Exploiting multiple levels of parallelism of Convergent Cross Mapping

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Bo Pu

©Bo Pu, September 2019. All rights reserved.

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Or
Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building
110 Science Place
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

Identifying causal relationships between variables remains an essential problem across various scientific fields. Such identification is particularly important but challenging in complex systems, such as those involving human behaviour, sociotechnical contexts, and natural ecosystems. By exploiting state space reconstruction via lagged embeddings of time series, convergent cross mapping (CCM) serves as an important method for addressing this problem. While powerful, CCM is computationally costly; moreover, CCM results are highly sensitive to several parameter values. Current best practice involves performing a systematic search on a range of parameters, but results in high computational burden, which mainly raises barriers to practical use. In light of both such challenges and the growing size of commonly encountered datasets from complex systems, inferring the causality with confidence using CCM in a reasonable time becomes a biggest challenge.

In this thesis, I investigate the performance associated with a variety of parallel techniques (CUDA, Thrust, OpenMP, MPI and Spark, etc.,) to accelerate convergent cross mapping. The performance of each method was collected and compared across multiple experiments to further evaluate potential bottlenecks. Moreover, the work deployed and tested combinations of these techniques to more thoroughly exploit available computation resources. The results obtained from these experiments indicate that GPUs can only accelerate the CCM algorithm under certain circumstances and requirements. Otherwise, the overhead of data transfer and communication can become the limiting bottleneck. On the other hand, in cluster computing, the MPI/OpenMP framework outperforms the Spark framework by more than one order of magnitude in terms of processing speed and provides more consistent performance for distributed computing. This also reflects the large size of the output from the CCM algorithm. However, Spark shows better cluster infrastructure management, ease of software engineering, and more ready handling of other aspects, such as node failure and data replication. Furthermore, combinations of GPU and cluster frameworks are deployed and compared in GPU/CPU clusters. An apparent speedup can be achieved in the Spark framework, while extra time cost is incurred in the MPI/OpenMP framework. The underlying reason reflects the fact that the code complexity imposed by GPU utilization cannot be readily offset in the MPI/OpenMP framework. Overall, the experimental results on parallelized solutions have demonstrated a capacity for over an order of magnitude performance improvement when compared with the widely used current library rEDM. Such economies in computation time can speed learning and robust identification of causal drivers in complex systems.

I conclude that these parallel techniques can achieve significant improvements. However, the performance gain varies among different techniques or frameworks. Although the use of GPUs can accelerate the application, there still exists constraints required to be taken into consideration, especially with regards to the input data scale. Without proper usage, GPUs use can even slow down the whole execution time. Convergent cross mapping can achieve a maximum speedup by adopting the MPI/OpenMP framework, as it is suitable to computation-intensive algorithms. By contrast, the Spark framework with integrated GPU accelerators still offers low execution cost comparing to the pure Spark version, which mainly fits in data-intensive problems.

Acknowledgements

To begin with, I would like to express my genuine gratitude to my supervisor, professor Nathaniel D. Osgood, for the continuous support of everything during my master study here with his patience, motivation, and immense experience. I could not have imagined having a better advisor. During my study period here, I have learned a lot from my supervisor not only on the academic level but also in his high ethical standards. His enthusiasm, generosity, care for others and passion for research are influenced me deeply, which will always remind me to become a better person in my rest life. Moreover, I take it as one of the luckiest things in my life to study and work with my supervisor. Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Dwight Makaroff, Prof. Derek Eager, and Prof. Francis Bui, for their insightful comments and thoughtful suggestions. And also for the questions which incented me to widen my research from various perspectives.

Also, I would like to acknowledge my fellow lab mates in for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last two years. Mainly, I would like to thank Weicheng Qian, Yang Qin and Lujie Duan for the advices on parallel implementations; Weicheng Qian offers tremendous help and support in the implementation details and theory discussion based on his knowledge. Lujie Duan provides a lot of insights when we discuss the cluster configuration and system-level problems. I would also like to thank all the professors and stuff in the computer science department, especially Christine, Sophie and Gwen for their guidance during my master study period.

Finally, I want to thank my family for the support when I study here. I haven't have the chance to go back China when I came here two years ago. My mother Dirong Chen and my father Shaoming Pu take care of everything so I can pursure my master degree without any concern. Your love, support and encouragement are always my impetus to move forward.

Contents

Permission to Use i

Abstract ii

Acknowledgements iii

Contents iv

List of Tables vi

List of Figures vii

List of Abbreviations ix

1 Introduction 1
- 1.1 Motivation 1
  - 1.1.1 Detecting Causality using Convergent Cross Mapping 1
  - 1.1.2 Dynamical System Theory 2
  - 1.1.3 Takens' Embedding Theorem 3
  - 1.1.4 Convergent Cross Mapping 4
- 1.2 Research Goals 6
- 1.3 Thesis Organization 7

2 Background 8
- 2.1 Literature Review 8
  - 2.1.1 Convergent Cross Mapping Basics 8
  - 2.1.2 Past work in CCM Performance Improvement 11
- 2.2 Introduction of Parallel Methodologies 12
  - 2.2.1 Overview and Categories 12
  - 2.2.2 GPU architecture and CUDA programming 13
  - 2.2.3 Multi-core machine architecture and multithreaded programming 14
  - 2.2.4 Cluster architecture and distributed programming 15
- 2.3 CCM Algorithm Analysis 16
  - 2.3.1 Public Library of CCM: rEDM 16
  - 2.3.2 Parallel Design of CCM Algorithm and Comparison 17
- 2.4 Summary 20

3 Exploiting GPU Acceleration of Convergent Cross Mapping 21
- 3.1 Introduction 21
- 3.2 Methodology 22
  - 3.2.1 Brute Force kNN Search 22
  - 3.2.2 Pearson's Correlation Coefficient 25
- 3.3 CUDA Implementation 25
  - 3.3.1 Pairwise Distance Calculation Kernel 26
  - 3.3.2 Radix Sort Operation Kernel 27
  - 3.3.3 Pearson's Correlation Coefficient Kernel 28
- 3.4 Experiments 28
  - 3.4.1 Setup 28
  - 3.4.2 Results 29
- 3.5 Conclusion 32

4 Exploiting Cluster Parallelism of Convergent Cross Mapping 34
4.1 Introduction 34
4.2 Parallelizing CCM using Hybrid MPI/OpenMP Framework 35
4.3 Parallelizing CCM Using Apache Spark Framework 36
4.4 Experiments 39
4.4.1 MPI/OpenMP 40
4.4.2 Apache Spark 41
4.4.3 Discussion 46
4.5 Conclusion 47

5 Exploiting Cluster Parallelism with GPU Acceleration of Convergent Cross Mapping 48
5.1 Introduction 48
5.2 Integrating GPU accelerators into Spark 49
5.2.1 Methodology 49
5.2.2 Performance issues 51
5.2.3 Integrating GPU accelerators 52
5.3 Integrating GPU accelerators into MPI 53
5.3.1 Methodology 53
5.3.2 Integrating GPU accelerators 55
5.4 Experiment 56
5.4.1 Setup 56
5.4.2 Results 57
5.5 Conclusion 59

6 Conclusion 61
6.1 Discussion and Conclusion 61
6.1.1 Summary 63
6.2 Future Work 63
6.3 Contributions 65
6.4 Publications Related to the Thesis 66

References 67

Appendix A Documentation 71
A.1 Codebase 71
A.2 Sample Configuration File 71

List of Tables

2.1 Notation 17

4.1 Baseline setting 40
4.2 Implementation Levels 41
4.3 Elasticity Analysis 42
4.4 Cluster Configurations 45

5.1 Hardware configuration of the hybrid cluster 56
5.2 CCM baseline parameters setting 57
5.3 Implementation Levels 57

List of Figures

1.1 This diagram demonstrates the methodology of attractor reconstruction via delay embedding. The true attractor is projected into a time series by some measurement functions, from which an image of the attractor can be formed by delay reconstruction, up to some diffeomorphism. 3

2.1 GPU device architecture overview. 14
2.2 Shared memory system architecture overview. 14
2.3 Distributed memory system architecture overview. 15

3.1 Image adapted from [53] illustrates the kNN search problem given the embedding dimension parameter $E = 3$, which means that k is 4, in the reconstructed state space. The blue points refer to the points in RP set while the red points are in QP set. Reflecting the fact that $k = 4$, the circle gives the distance between the query point and the fourth closest reference point. . 22
3.2 An example of a pass in the radix sorting operation on one chunk of length $n = 20$ within 4-bit digits, which means that the maximum value is less than $2^4 = 16$. 24
3.3 An illustration of applying the radix sort to sort $k = 8$-bit integers by 2 passes of three steps aforementioned on $d = 4$-bit integers from the least significant bits to the next higher significant bits, and so on. 24
3.4 The parallel algorithm for pairwise distance calculation. Each block computes one sub-matrix of **OUT**, and the threads work on one pair of aligned sub-matrices of **IN**, as is illustrated as the figure. 26
3.5 The parallel prefix scan algorithm. The interval is changed from 1 to interval $log_2 n$. 27
3.6 The parallel algorithm for R samples, $(E + 1)$-dimension vectors, of Pearson's correlation coefficient calculation. Each thread computes one row of **OUT**, which takes one pair of the same row of **X** and **Y**. 28
3.7 The performance comparison of the pairwise Euclidean distances under different dimension E and the input sequence length T (Notably, then input matrix is $T \times E$). 29
3.8 The performance comparison of the radix sort operation in kNN search when applying the distances calculated from previous step. Sort can be performed either on local library size L or global time series T. 30
3.9 The performance comparison of Pearson's correlation coefficient computation for two $R \times [E+1]$ matrices given a certain library size L. 31
3.10 Among three kernel functions, the pairwise distance calculation kernel takes much longer time, which can be utilized as the benchmark to compare the performance of different GPUs. . . . 32

4.1 Simplified typical hybrid MPI/OpenMP scheme applied on CCM. 35
4.2 An example of the pipeline running distributed on Spark. 37
4.3 A diagram of CCM RDD transformation which takes multiple realizations as input and outputs prediction skills. 37
4.4 An illustration of the dependencies of two pipelines. After the distance indexing table is constructed in parallel, Spark will broadcast it to all nodes. In the next pipeline, the executors can look up in the table and fetch the $E + 1$ nearest neighbours quickly. 38
4.5 A diagram of Spark executing asynchronous pipelines. 39
4.6 The 3-node cluster contains 3 machines, which has been labelled from 0 to 2. Node 0 is the master node in the cluster and the machine which runs rEDM for comparison. 40
4.7 Yarn Mode utilizes all worker nodes in the cluster, while Local Mode only runs experiments on the master node. Yarn Mode significantly diminishes the average computation time of the parallel version of CCM with the help of worker nodes. 42
4.8 The parallel version uses all of the optimization methods with five 4-core workers in the cluster, while the single-threaded version is only executed on the master node without any parallel optimization. 43

4.9 The single-threaded version without using Spark consumes little time to finish when library size L is small. However, with increasing L, the time to search for nearest neighbors in a larger shadow manifold and the time to complete grows superlinearly. 44
4.10 The average running time can be reduced when more computational resources are available in the cluster. Under the same total computational resources, the computation time is larger with more inferior worker nodes given the rising time cost of networking I/O. 45

5.1 An overview of the GPU-enabled cluster. 49
5.2 A comparison of pure Apache Spark and Apache Spark with external program implementation. 50
5.3 CPU utilization plot in the cluster for three workers. 52
5.4 Overall methodology of integrating GPU with Spark framework. 52
5.5 Bottleneck elimination and performance comparison after introducing GPU into external application. 53
5.6 The comparison between pure MPI with CUDA and CUDA-Aware MPI architecture. Obviously, CUDA-Aware MPI provides a layer to unify the memory space between host and device. 54
5.7 Simplified hybrid CUDA-Aware MPI/OpenMP scheme applied on CCM. 55
5.8 This experiment is conducted in the cluster setup described in the Table 4.1. The computation part and disk I/O part are separately measured to compare with respect to differences in each. 58

6.1 The best speedup of Parallel CCM achieved in this thesis, when compared to rEDM. 62

List of Abbreviations

CCM	Convergent cross mapping
RDD	Resilient distributed datasets in Spark
MPI	Message passing interface
GPU	Graphics processing units
FPGA	Field-programmable gate array
kNN	k nearest neighbors framework
MIC	Intel Many Integrated Core that combines many cores onto a single chip
UVA	Unified Virtual Addressing in CUDA-aware MPI

Chapter 1

Introduction

1.1 Motivation

1.1.1 Detecting Causality using Convergent Cross Mapping

The detection of causality in complex systems has been studied for many years, in light of its importance for scientific study, design of models, decision-making, and other needs. Complex systems are referred to the systems whose behaviour is difficult to understand due to the fact that system behaviour is not directly understandable through understanding each component of the system in isolation. For such systems – where the "whole is greater than the sum of the parts", behaviour exhibits non-linear relationships on system states and depends heavily on hidden dependencies or unknown interactions caused by the inside (parts of the systems) or outside factors (environment) of the systems. However, systems involving various interacting variables and potential states are fundamental to the natural and social sciences.

The authentic causal understanding of such systems through their behaviours plays an essential role given the need to making effective decisions, especially including policy and financial domains [58]. In traditional analysis – inspired by linear systems – identification of correlation and covariation between variables has been widely applied in hopes of identifying causality in stationary time series. By contrast, complex nonlinear system variables can be positively linked at specific time windows, while at other time windows, such variables can appear unrelated or even negatively linked. Such conflicting evidence will often arise when traditional metrics like correlation and covariation are applied. Analysis using correlation or covariation becomes more difficult to justify with increasing recognition that nonlinear dynamics are ubiquitous in challenging decision-making and policy contexts. Although linear dynamic analysis is a well-developed technique with elegant theoretical underpinnings, most real-world systems manifest themselves in a much broader spectrum of possible complex behaviours. Possibilities include intermittency, discontinuous motion, and history dependence, including sensitivity to initial states. All of these seeming chaotic behaviours make complex systems unpredictable and challenging to understand. Furthermore, the methods, which are designed based on the linear systems, can lead to incorrect or even contradictory evidence regarding causality in nonlinear systems. Increasing recognition of the importance of such behaviour calls for a better criterion to evaluate causal connections in complex systems.

Different approaches have been pursued to overcome the difficulty associated with causal inference in complex systems. Firstly, controlled experimentation or investigation of underlying mechanisms have been applied to investigate causal relations among variables. The first of these requires a substantial investment of time and financial resources. For contexts in which we seek to understand human behaviour, the pursuit of controlled experiments frequently poses ethical concerns, due to the risk of imposing harm on subjects. The limits of such controlled studies – such as Randomized Controlled Trials (RCTs) – are particularly notable in the context of complex systems, which commonly exhibit reciprocal feedbacks, delays and non-linearities [38].

While research seeking to elucidate underlying mechanisms and causal pathways (e.g., biological pathways underlying certain types of cancer, or diseases such as Type 2 Diabetes) are ubiquitous in science, the time required to secure great progress in such studies is commonly measured in decades.

The limitations of such traditional routes to identifying causal linkages have driven investigation into alternatives. [26] sought to take advantage of Granger causality (GC) theory as a mechanism for testing nonlinear causality in time series. But this approach can be problematic, especially in weak to moderate coupling systems [53]. The method exhibits particular limitation on account of its assumption of separability. While separability [36] is characteristic of completely stochastic systems, in complex systems with the capability of displaying broad coupling, separability typically does not hold. Separability represents the perspective that the systems can be reconstructed piece-by-piece rather than as a whole, which flies in the face of the emergent behaviour routinely seen in complex systems. The assumptions of the GC method apply only under the condition that completely stochastic is the nature of the real world. However, most of the systems in the real world contain strong deterministic governing components, which behave with patterns. As such, dynamic systems theory can be fruitfully introduced to analyze principles underlying the dynamics of complex, non-linear systems, and to reason about their long-term qualitative behaviour.

1.1.2 Dynamical System Theory

Dynamical system theory [2] is a scientific area, generally employing mathematical methods such as differential or difference equations, to describe and understand the behaviour of complex systems. At any timepoint, a dynamical system has a state characterizable by a vector, which can be alternatively viewed as a point with finite dimensions in accompanying state space. In dynamic system theory, such states can evolve under certain deterministic or stochastic rules that provide guidelines as to how current state evolves into future state. Mathematical characterization of state and its time evolution (behaviour) form the foundation of this theory.

To the extent that the system is deterministic and dynamics are not entirely random, there will be an underlying manifold controlling the dynamics. As such, this theory introduces the concept of an attractor. Causally linked time-series variables share a common attractor manifold M, which means that the coordinate for one state space coordinate will typically be closely covarying with another, and the information associated

with one variable can be recovered from variables that it drives within the dynamics of the common attractor. Hence, reconstruction of nonlinear state space M' can serve as an important possible tool when seeking to infer causality in a dynamic system.

1.1.3 Takens' Embedding Theorem

In 1981, Takens' Theorem [55] demonstrated and proved the manner in which lagged coordinates formed by a time series could be employed as substitute variables to reconstruct the shadow manifold of the underlying dynamic system. Assuming M is a compressed manifold of a m-dimension state space, so a dynamical system actually can be thought as a diffeomorphism ϕ determining the trajectories on the compressed manifold M under the discrete time intervals in Takens' Theorem [13]. The reference to the diffeomorphism refers to the invertible function that maps two smooth manifolds whilst preserving similarity on local topology. In mathematics, given ϕ and M, an observation function $y : M \to \mathbb{R}$ can be applied to construct an embedding M' of M in $2m+1$ dimensions. The complete transformation form in Takens' Theorem is $\Phi_{(\phi,y)} : M \to \mathbb{R}^{2m+1}$, where $\Phi_{(\phi,y)}(X) =< y(X), y(\phi(X)), y(\phi^2(X)), ..., y(\phi^{2m}(X)) >$ [13]. Here, the components on the right side of the equation represent the time-lagged variables of the original dynamics on M. As we can observed from the equation, such mappings (in the observation function) involve a single time series, which only represents a subset of possible mappings when considering the number of time series and lagged values. The reconstructed embedding may not preserve the global topology information of original manifold M. But still, the every local neighbourhood in the topology of the original manifold can be preserved, which remains a useful conceptual stepping stone for information recovery by searching nearest neighbours of reconstructed embedding manifold M'.

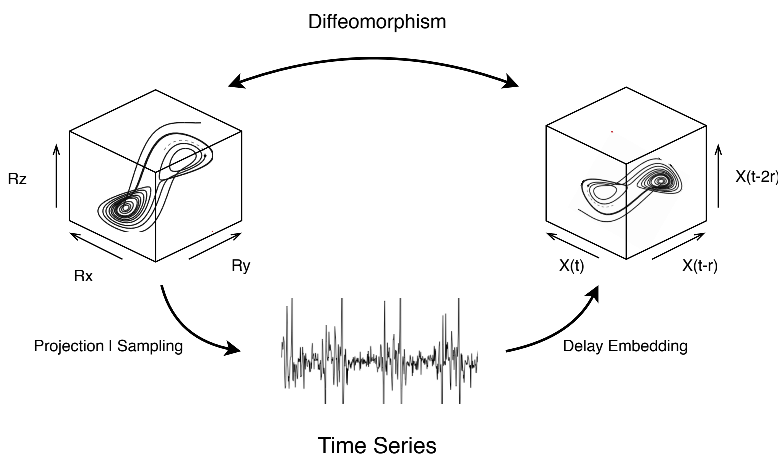

Figure 1.1: This diagram demonstrates the methodology of attractor reconstruction via delay embedding. The true attractor is projected into a time series by some measurement functions, from which an image of the attractor can be formed by delay reconstruction, up to some diffeomorphism.

As shown in Figure 1.1, the real attractor M, usually characterized as a surface or manifold, is defined by the trajectories in three-dimensional space. For simplicity, a manifold can be considered as a generalized,

E-dimensional surface embedded in some higher dimensional space, where the dimension of the manifold may be irregular and fractal. In equation (1.1), the time series X (where T is maximum index) can be viewed as sequential projections of the motion or samples with a certain time interval on the real attractor M in the dynamic system, as follows.

$$X = < x_1, x_2, ..., x_T > \tag{1.1}$$

Takens' Theorem above states that mathematically valid and equivalent reconstructions M' of the attractor can be created using lags of just a single time series (a sequential projection) by substituting those lagged values for unknown or unobserved variables, as each time series is a function of the system state and in general is driven by – and thus contains information regarding – several state variables. In equation (1.2), $E = 2m + 1$ is the reconstruction embedding dimension and τ ($\tau > 0$) is the general delay as lagged value.

$$\overrightarrow{x_t} = < x_t, x_{t-\tau}, ..., x_{t-(E-1)\tau} > \tag{1.2}$$

1.1.4 Convergent Cross Mapping

Convergent cross mapping [53], proposed by Dr. George Sugihara in 2012, is a statistical test based on Takens' Theorem that can be used to detect and help quantify the relative strength of unidirectional and bidirectional causal relationships between two variables X and Y drawn from the same coupled complex system. Essentially, to investigate whether Y is causally influencing X, we determine whether the shadow manifold M_X encodes information about Y. To do so, we take advantage of the fact that if Y is governing/driving/influencing X, the value of Y is part of the state of the system underlying X. By the definition of state space, information concerning the state (and, thus, value) of Y would need to be captured (encoded) by the location in the state space of the system driving X. By contrast, if Y is not driving X, Y does not form an element of the state space of X, and the location in state space of the system driving X should not encode any more information about the value of Y than mere statistical dependence on another aspect of state. Then we assess whether (and to what degree) it is possible to estimate (infer) the value of Y at a given time t using information from the observation of Y associated with the closest points within shadow manifold M_X reconstructed from the delay embedding X. If the ability to use X to recover Y rises as one considers shadow manifolds of additional density – and reconstructed from longer time series – it suggests that Y is driving X. By contrast, a merely statistical dependence of Y on an element of the system state underlying X or X itself would not lead to a notable rise in the ability to predict Y with shadow manifold density.

More specifically, the algorithm uses lag embedding of time series X to reconstruct the shadow manifold M_X [31]. To rebuild a state space of dimensionality E – including unobserved (latent) variables within the system – from a time series X, we can substitute each (non-boundary censored) point of that time series by an E-dimensional vector whose elements are successive lagged values drawn from that time series separated

by time index τ, per equation (1.2). Within the CCM algorithm, in order to assess if variable X is causally governed by variable Y, we attempt to predict the value of Y on the basis of the state space reconstructed from X (see below); for statistical reliability, this must be performed over a large number R of realizations. To assess causality, we examine whether these results converge as we consider a growing number L of data points – and thus a greater manifold density – within X within our reconstruction (see below).

Overall, the results of CCM are sensitive to the parameters below (the overall notation of CCM is listed in Appendix A):

E**:** This parameter is the estimated embedding dimension of the dynamic system. For simplex projection, E will typically range from 1 to 10. Eckmann [16] demonstrated the fundamental limitations for estimating dimensions of dynamic systems in 1992. The difficulty of accurately estimating E requires researchers testing and running CCM for different possible values of E.

τ**:** This fundamental parameter represents the embedding delay used in shadow manifold reconstruction. If appropriate lags are used, the reconstruction preserves the essential mathematical properties of the original system: Reconstructed states will map one-to-one to actual system states, and nearby points in the reconstruction will correspond to similar system states. In the presence of high autocorrelation between successive measured values in X, smaller values of τ will lead to successive coordinates in the embedding vector holding highly similar values; by contrast, larger values of this parameter will yield embedding vector elements less subject to autocorrelation. However, the estimation of the most favorable embedding delay is often unclear, and current practice explores a variety of possible values.

L**:** Another parameter central to the definition of CCM, L counts the size of the subsequence of the embedded library extracted from the time series for the purposes of state space reconstruction. In general, a prediction skill that initially increases along with rising L and then converges with a positive plateau value implies a causal relationship. With more data, the trajectories defining the attractor fill in, resulting in closer nearest neighbours and declining estimation error (corresponding to a higher correlation coefficient) as L increases.

