

4.1.1 Disadvantages of Parallel Processing

Coordinating communication between the various processors can create bottlenecks and data integrity issues, which leads to slower execution or inaccuracies. Parallel programming is not simply an extension of serial programming; programs need to "think in parallel".[1] Thus parallel algorithms are more complex and require more work and time from a programming standpoint. If one is not careful race conditions can arise within a simulation. This occurs when the sequence or timing of processes or threads depend on a shared state, which can create data corruption when events "collide" or happen in an unintended order. Deadlock is another possible issue. This happens when two or more competing actions are waiting for the other to finish, resulting in neither ever completing.

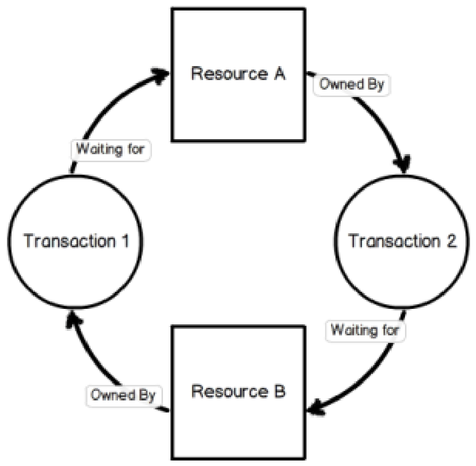

Figure 2: Deadlock

Sequential processing is a good method for processing data one step at a time and is ideal in situations when the CPU is performing a calculation that depends on the result of the previous calculation. Processing these kinds of calculations, those that cannot be parallelized, will slow down a parallel program.

4.1.2 Advantages of Parallel Processing

In theory using additional resources will shorten completion time thereby creating potential computational cost savings. Since parallel clusters can be built from inexpensive, commodity components they are ideal for gaining speedup without sacrificing cost. A single computing resource can only do one thing at a time, while multiple computing resources can be doing many things simultaneously. Parallel processing is much faster than sequential processing when it comes to doing repetitive calculations on substantial amounts of data. A parallel processor is capable of multitasking on a large scale, and can therefore simultaneously process several streams of data. Compared with serial systems, parallel systems permit more freedom of expression in problem analysis and programming.[1] Such freedom expands the type and depth of problems that can approached and analyzed.

4.2 Flynn's Taxonomy

Flynn's taxonomy dating from 1966 does not adequately reflect current architectural designs, but according to Modi [1] it is nevertheless a solid guideline and remains useful today. Flynn's taxonomy distinguishes multi-processor computer architectures by classification along the two independent dimensions of instruction stream and data stream. Each of these dimensions can have only one of two possible states: single or multiple.

Single Instruction, Single Data (SISD): A serial computer with the CPU acting on one instruction stream and using one data stream as input, both in any one clock cycle. This type has deterministic execution and is the oldest type of computer. Some examples are older generation mainframes, minicomputers, workstations and single processor/core PCs.

Single Instruction, Multiple Data (SIMD): A type of parallel computer where all processing units execute the same instruction at any given clock cycle. Each processing unit can operate on a different data element. This is best suited for specialized problems characterized by a high degree of regularity. This type has synchronous and deterministic execution with two varieties: processor arrays and vector pipelines. Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution.

Multiple Instruction, Single Data (MISD): A type of parallel computer in which a single data stream is fed into multiple processing units. Each processing unit operates on the data independently by separate instruction streams. Few, if any, actual examples of this type of parallel computer exist or existed.

Multiple Instruction, Multiple Data (MIMD): A type of parallel computer in which every processor may be executing a different instruction stream and may be working with a different data stream. The execution can be synchronous or asynchronous, deterministic or non-deterministic. Currently, the most common type of parallel computer and some examples include most current supercomputers, networked parallel computer clusters and multi-core PCs. The cluster for this study is MIMD.

