Parallelizing Applications With a Reduction Based Framework on Multi-Core Clusters

Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Venkatram Ramanathan, B.Tech

Graduate Program in Computer Science and Engineering

The Ohio State University

2010

Thesis Committee:
Dr. Gagan Agrawal, Advisor
Dr. Radu Teodorescu
© Copyright by
Venkatram Ramanathan
2010
ABSTRACT

Data mining has emerged as an important class of high performance applications. At the same time, most parallel platforms today are clusters of multi-core machines. Thus, one of the major challenges today is achieving programmability and performance for data mining applications on multi-core machines and cluster of multi-core machines. FREERIDE (FRamework for Rapid Implementation of Datamining Engines) is a middleware developed based on the observation that the processing structure of a large number of data mining algorithms involves generalized reductions. FREERIDE offers a high-level interface and implements both distributed memory and shared memory parallelization.

In this thesis, the Wavelet Transformation algorithm is considered and it is shown how it can be modeled as a generalized reduction structure. It is parallelized using the FREERIDE middleware. It is shown that the algorithm can be parallelized in a communication and storage efficient manner. By this method, a good parallel efficiency with a speedup of around 42 on 64 cores is achieved.

The second algorithm considered is a challenging new data mining algorithm, information theoretic co-clustering. This algorithm is parallelized using FREERIDE middleware. It is shown that the main processing loops of row clustering and column clustering of the Co-clustering algorithm essentially fit into a generalized reduction structure. A good parallel efficiency is achieved and a speedup of nearly 21 is reported on 32 cores.
I dedicate this work to my parents Ramanathan Venkitaraman and Sudha Ramanath, my sister Keertana Ramanathan and my adviser Dr. Gagan Agrawal.
ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Gagan Agrawal, for his vision and constant support throughout the course of my Masters. His excellent technical foresight and guidance helped me direct my energies towards the right goals. I would also like to thank Dr. Radu Teodorescu for agreeing to serve on my Master’s examination committee.

Special thanks also to all my friends at the Data Grid Lab, especially Vignesh Trichy Ravi and Wenjing Ma, and to Dr. Muthu Manikandan Baskaran for keeping me motivated and making this journey a really enjoyable one.

Finally, this endeavor would not have been possible without the support of my parents, who have encouraged and motivated me to no end. It is to my parents and to my advisor that I would like to dedicate my work to.
VITA

2004-2008 . . . . . . . . . . . . . . . . . . . . B.Tech, Information Technology,  
   PSG College of Technology  
   Coimbatore, India.

2008-2010 . . . . . . . . . . . . . . . . . . . . Masters Student,  
   Department of Computer Science and Engineering,  
   The Ohio State University.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

  Studies in:  
    Parallelizing Data Mining Algorithms   Prof. Gagan Agrawal
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract ............................................................... iii</td>
</tr>
<tr>
<td>Dedication ............................................................. iv</td>
</tr>
<tr>
<td>Acknowledgments ....................................................... v</td>
</tr>
<tr>
<td>Vita ................................................................. vi</td>
</tr>
<tr>
<td>List of Figures ......................................................... ix</td>
</tr>
</tbody>
</table>

Chapters:

1. Introduction ........................................................... 1
   1.1 A Brief History of Parallel Computing ......................... 1
   1.2 Evolution of MPI and OpenMP ................................... 2
   1.3 Evolution of MapReduce .......................................... 4
   1.4 FREERIDE Middleware and Parallelization Techniques .......... 5
      1.4.1 Shared Memory Parallelization ........................... 8
   1.5 Target Applications And Contributions ........................ 10

2. Parallelizing Wavelet Transform on FREERIDE .................... 12
   2.1 Sequential Wavelet Transform ................................. 14
   2.2 Parallelizing Wavelet Transform Algorithm with FREERIDE .... 17
      2.2.1 Shared Memory Parallelization ......................... 19
      2.2.2 Distributed Memory Parallelization .................... 19
      2.2.3 Hybrid Parallelization and an Optimization ............. 20
      2.2.4 Generating Index For Output Values .................... 22
   2.3 Experimental Results ............................................. 22

vii
2.3.1 Performance of Shared and Distributed Memory Parallelization . 25
2.3.2 Scalability of Hybrid Parallelization . . . . . . . . . . . . . . . 29
2.4 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

3. Parallelization Of Information Theoretic Co-clustering On Freeride . . . . . . . 34
  3.1 Co-clustering algorithm . . . . . . . . . . . . . . . . . . . . . . . . . 35
  3.2 Parallelizing Co-clustering with FREERIDE . . . . . . . . . . . . . . . 40 
    3.2.1 Co-clustering Algorithm and Generalized Reductions . . . . . . . 40 
    3.2.2 Distributed Memory Parallelization . . . . . . . . . . . . . . . . 42 
    3.2.3 Shared Memory Parallelization . . . . . . . . . . . . . . . . . . . 45 
  3.3 Experimental Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
    3.3.1 Performance from Shared and Distributed Memory Parallelization 49 
    3.3.2 Results from a Larger Dataset . . . . . . . . . . . . . . . . . . . 49 
    3.3.3 Impact of Preprocessing on Scalability . . . . . . . . . . . . . . 54 
  3.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

4. Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57

Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Processing Structure: FREERIDE (left) and Map-Reduce (right)</td>
<td>7</td>
</tr>
<tr>
<td>1.2</td>
<td>Memory Layout for Various Locking Schemes</td>
<td>9</td>
</tr>
<tr>
<td>2.1</td>
<td>A Four Dimensional Dataset over which Wavelet Transform Needs to be Calculated</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Sequential Discrete Wavelet Transformation Over One Time-Series</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>Function to Calculate Index of a Final Output Value</td>
<td>22</td>
</tr>
<tr>
<td>2.4</td>
<td>Parallel Performance on a Shared Memory Platform - DS4</td>
<td>23</td>
</tr>
<tr>
<td>2.5</td>
<td>Parallel Performance on a Shared Memory Platform - DS5</td>
<td>24</td>
</tr>
<tr>
<td>2.6</td>
<td>Parallel Performance on a Distributed Memory Platform - DS4</td>
<td>24</td>
</tr>
<tr>
<td>2.7</td>
<td>Parallel Performance on a Distributed Memory Platform - DS5</td>
<td>25</td>
</tr>
<tr>
<td>2.8</td>
<td>Parallel Performance With Hybrid Parallelization DS1</td>
<td>25</td>
</tr>
<tr>
<td>2.9</td>
<td>Parallel Performance With Hybrid Parallelization DS2</td>
<td>26</td>
</tr>
<tr>
<td>2.10</td>
<td>Parallel Performance With Hybrid Parallelization DS3</td>
<td>27</td>
</tr>
<tr>
<td>2.11</td>
<td>Hybrid Parallelization (Without Optimization) - DS4</td>
<td>27</td>
</tr>
<tr>
<td>2.12</td>
<td>Hybrid Parallelization (With Optimization) - DS4</td>
<td>28</td>
</tr>
<tr>
<td>2.13</td>
<td>Hybrid Parallelization (Without Optimization) - DS5</td>
<td>28</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>2.14</td>
<td>Hybrid Parallelization (With Optimization) - DS5</td>
<td>29</td>
</tr>
<tr>
<td>2.15</td>
<td>Hybrid Parallel Wavelet Transform Implementation</td>
<td>32</td>
</tr>
<tr>
<td>2.16</td>
<td>Hybrid Parallel Wavelet Transform Implementation With Optimization</td>
<td>33</td>
</tr>
<tr>
<td>3.1</td>
<td>Preprocessing in Co-clustering</td>
<td>35</td>
</tr>
<tr>
<td>3.2</td>
<td>Overview of the Co-clustering Algorithm</td>
<td>36</td>
</tr>
<tr>
<td>3.3</td>
<td>Main Computational Steps in Co-Clustering</td>
<td>38</td>
</tr>
<tr>
<td>3.4</td>
<td>Main Computational Steps in Co-Clustering (Contd)</td>
<td>39</td>
</tr>
<tr>
<td>3.5</td>
<td>Parallel Co-clustering Algorithm</td>
<td>43</td>
</tr>
<tr>
<td>3.6</td>
<td>One Iteration of Reduction Loop</td>
<td>45</td>
</tr>
<tr>
<td>3.7</td>
<td>Parallel Performance on a Shared Memory Platform - 1 GB dataset (both environments)</td>
<td>47</td>
</tr>
<tr>
<td>3.8</td>
<td>Parallel Performance on a Distributed Memory Platform - 1 GB dataset (both environments)</td>
<td>47</td>
</tr>
<tr>
<td>3.9</td>
<td>Parallel Performance With Hybrid Parallelization - 1 GB dataset (env1)</td>
<td>48</td>
</tr>
<tr>
<td>3.10</td>
<td>Parallel Performance With Hybrid Parallelization - 1 GB dataset (env2)</td>
<td>48</td>
</tr>
<tr>
<td>3.11</td>
<td>Parallel Performance With Hybrid Parallelization - 4 GB dataset (env1)</td>
<td>49</td>
</tr>
<tr>
<td>3.12</td>
<td>Parallel Performance With Hybrid Parallelization - 4 GB dataset (env2)</td>
<td>50</td>
</tr>
<tr>
<td>3.13</td>
<td>Performance of the Preprocessing Stage - 1 GB dataset (env1)</td>
<td>51</td>
</tr>
<tr>
<td>3.14</td>
<td>Performance of the Preprocessing Stage 1 GB dataset (env2)</td>
<td>51</td>
</tr>
<tr>
<td>3.15</td>
<td>Speedup Excluding Preprocessing Stage - 1 GB dataset (env1)</td>
<td>52</td>
</tr>
<tr>
<td>3.16</td>
<td>Speedup Excluding Preprocessing Stage - 1 GB dataset (env2)</td>
<td>52</td>
</tr>
</tbody>
</table>
3.17 Performance of Preprocessing Stage - 4GB dataset (env1) ............ 53

3.18 Speedup of Preprocessing Stage - 4 GB dataset (env2) ............... 53
Parallel computing is the Computer Science discipline that deals with the system architecture and software issues related to the concurrent execution of applications. It has been an area of active research interest and application for decades. It was mainly the focus of high performance computing, but now it is emerging as the prevalent computing paradigm due to the semiconductor industry’s shift to multi-core processors.

1.1 A Brief History of Parallel Computing

The interest in parallel computing dates back to the late 1950s, with advancements surfacing in the form of supercomputers throughout the 60s and 70s. These were shared memory multiprocessors, with multiple processors working side-by-side on shared data. In the mid 1980s, a new kind of parallel computing was launched when the Caltech Concurrent Computation project built a supercomputer for scientific applications from 64 Intel 8086/8087 processors [9]. This system showed that extreme performance could be achieved with mass market, off the shelf microprocessors. These massively parallel processors (MPPs) [10] came to dominate the top end of computing, with the ASCI Red supercomputer\(^1\) in 1997 breaking the barrier of one trillion floating point operations per second.

\(^1\)Please see: http://www.computermuseum.li/Testpage/ASCI-RED-Supercomputer.htm
Since then, MPPs have continued to grow in size and power. Starting in the late 80s, clusters came to compete and eventually displace MPPs for many applications. A cluster is a type of parallel computer built from large numbers of off-the-shelf computers connected by an off-the-shelf network. Today, clusters are the workhorse of scientific computing and are the dominant architecture in the data centers that power the modern information age.

Today, parallel computing is becoming mainstream based on multi-core processors. Most desktop and laptop systems now ship with dual-core microprocessors, with quad-core processors readily available. Chip manufacturers have begun to increase overall processing performance by adding additional CPU cores. The reason is that increasing performance through parallel processing can be far more energy-efficient than increasing microprocessor clock frequencies. In a world which is increasingly mobile and energy conscious, this has become essential. Fortunately, the continued transistor scaling predicted by Moore’s Law will allow for a transition from a few cores to many.

1.2 Evolution of MPI and OpenMP

The software world has been a very active part of the evolution of parallel computing. Parallel programs have been harder to write than sequential ones. A program that is divided into multiple concurrent tasks is more difficult to write, due to the necessary synchronization and communication that needs to take place between those tasks. In order to aid these processes, there emerged standards like MPI [20] and OpenMP\(^2\). MPI is a language-independent communications protocol that supports both point to point and collective communication. Most MPI implementations consist of a specific set of routines

\(^2\)Please see: http://openmp.org/wp/
(i.e., an API) directly callable from Fortran, C and C++ and from any language capable of interfacing with such routine libraries (like C#, Java or Python). It uses Language Independent Specifications (LIS) for the function calls and language bindings. The first MPI standard specified ANSI C and Fortran-77 language bindings together with the LIS. The draft of this standard was presented at Supercomputing 1994 (November 1994) and finalized soon thereafter. About 128 functions constitute the MPI-1.2 standard in its present definition. OpenMP is an implementation of multi-threading, a method of parallelization whereby the master thread (a series of instructions executed consecutively) forks a specified number of slave threads and a task is divided among them. The threads then run concurrently, with the runtime environment allocating threads to different processors. The section of code that is meant to run in parallel is marked accordingly, with a preprocessor directive that will cause the threads to form before the section is executed. Each thread has an id attached to it which can be obtained using a function (called omp_get_thread_num() in C/C++ and OMP_GET_THREAD_NUM() in FORTRAN). The thread id is an integer, and the master thread has an id of 0. After the execution of the parallelized code, the threads join back into the master thread, which continues onward to the end of the program. The number of threads for execution can be determined either statically (by environment variables) or dynamically (by a function call). By default, each thread executes the parallelized section of code independently. Work-sharing constructs can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both Task parallelism and Data parallelism can be achieved using OpenMP in this way.

