2013

On-the-fly tracing for data-centric computing : parallelization, workflow and applications

Lei Jiang
Louisiana State University and Agricultural and Mechanical College, ljiang@cct.lsu.edu

ON-THE-FLY TRACING FOR DATA-CENTRIC COMPUTING: PARALLELIZATION, WORKFLOW AND APPLICATIONS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Lei Jiang
B.E. University of Science and Technology of China, 2005
M.S., Wuhan University, 2007
M.S., Louisiana State University, 2011
May 2013

Acknowledgments

I would like to thank my supervisor Dr. Gabrielle Allen. The research work in the thesis is enabled through her funded projects. She envisioned the multidisciplinary research with a high level of application significance. She also has been helpful with my study and life here at Louisiana State University (LSU) during the five years since my arrival. I am also thankful to the other two members in my graduate advisory committee: Dr. Jian Zhang in Department of Computer Science, Dr. Q. Jim Chen in Department of Civil and Environmental Engineering and Dr. Michael Tom in Department of Mathematics: they provided lots of helpful comments and suggestions in the course of dissertation writing.

Thanks to the fellow students and many nice staff members and professors I met at LSU. The atmosphere has made me become more motivated in investigations and discussions with regard to academic research. The help from LSU technical staff for high-performance computing resources is appreciated as well.

Lastly, thanks to my parents for their infinite love and support.

Table of Contents

List of Tables

List of Figures

Abstract

As data-centric computing becomes the trend in science and engineering, more and more hardware systems, as well as middleware frameworks, are emerging to handle the intensive computations associated with big data. At the programming level, it is crucial to have corresponding programming paradigms for dealing with big data. Although MapReduce is now a known programming model for data-centric computing where parallelization is completely replaced by partitioning the computing task through data, not all programs particularly those using statistical computing and data mining algorithms with interdependence can be re-factorized in such a fashion. On the other hand, many traditional automatic parallelization methods put an emphasis on formalism and may not achieve optimal performance with the given limited computing resources.

In this work we propose a cross-platform programming paradigm, called "on-the-fly data tracing", to provide source-to-source transformation where the same framework also provides the functionality of workflow optimization on larger applications. Using a "big-data approximation" computations related to large-scale data input are identified in the code and workflow and a simplified core dependence graph is built based on the computational load taking in to account big data. The code can then be partitioned into sections for efficient parallelization; and at the workflow level, optimization can be performed by adjusting the scheduling for big-data considerations, including the I/O performance of the machine. Regarding each unit in both source code and workflow as a model, this framework enables model-based parallel programming that matches the available computing resources.

The techniques used in model-based parallel programming as well as the design of the software framework for both parallelization and workflow optimization as well as its implementations with multiple programming languages are presented in the dissertation. Then, the following experiments are performed to validate the framework: **i)** the benchmarking of parallelization speed-up using typical examples in data analysis and machine learning (e.g. naive Bayes, k-means) and **ii)** three real-world applications in data-centric computing with the framework are also described to illustrate the efficiency: pattern detection from hurricane and storm surge simulations, road traffic flow prediction and text mining from social media data. In the applications, it illustrates how to build scalable workflows with the framework along with performance enhancements.

Chapter 1
Introduction

The prevalence of data-centric computing, as well as the emerging field of data-enabled science and engineering, is leading to a wide range of sophisticated computational models for high-level knowledge discovery in large-scale data requiring more advanced computing resources. MapReduce [1], as a relatively new programming model, means that canonical parallelism can be imposed to data-centric computing via a straightforward way of task partitioning: a part of the input data can be mapped to a processing unit and the final result is then collected from all the processing units running in parallel. Now, the new parallel architecture, along with Hadoop [2], the implementation of MapReduce with efficient hardware integration, is being widely used in many fields. In particular, the interoperability of Hadoop enables cross-platform data processing and makes parallel computing easier for data scientists who are not familiar with low-level system programming. Now many real-world applications are involved with big data and this prompts the deployment of the new programming model especially for data-intensive computing. As it significantly enhances the efficiency of large-scale data processing, the new programming paradigm enables users to work under a new platform and can directly map their data-centric tasks to the framework in a convenient manner. Ideally, considering that there is always a non-high-performance solution that users just run it on a single PC, it would be the best for them to transplant the model and data to a new high-performance platform with the least additional work – in an automatic way. So, the following scenario is assumed as the target throughout the dissertation, and is indeed common for many researchers and data analysts: there are already models or codes that can be successfully executed as a small-scale solution, and it is desired to accommodate large-scale data and high-performance computational resources for the existing models whose effectiveness has been proven. Intuitively, an accelerator is needed to parallelize the model. It is actually not enough as platform support is also important at the workflow level. To illustrate the vision, we would start from giving some general comments MapReduce, and then propose the idea of a new solution.

As an advanced programming model, one advantage of MapReduce is that the parallelism is almost independent from the original data processing algorithm, or the slow sequential code itself: the same routine can be applied for each partition of data sets, the user code is like an argument that is simply passed to the higher-level parallel architecture through its APIs and then scaled up. In other words, MapReduce maintains the original model and only imposes *parallelism on data*. It works in a straightforward manner for data analysis that doesn't require data dependence within its procedure. For example:

A large text data set includes very many essays (more than 1 million) and each essay typically consists of more than 1000 words. In order to efficiently calculate the average length of each essay throughout the entire data set, it would be advantageous to use MapReduce model in the following way: i) partitioning the data set and mapping each portion of it to a processor (or a node of a cluster. hence "processor" is used to represent both); ii) obtaining the average length for those on the same processor while counting the number of essays here (map) ; iii) collecting the results across all the processors and calculate the total result (reduce) by using a weighted average.

The above example suits MapReduce very well as the same model (calculate the average length of a given data set of essays) can work on each processor and all the user needs to provide is a reduce function (or combination function) that computes the weighted average. That is, the model itself can be regarded as a black box and can be repeatedly called no matter whether MapReduce is needed or not. However, this leaves one issue behind: the parallelism fails to exploit multiple levels of parallelism in a piece of user code [3] and isn't designed to encourage data exchanging or dependence in the middle of computation [4], while in many machine learning algorithms, the iterative solver would consistently require the result trained from the last data record. To make it more explicit, the following instance illustrates the situation:

A large text data set is given again but now we want to find the variance of the length of the essays. When performing the computation using MapReduce, it has to be divided into three steps: i) *counting the length for each essay and then getting the average for all the essays using MapReduce (as in the above example); ii) taking the average as an argument and then calculate the sum of square for the deviation to average using MapReduce and iii) Collecting the result and divide the sum of square by the number of essays, whose output will be the variance.*

So, the MapReduce routine has to be adopted twice here because of data dependency, and the single model, which is to get the variance of length for any given text data set, can't be iteratively called in this case. While this is considered as a circumstance that the parallelism can't only be imposed on data, a model can contain numerous subroutines, where some are well compatible with MapReduce, some even involve hierarchical parallelism [5] and some have to bear with sequential execution.

On the other hand, automatic parallelization for code generation [10, 11, 12], which aims to perform source-to-source transformation to directly make user programs parallelizable, has been a popular research topic for over twenty years. Traditional parallelization approaches focus more on the formalism, especially for the optimization of loop nest [13]. With [12] as an example, normal form and separable normal form are defined in the code and can be extracted. Thereby, nested loops are reduced in the newly generated code so it is easily configured with loop-based accelerators such as OpenMP. By this means, while the generated code essentially optimizes the compilation performance, data dependency can't be substantially altered.

More specifically, tiling [14, 15, 16], as a key technique used in automatic parallelization, are exerted on nested loops and can achieve better practical performance for reducing loop irregularities and utilizing data locality. As the output is still source code and doesn't take the available computational resource into account, it mostly perform *ad-hoc* processing in programs. For the same example of calculating the average and variance of text length on a big data set, it would not give a better solution than the aforementioned as long as the actual code is nicely written. Moreover, compiler-level parallelization is also an option in the context of multi-core computing [17, 18]. Dependency-free iterations of a for-loop can be extracted and assigned to processors so the construction of dependency graph still becomes the precondition.

In sum, for parallelizing a given piece of code, extracting the concurrency is a primary task as it determines what can be executed in parallel and what can't; afterwards, matching the level of parallelism to the availability of high-performance computing resources is also important to ensure the actual execution sequence is optimal. At a higher level of parallelization, MapReduce doesn't limit the form of user code at all (but it is users' responsibility to impose it properly) and allows high interoperability between the user program and parallel architecture, while the majority of automatic parallelization methods for code generation have requirements on user program and the resultant code usually doesn't dynamically interact with the system side. Thus, they represent two directions in the development of parallel programming.

In this dissertation, we propose a new approach for data-centric computing which lies at the middle of the two directions: with the concept of a *big-data approximation*, which means that the parallelization algorithm only focuses on the routines with regard to the input data set of the program and all other computations are considered trivial, the problem of automatic parallelization is simplified. Then, our approach consists of two major stages at the code-parallelization level: big-data tracing and dynamic task scheduling: *i)* mid-level codes are generated from user program, including the scan of code for parallelism exploitation and *ii)* a task scheduler arranges the execution routine before runtime based on the available computing resources, the core data dependence graph and the parallelization granularity in between. Fig. 1.1 depicts the relationship of our work to existing techniques in several aspects.

So, after introducing related work in Chapter 2, the approach of on-the-fly tracing (OTF-T) is described in Chapter 3 as the way to exploit the parallelism from code based on big-data approximation, followed by the corresponding parallelization mechanism: the structure of core dependence graph, with the assumption of big-data approximation that considers all the computation beyond the large-scale data input trivial, is defined as a simplification to the traditional directed acyclic graph. Then, parallelization is performed through the operations on the core dependence graph: a

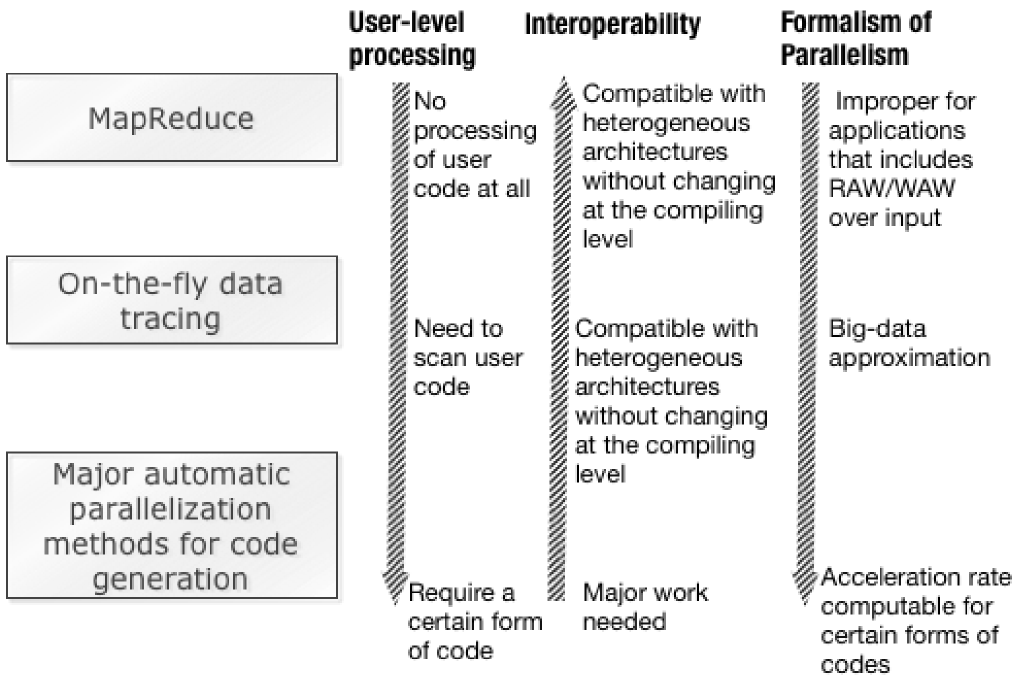

User-level processing
Interoperability
Formalism of Parallelism
MapReduce
On-the-fly data tracing
Major automatic parallelization methods for code generation
No processing of user code at all
Need to scan user code
Require a certain form of code
Compatible with heterogeneous architectures without changing at the compiling level
Compatible with heterogeneous architectures without changing at the compiling level
Major work needed
Improper for applications that includes RAW/WAW over input
Big-data approximation
Acceleration rate computable for certain forms of codes

FIGURE 1.1: High-level comparison between existing techniques for parallel data-centric computing and the work in this dissertation

cut on the graph can partition the code into sections and such operations can be recursively adopted; after codes are transformed into sections, it then becomes a scheduling problem for arranging all the sections in a proper sequence with the available computing resources. Then, in Chapter **??**, the concept is extended to workflow level that includes multiple modules or programs that are related to the same big data. A general task descriptor is defined and the criteria with regard to big data are involved in workflow optimization on large computers. No matter a task descriptor actually refers to a code section, or a sophisticated module in scientific computing, the scheduling fits both the scenarios with a goal of minimizing the delay time while considering the non-trivial large data transmission in synchronization.

Compared to traditional methods in workflow optimization (or optimal resource selection) [19, 20], the time for data transmission is an important factor in impacting performance. The larger the data size is, the more expensive the cost of moving data (for task execution) is. To solve this problem, the big-data approximation helps to identify the critical path as the *focus tasks* of the workflow and others can be *branches*, based on the level of big-data involvement (although the information may not be given for each task). Focus tasks have higher priority and data moving should be minimized between them. Then, while there is more freedom for each branch task, they can be handled using a greedy approach. Thereby, by using the same concept in two different levels in high-performance computing, such a software framework, presented in Chapter 4, can be designed and implemented.

To verify the performance of the framework in real-world applications, a series of experiments are performed: first, parallelization speed-up is measured using typical data analysis routine such as naive Bayes and k-means algorithm on multicore and cluster architectures; then, three typical data-intensive computing tasks using the framework are illustrated as well: *i)* pattern detection from hurricane and storm surge simulations; *ii)* road traffic flow prediction and *iii)* text mining from social media web (mainly Twitter). Each application is described in Chapter 5, Chapter 6 and Chapter 7 with details for its purpose, problem setting, the original computational model(s) and the use of the OTF-T framework to accelerate them on large computers. Lastly, we conclude this dissertation in Chapter 8.

Chapter 2
Related Work

Related work falls into the three areas: *i)* MapReduce and related frameworks (Section 2.1); *ii)* parallelization techniques (Section 2.2) and *iii)* workflow optimization and scheduling (Section 3.4). Each part is related to the core of this dissertation in a different way: *i)* is a programming model for its ease of applicability and relatedness to data-intensive computing; the techniques in *ii)* are also relevant or used in the on-the-fly tracing framework; and *iii)* is the extension of the concept of big-data approximation at a higher level for workflow on high-performance computing resources or grid/cloud computing.

2.1 MapReduce and Related Frameworks

MapReduce [1] was first designed and implemented at Google Inc. for processing large amounts of raw data, especially for crawled documents on the web. It has become increasingly popular over the years as such a procedure of data processing is general in many applications, including some significant one in influenza epidemics detection [21] and genome analysis [22]. While it was first deployed on large clusters, scalability is not an issue as it doesn't require communication between processors or nodes of clusters until the final result is to be collected. Such generality extends the outreach of MapReduce: Pig Latin [23] makes MapReduce even easier to use in SQL-based database queries so users can maintain their code at the high level without additional scripting: a data flow language is defined on top of SQL in order to accommodate nested data model, such as ”FILTER”, ”FOREACH” and ”COGROUP”. Such an integration with a database leads to a new environment for data processing and Pig is now a popular tool as a top-level Apache project [24].

A scheduling scheme, longest approximate time to end (LATE) is then designed to improve the MapReduce performance on heterogeneous environments[25] in comparison to the Hadoop task scheduler: while Hadoop holds a homogeneous assumption in scheduling tasks, the LATE scheduler performs speculative execution by prioritizing, selecting fast nodes and capping to prevent thrashing.

This scheduler can be run on Amazon EC2 (Amazon Elastic Compute Cloud) [26] to extend the MapReduce paradigm in cloud computing.

Moreover, with the computing capability being also convenient for graphics processing units (GPU), the corresponding MapReduce-based framework is available: Mars [27] becomes such by bridging the parallelism to CUDA, NVIDIA's parallel computing architecture mainly used for GPU. Several applications are tested for performance comparison, and according to the authors, by utilizing *char4* data type in GPU as optimization, the Mars framework achieves less computational time than using CPU.

Typical applications of the MapReduce-based framework, which can be directly used by machine learning and data mining specialists, also emerged in recent years. As a representation, NIMBLE [7] provides such services towards "parallelization with limited effort". Abstract tasks are defined and two kinds of such differentiate the computations with data involved or not. Then, tasks can be spawned across processors and form DAGs (directed acyclic graphs) to indicate the chain of execution. While the framework prompts the level of parallelism to explicit tasks to give the execution more flexibility, it still requires the users themselves to define "tasks" although the primary concern for machine learning and data mining specialists is "where to find a dispensable task". Other recent application-specific framework includes [28, 29] (with MapReduce) and Pregel [30] for large-scale graph processing.

On the other hand, an alternative of MapReduce, which provides similar functionality at a different level, can be Dryad [31]. Dryad is designed as a general-purpose execution engine and allows users to set up "vertices" and "edges" in the Dryad graph as a job schedule. Dryad operates at a higher level and as such has its own scripting language "Nebula" for developers. With more interoperability, Dryad supports more applications including SQL query and data mining. However, while it is not designed for data-centric computing, it would still be practically difficult or labor-intensive to harness a specific application to such a platform that is more standalone. It is then improved to DryadLINQ [32], which is capable of compiling a set of queries as constructs to make the execution more straightforward.

Though not very often, the disadvantage of MapReduce has also been discussed in multiple aspects. [3] points out that the MapReduce-style parallelism only supports one-way scalability, which means it only targets large-scale data and there is no necessary flexibility in middle of data processing. [4] also mentions the difficulty of handling data processing/analytics models that include synchronization in between such as online learning and Monte Carlo simulations. Such worries are at an early stages before a replacement emerges. However, it indicates that combining system-level programming model with compilation-level pre-processing (or code transformation) would be a plausible solution.

2.2 Parallelization Techniques

Compared to system-level programming frameworks for large-scale computing, parallelization has a relatively longer history. It has been a target for the source code itself that sequential code can be be transformed into a multi-threaded or vectorized code in order to utilize advanced computing resources, including multi-core processors, commodity clusters, commercial clouds or GPUs. In fact, automatic parallelization has been recognized as a difficult problem [33]. Currently, multi-core architecture based on shared memory is always referred to for *automatic parallelization* for its platform independence and neatness of the form of code, but it also limits the power of automatic parallelization at the concept level, although automatic code generation is also feasible for other platforms. The techniques required is also different: parallelization always put an emphasis on ad hoc analysis on code, especially on loops (while loops usually take the majority of execution time).

At the compilation level, many parallelization techniques are platform dependent but some general schemes still work as almost all the programming languages share the same features. As a tradition, polytope model [13, 34] is introduced in the dissertation as the main non-trivial method. Polytope model has been invented for decades and is still a primary method in current parallelization tools, it is a mathematical model that can be specified by two polynomials and the inequations can be listed for the dependence in nested loops. Solving it means tiling the loops in a proper way that a set of iterations are rearranged to be concurrent. At the same time, the limitation of the polytope model exists in the fact that it is hard to handle irregular control in programs. And some approximation methods have to be used while polytope model is "absolute" for in standard settings of programs. Nevertheless, its formality still resulted in many later works [16, 35, 36].

In essence, the aforementioned method also indicates that there would be a canonical solution for given forms of code, so application-specific parallelization techniques are the other part of the work. [37] specifies the parallelism for tree reductions in dynamic programming: the parallelism can be extrated from the tree contraction algorithm and then a code generator can be designed to follow the pre-defined concurrency along tree nodes with optimization mechanism. [12] analyzes the form of code for matrix multiplication and uses a normal form to represent it, and then the parallelization algorithm converts the actual code into such a form; the parallelizer is practically implemented on top of a series of analyzers and its performance is illustrated via testing multiple numerical solvers with multi-threading.

Thread-level speculation (TLS) is another direction that facilitates automatic parallelization [11]. It allows the parallelism to be seen and arranged by compilers solely, instead of just transforming the code to another style. So, codes can be pre-fetched and assigned to processors without any change. Then, necessary run-time checks of variable values are necessary to ensure the parallelism is really adoptable. The mainstream implementations rely heavily on the support of hardware or compilers: speculative regions are set up and buffered in cache, or speculative pipelining is applied by multi-scalar [39]. So, thread-level speculation essentially simplifies manual parallelization by establishing additional structures directly on hardware and its performance is closed bound to the machine configurations in terms of dynamic thread length and number of speculative regions [40]. The sophistication of its deployment thereby doesn't make the technique itself lead to a general standard of automatic parallelization. In more recent work, it was elaborated to nested thread-level speculative parallelism by adding the in-thread mechanism based on conditional probability [41].

2.3 Workflow Optimization and Scheduling

In comparison to MapReduce-related frameworks and parallelization techniques, workflow is at a higher level in the system so the actual computational model in a workflow is more difficult to be described but can only be abstracted. Middleware, in grid/cloud computing or any distributed environment, can operate on workflows. Also, it makes the concept of ”workflow” become a generic term that can be applied in many processes or applications.

In the context of this dissertation, workflow refers to ”scientific workflow” [42], which consists of execution components, resources for computation, the mapping between resource to workflows, and other models on top of the core flow (e.g. control flow model and data flow model). A well-defined workflow should be fault-tolerant. In order to optimize a workflow, the mapping and scheduling algorithms and other related mechanisms would be taken into account: Pegasus [43], a known workflow management system, could partition the original workflow into any number of sub-workflows and cluster the tasks for fine granularity in execution. In grid or cloud computing, the resources may not be immediately available and dynamically change over time, so it prompts the workflow to be capable of performing optimization adaptively. [19] designs a discovery service in workflow so that the resources, represented by web services, can be dynamically selected through active discovery using service name, metadata or ontology; and then an optimization service exerts additional criteria to select the best one from a pre-selected list. In grid computing, a blackboard approach [44] from artificial intelligence allows an agent to actively search for optimal QoS (Quality of Service) parameters to ensure the estimated cost is minimized at each step. Then, regarding resource scheduling in a situation that the load of tasks is larger than the capacity of real-time processing, a set of heuristic algorithms are proposed in [45] based on the formalization of mapping, communication model and latency for linear workflow optimization, and a ”longest first” scheme is shown to be the optimal in the authors' experimental setting, indicating that it is preferable to choose the currently longest operation as the next schedule in the queue.

Admittedly, some work in workflow optimization is based on hypothetical circumstances while real conditions and restrictions may not be fully considered. For data-centric computing, the time for moving data is non-trivial and I/O performance would play a significant role in workflow as well. While it already becomes the trend and real challenges in many applications, [46] mentions the challenge in scientific workflow modeling with respect to the complication of data size and the increasing procedures in the workflow, and points out that data would important in the process of workflow construction and should be individually treated on top of other components.