R**:** This parameter – whose notation is less standardized in the literature – refers to the count of random subsamples (realizations) taken of a given size L. To enhance statistical reliability, the value of R is commonly set to 250 or larger. By determining the statistical confidence associated with the tests, this parameter is an important feature of any empirical study and statistical measurement. In the algorithm, multiple random realizations for a given library size L can reduce bias, produce more accurate estimates, and better reveal trends with growing L. Alternatively with researchers using larger sample sizes, higher values of this parameter impose elevated levels of computational burden in the form of longer running times and elevated space consumption for CCM output.

Running CCM across a wide range of different parameter settings is necessary to obtain a reliable causal reference (hyperparameter tuning), and thus imposes a relatively high computational overhead. As for

various data science tasks, parallel and distributed processing can enhance the computational performance and shorten the execution latency.

In this thesis, following additional background on CCM and the literature of parallel techniques, we describe different versions of CCM parallel implementation which take advantages of contemporary parallel and distributed processing techniques. For instance, the CUDA framework [37] provided by NVIDIA GPUs, the MapReduce framework [12] provided by Apache Spark (henceforth, "Spark") and the hybrid [50] framework provided by MPI/OpenMP will be studied and applied on CCM. The thesis then presents an informal performance evaluation and comparison of these frameworks. We conclude from the experiments that, with parallel techniques and cloud computing support, researchers can use CCM to confidently infer causal connections between larger time series in far less time than is required by extant libraries implementing the general CCM algorithm.

1.2 Research Goals

As mentioned earlier, Convergent Cross Mapping is an algorithmic technique based around the idea of shadow manifolds reconstructed via lag coordinate embedding. In order to estimate the causal relationship with confidence, proper parameters E, τ for the dynamic systems should be applied. Another limitation of CCM concerns the library size L and sample size R. Inference of the causal relationship is based on how ρ changes along with L, as judged by a sample of R such values for each value of L. Determining the ensemble of such values of ρ imposes a heavy computation workload. CCM has been restricted in its range of application because of the high computational complexity; for example, while the length of modern time series may run into the millions, use of rEDM with values of L in the range of just 5000 can require overnight computation. To address the computational disadvantages of CCM, variants for the algorithm have been considered and offer significant results. However, these studies suffer from other limitations, and they do not offer a general solution.

This thesis seeks to investigate means of supporting scientifically reliable causal inference and prediction in a reasonable time by applying multiple levels of parallelism on convergent cross mapping. In consideration of the increasing availability of affordable computer hardware supporting parallel computation in recent years, various algorithms have been redesigned to take full advantage of computational resources. For example, the application of deep learning has been made more accessible on account of the fact that the matrix-based networks can be accelerated using GPUs. Additional Big Data related topics have become popular following the decreasing cost of computer hardware, and the introduction of cluster computing. The thesis being investigated here is that, subject to availability of sufficient computational resources, the parallel implementation of CCM will dramatically reduce the computation time required to conduct the analyses of causality.

1.3 Thesis Organization

The thesis is organized as follows. Chapter 2 provides background regarding both convergent cross mapping and several parallel processing techniques. This chapter includes a literature review on the foundational papers characterizing the CCM technique, as well as current work which improves the performance of CCM in certain conditions, and its related applications. Chapter 3 introduces possible methods for GPU acceleration and optimization when performing CCM on a single machine. The performance comparison between CPU and GPU will be presented. The concluding section of the chapter will summarize the advantages and limitations of GPU acceleration. Chapter 4 follows the emphasis on parallel implementation by extending the parallel techniques to take advantage of clusters of machines in which implementations based on MPI and Apache Spark will be discussed and compared. Chapter 5 combines the parallel techniques proposed in Chapter 3 and chapter 4 to more fully exploit multiple levels of parallelism. However, the use of multiple parallel technologies elevates code complexity and imposes performance bottlenecks. Also, this chapter lays out an evaluation and discusses possible solutions. Finally, chapter 6 provides a concluding summary of this research and discusses the contributions of this thesis as well as promising prospects for future work.

Chapter 2

Background

2.1 Literature Review

There are two subareas of the literature relevant to the research purpose of this thesis: Convergent Cross Mapping and parallel and distributed programming techniques. Convergent Cross Mapping basics covered here focus on the essentials of the theory and recent applications. Past work on CCM improvements will be listed and compared based on their advantages and disadvantages. The coverage of background on parallel techniques introduces the reader to a subset of prominent existing parallel models and frameworks affording ready application to convergent cross mapping to improve the computation speed. Parallel methods covered relate to two primary types of resources: The hardware architecture (which we call parallel computers) and its corresponding parallel software models. The resulting parallelization is achieved with the support of both hardware and software implementation.

2.1.1 Convergent Cross Mapping Basics

In 2012, Sugihara [53] built on ideas from Takens' Theorem [55] to propose convergent cross mapping (CCM) to test causal linkages between non-linear time series observations. This approach has enjoyed diverse applications. For example, Luo [39] successfully revealed underlying causal structure in social media and Verma [57] studied cardiovascular and postural systems by taking advantages of this algorithm.

In empirical dynamic modeling, time series – samples on the time axis – reflect the system states or behaviors, and can be understood as sequential projections of the underlying state of the associated complex system. In accordance with the theory of time series embedding, such a time series encodes information about the aspects of the system state that govern this variable. By Takens' theorem, to reconstruct a state space of dimensionality E – including unobserved (latent) variables within the system –, we can substitute each point of that time series by an E-dimensional vector whose elements are τ lagged values drawn from that time series. In order to assess if variable X is causally governed by variable Y in CCM, we attempt to predict the value of Y on the basis of the state space reconstructed from X (see below). For statistical reliability, this prediction must be repeated over a large number R of realizations. To assess causality, we examine whether these results "converge" as we consider a growing number L of data points within X within our reconstruction (also see below).

We provide here a brief intuition for why and how CCM works. Consider two variables X and Y, each associated with eponymous time-series and – further – where X depends on Y. For example, consider a case where for each time point X measures the count of hares, and Y that of lynx. If X (hares) causally depends on Y (lynx), the dynamics of X (e.g., a rapid rise or persistent drop in the hare population) will often tell us much about the state of other areas of the system that governs it, including Y (e.g., that there are likely to be few or many lynx around, respectively). The converse is true as well, if Y (lynx) causally depends on X (hares), observing the values of Y over time (e.g., a steep drop or a plateauing in lynx numbers) tells us about the state of governing factors, including X (here, the fact that the number of hares is too small to feed the lynx population effectively, or that they are roughly in balance with lynx, respectively). An implication of the first of these cases, where X depends on Y – captured by Takens' Theorem – is that information on the state of Y is encoded in the state space reconstructed from X, meaning that points that are located nearby within X's reconstructed state space will be associated with similar values for Y, and can thus be used to make accurate (skillful) prediction of the value of Y. In most cases, such prediction of one variable (e.g., Y) within the reconstructed state space of another (X) can be achieved by nearest neighbor forecasting using simplex projection [54] – that is, by considering the contemporaneous value of Y associated with the nearest neighbours to the point being considered in shadow manifold M_X. Pearson's correlation coefficient between observed and predicted values of Y can be applied over a "library" of a given length L to measure prediction skill. Details regarding the CCM algorithm can be found in Section 2.3.

Simplex Projection

Simplex projection, employed by convergent cross mapping, can be a valuable tool to distinguish chaotic time series from random noise. The central idea behind this tool is that the behavior of similar events in the past can directly forecast the events in the future. For the sake of simplicity, it involves tracking the evolving forward pattern among nearby points in the embedding manifold we reconstruct using lagged values. So, it belongs to a kind of nearest-neighbor forecasting algorithm with relatively high computational complexity.

There are two parameters associated with the simplex projection stage of CCM: Embedding dimension E and lag τ to create lagged-coordinate vectors for the manifold M_X. A high-fidelity one-to-one map will be presented between the reconstructed attractor and original attractor if the appropriate parameters are chosen. If the estimation of E is smaller than the appropriate one, the reconstructed states will directly overlap with each other, as they exhibit separation and structure higher dimension that may project to the same region of lower embedding dimension. As such, poor estimation of the parameter E can lead to poor forecast performance, and, as a result, the system behaviors cannot be captured within the reconstructed shadow manifold. Sugihara & May [54] use prediction skill as an indicator to identify whether E is the optimal embedding dimension. In this paper, the author argues that if we observe that forecast skill peaks at $E = 2$, it indicates that the real attractor manifold underlying input time series are unfolded best in 2 dimensions, which means E should be 2 to best approximate the real embedding dimension. Another

similar concept relates the actual dimensionality of the dynamic systems, which is not an exact equivalent to the embedding dimension. Estimation of the actual dimension of the corresponding dynamic systems should consider additional factors, including observational error, process noise, and time series length, and the system modes. Notably, the actual dimensionality of a dynamic system is often determined by the underlying state variables associated with it, while the dimensionality of the manifold E is the dimension that gives the highest prediction skill, which generally smaller than the actual dimensionality of the state space in complex systems [39].

Pearson's Correlation Coefficient

When applied to a sample, Pearson's correlation coefficient – also called the sample Pearson correlation coefficient, or simply put as the sample correlation coefficient – is commonly represented by r_{xy} or ρ_{xy}. This metric is widely used across various domains to measure the relation between observations. In CCM, the (Pearson) correlation coefficient between a sample predicted and a sample of observed values is treated as the prediction accuracy (skillfulness), which primarily serves as testing the degree to which information regarding Y is captured within the state space of X. Closely resemblant local topological structures suggest a causal connection between time series Y and X. With the correlation coefficient, evaluating such prediction skill becomes less computationally intensive and is relatively straightforward. Given two sequences in any of the sample, predicted values py and corresponding values y of equal length $E+1$ (representing the $E+1$ nearest neighbors found in BF kNN search), the Pearson's correlation coefficient between two sequences is defined as the covariance of the two sequences divided by the product of their standard deviations. That is, given paired data $\{(py_1, y_1), \ldots, (py_{E+1}, y_{E+1})\}$ consisting of $(E+1)$ pairs, $r_{py,y}$ is defined as:

$$r_{py,y} = \frac{\sum_{i=1}^{E+1}(py_i - \overline{py})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{E+1}(py_i - \overline{py})^2}\sqrt{\sum_{i=1}^{E+1}(y_i - \overline{y})^2}} \tag{2.1}$$

Where $\overline{py}$ and $\overline{y}$ are the sequence mean for py and y, respectively. However, the standard correlation coefficient is just one of several alternative measures of the agreement between predicted and observed values. There are some other methods that have been applied to measure the prediction accuracy. Mean absolute error of predictions (MAE) and root mean squared error of predictions (RMSE) serve as alternative measures to the Pearson correlation coefficient metric when measuring skillfullness. The reporting of skillfulness using such metrics can be supported by the Sugihara research group's contributed public library rEDM. However, among these metrics, the Pearson's correlation coefficient remains the most commonly used, and the parallel implementations in this thesis only consider this measurement in producing reliable prediction results across an ensemble of R different samples.

Identify Causality

A key need within CCM is to distinguish *causal* dependence of one variable on another from purely statistical dependence (e.g., due to covariation) – which might also support high prediction skill within a given reconstructed manifold. The two can be distinguished by assessing how prediction skill changes as the number data points used in the embedding – the so-called *library size* L – rises. To assess whether X is causally governed by Y, we evaluate how successively larger counts L of data points in X change the skill with which values of Y can be predicted. If such prediction skill rises monotonically with L, it indicates that Y is causally driving X (putting aside certain exceptional cases). By contrast, the presence of a merely statistical dependence of X on Y may lead to a high level of prediction skill even in small libraries, but will not lead to such convergence as L rises – with a monotonic rise in prediction skill to some plateau. In order to confidently assess this convergence with rising L in light of stochastic selection of the library, the prediction skill must be assessed over R random subsamples of the time series for each value of L. The speed of convergence and the magnitude of the achieved prediction skill for large L indicates the existence and strength of the causal dependence of X on Y in the context of the assumed values for E and τ.

2.1.2 Past work in CCM Performance Improvement

Despite the fact that CCM is increasingly widely applied, there remain pronounced computational challenges in applying the tool for the moderate and large time series that are prominent features of the "big data" era. At the same time, securing confidence in inferences regarding causality in the dynamic systems makes highly desirable not just use of appropriate parameter values, but also a least a moderately long time series are required for the original CCM [45] – a time series with length well into the hundreds, if not the thousands of observations. As such, since CCM's first appearance in 2012, a number of modifications and improvements have been proposed to handle this drawback. In 2014, Ma et al. [41] developed cross-map smoothness (CMS) based on CCM, which has the advantage of allowing for a shorter time series. Compared to original CCM, CMS can be used for time-series of length in the order of $T = 10$, whereas CCM arguably requires time-series of length in the order of $T = 10^3$ to yield reliable results.

Additionally, works [7], [54], [31], [33] investigated and introduced mathematical methods to properly estimate parameters required by CCM (embedding dimension E, time delay τ and library length L). For example, from the previous study of CCM, estimation of the embedding dimension method in nonlinear theory from the underlying attractor often begins with the reconstruction the state space, and then a calculation of the dimension of the putative attractor using some variant of the Grassberger Procaccia algorithm [15]. A correlation integral is calculated in this algorithm to estimate the dimension E. Such work expanded CCM-related research and also provided methods for quickly inferring causality in certain circumstances. However, previous research has not sought to accelerate CCM using parallel techniques. With the growing prevalence of hardware and software support for effective parallelization, it is worthwhile to investigate the opportunities

for elevating performance, including by parallelizing the hyperparameter tuning process. As with many other machine learning algorithms, through effective use of parallel techniques, exhaustive searches on discrete parameter grids can be performed and compared without explicit empirical estimation of different CCM parameters.

2.2 Introduction of Parallel Methodologies

2.2.1 Overview and Categories

Parallel processing involves the simultaneous execution of multiple computational processes. Application of such techniques generally reduces the total computational time but requires support in the form of parallel algorithms, programming languages with the backing of parallel algorithms, multitasking operating systems and often multi-core hardware [47]. As such, hardware platform and software environments should be taken into consideration together to exploit computation performance benefits from parallel computing [34] effectively.

According on the feature of the instruction and data stream, computers can be categorized into four classes based on Flynn's taxonomy [17]:

- Single instruction stream and single data stream computers are simplified as SISD.
- Single instruction stream and multiple data streams computers are simplified as SIMD.
- Multiple instruction streams and single data stream computers are simplified as MISD.
- Multiple instruction streams and multiple data streams computers are simplified as MIMD.

SISD has no parallel capability, and it is believed that MISD does not physically exist in the industry. Most parallel techniques and programs rely instead on SIMD and MIMD computers, which can support parallelization.

SIMD computers contain one control unit and multiple processing units. In today's context, such multiple processing units often refer to GPU many-core architecture. In GPUs, every thread in a thread block executes the same instruction simultaneously on different data.

By contrast, many MIMD systems can be characterized according to the memory model employed. In *shared-memory systems*, multiple CPUs share the same physical memory. On the other hand, message-passing systems are typically featuring distributed CPUs with independent memory for different sets of CPUs, such as those encountered in contemporary computational clusters, with communication taking place via message-passing mechanisms. As such, in this section, three types of modern parallel computers and the basics of the corresponding program paradigms will be discussed.

2.2.2 GPU architecture and CUDA programming

Graphics processing units (GPUs) [6] were initially designed to satisfy the demand for higher quality graphics in video games, so as to create a more realistic 3D environment. In the past decade, GPUs have gradually evolved to highly influential parallel processing platforms, offering high throughput on account of the enormous number of cores. Unlike multi-core machines, only with the ability to run just a few threads in parallel at one time – for example, four threads at the same time on a quad core machine – GPUs can run hundreds or thousands of threads concurrently. Although various restrictions apply, the high potential that such devices provide for performance enhancement is the fundamental reason underlying their popularity.

NVIDIA CUDA architecture consists of a large set of streaming multiprocessors (SMs), where each such SM includes some streaming processors (SPs). One SP can execute precisely one thread [43], and one SM can run groups of threads in lockstep. As such, the SM/SP hierarchy addresses synchronization mechanisms for independent subsets of data on the GPU device. Only SPs can be synchronized by means of critical section mechanisms, such as via declaring synchronization barriers. By contrast, the independence of threads in distinct SMs means that the hardware can run faster if the algorithms have specific independent chunks which can be assigned to different SMs. Parallel invocations of kernel functions to be undertaken concurrently are grouped into blocks, which are then distributed among available SMs. Each block has up to three dimensions – reflecting its original use in processing 3D images for video games – and contains a maximum of 1024 threads. GPU devices must be attached to a CPU host to operate. The communication between host and GPU is via a PCI-Express bus. Each GPU has its own memory, which is associated with a memory space disjoint from that of host memory. To work with GPUs, data and instructions have to be copied back and forth between memory in hosts and GPUs through the PCI-Express bus. As such, the data transfer overhead can serve as the main bottleneck for the performance. Also, each GPU consists of different types of memory. For instance, the device memory – referred to as global memory when using CUDA API function *cudaMalloc* – is large and accessible by all threads in different blocks, but has lower throughput. By contrast, shared memory, allocated by declaring variables using CUDA API function *__shared__*, is small but fast, but is only allocated for the corresponding thread block. Figure 2.1 depicts the memory structure associated with GPUs, but omits other types of memory, such as registers and caches.

To facilitate efficient general purpose computing on GPUs, NVIDIA has developed the Compute Unified Device Architecture (CUDA) language [37] as a vehicle for programming on their GPUs. Syntactically, the language essentially serves as a slight extension of C and aims to provide a uniform interface that works with multi-core machines in addition to GPUs. A CUDA program consists of two portions: the code to be run on the host (CPU part) and the code to run on the (GPU) device (GPU part). A function that is called by the host to execute on the device is called a kernel function, which is identified by *__global__* keyword. The same kernel function is typically performed by thousands of threads, which are grouped into blocks. Two CUDA structures, threadIdx (thread index) and blockIdx (block index), are used in combination to associate a thread with a different piece of data for parallel data computation. And the threads in the same block can

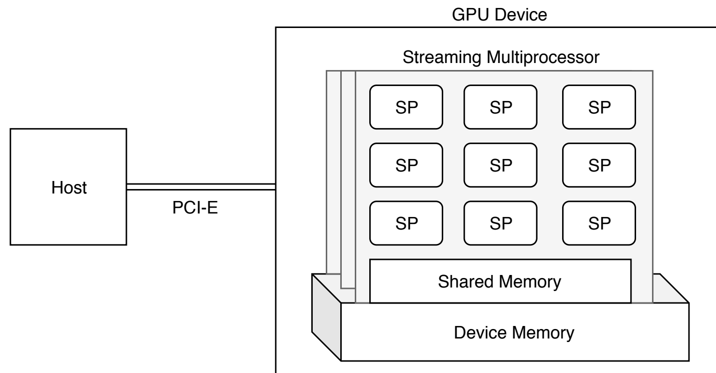

Figure 2.1: GPU device architecture overview.

synchronize their operation, using the *__syncthreads()* keyword.

2.2.3 Multi-core machine architecture and multithreaded programming

A standard contemporary MIMD computer is the shared-memory multi-core machine [14], which has multiple CPUs, as shown in Figure 2.2. Parallel execution on this kind of machine is typically achieved via multithreading. A thread is similar to an operating system (OS) process, but with much less overhead, and without a large dedicated space. Most current programming languages, including C++, Java and Golang, support multithreaded programming. Efficient parallel execution of a program requires parallel accessing of memory. This task is facilitated by dividing memory into separate modules or banks. This way accesses to different memory elements can be undertaken in parallel. However, the conflict of memory access (read or write operation) by different threads can lead to data inconsistency. Critical section operations are introduced to address this problem. Such barriers enforce the constraint that more than one thread is not allowed to execute the code simultaneously. To achieve a certain degree of synchronization, several methods (lock, mutex) [44] should be applied in multithreaded programming. However, a critical section typically serves as a potential bottleneck in a parallel program, as this part is serial instead of parallel. Another potential bottleneck is imposed by designated barriers, which are places in the code that all threads must reach before continuing. The existence of such obstacles may result in some threads being idle, while other threads still have a large amount of work.

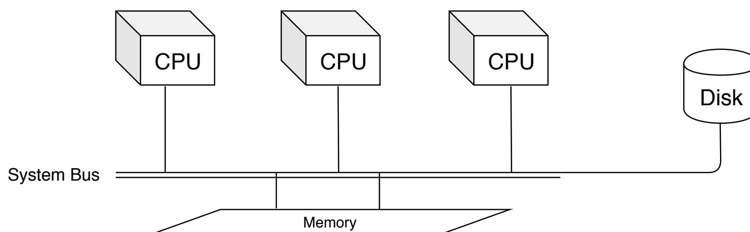

Figure 2.2: Shared memory system architecture overview.

The Open Multi-Processing (OpenMP) library [11] in C++ offers researchers a higher-level of threading

by hiding lower-level details. It primarily includes a collection of compiler directives and callable routines to express shared-memory parallelism. In order to support designing parallel algorithms without handling threads, this library provides developers with pragma descriptive commands such as *parallel*, *barrier*, *critical*, etc..

Apache Spark [64] with the local mode is another framework which can exploit parallelism implicitly on a multi-core machine through Java Virtual Machine (JVM). Spark shows better data management in the cluster and introduces the transformation pipelines on the immutable in-memory data structures (Resilient Distributed Dataset) [63]. Overall, these popular parallel techniques can be applied to the multi-core machine conveniently to improve algorithm execution speed as a whole.

2.2.4 Cluster architecture and distributed programming

Another popular MIMD computational framework consists of distributed multiple machines with network-based connections – a configuration referred to as a cluster. However, the network is a notable weak point in this kind of systems. The nodes would have separate copies of the data to process in parallel as the distributed memory model. Due to the distributed memory model and network latency, the scatter or gather operations will increase the overall latency of the cluster. Such network communication and data transfer can be a central bottleneck for computation performance. However, the cluster can be efficiently scaled out by adding more nodes (computation resources). Another apparent advantage is that the execution time can decrease dramatically as more workers share the workload for particular jobs.

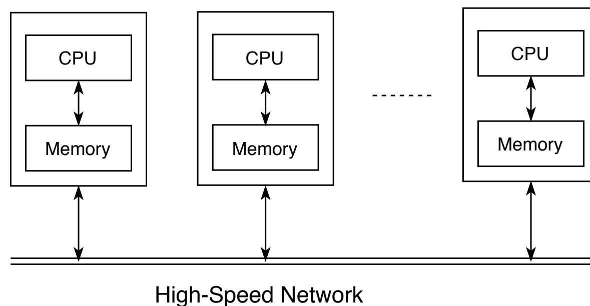

Figure 2.3: Distributed memory system architecture overview.

At present, the Message Passing Interface (MPI) [29] and Apache Spark platform on Yarn [56] each offer powerful interfaces for performing applications at scale across computational clusters. As such, we will compare the performance of these two methods in scaling the Convergent Cross Mapping algorithm.

The APIs in the MPI framework utilize an in-memory and in-place programming model. So the computations and communications take place in the identical process under the same scope, which means that different processes execute the same code and communicate to one another in the same group (MPI_COMM_WORLD) with standard MPI protocols [30].

By contrast, the Big Data framework Apache Spark in Yarn mode, has adopted a directed acyclic graph

(DAG) transforming workflow and execution model in order to process large amounts of data. In this kind of programming model, operators are applied on distributed data sets which produce different distributed data sets in the cluster. This concept provides robust yet straightforward programming APIs. These APIs are usually written following functional programming principles [49], making them less error prone and easy to program. A DAG transforming execution model separates the communications and computations by allowing computing to occur in self-contained tasks, and not permitting communication within task execution. The jobs undertake stateless computations on the data. More importantly, Apache Hadoop Yarn offers schedule and resource manager services within the cluster. Without explicit coding, Yarn can perform its scheduling function based on the resource requirements of the submitted job. This stands in contrast to MPI, which represents static resource allocation without any automation. As such, Yarn can fully take advantages of cluster resources and improve their utilization.