However both MPI and OpenMP have their own disadvantages. Programming using MPI is generally considered to be difficult and time consuming. Given a serial program,
converting it to a parallel version using MPI is non-trivial and requires quite a few program-
ning changes. MPI programs are also harder to debug. Though gdb offers some support
for debugging MPI programs, it is not very conducive to debugging as gdb was not written
with MPI in mind. OpenMP has its disadvantages as well. It currently only runs efficiently
in shared-memory multiprocessor platforms. Programs need to be compiled using a comp-
piler that supports OpenMP. It does not allow for synchronization of subsets of threads.
Moreover, there is implicit communication between threads via shared variables, thereby
making it difficult to debug.

1.3 Evolution of MapReduce

MapReduce is a software framework introduced by Google to support distributed comput-
ing on large datasets on large clusters [5]. This framework was inspired by the map
and reduce functions commonly used in functional programming although their function
in the MapReduce framework is not the same as their original forms. MapReduce libraries
have been written in C++, C#, Erlang, Java, Python, Ruby, F#, R and other programming
languages. MapReduce is a framework for processing huge datasets on certain kinds of
distributable problems using a large number of nodes. Computational processing can occur
on data stored either in a file system (unstructured) or within a database (structured).

MapReduce has been used in parallelizing applications like web access log stats, in-
verted index construction, document clustering, machine learning, distributed sort, dis-
tributed grep and word count [5]. The map-reduce programming model can be summa-
rized as follows [6]. The computation takes a set of input points and produces a set of
output \{key, value\} pairs. The user of the map-reduce library expresses the computation
as two functions: Map and Reduce.
Map, written by the user, takes an input point and produces a set of intermediate \( \{key, value\} \) pairs. The map-reduce library groups together all intermediate values associated with the same key and passes them to the \textit{Reduce} function.

The \textit{Reduce} function, also written by the user, accepts a key and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically, only zero or one output value is produced per \textit{Reduce} invocation. Hadoop is a popular open-source implementation of MapReduce and is written in Java\(^3\).

### 1.4 FREERIDE Middleware and Parallelization Techniques

In this section, a middleware called FREERIDE (FRamework for Rapid Implementation of Datamining Engines), developed at Ohio State \([14, 15, 16]\) is described. FREERIDE offers a reduction based API, with many similarities to the map-reduce API \([6]\). Recent work has shown a large performance advantage with FREERIDE for several data mining applications \([13]\). FREERIDE was developed based upon an observation that the structure of parallel algorithms for the above problems essentially involves a generalized reduction. This observation is used to support parallelization on both distributed memory and shared memory settings. It has been used successfully for a number of common mining algorithms, including \textit{apriori} and \textit{FP-tree} based association-mining, \textit{k-means} clustering, decision tree construction, \textit{k-nearest neighbor} search \([13]\), as well as a scientific feature extraction algorithm.

There are some subtle but important differences in the API offered by FREERIDE and map-reduce. First, FREERIDE allows developers to explicitly declare a reduction object and perform updates to its elements directly, while in Hadoop/map-reduce, the reduction

\(^3\)Please see: http://hadoop.apache.org/
object is implicit and not exposed to the application programmer. Another important distin-
tinction is that, in Hadoop/map-reduce, all data elements are processed in the map step
and the intermediate results are then combined in the reduce step, whereas in FREERIDE,
both map and reduce steps are combined into a single step where each data element is
processed and reduced before the next data element is processed. This choice of design
avoids the overhead due to sorting, grouping, and shuffling, which can be significant costs
in a map-reduce implementation. Furthermore, this also alleviates the need for storage of
intermediate \((key, value)\) pairs, which can require a large amount of memory, and slows
down several data mining implementations [13].

The following functions must be written by an application developer as part of the API:

**Local Reductions:** The data instances owned by a processor and belonging to the subset
specified are read. A local reduction function specifies how, after processing one data
instance, a *reduction object* (declared by the programmer), is updated. The result of this
process must be independent of the order in which data instances are processed on each
processor. The order in which data instances are read from the disks is determined by the
runtime system.

**Global Reductions:** The reduction objects on all processors are combined using a global
reduction function.

**Iterator:** A parallel data-intensive application comprises of one or more distinct pairs of
local and global reduction functions, which may be invoked in an iterative fashion. An
iterator function specifies a loop which is initiated after the initial processing and invokes
local and global reduction functions.

Throughout the execution of the application, the reduction object is maintained in main
memory. After every iteration of processing all data instances, the results from multiple
threads in a single node are combined locally depending on the shared memory technique chosen by the application developer. After local combination, the results produced by all nodes in a cluster are combined again to form the final result, which is the global combination phase. The global combination phase can be achieved by a simple all-to-one reduce algorithm. If the size of the reduction object is large, both local and global combination phases perform a parallel merge to speed up the process. The local combination and the communication involved in the global combination phase are handled internally by the middleware and is transparent to the application programmer.

Fig. 1.1 further illustrates the distinction in the processing structure enabled by FREERIDE and map-reduce. The function Reduce is an associative and commutative function. Thus, the iterations of the for-each loop can be performed in any order. The data-structure RObj is referred to as the reduction object.
1.4.1 Shared Memory Parallelization

Initially, the challenges in enabling multi-processing for the applications FREERIDE targets are described. Consider the processing structure shown earlier in Figure 1.1. The main correctness challenge in parallelizing a loop like this on a shared memory machine arises because of possible race conditions (or access conflicts) when multiple threads update the same element of the reduction object. A number of factors make these loops challenging to execute efficiently and correctly:

- It is not possible to statically partition the reduction object so that different processors update disjoint portions of the collection. Thus, race conditions must be avoided at runtime.

- The execution time of the function Compute can be a significant part of the execution time of an iteration of the loop. Thus, runtime preprocessing or scheduling techniques cannot be applied to avoid race conditions.

- In many of algorithms, the size of the reduction object can be quite large. This means that the reduction object cannot be replicated or privatized without significant memory overheads. Moreover, because of the number of elements in the reduction object, the memory overhead of locks (or latches) can also be significant.