Chapter 3
On-the-fly tracing for parallelization

While MapReduce is easy to deploy and compatible with different platforms or programming languages, it lacks the capability in handling a sophisticated piece of code for data analysis that requires synchronization or data dependence in between. On the other hand, current parallelization techniques usually target a certain form of code and the elaboration of details in coding style limits its generality. In this chapter, a new approach, considered as a blending of the advantages of both MapReduce and parallelization, is presented. At the system level, the same view as MapReduce is adopted here: the data-centric task comes with a large-scale data input, and in the course of execution, it can do more than MapReduce as the on-the-fly tracing is exerted on the source code itself, which is the same as the aforementioned parallelization techniques, to exploit parallelism and assign the partitioned sections to available computing resources. "On-the-fly" means that the parallelization is not rigid and can adjust itself accordingly (e.g. users' knowledge in some time-consuming functions can help with the parallelization at a higher level). As mentioned in Chapter 1, the on-the-fly tracing still requires scanning the source code but doesn't require the formality of the code itself. It does give parallelization more flexibility though the approach itself can lead good parallelization results.

3.1 Concept

In data-centric computing, it is considered that there is logically one input data set for a task, which can be very large. So, the parallelism is then to be exploited over data: in MapReduce, the parallelism of the whole task is equivalent to that of data, assuming that there is no dependency at all between each partition. The parallelism that we address throughout the paper doesn't hold the same assumption but requires the scanning of user code for parallelism exploitation.

Thus, we consider parallelism with big-data approximation. In many circumstances, a function is regarded as a unit of user code input. Then, a couple of definitions can be made.

Definition 1. *Big-Data Approximation: for parallelizing a function $f(\mathcal{D}, ...)$, where $\mathcal{D}$ is the only logical data input of f, only the parallelization of routines that contain $\mathcal{D}$ or its mappings is needed, while the time for other computations is considered as a negligible part in the total.*

With *big-data approximation*, the parallelization is simplified for the reduction of data dependence as well as formality. That is, in order to parallelize the code, we only need to check and process variables that are related to the large-scale input, namely *tracing*, and no additional processing is necessary for other parts of the code. The purpose of tracing, in short, is to find out all the variables in the program that are a copy of, or depend on any part of the big-data input, which are called a mapping of that part of input. It can also be written in a function:

Definition 2. *Mapping: in a piece of source code, if there are two defined variables (or symbols, objects etc.) s and d, where s is a copy or a part of d through assignment (e.g. $s = d$, $s = d[1]$, $s = d[2][3]$), then s is a mapping of d). Also, taking index into account for a large-scale variable, s is an argument of the mapping function of d, whose return value is the index or component (for simplicity, both are denoted as ind with a subscription of the original variable) ind_s that s is dependent on.*

$$\mathcal{M}(d, s) = ind_s$$

Obviously, any variable is a mapping of itself and the mapping relationship is transitive: if s is mapping of d and t is a mapping of s, then t is also a mapping of d. In this sense, a series of mappings can be found throughout the program and form a directed acyclic graph to indicate dependency as in many parallelization works [47, 48]. The piece of code in Figure 3.1, which performs principle component analysis in R [49], is used to illustrate the basic idea of tracing.

3.2 Parallelism with big-data approximation

In data-centric computing, it is usually reasonable to consider that there is one logical input data set for a task, which can be very large. In MapReduce, parallelism is only imposed on data: the original code is assumed to be equivalent to running the same code multiple times with a small portion of data and combining their results. While the assumption may not hold for many programs in statistical analysis and data mining, the scanning of code is required. Thus, a couple of definitions are made as follows:

Definition 1. *Big-Data Approximation: for parallelizing a function $f(\mathcal{D}, ...)$, where $\mathcal{D}$ is the only logical data input of f, only the parallelization of routines that contain $\mathcal{D}$ or its mappings is needed, while the time for other computations is considered as a negligible part in the total.*

program as a data-centric task, the processing times of the two types of routines can be retrieved respectively. 7.1 shows such statistics based on three well-defined algorithms for data-centric tasks: *K-means*,

With *big-data approximation*, the parallelization is simplified for the reduction of data dependence as well as formality. That is, in order to parallelize the code, we only need to check and process variables that are related to the large-scale input, namely *mapping*, and no additional processing is necessary for other parts of the code. The purpose of tracing, in short, is to find out all the variables in the program that are a copy of, or depend on any part of the big-data input, which are called a mapping of that part of input. In other words, the negligible part of computation can be copied over computing units without considerably hurdling the overall performance.

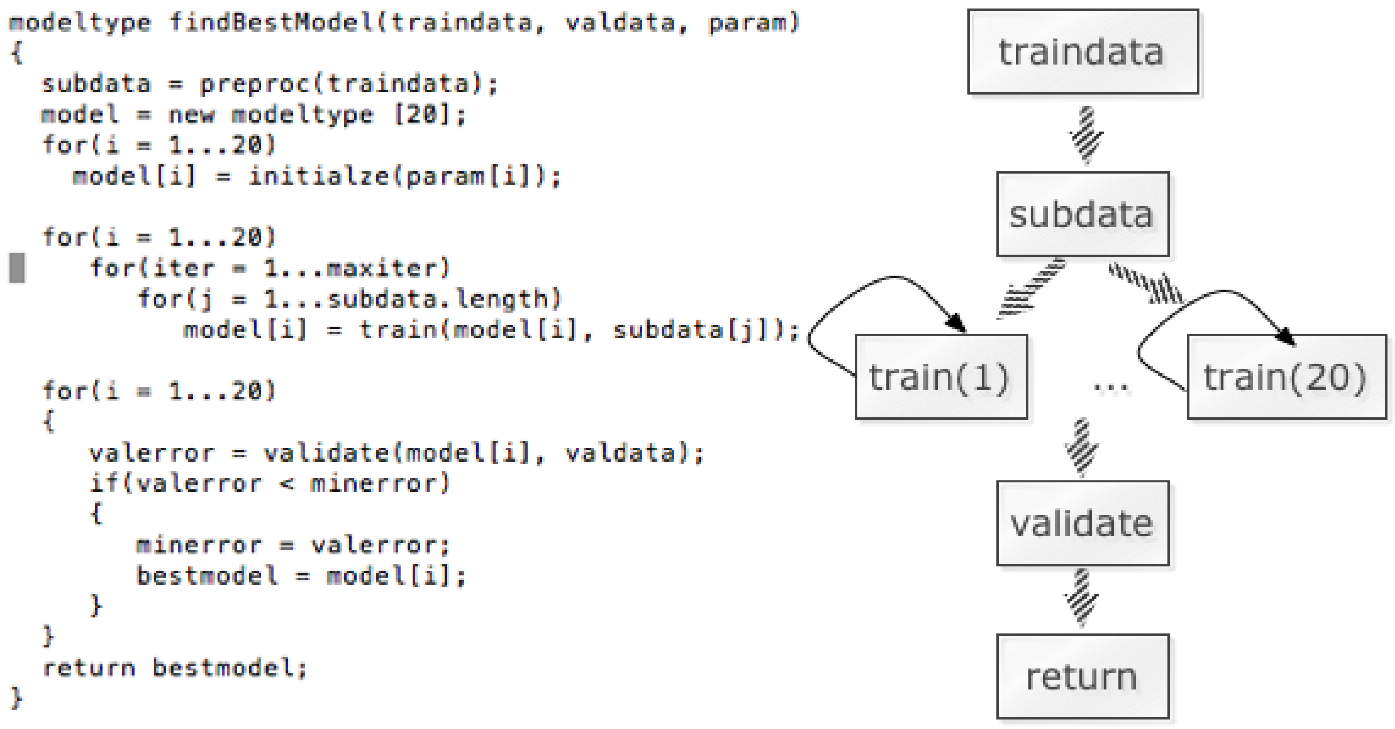
Definition 2. *Mapping: in a piece of source code, if there are two defined variables (or symbols, objects etc.) s and d, where s is a copy or a part of d through assignment (e.g. $s = d$, $s = d[1]$, $s = d[2][3]$), then s is a mapping of d). Also, taking index into account for a large-scale variable, s is an argument of the mapping function of d, whose return value is the index or component (for simplicity, both are denoted as ind with a subscription of the original variable) ind_s that s is dependent on.*

$$\mathcal{M}(d, s) = ind_s$$

Obviously, any variable is a mapping of itself and the mapping relationship is transitive: if s is mapping of d and t is a mapping of s, then t is also a mapping of d. In this sense, a series of mappings can be found throughout the program and form a directed acyclic graph to indicate dependency as in many parallelization works [47, 48].

Then, to establish the dependence graph based on the big-data input and its mappings, it has one starting node while the result of the program correspondingly becomes the only ending node. That is, the dependence graph would be a flow network with a source and a sink, namely *core dependence graph*.

Using a typical routine in machine learning as an example, the difference between the conventional dependence graph (as a directed acyclic graph (DAG)) and core dependence graph is shown in Figure 3.1



```
modeltype findBestModel(traindata, valdata, param)
{
  subdata = preproc(traindata);
  model = new modeltype [20];
  for(i = 1...20)
    model[i] = initialze(param[i]);

  for(i = 1...20)
     for(iter = 1...maxiter)
        for(j = 1...subdata.length)
           model[i] = train(model[i], subdata[j]);

  for(i = 1...20)
  {
     valerror = validate(model[i], valdata);
     if(valerror < minerror)
     {
        minerror = valerror;
        bestmodel = model[i];
     }
  }
  return bestmodel;
}
```

FIGURE 3.1: Code example and core dependence graph

From the core dependence graph, we eliminate all the unrelated variable from the routine as they can be either pre-evaluated or trivial to be duplicate the computation over processors. The convenience brought by core dependence graph exists in two aspects: *i)* As all the dependencies in the flow network origins from the source, the level of dependency is identifiable at any position in the code, which means the graph can be horizontally partitioned into cross-sections and each vertex in the cross-section has a definite dependency set; *ii)* For those vertices who dependency sets have no intersection, they can be executed in parallel. Thereby, the *dependency set* is also uniquely identifiable from the source. So, in order to parallelize the piece of code, a systematic way of extracting the dependency set from each vertex and scheduling the execution sequence is desired, where the execution sequence must follow the dependency restriction while matching the number of computing resources.

Compared to the conventional directed acyclic graph, the dependency set is easier to be defined when a single source is given. Without the single source from big-data approximation, the cost of evaluating dependency is much higher and the routines of parallelization then become complicated. The exact partitioning mechanism of core dependency graph will be fomalized in **??**.

3.3 On-the-fly Tracing for Parallelism Exploitation on Demand

While the parallelization problem has been converted to building/partitioning the core dependence graph and scheduling the execution sequence, the entire parallelization pipeline includes the following stages: *i)* Big-data tracking (building the graph structure); *ii)* Weight estimation (estimating the running time of each vertex of the graph) and *iii)* Dynamic partitioning and scheduling. Stage *i* is in the same style of traditional parallelization which is actually a parser of code and whose implementation is certainly programming-language-specific. Then Stage *ii* only deals with the abstract graph structure where the representation of each vertex and edge is generalized: an estimate of execution time, based on the level of involvement of big data or other knolwedge, is obtained throughout the graph. Based on *ii)*, the weighted graph is then partitioned and an execution sequence can be then generated to be compatible with the amount of available computing resources.

3.3.1 Big-data tracking

As most parallelization techniques involve scanning the code line by line, big-data tracking also means scanning through the code and encloses the abstracion onto a connected data structure (abstract syntax tree for many parsers). So, the core dependence graph should be complete to cover all the contents of the code while simplifying the way of abstraction to make parallelization efficient.

The scan is performed line by line and for each line: each line is evaluated for whether or how much it depends on the user-specified big data input.The procedures is described as follows:

Algorithm 1 Core dependence graph construction

```
Denote A unit of the code as a line or a block (if-block or for-block) as $u_i$
where a single line $u_i$ can be in the form of $u_{i-b} = u_{i-a}$ ($u_{i-a}$ and $u_{i-b}$ are expressions)
Initialize the successor set $S$ as an empty set
where $i$ is in the same order of the original code
add the big data input $D$ into $S$, with the index object "all" and an update list $\{0\}$
for each $u_i$ at line number $l_i$
 if $u_i$ is in the form of $u_{i-1} = u_{i-b}$
  if $u_{i-a}$ involves reading/calling any $s_j$ in $S$
   denote the name of referenced variables as $s_{i_1}...s_{i_k}$
   extract the corresponding index object(s) $\{ind_{i_1}...ind_{i_k}\}$ from $u_{i-a}$ for any reference
   of successors
   if $u_{i-b}$ includes an index itself
    create a self-index object $ind_{s-i}$ for $u_{i-b}$
   else
    create a null self-index object ($ind_{s-i} = null$)
   end if
   if $u_{i-b}$ is not in $S$
    add the variable name in $u_{i-b}$ to $S$ as a new sucessor with an empty attribute list
   end if
   define a tuple $t_i = (l_i, \{(s_{i-1}, ind_{i_1})...(s_{i-k}, ind_{i_k})\}, ind_{s-i})$ as an attribute object
   add $t_i$ to $u_{i-b}$'s attribute list in $S$
  end if
 end if
end for
for each element $s_i$ in $S$ (in the same sequence as they were added)
 for each member $l_j$ in the attribute list of $s_i$ (identified by the line number)
  create a vertex in the graph as $V(s_i, l_j)$
  for each element $(s_{i_k}, ind_{i_k}$ in the index object
   find the largest line number $l_{max-j}$ in the successor attribute list, whose self-index object covers $ind_k$
   create an edge from vertex $V(s_{i_k}, l_{max-j})$ to $V(s_i, l_j)$
  end for
 end for
end for
```

In Algorithm 1, the term of index object or self-index object is repeatedly used as the portion of array data that is represented in a line (for the assignment of the variable itself or used in the assignment of another variable). While index representation is uniform in a certain programming language (e.g. many use "[ ]"), the index extraction here is straightforward and generic. Considering in the actual code, an index can be an arbitrary number within the size of data or irregularly distributed. We define the data structure of index object (including self-index) for minimum description length.

An *index object* is a set of index elements of a certain format. An *index element* can be either a single index or a range, where a range can represent any number of indices as long as they can be arranged in a sequence with a fixed interval. Then, the “merge” operation is defined over the data structure to combine ranges (or single indices included in the range) from one variable. For multi-dimensional array, it makes each dimension independent from others and the “merge” operation is only availble at one dimension if all others are exactly the same.

Thus, the core dependence graph is built and the next task is to properly partition it to exploit parallelism.

3.3.2 Weight estimation

In the core dependence graph, each vertex comes with a weight which indicates the time estimation of the corresponding code section. Following the big-data approximation, the code outside the core dependence graph, which doesn't involve computations regarding the big data input at all, would be estimated as very low (negligible) weight so it won't affect the efficiency of parallelization. Meanwhile, the code can include a function call whose time complexity may not be known to users. So, as a generic case without the other knowledge about the function calls, it is assumed that *the weight of one vertex in the core dependence graph is in proportion to the frequency of the use of big data input and its successors.*

In practice, the parallelization tool attempts to get the exact time complexity of each function call by taking additional information from an external source. If such information is not available, the parallelization program would give an empirical guess. In our experiments, we will show how different guesses affect the system performance.

3.3.3 Partitioning and scheduling on demand

In the previous two subsections, the parallelization task has been transformed to a structured problem:

Given a flow network, where each vertex has a weight that indicates the execution time, and the amount of resources, it is needed to find an optimal schedule such that the total time of executing the routines in all the vertices, without breaking the interdependence, is the shortest. The computation time of one vertex, as the product of execution time and usable resources, is in proportion to its weight.

To address the scheduling problem, graph partition is to be performed in order to create *execution sections* that pass an exact set of codes with their desired resources to the processor. Considering system implementation, the code needs to be partitioned into sections that a synchronization is needed after the execution of each code section. Then, for a single section, the shortest execution time is desired for non-interdependent tasks. The rest of this subsection is mainly the algorithms with regard to finding the optimal mechanism of dividing code to sections (namely *horizontal partition*) and the corresponding in-section parallelization approach (namely *vertical partition*).

3.3.3.1 Horizontal partition

A code section can include multiple vertices from the core dependence graph but they must be interdependence-free from each other. So,the fewer sections, the better for scheduling. In other words, a section should cover as many vertices as possible, as long as there is no interdependence between any vertex in the section. Algorithm 2 details the partitioning mechanism, which is in a similar manner to critical path method.

Algorithm 2 Horizontal partitioning

```
**Denote** A vertex of core dependence graph as $v_i$
where a vertex includes the successor set (with their attribute lists) along with
starting and ending line numbers in the code
**Initialize** an empty set of subgraphs
perform a **breadth-first traversal** to retrieve a **prefix list** $l_{pre-v_i}$
after source to each vertex $v_i$ generate a vertex list $l_v$ in the
ascending order of the number of prefixes
**while** $l_v$ is not empty
  **find** all the vertices that has no prefix in $l_{pre-v_i}$ through $l_v$ as a set $vs_{pre}$
  **for** any vertex $v_j$ in $vs_{pre}$
    **add** $v_j$ into the current subgraph $Sub_k$
    **remove** $v_j$ from any other vertices out of $vs_{pre}$ from their prefix list
  **end for**
  $k = k + 1$
**end while**
```

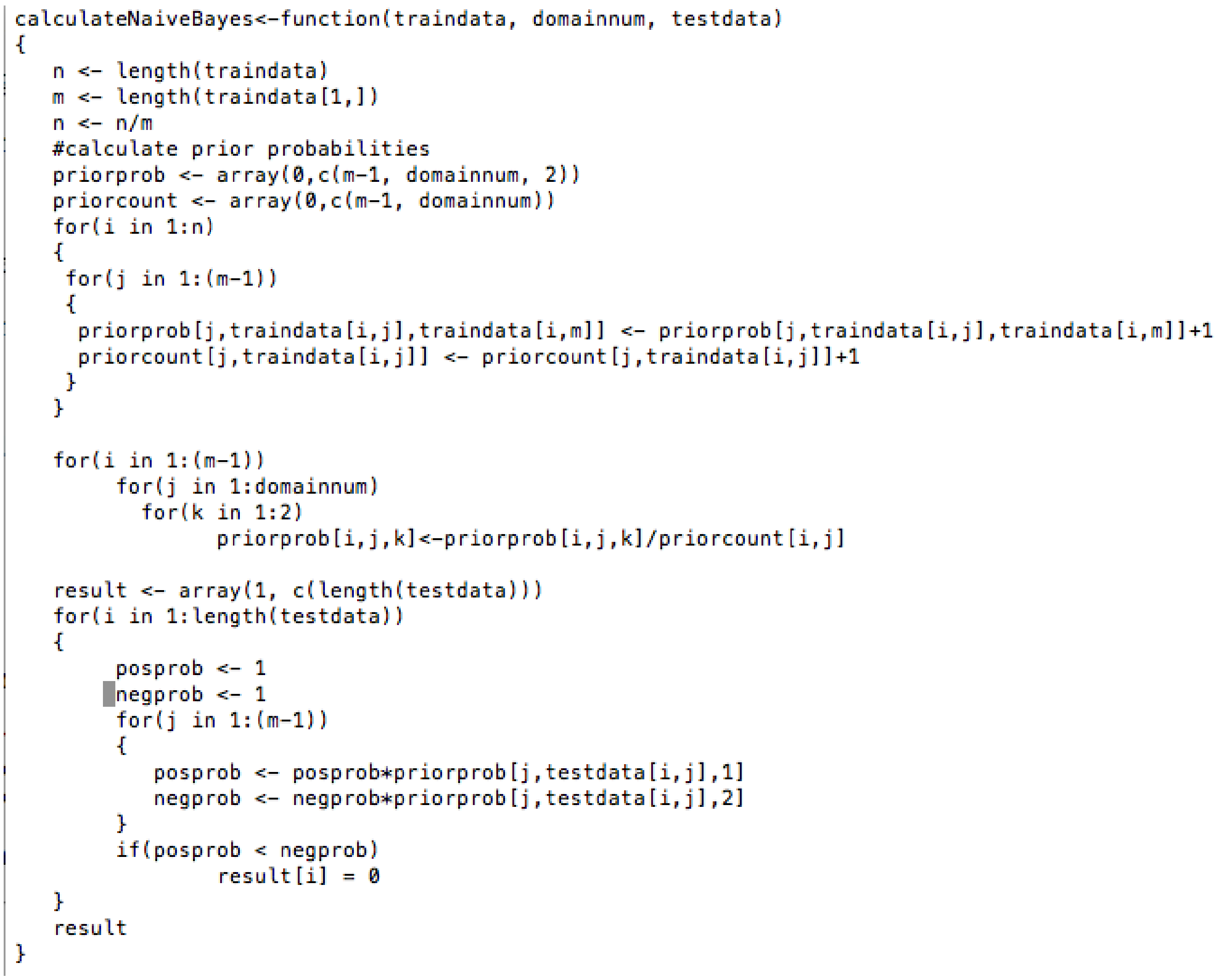
The time complexity of the algorithm is $O(V + E)$ as it requires a scan of all the vertices and also traverses all the edges in the course of removing a certain vertex from the prefix lists of others.

3.3.3.2 Vertical partition

An individual code section, whose vertices don't depend on each other, is executable with the results from the last section throughout the code. So, scheduling a section across computing units becomes a standalone task without considering the context. So, vertical partition, based on the vertice in a section, means the generation of executable parallel code with the given computing resources. While one vertex represents a single task that can be parallelized where the weight indicates its execution time, the in-section parallelization can be straightforward that different tasks can be scheduled one by one with each task executed by the maximum available computing resources. Meanwhile, many sections include loops with a considerable number of iterations. Mapping the code section to processors in iteration level is the focus of vertical partition.

In this step, the bidirectional tracing is required in order to exploit the relationship between "result" and "data". While conventional parallelization usually doesn't differentiate the two concepts in scanning the code and tries to process all the variables appearing throughout the code. The separation scheme still puts parallelism on data and take effect at a high level to avoid the complex processing.

So, in the spirit of MapReduce, the focus is still to find how data can be placed to multiple processing units. It starts with some real-world code segments:



```
calculateNaiveBayes<-function(traindata, domainnum, testdata)
{
   n <- length(traindata)
   m <- length(traindata[1,])
   n <- n/m
   #calculate prior probabilities
   priorprob <- array(0,c(m-1, domainnum, 2))
   priorcount <- array(0,c(m-1, domainnum))
   for(i in 1:n)
   {
    for(j in 1:(m-1))
    {
     priorprob[j,traindata[i,j],traindata[i,m]] <- priorprob[j,traindata[i,j],traindata[i,m]]+1
     priorcount[j,traindata[i,j]] <- priorcount[j,traindata[i,j]]+1
    }
   }

   for(i in 1:(m-1))
        for(j in 1:domainnum)
          for(k in 1:2)
                priorprob[i,j,k]<-priorprob[i,j,k]/priorcount[i,j]

   result <- array(1, c(length(testdata)))
   for(i in 1:length(testdata))
   {
        posprob <- 1
        negprob <- 1
        for(j in 1:(m-1))
        {
           posprob <- posprob*priorprob[j,testdata[i,j],1]
           negprob <- negprob*priorprob[j,testdata[i,j],2]
        }
        if(posprob < negprob)
                result[i] = 0
   }
   result
}
```

FIGURE 3.2: A piece of example code for naive Bayes model (binary classification) with training data and test data

The naive Bayes model training for binary classification includes two for-loops, which form the core part of the function. The function takes three inputs: training data set, the number of discrete values for each field (domainnum) and the test data set and outputs an array as the classification result of the test data. Regarding "traindata" in the argument as the big data, the function itself can't directly work with MapReduce but the same idea can apply as shown in Figure 3.3.