Overall, there are three popular architectures whose performance is investigated in this thesis: GPUs, shared-memory systems and message-passing systems. Each different parallel system has its performance advantages and bottlenecks. Pronounced bottlenecks for the shared-memory system arise from critical sections and barriers, while notable bottlenecks associated with message-passing system mainly come from network communication. As for GPUs, the overhead of allocating memory and loading data between CPU memory and GPU memory through the PCI-Express bus is often a central bottleneck, as it requires data to be transferred for operations on GPU cores. In this thesis, different parallel techniques are adopted to lower the bottlenecks and delay in the parallel version of the CCM application to minimize execution time.

2.3 CCM Algorithm Analysis

2.3.1 Public Library of CCM: rEDM

Empirical dynamic modeling, often referred to as EDM, is an advancing non-parametric method for modeling nonlinear dynamic systems. The **rEDM** package [61] in the R statistical package includes several EDM methods, including convergent cross mapping, and is published by the research group of the author of CCM, the Sugihara Lab (University of California San Diego, Scripps Institution of Oceanography). This free package from the R CRAN repository can be readily installed on any machine with R by running the command *install.packages(”rEDM”).* to perform causal inference by invoking the **ccm** API function. The **rEDM** functions are designed to accept and generate data in common R data formats. The library is available in an open source capacity on Github. The **rEDM** library author used the performance-limited technique of C++ single threading to implement convergent cross mapping algorithms: the **ccm** function. This library is implemented using C++ single threads with an R language interface, which can only be executed on a single machine.

2.3.2 Parallel Design of CCM Algorithm and Comparison

The main motivation behind developing parallel algorithms lay in the motivation to reduce the computation time of an algorithm using parallel machines. Thus, evaluating the execution time and corresponding time complexity of an algorithm is extremely important in evaluating the success of these efforts. In the analysis of parallel algorithms, the number of processes n ($n > 1$) is normally introduced when considering time complexity. Also, there are several additional parameters involved for the CCM algorithm, which are listed in Table 2.1 – $LSet$, $ESet$ and $TauSet$. A detailed analysis of the time complexity will be presented afterwards.

Table 2.1: Notation

X, Y	Two variables in the form of time series
$\hat{X}_t$	The time-lagged vector at time t in time series X
M_X	The shadow manifold reconstructed using time lags in X
$\widehat{Y}_t \vert M_X$	The estimate of variable Y obtained by cross mapping using the shadow manifold M_X
$LSet$	The set of subsequences lengths (hyper parameter candidates)
L	The length of subsequences
$ESet$	The embedding dimensions set (hyper parameter candidates)
E	The embedding dimensions of shadow manifolds
$TauSet$	The τ set (hyper parameter candidates)
τ	The embedding delay used in the shadow manifold reconstruction
T	The full length of the input time series
R	The number of realizations (samples)
n	The number of processes

As mentioned earlier, CCM is based on simplex projection. The simplex projection belongs to a nearest-neighbor searching algorithm that estimates kernel density using exponentially weighted distances on the reconstructed shadow manifold. Consider two time series of length T as input, $X = \{X_1, X_2, ..., X_T\}$ and $Y = \{Y_1, Y_2, ..., Y_T\}$ and the design parameter $L \in LSet$, $E \in ESet$, $\tau \in TauSet$ and R. CCM begins by constructing the lagged-coordinate vectors $\hat{X}_t = < X_t, X_{t-\tau}, X_{t-2\tau}, ..., X_{t-(E-1)\tau} >$ for the range of $t \in \{1 + (E-1)\tau, T\}$. This set of lagged-coordinate vectors is often referred to as the shadow manifold M_X. In order to produce the cross-mapped estimate of target value Y_t, denoted by $\widehat{Y}_t|M_X$, for each sample $\in R$, randomly draw L embedding vectors from the full time series. The next step consists of locating, for each embedded point associated with time t, the corresponding lagged-coordinate vector $\hat{X}_t$ on M_X and finding its $E+1$ nearest neighbors. Next, sort the indices based on the distance from $\hat{X}_t$ in ascending order to obtain top $E+1$ nearest neighbors. Note that $E+1$ is the minimum number of points required to bound a simplex projection in an E-dimensional space. The equation below is then applied to obtain the estimate $\widehat{Y}_t|M_X$ of

variable Y:

$$\widehat{Y}_t|M_X = \sum w_i Y_{t_i} \tag{2.2}$$

Where the index $i \in [1, E+1]$ from the sorted indices list and the exponential weight w_i is based on the distance between point $\hat{X}_t$ and its ith nearest neighbor:

$$w_i = \frac{u_i}{\sum u_j}, j = 1...E+1 \tag{2.3}$$

$$u_i = \exp^{\frac{-d[\hat{X}_t, \hat{X}_{t_i}]}{d[\hat{X}_t, \hat{X}_{t_1}]}} \tag{2.4}$$

where $d[\hat{X}_t, \hat{X}_{t_i}]$ represents the Euclidean distance between two lag vectors. Finally, Pearson's correlation coefficient is applied to evaluate the similarity of the estimated sequence (predicted) $\widehat{Y}_t|M_X$ and target sequence Y_t over all points in the library. The coefficient value indicates how skillful they match.

Serial Version

The serial algorithm pseudocode in Algorithm 1 presents how CCM in rEDM evaluates the existence and strength of a possible causal connection: $Y => X$. As is clear from the listing, there are several nested loops, which can be parallelized in accordance with the dependencies associated with the calculation.

Algorithm 1 CCM serial algorithm

1: INITIALIZE $\rho = \emptyset$
2: **for** E in $ESet$, τ in $TauSet$, separately **do**
3: Construct shadow manifold M_X for embedding dimension E and delay τ on X
4: **for** L in $LSet$ **do**
5: **for** *sample* = 1,...,R **do**
6: $Seq \leftarrow$ randomly draw sampled lagged vectors of length L from M_x with replacement.
7: **for** query point q in Seq **do**
8: $Dist_{Seq} \leftarrow$ calculate Euclidean distances to q for $R \in Seq$.
9: $Seq_{sort} \leftarrow$ sort Seq based on $Dist_{Seq}$.
10: $K \leftarrow$ find top $E+1$ indices in Seq_{sort}.
11: $\rho \leftarrow$ calculate correlation $\rho^{E,\tau,L}_{sample}$ between Y_t and $\widehat{Y}_t|M_X$ with indices $t \in K$.
12: **end for**
13: **end for**
14: **end for**
15: **end for**

Generally, there are two steps in the serial implementation. The first step lies in constructing the shadow manifold M_x for all combination of parameters E and τ. The time complexity is $O(|ESet| \times |TauSet| \times E \times T)$. While in the second stage, state space searching, the time complexity is $O(|ESet| \times |TauSet| \times |SetL| \times R \times$

$L^2 \times log(L))$ when we consider the time complexity of the sort operation as generally being $O(klogk)$ for problem size k.

Parallel Version

As we can observe from the serial version of CCM adhered to by the rEDM package above, repeated calculation of distances and sorting operations can be significant performance bottlenecks. As such, a parallel design of CCM is proposed in this thesis, one which trades added space consumption for a reduction in execution time. To the end, a global sorted distance matrix can be calculated and memoized for further need. Such an approach can pave the way for GPUs or clusters to process the data-intensive task, the derivation of the sorted distance matrix, in a more efficient way. The pseudocode of the CCM parallel algorithm is presented in Algorithm 2.

Algorithm 2 CCM parallel algorithm

1: INITIALIZE $\rho = \emptyset$
2: **for** E in $ESet$, τ in $TauSet$, separately **do**
3: Construct shadow manifold M_X for embedding dimension E and delay τ on X
4: **for** i in T in parallel **do**
5: $GDist_{Seq} \leftarrow$ calculate Euclidean distances of i to $q \in T$.
6: $GSeq_{sort} \leftarrow$ sort point index based on $GDist_{Seq}$.
7: **end for**
8: **for** L in $LSet$ **do**
9: **for** $sample$ = 1,...,R in parallel **do**
10: $Seq \leftarrow$ randomly draw sample from M_x with replacement L times (yielding a L-length vector).
11: **for** query point q in Seq **do**
12: $K \leftarrow$ find top $E+1$ indices for q in $GSeq_{sort}$.
13: $\rho \leftarrow$ calculate correlation $\rho^{E,\tau,L}_{sample}$ between Y_t and $\widehat{Y}_t|M_X$ with indices $t \in K$.
14: **end for**
15: **end for**
16: **end for**
17: **end for**

For the parallel version, the first preprocessing step takes time $O(\frac{|ESet|\times|TauSet|\times T^2\times logT}{n})$. As such, the overall time complexity of the state space searching with the preprocessed data is $O(\frac{|ESet|\times|TauSet|\times|LSet|\times R\times L^2}{n}) < O(\frac{|ESet|\times|TauSet|\times R\times T\times L}{n})$ as $L < T < |LSet| \times \bar{L}$. The total input time series length is larger than the subsample sequence L but for regularly sampled values of L up to size T, always smaller than the average length of the subsample sequence times the number of testing sets for L: $|LSet|\times\bar{L}$. In this case, the time complexity is determined by the preprocessing step. As such, the overall time complexity is $O(\frac{|ESet|\times|TauSet|\times T^2\times logT}{n})$,

which is smaller than that for the serial version $O(|ESet| \times |TauSet| \times |LSet| \times R \times L^2 \times logL)$ even when n is 1. When we apply the parallel version in the cluster, the power of parallel execution (as captured by n) is expected to dramatically decrease the overall execution time of CCM.

2.4 Summary

Previous improvements on CCM typically trade off potential accuracy for relatively fast execution, and the assumptions in some methods cannot be safely maintained in specific contexts, such as noisy time series observations. However, the computational performance of the original sequential CCM can be improved by the introduction of parallel computing or heterogeneous computing techniques. With a parallel design for the CCM algorithm, the overall execution time can be significantly reduced using GPU devices or other distributed computing frameworks such as MPI or Spark [42], [51]. Most notably, use of a global distance matrix and sorting operation as a preprocessing stage can be a good fit for GPU computing or cluster techniques. GPU devices are excellent for massive data-parallel workloads, which are integrated as the leading accelerators for deep learning based algorithms and image-related process tasks. Also, cluster parallel methods can dramatically improve algorithmic performance by effectively exploiting cluster-based or heterogeneous computational capacity. In light of the established opportunities for such performance enhancement, a parallel version of CCM should be implemented to allow researchers to evaluate the existence and strength of causal connections between the measured time series in a robust and lower-latency fashion.

Chapter 3

Exploiting GPU Acceleration of Convergent Cross Mapping

3.1 Introduction

Reliable causal inference via CCM as to whether one time series variable (e.g., Y) is causally driving another time series variable (e.g., X) requires estimation of the degree to which, given a particular time series point t, prediction of the value of Y_t can be made on the basis of the closest points to the embedded vector corresponding to X_t within the reconstructed shadow manifold M_X in embedding dimension E. There are several steps related to this estimation. The first step takes advantage of Takens Theorem [55] establishing that mathematically valid and equivalent reconstructions of an attractor can be created using lags of just a single time series. In a second major step, nearest neighbour forecasting in simplex projection [54] given the library size L can be applied on the reconstructed attractor to identify the unique states. Finally, Pearson's correlation coefficient equation can be utilized to estimate the prediction accuracy on the basis of the information of the nearest neighbours in the reconstructed state space M_X.

In this process, k nearest neighbours searching, which can be framed in terms of an instance of the general k nearest neighbour (kNN) problem, has been used to define the similarity in reconstructed embedding space between two variables. Unfortunately, these estimations are computationally intensive, since they rely on searching neighbours among large sets of E-dimensional embedding vectors. Generally, this computational burden can be reduced by pre-structuring the data, e.g., using KD trees as proposed by the approximated nearest neighbour library [46] or using Morton ordering to preprocess it in parallel [9]. Yet, the opening of graphics processing units (GPU) to general-purpose computation by means of the CUDA API and competing platforms such as OpenCL offers researchers a robust platform with notable parallel calculation capabilities. Garcia [19] proposed a CUDA implementation of kNN search and compared its performance to that of several CPU-based implementations, demonstrating a speed increase by up to one or two orders of magnitude.

A further possible improvement for CCM is to implement the GPU-based Pearson's correlation coefficient for multiple samples R. Work such as Chang [8] studied and compared the performance between CPU-based and GPU-based implementation of the Pearson's correlation coefficient function, and their results show an approximately 40x speedup for large input data size with the support of powerful GPUs.

3.2 Methodology

3.2.1 Brute Force kNN Search

Principle

The kNN search is a common topic in similarity-related problems and the most common metrics to describe closeness is Euclidean distance. Considering the background of CCM, it can be characterized as follows: Let $RP = rp_1, rp_2, ..., rp_T$ be a set of T reference points (lagged vectors) with values in embedding space $\mathbb{R}^E$, and let $QP = qp_1, qp_2, ..., qp_T$ be a set of T (the length of time series) query points in the same embedding space. The kNN search problem requires identification of the k ($k = E + 1$) nearest neighbours of each query point $qp_i \in QP$ in the reference set RP, given a specific distance metric to describe the similarity. Commonly, the Euclidean distance is used, and two sets (the query set and reference set) are the same in CCM, with the proviso that the k nearest neighbours to a specific reference and query point rp_i cannot include the reference point at index i itself. Figure 3.1 gives an overview of kNN search as a part of the CCM algorithm with $k = 4$ and for a point set with values in $\mathbb{R}^E$.

One straightforward approach to address kNN search is the brute force (BF) algorithm. For each query point qp_i, the BF algorithm is the following:

1. For each query point qp_i (the red one in Figure 3.1), do:
2. - Compute all the distances qp_i and $rp_j, j \in [1, T]$.
3. - Sort the points based on the computed distances.
4. - Select the k reference points corresponding to the k smallest distances after excluding itself $[j! = i]$.
5. Repeat above steps for all query points QP.

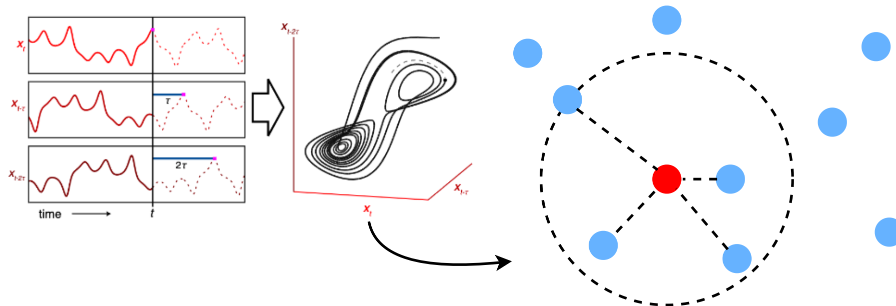

Figure 3.1: Image adapted from [53] illustrates the kNN search problem given the embedding dimension parameter $E = 3$, which means that k is 4, in the reconstructed state space. The blue points refer to the points in RP set while the red points are in QP set. Reflecting the fact that $k = 4$, the circle gives the distance between the query point and the fourth closest reference point.

The main issue of this algorithm lies in its high computational complexity: $O(T^2E)$ for the distance computations and $O(T^2 logT)$ for the sorting operation. Several kNN algorithms have been proposed in the literature [21], [40], [10] to reduce the computation time. However, some methods involve preprocessing these points and maintaining a kind of structure to query which is not readily parallelizable on GPUs. Also, other algorithms are designed with the support of recursion, which is not suitable for GPU architecture. Actually, the first two steps of the BF algorithm are already highly-parallelizable as they can be formed as matrix-based calculation. With the support of powerful GPUs, research such as [20] demonstrates that the BF algorithm can generally be faster than a corresponding CPU-based implementation by up to a factor 10.

Parallel pairwise distance calculation description

The computation of this matrix was fully parallelized, reflecting the fact that the distances between pairs of points are independent: Each thread computed the distance between a given query point $qp_i \in QP$ and a given reference point $rp_j \in RP$. Particular in CCM algorithm, both points set QP, and RP refer to the lagged-vector in reconstructed shadow manifold space M_X, where it contains T lagged points with E embedded dimension in total. To achieve such parallelism on GPU, the distance calculation work should be divided into CUDA programming threads, and then the GPU work distributor will automatically allocate thread blocks to Streaming Multiprocessors in GPU. Threads are then divided into groups, which contain 32 threads as a warp, and these wraps will be dispatched to execution units. It is worthwhile to note that padding is required when the input data is not divisible by 32. This is the constraint brought by the attribute of GPU architecture.

Parallel sort algorithm description

Along with the increasing popularity of GPU programming, various algorithms are designed and implemented for GPU architectures. In 2009, Satish et al. [52] introduced several efficient parallel sorting algorithms for manycore GPUs, taking advantages of the full programmability offered by CUDA. The most efficient one is radix sort, which reduces the complexity of sorting n input records to $O(n)$, as it uses a counting sort rather than a comparison sort approach.

Radix sort relies on the reinterpretation of a k-bit key as a sequence of d-bit digits, which are considered one at a time. The basic idea is that splitting the k bits of the keys into smaller d-bit digits results in a small enough radix $r = 2^d$, such that the keys can efficiently be partitioned into r distinct chunks. Figure 3.2 is an illustration of radix sort given the input array $(7, 2, 9, 0, 1, 2, 0, 9, 7, 4, 4, 6, 9, 1, 0, 9, 3, 2, 5, 9)$ of length $n = 20$. In order to rearrange them in ascending order, the radix sort algorithm adopts three steps for each chunk as a pass:

1. Count the occurrences of each number (as the key) to fill the Count Table.

2. Prefix sum scan over the count table to fill an Offset Table.

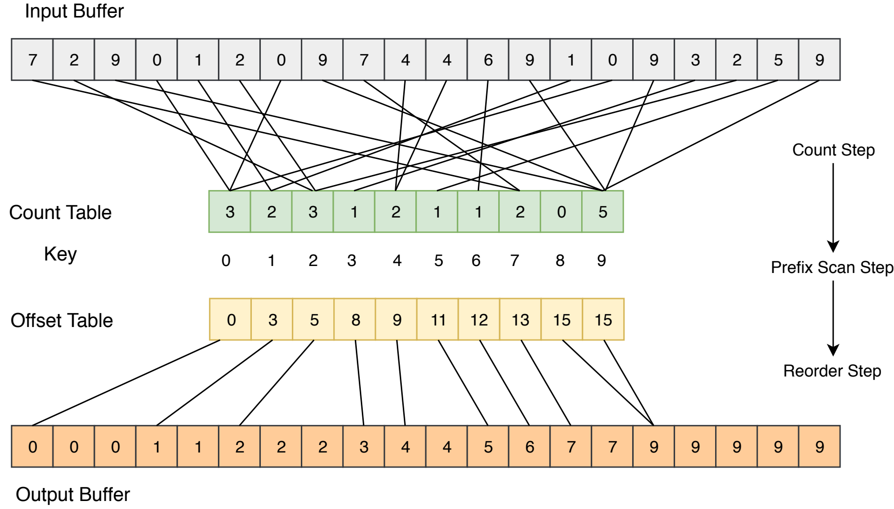

Figure 3.2: An example of a pass in the radix sorting operation on one chunk of length $n = 20$ within 4-bit digits, which means that the maximum value is less than $2^4 = 16$.

3. Reorder the number based on the Offset Table.

The prefix sum is the sum of all values in preceding locations in the sequence: In this case, those to the left of the current location. These values are treated as the beginning addresses of the corresponding keys in the output buffer.

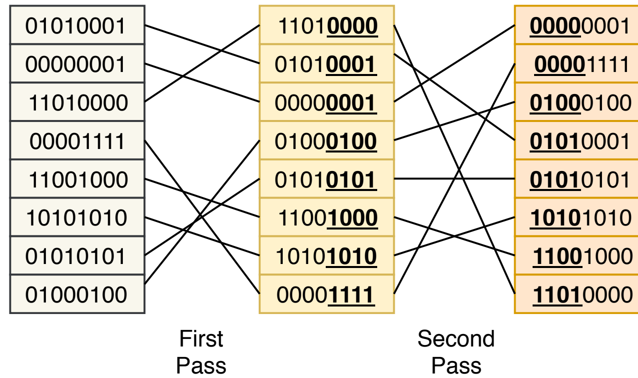

Figure 3.3: An illustration of applying the radix sort to sort $k = 8$-bit integers by 2 passes of three steps aforementioned on $d = 4$-bit integers from the least significant bits to the next higher significant bits, and so on.

Given input data of size n, radix sort performs k/d passes of three steps each, and each pass takes $O(n+2^d)$ time in a SISD architecture. Hence, the total time complexity for the input data with any integer $d > 0$ which contains k bits can be represented as $O((k/d)(n + 2^d))$. This algorithm can perform better under a GPU SIMD architecture. Since the primary performance bottleneck in the kNN search problem – the sort operation – can be addressed using radix sort with GPU acceleration, the computation time required for CCM can be dramatically reduced.

3.2.2 Pearson's Correlation Coefficient

In CCM, given two sequences in any of the sample, predicted values py and corresponding values y of equal length $E+1$ (the $E+1$ nearest neighbours found in BF kNN search), Pearson's correlation coefficient between two sequences is defined as the covariance of the two sequences divided by the product of their standard deviations. That is, given paired data $\{(py_1, y_1), \ldots, (py_{E+1}, y_{E+1})\}$ consisting of $(E+1)$ pairs, $r_{py,y}$ is defined as:

$$r_{py,y} = \frac{\sum_{i=1}^{E+1}(py_i - \overline{py})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{E+1}(py_i - \overline{py})^2}\sqrt{\sum_{i=1}^{E+1}(y_i - \overline{y})^2}} \tag{3.1}$$

Where $\overline{py}$ and $\overline{y}$ are the sequence mean for py and y, respectively. In CCM, the correlation coefficient r_{xy} is calculated for the actual and predicted values of y, which is based on the exponential-weighting of the $E+1$ nearest neighbour search results according to the distance. Also, the $r_{py,y}$ will be calculated repeatedly for R realizations to obtain reliable results. These values can be readily formed, as X with each row i is py, and Y with each row i is y for realization r_i. Hereby, the independence of row-based calculations makes it possible for a thread of CUDA kernel to process a pair of sequences at a time and produce a $1 \times R$ output matrix.

3.3 CUDA Implementation

NVIDIA released the CUDA language in 2008 [37], which is an extension of C. Briefly, the Single Program Multiple Data (SPMD) code is written using a CUDA kernel function, the data to be operated on are copied from CPU RAM to the global memory of the device, and the C program running on the CPU initiates the data-parallel computation via a kernel function call. A GPU kernel function contains the code that will be executed simultaneously by the GPU processors, and CUDA uses the function type qualifier *__global__* to declare that a function is a GPU kernel function at compile time. As such, the implementation covered here was written using the CUDA API and was composed of three kernels (CUDA functions) in the CCM which are executed on the GPU device.

- The first kernel calculates the pairwise Euclidean distance matrix of size $T \times T$ containing the distances among the T lagged points in reconstructed shadow manifold and the same lagged points, where each point actually is an E-dimensional lagged vector.
- The second kernel performs the radix sort given the distances between any query point and all T E-dimensional reference points, where such distances serve as the output of the first kernel.
- The third kernel computes the Pearson's correlation coefficient between two R-length sequences (predict sequence and corresponding target sequence), where each sequence contains $E+1$ nearest neighbours generated by the kNN search step.