- The updates to the reduction object are fine-grained. The reduction object comprises a large number of elements that take only a few bytes, and the for-each loop comprises a large number of iterations, each of which may take only a small number of cycles. Thus, if a locking scheme is used, the overhead of locking and synchronization can be significant.
Two obvious approaches for addressing these problems are, Full-replication and full-locking. In the Full-replication approach, each processor can update its own copy of the reduction object and these copies are then merged together later. In the full-locking approach, one lock or latch is associated with each aggregated value. However, in our experience with data-intensive algorithms, the memory hierarchy impact of using locking when the reduction object is large was significant. It was observed that supporting a large numbers of locks results in overheads of two types. The first is the high memory requirement associated with a large number of locks. The second overhead comes from cache misses. Consider an update operation. If the total number of elements is large and there is no locality in accessing these elements, then the update operation is likely to result in two cache misses, one for the element and second for the lock. This cost can slow down the update operation significantly on modern machines with deep memory hierarchies.
To overcome these overheads, two new schemes for parallelizing data-intensive algorithms have been designed. These techniques are, Optimized-full locking, and Cache-sensitive locking [15]. The memory layout for full-locking, Optimized-full locking, and Cache-sensitive locking is shown in Figure 1.2.

**Optimized-full Locking:** Optimized-full locking scheme overcomes the large number of cache misses associated with full-locking scheme by allocating a reduction element and the corresponding lock in consecutive memory locations. By appropriate alignment and padding, it can be ensured that the element and the lock are in the same cache block. Each update operation now results in at most one cold or capacity cache miss.

**Cache-Sensitive Locking:** Another technique developed is Cache-sensitive locking. Consider a 64 byte cache block and a 4 byte reduction element. A single lock is used for all reduction elements in the same cache block. Moreover, this lock is allocated in the same cache block as the elements. So, each cache block will have 1 lock and 15 reduction elements. Cache-sensitive locking reduces both the types of overhead associated with full-locking.

### 1.5 Target Applications And Contributions

This thesis has considered the problem of parallelizing applications on clusters of multicores using a reduction based API. The first algorithm considered is the Wavelet Transform algorithm [22]. The Wavelet Transform under consideration, Haar Wavelet Transform, may be considered to simply pair up input values, storing their differences and passing their sum. This process is repeated where the differences in the sums is stored and their sum is passed over till we get $2^n-1$ differences and one final sum. The algorithm has been analyzed and it is explained how parallelizing the algorithm on a distributed environment
can be very communication intensive. A novel method of looking at the algorithm as a
generalized reduction is then proposed. It is shown that viewing this application as a re-
duction reduces the communication overhead between the threads and nodes in shared and
distributed environments. This parallel algorithm is then implemented on FREERIDE.

For the second part of the thesis, the Information Theoretic Co-clustering algorithm [7] is
considered. Co-clustering considers clustering along two dimensions, i.e., original points
can be viewed as being in a two dimensional array, and the rows are clustered into a spec-
ified number of row-clusters and the columns are clustered into a specified number of col-
umn clusters. This process repeats itself till a convergence point is reached, beyond which
the row and column clustering does not change in successive iterations. This algorithm is
parallelized on FREERIDE by representing it as a generalized reduction. The experiments
were conducted on clusters of multi-cores. Our implementations were experimentally eval-
uated on a cluster of SMPs, using different datasets. Both the algorithms scale well with
increasing number of threads and nodes. For the Parallel Wavelet Transform algorithm,
a speedup of 42 using 64 cores is reported. For the Parallel Co-clustering algorithm, a
speedup of up to 21 using 32 cores is reported.

The rest of the work is organized as follows. Chapter 2 presents the Wavelet Trans-
form algorithm, shows how parallelizing it can prove to be communication intensive and
discusses how it can be expressed as a generalized reduction. It also discusses the im-
plementation details and reports and explains the performance. Chapter 3 deals with the
Co-clustering algorithm, discusses how it was parallelized by representing it as a reduc-
tion function and reports the performance with explanations. This work is concluded in
Chapter 4.
CHAPTER 2

PARALLELIZING WAVELET TRANSFORM ON FREERIDE

This chapter focuses on an application based on the discrete wavelet transformation algorithm [22]. Wavelet transformation is an important tool in signal processing applications such as data and image compression, noise reduction, pattern recognition and fMRI processing. The algorithm itself requires several shifts of data, and indeed, the previous parallel algorithms for this problem have been communication-intensive [3, 17, 21]. These parallel algorithms do not appear suitable for mapping to a reduction-based framework.

A new approach for parallelizing the discrete wavelet transformation is developed. This makes it suitable for implementation on a reduction-based framework. It is then implemented using FREERIDE [14, 15, 16].

This work is particularly driven by medical imaging, and more specifically, functional MRI (fMRI). Medical imaging is an emerging and important area requiring large-scale data analysis\(^4\). This area includes techniques and processes used to create and analyze images of human anatomy and physiology, with the goal either being clinical use for monitoring disease and treatment, or achieving advances in medicine. fMRI is fast becoming the pre-eminent imaging method of neuro-imaging and is being used to explore and probe the

\(^4\)Please see: http://en.wikipedia.org/wiki/Medical_imaging
mechanism of brain activation [12]. Functional neuro-imaging scanners acquire spatio-temporal scalar data, with a time-series associated at every 3D voxel. The 4D time-series neuro-imaging is seldom viewed directly, but instead is subject to elaborate processing, analysis and inferencing [12, 1, 8]. The underlying algorithms essentially seek spatial-temporal patterns from noisy and poorly resolved datasets. Wavelet transform is often used as the method for noise reduction.

Since in fMRI and other similar applications, the behavior of the physical system is being studied across a time series, the wavelet transform is carried out across the time series on each of the points on the physical system. The dataset can thus be visualized as a hypercube with multiple cubes along the fourth dimension representing the time series as shown in Figure 2.1. Considering the fact that these datasets can be very large, parallelizing this application on a cluster of multi-cores will be useful.
This chapter focuses on the use of FREERIDE for creating a parallel and scalable implementa-
tion of the wavelet transform algorithm applied on a four dimensional dataset, such as those obtained from fMRI. The implementation is experimentally evaluated on a cluster of multi-cores using several different datasets. A speedup of up to 42 is reported using 64 cores.

The rest of the chapter is organized as follows. Section 2.1 describes the wavelet transform algorithm and discusses possible ways of parallelizing it. Section 2.2 describes how the wavelet transform algorithm can be expressed as a generalized reduction operation and discusses how the algorithm was parallelized on FREERIDE. Section 2.3 presents the experimental results. This work is summarized in Section 2.4.

2.1 Sequential Wavelet Transform

```c
index=0;
for(factor = NUM_TIME_SERIES/2-1; factor > 0; factor/=2) {
    for(t = 0 to factor) {
        output[index] = input[2*t] - input[2*t+1];
        input[index] = input[2*t] + input[2*t+1];
        index++;
    }
}
```

Figure 2.2: Sequential Discrete Wavelet Transformation Over One Time-Series

The Discrete Wavelet Transform is defined for a sequence of input that has \(2^n\) numbers, representing a time-series of length \(2^n\). The output of the wavelet transform is a convolution along the time domain, resulting in a series again of \(2^n\) values. The particular transform
in consideration, the *Haar wavelet transform*, can be viewed as follows. The input values are paired up, their differences are stored and their sum is *passed*. This process is repeated till there are $2^n - 1$ differences and one final sum. The algorithm is formally shown in Figure 2.2. The first statement inside the inner loop stores the difference in the output array, and the second statement *passes* over their sum to the next iteration. To save space, the values are copied on to the input array itself. Each iteration of the outer loop involves half the steps of the previous iteration. In all, there are $\log(N)$ iterations of the outer loop.