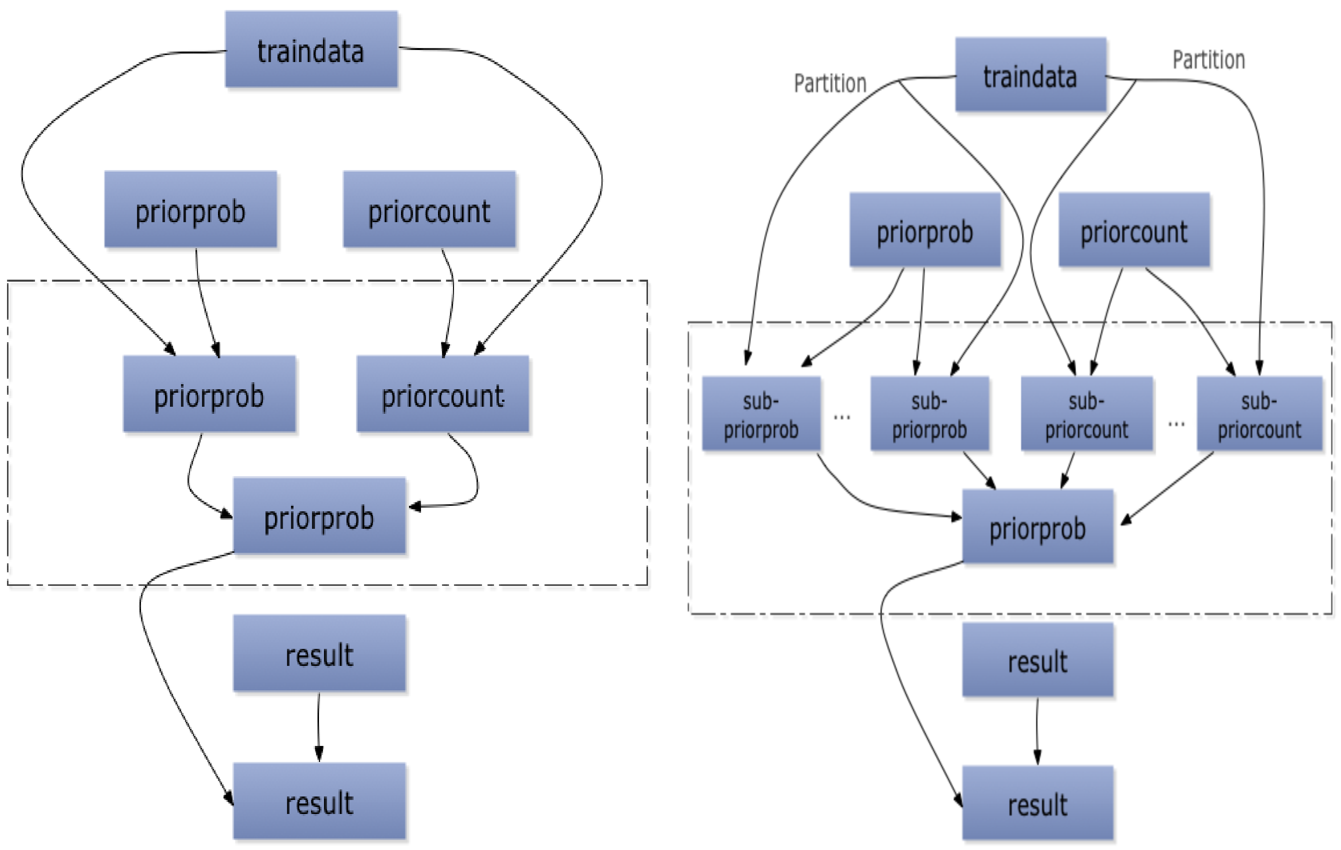


FIGURE 3.3: The core dependence graph of naive Bayes model training: left: graph before partitioning; right: data partitioning on graph

The left graph of Figure 3.3 shows the core dependency with horizontal partitioning, while the area within the dotted framebox is identified as the time-consuming part. Then, the vertical partitioning would detect the parallelism inside the for loop: in this example of naive Bayes training, the for-loop can be partitioned by mapping any chunk of data to a processor unit and combining all the results by summing them up.

On the contrary, conventional parallelization would mark the variables ”priorprob” (prior probability) and ”priorcount” as dependent on ”traindata”. While the indices of ”priorprob” and ”priorcount” applied in the value assignment completely depends on the value of a certain element of ”traindata”, it would be considered irregularity and hard to be recognized as a pattern of parallelization. In the spirit of big-data approximation, only ”traindata” is identified for read and there is no actual conflict inside the loop: ”priorprob” and ”priorcount” can be duplicated across processors and the use of "traindata" in the for loop follows a regular sequence: in the order of its first dimension (1-n) and for each 1st-dimension element, its fields is iterated.

Although this example turns out to be applicable for MapReduce-style on the for-loop, a series of code scan operations are adopted to the for-loop for parallelism detection without human knowledge. That is, lower-level processing on loops are then performed on vertices. While lots of related work in parallelization puts an emphasis on loop parallelization, we take advantage of big-data approximation that the size of the big data is known, which means that the value of indices over the big data input and its successors are accessible: the "cycle" is exploited to indicate the dependence inside the loop so it comes down to the level that we know which other iterations each iteration is dependent on. So, an assignment, based on the dependence in cycle, can be then delivered to each computing unit. This "ad-hoc" analysis asks for almost no assumption in the form of loop (e.g. do-all loop) so becomes more generic.

Definition 3 *A cycle in a loop is the non-separable pattern that the inter-dependence within a cycle from index $i_1 + km$ to $i_1 + (k+1)m$, where k is any non-negative integer (within the execution of loop), is the same. m is the length of the cycle if the condition is satisfied.*

Cycle detection is actually about building a sub-structure like building a core dependence graph within the loop. The loop iterator appears as a *virtual variable* that is used to identify pattern and *collector*, which only appears as the outlets of data and whose content is never used in the loop, is also identified to be distinct with ordinary data input and successors.

Then, the task of vertical partition is turned into finding separable paths over the cycle. In other words, a cycle is the real iteration with irregularity removed and it would make the latter operations valid. One cycle, as an atomic unit in the parallelization, can acquire all the data it needs from itself or previous cycles.

Algorithm 3 Cycle detection in the loop

```
Initialize find the index of the first iteration of the loop $ind$; set iteration range $R_{iter} = 1$; and also find the
Set two empty sets $S_{r0}$ and $S_{w0}$ for tracing data indices for read and write respectively
for each line in the first iteration
  Parse the current line of code
  if the line includes any read or write of the big data input
    add the index of data to $S_r$ or $S_w$
end for
Update the current iteration and index in sequence
while true
  Parse the next $R_{iter}$ iteration and get two similar index sets $S_{r1}$ and $S_{w1}$
  if $S_{r1} - S_{r0}$ is not empty and $S_{w1} - S_{w0}$ is not empty and $S_{r1} - S_{w0}$ is not empty
    return $R_{iter}$
  else
    $S_{r0} = S_{r0} + S_{r1}$
    $S_{w0} = S_{w0} + S_{w1}$
    $R_{iter} = R_{iter} + 1$
  end if
end while
```

The set subtraction as in $S_{r1} - S_{r0}$ is to verify whether one index set includes all the indices in the last iterations. If it does, then the result is not an empty set. Implicitly, it sets a default that every index in the big data is to be either read or written so that the consecutive indices can be always found throughout the scan. However, even if the condition doesn't hold, it would not result in any conflict as any missing index would be simply skipped and doesn't affect the algorithm. So, to make the program capable of data parallelism, a straightforward way of partitioning is to cut the data into chunks according to consecutive indices. Actually, as long as the a chunk of data can be expressed with a pattern, it can be all handled in our platform. Moreover, the scan establishes the relationship between data indices and result, and constructs the parallelism that maps a subroutine to the result to the corresponding portion of data and recognizes the operator in combining sub-results, namely *data-result parallelism*, which provides more flexibility than MapReduce. The combination function is extracted from the operation list.

In the same way, for nested loops, the concept of cycle can work on any level of loop as long as such a loop iterates with the progression of big data indices, where all lower-level loops can be regarded as a whole. Also, a hierarchical cycle built from multiple levels of loops is feasible with data dimension reduction. More details will be provided in Chapter 4.

While conventional dependency usually doesn't separate "result" from "data", we elaborate the concept of data-result parallelism and differentiate it from the conventional parallelization. As it is also assumed that only one result at the end of program as well as the termination of a for loop, the data can be partitioned under certain conditions that is verifiable from the source code.

The two checks are performed in the code scan: *mutability check* and *associativity detection*: in mutability check, it checks whether two cycles are swappable by inspecting the operator from data to result; the associativity test, if immutability is detected, then identifies whether operations in cycles can be combined with associative property. These two checks will define the data parallelism in such a new paradigm.

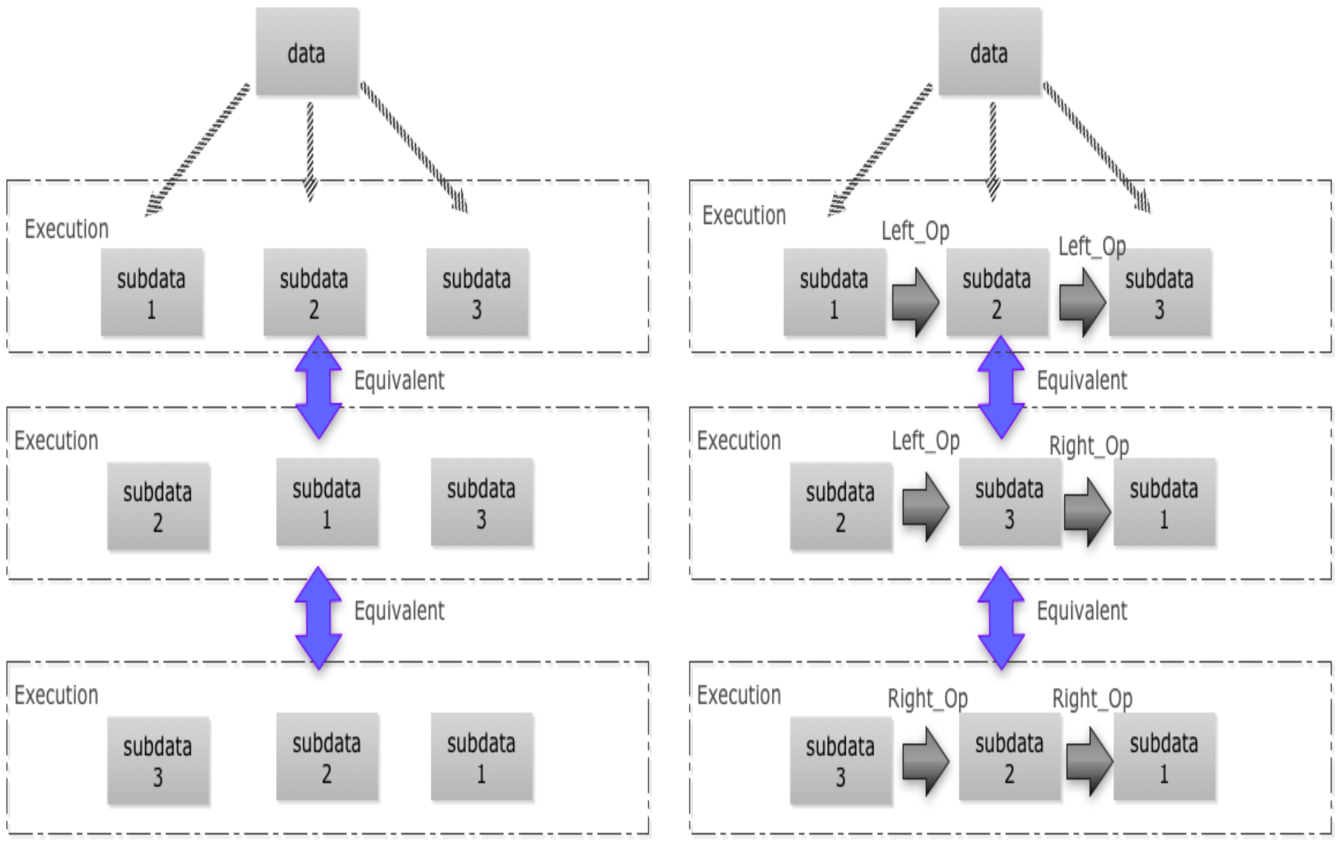


FIGURE 3.4: The illustration of mutability and associativity in executing a routine on data: left: mutability; right: associativity

The two tests, to a large extent, is about the nature of the operator from data to result in each cycle. So, the following algorithm illustrates how the operator is identified from code scan. The operator, as a concept in the loop, consists of multiple basic type-specific operators with regard to each write to the result or its mappings. The following routine illustrates how to collect all the basic operators for later identification.

Algorithm 4 Operator collection in the loop

```
Prestep: find the previous node(s) $V_{pre}$ right before entering the loop and the node(s) $V_{next}$ after the loop.
Initialize an empty operator list $L_o$ and an empty set $S_{map}$ for storing mappings
add all the mappings in $V_{pre}$ to $S_{map}$
for each line of one iteration
  if a mapping is found and $S_{map}$ doesn't contain it
    add the new mapping to $S_{map}$
end for
textbffor each line of one iteration
  if this line includes a "write" to one of the mappings
    add the rule into $L_o$ in the format: mapping name, type, operator
for
```

Having the operator list L_o, the next step is to identify the nature from elements of the list and their relationship to the final result. Hence, the following two algorithms are derived. In order to make the nature of operation list preserved, it requires all the basic operations maintain the same throughout the loop.

Algorithm 5 Mutability check

```
for each operation in the list
  if the target variable of this operation is an output node of the loop
    if the operation is not order-free
      return: mutability detection fails
    end if
  end if
end for
return: mutability detection succeeds
```

Mutability check can lead to MapReduce-style parallelism in a smaller scale: the data corresponding to a mutable cycle can be assigned to a processor and then the result can be collected from processors and combined with a reducer function. On the other hand, associativity can bring parallelism with only one restriction on the reducer function: the reducer function must maintain the same order as mapping when collecting results from different processing units. For associativity detection, a simple check based on data type may still work, but a direct test can be performed based on its definition: all the basic operations can re-organized as chains without rolling back, then associative rules can apply.

Algorithm 6 Associativity detection

```
Initialize an array of linked lists $L_{op}$ where the entry of each element is initialized by each basic
operation in the list
for each operation in the list
  if the basic operation is not associative
    return: associativity detection fails
end for
for each line in the cycle
  parse the line
  if the target of writing is also an entry of $L_{op}$
    if the read list includes an entry of $L_{op}$
      merge two entries as a linked list and mark the original entry as empty
    else if the read list includes an empty entry of $L_{op}$
      return: associativity detection fails
    end if
  end if
end for
return: associativity detection succeeds
```

Passing such two tests ensures that the data involved in the loop can be partitioned by index without affecting the result. Between the two tests, mutability is preferred, if passed, over associativity: mutability means the nature of the task enables order-free partitioning over data and the combination function is thereby order-free too. Although the flexibility in data partitioning may not bring an advantage of time gain (for given resources), an order-free combination function can reduce the time in collecting results if an ordered operation takes more than constant time. An even more sophisticated case is that the combination function is mutable while the the entire operation set can only pass associativity test. In this case, an additional mutability test can be performed on the subset of operation list: those which directly write values to the mappings that appear as the node after the loop.

Using horizontal and vertical partitioning, a piece of given source code can be preprocessed and transformed in a new version that can be executed in parallel on large computers. Up to this point, the on-the-fly tracing mechanism can scan a piece of code, which may represent a series of complex routine with a big data input, for parallelization. While the main effort is usually exerted on loops, the on-the-fly tracing maps the loop iteration to data index and tries to separates the dependency on the big data input (while all other computations are duplicable over processors). The mechanism works on a higher level than conventional parallelization techniques and avoids the formality brought by exploiting the full range of dependency or converting the code to another form in an abstract level.

At the level of code parallelization, the exploited "data-result" parallelism, based on mutability and associativity check, ensures that the data involved in a loop can be grouped without affecting the legibility of results. In practical implementation, there is also lots of attention paid to lower-level processing of code. Prior to discussing some of the details in Chapter 4, the next section is about optimizing the parallelization performance for the given computing resources.

3.4 Resource-Aware Workflow Optimization

The parallelization works on the compiler side to generate efficient code. On top of that, the scheduling problem, which represent the

3.4.1 Scheduling with Big-Data Approximation

Provided the aforementioned approach, it already converts the parallelization problem to the workflow level: a piece of code is partitioned to sections that can be independently executed on any processing unit and its result is to be passed and combined. So, a task descriptor becomes a generic abstract unit for a code section or a sophisticated module in scientific computing. So, the scheduling problem can then be defined as follows:

There are a set of tasks that need to be executed. All of them directly or indirectly depend on a large-scale data set. Given limited computing resources on multiple clusters and the execution of each task is subject to the availability on the machine, the optimal operational sequence of execution is *desired in order to minimize the delay time, which includes waiting in queue and the non-trivial I/O transmission for data.*

While linear scheduling [45] is an option for this problem, the time for data transmission is also non-trivial and needed to be minimized, which means data synchronization could add significant cost. Then, the first principle becomes maintaining the data locality for time-consuming routines. The following greedy-based scheduling algorithm is proposed, which matches the result in [45]: longest task comes first.

- *Pre-step 1*: Set up a current scheduling queue and initialize it as an empty set. Estimate the time (computational cost) and requested number of resources (no more than the amount of available resources) for each task and add such two attributes to the task. Also, an insertion range is defined associated with the task too, which means the available range for inserting it to the current scheduling list, bearing the constraints brought by data dependency. The insertion range is initialized as the entire scheduling queue.

- *Pre-step 2*: All the tasks are in the candidate list at the beginning. Maintain a variable-resource list that records the location (which processor) of each mapping with the size of the mapping.

- *Step 1*: Find the task with the highest weight, denoted as w_{max}, schedule it at the first available place with all the requested resources. As for the exact processors it should be assigned to, an embedded greedy algorithm is adopted to select those processors which have the largest size of data that it is dependent on.

- *Step 2*: Add the currently highest-weight task T_i into the scheduling queue. Update the insertion range of all other tasks. Remove T_i in the candidate list while returning a dependency list of T_i to indicate which tasks its data depends on.

- *Step 3*: Update the weight of all the tasks in the dependency list of T_i by adding w_{max} to their weights

- *Step 4*: Go to *Step 1* until the candidate list is empty.

This scheduling algorithm assumes that the time-consuming task also indicates the generation of the largest data. So, while the in-task synchronization is not considered as a factor, assigning the most time-consuming tasks with the most resources implicitly reduces the data synchronization between tasks.

3.4.2 I/O Performance and Its Estimation

In the real-world scheduling, which usually occurs in grid or cloud computing, a task may not be immediately scheduled because of the public queue that includes tasks from other users is not available. Waiting is then necessary. However, in this setting, there are usually multiple resources, such as different clusters, available for job submission. It leads to dynamic resource selection. To utilize the resources while still reducing the cost of large data transmission, it would be desired to measure the actual I/O performance in the specific machines. The measurement would also alleviate the possible failure of scalability due to excessive communication time between processors (usually on a cluster).

In this scenario, a pre-evaluation of the task is necessary to anticipate the possible bottleneck. As long as the code is accessible, it would be desirable to explore all the functions and get information out of it: e.g. disk I/O frequency with respect to the size/dimension of data, the need of data transmission if the resource includes multiple nodes on a cluster. In real-world large applications, there are a lot of factors that have an effect in actual performance. So, it would be desirable to use the on-the-fly tracing approach as a helper to find where to parallelize the task, and then come up with a practically best non-trivial scalable solution using human intelligence, which may contain lots of trade-off in an engineering setting.

3.5 Performance Benchmarking

In this section, the experiment results are presented. The experiments include the scaling test that measures parallelization performance on both multicore and commodity cluster architectures. Four typical algorithms in machine learning, with multiple data sets, are selected.

3.5.1 Test cases and data sets

To illustrate the effectiveness of the framework, four typical examples of machine learning algorithms are selected: naive Bayes classifier, k-means algorithm, expectation maximization (EM) and logistic regression.

Naive Bayes classifier: naive Bayes classifier is a probabilistic model that applies Bayes theorem with a strong assumption of the independence between fields in a data set. Naive Bayes is a parametric model but all the parameters are directly estimated based on data frequencies from the training data set, and then calculate the posterior probability for each possible class for a test data record. The parallelization is mainly exerted on the part of iteratively retrieving frequencies for all the field values from the training data.

k-means algorithm: k-means algorithm is a common method for clustering in which the number of centroids is specified in its input as k. While the two major steps of the iterative algorithm require a synchronization for the entry of next iteration, the parallelism only exerts on the stage of finding the set that every data record belongs to (namely, assignment step) of each iteration. It is an order-free operation.

Expectation Maximization (EM): EM is also a parametric method that aims to estimate the parameters with maximum likelihood in Bayesian inference. Similar to k-mean algorithm, the parallelism falls into the E-step only, which obtains updated state estimates by iterating the current parameters through data. The M step again requires an update to all the parameters used in the next step.

Logistic regression: Logistic regression is commonly used to make predictions for a binary of multi-category label. The training of model can pass the associativity test, so the increment calculated from a portion of data can be added to the model itself. Also, the combination function is an order-free one.

The data sets used in the experiment are from UCI Machine Learning Repository[2] . Several datasets, such as Adult, 1990 US Census, and Census Income are selected. In order to examine the relationship between performance with increasing size of data, the data records are duplicated for constructing a new larger data set without essentially changing the computing task.

The algorithms are re-written in R to be in a format that is friendly for the ”on-the-fly tracing” framework: the data input is identifiable as one of the arguments, and there is another input that suggested the available computing resources (e.g. 8 cores, 4 nodes with 8 cores on each).

3.5.2 Experimental results

First, the examples are parallelized and executed on a multicore architecture of two Quad Core Intel Xeon 2.33GHz 64-bit processors and 8 GB of memory. Figure 3.3 shows the scaling performance: the solid line shows the average speed-up compared to the time executed with a single core; the two dotted lines give the worst and best speed-up in around 5 tests in a row.