3.3.1 Pairwise Distance Calculation Kernel

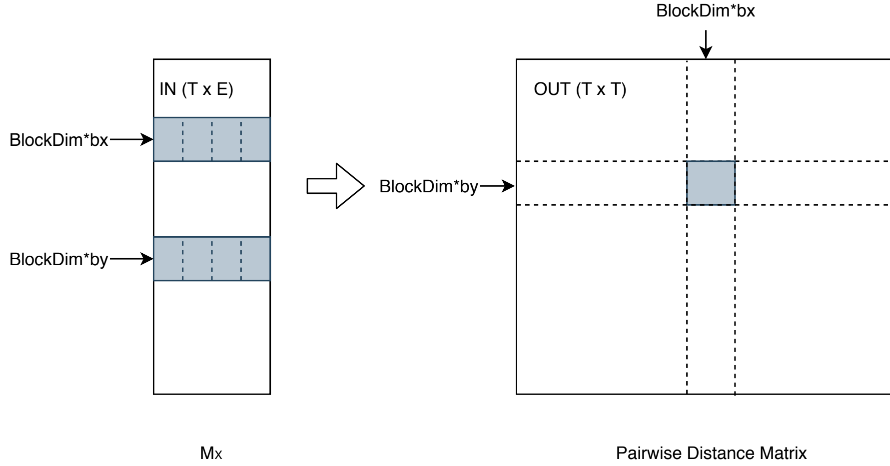

Figure 3.4: The parallel algorithm for pairwise distance calculation. Each block computes one sub-matrix of **OUT**, and the threads work on one pair of aligned sub-matrices of **IN**, as is illustrated as the figure.

The kernel input, a $T \times E$ matrix (reconstructed shadow manifold) in which each row is the E-dimensional lagged point, is stored at device memory for SP to access. The kernel computes the pairwise distances among these points (different pair of rows) and produces a $T \times T$ symmetric matrix as output, which is stored at device memory too.

The CUDA version for pairwise Euclidean distance algorithm, as shown in Figure 3.4, uses one thread for one entry in the **OUT** matrix, which means there are T^2 threads. The threads are organized into $BlockDim \times BlockDim$ two-dimensional blocks, which will be run on the GPU cores. And these blocks are organized into the $\frac{T}{BlockDim} \times \frac{T}{BlockDim}$ sub-matrix format. Generally, $BlockDim$ is supposed to be 16 for Maxwell or more advanced GPU architectures, while it is supposed to be 4 for Kepler or older architecture. A thread orients itself through its block and thread indices in the following way:

$$\begin{aligned} bx = blockIdx.x; by = blockIdx.y; \\ tx = threadIdx.x; ty = threadIdx.y. \end{aligned} \tag{3.2}$$

With the above coordinate system, a thread is responsible for calculating the entry in the matrix **OUT** at row $BlockDim * by + ty$ and column $BlockDim * bx + tx$. Let us assume that a thread needs to calculate the entry (i, j) in the **OUT** matrix. It will first load the two sub-matrices of **IN** anchored by the variable y and x at the corresponding upper left corners. The synchronization function $__syncthreads()$ needs to be called for imposing a barrier before accumulating its own partial Euclidean distance in a temp variable. Then the threads have to be synchronized again before processing the next pair of sub-matrices in **IN**. These procedures continue to be executed until all blocks finish the calculation on the corresponding entry.

3.3.2 Radix Sort Operation Kernel

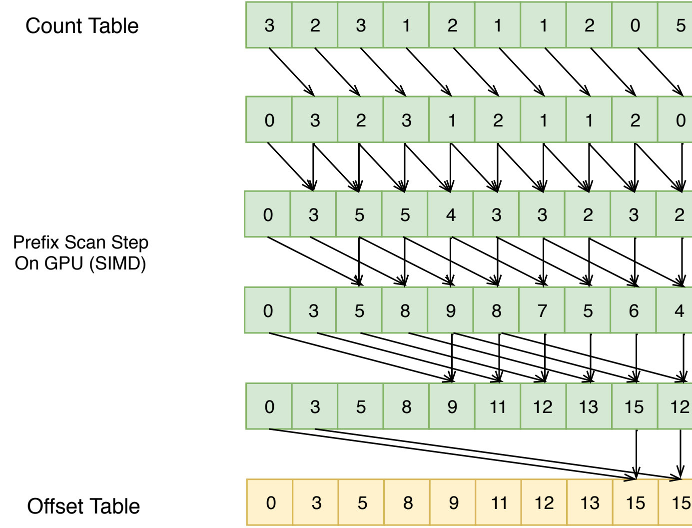

Figure 3.5: The parallel prefix scan algorithm. The interval is changed from 1 to interval $log_2 n$.

The radix sort applied on each chunk includes three stages: count, prefix scan, and reorder. For the counting stage, the input is an array of keys, and the output is a matrix of counts of each value in the input. The straightforward implementation is taking advantage of shared memory with an atomic increment operation on the counter of keys for each block. Then the count matrix for the entire input represents the summation of the individual tables in each block. Threads can be used to sum up all the count values and write them to the global memory. As for the prefix scan stage, the traditional sequential scan algorithm is poorly suited to GPUs because it does not take advantage of GPU data parallelism. The parallel version of the prefix scan is based on the algorithm presented by Hillis and Steele [27] and demonstrated for GPUs by Harris [25]. Figure 3.5 illustrates this operation. Although the parallel prefix scan performs $O(nlogn)$ addition operations, it takes $O(logn)$ time complexity to finish which can be a huge improvement in performance. The last stage is to write the data back to the appropriate location in global memory. For example, if the size of the block is $16 \times 16 = 256$, the GPU takes two steps to finish the reordering process. Firstly, block reads 256 keys and sorts locally. The system then writes them back to global memory. The corresponding global index address of value n is calculated by:

$$c_i = i - lo^n + go_b^n \tag{3.3}$$

where i is the local start index in the block, lo^n is the local offset, and go_b^n is the global offset of the value n processed by block index b. The above stage will be repeated for d chunks (c_1, c_2, ..., d_d) from least significant bits to the next significant bits, and so on. In the kernel, the Thrust library [3] $thrust :: sort_by_key$ is used to implement the sort operation as it contains the CUDA code of the radix sort function. At a general level, Thrust is a productivity-oriented library for CUDA, as it is an analog of C++ standard template library.

3.3.3 Pearson's Correlation Coefficient Kernel

As shown in Figure 3.6, Pearson's correlation coefficient kernel takes two sub-matrices and outputs the corresponding value in the buffer of the output matrix.

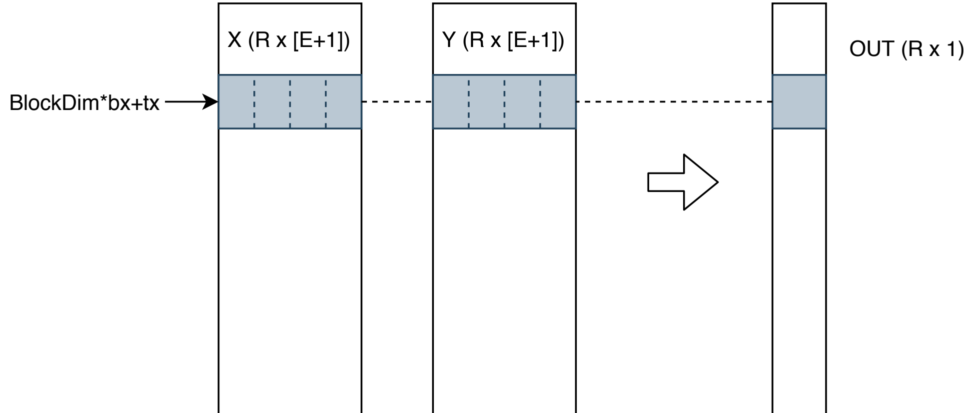

Figure 3.6: The parallel algorithm for R samples, $(E+1)$-dimension vectors, of Pearson's correlation coefficient calculation. Each thread computes one row of **OUT**, which takes one pair of the same row of **X** and **Y**.

As applied here, the Pearson's correlation coefficient measures the linear correlation between two sequences x and y. In order to achieve the parallelism on GPU, R realizations in CCM of predicted values x and corresponding y, each of them is in $E+1$ dimension, form the matrix-like input X and Y, with the size assuming to be $R \times (E+1)$. A single thread on CUDA works on the input (X_i and Y_i, which are the same row) to produce the corresponding ρ in the output matrix OUT. A similar coordinate system to that used in the parallel pairwise distances calculation can be applied in this algorithm. In CUDA, the thread i is identified by $BlockDim \times bx + tx$. It only processes the corresponding sequences (row) X_i and Y_i and generates the ρ_i in the OUT matrix at position i.

3.4 Experiments

This section evaluates the performance of the GPU-based parallel algorithms in a set of experiments, and baseline C++ implemented algorithms are executed on the CPU host for comparison.

3.4.1 Setup

In the first experiment, a performance comparison between GPU implementations and CPU implementations is conducted with the following hardware and software setup. The hardware specification of the desktop machine consists of a Dell Inspiron 3.60GHz with 4 Intel i3-8100 CPUs and 4GB of DDR3 memory. The graphics card used on this machine is an NVIDIA GeForce GT 710 with 192 cores (12 SMs × 16 SPs or cores) and 2GB of DDR3 memory, which supports a PCI Express 2.0 port. The desktop operating system is

Ubuntu 16.04, and the C/C++ and CUDA code is compiled by CUDA Toolkit 10. The computation time for GPU CUDA includes the data transfer between the GPU card and CPU RAM but does not include the time spent on random data generation. The CPU code is single-threaded and is drawn from the rEDM public library on Github.

The second experiment conducts a performance comparison between different GPUs, evaluating the performance gain by upgrading the GPU hardware. There are two NVIDIA GPU configurations considered, corresponding to GeForce GT 710 and GeForce GT 730. The first one contains 192 cores, while the second carries twice that number: 384 cores.

3.4.2 Results

Performance comparison of three kernel functions

This chapter investigates the results of the GPU optimization of CCM in light of the three implemented CUDA kernel functions: The pairwise distance matrix calculation, radix sorting of the distance matrix and Pearson's correlation coefficients. The first two kernel functions belong to the kNN search BF method. The third kernel function takes the predicted results and observations from multiple realizations R of given library size L to compute the correlations ρ. As noted in earlier chapters, causality can be inferred examining how the ensembles of ρ change with rising L.

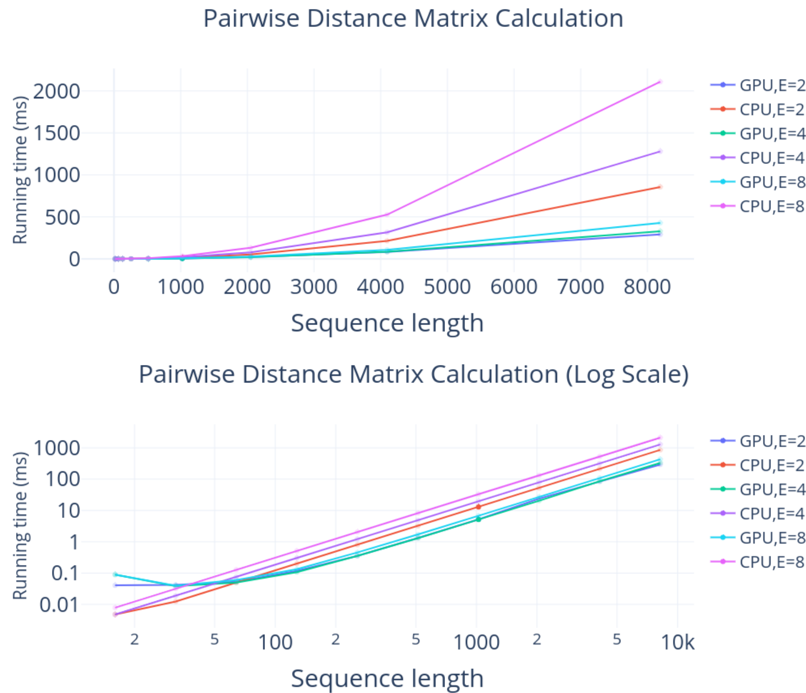

Figure 3.7: The performance comparison of the pairwise Euclidean distances under different dimension E and the input sequence length T (Notably, then input matrix is $T \times E$).

Figure 3.7 shows the computation time comparison of the pairwise Euclidean distance under different

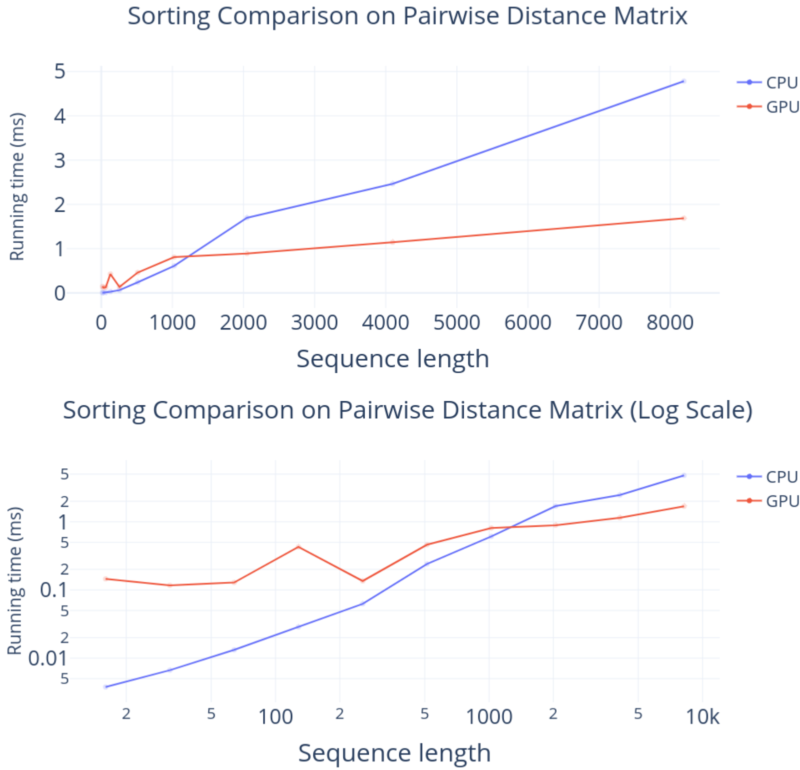

Figure 3.8: The performance comparison of the radix sort operation in kNN search when applying the distances calculated from previous step. Sort can be performed either on local library size L or global time series T.

values of parameters E for both CPU and GPU implementation. For this experiment, the number of points varies from 16 to 8192, which is uniformly generated in E dimensional space ($E = 2, 4, 8$). The computation times for CPU code are plotted for comparison with those using GPU code at a specific problem size. Given input sequence T with small length, communication, and data transfer become the main bottleneck. As such, the advantage of manycore GPU computation power can only be demonstrated when most of the work is spent on computation, instead of copying data back and forth. The embedding dimension E has a profound impact on the calculation of the pairwise distance for CPU code. However, when executing kernel functions on the GPU, dimension E becomes less critical, and the computation time does not in the marked fashion seen in the CPU code. Also, we have observed that significant speedup can be achieved by using GPUs when the sequences T and dimensions E are particularly large. The performance of the distance matrix calculation can achieve approximately 3x performance speedup over CPU-based implementation using GeForce GT 710 with $T = O(10^3)$. In addition, the intersection between the GPU and CPU curves on Figure 3.7 demonstrate that the running time can be accelerated by GPU even when the input matrix size is small.

As shown in Figure 3.8, the radix sort performance comparison between CPU and GPU reveals the same broad pattern as for the pairwise distance calculation. However, the intersection between two curves shows that the GPU outperforms the CPU under a certain length of the input data, which demonstrates that the

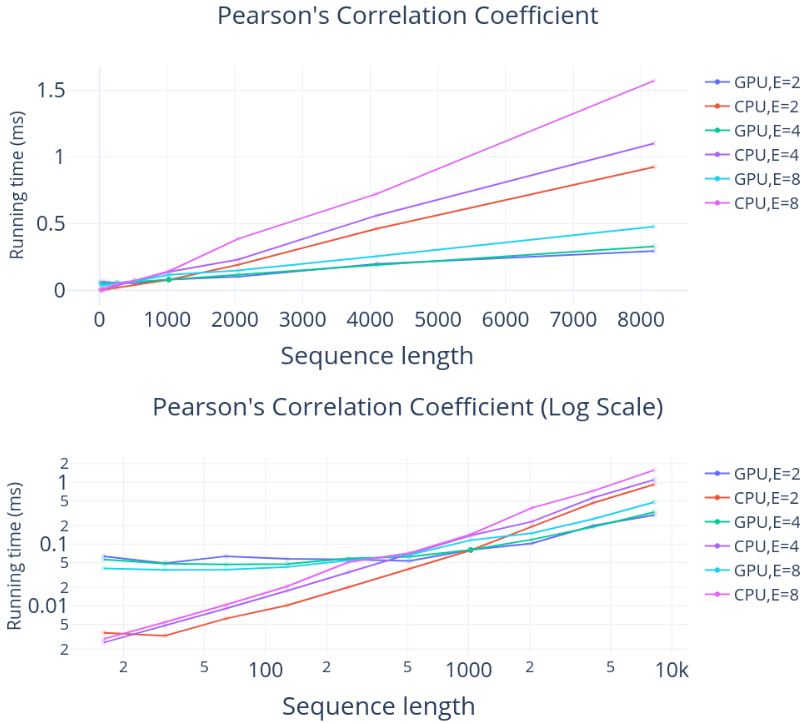

Figure 3.9: The performance comparison of Pearson's correlation coefficient computation for two $R \times [E+1]$ matrices given a certain library size L.

performance advantage of GPU relative to CPU is not as pronounced as for the pairwise distance kernel. The most significant difference is the complexity of the work. Radix sorting requires cutting multiple chunks, and for each chunk, three steps are involved. By contrast, the pairwise distance calculation only applies a Euclidean formula for each entry. The difference in the intersection point reflecting the work complexity serves as a reminder of the fact that GPUs are offered particularly pronounced benefits when accelerating computations involving massive data being processed with simple operations.

Moreover, the relative computational powers of CPUs and GPUs can influence the intersection point as well. For more advanced GPU architecture (Maxwell or later), GPU can perform better than CPU even when sequence length is small. (This part is discussed in the GPUs performance comparison experiment). The cost of data transfer on PCI-Express between RAM and device memory gradually becomes less important, and the benefit of sharing work by many GPU cores dominates the whole computation.

Pearson's correlation coefficient implemented by the CUDA kernel demonstrates a slight improvement when comparing CPU code and GPU code in Figure 3.9. The lines of GPU and CPU intersect at a high sequence length (sample size) $R = O(10^3)$. However, the realization R frequently smaller than 1000. Unless the dimension of inputs E is large, or more advanced GPU architectures are adopted, the kernel of the Pearson's correlation coefficient is unable to bring any strong performance improvement in CCM.

Performance comparison of different GPUs

The performance depends on many factors. Although GeForce GT 730 contains approximately twice the number of cores of the GT 710, the results in Figure 3.10 do not show proportional speedup in execution time. For simplicity of exposition, we can characterize the GPU computations as taking place in three stages. Memory allocation and data transfer is the first block. As such, the bandwidth and device memory plays an essential role in this stage. In comparison, GT 710 and GT 730 are similar in bandwidth and device memory, reflecting the fact that they were released in the same year. The second stage consists of on-core computation. In this stage, the most direct factor is the count of GPU cores; the more cores an architecture contains, the more computational power it possesses. As mentioned above, GeForce GT 730 contains twice as many cores as GT 710. At this stage and for this workload, GT 730 should, in theory, have roughly twice the computational capacity as the GT 710.

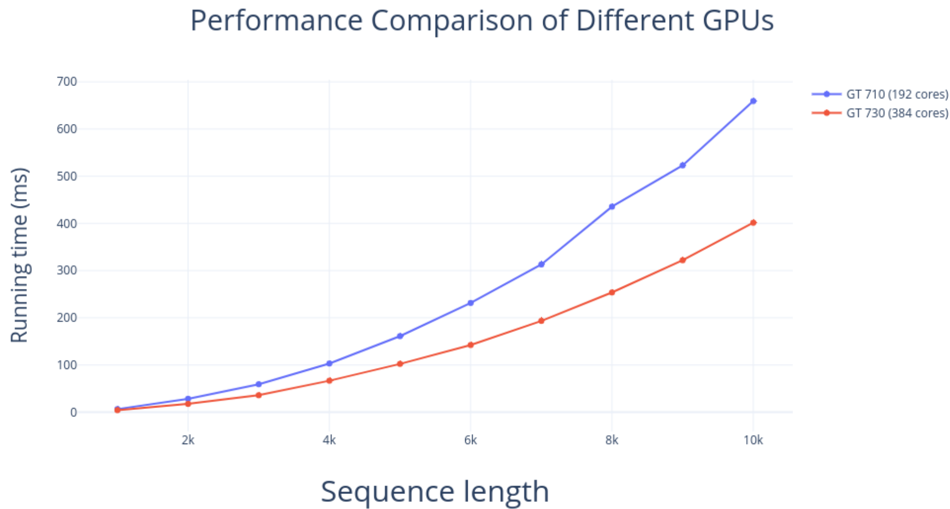

Figure 3.10: Among three kernel functions, the pairwise distance calculation kernel takes much longer time, which can be utilized as the benchmark to compare the performance of different GPUs.

However, Figure 3.10 emphasizes that the number of cores is not the only measurement of performance, but there are many other factors, such as bandwidth and device memory, which determine the overall performance. Nonetheless, the result suggests that upgrading GPU hardware configuration is likely to make a substantial difference in some CCM scenarios.

3.5 Conclusion

GPU-enabled acceleration of CCM suffers from some architectural disadvantages. Firstly, GPU computation can be slightly different from CPU results as the floating point computations are not assured of producing the same results across any set of processor architectures. Furthermore, the data-parallel feature of GPU directly leads to the different operations or code organization compared to sequential programming paradigm when

implementing similar algorithms. Besides, insufficient memory on GPU is often the primary performance bottleneck, sometimes causing unexpected results. More importantly for this implementation, input time series whose length exceeds a certain threshold can lead to a system error when the machine allocates and transfers the data into the GPU device. Obviously, the last kernel function is useless using current GT 710 as R generally smaller than 1000, which is slower than the CPU-based code from experiment results. Finally, GPU acceleration can only be fruitfully applied for certain problems amenable to characterization with SIMD algorithms. The operations related to recursion or complex data structures show substantial difficulty both in parallelization and implementation.

Although GPUs have been proven to implement a smaller range of data-parallel algorithms efficiently – such as matrix-based or vector-based algebra problems – they still remain useful over a much broader class of problems and provide practical methods to accelerate some specific functions for CCM implementation. The results of the experiments shown above indicate that the GPU can achieve significant performance acceleration under some conditions, such as for larger sequences of input data and for certain GPU hardware configurations. The experiments demonstrate three possible ways to accelerate key algorithm used in the CCM application. In most cases, the GPU will outperform the CPU at a particular input data scale, and it is important to identify at what scale we should bring GPU acceleration into the application. The underlying reasons for many of the tradeoffs and limitations can be found in the architecture of GPUs. GPUs are dramatically fast in terms of theoretical, computational power (FLOPS, Floating Point Operations Per Second). But they are often throttled down by the effective memory bandwidth. Limitations in memory transfers and overhead limit the situations that can benefit from pronounced GPU acceleration. Under current GeForce GT 710 hardware configuration, the CCM acceleration enabled by GPU implementation (only including first two kernel functions) with time series input length T on the order of $n = 10^3$ and when parameters $R = 250$, $\tau = 1$ and $E = 2$ is approximately 1.7x faster than for the single-threaded rEDM version. Such results must be considered in light of the fact that the low quality of GeForce GT 710 product, which only contains 192 cores.

Chapter 4

Exploiting Cluster Parallelism of Convergent Cross Mapping

4.1 Introduction

The growth in distributed processing frameworks in recent decades has been driven in substantial part by the rise in extant dataset size and the computational demands of algorithms. The execution performance can be enhanced with High-Performance Computing frameworks through support by multiple machines featuring interconnected networks. As a large fraction of contemporary process jobs is both data-intensive and computation-intensive, two distributed computing frameworks, Apache Spark and MPI, are frequently adopted to scale algorithms in cluster-based environments.

MPI is a C++ message passing library specification which defines a message passing model for parallel and distributed programming, which was laid out in early 1996 by Gropp [23]. MPI extends from a serial program executed by a single process within a single machine to multiple processes distributed across a cluster of nodes. MPI utilizes the resources of all of those nodes at once by facilitating the communication between them across the network. MPI standard includes several communication primitives such as to send, receive, scatter, gather, etc. In 2002, an MPI implementation, MPICH2 [22], was launched that provides remarkably great performance via minimizing the data transfer latency and corresponding network injection rate, and concurrently maximizing the bandwidth and maintaining a balance of resource utilization. Currently, MPICH2 packages are available in many UNIX distributions and Windows platforms.