To illustrate this algorithm, consider a wavelet transform over a time series of length 8 over the elements $a_1, a_2, \ldots, a_8$. The result is the following series: $a_1 - a_2, a_3 - a_4, a_5 - a_6, a_7 - a_8, a_1 + a_2 - a_3 - a_4, a_5 + a_6 - a_7 - a_8, a_1 + a_2 + a_3 + a_4 - a_5 - a_6 - a_7 - a_8$, and finally, $a_1 + a_2 + \ldots + a_8$.

Let the time series be of the length $N = 2^n$, and the elements be denoted as $a_1, a_2, \ldots, a_N$. It can be seen that an element at the index $i$ in the input array contributes to the values at indices $i/2, i/4 + N/2, i/8 + N/2 + N/4, \ldots, N - 2$, and $N - 1$ (with output indices ranging from 0 to $N - 1$). Formally, it contributes to $n$ values, whose indices can be summarized as

$$\sum_{j=2 \ldots k} \frac{N}{2^{j-1}} + \frac{i}{2^k}$$

with $k$ ranging from 1 to $n$, and the last value, at the index $N - 1$. For example, the term $a_1$ contributes to the final values of the element at indices 0, 4, 6, and 7. Similarly, $a_3$ contributes to the output values at the indices 1, 4, 6, and 7.

A straight-forward parallelization of this algorithm for a distributed memory setting can be very communication-intensive. This can be seen from the last two values of the output. The second last value in the wavelet transform of a given input is the difference between the sum of the first $N/2$ elements and the sum of the second $N/2$ elements. The last value
of the wavelet transform is the sum of all the elements in the input series. This implies that if there are $P$ nodes, last node, which will contain the last $N/P$ values of the output, will require all the values from all the nodes. Even the first node can only calculate $N/2P$ final values with the data that it has. In order to generate the remaining $N/2P$ values, it requires the data from the second node. Besides the communication costs, another challenge is the programmability of such an application on a multi-core cluster. A hybrid programming model, i.e., combination of MPI and OpenMP, will require a high level of expertise and effort.

Map-Reduce can be used in parallelizing this application. In parallelizing the Wavelet Transformation using the map-reduce programming model, each element $a_i$ can be read once and $O(\log(N))$ \{key, value\} pairs can be generated. Here, key refers to the index of the output element, and value is either $a_i$ or $-a_i$. Once the map operation is done, the $N\log(N)$ output pairs with $N$ distinct values of key can be sorted. The reduce function will just be the summation of values.

In the given example, when $a_1$ is read, the following \{key, value\} pairs are output by the map function: \{0, $a_1$\}, \{4, $a_1$\}, \{6, $a_1$\}, and \{7, $a_1$\}. Similarly, when $a_2$ is read, the following \{key, value\} pairs are produced: \{0, $-a_2$\}, \{4, $a_2$\}, \{6, $a_2$\}, and \{7, $a_2$\}. This process continues for all the 8 values of input. The map-reduce library sorts and groups together all the intermediate values associated with the same key and passes them to the Reduce function. The reduce function will accumulate values belonging to each key by summation.

This method however, is inefficient for two main reasons. Firstly, as the size of the input time series increases, storing the $O(N\log(N))$ intermediate output pairs become extremely
expensive. For instance if there are 131072 points in the time series, the size of the intermediate \{key, value\} pairs will be 17 times the size of the input data itself. Secondly, sorting the intermediate values in a distributed environment will involve a high cost. Though some of this cost can be reduced by having a combine function within each node, the volume of intermediate data will still be unmanageable.

The parallelization approach using FREERIDE for wavelet transforms is described in Section 2.2.

### 2.2 Parallelizing Wavelet Transform Algorithm with FREERIDE

This section describes the parallelization of the wavelet transformation algorithm using FREERIDE’s reduction-based API.

The main idea is as follows. Let the time-series be of length \(T\), and let there be \(N\) nodes, so that each node has \(T/N\) consecutive values of the input. It turns out that each node is capable of producing \(T/N - 1\) values of the output by processing values locally, i.e., without involving any communication. In all, \(T - N\) of the \(T\) values can be produced without any communication, whereas the remaining \(N\) values involve interprocess communication. Thus, a reduction object of size \(N\) can be allocated and initialized on each node, and each node can update it with its own contributions towards the final values. Subsequently, a global reduction can be performed to obtain these final \(N\) values.

One issue with this approach is that the output series elements are not going to be consecutive. To allow construction of the final transformed series, the final output index is computed and stored with each value.
This approach is explained with the help of an example. Let us consider the following input sequence: \( a_1, a_2, \ldots, a_{32} \) for which the wavelet transform needs to be calculated. Let us assume the following: Let the number of nodes be 2, and the let there be 4 threads per node. This implies that node 0 has values \( a_1, \ldots, a_{16} \) and node 1 has the values \( a_{17}, \ldots, a_{32} \). Each thread will be responsible for 4 values. In this case, the size of the reduction object during a local accumulation phase, i.e., for finalization of values across the threads on each node, is \#Threads, or 4.

The parallel algorithm can proceed with the following steps:

1. Thread 0 can compute the values \( a_1 - a_2, a_3 - a_4, a_1 + a_2 - a_3 - a_4 \). It then calculates \( a_1 + a_2 + a_3 + a_4 \) and updates the first value of its local copy of the reduction object with this value. It also sets the remaining values in its copy of the reduction object to 0. Thread 1 simultaneously calculates \( a_5 - a_6, a_7 - a_8, \) and \( a_5 + a_6 - a_7 - a_8 \) locally. Then, it updates the second value of its copy of the reduction object with the value \( a_5 + a_6 + a_7 + a_8 \), and sets the remaining values to zero. Threads 2 and 3 compute their local values similarly and update their reduction objects with \( a_9 + a_{10} + a_{11} + a_{12} \) and \( a_{13} + a_{14} + a_{15} + a_{16} \), respectively.

2. All the threads call an accumulate function whereby the reduction objects of all the threads are accumulated, and one copy of the reduction object is created on each node. Here, the master thread accesses the reduction object and runs the same algorithm for the data on the reduction object. This step produces 3 local final values and one value, which is the sum of all the 16 elements of the reduction object. This value now needs to be communicated among the nodes.