Figure 3.5 shows that the parallelization works well on multicore with a considerable amount of speed-up: all the cases got around 7 times faster with 8 cores than running on a single core. The fewer the cores are, the more it approches an optimal speed-up. The variance in different runs doesn’t becomes a major factor.

Then, the effect of data size is examined. By constructing larger data sets with proportional sizes of 2, 4, 8, 16 and 32 times greater than the original, the execution time with multicore is again compared with a serial version with the same input data set.

Moving from multicore to a cluster with many nodes, the data transport between nodes can takes time. From the perspective of parallelization, the time doesn’t count into the experiment but is considered preprocessing: the data is replicated across different nodes before the program starts. Throughout its execution, no data is explicitly exchanged between nodes but the intermediate results can be synchronized with it is necessary.

In Figure 3.7, the speed-up in a cluster is illustrated with up to 8 nodes (64 cores in total). Overhead can be identified when the number of cores increases. In this setting, inter-node communication is

[2] http://archive.ics.uci.edu/ml/

handled with MPI (Message Passing Interface) while there are options for parallelization within a node such as multiprocessing (via system call), multithreading and OpenMP.

The above scaling tests illustrates the effectiveness of parallelization, so it can then be moved to larger-scale data-centric applications.

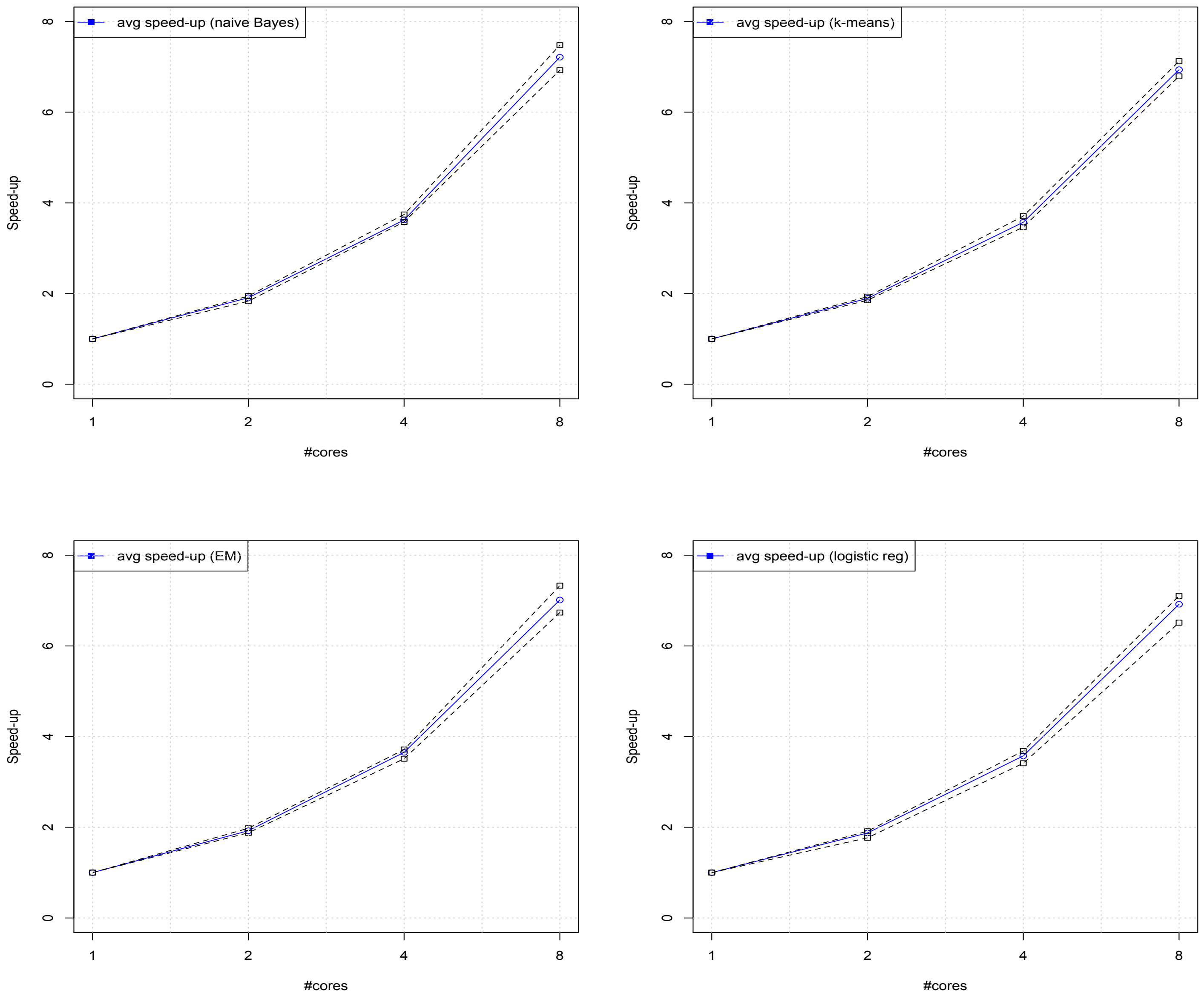
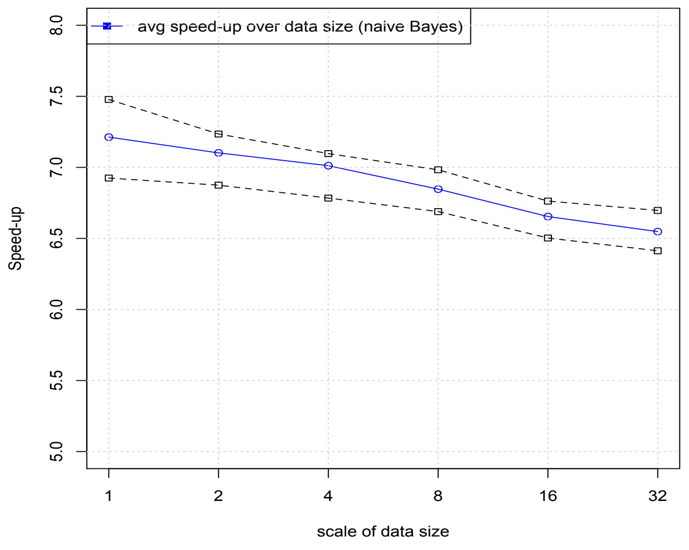
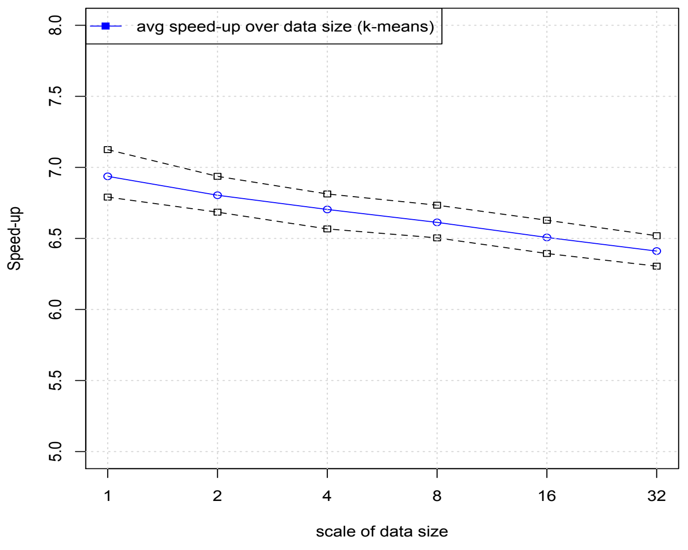
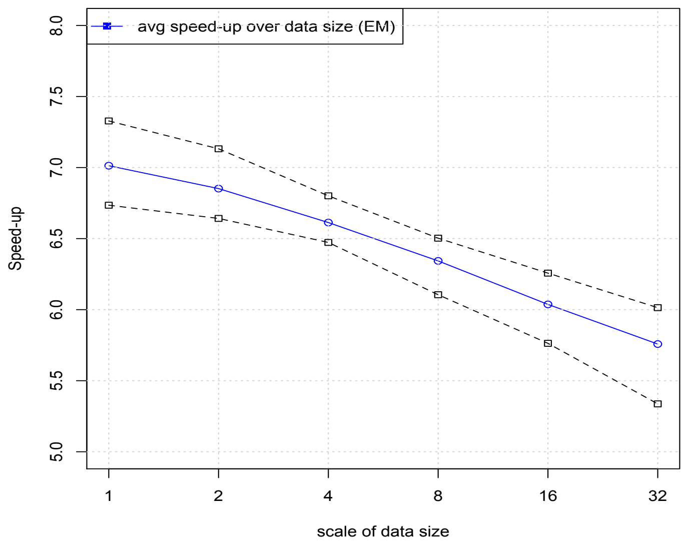


FIGURE 3.5: Scaling test for parallelization on multicore: topleft: naive Bayes; topright: k-means; bottomleft: EM; bottomright: logistic regression







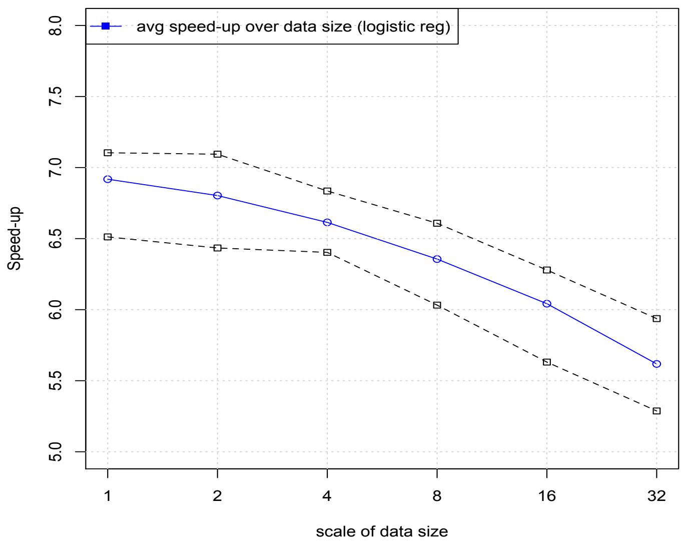


FIGURE 3.6: Scaling test over data size on multicore: topleft: naive Bayes; topright: k-means; bottomleft: EM; bottomright: logistic regression

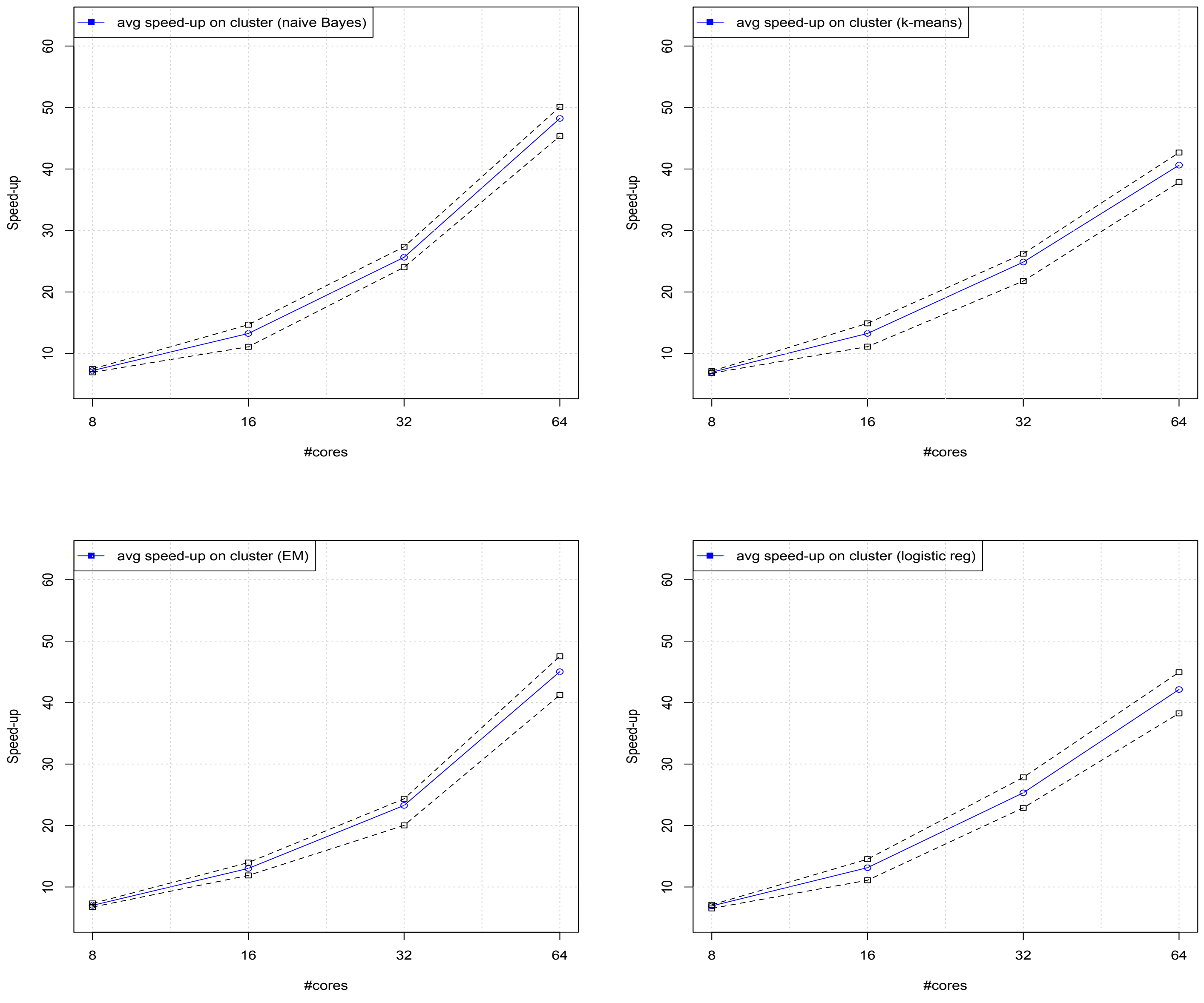


FIGURE 3.7: Scaling test on cluster: topleft: naive Bayes; topright: k-means; bottomleft: EM; bottomright: logistic regression

Chapter 4

Framework Implementation

To implement the framework as mentioned in Chapter 3, it includes much feature on the compiler side for parallelization, while it also requires scheduling on a higher level.

4.1 Design and Platform-Specific Issues

The structure of core dependence graph is universal for all the programming languages with regard to the data flow. Its implementation, however, varies. Considering R [49] becomes a very popular domain-specific programming language for statistical computing, data analysis and data mining, the implementation starts from R. In order to impose higher-level data structure to R, low-level functions must be implemented before hand, including scanning the source code to identify an expression, a line of code (as R doesn't require a semicolon as line separator), and a block (e.g. for loop). These language-specific functions are not yet complete in R so the implementation starts from making R capable of recognizing a specific variable and its status in a line (e.g. whether read-after-write/write-after-write conflict is in the line).

More specifically, a parser is first needed for identifying a line, where all the read and write operations can be recognized. Meanwhile, all the variables need to be recorded. Then, an additional scan is performed to establish the data structure of core dependence graph.

4.2 Parser

Then a self-defined parser is needed in the framework. Every single line of code is to be identified by the legibility of syntax. Three parts can be included in a line: input expression, output expression and the assignment symbol. In R, a single line with only one element is also allowed. So, a stack is used to trace the status of scanning and the piece of code is then segmented by lines.

To the next level, each variable mapped from the big data input is to be recognized too: provided the expression-level identification, a tree structure can be built from the second scan. While no type is needed to be declared at variable initialization, the tree structure, and thereby the core dependence graph doesn't have any associated type either. At this moment, all the loops are considered blocks and the if-statements are scanned to include the most time-consuming part.

Then, for the convenience of adopting the algorithms in Chapter 3, some of the data types are desired at the stage of cycle detection and mutability/associativity tests: in particular, the big data input is identified as a multi-dimensional array and its index is considered to be known to the program in order to identify the dependency inside a loop. However, the possible irregularity of the code can elaborate this task: as index is an important part of the parallelism, there are two settings for its special handling.

range object is used to express a range efficiently. The level of irregularity of a range indices would determine the expression length of such an object. A basic range expression is a series of numbers with a fixed interval in between. The combination of multiple intervals, including a single value, can form any range of indices.

index tracking is for handling irregularity in the code where an index is not directly available as an integer. While it can be represented by a variable and indirectly mapped to a specific value, a tracking would be helpful in this case. Furthermore, there is one situation that the value of index can't be tracked: an index is dependent on the value of input data. While such a case requires the value of data input to be bounded, related information would be desirable from input. However, on the other hand, it can be substituted by the index of the data input, which won't affect the parallelism on data at all.

4.3 Nested loops and embedded function calls

Nested loops are challenging in conventional parallelization as it complicates the dependency from iteration to iteration. In on-the-fly tracing, only one single data input is to be traced, so the complexity would not increase with iteration or level of loops. In multicore architecture, the program chooses one level of loop for parallelization (usually with the one going through the most indices). So, it is still a matter of index tracking that defines the parallelism.

Embedded function calls become another issue but it is common in practical programming. Then, the macro level and weight estimation can be taken advantage of: once the code of the embedded function call is accessible, it can be regarded as a block. and then the processing would be similar to horizontal and vertical partitioning: it is treated as a possibly time-consuming for loop at first for horizontal partitioning, and then the parallelization of scanning itself can be added back to the upper-level function with a weight.

4.4 Code generation

After parallelism exploitation, the execution of the source code would be transformed to another way: the original code is partitioned to sections where each section is capable of being independently executed. For instance, a for loop itself can become a function with the data and indices as the input. While only the computation related to the big data input is considered non-trivial, all other subroutines is duplicable over processors: in a multicore architecture, it is preferred to have a preprocessing function for all those "trivial" calculations without worrying about excessive details.

A section is then called by the pre-generated code template. The code template differs from parallel platforms: multithreading and MPI templates are included in the framework. The advantage of R's interoperability enables the call of an arbitrary function using multithreading. For MPI, it includes the routine of sending results across processors for synchronization in a type-specific manner. Then, with the position for synchronization identified by on-the-fly tracing, the synchronization routines can be inserted into the template.

The modulization of function also gives some flexibility to hybrid parallel programming with both MPI and multithreading. In a cluster, such a template is applied for fully utilizing the computing resource. Figure 4.1 is an example of the code template for multithreading.

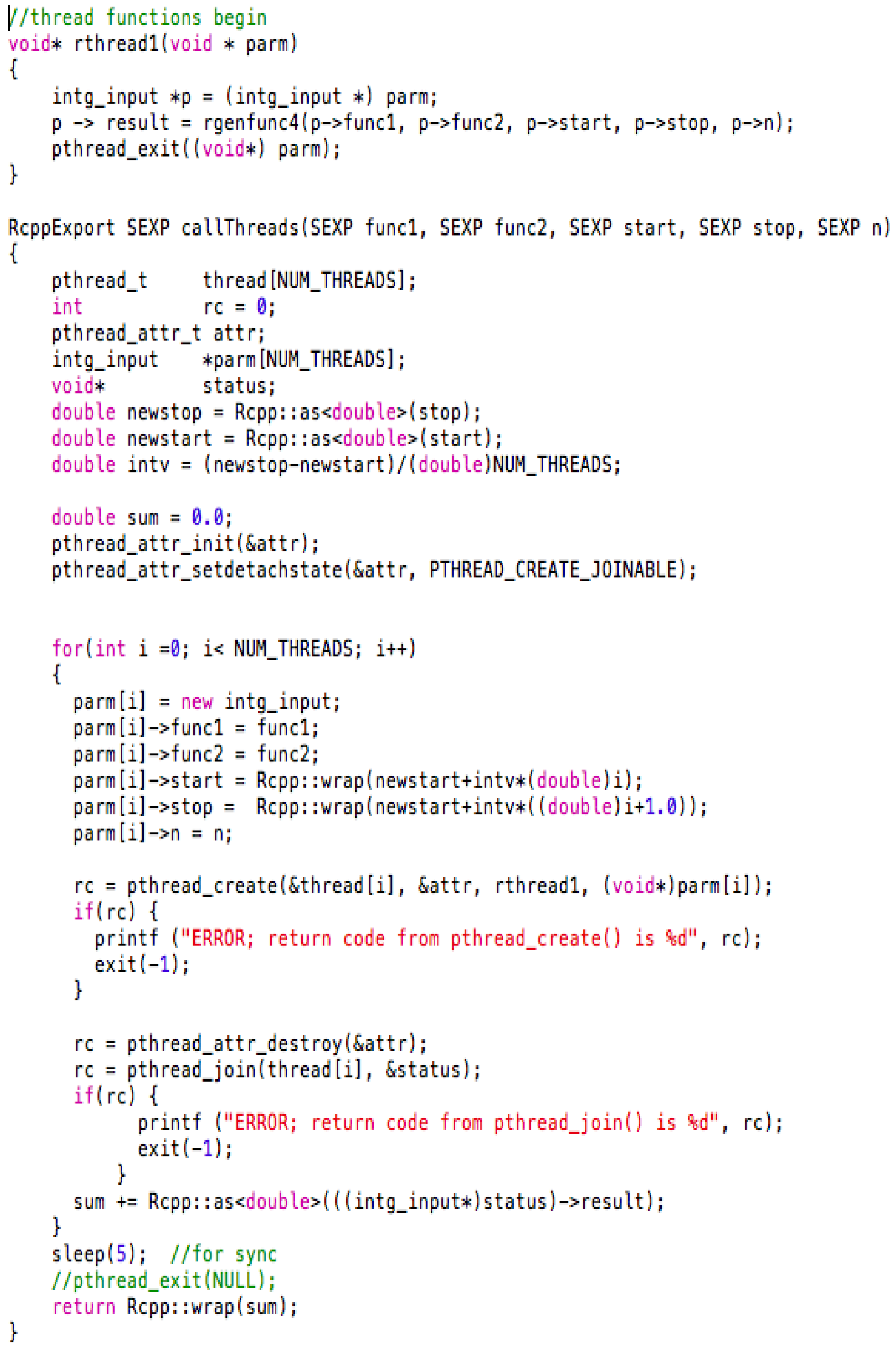
4.5 Put It All Together

Based on the above description, the entire workflow for parallelization can be established as shown in Figure 4.2. The solid frameboxes represent the steps in parallelization and the other boxes with a dotted-line suggests the status of source code at each stage.

While parse functions on the source code as a rough level of scan, symbol processing means tracing all the big-data-related variables including necessary preprocessing regarding data indices. While the core dependence graph is built upon symbol processing, horizontal partitioning can convert code into sections along with weight estimation. At this level, weight estimation is first based on the level of involvement of big data. Then, a detailed check, especially on for-loops is then adopted: vertical partitioning finds the way that a loop is parallelized and this completes the weight estimation.

Based on the available resources, a schedule of the code sections is generated and can fit in the corresponding code template. Finally, the code templates embed all the code sections and become ready for parallel execution.