On the other hand, MapReduce and its variants have been highly successful in implementing large-scale and data-intensive algorithms and applications. Specifically, the Apache Spark [64] platform can support such variants in a memory-conserving fashion, while preserving the scalability and fault tolerance inherited from MapReduce. In addition, Spark further offers a powerful interface for performing both interactive and batch analyses and a simple, scalable application programming interface. The widespread usage of the Spark framework relies on its functional programming style, highly abstract APIs and support for various distributed storage architectures.

In this chapter, two frameworks are adopted to parallelize CCM in cluster environments. The performance of each is compared in the experiments. The chapter concludes with an analysis and discussion.

4.2 Parallelizing CCM using Hybrid MPI/OpenMP Framework

Large-scale cluster system trends motivate the consideration of hybrid programming models. HPC systems are rapidly increasing in scale in terms of numbers of nodes and numbers of cores per node. These hardware settings encourage the use of shared-memory models, with nodes equipped to exploit fine-grain parallelism to achieve better load balance and memory utilization, and the use of message-passing models among nodes to simplify communication and data partition overhead. Most hybrid programming models exploit coarse-grain parallelism at the task level and fine-grain parallelism at the loop level. The hybrid MPI/OpenMP framework typically follows the same rules to achieve multiple levels of parallelism. To parallelizing CCM, the hybrid MPI and OpenMP parallel programming can support a higher degree of parallelism on the cluster compared to the use of either MPI or OpenMP alone. Still, the implementation challenges of Hybrid MPI/OpenMP programming lie in its flexibility and high customizability. Developers have to handle threads, partitioning data and allocate tasks with primitive message communication protocols directly.

Iterative parallel computation dominates the execution of scientific applications, especially for CCM. Figure 4.1 depicts the iterative hybrid MPI/OpenMP computation scheme, which partitions the computational space for the parameters E and τ into subdomains, with each subdomain being handled by an MPI task. Also, it is notable that there can be different phases involved in hybrid MPI/OpenMP programming. The communication phase (MPI operations) exchanges subdomain boundary data or computation results among tasks. And the computation phases are parallelized with OpenMP constructs following the communication phases.

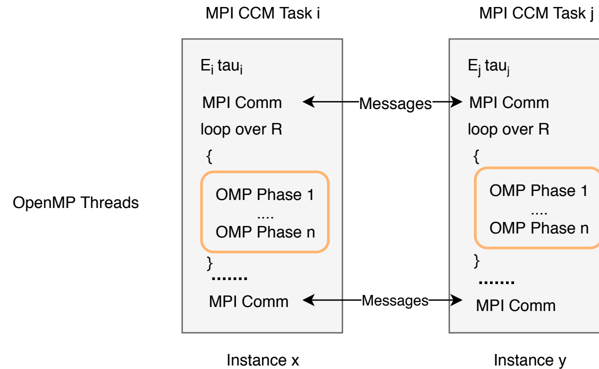

Figure 4.1: Simplified typical hybrid MPI/OpenMP scheme applied on CCM.

To achieve a parallel version with MPI/OpenMP, several operations need to be introduced. The collective communication protocols in MPI allows the program to exchange data across all processes. There are mainly three types of collective communication suited for use by the CCM algorithm:

Broadcast: one process sends a message to every other process. The related MPI API function is **MPI_Bcast**. For example, some common data – such as input time series or distance matrices – should be broadcasted to all processes.

Scatter: A single process (master) partitions the data to send pieces to every other process. The task can execute under different parameter combinations for each process by invoking the scatter operation. The associated function is MPI API is **MPI_Scatter**.

Gather: A single process (master) assembles the data from different processes in a buffer. This operation serves as the function to collect results after computation. The MPI API employed is similar to that for the MPI Scatter operation: **MPI_Gather**.

With these collective operations, MPI can allocate workload to different nodes in the cluster by partitioning the parameter sets and assigning each task to a unique combination of E, τ, and L. When it comes to the computation phase on a single node, some OpenMP operations take responsibility for parallelizing CCM in a fine-grain way. In OpenMP, most of these operations are expressed via **pragmas**, i.e., directives. For example, the **parallel** directive is used to create a group of threads. And work sharing directives (such as **for** and **sections**) are used to distribute units of work among threads in the group. Also, the directives related to the synchronization (**critical** and **barrier**) play an important role in maintaining data consistency in the shared-memory programming model. Because the loop over R realizations is independent, forking a group of threads, where one thread only runs one realization in CCM, is implemented by adding **#pragma omp parallel for** above the iterations of the loop. At the same time, **#pragma omp critical** is used to collect the prediction skills produced by each thread in a serial way.

The hybrid programming model employed here utilizes a combination of MPI and OpenMP. MPI mainly governs the inter-node communication, data partition and task allocation in the cluster. By contrast, OpenMP is responsible for the shared-memory multithreading inside of each node, which represents the second degree of parallelism. Although the mixed techniques cannot address load balancing issues, this scheme for parallelizing CCM is capable of accelerating the whole execution speed by up to a factor of 10 in the cluster. While, some studies [50] demonstrate that a pure MPI implementation can be even faster than an MPI/OpenMP hybrid framework, for CCM, the hybrid model can be a good fit, as MPI can handle the high-level parallelism of the CCM hyperparameter tuning over parameter grid, and OpenMP can handle the low-level parallelism associated with the subsamples in different L. The pure MPI framework has to apply the same message communication protocol inside the node, which will cause unnecessary communication overhead. Also, without OpenMP, parallelizing multiple realizations will become a challenging issue in the implementation.

4.3 Parallelizing CCM Using Apache Spark Framework

Compared to Hybrid MPI/OpenMP framework, it is convenient to implement using the Spark framework without explicitly handling threads. To achieve a parallel version with Spark, two core concepts have to be introduced: the Spark Resilient Distributed Dataset (RDD) [63] and Pipeline. The former is the immutable data structure that can be operated in a distributed manner, which brings significant benefits for concurrently

draw R subsamples of time series to assess Cross-Mapping convergence. As for the Pipeline, it is specified as a sequence of stages, and each stage transforms the original RDD to another RDD accordingly. These stages will run in order, and the input can be transformed as it passes through each stage. In summary, the definition of Pipeline supports an elegant design for a parallel CCM algorithm manipulating RDDs in Spark.

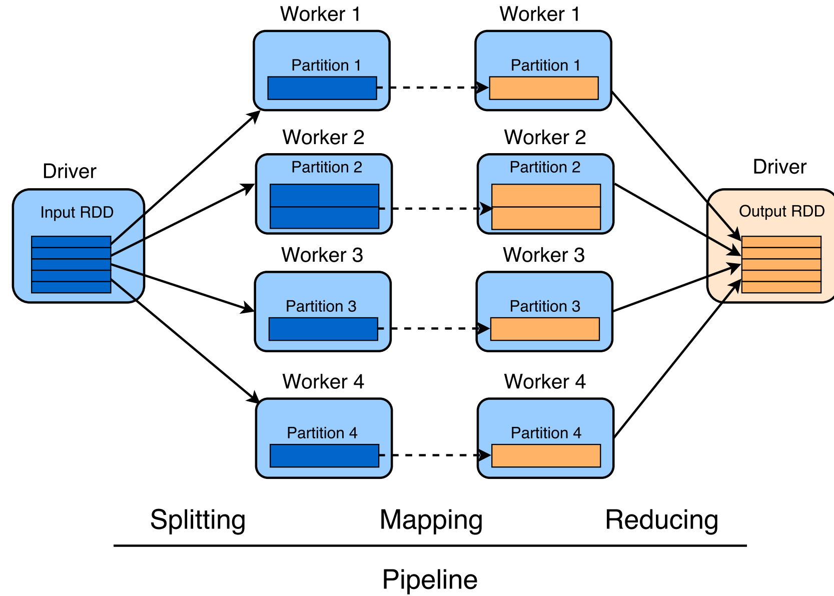

Figure 4.2: An example of the pipeline running distributed on Spark.

CCM Transform Pipeline

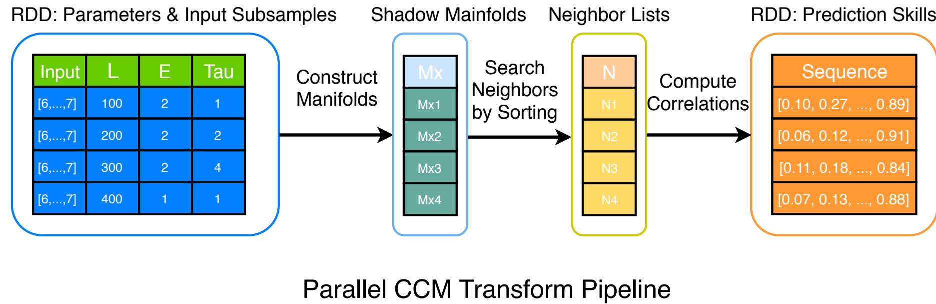

Figure 4.3: A diagram of CCM RDD transformation which takes multiple realizations as input and outputs prediction skills.

Consider applying CCM to test if the variable associated with time series Y is being driven by the variable associated with time series X. In the corresponding transform pipeline, the parallel version of CCM is implemented as several stages to transform the RDD of R random subsamples of the time series to the RDD of prediction skills for a given (τ, E, L) tuple. To start the transformation, an input RDD is created, which includes a pair of subsamples of lengths L of each of the time series, and values for each of two parameters

(τ, E). The output of the CCM transform pipeline is an RDD of sequences of prediction skills. In the whole procedure, Spark operates the complete transformation in parallel without extra coding, as shown in Fig. 4.3.

Distance Indexing Table Pipeline

The CCM transform pipeline above achieves the aim of running CCM concurrently on multiple realizations R. However, there is still considerable potential for further optimization for this Pipeline. Apparently, the most time-consuming part in the CCM computation lies in the $E+1$ nearest neighbor searching for every lagged-coordinate vector (τ) in the shadow manifold. For every point in the input RDD, the CCM transform pipeline computes the distances to all lagged-coordinate embedded vectors of subsamples, sorts them and, finally, takes the top $E+1$ as the nearest neighbors. This process is inefficient because of its repeated sorting and calculation for all R realizations. It is particularly notable that, as the length of subsamples L used for computation increases, the running time will grow superlinearly.

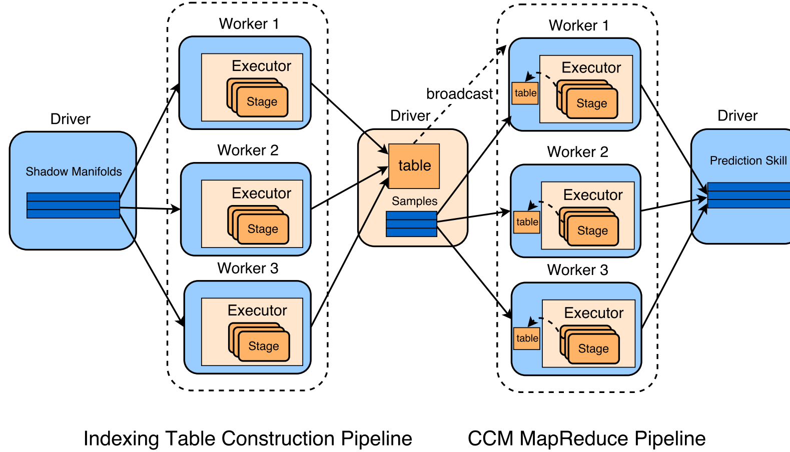

Figure 4.4: An illustration of the dependencies of two pipelines. After the distance indexing table is constructed in parallel, Spark will broadcast it to all nodes. In the next pipeline, the executors can look up in the table and fetch the $E+1$ nearest neighbours quickly.

One way of lowering the computational costs involved is to break down the nearest neighbors searching of CCM transformations into two parts: Construction of a distance indexing table, and nearest neighbors searching based on the constructed table. The first part can be achieved by setting another Pipeline as a preprocessing step before applying the CCM transform pipeline. After building the distance indexing table, Spark can broadcast this table to each worker node on the cluster a single time, rather than sending a copy of it every time they need it, as shown in Figure 4.4. The Pipeline for constructing the distance indexing table will be executed concurrently on the entire input time series, and compiling it also reduces a significant amount of repeated calculation in the CCM transform pipeline. From the experiment results, it is clear that the total computation time decreases in a pronounced fashion. As the library size, L grows, the time spent on searching for the nearest neighbors increases correspondingly, and pre-building the distance indexing table secures increasing benefit. The algorithm details can be found in the time complexity analysis section.

As mentioned in the above, examining prediction skill for differing values of L is essential for quantifying prediction skill in a fashion that reveals whether convergence – the hallmark of a causal connection – occurs. Thus, experimenting with a wide range of L is important in assessing the causality. Considering that two other parameters (E and τ) are typically small values (commonly less than or equal to 10) used in simplex projection, speeding up this algorithm for large counts of distinct values of L is of great importance.

Asynchronous Pipelines

In Spark, the Action operation of the Pipeline triggers the job submission. The driver will send it to all executors, and then each job will be partitioned into many tasks. The next job is unable to be executed until all tasks of this job are completed. The Asynchronous Pipeline permits the execution of multiple jobs on the cluster, which better exploits the computational resources available.

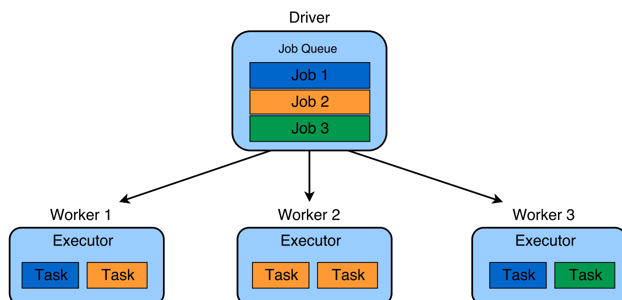

Figure 4.5: A diagram of Spark executing asynchronous pipelines.

After a pipeline is created to run Convergent Cross Mapping, a job is generated in the master node and then submitted to the cluster and partitioned into many tasks running in the executors of worker nodes. The CCM parameter settings are defined in the job submission and are, in general, constant. The next job cannot be generated until the application finishes the current job; the executions of these two jobs are always performed in a synchronous fashion. If we perform two pipelines one after other, they always execute sequentially.

As such, we adopt some asynchronous mechanisms to increase parallelism and execute different pipelines concurrently. FutureAction is one of the means to undertake asynchronous job submission in Spark. It provides a native way for the program to express concurrent pipelines without having to deal with the detail complexity of explicitly setting up multiple threads. A specific pipeline is parameterized explicitly by its own CCM parameter settings to generate prediction skills. In this way, we can achieve running various combinations of the parameters (L, τ, and E) in parallel by executing multiple concurrent pipelines.

4.4 Experiments

The baseline scenario of CCM parameters, as shown in Table 4.1, is set for the comparison in the experiments. The experiments on different clustering framework will be configured and conducted separately. And the

overall comparison among different parallel versions will be presented in the next chapter.

Table 4.1: Baseline setting

Parameter	Value
Time series (T)	4000
R	500
$LSet$	$[500, 1000, 2000]$
$ESet$	$[1, 2, 4]$
$TauSet$	$[1, 2, 4]$

4.4.1 MPI/OpenMP

In the following experiments, the MPI/OpenMP hybrid version of CCM will be tested on a cluster of three HPC machines. The master node contains eight cores while the other two machines have four cores. Unlike the Spark cluster, the master node in an MPI/OpenMP cluster still counts as a computation node. In order to collect the output CSV files, the Network File System (NFS) has been installed on the master node, and a shared directory has been created, to which each node enjoys access.

Performance Comparison

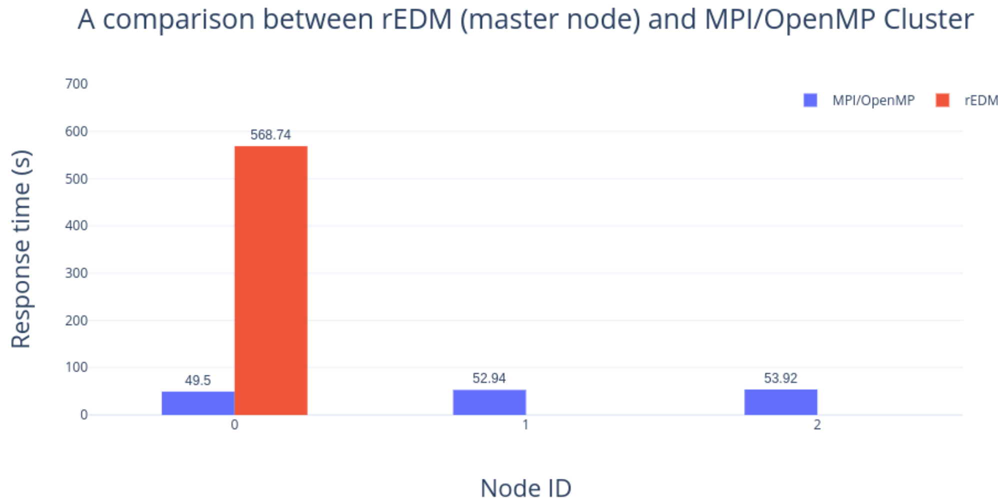

Figure 4.6: The 3-node cluster contains 3 machines, which has been labelled from 0 to 2. Node 0 is the master node in the cluster and the machine which runs rEDM for comparison.

This experiment only compares the performance of MPI/OpenMP implementation with the currently existing public library rEDM R package, where rEDM will be tested on a single machine – the master node with 8 cores. Since rEDM is single-threaded, the number of CPU cores has no impact on its performance.

As shown in Figure 4.6, the results indicates that MPI/OpenMP parallel implementation can achieve an approximately 10.5x speed when compared with rEDM for the baseline scenario on a 3-node cluster [including output CSV files], when we view the maximum response time as the finishing time of the cluster program. Obviously, the parallel version can perform more favorably with a more powerful cluster (vertical scaling) or adding more workers (horizontal scaling).

4.4.2 Apache Spark

In the following experiments, the Spark parallel version of CCM will be run three times on the Google Cloud Platform (GCP) to obtain the average computation time. GCP can change cluster settings (the type or count of workers) more easily, and it configures the Yarn cluster in advance; the only thing remaining is submitting the jar file and starting the job. The cluster setting used here consists of one master node and five worker nodes with four cores CPU and 15 GB Memory.

Overview of Improvements

This experiment compares the performance improvement of different implementations on the baseline scenario. These different versions of CCM in Table 4.2, implemented using Scala Spark, are submitted in Yarn Mode and Local Mode, separately. Yarn Mode – or cluster mode – will exploit all the worker nodes existing in the cluster, while Local Mode only runs applications on a single machine – the master node.

Table 4.2: Implementation Levels

	Implementation Level
Case A1	Single-threaded CCM (no RDD & Pipeline)
Case A2	Synchronous CCM Transform Pipelines
Case A3	Asynchronous CCM Transform Pipelines
Case A4	Synchronous Distance Indexing Table & CCM Transform Pipelines
Case A5	Asynchronous Distance Indexing Table & CCM Transform Pipelines

The results are shown in Fig 4.7. Several conclusions can be drawn from the experimental results for different levels of parallel implementation. Firstly, the single-threaded version of CCM imposes a heavy computational cost, and there is no difference between two modes as they do not utilize the worker nodes in the cluster. Next, asynchronous pipelines can only reduce computation time in Yarn mode. Comparison of the CPU utilization rates indicates that the asynchronous pipelines cannot offer more parallelization when the CPU utilization already reaches full throttle. However, when running with Yarn, the worker nodes still have room to improve utilization rates. Also, as seen from the results, the Spark full parallel version (*Case A5*) offers approximately 1.2% the running time of the single-threaded version. Ultimately, the most significant improvement of marginal computation performance lies in adding the distance indexing table pipeline based

on the CCM Transform pipeline. It reduces the computation time cost by over 80% relative to the baseline. Such marked improvement shows the parallel version of CCM benefits strongly from establishing the distance indexing table globally for the nearest neighbors searching Pipeline.

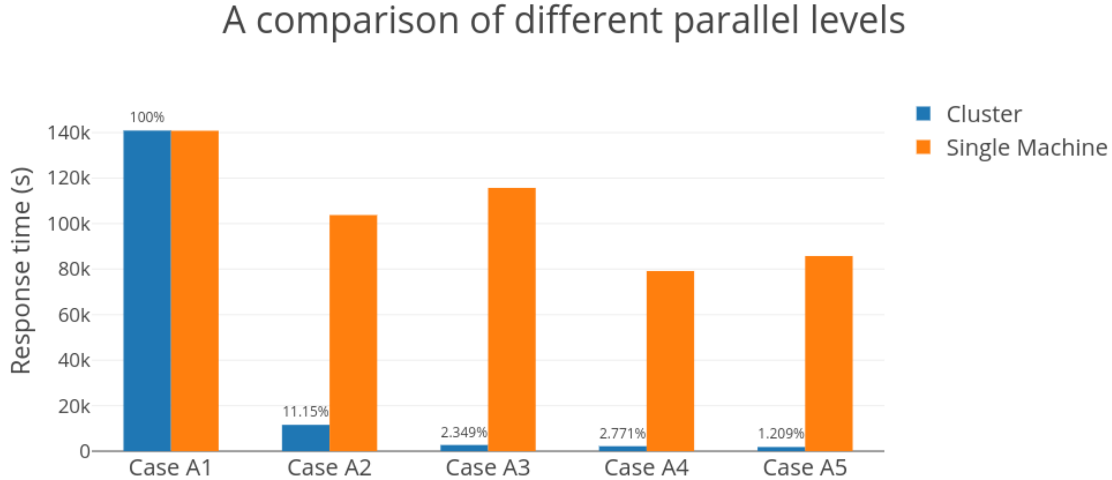

Figure 4.7: Yarn Mode utilizes all worker nodes in the cluster, while Local Mode only runs experiments on the master node. Yarn Mode significantly diminishes the average computation time of the parallel version of CCM with the help of worker nodes.

When compared to what is most likely the most popular existing public CCM implementation – the rEDM R package created by Hao Ye et al. [61] using the lower level language C++ – our Spark parallel implementation (*Case A5*) is approximately 15x faster than rEDM for the baseline scenario on current cluster setup on Google Compute Platform [excluding output CSV files]. Moreover, it is clear that the parallel version will be able to perform more favorably with a more powerful cluster (vertical scaling) or by adding more workers (horizontal scaling).

Elasticity Analysis

Table 4.3: Elasticity Analysis

<table>
<tr><th>Parameter varied</th><th>parameter</th><th>Case B1</th><th>Case B2</th><th>Case B3</th></tr>
<tr><td rowspan="2">$LSet$</td><td>$LSet$</td><td>[500]</td><td>[1000]</td><td>[2000]</td></tr>
<tr><td>others</td><td colspan="3">the same as baseline scenario</td></tr>
<tr><td rowspan="2">$ESet$</td><td>$ESet$</td><td>[1]</td><td>[2]</td><td>[4]</td></tr>
<tr><td>others</td><td colspan="3">the same as baseline scenario</td></tr>
<tr><td rowspan="2">$TauSet$</td><td>$TauSet$</td><td>[1]</td><td>[2]</td><td>[4]</td></tr>
<tr><td>others</td><td colspan="3">the same as baseline scenario</td></tr>
</table>

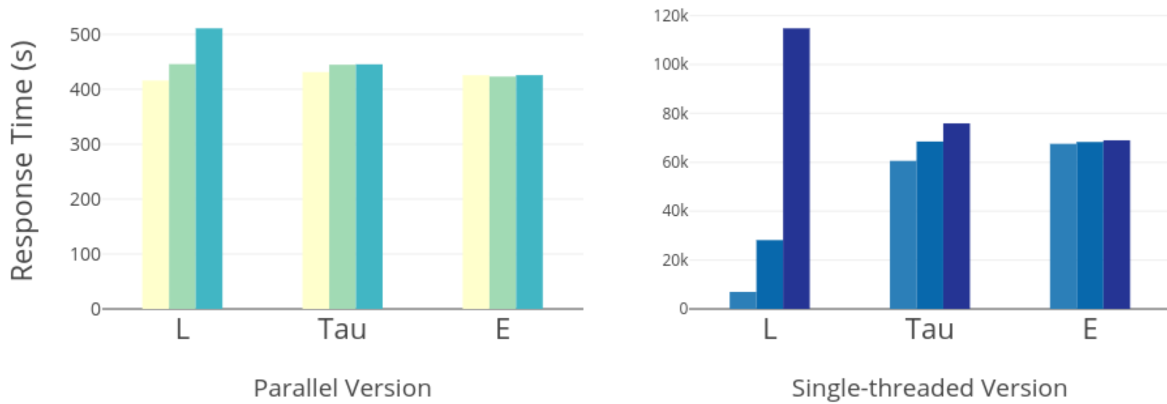

Figure 4.8: The parallel version uses all of the optimization methods with five 4-core workers in the cluster, while the single-threaded version is only executed on the master node without any parallel optimization.