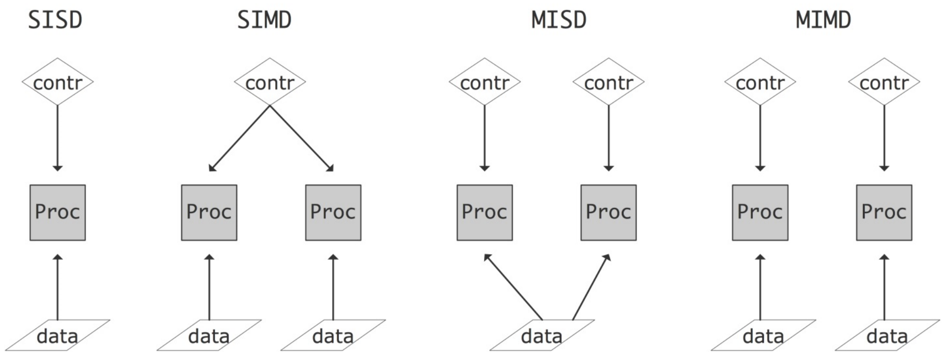

Figure 3: Flynn's Taxonomy

4.3 Limits

An obvious limit for small scale clustering, e.g. an at home scenario, can be limited by funding and the necessary space to set up and maintain the hardware.

For large scale clustering overall energy consumption to run and cool the CPUs present concerns about energy consumption and its' related costs.

4.3.1 Floating Point Representation and Error Propagation

Error can be introduced in several ways; mathematical modeling, blunders, uncertainty in physical data, machine errors, and mathematical truncation errors.[3] Since the model addressed in this paper is uncomplicated and the data is limited, the focus and main concern of introduced error is machine error and truncation. Any program, serial or parallel, is limited by the accuracy it can achieve and by how quickly it can obtain the desired results. Discussion of accuracy follows and discussion of algorithm issues is reserved for chapter 4. Because computers use a finite number of bits to represent real numbers, they are only able to represent a finite subset of the real numbers. Real numbers are converted to floating point numbers, in doing this they are either chopped or rounded.[3] This limitation presents two difficulties; first the represented numbers cannot be arbitrarily large or small and second there must be gaps between them. [5] Worse still is that it is not enough to represent real numbers; they must be used in computations, which can generate even more inaccuracies.[5]

In words, every operation of floating point arithmetic is exact up to a relative error of size at most $\epsilon_{machine}$.[5] Given by

$$\epsilon_{machine} = \begin{cases} 1/2\beta^{1-t} & \text{Chopped} \\ \beta^{1-t} & \text{Rounded} \end{cases}$$

where $\beta \geq 2$ and is the base or radix (typically 2) and $t \geq 1$ and is precision.[5]

4.3.2 Amdahl's Law

A theoretical index, speedup, has been used for measuring the performance of parallel algorithms.[1] Amdahl's law, also known as Amdahl's argument, is named after computer architect Gene Amdahl. It is used to find the upper limit of the expected improvement to an overall system when the system is only partially improved. Often, it is used in parallel computing to predict theoretical maximum observed speedup using multiple processors. Amdahl's law states that the possible performance improvement to be gained from using a faster mode of execution is limited by the fraction of the time the faster mode can be used.[1] Every parallel algorithm has a sequential component that will eventually limit speedup. According to Amdahl's Law, potential speedup is defined by the fraction of code that can be parallelized.

$$speedup = \frac{1}{1-p}$$

Where no speedup corresponds to $p = 0$ and if all of the code is parallelized $p = 1$. Theoretically speedup is then infinite. Introducing the number of processors performing the parallel fraction of work, the relationship can be modeled by

$$speedup_n = \frac{1}{1 - p + \frac{p}{n}}$$

where n is the number of processors. Amdahl's law is an algorithm that uses the law of diminishing returns that determines the overall speedup of the program. The limit of $speedup_n$ as n approaches infinity will simply be *speedup*, that is as n gets large the p/n term will tend to zero.