Although the implementation in this work is mainly in R (including parser, symbol processing, and partitioning) and C/C++(code templates as backend), the same workflow can be implemented in other programming languages too. A generic framework would include most of the language-specific syntax and coding styles for variable tracking and type-specific handling.



```
//thread functions begin
void* rthread1(void * parm)
{
    intg_input *p = (intg_input *) parm;
    p -> result = rgenfunc4(p->func1, p->func2, p->start, p->stop, p->n);
    pthread_exit((void*) parm);
}

RcppExport SEXP callThreads(SEXP func1, SEXP func2, SEXP start, SEXP stop, SEXP n)
{
    pthread_t      thread[NUM_THREADS];
    int            rc = 0;
    pthread_attr_t attr;
    intg_input     *parm[NUM_THREADS];
    void*          status;
    double newstop = Rcpp::as<double>(stop);
    double newstart = Rcpp::as<double>(start);
    double intv = (newstop-newstart)/(double)NUM_THREADS;

    double sum = 0.0;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);


    for(int i =0; i< NUM_THREADS; i++)
    {
      parm[i] = new intg_input;
      parm[i]->func1 = func1;
      parm[i]->func2 = func2;
      parm[i]->start = Rcpp::wrap(newstart+intv*(double)i);
      parm[i]->stop =  Rcpp::wrap(newstart+intv*((double)i+1.0));
      parm[i]->n = n;

      rc = pthread_create(&thread[i], &attr, rthread1, (void*)parm[i]);
      if(rc) {
        printf ("ERROR; return code from pthread_create() is %d", rc);
        exit(-1);
      }

      rc = pthread_attr_destroy(&attr);
      rc = pthread_join(thread[i], &status);
      if(rc) {
            printf ("ERROR; return code from pthread_join() is %d", rc);
            exit(-1);
          }
      sum += Rcpp::as<double>(((intg_input*)status)->result);
    }
    sleep(5);  //for sync
    //pthread_exit(NULL);
    return Rcpp::wrap(sum);
}
```

FIGURE 4.1: A code template for multithreading with R

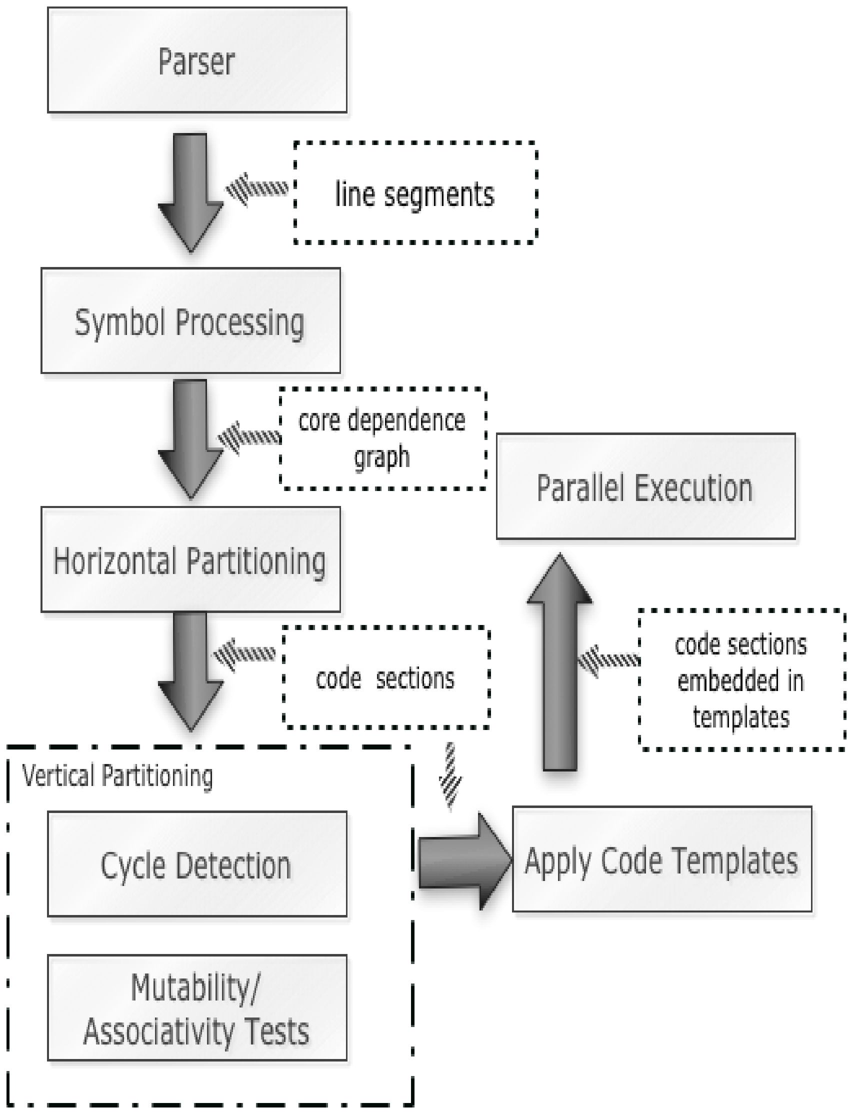


FIGURE 4.2: The modules with source code status in on-the-fly tracing

Chapter 5

Application I: Pattern Detection from Hurricane and Storm Surge Simulations

The tremendous impact of hurricanes in the Gulf of Mexico, represented by Katrina and Rita in 2005, and Gustav and Ike in 2008, makes it important to develop effective methods for making hurricane-related predictions, especially for the storm surge heights from location to location. Due to the large scale and complexity, physics-based simulations now become the primary approach as used for official forecasting while more than 2000 CPU hours is needed for a 6-day hurricane run on a computational mesh with millions of nodes. To enhance the efficiency, we develop a data-centric approach to constructing surrogate models from a range of simulations, which exploit the correlation hidden in simulation data and use functional data analysis to model the storm surge response to multiple factors such as hurricane track and the level of wind velocity. After preprocessing the large data, the scale of the problem is then down to data mining on surge and wind profiles at multiple locations along the coast. So, spatio-temporal patterns can be discovered using a set of R packages and the results would be useful for predictions in the hurricane season.

As a complex problem with its multi-physics nature in science [50], numerical simulation [51] is the conventional method for governmental decision support, which essentially models the relationship between horizontal currents and the vertical surge height by solving the depth-integrated Navier-Stokes equation driven by the hurricane generated wind field. However, such a deterministic approach is prone to uncertainty from multiple sources: the predicted hurricane track could be different from the actual one and the wind model may not be very accurate, while the two factors serve as the precondition of the surge model. Thus, a data-centric method would better address the issue of uncertainty with a framework with probability and confidence interval in statistics. By running a large number of hurricane and storm surge simulations and archive them, such data are useful in performing real-time predictions.

While recent work concerning the uncertainty in hurricane-related applications [52, 53] is mainly focused on the hurricane track trajectory and wind intensity, we propose a general framework that aims to detect patterns between track trajectory, wind intensity and surge profile at multiple points of interest (e.g. tide gauges), whose results can potentially provide direct guidance for evacuate planning. With a simulation data archive available [54], the following methods are developed and implemented for pattern detection from hurricane and storm surge simulations: response function model [55], correlation-involved surrogate modeling, where granger test is extensively performed to find correlation between a set of attributes [65] as well as spatial points, and the combination of multiple patterns with statistical inference.

This application can fit the on-the-fly tracing for parallelization because it is a typical example of data-intensive computing: each storm surge simulation, run on parallel computers, can generate more than 30GB raw data. Preprocessing always takes a long time and after that it also includes statistical methods and data mining algorithms for parallelization. Scaling the data mining to multiple simulation runs for continuous knowledge discovery is challenging.

5.1 Modeling Simulation Response using Basis Functions

A response function can represent the output variation in response to the parameter space of simulation input. Functional data analysis [66] is used to construct such models. That is, multiple sets of basic functions are chosen as universal approximators for regression and each element in the parameter space contribute to one set. We denote y as the target variable, extracted from the simulation as $y = \mathcal{F}(x, p_i)$

$$y_i = \alpha + \sum_{j=1}^{n} x_j \beta_j + \epsilon_i \tag{5.1}$$

where α is the intercept, $\beta_j(t)$ represents a basis coefficient expansion and ϵ_i is the residual. Then, x_j is one element from the parameter space, while n is the total number of parameters.

To fit this model, it would be converted to a form that least-square regression can be finally used. The optimization criterion become

$$\{\tilde{\alpha}, \tilde{\beta}\}_\lambda = \arg\min_{\alpha,\beta} \{\sum_{i=1}^{M} [y_i - \alpha - \sum_{j=1}^{n} x_j \beta_j] + \lambda \sum_{j=1}^{n} (\beta_j')^2\} \tag{5.2}$$

where λ is the regularization parameter that restricts the value range of β_j and thus avoids excessive local fluctuation in the estimated function, and M is the number of data points, equivalent to the number of simulations.

From the perspective of regression analysis, the basis function system has the following differences in comparison with ordinary curve fitting (e.g. using a polynomial): *i)* to perform curve fitting for a given function, the solution procedures always include least-squares regression in the end, which requires to solve a linear system. As for the linear system, it is desired that the number of equations should be no less than the number of known variables; and *ii)* It is computationally expensive to treat every independent variable equally in the design function. For example, when using a second-order polynomial to fit a function $y = f(x_1, x_2)$, it is not known a priori that which of x_1 and x_2 affect y in the first order or second order. Then, the design function is

$$y_i = a_1 x_{i1}^2 + a_2 x_{i2}^2 + a_3 x_{i1} + a_4 x_{i2} + a_5$$

Thus, at least 5 sampling points of $\{(x_{i1}, x_{i2}, y_i)\}$ are needed although there are only 2 independent variables.

In the surrogate model, the application can be described as: Suppose that it is needed to find a surrogate model to fit the scalar response. The input of the simulation model includes n scalar variables $\{x_1, x_2, ..., x_n\}$ and the simulation output is just a scalar value y for the given input set. Each simulation is regarded as a *record*, so to use ordinary least-square fitting, it is required to have at least $n + 1$ records for a linear model and much more for higher-oder models. Otherwise, no solution can be found.

Therefore, with the limitation in representational power and computation using ordinary models, basis function system has the advantage of using a hierarchical structure to embed the coefficients to the model, while in essence it is still a linear model: in Eq. 5.1, both x_j and β_j are represented using basis functions. And here x_j itself can be independently represented as a function by the basis function system. So, it substantially addresses the issue of model expressiveness as long as the input is functional. And this is real in many applications or can always be designed to be functional.

In the above description, the response function is compared with ordinary least-square regression while both are supposed in the form of $y = f(X^n)$, where both y and each $x_k \in X^n$ are scalar values. This category of surrogate model is targeted to a single simulation output as (e.g. the maximum value at a specific location in the storm surge simulation) as the *scalar response* to the input. On the other hand, many simulations are performed with time evolution, meaning that the output at a POI is time series instead of a single value. While time series provide more information and the analysis of time series become more sophisticated, response functions can be more advantageous here. Thereby, surrogate models that reflect *functional response* are desired.

To illustrate the use of basis function system for modeling functional response, the form of surrogate model is re-defined as

$$y_i(t) = \sum_{j=1}^{n} x_{ij}\beta_j(t_j) + \epsilon_i \tag{5.3}$$

With the target variable y becoming a function of time t, the corresponding problem setting is slightly different. While in Eq. 5.1 the goal is to find out the coefficients that well fit the scalar response, Eq. 5.3 embeds such coefficients in term β_j as the representational power determines that there much be such solutions. However, with regard to any of $y_1, y_2, ...y_M$, optimal coefficients are the target of this type of model given the simulation input. While these input parameters $x_1, x_2, ...x_n$ are pre-defined and don't vary with time in the setting of simulation models, they are not functional at this stage, although in the a general model it can also be a function of time t as $x_j(t)$.

Similar to scalar response, the solution of the surrogate model for functional response can also be converted into the form of an optimization problem:

$$\tilde{\beta} = \arg\min_{\beta}\{\sum_{i=1}^{M} \int [y_i(t) - \sum_{j=1}^{n} x_{ij}\beta_j(t)]^2 \, dt\} \tag{5.4}$$

Although response functions demonstrate their representational power as universal approximators, domain knowledge is helpful to make sure that the model doesn't include misleading relationships, in order to avoid data dredging.

5.2 Correlation-Involved Surrogate Model

While response function can directly explore the response to simulation input, it is also interesting to discover the spatio-temporal correlation from a point of interest to another. In domain science, there also exists the theoretical foundation of the correlation: as discussed in [?], the coastal-basin geometry has a profound effect to the storm surge. While a simulation output doesn't include only a single value of the target variable, such correlation relationships between locations or variables can be helpful as a new type or a component of surrogate models.

In the context of surrogate model for simulations, while the correlated links between variables or locations may not be known a priori, the target is to find such causal links that tend to be invariants, with the examination of the variance across different simulations. Thus, statistical hypothesis testing is important to find out the links with a certain confidence level before proceeding to model construction. To specify the correlation across locations in the simulation output, y is again denoted as time series extracted from the simulation at location p_i: $y(p_i, t) = \mathcal{F}(x, p_i, t)$.

5.2.1 Granger Testing for Correlation Detection

Granger causality test [63] is performed to find out the specific links in the given data. The POIs $p_1, p_2, ...p_s$ in one simulation form the search space, while any directional pair $\{p_i \rightarrow p_j\}$ can be a link using a bivariate test and a multivariate test involves more points such as $\{p_{i_1}, p_{i_2}, ...p_{i_k} \rightarrow p_j\}$. The result of the test, F-ratio and the corresponding P-value, is calculated based on the comparison between predicting the time series at p_j using only its own values and using time series from auxiliary locations $p_{i_1}, p_{i_2}...p_{i_k}$ along with its own. Then,

$$SSQE_R = \sum_{i=l}^{t} [y(p_j, i) - f(y(p_j, i-l), ..., y(p_j, i-1)]^2 \tag{5.5}$$

$$SSQE_U = \sum_{i=l}^{t} [y(p_j, i) - f(y(p_j, i-l)...y(p_j, i-1), y(p_{i_1}, i-l), y(p_{i_1}, i-l+1)...y(p_{i_k}, i-1)]^2 \tag{5.6}$$

$$F_{granger} = \frac{(SSQE_R - SSQE_U)/l}{SSQE/[t - l(1+k) - 1]} \tag{5.7}$$

In the above three equations, $SSQE_R$ is the sum of square error using the restricted method, which only involves the values at the same location and $SSQE_U$ is that using the unrestricted method, which is elaborated using values from locations $p_{i_1}, ...p_{i_k}$. Thus, the F-ratio is calculated based on the sum of square error with the specified degree of freedom in the problem setting. l represents the lag in prediction. The P-value is then easily obtained according to the value of $F_{granger}$, suggesting the probability that the null hypothesis (the values of the given auxiliary locations can improve the time series prediction) can hold.

In practice, fifteen to more than two hundred locations are selected as in different data sets, including tide gages in Louisiana, Mississippi and Alamaba [64] as well as buoy stations.Therefore, the search for the correlation relationships between locations would not result in costly computation.

5.2.2 Spatiol-Temporal Causal Modeling

Spatio-temporal causal modeling method [65], proposed by Lozano *et al.*, is used for applying Granger causality to modeling the climate change attribution. The *spatio-temporal causal link*, which represents the causality between s set of time series, is defined by the following regression with regard to the time series at one location and its neighbors:

$$y(p_i, t) = \sum_{k=1}^{u} \sum_{l=1}^{s} \alpha_{k,l} y(p_k, t-l) \tag{5.8}$$

Also if more variables other than y are also involved in the model, then

$$y(p_i, t) = \sum_{k=1}^{u} \sum_{l=1}^{s} \alpha_{k,l} y(p_k, t-l) + \sum_{k=1}^{u} \sum_{l=1}^{s} \beta_{k,l} x(p_k, t-l) \tag{5.9}$$

where k represents a relative locations while l is the lag; the above $\alpha_{k,l}$ and $\beta_{k,l}$ are coefficients to be solved. The residual term is omitted here.

The solution of the coefficients in correlation-involved models is straightforward using least-square regression. However, a careful selection of the auxiliary locations is important. If the links can't result in stable helpfulness in prediction, it would be better to discard the link in the model.

5.3 Hybrid Surrogate Model

To consider a general model that works for more than one applications, it is worth incorporating the factors as mentioned in previous sections into one resultant model that can better represent the characteristics of the simulation data. Thereby, a hybrid surrogate model is taken in to account.

5.3.1 Procedural Surrogate Modeling

While large-scale simulations mimic the behavior of complex systems, it is assumed that both the response function and correlation-involved model can reflect a part of the characteristics of data. Then, it is necessary to revisit the problem setting. One simulation subject to surrogate modeling is considered to be a function

$$\{y(p_1, t), y(p_2, t), ..., y(p_s, t)\} \leftarrow \mathcal{F}(X^n) \tag{5.10}$$

When we only focus on one single point of interest p_j, the response function model can be constructed with a basis function system,

$$y(p_j, t) = f(X^n, C, \phi) + \epsilon_0 \tag{5.11}$$

As a simulation model is considered to be noise free and thereby can be representable by a specific form of function system, ϵ_0 is not regarded as random effects but can be further interpreted. Thus, it is thereby assumed that ϵ_0 results from: *i)* insufficient information is obtained from the selected point of interest p_j; and *ii)* the error from selected basis function system (error in computation or coefficients). While *ii)* is solely attributed to the response function modeling, *i)* can be alleviated by adding more information to the same model. So it is necessary to embed both the response function and correlation-involved effects into the same model. Because the response function is constructed independently from point to point, it is regarded as the *primary effect* and the spatio-temporal correlation between points of interest becomes the *secondary effect*.

Then, according to 5.2, spatio-temporal causal modeling can be applied on ϵ_0

$$\epsilon_0(p_j,t) = \sum_{k=1}^{u}\sum_{l=1}^{s}\epsilon_0(p_k,t-l) \tag{5.12}$$

or

$$\epsilon_0(p_j,t) = \sum_{k=1}^{u}\epsilon_0(p_k,t) \tag{5.13}$$

Eq. 5.12 suggests a typical model in the form of time series prediction as no concurrent values are used in the predictors, while Eq. 5.13 sets the lag l as 0 as concurrent values. Eq. 5.12 extends the capability of surrogate model as it can use real-time values from neighbors to improve the quality of prediction. Different from response functions, the model that Eq. 5.13 suggests would utilize historical data as a part of the model input, as offline simulations must be archived and remain available for the surrogate model. In contrast, for response functions, the model coefficients are obtained through training with historical data and in the scenario of prediction, the surrogate can function with the model itself without any additional data.

Putting the two procedures together, the resultant model is

$$y_i(p_j) = \mathcal{F}_i(X^n,p_j) = f_1(X^n) + f_2(y(p_1),...,y(p_u)) + \epsilon' \tag{5.14}$$

where f_1 and f_2 are referred to as the response function and correlation-involved model, respectively. The denotation of time, t, as discussed with different cases, is omitted to keep the generality of the model.

This method is called *procedural surrogate modeling* as it involves primary and secondary stages and models the effects in such an order.

5.3.2 Combining multiple results

The hybrid surrogate model is constructed with the two steps as described above, by differentiating response function from correlation-involved model as primary and secondary effects. However, there always exist multiple results based on the type of model, the model parameter setting, and the data used for model training. In this subsection, a general approach is presented.

Suppose that there is multiple models m_1, m_2, ... ,m_r ($m_k = f_k(X^n, y_{p_{1 \ldots j}})$, for $k \in 1, 2, ..., r$) are trained using data sets from a pool of training data $\mathcal{D}_{\mathcal{T}}$. Another set of validation data $\mathcal{D}_{\mathcal{V}}$ is available then for combination these results. The training and validation data sets are substantially the same using the same simulation model (statistically, they come from the same distribution), but may be localized to different regions of the general distribution. While the test data, as the real prediction task, may have more similarity with the validation data, it would lead to a combination in Eq. 5.15

$$\bar{f}(X^n, y_{p_{1 \ldots u}}) = \sum_{k=1}^{r} b_k m_k \tag{5.15}$$

where b_k is the combining coefficient associated with the model m_k. So, the optimal $b_1, ..., b_r$ can be obtained using least-square regression:

$$\{b_1, ..., b_r\}_{\mathcal{D}_{\mathcal{V}}} = \underset{b_1 \ldots b_r}{\arg\min} \{\sum_{k=1}^{M_V} [\bar{f}(X^n, y_{p_{1 \ldots u}}) - \mathcal{F}(X^n)]^2\} \tag{5.16}$$

where M_V is the size of validation data set (the number of simulations for validation)

When multiple models, especially for both response and correlation models are in use, domain knowledge is desired for combining them with a sound reason. While the validation model can lead to a full-range exploration with simulations over a high-resolution grid, an applicable scenario is: *i)* surge response model is used for points of interest which has a higher confidence level; *ii)* Patterns with spatio-temporal correlation are then applied for predicting other locations. In this way, the surge profiles of multiple locations can be acquired with statistical metrics. The reasonability of this scenario exists in the geometry of the target region would largely determine the predictability. A set of locations would emerge (e.g. Grand Isle, LA in the experiments of the dissertation) first. Then, rather than a surge response with high nonlinearity or dynamics, correlation model can better estimate the values at other locations. In other words, the nature of geometry significantly affects the nature of patterns: some patterns are globally stable, while restriction is usually exerted on local patterns. In order to find more local patterns with spatio-temporal correlation, hurricane tracks can be clustered based on its landfall location or direction.

5.4 Surrogate Detection with Statistical Inference

The approach to surrogate model construction from simulations is applicable for almost all the simulations that output time series in a domain. However, a good surrogate model, defined by the fitness and the variance across simulation samples, may not be found for every simulation model, especially for those chaotic or the models that comprise random effects themselves. In the statistical context, the task is to detect all the possible surrogate models with a certain confidence level. Statistical inference based on surrogate modeling is discussed in the section.

5.4.0.1 Model Validation

A set of criteria and methods are described in order to validate the model. Other than root mean square error or mean absolute error, *coefficient of determination*(R^2) is used as a measure of how well future outcomes are likely to be predicted the model, represented by

$$R^2 = 1 - \frac{\sum_i^M (y_i - f_i)^2}{\sum_i^M (y_i - \bar{y})^2} \tag{5.17}$$

Another factor that can impact model performance is model input representation. The representation of parameter space $\{x_1, x_2, ...x_n\}$ is converted from the original input of the simulation model, and it is worth noting that domain knowledge can help with setting up a better representation to capture the characteristics of simulations. It would be important in experiments that the designed simulations can cover a problem domain, although it may not be clear a priori.

One way to test whether the designed simulations, as the training data of surrogate models, cover the problem domain is to use cross-validation: randomly dividing the data into several equivalent sets, and use most of them for training while leaving the rest for testing. A good design, as well as a good model, should not be expected to have much variance among different combinations of the data sets.

5.5 A Scalable Workflow for High-Performance Execution

While this is relatively a complicated application with data from multiple sources: a hurricane track generator that outputs a range of hurricane tracks used for storm surge simulation; the storm surge simulation outputs, including wind data. Preprocessing from the raw simulation data becomes a significant time-consuming part out of this application: considering the size of raw data, the preprocessing is written in C programming language and the parallelization is exerted on directly partitioning the data or tasks (of extracting data at multiple locations) and passing them to different processors. Figure 7.2 describes the high-level workflow.