As a grid of parameter settings been looped over for the best results to infer causality, testing the elasticity of running time with respect to a change in a given parameter value is valuable. Two versions of CCM will be tested with the parameter settings as shown in Table 4.3. Specifically, we investigate the performance of the maximally parallel version (*Case A5*), and the single-threaded version in the implementation of *Case A1*, which has not implemented any pipeline.

Intuitively, each of these cases varies only one parameter from the baseline for comparison. When doubling parameter L, the average run time increases 4.06x using the Spark single-threaded version, and 1.11x using the Spark parallel version. Similarly, doubling parameter τ and E has almost no impact on running time in the parallel version. However, doubling τ indeed increases the running time to 1.13x in the single-threaded version surprisingly, while doubling E only increases a little bit on the total running time. Actually, both E and τ are supposed to influence the reconstruction process of shadow manifold M_x, which is a small part of overall computation time. However, the shape of M_x space may impact the nearest neighbor searching. Also, the inconsistency of performance on JVM could be another possible underlying reason.

Given the experiment results, library size L imposes a relatively significant impact on the computation time for the single-threaded version. As depicted directly in Fig 4.9, the computation time grows superlinearly for the single-thread version with rising library size L. But in the parallel versions, especially with the distance indexing table, the computation time grows slowly and linearly, as the time complexity of nearest neighbors searching is $O(n)$. Also, parallel implementation minimizes growth rates of execution time on E and τ. In summary, the values of these parameters, especially for L, do influence execution time for both the single-threaded and parallel versions. However, with current optimization of the parallel methods, the impact of growth in parameter values shrinks, which make testing relatively large parameters for causality assessment

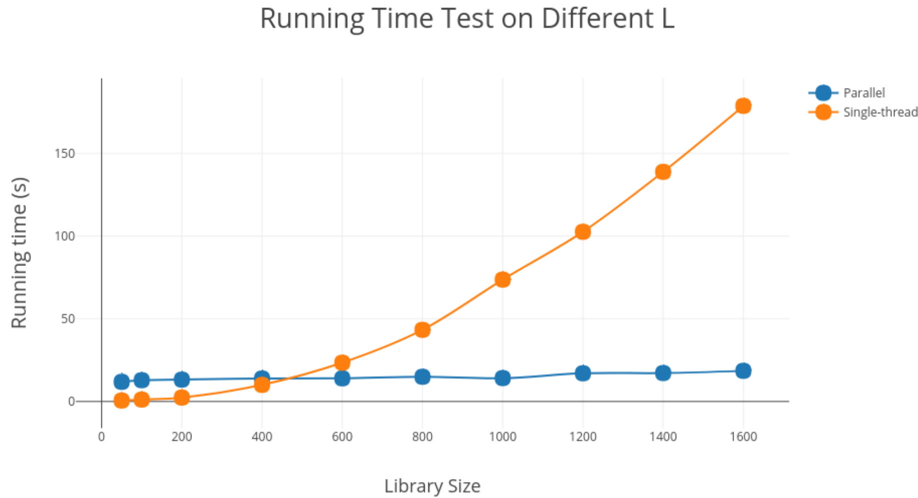

Figure 4.9: The single-threaded version without using Spark consumes little time to finish when library size L is small. However, with increasing L, the time to search for nearest neighbors in a larger shadow manifold and the time to complete grows superlinearly.

a reality.

Vertical and Horizontal Scalability

The capacity to submit a job in a cloud cluster platform like GCP paves the way for researchers to analyze scaling related performance issues. GCP provides simple APIs to add or delete nodes in the cluster, or even to change the number of CPU cores per node. Scaling can work either vertically or horizontally. In vertical scaling, more components (like CPU, RAM) are added to one node. This will eventually reach a limit and does not ensure fault tolerance. By contrast, in horizontal scaling, more relatively weak nodes are added to a cluster. There is a master node which controls how tasks are split across the worker nodes, which leads to fault tolerance.

To fully comprehend the influence of similar changes on the context of computational clusters, we conducted this experiment with respect to 5 different cluster configurations, as depicted in Table 4.4.

For this experiment, the baseline scenario is tested in all cases in Table 4.4 using the aforementioned parallel methods. All of the clusters are set up on Google Dataproc running Apache Spark and use Yarn as the resource manager. Yarn [56] is a software rewrite that decouples MapReduce's resource management and scheduling capabilities from the data processing component, enabling Hadoop to support more varied processing approaches and a broader array of applications. It provides a robust and convenient means of increasing the scalability of Spark programs. For each case, there is one master node and multiple worker nodes, with the same machine types within the management of Yarn; each machine has a certain amount of cores and memory, where a 'core' is implemented as a single hardware hyper-thread on an Intel Skylake Xeon 2.0 GHz processor.

Table 4.4: Cluster Configurations

	Worker Nodes	Cores per Machine	Memory per Machine
Case C1	5	4	15 GB
Case C2	4	4	15 GB
Case C3	3	4	15 GB
Case C4	2	4	15 GB
Case C5	2	6	22.5 GB
Case C6	2	8	30 GB
Case C7	2	10	37.5 GB

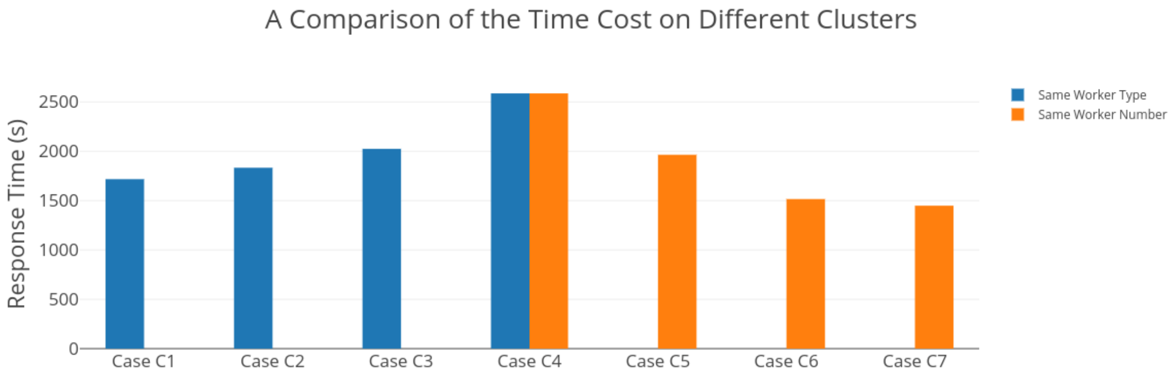

Figure 4.10: The average running time can be reduced when more computational resources are available in the cluster. Under the same total computational resources, the computation time is larger with more inferior worker nodes given the rising time cost of networking I/O.

We take Case C1 as the baseline cluster configuration setup. Cases C2, C3, C4 change the count of worker nodes, and Cases C5, C6, C7 change the number of cores in each worker node so that we can compare the efficiency with the availability of different proportional computational resources in the cluster. By contrast, Cases C1 and C7, Cases C2 and C6, Cases C3 and C5 utilize the same total count of CPU cores within each pair, but each item within the pair exhibits a different count of worker nodes. Decreasing the number of workers, which limits the extent to which computation can be distributed across machines, also reduces the amount of network I/O required.

The results of these configuration cases are demonstrated and compared in Fig 4.10. The computation time drops in the expected fashion with growing levels of computational resources – namely, CPU cores – are added into the cluster, but the marginal utility of worker nodes in the cluster is decreasing. From Case C1 to Case C4, by removing one worker node, the computation time does not increase proportionally; the increased percentages are 6.72%,10.45%, and 27.76%, respectively. The results make sense, as it takes time to schedule and distribute the jobs across these worker nodes through the network. With more worker nodes

added in the cluster, the time required for scheduling and distributing becomes the bottleneck for the cluster. A similar situation obtains when maintaining the same count of workers invariant, but changing the count of cores per worker (and the total count of cores). This point demonstrates that the total computation time will not always be reduced as more computation resources become available.

Compared to these cases, the network I/O cost should not be neglected when considering the same computation resources. The computational capacity of worker nodes matters if we have the same computation resources. However, the time difference is not enormous, and similar efficiencies can be achieved with more inferior worker nodes.

4.4.3 Discussion

The MPI/OpenMP framework comes from the low-level language C++, which only supports the communication protocols for parallel computing. It requires explicitly handling the assignment of tasks and threads. Moreover, the MPI standard does not currently support fault tolerance and automatic garbage collection. The headache of developing the MPI/OpenMP parallel version of CCM accurately and robustly becomes the main challenge issue. However, MPI is currently the dominant model used in high-performance computing among parallel programs running on distributed memory systems, while OpenMP is becoming the standard for shared memory parallel programming account of its high performance. The combination of two standards has the ability to secure high execution speed when compared to other prominent frameworks.

By comparison, the Spark framework provides relatively convenient APIs to exploit parallelism in algorithms such as CCM without handling threads or processes. This work conducted experiments demonstrating the performance benefits of exploiting the parallelism in the CCM algorithm using Spark and comparing the performance difference for the different scaling methods. The scalability of Spark offers considerable benefits in accelerating the execution with the support of clusters, allowing for a significant reduction in running time when adding more worker nodes into the cluster. Of critical importance for the robust application of Spark, these performance gains make this algorithm a valuable modeling tool to assess causality with confidence in an abbreviated time. Such gains are particularly important in the context of high-velocity datasets involving human behavior and exposures, such as are commonly collected in human social and sociotechnical systems. While it demonstrated the potential for marked speedups, this work suffers from some pronounced limitations. Construction of the distance indexing table trades off higher space consumption for savings in computation time; for large shadow manifolds from a large value of L, the indexing table can require large quantities of system memory. However, as previous study [41] shows, CCM can produce reliable results when input time-series length is on the order of $n = 10^3$, for which the required memory space lies well within the range of what most current hardware can offer.

4.5 Conclusion

Results obtained from experiments show hybrid MPI/OpenMP framework indeed outperform Spark framework by more than one degree of magnitude in terms of execution speed. More importantly, hybrid MPI/OpenMP can offer more consistent performance as Yarn will handle the scheduling service at run time. CCM algorithm is not a good fit for the MapReduce model in Spark. The intensive computation mainly comes from the complexity of searching nearest neighbors rather than from the scale of the time series. In addition, when required to output massive CSV files, Spark becomes dramatically slow in terms of gathering data and disk I/O operations. The experiments on GCP only contains the CCM computation part, which is quite efficient in the cluster. The results show better performance on MPI/OpenMP environment than Spark mainly because of the following reasons.

Firstly, in MPI framework, the freedom of the programmer to choose the memory requirements, in terms of the mutable data structures, supports better performance. By contrast, in Spark, the memory allocation and task scheduling are purely under the control of the Spark on Yarn processing framework, and intervention is impossible. Although it reduces coding complexity by automatically making decisions, this inflexibility offers lower performance advantages. Secondly, C++ is relatively lower-level programming language than Scala, which means it secures additional economies by being closer to the hardware – using native code rather than within the JVM. However, MPI lacks a common runtime for large data processing environments. Hence, bridging the gap between big data processing and HPC by incorporating MPI with the integrated I/O management and fault tolerance is one possible motivating scientific study. In the cluster implementation of parallel CCM, the challenges come from the complexity of the algorithm, rather than from the input time series scale. From these experiment results, the MPI framework exhibits comparable advantage to Spark framework in parallelizing such computation-intensive algorithm.

Chapter 5

Exploiting Cluster Parallelism with GPU Acceleration of Convergent Cross Mapping

5.1 Introduction

As introduced in previous chapters, GPUs have been leveraged as accelerators in speeding up complex numerical workloads, in part due to the density of the cores and power efficiency. While formidable parallelization tools for certain needs, the performance is bounded by the limited memory bandwidth, which cannot process large sets of data at once. On the other hand, CPU-based cluster computing frameworks such as MPI and Spark can partition and distribute large sets of data and tasks in parallel. Even given such partitioning, complex algorithms applied to the data unit within a single node can still consume a large number of CPU cycles. As a result, the combination of GPU and CPU-based cluster computing frameworks offers particularly attractive prospects for resolving both big data and intensive computation problems.

Recently, large heterogeneous GPU/CPU clusters have gained increasing popularity in the scientific computing community, as applying analytic algorithms on big data sets requires tremendous computational capacities. With the extensive usage of machine learning and – especially – deep learning algorithms, GPU acceleration in the cluster can significantly boost the performance with limited hardware modification. For instance, Jacobsen et al. [29] implemented mixed MPI-CUDA application in 2010 by transferring data over MPI and computing on the GPUs. Li et al. [35] contributed to the HeteroSpark framework in 2015 by integrating GPU acceleration into the current Spark framework to leverage data parallelism and gain promising benefits. Work such as [5], [60], [62] did similar experiments for large image/video datasets on GPU-enabled clusters by applying different frameworks. All of them achieve multiple levels of parallelism by assigning a less computationally intensive part to CPUs (coarse-grained parallelism), and more intensive parts to GPUs (fine-grained parallelism).

However, parallel programming using hybrid GPUs and cluster computing frameworks still remains a challenging problem, as a large number of issues have to be resolved to manage system complexity. To develop such applications or algorithms, which can run both on GPUs and multi-core CPU clusters, efforts for efficient distribution of workloads should be undertaken not only across cluster nodes but also among multiple CPUs and GPUs. Furthermore, the relative performance of operations on GPUs and CPUs needs to

be compared and incorporated into scheduling decisions. On top of these challenges, data copy and exchange principles have to be redesigned to minimize overhead for the communication stage. These issues frequently lead to underutilization of the resources in hybrid GPU/CPU clusters and thus, require extra coding and algorithm redesign to address them in order to release the power of hybrid parallel programming. Overall, taking advantages of hybrid platforms is challenging but often worthwhile.

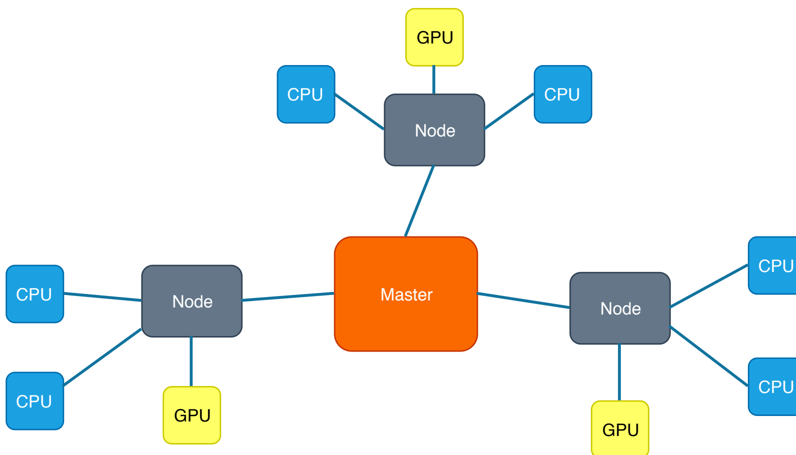

Figure 5.1: An overview of the GPU-enabled cluster.

In this chapter, a new parallelization strategy for CCM is proposed, which can execute on hybrid GPU/CPU cluster systems. The related optimizations include efficient allocation of workloads across multiple nodes as well as coordinated use of GPUs and CPUs on a computing node. Applying these optimizations makes it possible to process the CCM algorithm in an efficient way. A comparison between performance using a CPU only cluster framework and a hybrid CPU/GPU cluster framework is presented in the experiments section to demonstrate the overall improvements brought by GPUs.

5.2 Integrating GPU accelerators into Spark

5.2.1 Methodology

Apache Spark is a robust distributed computing and big data processing framework, whose tasks are only traditionally performed on CPU. For some problems, a low degree of parallelism and power inefficiency may restrict cluster performance and scalability. Heterogeneous accelerators, such as FPGAs, GPUs, and MICs exhibit more efficient performance advantages. However, these heterogeneous accelerators can only be applied using certain languages and compilers, particularly for GPU programming. Although NVIDIA GPUs power millions of machines, workstations and supercomputers around the world, and accelerate computationally-intensive tasks for many researchers and developers, NVIDIA CUDA doesn't offer strong support for either Scala language or Apache Spark framework, in part because they are both written using different technology stacks. The most feasible method to integrate GPU accelerators within the existing Spark framework is

through utilizing the **pipe** method for RDDs, which can resolve the gaps between GPU programming and the Spark framework. ScalaNLP [24], the most popular machine learning and numerical computing libraries using Scala, employs this method to leverage GPU CUDA computation power within the Spark framework. In addition, Tensorflow, a popular deep learning framework, adopts the **pipe** operator to deploy a neural network pipeline running with GPU accelerators. These implementations pave the way for us to investigate GPU-enabled Spark programming.

The implementation of the **pipe** method in Spark will invoke the system call ($Runtime.getRuntime.exec()$) to execute the external process/program in the system. Each process has three streams associated with it: *InputStream*, *OutputStream* and *ErrorStream*. The **pipe** operator utilizes exactly these streams to communicate with the external process. More specifically, it will redirect the data in Resilient Distributed Dataset (RDD) as the InputStream to the external program. Afterward, any output generated by the external process/program will be redirected back to Spark context via OutputStream and saved as records in a separate output RDD. The external process/program can be implemented in any language, but it has to meet the condition that the program should be compiled and put in a shared directory accessible by each node (the executable file path is required to be passed through pipe operator). In order to use the **pipe** operator correctly in a distributed system, the Network File System (NFS), which offers users on machine access to files over a computer network much as local storage is accessed, should be configured in the Yarn cluster beforehand.

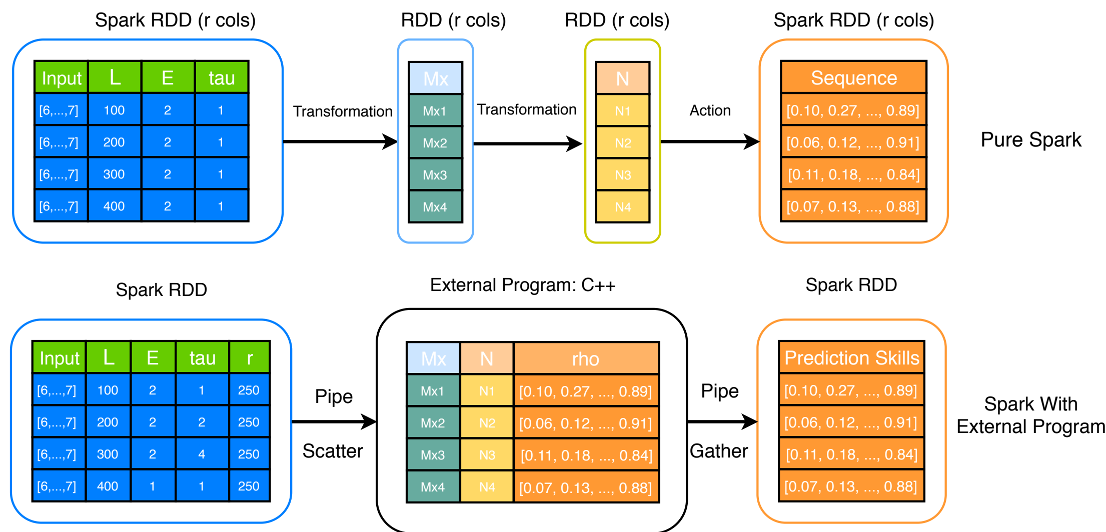

Figure 5.2: A comparison of pure Apache Spark and Apache Spark with external program implementation.

As shown in Figure 5.2, the **pipe** operator allows the developer to process the data in an RDD with the external programs. As OpenMP, CUDA and Thrust [3] techniques would be employed in the development of the CCM application to achieve parallelism, the **pipe** operator becomes the primary tool to achieve heterogeneous computing on the Spark platform. In the early stages of developing the parallel Spark version

of CCM application, the different combination of parameters in CCM was populated into one Spark RDD. Then the driver of the program submitted the **pipe** operator to the yarn cluster, and the scheduler assigned different tasks to the executors in the corresponding worker machines. In each task, C++ with OpenMP is used to parallelize R subsamples of prediction skill calculation based on the RDD column R.

By contrast, the pure Spark implementation generated multiple RDDs with R columns. The pure Spark would then automatically parallelize each sample over the mapping function, or transformation, stage. The execution of multiple RDDs (containing different combinations of parameters E, L and τ) are mainly conducted sequentially or concurrently (as has been introduced in the previous chapter when we use *FutureAction*). Overall, the Spark works as a distributor for different tasks among all nodes in the cluster, where each task has a unique combination of CCM parameters. The task is handled by the C++ OpenMP external program.

5.2.2 Performance issues

However, the running time of the integrated Spark CCM with **pipe** operator does not decline inversely with the number of workers added in the cluster. The poor performance not only stems from the network communication overhead (scatter and gather operations) but also reflects unbalanced task execution times [59]. Adopting **pipe** operator can directly cause Yarn resource manager malfunction as scheduling becomes difficult when invoking the system call. The bottleneck of data partitioning and network communication is native to message-passing systems, occur as the significant overheads for using this model with cluster computing. By contrast, for CCM, the other bottleneck – unbalanced task execution time – is mainly caused by heterogeneity in the L parameter (L specifies the library size for the input time series in a realization, where that library is used for manifold reconstruction). When it comes to the stage of generating shadow manifolds, for larger L, the greater the computational item required to search for the top k nearest neighbors (the kNN search problem). As explained in the previous chapter, for each of the R realizations, the task is supposed to sort the lagged points based on Euclidean distance in the embedded space, with execution time that varies in marked ways for different values L. Such heterogeneity will result in poor load balancing problems, which lead to system underutilization. This implication is evident in the results shown in Figure 5.3. The CPU utilization metrics are collected using Grafana [4]. After the job is submitted, of all tree workers in the cluster, node 3 exhibits a long idle time after finishing the task with small L, while node 1 is still running at full capacity. The idle time in node 3 directly leads to inefficiency and poor performance in cluster computing. The color in the heatmap represents the degree of CPU utilization during the 10-second window for each worker. The current integrated framework fails to exploit the computation capacity of each node fully. Some jobs have already been completed, while other jobs assigned heavier workload are still in execution. Inappropriate job scheduling leads to waste of computation resources.

The bottleneck identified above can be addressed in two primary ways considered here. The first strategy lies in customizing the scheduling policies of the Yarn resource manager in Hadoop, which will assign the tasks based on the value of L in the RDD. In this way, the workload can be distributed more uniformly among

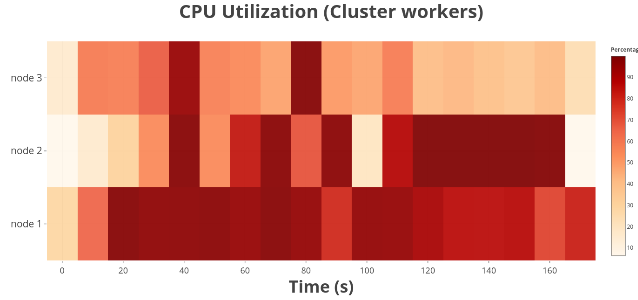

Figure 5.3: CPU utilization plot in the cluster for three workers.

all the workers. However, modifying the scheduling policies at the bottom level of the Yarn implementation is challenging and time-consuming. A second feasible strategy is to optimize the CCM algorithm by eliminating the association between the execution time and parameter L. The second strategy is based on the similar idea introduced in the previous chapter: breaking down the nearest neighbor searching into two parts: construction of a global distance-sorted matrix before undertaking the R realizations, and querying that table when computing the nearest neighbors for those realizations. As this strategy only optimizes at the application algorithm level and offers the possibility of introducing GPU accelerators, the next section will focus on its implementation details.