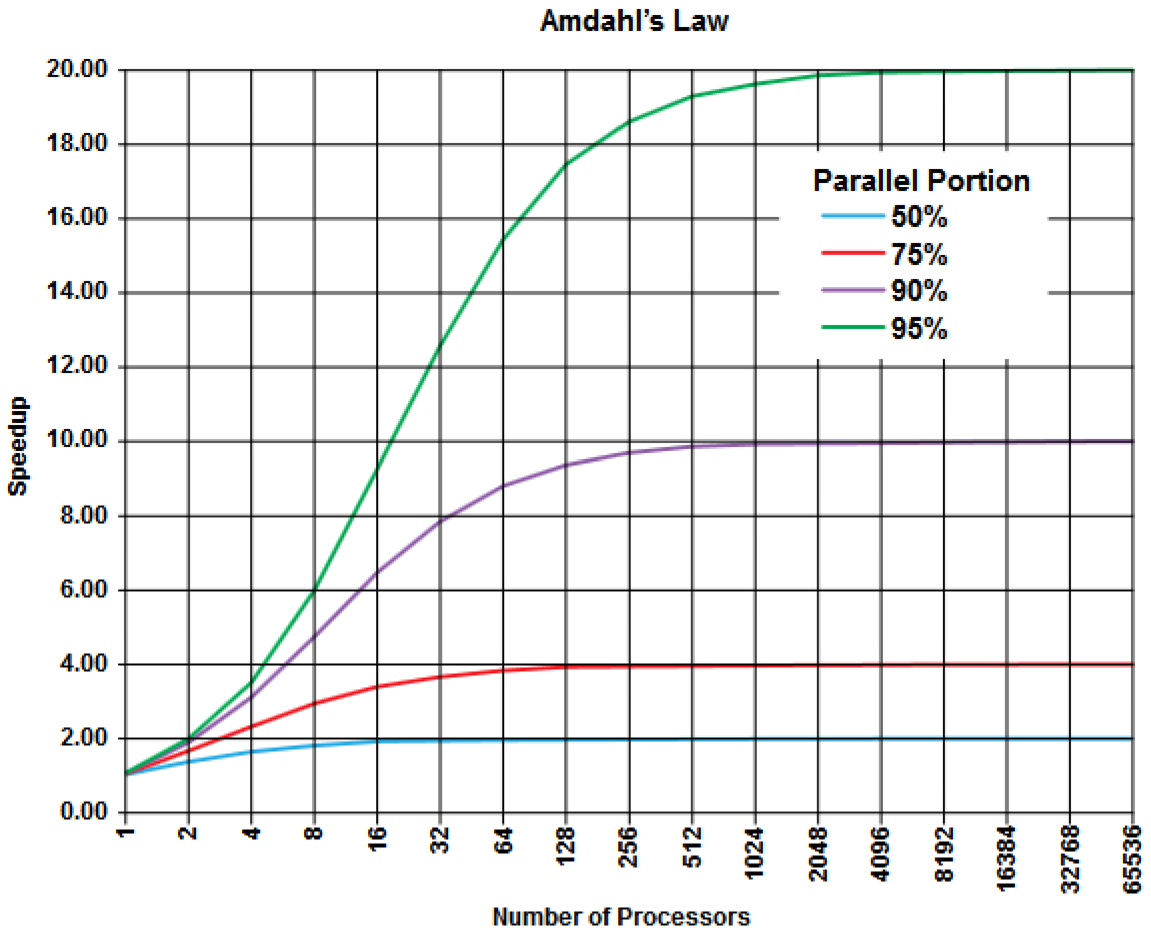

Figure 4: Amdahl's Law

It is obvious that there are limits to the scalability of parallelism. For example:

n	$p = .50$	$p = .90$	$p = .99$
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1,000	1.99	9.91	90.99
10,000	1.99	9.91	99.02
100,000	1.99	9.99	99.9

Table 1: Speedup

It follows that problems which increase the percentage of parallel time with size are more scalable than problems with a fixed percentage of parallel time.

4.4 Costs

In general, parallel programs are more complex than corresponding serial programs, perhaps by an order of magnitude.[1] Not only are there multiple instruction streams executing concurrently but there is also data flowing between them. The costs of complexity are measured in programmer time for every aspect of the development cycle in the form of design, coding, debugging, tuning, and maintenance. The primary focus of parallel programming is to decrease execution wall clock time. In order to accomplish this, more CPU time is required. For example, a parallel code that runs in 5 minutes on 5 processors might actually use 25 minutes of CPU time. The amount of memory required for parallel codes can be greater than what is required for serial codes. This can be caused by the need to replicate data and for overheads associated with parallel support libraries or subsystems. For short running parallel programs, there can actually be a decrease in performance compared to a similar serial program. The overhead costs associated with setting up the parallel environment, task creation, communications and task termination can comprise a significant portion of the total execution time for short runs.[9]

5 The Cluster

5.1 Beowulf Cluster

The Beowulf computer cluster is comprised of nearly identical, commodity-grade computer resulting in a parallel computing cluster made from inexpensive personal computer hardware. Construction and set up of the cluster relied heavily on [7], a rough guide. Each processor is autonomous and may operate as a standalone computer.[1] GNU/Linux, an open source operating system, was installed on all 5 nodes of the cluster using [8]. The distribution used was Debian 8.1 64-bit code named Jessie. A net install was used to choose which software would be downloaded and installed. A minimal install with no GUI was used for the nodes.