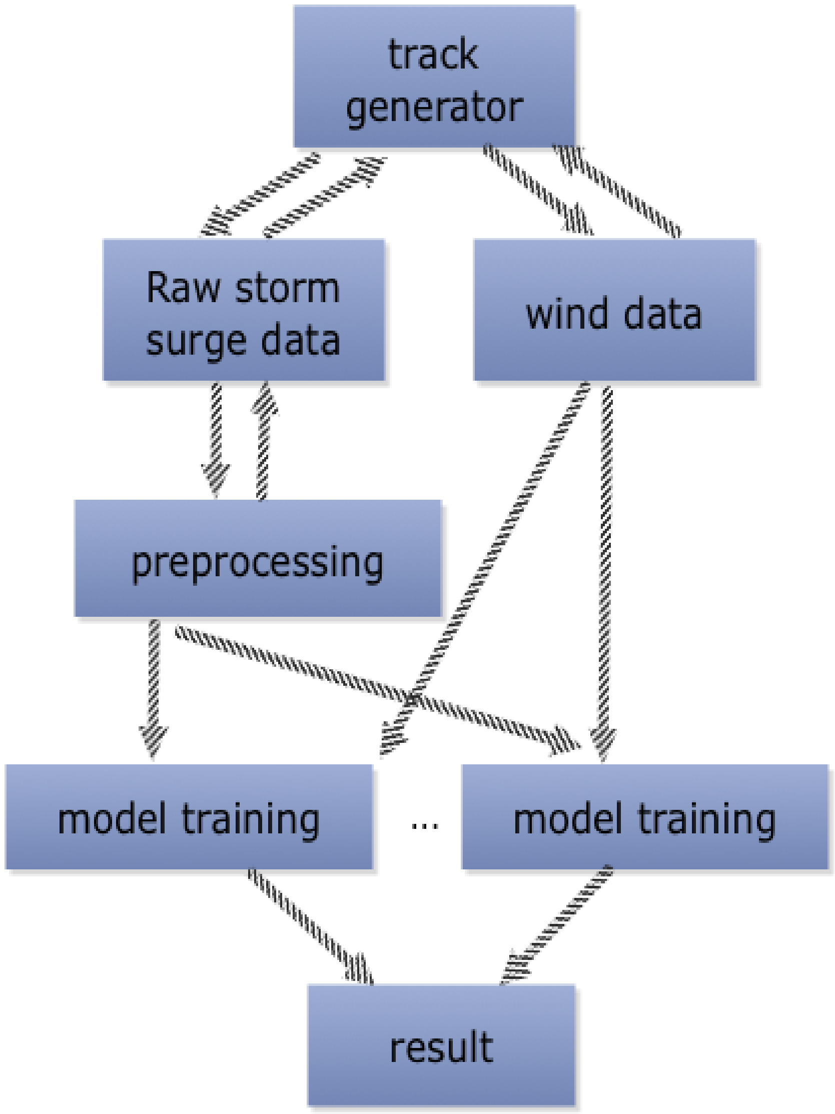


FIGURE 5.1: Workflow for pattern detection from hurricane and storm surge simulations

After preprocessing, the location-specific data shrink a lot in size and then model training becomes more straightforward. However, the challenge of this application exists in the data-driven nature: the hurricane generator can continuous produce new tracks and thereby new simulations will be arriving

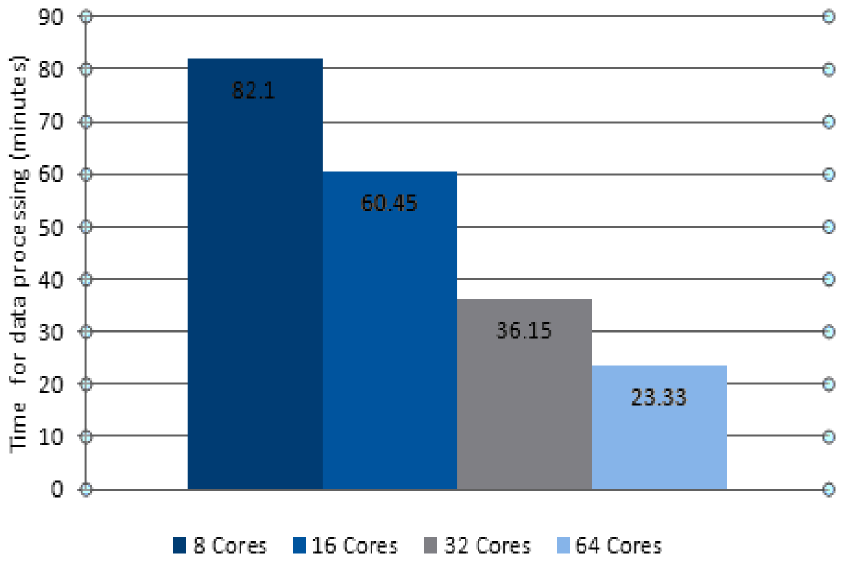


FIGURE 5.2: Scaling test for raw simulation data preprocessing

on demand. Then, the same workflow can be performed in an incremental manner: the addition of new data can bring changes to existing models. Moreover, combining data with parameters (e.g. different track angles on different regions) can also lead to new results. The parallelization framework can then be utilized: once new data become ready after preprocessing, a new set of model training is initialized with parallel architecture and thereby new models are produced. Such a workflow is then executed in a scalable and automated manner.

Then, when using the on-the-fly tracing framework to parallelize the application, it requires a little re-formatting of the code: *i)* the user code, from data preprocessing to the core analytical algorithm, should be written as functions. Then, the framework can recognize each function and scan it, including finding the embedded function as long as the source code is provided. *ii)* For those functions which are used but the source code is not included (e.g. it is included in another R library), it is desirable to give the function name and a weight as an input of the framework for better horizontal partitioning. However, such functions essentially can't be parallelized by the framework unless the source code is available. This same way in source code format in parallelization also applies to the other two applications in the dissertation.

In Figure 5.2 and 5.3, the scaling test of preprocessing raw simulations data and the number of spatio-temporal links over an increasing number of simulations as in 5.2.2 are shown. The continuous

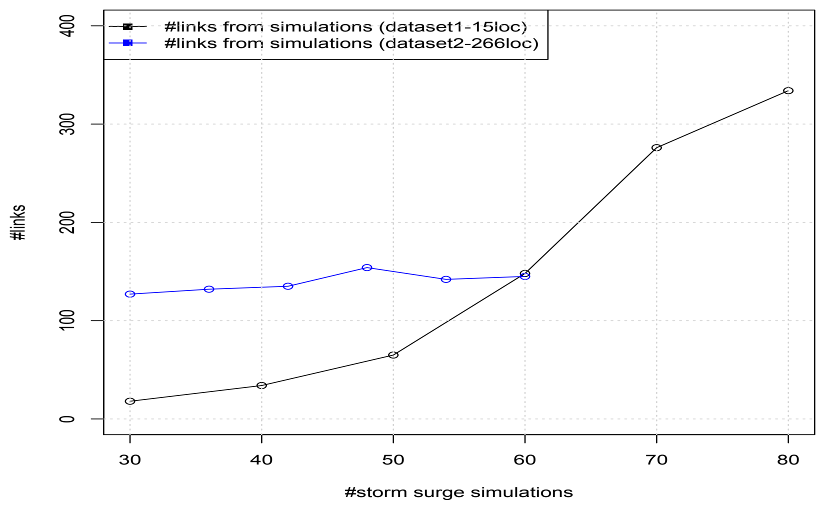


FIGURE 5.3: The number of spatio-temporal links found from an increasing number of simulations (with two data sets)

workflow would become an integral part of a data-driven system, where the outcome always reacts with the injection of new data.

Chapter 6

Application II: Road Traffic Flow Prediction

The real-time conditions of urban transportation play an essential role in city logistics, especially in determining the delivery time for businesses. A precise understanding of the traffic situation is helpful in scheduling and optimizing vehicle routing. Urban traffic flow, recorded or measured by sensors at road cross sections, well represents the level of road congestion. Thus, the prediction of traffic flow is important and directly leads to the estimation of travel time.

Since the last decade, soft computing techniques, including artificial neural networks, genetic algorithm, fuzzy models and simulated annealing, have been widely used for the prediction of short-term traffic flow [67, 68, 69], as well as travel time estimation [70, 71]. With autocorrelation in the traffic flow data (e.g. the current road condition depends on that of 10 minutes ago), ARIMA (Autoregressive Integrated Moving Average) and neural networks are commonly used to detect implicit pattern over a short period. The representational power is shown by training error in the process of model construction. However, the error over test data, which means the traffic volume series from another time slot or a new day, can be very large due to the inconsistency of pattern. That is, the temporal dynamics brings uncertainty to this prediction task. In order to handle the dynamics, an individual predictor would be insufficient to describe the entire data domain but can still take effect in the selected time slot.

Therefore, a straightforward method is to add the number of predictors and accordingly call a part of them in the specific period. The combination of the predictive models can lead to the increase of prediction accuracy: e.g. in [72], the traffic flow regimes are identified and each pattern corresponds to a predictor, and thereby the pattern-based prediction results in a better performance; [73] uses seasonal support vector regression with chaotic simulated annealing to deal with cyclic trend of the traffic volume time series. In this way, it brings the prediction task from "how to train a single model" to "how to train a single model" to "how to select appropriate models to use and combine them".

In this section, we present a high-performance dynamic framework for predicting short-term traffic flow: first, we compare the recurrent patterns in traffic flow between days and propose drifting traffic regimes as extracted from data; and an ensemble learning approach to concept drift is then described to make online predictions. That is, we always maintain a pool of predictor candidates and make selection in accordance to the current traffic regime; also, to comply with the unexpected changes (e.g. brought by weather effect or road emergency), new predictors are trained in a real-time manner to make up the model deviation brought by the temporal dynamics. Such a dynamic framework takes the ever-changing nature of data into account and can effectively select or create predictors as a response to the change in traffic situation.

6.1 Predictive Models with Drifting Regimes

In this section, we propose the concept of drifting traffic regime, which better describes the characteristics of traffic flow data, and then present the ensemble learning method for predicting with concept drift.

6.1.1 Drifting Traffic Regime

Following the trend of previous research, the traffic volume of a day can be clustered based on a set of criteria: road occupancy, traffic pattern recurrence, and the temporal evolution of the deterministic structure [68]. Such statistics can detect the common and recurrent pattern in traffic flow very well, but it is worth denoting that the phase-state pattern, as well as its starting and ending times, varies from day to day in data. We illustrate the variation in Figure 1.1.

The contrast between two typical days indicates the fact that the boundary between traffic regimes is transitive and differs by day and site. That is, it is desired to identify regime transition and choose corresponding predictor(s) in a real-time manner since the traffic flow can unexpectedly drift to another regime. In this way, the same clustering scheme may not be universally applicable although regimes can overlap in between. To formally define the drift, we define each traffic regime as a 3-tuple (C^i, T_s^i, T_e^i) ($C^i \in \{1, 2, ...m\}$, m is the total number of clusters for each day's traffic flow, and T_s^i and T_e^i represent the starting and ending time for the regime). Then, a series of regimes are defined for

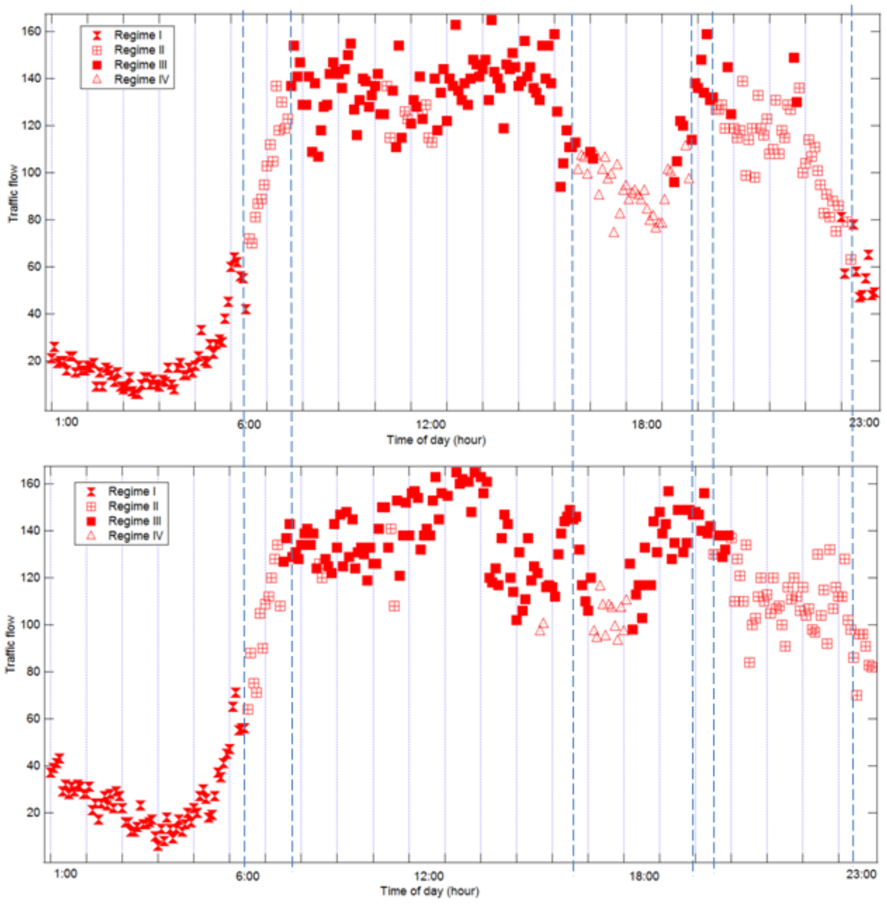


FIGURE 6.1: The variation of traffic regimes between two typical days

each day:

$$D_j = \{(C^1, T_s^1, T_e^1), (C^2, T_s^2, T_e^2), ..., (C^{N_j}, T_s^{N_j}, T_e^{N_j})\} \tag{6.1}$$

Without drift, the following assumption holds:

$$\begin{cases} m_1 = m_2 = ... = m_K \\ C_1^k = C_2^k = ... = C_K^k \\ T_{s,k}^1 = T_{s,k}^2 = ...T_{s,k}^K \\ T_{e,k}^1 = T_{e,k}^2 = ...T_{e,k}^K \\ N_1 = N_2 = ... = N_K \end{cases} \tag{6.2}$$

where $k \in \{1, 2, ..., N_j\}$ and K is the total number of days over all the sampling data.

Drifting traffic regime means the above assumption doesn't hold. In order to avoid an over-complex predictive model, we relax the assumption to the following:

$$m_1 = m_2 = ... = m_K \tag{6.3}$$

This means that there are a set of global regimes that are applicable to each day's data but a different phase-state pattern is assigned with drifting time ranges.

6.1.2 Online Ensemble Method for Predicting with Concept Drift

The global regimes allow us to train universal predictors that can take effect in a specific region and a learner combination is dynamically assembled for real-time prediction. While the traffic regimes are inherently drifting, the handling of concept drift, also as an challenging task in machine learning [?], becomes important. Meanwhile, the ability of incremental processing of neural network learner is utilized, meaning that the weights of each predictor is changed in real time after each new data record is passed. Online learning helps each individual learner to adjust its weights to adapt more to the current situation. We introduce the two schemes respectively.

6.1.2.1 Online neural network learning

Online learning is common for time series prediction. Unlike a classification task, in which all the records in data set can be regarded as under the same distribution and a generalized learner can thereby be trained, time series data don't guarantee the coherence of model performance. So, the model itself, or more specifically, the weights associated with the neural network learner, is changing over time.

The input-output relationship of a multi-layer perceptron (MLP) neural network can be represented by

$$\hat{V}(t+\tau) = g(\sum_i [w_i' f(\sum_j w_{ij} X_j)]) + b \tag{6.4}$$

where the predicted value $\hat{V}$ at time $(t+\tau)$ is the output and the input vector X consists of data points no later than time t; w_i' (on hidden-output layer) and w_{ij} (on input-hidden layer) are adjustable weights and b is the bias term; $f(.)$ and $g(.)$ are activation functions on hidden and output layers for converting the parameter space to a desired range.

In the training process, including on the online mode, each data record is passed to the network structure and the weights are updated using a back propagation algorithm:

$$\Delta w_{ji} = \eta \delta_j x_{ji} \tag{6.5}$$

where Δw_{ji} is the increment for each weight and δ_j represent the partial derivative of error over the layer (e.g. $\delta_j = \hat{v}(1-\hat{v})(v-\hat{v})$ for the output layer; v is target value and $\hat{v}$ is network output), and x_{ji} stands for the layer input. η is a pre-specified constant (e.g. 0.05) as learning rate.

In practice, online neural network training is implemented with the entire network structure reserved all the time in training and online prediction. That is, the trained neural network structure is not encapsulated as an object so the weights of each learner can keep being updated.

6.1.2.2 Dynamic weighted ensemble

Online learning can alleviate the bias brought by the temporal dynamics of traffic flow. However, when the pattern is significantly changed, weight adjustment of an individual learner doesn't suffice. The dynamic weighted ensemble (DWE) learning approach, adapted from dynamic weighted majority developed by Kolter and Maloof [74][1], provides more flexibility by changing the weights of individual predictors, replacing current members or adding new predictors.

In handling concept drift, transferability and responsiveness of the predictor ensemble implicitly become a part of the goal along with accuracy. In a dynamic weighted ensemble E, we denote each individual predictor as e_j ($j \in 1, 2, ...k$), where k is the size of *predictor candidate pool*. Each predictor in the pool has a chance to be selected in real-time prediction based on performance. A weight w_i is associated with each selected predictor in the ensemble, while $i \in 1, 2, ..., m$ and m is the ensemble capacity (no more than m predictors are allowed to be involved in prediction) and the current size of ensemble is m_c. The prediction is then made by the ensemble output:

$$\begin{cases} \hat{V}_E(t+\tau) = \sum_i w_i \hat{V}_{e_j}(t+\tau) \\ \sum_i w_i = 1 \end{cases} \tag{6.6}$$

While all the candidates in the pool are pre-trained with historical data, the scheme also allows the involvement of new predictors to which only the online stream is feed. It is used for dealing with the abrupt change in concept, when existing predictors may perform poorly.

In real-time prediction, a simple online weight updating scheme is adopted:

$$\begin{cases} w_i(t+1) \leftarrow \beta w_i(t) & \text{if } |v(t) - \hat{v}(t)| > \phi_i \\ w_i(t+1) \leftarrow w_i(t) & \text{otherwise} \end{cases} \tag{6.7}$$

where ϕ_s is the error threshold for individuals, $v(t)$ and $\hat{v}(t)$ are the real and predicted values at time t.

Meanwhile, all the weights are always normalized in order to maintain the ability to predict as an ensemble:

$$w_i'(t+1) \leftarrow \frac{w_i(t+1)}{\sum_i w_i(t+1)} \tag{6.8}$$

6.2 A Dynamic Framework for Predicting Traffic Flow with Drifting Regimes

In this section, we describe the online prediction scheme as a framework. The generic procedures are detailed step by step. As mentioned in Section 6.1, with pre-trained predictors, a candidate pool is constructed for online prediction. The procedure, in general, starts from training such regime-specific predictors, and then the forming and updating of the ensemble are performed.

Procedure 1 *Regime-specific predictors training*: the recurrent states and their temporal evolution make pre-trained learners useful as a preparation for online learning and predicting. Using the same method for traffic pattern clustering [68], general traffic regimes can be identified and then corresponding predictors can be trained. There can be more than one predictor for each regime, with different selections in input data as well as the diversity in internal structure and/or weights of predictors themselves. All these predictors construct the predictor candidate pool. The pool is bounded by size k but k is expected to be sufficiently large as more candidates can provide more flexibility against the drift in online prediction.

Procedure 2 *Ensemble initialization*: Only a few predictors takes effect during online processes so the criteria of initial selection also come from the traffic regime based on time. Then,a weight is assigned to each selected predictor, which is all equal at the beginning. We name those selected predictors as *run-time predictors*. A new predictor, which only gets trained in the online process (namely *online predictor*), is also created in the candidate pool.

Procedure 3 *Real-time predicting with weight updating*: Each new data record is passed to all the candidates, with prediction error saved. Weight updating (in Sub-section 6.1.2.2) only applies for run-time predictors. All the weights are saved within the period and become useful for ensemble membership control.

Procedure 4 *Online training*: Online predictors are also maintained by training them with new data. Such predictors are distinguished from regime-specific predictors but sometimes they show better adaptability. If the ensemble membership control is the regime-only scheme, this procedure is not necessary.

Procedure 5 *Ensemble membership control*: After each p records are passed as a period, it is the time to check with the ensemble performance and update the membership of each predictor. With the global error threshold ϕ_g as a watershed, a new predictor, either a regime-specific predictor or an online one, will be added depending on the recorded error in this period. Existing run-time predictors are also subject to removal, determined by its weight. Those with weights lower than the given weight qualification threshold w_ϵ should be removed. Therefore, unlike ensembles with fixed-

TABLE 6.1: Overall performance comparison between dynamic weighted ensemble (regime-only) and Bayesian combined model (8 regime-specific NNs)

	Dynamic weighed ensemble		Bayesian combined	
	AARE(%)	MAE (v/5m)	AARE (%)	MAE (v/5m)
Day 1	10.26	6.87	11.03	7.12
Day 2	10.47	6.50	11.93	7.35
Day 3	12.03	7.07	13.00	7.62
Day 4	10.29	7.23	11.25	7.69
Day 5	9.58	6.50	10.64	7.17
Day 6	10.82	7.08	11.94	7.49
Day 7	10.51	6.94	11.56	7.61
Day 8	11.97	6.95	12.99	7.46
Day 9	10.29	6.55	10.89	6.99
Day 10	13.72	7.45	15.16	8.05
Day 11	11.75	7.08	12.41	7.54
Day 12	12.41	6.44	13.81	7.16
Overall	11.18 ± 1.20	6.89 ± 0.32	12.22±1.34	7.44±0.30

sized learners [?], the number of run-time predictors can be variable but also bounded by capacity m to avoid an over-complex ensemble. Figure **??** shows all the above procedures in a flowchart.

The baseline results, as shown in Table 6.1, illustrates the accuracy of the predictive model in predicting the short-term traffic flow in 12 consecutive days. As a comparison, another approach, Bayesian combined model, is also executed and the dynamic weighted ensemble show better accuracy in most of the days.

6.3 A Scalable Workflow for Parallel Execution

While applying the on-the-fly tracing framework to this application, the high-level workflow is first constructed with modules specified in Figure 6.2. As in Chapter 3, each module is subject for a scan and then the optimal parallelization is obtained for scheduling. In this application, it requires the training model to read data multiple times (as an input stream) and the different training model in a pool is always subject to selection for a best one at a given time. The sharing pattern determines multicore architecture would fit this application: all the training models are in the shared memory without the hassle of inter-node communication and the realtimeliness of this application also asks for a minimum delay.