5.2.3 Integrating GPU accelerators

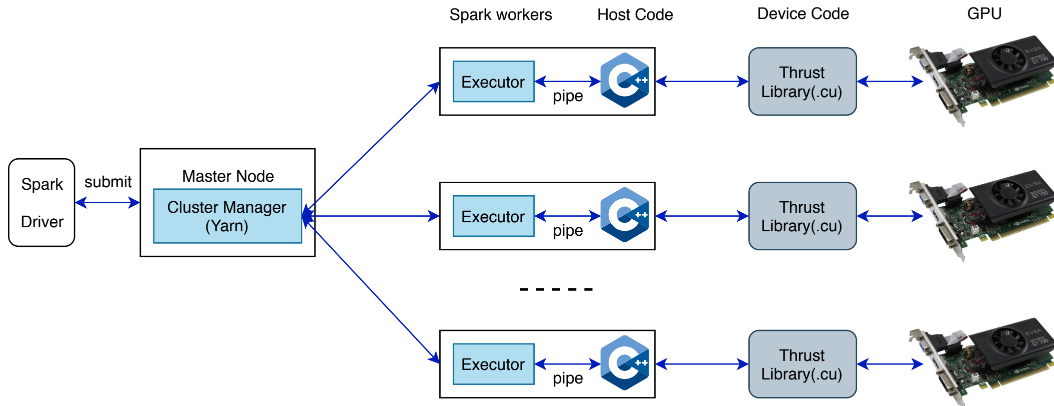

Figure 5.4: Overall methodology of integrating GPU with Spark framework.

As explained above, the first part of this strategy (pairwise distance matrix calculation and row-based radix sorting on the calculated matrix) is treated as a preprocessing step and can be accelerated through GPUs. The second part can be stimulated using OpenMP over multithreading for R realizations. In this way, the overhead of up-front matrix construction trades off with the query efficiency for each realization. The overall methodology of this integrated implementation is depicted in Figure 5.4.

Within this implementation, Apache Spark invokes a C++ application through **pipe** operator by passing

a JSON-format string (encoded from the data in RDD) as the input parameters of the C++ main function. In the external C++ program, the time series and related CCM parameters E, τ, R can be correspondingly decoded based on the JSON format. The output generated by the external application will then be transferred back in the Spark master node. In this process, Spark serves as the data partitioning and task scheduling tool, while the C++ application with CUDA implements the main body of the CCM algorithm. After accelerating C++ using GPUs in the integrated implementation, an experiment was conducted to investigate the performance gain associated with introducing the preprocessing step. The maximum running time determines the response time for the entire application. As a result, the distribution of running time for different tasks becomes smaller and more uniform and is far less strongly affected by the parameter L.

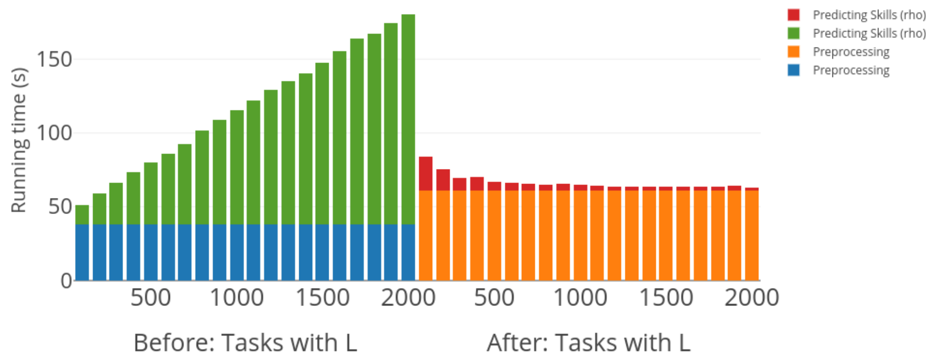

Figure 5.5: Bottleneck elimination and performance comparison after introducing GPU into external application.

As shown in Figure 5.5, after optimizing the kNN search strategy using GPUs in a C++ multithreading stage, the time spent on the process of computing with the reconstructed shadow manifolds (constructing the global pairwise distance matrix and sorting based on rows) may increase, but the time for querying has dramatically declined. Meanwhile, the running time of a task has become less dependent on the parameter L, which makes the distribution of task execution time more uniform, thereby reducing distributed computing bottlenecks.

5.3 Integrating GPU accelerators into MPI

5.3.1 Methodology

Use of regular MPI with CUDA embedded programming procedure is typically undertaken by compiling C extension files and CUDA extension files separately, and then integrating those files into the target executable application. In this process, the MPI paradigm only covers the communication among different processes while, the CUDA implementation focuses on leveraging the computational workload to the GPU device for

acceleration. As mentioned earlier, MPI only offers the basic communication standard APIs, without any fault-tolerant check. The implementation challenge dramatically increases, especially after adding CUDA kernel functions to accelerate part of computation within a single process, and overall complexity of regular MPI with CUDA embedded remains the biggest factor in causing potential errors when dynamical allocating or releasing memory on host and device. The need to manage the complexity of the distributed memory model in message-passing systems and heterogeneous memory model inside each node imposes a pronounced risk for the robustness of the entire application.

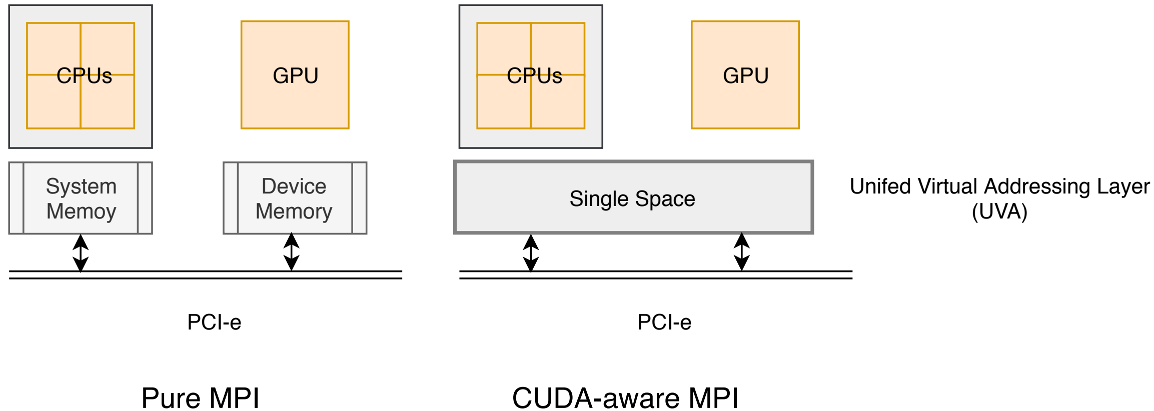

Figure 5.6: The comparison between pure MPI with CUDA and CUDA-Aware MPI architecture. Obviously, CUDA-Aware MPI provides a layer to unify the memory space between host and device.

In 2013, [32] introduces a brand-new strategy, **CUDA-Aware MPI**, to embed GPU CUDA programming into cluster frameworks such as MPI. This strategy unifies the host memory space and the device memory space in MPI and frees the programmer from precise CPU-GPU data movement and CPU/GPU buffer management. This means that the MPI library can send and receive GPU buffers directly and without explicitly allocating memory and copying data between the host and device. As a result, this feature alleviates the overall coding complexity by warping a unified virtual addressing layer behind the MPI communication APIs. Furthermore, it paves the way to solve problems characterized by a large data size to process or requiring long response time on data-parallel tasks. Currently, this feature is then integrated into several popular implementations: the OpenMPI 1.7 series or MVAPICH2 series. Figure 5.6 compares the basic difference between pure MPI with CUDA embedded and CUDA-Aware MPI. Particularly in CUDA-Aware MPI, the Unified Virtual Addressing (UVA) works as a hidden layer, which can detect the memory type automatically. When the MPI runtime executes any MPI call, it will first check the memory type of send or receive buffers passed by the program. The memory type check and data transferring mechanism are managed by CUDA-Aware MPI itself without extra effort.

Compared to regular MPI with CUDA embedded, the CUDA-Aware MPI runtime directly controls the allocation, utilization, and management of this memory with accelerated communication over network and GPU devices. In detail, the buffer called *vbuf* will be allocated and freed in $MPI_Init()$ and $MPI_Finalize()$, respectively. Once *vbuf* is allocated, it will be divided into blocks and organized as a buffer pool. In the

next section, a CUDA-Aware MPI implementation of CCM and corresponding comparison with previous MPI/OpenMP implementation will be presented.

5.3.2 Integrating GPU accelerators

In the application of GPU-Aware MPI design to CCM, the first consideration concerns the place where GPU accelerator can be introduced in the MPI version of CCM. As illustrated in Chapter 3, the traditional kNN search part can be reduced to a vectorized distance calculation and sorting problem for each lag query point in shadow manifold M_x, where the GPU can offer its computational power to speed such processes up. Specifically, there is a corresponding shadow manifold $M_{x,i}$ for any unique combination of parameter E_i and τ_i. So if the parameter grid, where the program performs an exhaustive search, is large enough, it can impose a massive workload on GPU. However, these can be highly leveraged with the GPU/CPU cluster. In this way, GPU acceleration can be scaled out like CPU cores horizontally. The primarily remaining challenge is the memory limit.

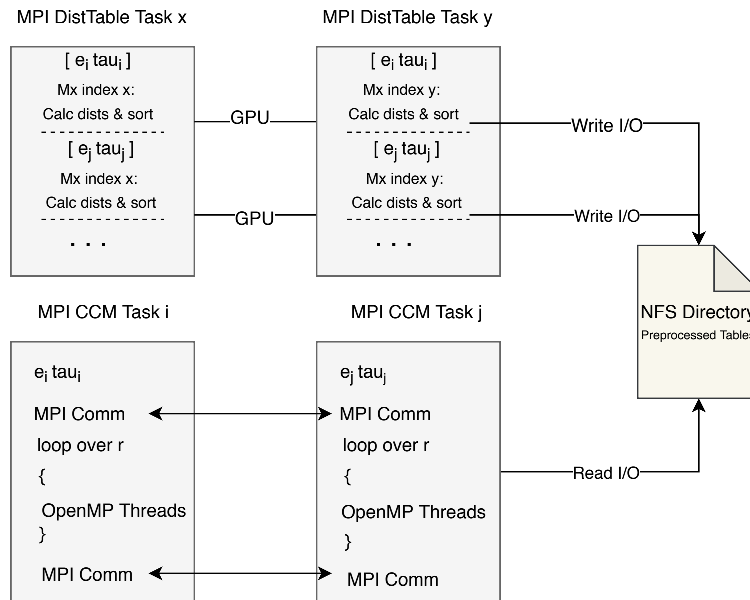

Figure 5.7: Simplified hybrid CUDA-Aware MPI/OpenMP scheme applied on CCM.

For many practical problems, the large output of vectorized nearest neighbors for different $M_{x,i}$ will exceed the maximum memory capacity of every single node. As shown in Figure 5.7, Network File System (NFS) hereby takes care of this challenge by persistence storage into the shared directory. Compared with MPI/OpenMP when applied in a pure CPU-based cluster, a different type of task will be launched before the CCM computation task. It is quite similar to the two pipelines used in the Scala Spark implementation (Chapter 4). The primary difference lies in the way that we handle the preprocessed tables (or matrices). In Scala Spark, a broadcast mechanism has been adopted to share the data or variables across the whole cluster. By contrast, in CUDA-Aware MPI, there is no such mechanism inside the MPI framework, and the

best strategy is to persist these tables into a shared directory like NFS. This step is expected to bring extra I/O cost in return for introducing a certain degree of parallelism using GPU. In essence, the performance gain can only be achieved when the benefit of GPU computation on vectorized nearest neighbors offsets the extra time cost of the requisite disk I/O operations.

To summarize, CUDA-Aware MPI is becoming an increasingly popular programming paradigm to combine different parallel devices to scale computation-intensive algorithms. The GPU hardware quality in the clusters is a key determinant as to whether such a paradigm can secure a net performance gain. When integrating GPU computation power into MPI, the extra data partitioning, transferring and memory allocation on devices, counting as the overhead of introducing another degree of complexity, should be taken into consideration. And these time costs are necessary to achieve multiple degrees of the parallelism on CCM. By redesigning the operations in MPI and introducing one more stage before simplex projection in CCM, the CUDA-Aware MPI version definitely can handle large distance matrices as GPUs of the whole cluster support much more device memory. But still, the overall execution time and related performance require further investigation and comparison. In the next section, the experiment is conducted, and its results will be shown.

5.4 Experiment

5.4.1 Setup

GPU-enabled Cluster

As the Google Cloud Platform cannot support GPU-enabled clusters with customized kernel function execution, a 4-node HPC cluster, which is configured and prepared in the CEPHIL lab, was used to evaluate the performance of the proposed GPU-enable Spark or MPI/OpenMP framework with the hardware configuration shown in Table 5.1. The network connecting these nodes is a 1000Mbps high-speed LAN. Each node is installed with the GNU/Linux Ubuntu 16.04 system, Java 1.8, GCC/g++ 5.4.0, CUDA 10.0 and Spark 2.3.0.

Table 5.1: Hardware configuration of the hybrid cluster

Node	Specification
master	8 cores CPU, 8 GB memory, No GPU (dedicated Spark master)
worker 1	4 cores CPU, 8 GB memory, GeForce GT 710 (192 cores)
worker 2	4 cores CPU, 8 GB memory, GeForce GT 730 (384 cores)
worker 3	8 cores CPU, 16 GB memory, GeForce GT 730 (384 cores)

Table 5.2: CCM baseline parameters setting

CCM-related Notation	Parameters sets
T	10000
R	250
$LSet$	range is $[100, 1000]$, interval is 100
$ESet$	$[1, 2, 4]$
$TauSet$	$[1, 2, 4]$

CCM hyperparameters

The baseline scenario for CCM parameters, as shown in Table 5.2, is set for the comparison in the experiments comparing multi-level parallelism. Time series generated by an Anylogic agent-based model serves as the input to CCM. In the following experiments, the implemented parallel CCM application will be run using these configurations in the aforementioned cluster to obtain the overall computation time. All distributed versions will be tested and compared on the same cluster with the same CCM parameter grid. Afterward, the average computation time can be the metric to compare the efficiency in different cases.

5.4.2 Results

Table 5.3: Implementation Levels

	Implementation Level
Case 1	Spark with optimized external C++ & GPU implementation
Case 2	Spark with external C++ implementation
Case 3	Pure Scala Spark implementation (introduced in chapter 4)
Case 4	MPI/OpenMP implementation (introduced in chapter 4)
Case 5	CUDA-Aware MPI/OpenMP implementation

For this experiment, the baseline scenario is tested on all cases in Table 5.3 using the parallel methods introduced in section 5.2.1. All of the clusters are set up on local machines running Apache Spark and use Yarn as the resource manager. The results of these configuration cases are demonstrated and compared in Fig 5.8. Three Spark related implementations will be compared to evaluate the performance gain after introducing GPU accelerators into the Spark cluster.

Several conclusions can be extracted from this experiment results. Firstly, considering *Case 1 Case 2* and *Case 3*, the pure Scala Spark implementation imposes a heavy cost in terms of disk I/O operation. This reflects the fact that the master node has to gather all prediction skills calculated in different workers and then write the output into the disk (CSV format files). The associated overhead constitutes approximately

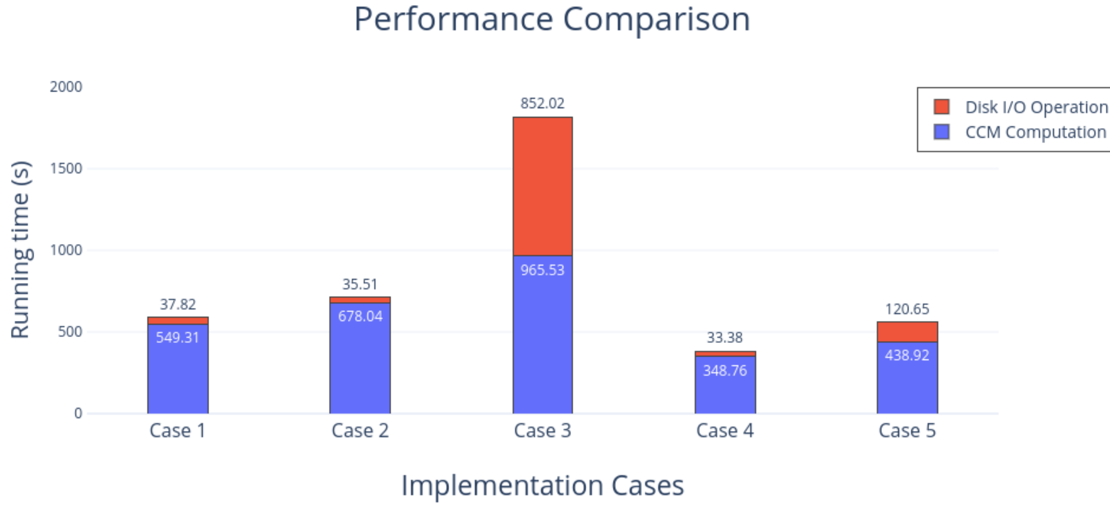

Figure 5.8: This experiment is conducted in the cluster setup described in the Table 4.1. The computation part and disk I/O part are separately measured to compare with respect to differences in each.

50% of the whole execution time. Secondly, the job of the Spark version without optimization does not fully utilize CPU resources, as unbalanced tasks are scheduled among three workers. Following optimization via precomputing the distance table, the duration of such tasks are markedly less heterogeneous, and the system overcomes the performance bottleneck and offers more computational resource support, allowing the utilization rate of CPUs for each node to reach almost 100% during the computation time. However, when comparing the pure Scala Spark version with the PySpark version which calls C++ external application, the experiments demonstrate the computation efficiency brought by the programming language. Scala utilizes the objects almost everywhere and the boxing and unboxing operations, which are the processes of converting a value type back or to any interface type implemented by this value type, significantly reduce the efficiency of the pure Scala implementation. Within, with the C++ external application in Pyspark, Spark only takes care of distributing the parameters and input time series, and a C++ external application handles the core computation of CCM. Without doubt, C++ language has advantages in terms of high efficiency and performance compared to Scala. These results lead to a performance difference of close to 3 times in runtime, despite the overhead of converting the distributed data using a JSON-format string. Also, the programming language efficiency serves as a major reason why MPI/OpenMP framework can achieve much faster execution time compared with the Spark framework even accelerated by GPU. As seen from the results, the computation time for MPI/OpenMP with GPU accelerated distributed version is approximately faster by 30%. With the support of three workstations and their GPU devices, the parallel implementation of CCM for the current cluster setting can achieve approximately 9x speedup when compared to the public library rEDM for baseline parameter combinations. The execution will be further shortened by adding more workers n or upgrading GPU devices in the cluster. As the CCM algorithm is not fully embarrassingly parallel, the

external network communication, serial execution part, and disk I/O operations can be a factor influencing the full execution time.

When only considering *Case 4* and *Case 5*, which mainly adopt the MPI framework to distributed tasks among workers in the cluster, both implementations are faster than any version using the Spark framework. A different node may take different time on the CCM computation and I/O operation. Figure 5.8 only demonstrates the average time among three workers in the GPU/CPU cluster. But surprisingly, CUDA-Aware MPI is slower than pure MPI implementation, even for the core CCM computation part. Disk I/O is much more intensive as this version has to generate the massive preprocessed tables in the first stage and load them in the second stage. As for the CCM computation part, one possible reason behind is that two different task types make the whole program execute in a more serial way. As a result, the benefit of GPU accelerators cannot offset the unexpected time cost. In order to analyze the potential factors, we even choose different input time series I and a number of combination of hyperparameters E_i, τ_i to compare *Case 4* and *Case 5*. The result shows similar patterns. It appears that the introduction of GPU accelerators to the MPI/OpenMP framework brings another degree of complexity, which cannot be offset by GPU acceleration (with current architecture). As CUDA-Aware MPI is capable of automatically managing the data transferring between GPU devices and hosts, the underlying reason why it causes the performance issue still remains unclear. Currently, pure MPI/OpenMP framework outperforms all other distributed CCM versions.

Overall, as explained in the previous chapter, the MPI framework has a strong performance advantage in parallelizing computation-intensive algorithms compared to the Spark framework. Particularly compared to the performance gain under GPU acceleration, Spark framework achieves better performance gain compared to MPI framework, likely also due to the fact that Spark undertakes a system call on the C++ (low-level language) external application. The pure Scala Spark, as a high-level framework, involves extra overheads like Task Initiation, Scheduler Delay, Task Deserialization and Garbage Collection. These additional overheads directly slow down the overall execution time. As such, when comparing the total execution time, the MPI framework has a substantial advantage with/without GPU acceleration in terms of execution latency. By contrast, from a software engineer's point of view, the time spent on creating MPI-version program is almost 10-fold than Spark-version one. To implement the CCM algorithm, approximately 1200 lines of C++ and CUDA code are created when comparing to less than 200 lines of Scala Spark code. The difference directly leads to the prevalence of Spark framework in the industry. Currently, Spark is the most popular framework and widely adopted by engineers to scale algorithms and data-driven business, while MPI framework only remains as a primary tool in the scientific area.

5.5 Conclusion

There are many available techniques to increase the degree of parallelism: OpenMP, CUDA, Thrust, Spark, and MPI. The idea behind parallelization is to always take full advantage of the computing resources available.

Particularly in this chapter, experiments were conducted to demonstrate the benefits of redesigning the Cross Convergent Mapping using a mixture of parallel techniques. Such redesign not only offers support to deal with a high volume of data but also provides greater mechanisms to shorten response latency. Of critical importance for the robust application of the parallel CCM, these performance gains make it possible for the application of the parallel CCM to be applied to a broad set of parameter combinations within in an abbreviated time. Of equal importance, they can make feasible truly interactive exploration of CCM results, where batch computation was previously required.

Still, this work is encumbered by some important limitations. The bottleneck of many parallel computing applications on the message-passing system is the communication and scheduling cost; by contrast, the bottleneck of the shared-memory system is the critical section. CCM is not a typical embarrassingly parallel algorithm, and there remain many challenges to make it parallel at different levels. This chapter introduces a way to parallelize the CCM application using a mixture of parallel techniques and eliminating bottleneck by redesigning the algorithm. As shown from the experiment results, only the Spark framework with CUDA embedded application achieves good performance and dramatically shortens CCM execution latency comparing pure Spark framework. This approach may also shed light on tradeoffs associated with the implementation of other similar parallel algorithms, which are increasingly sought within the sphere of large-scale data analysis.

Finally, this chapter provides a method and demonstrates its efficiency in addressing poor load balancing in parallel application by redesigning the algorithm. For some problems – such as here – this operation can reduce time complexity and reduce the heterogeneity in task cost. At the same time, it is clear that frequently the disadvantage of this method lies in the extra cost for communication and data transfer. However, similar to what happened in GPU acceleration, the communication overhead can be offset if the execution time saved in the computational task is large enough. In addition, the mapping algorithm to address the load balancing issue is often straightforward and effective. In summary, by resolving inherent communication bottlenecks and load balancing performance issues, parallel versions of CCM can be accelerated markedly when compared to the sequential implementation.

Chapter 6

Conclusion

6.1 Discussion and Conclusion

In this thesis, various parallel techniques have been explored and studied to accelerate CCM from different directions. As these experiments demonstrate in the above chapters, any parallel technique is hardly a panacea. Various factors are influencing the performance with parallelization.