The nodes were all connected through LAN with a five port switch. The LAN allowed the configuration of SSH [12] and NFS [11]. To setup the LAN, on each node a static IP and hostname was assigned. SSH or Secure Shell is a network protocol for initializing text-based shell sessions on remote machines. This makes the operation of the cluster very easy. All the nodes can be controlled through the head node using SSH. Also Open MPI requires that the head node can SSH into each machine with no password [10]. To setup SSH, using [12], some cryptographic keys were created on the head node, and then copied to each node. With these keys, the master can SSH to any node with no password necessary. Network File System (NFS) allows the nodes to access a directory on the master. This seems to be the easiest way to transfer files between nodes. To setup NFS, using [11], a directory on the head node was created, changes were made its security to allow access, and then the directory was exported. For the nodes, a place to mount the shared directory was created and changed to mount the directory automatically during the boot process.

5.2 CPU Specifications

Model	CPU MHz	Cache Size	CPU Cores
AMD Athlon IIx4 630 Processor	800	512 KB	4
Intel Core 2 Duo CPU E7500 @ 2.93GHz	2926.2333	3072 KB	2
Intel Core 2 Duo CPU E7500 @ 2.93GHz	2926.01	3072 KB	2
Intel Core 2 Duo CPU E8400 @ 3GHz	2000	6144 KB	2
Intel Core 2 Duo CPU E7500 @ 2.93GHz	1600	3072 KB	2

Table 2: Cluster CPU Specifications

5.3 Open MPI

A message passing interface (MPI) is a communications protocol used for programming parallel computers. Open MPI is an open source messaging passing interface implementation that is developed in a true open source fashion and maintained by a consortium of academic, research, and industry partners. Open MPI is able to pool resources, technologies, and expertise from across the computing community and thus provides a high performance message passing library, as such it is used in many TOP500 supercomputers. Open MPI provides a wrapper for compilers and some simple commands to run a process on the cluster. All documentation and download links for Open MPI were used from [9] and [10] whereas [13] was used for tutorials and additional help topics. Open MPI is the backbone of the cluster.

6 Memory Structure

The cluster addressed in this paper uses the distributed memory model and as such other memory structures are not discussed though they exist. This model demonstrates the following characteristics according to [13]

- A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines.
- Tasks exchange data through communications by sending and receiving messages.
- Data transfer usually requires cooperative operations to be performed by each process.

6.1 Distributed Memory Model

Distributed memory systems, like shared memory systems, vary widely, but do share a common characteristic. Distributed memory systems require a communication network which connects inter-processor memory.

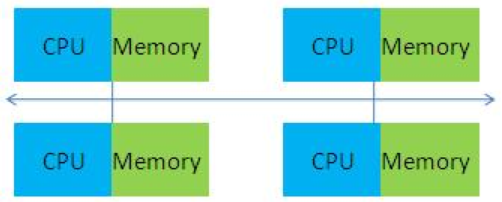

Figure 5: Distributed Memory Structure

In a distributed memory structure processors have their own local memory and operate independently. Processor memory addresses do not map to one another, so there is not global address space across all processors. The concept of cache coherency does not apply since changes made to each processors local memory have no effect on the memory of other processors. From a programming perspective, message passing implementations usually comprise a library of subroutines. Calls to these subroutines are embedded in source code. Thus the programmer is responsible for determining all parallelism. Synchronization and data access among processors is the left to programmer to explicitly define how and when data is communicated. The network ”fabric” used for data transfer is simply an Ethernet cable.

6.1.1 Disadvantages

Programmer responsibility for many of the details associated with data communication between processors can be time consuming and ineffective. It can be difficult to map existing data structures, based on global memory, to this memory organization. Data residing on a remote node may take longer to access than node local data and can result in non-uniform memory access time. Presentation of this type of an issue did not occur due to the small scale of this cluster.

6.1.2 Advantages

This memory structure is cost effective since off the shelf commodity grade processors and networking can be used. An increase in the number of processors creates a directly proportional increase the size of the memory, hence memory is scalable. Each processor can efficiently access its own memory without interference and the overhead incurred with trying to maintain global cache coherency.

7 Programming Algorithm

7.0.3 Bottlenecks and Hot-spots

To identify bottlenecks in the program it is best to look for inhibitors to parallelism. One common class of inhibitor is data dependence; this can be seen in the sequential nature of the Fibonacci sequence.