3. We now need to accumulate the values across nodes to determine the last \#Nodes values of the output series. For this, a global communication step is performed where each
node updates its corresponding part on the global array and all the values are combined to form the global reduction object.

The details of these steps are explained in the following section. The initial implementation is also shown in Figure 2.15. As explained earlier, the implementation is driven by the needs of wavelet transforms on fMRI data. Thus, a 3-dimensional space (volume) is considered, where a time-series is computed for each voxel.

### 2.2.1 Shared Memory Parallelization

Consider parallelization across threads within a single node. The input values are shared between the threads within a node. The access of each thread is restricted to its corresponding $N_t = \#\text{Input elements}/\#\text{Threads}$ elements. As explained above, each thread can calculate the $N_t - 1$ values of the output independently. Each thread also contributes the sum of all the elements to one element of the reduction object. This is done by the accumulate call in the Figure 2.15.

At this stage, we have the $\#\text{Threads}$ partial contributions of each thread towards the last $\#\text{Threads}$ values. The next step in shared memory parallelization is to extract these $\#\text{Threads}$ partial values from the appropriate positions in the reduction object and run the same algorithm on this data. This will produce the last $\#\text{Threads}$ values of the output series.

### 2.2.2 Distributed Memory Parallelization

Now, consider the case when we use one core on each node, but have several nodes. The parallelization is quite similar to the shared memory parallelization. The input data is split among the nodes and each node does the same work that would be done by each of the threads in the shared memory case. Each node calculates the final values that it can
calculate locally, and then updates the reduction object with the calculated values in its corresponding position on the reduction object. This is followed by a global combination step, where the reduction objects of all the nodes are accumulated. This is shown as the global_accumulate call in Figure 2.15.

Since each node calculates $N_p - 1$ values, where $N_p = \text{Input elements} / \text{Nodes}$, there are still $\text{Nodes}$ values that need to be calculated. For this, the sum of all values of each node is retrieved from the reduction object. The wavelet transform is again run on these retrieved values in the master node to generate the last $\text{Nodes}$ output values.

### 2.2.3 Hybrid Parallelization and an Optimization

In a hybrid environment, the input data is distributed among the different nodes and multiple threads within each node share the data.

The version of FREERIDE initially available for this study only allowed programmers to declare a single reduction object, for both shared memory and distributed memory parallelization. Thus, the size of the reduction object for the initial implementation (Figure 2.15) was $\text{Nodes} \times \text{Threads}$. This denotes the number of values in the output that need to be accumulated across nodes or threads.

After processing on each thread, each thread updates a part of the reduction object. It turns out that threads within the first node update only the first $\text{Threads}$ values, the threads within the second node only update the next $\text{Threads}$ values, and so on. However, because of the current API and implementation, the entire reduction object is first accumulated on each node, and then communicated from each node to a master node. All values that could not be calculated by processing on each thread are sequentially computed on the master node.
Recall that the wavelet transform is being performed over time-series in a 3D volume. Thus, the total size of the reduction object can get very large. Moreover, an unnecessarily large fraction of the processing is performed sequentially on each thread. To help reduce the communication, an optimization was performed, which required some changes to the FREERIDE API. The reduction object that accumulates values across threads within a node is separated from the reduction object that needs to be communicated across threads. Thus, for each time series, only $\#Nodes$ values need to be communicated from each node to a central node.

The optimized implementation (Figure 2.16) works as follows. Each thread contributes the sum of all its values to the reduction object, which gets accumulated across the different threads in a local combination step (accumulate call in Figure 2.16). After this local combination within each of the nodes, the wavelet transform algorithm is again run on the partial data from the accumulated reduction object. This gives us all the output elements that could be computed locally within each of the nodes.

To compute the parts of the wavelet transform that involve data from more than one node, these nodes need to update their values in another part of the reduction object, represented by a Global Array. Each node fills in its corresponding values block based on its id. This is done by the global_accumulate call in Figure 2.15. It sets the remaining values of its copy of the Global array to zero. This step is followed by a global combination step, where the global arrays of all the nodes are aggregated.

After the global combination step, the master node has all the final output values and the $\#Nodes$ partial values for each time-series. This is used to calculate the last $\#Nodes$ values for the wavelet transform for each of the points in the 3D voxel.

Figure 2.16 shows how the hybrid parallelization technique with optimization works.
calculate_index_function()
{
    NumValPerChunk = NumTimeSeries / (#Nodes × #Threads)
    term = t + NodeID × NumTimeSeries / #Nodes
    ChunkID = NodeID * #Threads + ThreadID
    index = term + (ChunkID × NumValPerChunk)/2^{Iteration+1}
    if(Iteration=0)
    {
        return index
    }
    else
    {
        for(i = 1 to Iteration)
            index+= N/2^i
    }
    return index
}

Figure 2.3: Function to Calculate Index of a Final Output Value

2.2.4 Generating Index For Output Values

It was shown that an input value at index $i$ will contribute to values at indices $i/2$, $i/4+N/2$, $i/8+N/2+N/4$, $\ldots$, $N−1$, and $N$. This, together with the node identified can be used to generate the index in the output array where the current value being calculated should go. This process is shown in Figure 2.3.

2.3 Experimental Results

In this section, the performance of the parallel implementation of the wavelet transformation algorithm is evaluated, focusing on parallel scalability on both distributed and
shared memory settings. The experiments were performed on a cluster of multi-core machines, where each node comprises a dual Intel Xeon CPU E5345, with a quad-core CPU. Thus, each node has 8 cores. Each core has a clock frequency of 2.33 GHz and each node has 6 GB main memory. Eight such nodes in the cluster are connected by Infiniband.

Experiments were conducted on datasets of different sizes and characteristics by varying two parameters of the dataset, the size of each dimension of the spatial cube \( p \) and the number of time-steps in the time-series \( s \). Thus, the dataset size is \( p^3 \times s \). Five datasets were generated with the following dimensions.

1. \( p = 10; \ s = 262144 \) (**DS1**)
2. \( p = 32; \ s = 2048 \) (**DS2**)
3. \( p = 32; \ s = 4096 \) (**DS3**)
4. \( p = 32; \ s = 8192 \) (**DS4**)
5. \( p = 39; \ s = 8192 \) (**DS5**)
Figure 2.5: Parallel Performance on a Shared Memory Platform - DS5

Figure 2.6: Parallel Performance on a Distributed Memory Platform - DS4
2.3.1 Performance of Shared and Distributed Memory Parallelization

In this subsection, the speedups achieved from shared memory parallelization mode (1 multi-core node) and distributed memory mode (multiple nodes with 1-core each), were evaluated separately. The goal was to understand the scalability achieved with each mode of parallelization. Results from the datasets DS4 and DS5 are reported in this subsection.
Results from Shared Memory Mode: Figures 3.7 and 2.5 show the scalability on datasets DS4 and DS5, respectively. The results are very similar, with speedups of 1.98/1.99 for 2 threads, 3.37/3.40 for 4 threads and 5.50/5.47 for 8 threads. While the application is scaling with increasing number of threads, it is not linear. This is because of the need for combining $O(p^3)$ values, and also because of limited memory bandwidth available on each node, which slows down the processing when 4 or 8 cores are used.