In Chapter 3, three CUDA kernel functions are proposed and implemented to accelerate three parts of the Convergent Cross Mapping algorithm (CCM). These parts frame elements of the algorithm into a certain matrix or vector based problem to fit into the data-parallel characteristics required for GPU acceleration. When the input data scale reaches a certain threshold, GPU **kernel functions** outperform corresponding CPU-based implementation as the computational power of (many-core) GPU can offset the overhead associated with the added memory allocation and data transfer. In addition, the experiment comparing the performance of different GPU architectures shows that, by upgrading the GPUs, such threshold and the overall execution time can be explicitly shortened.

In Chapter 4, two cluster frameworks, Apache Spark and Hybrid MPI/OpenMP, are adopted and compared to evaluate the cluster parallelism in CCM. Apache Spark provides high-level functional programming APIs and immutable data structures like **RDD**, **DataFrame** and **DataSet**, to parallelize data-intensive tasks without the need for application programmer handling of threads or (frequently) data partitions. Furthermore, Yarn, as the main resource manager, even offers the support of scheduling policies to enlarge the resource utilization without too much effort. As such, Apache Spark framework offer adequate performance in the industry on Big Data related problems, while reducing software engineering complexity for data scientist users. By contrast, while the Hybrid MPI/OpenMP framework exhibits several fold higher performance in handling computation-intensive tasks. MPI only formulates a set of data point-to-point and collective communication standards in the cluster. All implementations of MPI only provide the corresponding lower-level functions like **MPI_Bcast**, **MPI_Scatter** and **MPI_Gather**. With limited primitive functions, implementing a complex algorithm like CCM via MPI can be tricky and time-consuming. However, most computation-intensive algorithms indeed benefit from such flexibility, and the overall execution latency offers significant time savings over that for Apache Spark.

In Chapter 5, the marriage of GPU acceleration and cluster frameworks have been explored to accelerate

CCM further in GPU/CPU clusters. In the Spark framework, the **pipe** operator is employed to combine CUDA programming with a JVM framework. However, at the same time, the introduction of the **pipe** operator eliminates the advantages of the automatic scheduling service provided by Yarn. As a result, the unbalanced work allocation, which is mainly caused by library size L, becomes the main bottleneck. Hence, GPU acceleration is hereby integrated into external applications to address CPU underutilization by executing pairwise distances matrix and row-wise sorting kernels. In this way, the preprocessing step dramatically reduces the impact of parameter L and makes the overall execution time far closer to the uniform. By contrast, in the Hybrid MPI/OpenMP framework, the **CUDA-Aware MPI** is widely applied, as the API is straightforward to use. The CUDA-aware MPI utilizes UVA, a single address space technique, to automatically manage data transfer between CPU and GPU device. However, in order to use GPU kernel functions, the massive row-wise sorted distance matrices have to be "cached" persistently to the disk (here, using the Network File System), which will generate extra overhead associated with disk I/O operations. Furthermore, from the experiment results, the overall execution latency of CUDA-Aware MPI/OpenMP version is slower than the pure MPI/OpenMP version. It appears that the benefit introduced by these kernel functions cannot offset the extra overhead brought by the coding complexity, especially for the CUDA-aware MPI technique.

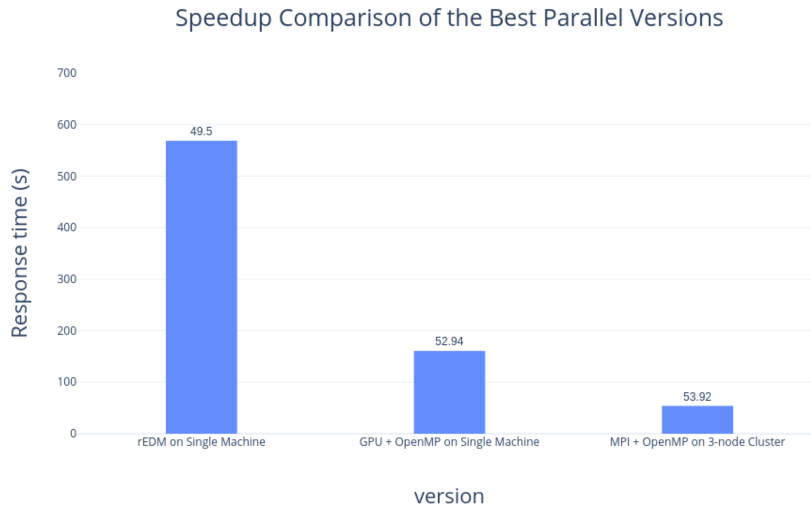

Figure 6.1: The best speedup of Parallel CCM achieved in this thesis, when compared to rEDM.

Overall, different parallel versions can accelerate or scale CCM in different proportions (see Figure 6.1). Compared to the *rEDM* package, the C++ single-threaded version with GPU acceleration (excluding the Pearson correlation coefficient kernel) on a single machine can be around 1.7x faster on the order of input time series $T = 10^3$. But after implementing OpenMP to run multiple realizations in parallel, the best parallel version can run 3.5x faster than rEDM on a single machine with GPU. When considering the cluster environment with GPU devices, pure MPI/OpenMP is the most efficient version to scale CCM in term of both the computation and disk I/O stages among all cluster versions, and can achieve a speed of 10.6x when

compared to rEDM on the single machine. Apache Spark framework can be accelerated using GPU with the **pipe** operator, but to a large extent, the extra efficiency is contributed by the efficiency of external C++ applications. Pure Scala Spark version can be relatively easy for data scientists to implement in a functional style. But generating CSV output files on HDFS remains a central challenge, as the output of large ensembles of prediction skills ρ for different values L – the key to identifying the causal connections – must be generated. The fact that CCM is associated with not just a large input, but often an even larger output distinguishes this problem from many others addressed via the Spark framework and requires careful consideration of the I/O requirements.

6.1.1 Summary

As noted in this thesis, effectively undertaking parallel techniques requires the support of hardware, programming languages and associated libraries or frameworks, and parallel algorithms. This thesis explores different parallel methods on different parallel platforms to implement a parallel version of the CCM algorithm and investigates the performance implications of combinations of alternative approaches. The experiments in chapters 3, 4, 5 demonstrate overall performance advantages compared to current single-threaded implementation of rEDM. By making it easier to investigate a larger range of library sizes, a denser set of such library sizes, and a broader set of values of parameters E and τ, the acceleration of CCM can support speedier and more accurate and insightful causality inference for dynamic systems, which paves the way for systems modeling for policy insight and other applications. Such parallel techniques offer broad prospects for application to other sequential machine learning or data mining algorithms to accelerate the development of artificial intelligence in different spheres. The parallel implementations of convergent cross mapping explored in this thesis, which accelerate the causality inference process, offers the potential to further serve as a valuable reference and guidance for developers and researchers.

6.2 Future Work

There are several directions in this thesis which have not been further investigated.

Firstly, CCM can be potentially optimized at the algorithm level, especially in terms of the central mechanism of searching for nearest neighbors. The nearest neighbors problem is the core component of a variety of applications, usually involving similarity searching. Hence, the optimization of this problem has been widely studied in past literature, including by [1], [18] and [28]. In this research, one of the approximate nearest neighbors algorithm, which is based on locality-sensitive hashing (LSH), has been demonstrated to be efficient in high dimensional spaces. The approximate solutions trade off a certain accuracy for performance. As the statistical measurement in CCM takes hundreds to thousands of samples and the causal effect can only be observed from the change in the ρ ensemble as library size L increases, we suspect that the algorithm is likely to still work with minimal impact if there is substitution of approximate $E + 1$ nearest neighbors

for the exact $E + 1$ nearest neighbors in CCM. The general idea of LSH is to utilize a family of functions to hash lag points into buckets so that the points near each other are located in the same buckets with high probability. By hashing into different buckets, LSH can be easily parallelized, and the time complexity will be reduced to $O(n)$ from $O(nlogn)$. However, the validity of approximate nearest neighbors still requires mathematical proof, and this method may be sensitive to noise in the time series, or the structure of the state space.

Another potential research direction lies in accelerating CCM through the use of FPGA devices. FPGA vendors have contributed an increasingly popular hardware platform alternative to GPU devices to process computational-intensive algorithms, with particular attractiveness in light of FPGA flexibility and customizability. FPGAs are still in their early stage in development in accelerating machine learning and Big Data related algorithms, and only a few companies have released commercial FPGA products to accelerate the next generation of AI, especially targeting the mobile 5G era. Works such as [48] demonstrate the potential high efficiency when FPGAs are applied to image classification tasks when integrating with Deep Neural Networks (DNNs). As such, FPGA platforms serve an interesting alternative to GPUs for CCM implementation. FPGA-based solutions are generally energy-saving and can be embedded into smart devices even without the need for mediation by a host CPU. However, the character of FPGAs also imposes a certain degree of difficulty in algorithmic implementation for developers. For instance, the hardware description language (HDL) is required to manipulate/reconfigure the logic blocks inside FPGAs in order to implement complex digital computations. By contrast, the C-like CUDA language in GPU programming is more likely to be more familiar to – and consequently more quickly acquired by – developers.

Finally, it is necessary to consider how to deal with the output of CCM, in which most researchers are applying CCM seeks to observe the pattern of how prediction skills converge along with library size. Within this task, evidence suggests that a density plot is necessary to analyze the causality connecting input time series reliably. However, the code carrying out the core CCM analysis is often not most convenient or appropriate drawing figures such as histograms. The implementation considered here only achieves writing CSV files to the disk, allowing a script is written in an analysis and visualization tool – such as the statistical package R – will parse the data from CSV files and automatically convert them into plots. The output of the consequent massive data requires time-consuming disk I/O. Two attractive methods addressing this performance issue lie in building a machine learning classifier to distinguish causal from non-causal dependencies within the computer notes automatically, or implementing local plot generation functions co-located with the analysis code. The former option is more intelligent but carries a risk that classification tools such as deep learning will be unable to recognize the patterns, while the latter option offers greater freedom but requires a less flexible analysis and visualization framework. Both methods have their advantages and weaknesses, which cannot become a panacea for all scenarios.

6.3 Contributions

This thesis offers several primary contributions.

The first contribution of this thesis lies in improving the usability of convergent cross mapping by parallelizing this algorithm using different parallel techniques without sacrificing prediction accuracy. Moreover, different implementations explored can be flexibly chosen according to the hardware resources and software configurations available and desirable to accelerate the causality inference procedure. For instance, the GPU acceleration can be added if GPU hardware support is detected, and the choice of cluster mode or local mode could be determined by the argument passed by the researchers. Furthermore, if an advanced GPU-enabled cluster has already been configured and prepared, it is possible to achieve a high degree of parallelism to run a parallel version of CCM. Such performance achievements accelerate the speed of learning with regards to the causal structure of the underlying dynamic systems and can open up additional time for modeling and other studies – advances which pave the way for the real practice using this algorithm.

Secondly, this thesis provides implementations and associated findings for parallelizing existing sequential algorithms. Currently, machine learning and data mining algorithms often involve carrying out complex instructions on large data scales. Given this, findings and implementations from the thesis offer generalization opportunities for application to other similar algorithms. Notably, the nearest neighbors searching and Pearson correlation coefficient calculation in CCM are of vital importance in various fields to define the similarity of objects or structures. This work has implemented such algorithms using popular parallel techniques such as GPUs, MPI/OpenMP, and Spark described in chapter 3, 4, and 5. More importantly, this thesis offers implementation mechanisms for parallelization of sequential iterative algorithms and performance findings for the system that includes such mechanisms. For example, in CUDA programming, n-column vectors are formatted as a matrix in order to fit into the data-parallel programming paradigm. Also, setting the data transformation pipeline using a functional programming style in Spark provides another possible method for performing parallel iterative algorithms without the need for explicit thread handling. Furthermore, the scatter and gather operations in the distributed memory model offer a clear path to assign tasks for parallel execution. The mechanisms can be treated as effective elements for speeding up other machine learning or data analysis algorithms at scale.

The last – but not the least – contribution is that this thesis evaluates performance bottlenecks, and shares some possible directions to address these limitations in parallelizing algorithms. Parallel programs are not ensured a marked speedup on parallel hardware systems. Sometimes parallelized algorithms can be slower than the original sequential one when inappropriate parallel designs are applied when parallelizing the original algorithms [43]. During this process, a common mistake lies in not considering the recurring performance issues: Communication bottlenecks and load balancing. Only under embarrassingly parallel applications are such issues unlikely to be raised. The term embarrassingly parallel generally refers to an algorithm with low communication needs, which can be parallelized very easily by subdividing the data or tasks and assigning

to different processors. The resulting performance will increase linearly along with the number of processors. However, most algorithms require significant interaction between different items of data produced – such as those seen in sorting – or other interactions that are not embarrassingly parallel. Such algorithms can be parallelized but in a more complex, less obvious manner. The challenge lies in how to redesign the algorithms to enable parallel execution while balancing utilization of CPUs, GPUs, network, and resource constraints. Within the work of parallelizing CCM considered here, the need for effective load balancing represents the most central performance issue in the Spark framework with GPU embedded applications – i.e., keeping all processors busy as much as possible. This problem recurs prominently in discussions of parallel processing, often reflecting the fact that the workload of each thread or process handled may not always be uniform, with some tasks remaining idle while others are busy nearly all the time. Poor load balancing can be observed through collecting the performance utilization metrics and addressing accordingly. In this way, parallel algorithms can fully take advantage of computational power in parallel computers. Particularly in this thesis, the unbalanced workload mainly comes from the library size L in each subsample. L determines the windows size employed, and the corresponding state space size. The computational workload (include sorting) becomes more massive when the window size becomes relatively larger. However, after incorporating preprocessing of the distance matrix tables, the workload became more uniform, and the central load balancing issues were addressed.

6.4 Publications Related to the Thesis

A forthcoming paper – accepted in April 2019 – will be presented at and published in the Proceedings of the International Conference on Social Computing, Behavioral-Cultural Modeling and Prediction and Behavior Representation in Modeling and Simulation (SBP-BRiMS 2019), based on the content of chapter 4. The citation is as follows in vancouver style:

Pu B, Duan L, Osgood ND. Parallelizing Convergent Cross Mapping Using Apache Spark. InInternational Conference on Social Computing, Behavioral-Cultural Modeling and Prediction and Behavior Representation in Modeling and Simulation 2019 Jul 9 (pp. 133-142). Springer, Cham.

References

[1] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*, 51(1):117, 2008.

[2] Christoph Bandt and Bernd Pompe. Permutation entropy: a natural complexity measure for time series. *Physical review letters*, 88(17):174102, 2002.

[3] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for cuda. In *GPU computing gems Jade edition*, pages 359–371. Elsevier, 2012.

[4] Eugen Betke and Julian Kunkel. Real-time i/o-monitoring of hpc applications with siox, elasticsearch, grafana and fuse. In *International Conference on High Performance Computing*, pages 174–186. Springer, 2017.

[5] Reza Bosagh Zadeh, Xiangrui Meng, Alexander Ulanov, Burak Yavuz, Li Pu, Shivaram Venkataraman, Evan Sparks, Aaron Staple, and Matei Zaharia. Matrix computations and optimization in apache spark. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 31–38. ACM, 2016.

[6] André R Brodtkorb, Trond R Hagen, and Martin L Sætra. Graphics processing unit (gpu) programming strategies and trends in gpu computing. *Journal of Parallel and Distributed Computing*, 73(1):4–13, 2013.

[7] Liangyue Cao. Practical method for determining the minimum embedding dimension of a scalar time series. *Physica D: Nonlinear Phenomena*, 110(1-2):43–50, 1997.

[8] Dar-Jen Chang, Ahmed H Desoky, Ming Ouyang, and Eric C Rouchka. Compute pairwise manhattan distance and pearson correlation coefficient of data points with gpu. In *2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, pages 501–506. IEEE, 2009.

[9] Michael Connor and Piyush Kumar. Parallel construction of k-nearest neighbor graphs for point clouds. In *Volume Graphics*, pages 25–31, 2008.

[10] Michael Connor and Piyush Kumar. Fast construction of k-nearest neighbor graphs for point clouds. *IEEE transactions on visualization and computer graphics*, 16(4):599–608, 2010.

[11] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *Computing in Science & Engineering*, (1):46–55, 1998.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[13] Ethan R Deyle and George Sugihara. Generalized theorems for nonlinear state space reconstruction. *PLoS One*, 6(3):e18295, 2011.

[14] Ralph Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, 1990.

[15] Ivan Dvořák and Jan Klaschka. Modification of the grassberger-procaccia algorithm for estimating the correlation exponent of chaotic systems with high embedding dimension. *Physics letters A*, 145(5):225–231, 1990.

[16] J-P Eckmann and David Ruelle. Fundamental limitations for estimating dimensions and lyapunov exponents in dynamical systems. *Physica D: Nonlinear Phenomena*, 56(2-3):185–187, 1992.

[17] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[18] K Fukunage and Patrenahalli M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE transactions on computers*, (7):750–753, 1975.

[19] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using gpu. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–6. IEEE, 2008.

[20] Vincent Garcia, Eric Debreuve, Frank Nielsen, and Michel Barlaud. K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching. In *2010 IEEE International Conference on Image Processing*, pages 3757–3760. IEEE, 2010.

[21] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *Vldb*, volume 99, pages 518–529, 1999.

[22] William Gropp. Mpich2: A new start for mpi implementations. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 7–7. Springer, 2002.

[23] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[24] David Hall, Daniel Ramage, Jason Zaugg, Alexander Lehmann, Jonathan Merritt, Keith Stevens, Jason Baldridge, Timothy Hunter, Dave De-Caprio, Daniel Duckworth, et al. Scalanlp: Breeze, 2009.

[25] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876, 2007.

[26] Craig Hiemstra and Jonathan D Jones. Testing for linear and nonlinear granger causality in the stock price-volume relation. *The Journal of Finance*, 49(5):1639–1664, 1994.

[27] W Daniel Hillis and Guy L Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[28] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998.

[29] Dana Jacobsen, Julien Thibault, and Inanc Senocak. An mpi-cuda implementation for massively parallel incompressible flow computations on multi-gpu clusters. In *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, page 522, 2010.

[30] Supun Kamburugamuve, Pulasthi Wickramasinghe, Saliya Ekanayake, and Geoffrey C Fox. Anatomy of machine learning algorithm implementations in mpi, spark, and flink. *The International Journal of High Performance Computing Applications*, 32(1):61–73, 2018.

[31] Holger Kantz and Thomas Schreiber. *Nonlinear time series analysis*, volume 7. Cambridge university press, 2004.

[32] Jiri Kraus. An introduction to cuda-aware mpi. *Weblog entry]. PARALLEL FORALL*, 2013.

[33] Dimitris Kugiumtzis. State space reconstruction parameters in the analysis of chaotic time seriesthe role of the time window length. *Physica D: Nonlinear Phenomena*, 95(1):13–28, 1996.

[34] Vipin Kumar. *Introduction to Parallel Computing: Design and Analysing of Algorithms*. Springer, 1994.

[35] Peilong Li, Yan Luo, Ning Zhang, and Yu Cao. Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms. In *2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 347–348. IEEE, 2015.

[36] Karin E Limburg, Robert V O'Neill, Robert Costanza, and Stephen Farber. Complex systems and valuation. *Ecological economics*, 41(3):409–420, 2002.

[37] David Luebke. Cuda: Scalable parallel programming for high-performance scientific computing. In *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*, pages 836–838. IEEE, 2008.

[38] Douglas A Luke and Katherine A Stamatakis. Systems science methods in public health: dynamics, networks, and agents. *Annual review of public health*, 33:357–376, 2012.

[39] Chuan Luo, Xiaolong Zheng, and Daniel Zeng. Causal inference in social media using convergent cross mapping. In *2014 IEEE Joint Intelligence and Security Informatics Conference*, pages 260–263. IEEE, 2014.

[40] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases*, pages 950–961. VLDB Endowment, 2007.

[41] Huanfei Ma, Kazuyuki Aihara, and Luonan Chen. Detecting causality from nonlinear dynamics with short-term time series. *Scientific reports*, 4:7464, 2014.

[42] Jesus Maillo, Sergio Ramírez, Isaac Triguero, and Francisco Herrera. knn-is: An iterative spark-based design of the k-nearest neighbors classifier for big data. *Knowledge-Based Systems*, 117:3–15, 2017.

[43] Norm Matloff. Programming on parallel machines. *University of California, Davis*, 2011.

[44] John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[45] Dan Mønster, Riccardo Fusaroli, Kristian Tylén, Andreas Roepstorff, and Jacob F Sherson. Causal inference from noisy time-series datatesting the convergent cross-mapping algorithm in the presence of noise and external influence. *Future Generation Computer Systems*, 73:52–62, 2017.

[46] David M Mount and Sunil Arya. Ann: library for approximate nearest neighbour searching. 1998.

[47] John Nickolls, Ian Buck, and Michael Garland. Scalable parallel programming. In *2008 IEEE Hot Chips 20 Symposium (HCS)*, pages 40–53. IEEE, 2008.

[48] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, et al. Can fpgas beat gpus in accelerating next-generation deep neural networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 5–14. ACM, 2017.

[49] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala*. Artima Inc, 2008.

[50] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *2009 17th Euromicro international conference on parallel, distributed and network-based processing*, pages 427–436. IEEE, 2009.

[51] Jorge L Reyes-Ortiz, Luca Oneto, and Davide Anguita. Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf. *Procedia Computer Science*, 53:121–130, 2015.

[52] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–10. IEEE, 2009.

[53] George Sugihara, Robert May, Hao Ye, Chih-hao Hsieh, Ethan Deyle, Michael Fogarty, and Stephan Munch. Detecting causality in complex ecosystems. *science*, 338(6106):496–500, 2012.

[54] George Sugihara and Robert M May. Nonlinear forecasting as a way of distinguishing chaos from measurement error in time series. *Nature*, 344(6268):734, 1990.

[55] Floris Takens. Detecting strange attractors in turbulence. In *Dynamical systems and turbulence, Warwick 1980*, pages 366–381. Springer, 1981.

[56] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[57] Ajay K Verma, Amanmeet Garg, Andrew Blaber, Reza Fazel-Rezai, and Kouhyar Tavakolian. Analysis of causal cardio-postural interaction under orthostatic stress using convergent cross mapping. In *2016 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 2319–2322. IEEE, 2016.

[58] Andreas Wagner. Causality in complex systems. *Biology and Philosophy*, 14(1):83–101, 1999.

[59] Hao Wang and Baochun Li. Mitigating bottlenecks in wide area data analytics via machine learning. *IEEE Transactions on Network Science and Engineering*, 2018.

[60] Chao-Tung Yang, Chih-Lin Huang, and Cheng-Fang Lin. Hybrid cuda, openmp, and mpi parallel programming on multicore gpu clusters. *Computer Physics Communications*, 182(1):266–269, 2011.

[61] H Ye, A Clark, E Deyle, and G Sugihara. redm: an r package for empirical dynamic modeling and convergent cross-mapping. 2016.

[62] Yuan Yuan, Meisam Fathi Salmi, Yin Huai, Kaibo Wang, Rubao Lee, and Xiaodong Zhang. Spark-gpu: An accelerated in-memory data processing engine on clusters. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 273–283. IEEE, 2016.

[63] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[64] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

Appendix A

Documentation

A.1 Codebase

The project and related code is on the GitHub repository:
https://github.com/ALexanderpu/ParallelCCM

A.2 Sample Configuration File

The file name is ccm.cfg in the codebase:

```
[paths]
input=/home/bo/cloud/CCM-Parralization/TestInputCSVData/test_float_1000.csv
exteralProgram=/home/bo/cloud/CCM-Parralization/SparkVersion/sparkc
output=/home/bo/cloud/CCM-Parralization/Result
[inputs]
x=lynx
y=wolf
[parameters]
E=2,3 // ESet
tau=1,2 // tauSet
num_samples=250 // R
LStart=100 // LSet
LEnd=800
LInterval=100
[options]
GPUAcceleration=1
GenerateOutputCSV=1
```