This will be expanded to include file sharing access issues as well as junction conditional calculation time. Considering the structure of the cluster and the program, each processor can execute tasks without dependence, so bottlenecks of this type were not an issue for this simulation. Running tasks on the master rather than reserving it for communication and compilation of data could have created a bottleneck. However, due to limited communication required to perform tasks, the bottleneck, if any was negligible for the intents and purposes of this study. To identify the program's hot-spots it is important to know where most of the real work is being done. The majority of programs usually accomplish most of their work in only a few places. The work horse of this approximation is .... Other sections of the program that account for little CPU usage were ignored in favor of parallelizing the actual simulation.

7.0.4 Load Balancing

Load balancing distributes workloads across the computer cluster and aims to optimize resource use, maximize throughput, minimize response time, and avoid overload of any single resource.[9] Load balancing refers to the practice of distributing approximately equal amounts of work among tasks so that all tasks are kept busy all of the time to minimize task idle time. Load balancing is important to parallel programs for performance reasons. To achieve load balance equally partitioned work was assigned to each task. Each CPU was assigned the same job, that is each CPU ran the same simulation on individual domains. Though the master has a CPU with 4 cores while the nodes have CPUs with 2 cores, Open MPI handles the load balancing with the assumption that if the load among all nodes is balanced, then the overall execution time of the application is minimized. Dynamic load balancing algorithms make changes to the work distribution at run-time. This technique takes into account over-loaded or under loaded nodes. It uses current load information when making load balancing distribution decisions.

7.0.5 Embarrassingly Parallel

An embarrassingly parallel problem is one where little or no effort is required to separate the problem into a number of parallel tasks.[1] Embarrassingly parallel problems tend to require little or no communication of results between tasks and thus do not suffer from parallel slowdown. The simulation considered in this paper is embarrassingly parallel since there does not exists dependency or communication between the parallel tasks of each simulation.

8 Implementation

8.1 C++

C++[15] is the chosen programming language of this project for a variety of reasons. It is a widely used and accepted language; as such vast amount of documentation and plenty of robust libraries are available, all at no cost. Implementations are available on many platforms, hence it is portable. As a compiled language, C++ is more efficient than MATLAB or Python, which are interpreted languages. MATLAB was used for data manipulations and graphics however.

8.2 Parallelizing the Simulation

Due to the embarrassingly parallel nature of the simulation, parallelizing it was straight forward and uncomplicated. Unlike more complex algorithms in which sections of a total tasks and delegated to each CPU, each CPU was sent the exact same task. The task to be completed varied in .... The master then complied the information returned by each of the nodes.

9 Convergence Analysis

10 Results and Analysis

11 Conclusion

11.1 Future Research

A C++ code

References

[1] Jagdish J. Modi *Parallel Algorithms and Matrix Computations* Oxford University Press (1988) 5:7-9:11:36:55

[2] Stanley J. Farlow *Partial Differential Equations for Scientists and Engineers* Dover Publications (1993) 340

[3] Kendall E. Atkinson *An Introduction to Numerical Analysis Ed 2* (1989) 13:15:17-20:34

[4] John M. Zelle *Python Programming an Introduction to Computer Science Ed 2* Franklin, Beedle, and Associates Inc. (2010) 295

[5] Lloyd N. Trefethen and David Bau III *Numerical Linear Algebra* SIAM (1997) 97:99

[7] Kurt Swendson *The Beowulf HOWTO* (2004) available at http://en.tldp.org/HOWTO/Beowulf-HOWTO/intro.html

[8] Software in the Public Interest, Inc *About Debian* (2015) available at https://www.debian.org/intro/about

[9] *Open MPI: Open Source High Performance Computing* (2015) available at https://www.open-mpi.org/

[10] *Installing Open MPI* (2013) available at http://particlephysicsandcode.com/2012/11/04/installing-open-mpi-1-6-3-ubuntu-12-04-fedora/

[11] D Mizer *HOWTO: NFS Server/Client* (2009) available at http://ubuntuforums.org/showthread.php?t=249889

[12] Steve Kemp *Password-less logins with OpenSSH* (2005) available at https://www.debian-administration.org/article/152/Password_less_logins_with_OpenSSH

[13] Blaise Barney *Message Passing Interface, MPI* (2015) available at https://computing.llnl.gov/tutorials/mpi/

[15] Alex *Learn Cpp.com Tutorials to help you master C++ and object oriented programming* (2015) available at http://www.learncpp.com/