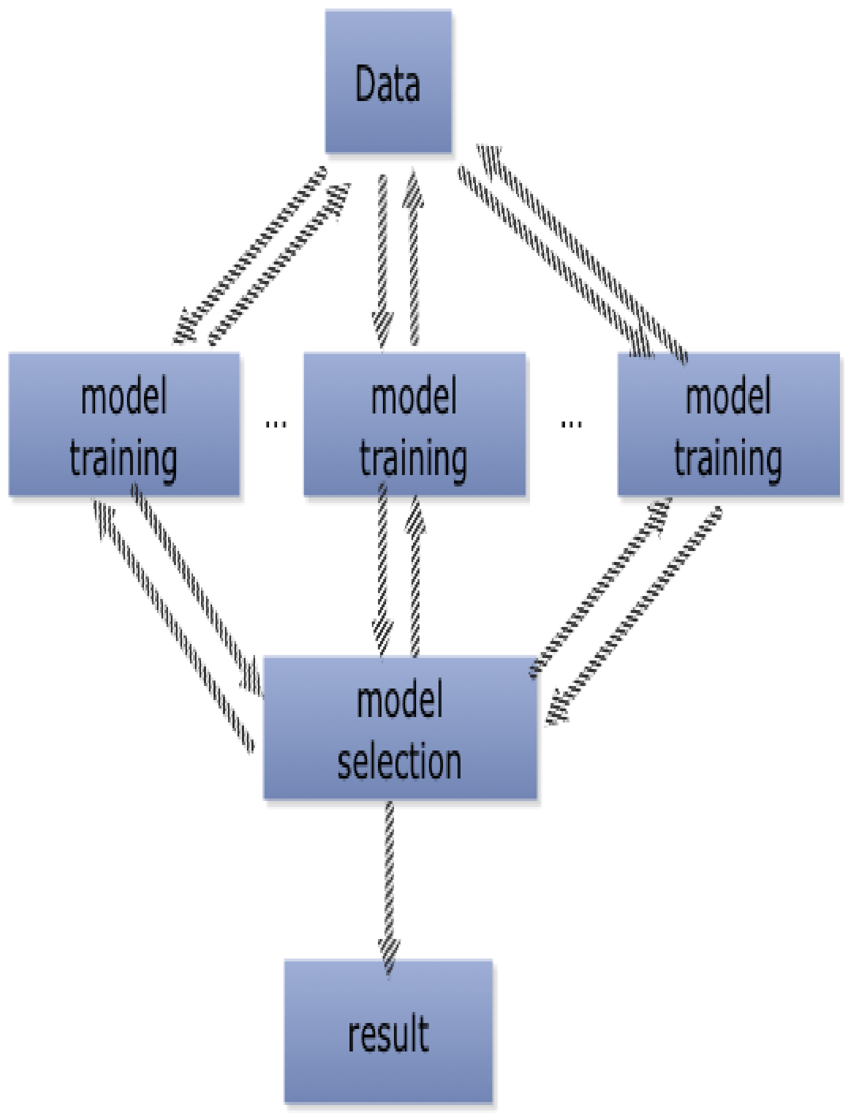


FIGURE 6.2: Workflow for road traffic flow prediction

So, the first level parallelism detected from code is on the training of multiple models (it is the outermost loop in the code). While there are always more than 10 models in the pool, a multithreading solution is adopted mainly on model training. Figure 6.3 gives the speed-up of model training and overall workflow respectively.

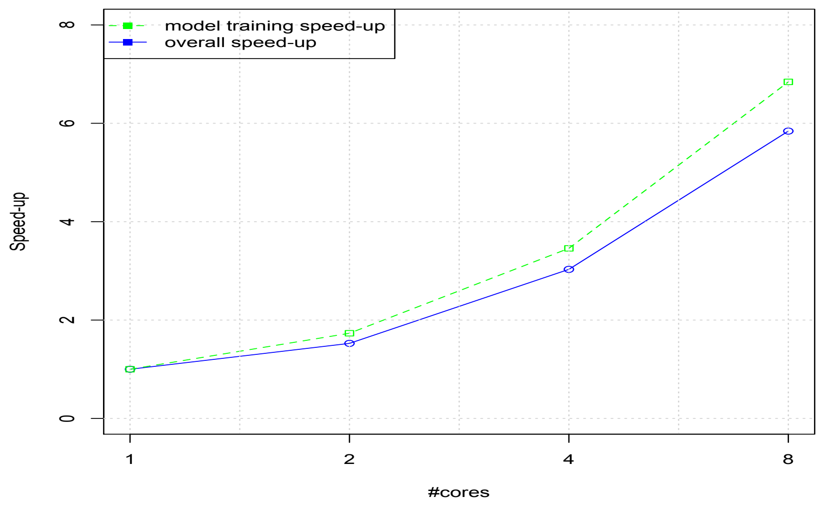


FIGURE 6.3: Speed-up for road traffic flow prediction

Chapter 7

Application III: Text Mining from Social Media Web

Information diffusion is one of the most significant transport phenomena in communications: "diffusion of innovations" [81, 84], as a theory that explains how new concepts, ideas, technical information or actual practices propogate among people and spread through cultures, has been widely accepted for fifty years and useful for enterprise decision making. In general, the diffusion of information, where the information is regarded as a new element in a connected structure, results in cascade-style patterns from one social entity to another. Over the past ten years, with the form of information diffusion becoming much more explicit on the web, more quantitative modeling approaches emerge in the field [78, 80]: the diffusion process, or its characteristics, can be modeled by a set of factors with regard to topics, individual sources and social networks. While existing work apparently puts more emphasis on the global dynamics of information propagation, or the overall distribution of certain topics, we exploit the correlation and similarity between topic trends and user influence in a local scale. Supported by our experiments, such local patterns are based on the following assumptions: *i)* topics, represented by key words in social media streams, are inter-correlated with each other in semantics as well as its trend of propagation; the closer in semantics, the more similar in its pattern of diffusion; and *ii)* Information sources whose online behaviors are similar tend to be more comparable in terms of online influence. Such rules, without global knowledge, would help with training instance-based models from online stream data, which is also in accordance with nature global disconnectedness of Twitter network structure [87].

More specifically for social media streams, where Twitter is a common example, lots of recent work shows promising discoveries. Information diffusion in Twitter, in which a "tweet" is the basic user-generated unit, can be tracked in a way that's easier. Explicit user interactions, such as "retweet" (one quotes another tweet on the timeline of herself via a one-button operation with acknowledging the original author) and "mention" (by mentioning another user identification in a tweet, the other user is notified as someone comments her tweet), provide researchers more convenience for monitoring the trend of topics. The features of information diffusion on blogs and microblogs are compared [87] in terms of distribution of contribution, speed of posting, navigation patterns and network structure; the tweets network is larger, more decentralized and locally connected, so large scale information diffusion may have limited efficiency due to the lack of global connectedness. [75] also shows that "Influencers" in Twitter can also be identified in Twitter through a regression tree. For robustness consideration in limited tweets that don't represent the complete microblogspace, an approximation approach to correcting missing data in information cascades is proposed in [83]. Topics and opinions on Twitter are also an area of interest: the relationship between "follow" and "retweet" is analyzed in [85] and the social media form of political polarization is shown in [77]. Besides, some sociological concepts, such as complex contagion, are also validated through information diffusion on Twitter [82].

On the other hand, as the characterization of information diffusion would lead to the predictability of the level of online influence [75], which can be measured by the topic popularity in media and blogs, the number of comments and the frequency of user interactions in social media, we propose such a predictive model in this paper: instead of attempting to simulation the effect of information diffusion, the correlation between individual online influence and a set of factors such as source credibility, the online behavior similarity and topicality, is exploited.

7.1 Characterizing Information Diffusion with Topic Variation

Prior to introducting a predictive model, there are several concepts for characterizing information diffusion, including the profile of information source and the effect of topic variation.

7.1.1 The level of information exposure

The user interface design of Twitter determines that your followers can see what you post and thereby tweets from those with more followers have a higher chance of being read, indicating a potentially higher influence. However, the level of information exposure is apparently not the only factor of tweet influence, as a high volume of followers doesn't necessarily mean that people are interested in what you talk about this topic. Figure 1.1 (a) describes the relationship using two auxiliary lines: those with really high influence are seemingly logarithmically linear correlated with the number of followers, but a large amount of them show that the sufficiency of followers don't make their tweets retweeted more.

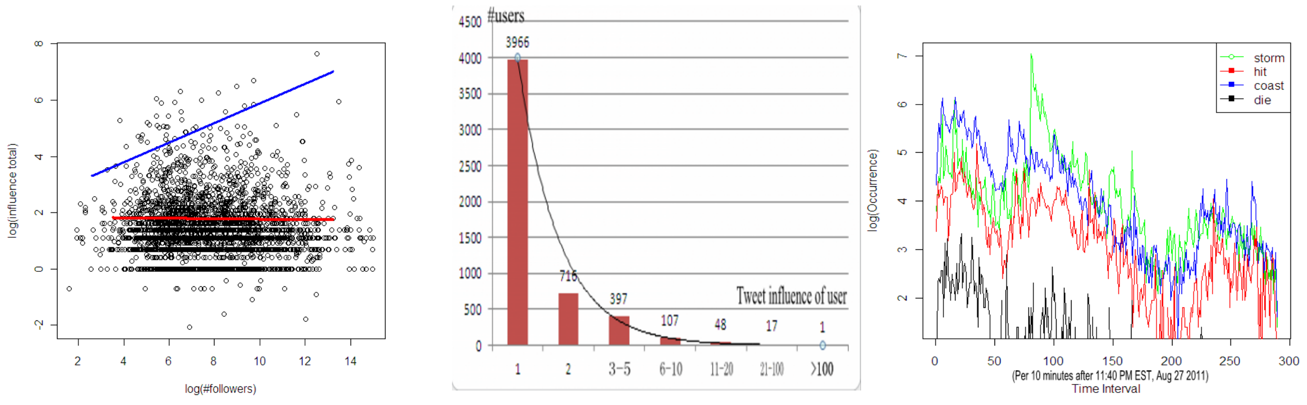


FIGURE 7.1: (a) The relationship between the number of followers and influence total;(b) Tweet influence by user counts; (c) The variation in topic evolution over time.

7.1.2 Online Behavioral similarity

When comparing the influence of two Twitter users, a higher level of online behavioral similarity usually suggests receive feedback at an equivalent level. Apparently, users' new tweets become the recordable activity in data, denoted as $Act(u_i) = \cup_s t_i$. Thereby, in a given time frame, the following two factors, the number of tweets and the time of tweet posting, are taken into consideration for evaluating the behavioral similarity.

Although it is empirical and data-dependent in selecting the boundary of each category, the natural break points in the statistics of tweet influence becomes an intuitive criterion and the majority of posting frequency is considered as a single level. Further, to show the time of tweet posting is important in measuring similarity, an exploratory statistical analysis in Figure 1.1(b) is shown to the number of users by different levels of tweets influence out of the 5,252 who at least have one retweet/mention.

7.1.3 Topicality

Apparently, the content of the tweet matters. Although all the tweets are related to a dominating topic, the opinion, as well as the wording, can implicitly affect the tweet influence. While the format of tweets makes natural language processing techniques a little harder compared to formal articles, we use a unigram model and consider that only those high-profile words in the community help to boost the tweet influence. Figure 1.1(c) shows the evolution of the top three words as shown under the dominating topic ”hurricane irene”. The semantic variation does take effect as the word ”die” is much less frequent after the hurricane made its landfall.

7.2 Local Patterns of Information Diffusion and Predictability

7.2.1 Problem definition

We define that the task is to predict the level of influence of a tweet as classification (e.g. whether the number of retweets and mentions is larger than 5), denoted as $R(t_i)$, within 1 hour from its posting in a topic-oriented community with other users' tweets and their influence available $(\cup_s u_j, t_j, R(t_j))$, including the other tweets of the same author. We don't consider a concurrent setting, so the topic trending is known and tweets at later time can be used too. Otherwise, the trend of information flow of the topic is eventually unforeseen without context provided.

The challenge of this problem exists in its long-tailed data distribution. Although those tweets with no explicit influence are filtered out of our data, most of the selected users still have a relatively low level of influence. Thereby, the extraction of features that make a tweet more influential could be subtle.

7.2.2 Neighborhood event detection

While a neighborhood is primarily referred to as a user with her behavior traceable, it is eventually the influence of a tweet that helps with prediction, namely *neighborhood event*: *i)* Firstly, all tweets that are posted around the time of target tweet are considered as neighborhood event candidates; *ii)* The users of such tweets are then checked to eliminate those whose influence is apparently in a distinct level; *iii)* Lastly, such tweets are sorted by behavioral similarity. Equation 7.1 shows the distance function between tweets.

$$Dist(t_i, t_j) = L(\frac{Inf(u_i)}{Inf(u_j)})abs(t_i.time - t_j.time) \tag{7.1}$$

where function Inf indicates the fundamental influence of users and tweets, which is mainly characterized by the tweet influence of known activities and the level of information exposure. L is a delimiter which returns $+\infty$ to indicate t_i and t_j are not comparable in terms of tweet influence in the context.

On the other hand, when topicality is considered, the hot words of the corpus can be extracted and ranked based on frequency. With each appearance of a word coming with a time stamp, the word frequency actually forms time series and dynamic time warping is adopted to compute the similarity between two words in a given time period. Then, the topicality distance becomes

$$TDist(t_i, t_j) = abs(\sum_{w_s \in t_i} Tr(w_s) - \sum_{w_t \in t_j} Tr(w_t)) \tag{7.2}$$

Function $Tr(w)$ denotes the normalized weight of a word in the tweet. The higher the weight is, the more frequent the word appears in the entire community (instantly or globally).

7.2.3 Generalized components

Though the global disconnectedness makes the predictive model more "local", generalized pattern could still be exploitted from the data set. The influence of a tweet regarding a specific event is likely to meet the evolution of event itself. As a baseline, *linear trending model* just associates the time-dependent topic popularity, represented by the real-time volume of tweets, with tweet influence. It is more commonly used as a component or default value in the case of sparsity (e.g. there are few occurencesof a user in a data set).

$$\bar{R}(u_i, t_i) = Inf(u_i) \sum_{w \in t_i} Tr(w) \tag{7.3}$$

where $Inf(u_i)$ defines the influence of user based on the logarithm of its number of followers.

Acquiring more user statistics globally could elaborate the generalized model, though it may not be feasible in real-time analysis. We also elaborate it to train a linear generalized model (with gradient descent) to detect the pattern when combining with the level of information exposure and the level of user involvement of this event (the number of tweets of a user in the event-related data set):

$$\bar{R}(u_i, t_i) = w_1 Inf(u_i) + w_2 u_i.tf(t_0, t_n) + w_3 \sum_{w \in t_i} Tr(w) + \sigma \tag{7.4}$$

where $tf(t_0, t_n)$ is an attribute of u_i which denotes the tweet frequency in the given time.

7.2.4 Constructing models

The dynamics of Twitter data prompts the training of a set of basic models using the aformentioned concepts. Finally, a mixture of the basic models is also included in the same framework with better accuracy.

7.2.4.1 K-nearest-neighbor combination (kNN)

The neighborhood event detection can help with finding at most k neighborhood events in data, a combination of them leads to a k-nearest-neighbor prediction. The level of information exposure, namely the logarithm of the number of user followers, and the distance measurements, both in behavior and topicality are used as coefficient in the combination.

Using a k-nearest neighbor combination,

$$\bar{R}(u_i, t_i) = \frac{\sum_{j=1}^{k} \arg\min_{w_1 Dist(t_i,t_j) + w_2 TDist(t_i,t_j)} R_{u_j,t_j}}{k} \tag{7.5}$$

7.2.4.2 Model mixture

Unlike a blend of several standalone models, the mixture is constructed based on the feature of tweets and users (like stepwise regression): quite a few users only posted 1-2 tweets in the topic and some users did it more frequently. For instance, for those who have more than 3 tweets, a linear trending model is used. Otherwise, k-nearest-neighbor model is the dominating model unless no qualified neighbor can be found.

7.3 Results and Discussions

7.3.1 Data sets

We use the Twitter streaming API[3] to collect tweet streams sorted by topics, which return tweets in a k-sorted order across the entire time domain. Without the loss of generality, topics in various fields are covered in the search, where the key words are usually a short phrase. In our test, we include topics of different types: "hurricane irene"(irene), "steve jobs"(jobs), "iphone" and "herman

TABLE 7.1: Accuracy of prediction with content free models over 5 data sets

	irene	jobs	iphone	cain	http
Period of Data Collection	Aug 27-29	Oct 8-10	Nov 8-10	Nov 8-10	Nov 3-4
Mixture	0.7865	0.6976	0.7209	0.7034	0.6856
LT-only	0.7235	0.6315	0.6348	0.6070	0.4925
kNN-only	0.7483	0.7247	0.6601	0.6841	0.8354

TABLE 7.2: Accuracy of prediction with the content-based component added

	irene	jobs	iphone	cain	http
Mixture	0.8132	0.7246	0.7098	0.7325	0.6511
LT-only	0.7586	0.6638	0.6849	0.6792	0.4883
kNN-only	0.7712	0.7418	0.6923	0.7231	0.8197

cain"(cain), the number of tweets of which ranges from 166,201 to 1,007,551. We extract relatively more influential users, whose tweets are retweeted or mentioned at least 5 times, from the data bunch because most of the users receive no retweets and making predictions with them is not an interest. Data preprocessing includes extracting the number of retweets and mentions within one hour after the tweet is posted, while around 85% retweets and mentions occur within this time frame. To show the comparison, we also crawled a topicless data set using the Twitter key word "http", which only contains tweets with a short url. All the tweets are collected in a non-stopping span of 2-3 days during August-Novemeber, 2011.

Among the four topics, it is understandable that the result of "steve jobs" shows relatively weaker trending while the data is crawled during October 8-10, a few days after this person passed away. On the contrary, "hurricane irene" shows stronger trending effect than others.

The involvement of content-based component represents two increments from the topic-free baseline models: *i)* The factor of topicality similarity is added as a criterion in neighbor detection: as one single tweet could have many neighbors in the baseline models, text-based neighbors amplify such a relationship; and *ii)* The global trending model based on the volume of tweets is replaced to trends by topic. So, from the results in Table 7.2, most of the predictions receive a boost in accuracy, which suggests that topicality, on top of the user popularity, does affect the influence of a single tweet.

[3] https://dev.twitter.com/docs/streaming-api

While our kNN model is almost parameter-tuning free, the effects of the number k in neighbor selection is not that apparent and data-dependent. Our experiments show, as a common rule, that when k is too large, the accuracy decay while a small k doesn't exploit enough potential of this method. Also, the training of generalized model shows that the effect of initial weights is rather limited as the linear trending component may not be decisive in this context.

7.4 A Scalable Workflow for Parallel Execution

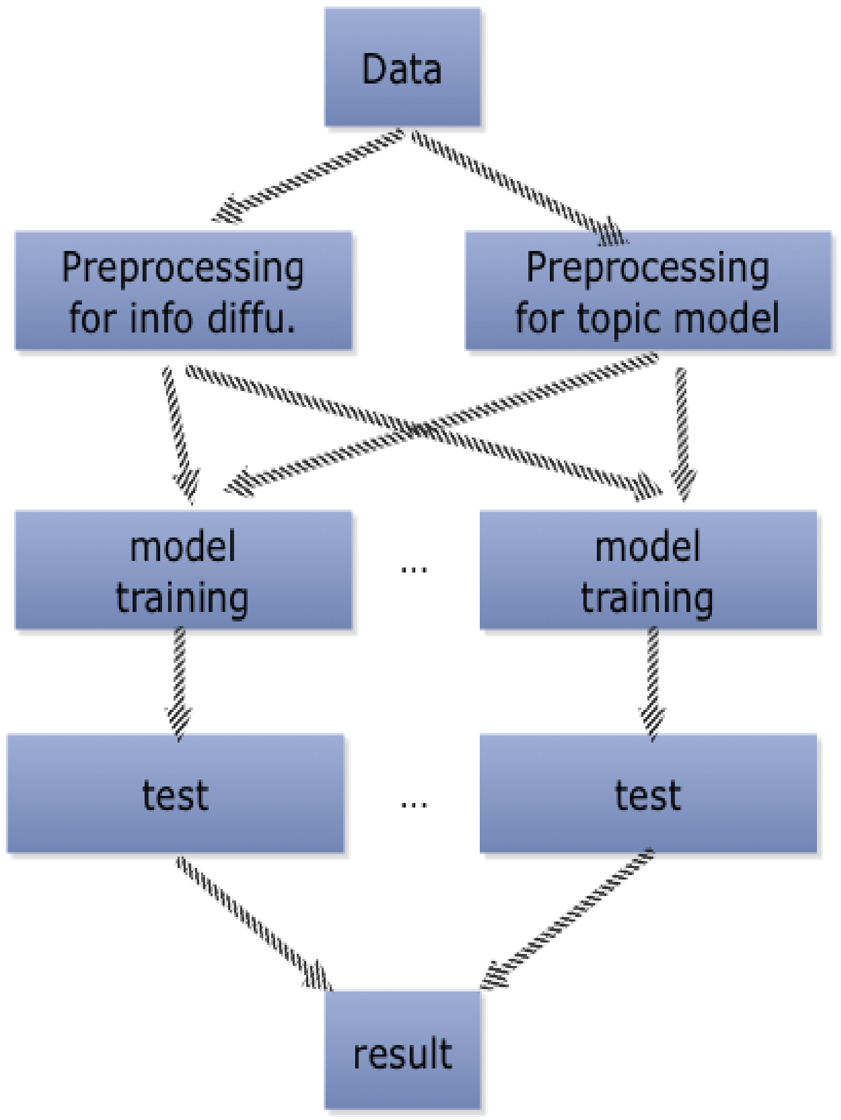


FIGURE 7.2: Workflow for text mining from social media web

This application involves a series of data processing routines: preprocessing for tracking information diffusion, preprocessing for constructing topic models, model training and prediction. The preprocessing part is friendly for a simple MapReduce-style parallelization and the model training, especially for the linear model, can pass the associativity test so is parallelizable too. Another bit of parallelization is for the training of different models in the same way as in Section 6.3.