Results from Distributed Memory Mode: Figures 3.8 and 2.7 show scalability of the parallel implementation in distributed memory platform for DS4 and DS5 datasets, respectively. In this case, only 1-core from each node is used. The overall scalability now is very close to linear, and better than the shared memory case, despite the need for communicating $O(p^3)$ values from each node, and combining them. This further confirms that the shared memory parallelization is limited because of the limited memory bandwidth on each node.
Figure 2.10: Parallel Performance With Hybrid Parallelization DS3

Figure 2.11: Hybrid Parallelization (Without Optimization) - DS4
Figure 2.12: Hybrid Parallelization (With Optimization) - DS4

Figure 2.13: Hybrid Parallelization (Without Optimization) - DS5
2.3.2 Scalability of Hybrid Parallelization

The results from hybrid distributed-shared memory platform for the datasets DS1, DS2, and DS3 are shown in Figures 2.8, 2.9, and 2.10, respectively. In this set of experiments, both the number of nodes as well as the number of cores used within each node were varied. The dataset DS1 is very distinct from DS2 and DS3, since it involves much fewer spatial coordinates, but has a longer time-series. The size of the reduction object, and therefore the volume of communication, is much smaller for DS1.

This distinction is reflected clearly in the results. Datasets DS2 and DS3 show overall speedups of 16.3 and 19.2 for 64 cores, i.e., 8 nodes running 8 threads each. In comparison, for the dataset DS1, an overall speedup of about 42 for 8 nodes running 8 threads each is achieved. This is because, for both DS2 and DS3, the communication time dominates over the computation time. Recall that the total computation is $O(p^3s)$, whereas the volume of communication per node or thread is $O(p^3)$. Since the time for communication
increases with increase in number of threads, there is a slight increase in time taken for 64 threads as opposed to 32 threads (8 nodes running 4 threads each).

For the results from the remaining two datasets, i.e., DS4 and DS5, the impact of the optimization performed for reducing the size of the reduction object is evaluated.

The performance of hybrid parallelization with and without optimization is shown in Figures 2.11 and 2.12 for DS4, and for DS5, they are shown in Figures 2.13 and 2.14. Without optimization, the performance increases on multiple nodes up to two threads. Beyond two threads, the size of the reduction object becomes bigger and this results in more communication and more contention between threads and between nodes. This limits the increase in performance for 4 and 8 threads. Without optimization, the speedup achieved on 64 cores is around 8 for both DS4 and DS5. With the Optimization, the speedup increases to 25.08 for DS4 and 24.45 for DS5.

The improvement in performance after the optimization is because each thread and each node updates the reduction object with just one value for every time series in the dataset. The final values that each thread calculates are stored locally and are not communicated across threads or nodes. This reduces the communication time significantly and hence the performance increase.

It should be noted that the performance is almost the same with and without optimization for shared and distributed memory systems. This is because the size of the reduction object is almost the same with and without the optimization in both cases.

The overall speedup of optimized versions for DS4 and DS5 are lesser than that of DS1. This is because both DS4 and DS5 have a bigger reduction object when compared to DS1, their speedups are not as high as that of DS1. Moreover, DS5 has a lesser speedup when
compared to DS4. This is consistent with the fact that DS5 has a slightly bigger reduction object.

2.4 Conclusions

This chapter reports on the parallelization of the Wavelet Transform algorithm using the FREERIDE middleware. It is shown how the Wavelet Transform algorithm can be expressed as a generalized reduction. The algorithm can thus be parallelized using FREERIDE. Using the same high-level, reduction based API, both shared memory and distributed memory parallelization are achieved. This work reports a speedup of up to 42 with 8 nodes running 8 threads each (i.e. 64 threads).
\* \* dimX, dimY and dimZ: \#Elements along X, Y and Z axes \* \* Start = ThreadID * (#Time Series per Node/#Threads) \* \* End = (ThreadID + 1) * (#Time Series per Node/#Threads)-1 \* \* Size of Reduction Object = #Threads × #Nodes \* 

red\_duction
{
  \text{index} = 0;
  \text{for}(i = 0 \text{ to dimX})
  \text{for}(j = 0 \text{ to dimY})
  \text{for}(k = 0 \text{ to dimZ})
    \text{for(factor = End; factor > Start ; factor = (Start + End)/2)
      \text{for}(t = Start \text{ to factor})
        \text{output(index, i,j,k) = input(2*t,i,j,k) - input(2*t+1,i,j,k);
          input(index, i,j,k) = input(2*t,i,j,k) + input(2*t+1,i,j,k);
          output_index[index] = calculate\_index\_function();
          index++;}
  \}
  \{ \* output(index-1,i,j,k) has partial sums \* \* and all other values are final values \* \}

  \{ \* Accumulating values across threads and nodes \* \}
  \text{for}(i = 0 \text{ to dimX})
  \text{for}(j = 0 \text{ to dimY})
  \text{for}(k = 0 \text{ to dimZ})
    \{ 
      \{ \* global\_accumulate(offsets, value) \* \}
      \text{global\_accumulate(ThreadID+(NodeID*#Threads),
        i,j,k,output(end,i,j,k));}
    \}
  \{ \* Accumulated data has partial contributions of \* \* all threads from all nodes \* \}
  \{ \* Perform W.T. On Data On Reduction Object to calculate \* \* the last #Threads × #Nodes output values \* \}
}

\text{Figure 2.15: Hybrid Parallel Wavelet Transform Implementation}
reduction() {
    index = 0;
    for(i = 0 to dimX)
        for(j = 0 to dimY)
            for(k = 0 to dimZ)
                for(factor = End; factor > Start; factor = (Start + End)/2)
                    for(t = Start to factor)
                        output(index,i,j,k) = input(2*t,i,j,k) - input(2*t+1,i,j,k);
                        input(index,i,j,k) = input(2*t,i,j,k) + input(2*t+1,i,j,k);
                        output_index[index] = calculate_index_function();
                        index++;
    for(i = 0 to dimX)
        for(j = 0 to dimY)
            for(k = 0 to dimZ)
                accumulate(ThreadID,i,j,k,output(index-1,i,j,k));
    for(i = 0 to dimX)
        for(j = 0 to dimY)
            for(k = 0 to dimZ)
                global_accumulate(nodeid,i,j,k,acc_data(#Threads-1,i,j,k));
}

Figure 2.16: Hybrid Parallel Wavelet Transform Implementation With Optimization
CHAPTER 3

PARALLELIZATION OF INFORMATION THEORETIC CO-CLUSTERING ON FREERIDE

Over years, the data mining area has seen development of several new approaches and algorithms. A popular new concept has been co-clustering [7]. Co-clustering can be viewed as a generalization of the clustering problem, which is one of most widely studied data mining problems. Popular clustering algorithms like k-means have been parallelized using map-reduce as well as FREERIDE in the past [13]. Co-clustering considers clustering along two dimensions, i.e., original points can be viewed as being in a two dimensional array, and the rows are clustered into a specified number of row-clusters and the columns are clustered into a specified number of column clusters. Many types of co-clustering algorithms have been proposed based on cluster shapes, the properties of the input data and optimization objectives. Comparison of different co-clustering approaches is available from two recent survey papers [18, 19]. Different optimization criteria have been proposed for co-clustering, such as minimum mutual information [7], sum-squared distance [4] and code length [2].