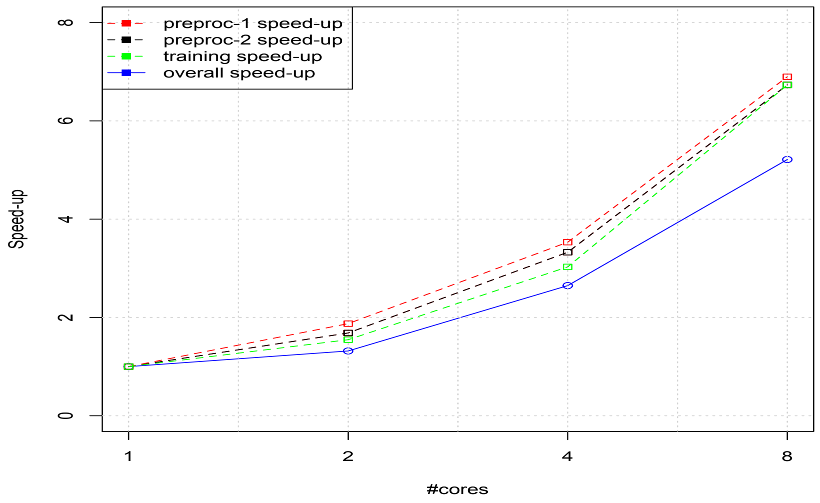


FIGURE 7.3: Speed-up for text mining from social media web

As the flowchart in Figure 7.2 shows, there is an explicit synchronization between the preprocessing of data and model training, where the preprocessed data must be ready before model training. Multicore architecture is again used for avoiding the time-consuming data transport between nodes, while the number of models is no more than 10 in this case. In Figure 7.3, the speed-up is shown for the different modules: preprocessing for information diffusion extraction (preproc-1), preprocessing for content information extraction (preproc-2), model training and the entire workflow. While the synchronization lowers down the overall speedup a bit, considerable performance gain is achieved with multithreading.

Chapter 8
Conclusion

This dissertation describes the design and implementation of on-the-fly tracing for parallelization with its applications for workflow optimization. The on-the-fly tracing is used to find core data dependencies in source code with an input of large-scale data set. In this way, big-data approximation is applied where all the computations unrelated to the big data input is considered negligible and can be duplicated over processing units. While MapReduce simply puts its parallelism on its input data, where the mapper and reducer functions need human effort for re-programming, the on-the-fly tracing technique first aims to this question: a given function, with an input of a large-scale data set, can be parallelized by data or not. So a code scan, with the setting of big-data approximation, is exerted on code to provide more flexibility.

With this general idea, each step of the approach is formalized in Chapter 3: the source code is to be transformed in sections with horizontal partitioning; then for loop parallelization, cycle detection and mutability/associativity tests are adopted to exploit the "data-result" parallelism that ensures the input data can be grouped for the execution on a single processor and then the subresults can be combined without affecting the final results.

The benchmark results in Chapter 3 show the effectiveness of this on-the-fly framework in scaling performance. Considerable speed-up is gained on both multicore and cluster architectures. Compared to MapReduce-style parallelism, the framework *i)* automatically detect the parallelizable part; *ii)* elaborates the essence of data parallelism using mutability/associability test in code scan, which avoids human efforts in thinking about whether a task can be solved using MapReduce and *iii)* dynamically scheduling the execution based on available computing resources. The features provided in the framework can make the "one-way" MapReduce mechanism more intelligent.

With the validation on benchmark results, the framework brings its application to non-trivial tasks: pattern detection from hurricane and storm surge simulations, road traffic flow prediction and text mining from social media web. Starting from a high-level workflow for each application, each module of the application is subject to parallelization while the code is written in a friendly style for on-the-fly tracing. In the same way, such modules are parallelized and the speed-up contributes to the overall workflow. Moreover, the modulization enables the re-organization of workflow, resulting in an optimization. Such a mechanism thereby has the potential to be applied in a wider range of real-world scenarios.

In sum, the on-the-fly approach presented throughout the dissertation enlarges the context of MapReduce and uses an automatic method for exploiting parallelism from the source code. The limitation of this approach comes from its practical implementation: the big data input must be defined in a specific programming language and all the processing is based on this. In the discussion of this work, much emphasis has been put in tracking the indices of data where the big data is treated as a multi-dimensional array. The framework is implemented in R for its popularity and interoperability in data analysis and related applications, and most experiments throughout the dissertation are with this platform too.So, towards a release of the toolkit to public users, more issues regarding genericity, especially in symbol processing and type-specific handling in code scan, should be considered and hard coded.

Further, such a framework has the potential of being widely applied in more real-world applications. While more and more areas are being heavily involved with big data and related operations, a “data-driven” workflow becomes common in today’s research community and industry. The approach of on-the-fly tracing is generic so can be extended to multiple programming languages and embedded into system software: it can help to detect the optimal workflow with given data-centric tasks and computing resources, as well as automatically conducting parallel execution in a modularized project environment.

Bibliography

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *Proc. of OSDI'2004.*

[2] Apache Hadoop: http://hadoop.apache.org/

[3] Z. Ma and L. G, "The limitation of MapReduce: a probing case and a lightweight solution," in *Proc. of the 1st Int'l Conf. on Cloud Computing, GRIDs, and Virtualization*, November 2010.

[4] J. Lin and C. Dyer, *Data-Intensive Text Processing with MapReduce*, published by Morgan & Claypool, 2010.

[5] M. Obata, J. Shirako, H. Kaminaga, K. Ishizaka and H. Kasahara, "Hierarchical parallelism control for multigrain parallel processing," in *Proc. of the 15th Workshop in Languages and Compilers for Parallel Computing (LCPC'2005)*, July 2005.

[6] J.J. Willcock, T. Hoefler and N. Edmonds, "Active Pebbles: parallel programming for data-driven applications," in *Proc. of International Conference of Supercomputing (ICS'2011)*, May-June 2011.

[7] A. Ghoting, P. Kambadur, E. Pednault and R. Kannan, "NIMBLE: a tookit for the implementation of parallel data mining and machine learning algorithms on MapReduce," in *Proc. of ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (KDD'2011)*, August 2011.

[8] K.J. Brown et al., "A heterogeneous parallel framework for domain-specific languages," in *Proc. of the 20th Int'l Conf. on Parallel Architectures and Compilation Technologies (PACT'2011)*, October 2011.

[9] B. Ren and G. Agrawal, "Compiling dynamic data structures in Python to enable the use of multi-core and many-core libraries," in *Proc. of the 20th Int'l Conf. on Parallel Architectures and Compilation Technologies (PACT'2011)*, October 2011.

[10] R. Eigenmann, J. Hoeflinger, Z. Li and D. Padua, "Experience in the automatic parallelization of four perfect-benchmark programs," in *Proc. of the 4th Workshop in Languages and Compilers for Parallel Computing (LCPC'1991)*, August 1991.

[11] J.G. Steffan and T.C. Mowry, "The potential for using thread-level data speculation to facilitate automatic parallelization," in *Proc. of the 4th Int's Symp. on High-Performance Computer Architecture (HPCA'1998)*, February 1998.

[12] S. Sato and H. Iwasaki, "Automatic parallelization via matrix multiplication," in *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'2011)*, pp. 470-479, June 2011.

[13] C. Lengauer, "Loop parallelization in the polytope model," in *Proc. of the 4th Int'l Conf. on Concurrency Theory (CONCUR'1993)*, August 1993.

[14] M.E. Wolf and M.S. Lam, "A data locality optimizing algorithm," in *Proc. of the Int'l Conf. of Programming Language Design and Implementation (PLDI'1991)*, 1991.

[15] N. Ahmed, N. Mateev and K. Pingali, "Tiling imperfectly-nested loop nests," in *Proc. of the 12th Int'l Conf. of High Performance Networking and Computing Science (SC'2000)*, November 2000.

[16] S. Krishnamoorthy et al., "Effective automatic parallelization of stencil computations," in *Proc. of the ACM SIGPLAN Conf. of Programming Language Design and Implementation (PLDI'2007)*, June 2007.

[17] N. Bliss, "Addressing the multicore trend with automatic parallelization," *Lincoln Laboratory Journal*, Massachusetts Institute of Technology, vol. 17, no. 1, 2007.

[18] U. Bondhugula et al., "Towards effective automatic parallelization for multicore systems," in *Proc. of the Next Generation Software Workshop (NGS'2008)*, April 2008.

[19] D.W. Walker, L. Huang, O.F. Rana and Y. Huang, "Dynamic service selection in workflows using performance data," *Scientific Programming-Dynamic Computational Workflows: Discovery, Optimization and Scheduling*, vol. 15, iss. 4, pp. 235-247, December 2007.

[20] Z. Yun et al., "A innovative application execution toolkit for multicluster grids," in *Proc. of Int'l Conf. of Cluster Computing (Cluster'09)*, August 2009.

[21] J. Ginsberg et al., "Detecting influenza epidemics using search engine query data," *Nature*, vol. 457, pp. 1012-1014, February 19, 2009.

[22] A. McKenna et al., "The genome analysis toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data," *Genome Research*, vol. 20, pp. 1297-1303, 2010.

[23] C. Olston, B. Reed, U. Srivastava, R. Kumar and A. Tomkins, "Pig Latin: a not-so-foreign language for data processing," in *Proc. of Int'l Conf. of ACM SIGMOD (SIGMOD'08)*, June 2008.

[24] Apache Pig: http://pig.apache.org/

[25] M. Zaharia, A. Konwinski, A.D. Joseph, R.H. Katz and I. Stonica, "Improving MapReduce performance in heterogeneous environments," in Proc. of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'2008), December 2008.

[26] Amazon Elastic Compute Cloud (Amazon EC2): http://aws.amazon.com/ec2/

[27] B. He, W. Fang, Q. Luo, N.K. Govindaraju and T. Wang, "Mars: a MapReduce framework on graphics processors," in *Proc. of the 17th Int'l Conf. of Parallel Arch. and Compilation Techniques (PACT'08)*, October 2008.

[28] B. Zhang, Y. Ruan, T.-L. Wu, J. Qiu, A. Hughes and G. Fox, "Applying Twister in scientific applications," in *Proc. of the 2nd IEEE Int'l Conf. on Cloud Computing Technology and Science (CloudCom'2010)*, November-December 2010.

[29] J.A. Stuart, C.-K. Chen, K.-L. Ma and J.D. Owens, "Multi-GPU volume rendering using MapReduce," in *Proc. of the ACM Int'l Symp. of High-Performance and Distributed Computing (HPDC'2010)*, June 2010.

[30] G. Malewicz et al., "Pregel: a system for large-scale graph processing," in *Proc. of Int'l Conf. of ACM SIGMOD (SIGMOD'10)*, June 2010.

[31] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proc. of the 7th European System Conference (EuroSys'2007)*, March 2007.

[32] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P.K. Gunda and J. Currey, "DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language," in *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*, December 2008.

[33] A. Zhai, C.B. Colohan, J.G. Steffan, and T.C. Mowry, "Compiler optimization of scalar value communication between speculative threads," in *Proc. of Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*, October 2002.

[34] P. Feautrier, "Automatic parallelization in the polytope model," *The Data Parallel Programming Model (in the series of Lectures Notes in Computer Science)*, vol. 1132, pp. 79-103, Springer, 1996.

[35] U. Bondhugula, A. Hartono, J. Ramanujam and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pp. 101-113, June 2008.

[36] L.-N. Pouchet, U. Bondugula, C. Bastoul, A. Cohen, J. Ramanujam and P. Sadayappan, "Combined iterative and model-driven optimization in an automatic parallelization framework," in *Proc. of the ACM/IEEE Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC'10)*, November 2010.

[37] K. Matsuzaki, Z. Hu, M. Takeichi, "Towards automatic parallelization of tree reductions in dynamic programming," in *Proc. of the 18th annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'06)*, pp. 39-48, July-August 2006.

[38] J.G. Steffan and T. Mowry, "The potential for using thread-level data speculation to facilitate automatic parallelization," in *Proc. of the 4th Int'l Symp. on High-Performance Computer Architecture (HPCA'1998)*, 1998.

[39] J.G. Steffan, C.B. colohan, A. Zhai and T.C. Mowry, "A scalable approach to thread-level speculation," in *Proc. of the ACM/IEEE Int'l Symp. on Computer Architecture (ISCA'00)*, 2000.

[40] M.K. Prabhu and K. Olukotun, "Using thread-level speculation to simplify manual parallelization," in *Proc. of ACM SIGPLAN Int'l Conf. of Principles and Practice of Parallel Programming (PPoPP'2003)*, June 2003.

[41] A. Kejariwal et al., "Exploitation of nested thread-level speculative parallelism on multi-core systems," in *Proc. of the 7th ACM Int'l Conf. on Computing Frontiers (CF'10)*, May 2010.

[42] E. Deelman, D. Gannon, M. Shields and I. Taylor, "Workflows and eScience: an overview of workflow system," *Future Generation Computer Systems*, vol. 25, iss. 5, pp. 528-540, May 2009.

[43] E. Deelman et al., "Pegasus: a framework for mapping complex workflows onto distributed systems," *Scientific Programming*, vol. 13, iss. 3, pp. 219-237, July 2005.

[44] E. Schikuta, H. Wanek and I.U. Haq, "Grid workflow optimization regarding dynamically changing resources and conditions," *Concurrency and Computation: Practice and Experience*, vol. 20, pp. 1837-1849, 2008.

[45] K. Agrawal, A. Benoit, L. Magnan and Y. Robert, "Scheduling algorithms for linear workflow optimization," in *Proc. of Int'l Symp. on Parallel and Distributed Processing (IPDPS'10)*, April 2010.

[46] S. Bowers, "Scientific workflow, provenance and data modeling challenges and approaches," *Journal of Data Semantics*, vol. 1, pp. 19-30, 2012.

[47] M.M. Baskaran, N. Vydyanathan, U. Bondhugula, J. Ramanujam, A. Rountev, P. Sadayappan, "Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors," in *Proc. of the 14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'09)*, pp. 219-228, April 2009.

[48] A. Buttari, J. Langou, J. Kurzak, J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, iss. pp. 38-53, January 2009.

[49] The R Project for Statistical Computing: http://r-project.org/

[50] D.T. Resio and J. Westerink, "Modeling the physics of storm surges," in *Physics Today*, vol. 61, iss. 9, pp. 33-38, 2008.

[51] ADCIRC Coastal Circulation and Storm Surge Model: http://adcirc.org/

[52] S.-S. Ho, W. Tang and W.T. Liu, "Tropical cyclone event sequence similarity search via dimensionality reduction and metric learning," in *Proc. of ACM SIGKDD Conf. of Knowledge Discover and Data Mining (KDD'10)*, August 2010.

[53] V. Jovanovic, M.H. Dunham, M. Hahsler, and Y. Su, "Evaluating hurricane intensity prediction techniques in real time," in *Proc. of Int'l Conf. of Data Mining Workshops (ICDMW'11)*, December 2011.

[54] H. Bhagawaty, L. Jiang et al., "Design, implementation and use of a simulation data archive for coastal science," in *Proc. of the ACM Symposium of High-Performance and Distributed Computing Workshops (HPDCW'10)*, June 2010.

[55] J. Ramsay, G. Hooker and S. Graves, *Functional Data Analysis with R and MATLAB*, published by Springer, 2009 (DOI: 10.1007/978-0-387-98185-7)

[56] D.T. Resio, J. Irish and M. Cialone, "A surge response function approach to coastal hazard assessment - Part 1: Basic concepts," *Natural Hazards*, vol. 51, no. 1, pp. 163-182, 2009.

[57] Z. Qian, C.C. Seepersad and V.R. Joseph, "Building surrogate models based on detailed and approximate simulations," *Journal of Mechanical Design*, vol. 128, iss. 4, pp. 668-677, 2006.

[58] J. Irish, D.T. Resio, and M. Cialone, "A surge response function approach to coastal hazard assessment - Part 2: Quantification of spatial attributes of response functions," *Natural Hazards*, vol. 51, no. 1, pp. 163-182, 2009.

[59] D. Abramson et al., "Parameter space exploration using scientific workflows," in *Proc. of International Conference of Computational Science (ICCS'09)*, pp. 104-113, 2009.

[60] J. Park and I.W. Sandberg, "Universal approximation using radial-basis-function networks," *Neural Computation*, vol. 3, no. 2, pg. 246-257, 1991.

[61] L.-X. Wang and J.M. Mendel, "Fuzzy basis functions, universal approximations and orthogonal least-squares learning," *IEEE Transactions on Neural Networks*, vol. 3, no. 5, pp. 807-814. 1992.

[62] A.R. Barron, "Universal approximation bounds for superpositions of a sigmoidal function," *IEEE Transactions on Information Theory*, vol. 39, no. 3, pp. 930-945, 1993.

[63] C.W.J. Granger, "Investigating causal relations by econometric models and cross-spectral methods," *Econometrica*, vol. 37, no. 3, pp. 424-438, 1969.

[64] NOAA (National Oceanic and Atmospheric Administration) Tides and Currents: http://tidesandcurrents.noaa.gov.

[65] A. Lozano, et al., "Spatial-temporal causal modeling for climate change attribution," in *Proc. of ACM SIGKDD Conference of Knowledge Discovery and Data Mining (KDD'09)*, 2009.

[66] J. Ramsay and B. W. Silverman, *Functional Data Analysis (2nd ed.)*, published by *Springer*, pp. 261-277, 2005.

[67] H. Yin, S.C. Wong, J. Xu and C.K. Wong, "Urban traffic flow prediction using a fuzzy-neural approach," in Transportation Research Part C: Emerging Technologies , vol. 10, iss. 2, pp. 85-98, Apr. 2002.

[68] E.I. Vlahogianni, M.G. Karlaftis and J.C. Golias, "Temporal evolution of short-term urban traffic flow: a nonlinear dynamics approach," in Computer-Aided Civil and Infrastructure Engineering, vol. 23, iss. 7, pp. 536-548, Oct. 2008.

[69] M. Tan, S.C. Wong, J. Xu, Z.-R. Guan and P. Zhang, "An aggregation approach to short-term traffic flow prediction," in IEEE Transactions on Intelligent Transportation Systems, vol. 10, no. 1, pp. 60-69, Mar. 2009.

[70] V. Hinsbergen and V. Lint, "Bayesian combination of travel time prediction models," in Transportation Research Record, vol. 2064, pp. 73-80, 2008

[71] X. Fei, C.-C. Lu and K. Liu, "A Bayesian dynamic linear model approach for real-time short-term freeway travel time prediction," to appear Transportation Research Part C, available online Apr. 1, 2011.

[72] E.I. Vlahogianni, "Enhancing predictions in signalized arterials with information on short-term traffic flow dynamics," in Journal of Intelligent Transportation Systems, vol. 13, iss. 2, pp. 73-84, 2009.

[73] W.C. Hong, "Traffic flow forecasting by seasonal SVR with chaotic simulated annealing algorithm," to appear in Neurocomputing.

[74] J. Kolter, M. Marloof, "Dynamic weighted majority: an ensemble for drifting concepts," *Journal of Machine Learning Research*, vol. 8, pp. 2755-2790, 2007.

[75] E. Bakshy, J.M. Hofman, W.A. Mason and D.J. Watts," Identifying 'influencers' on Twitter," *Proc. of the 4th ACM Int'l Conf. on Web Search and Data Mining (WSDM'11)*, Feb 2011.

[76] M.D. Choudhury, Y.-R. Lin, H. Sundaram, K.S. Candan, L. Xie and A. Kelliher. How does the data sampling strategy impact the discovery of information diffusion in social media? *Proc. of the 4th AAAI Int'l Conf. on Weblogs and Social Media (ICWSM'2010)*, May. 2010.

[77] Conover et al., "Political polarization on Twitter," in *Proc. of the 5th AAAI Int'l Conf. on Weblogs and Social Media (ICWSM'2011)*, July 2011.

[78] D. Gruhl, R. Guha, D.Liben-Nowell and A. Tomkins, "Information diffusion through blogspace," in *Proc. of the 13th Int'l Conf. on World Wide Web (WWW'04)*, May 2004.

[79] J. Leskovec, L.A.Adamic and B.A. Huberman, "The dynamics of viral marketing,", *ACM Transactions on the web*, vol. 1, iss. 1, May 2007.

[80] J. Leskovec, L. Backstrom and J. Kleinberg, "Meme-tracking and the dynamics of the news cycle," in *Proc. of the ACM SIGKDD Conference of Knowledge Discovery and Data Mining (KDD'09)*, June-July 2009.

[81] E.M. Rogers, *Diffusion of Innovations* (5th Edition), published by Free Press, 2003.

[82] D.M. Romero, B. Meeder and J. Kleinberg, "Differences in the mechanics of information diffusion across topics: idioms, political hashtags, and complex contagion on Twitter," in *Proc. of the 20th Int'l Conf. on World Wide Web (WWW'11)*, March-April 2011.

[83] E. Sadikov, M. Medina, J. Leskovec and H. Garcia-Molina. Correcting for missing data in information cascades. *Proc. of the 4th ACM Int'l Conf. on Web Search and Data Mining (WSDM'11)*, Feb 2011.

[84] B. Wejnert, "Integrating models of diffusion of innovations: a conceptual framework," *Annual Review of Sociology*, vol. 28, pp. 297-326, 2002.

[85] M.J. Welch, U. Schonfeld, D. He and J. Cho, "Topical semantics of Twitter links," in *Proc. of the 4th ACM Int'l Conf. on Web Search and Data Mining (WSDM'11)*, February 2011.

[86] J. Yang and J. Leskovec, "Modeling information diffusion in implicit networks," in *Proc. of 2010 IEEE Int'l Conf. on Data Mining (ICDM'10)*, December 2010.

[87] J. Yang and S. Counts, "Comparing information diffusion structure in weblogs and microblogs," in *Proc. of the 4th AAAI Int'l Conf. on Weblogs and Social Media (ICWSM'2010)*, May 2010.

Vita

Lei Jiang was born in Wuhan, China, in 1986. He grew up in Wuhan and was exceptionally recruited by the Special Class for the Gifted Young, University of Science and Technology of China, when he was fifteen, right after the compeletion of only one-year study in senior high school (which is normally for three years). Then he obtained a Bachelor of Engineering in Computer Science from University of Science and Technology of China in 2005 and a Master of Engineering in Applied Computer Science from Wuhan University in 2007, respectively. After less than one year study in Canada, Lei transferred to Louisiana State University in July of 2008.

Publications, Posters and Demos

1. L. Jiang, P. Patel, G. Ostrouchov and F. Jamitzky OpenMP-style parallelism for data-centered multicore computing with R. in Proc. of the 17th SIGPLAN Symp. on Prin. and Prac. of Parallel Prog. (PPoPP'12), February 2012.
2. L. Jiang, G. Allen, Q. J. Chen Scalable and automated workflow in Mining Large-Scale Severe-Storm Simulations, to appear in the Proc. of Int'l Conf. on Statistical and Scientific Database Management (SSDBM'11), July 2011.
3. H. Bhagawaty, L. Jiang, S. Pothanis, G. Allen, N. Brener and T. Kosar Design, Implementation and Use of a Simulation Data Archive for Coastal Science, in the Proc. of Int'l ACM Symp. on High-Performance Parallel and Distributed Computing Workshops (HPDCW'10), June 2010.
4. L. Jiang, J. Zhang, G. Allen Transferred Correlation Learning: An Incremental Scheme for Neural Network Ensembles , in the Proc. of International Joint Conference of Neural Networks (IJCNN'10), July 2010.
5. L. Jiang, Q. Fan, G. Allen and Q. Chen Towards an Integrated Problem-Solving Environment for Hybrid Numerical Models with Statistical Learning Components (demonstration paper), TeraGrid Conference, June 2009.