This chapter focuses on the use of FREERIDE for creating a parallel and scalable implementation of an Information Theoretic Co-clustering algorithm. It is shown that each iteration of the co-clustering algorithm essentially involves two generalized reduction loops.
It is shown how FREERIDE can be used for parallelizing the algorithm on cluster of multi-cores. The implementation is experimentally evaluated on a cluster of SMPs, using different datasets. A speedup of up to 21 using 32 cores is reported.

The rest of the chapter is organized as follows. Section 3.1 gives a brief overview of the Co-clustering algorithm. Section 3.2 describes how the co-clustering algorithm can be viewed as a series of reductions and how it is parallelized on distributed and shared memory systems. Section 3.3 presents the results from the experimental evaluation. This work is summarized in Section 3.4.

3.1 Co-clustering algorithm

```
Preprocess() {
  /* Assign Row and Column clusters */
  /* Compute Cluster Sizes */
  for i = 0 to NUM_ROW {
    for j = 0 to NUM_COL {
      pX[i] += value[i][j]
      pY[j] += value[i][j]
    }
  }
  /* normalize pX and pY */
  /* normalize value matrix */
  for i = 0 to NUM_ROW {
    for j = 0 to NUM_COL {
      xnorm[i] += value[i][j] \times \log (value[i][j])
      ynorm[j] += value[i][j] \times \log (value[i][j])
    }
  }
  /* Compute mutual info */
}
```

Figure 3.1: Preprocessing in Co-clustering
DO {
    {* Row Iteration *}
    Compute Rows Centroid();
    Reassign Row Clustering();
    Compute Compressed Matrix();
    Compute Cluster Centroids();
    Compute Objective Function();

    {* Column Iteration *}
    Compute Column Centroid();
    Reassign Column Clustering();
    Compute Compressed Matrix();
    Compute Cluster Centroids();
    Compute Objective Function();
} While (oldObjValue - newObjValue) > Threshold

Figure 3.2: Overview of the Co-clustering Algorithm

This section gives a brief overview of the Co-clustering algorithm [7]. Co-clustering is a data mining technique involving simultaneous clustering of rows and columns of a matrix. Unlike clustering algorithms like k-means [11], which classify points based on their distances from the centroid of a cluster, the Co-clustering algorithm clusters rows and columns into row clusters and column clusters, respectively. The criterion for performing this clustering is that their mutual information should be maximized. As a result, this algorithm generates bi-clusters, that is, a group of rows exhibiting similar behavior across a group of columns, and vice-versa.

The main steps of this algorithm are now described. The algorithm broadly comprises a preprocessing step (Figure 3.1) and a main iterative procedure (Figure 3.2). In the preprocessing step, the values for row cluster and column cluster are initialized and the row
cluster and column clusters’ sizes are calculated. After this, the following values are calculated: \( p_X \), which is the sum of all the values of each row of the matrix normalized by the total sum of all entries of the matrix. \( p_Y \) is the same for columns. Next, \( x_{\text{norm}} \) is the sum of product of values of each element of the row with the log of itself, for each row. \( y_{\text{norm}} \) is the same for columns.

Figure 3.2 gives an overview of the main computational steps in the algorithm. The algorithm works by fixing a column clustering and then iterating over the rows assigning each row to a cluster based on its Kullback-Leibler Divergence from the cluster. For probability distributions \( P \) and \( Q \) of a discrete random variable, the Kullback Leibler divergence of \( Q \) from \( P \) is defined as:

\[
D_{KL}(P \| Q) = \sum_i P(i) \times \log(P(i)/Q(i))
\]
computeRowsCentroid() {
    for r = 0 to NUM_ROW {
        rc = rowCL[r]
        for c = 0 to NUM_COL {
            cc = colCL[c]
            qYxhat[rc][c] += compressed_matrix[rc][cc] ×
            (pX[r]/pshat[rc]) × pY[c]/pyhat[c]
        }
    }
    {/* normalize qYxhat */}
}

Computing Row Centroid Function

reassignRC() {
    for r = 0 to NUM_ROW {
        minCL = rowCL[r]
        minDistance = ∞
        for rc = 0 to NUM_COL {
            tempDistance = KL_Divergence(qYxhat[rc],r)
            if tempDistance < minDistance {
                minCL = rc
                minDistance = tempDistance
            }
        }
    }
}

Reassigning Row Clustering

Figure 3.3: Main Computational Steps in Co-Clustering
Each row is thus assigned to the cluster from which its Kullback-Leibler divergence is minimum. Fixing the row clusters defined in the initial step, a similar iteration on columns is performed where the columns are assigned clusters based on their Kullback Leibler divergence from the clusters.

```java
computeCompressedMatrix() {
    for i = 0 to NUM_ROW {
        for j = 0 to NUM_COL {
            Acompressed[rowCL[i]][colCL[j]] += value[i][j]
        }
    }
}
```

**Computing Compressed Matrix Function**

```java
computeMarginal() {
    for i = 0 to Number of row clusters {
        for j = 0 to Number of column clusters {
            pxhat[i] += Acompressed[i][j]
            pyhat[j] += Acompressed[i][j]
        }
    }
}
```

**Computing Marginal Function**

Figure 3.4: Main Computational Steps in Co-Clustering (Contd)
Based on the new clustering, an objective function can be determined to estimate whether an improvement on clustering was achieved or not. This process is repeated till a convergence point, determined by the objective value, is reached. Several of the main computational steps involved in the process are shown in Figure 3.3 and Figure 3.4.

3.2 Parallelizing Co-clustering with FREERIDE

This section describes the parallelization of Co-clustering using FREERIDE. For using FREERIDE, an application must comprise one or more steps involving generalized reductions. Therefore, an initial discussion of how the Co-clustering algorithm meets this requirement is made.

Since the algorithm involves both row and column computations, a transpose of the matrix is precomputed. Both the original matrix and the transpose are divided into several files and distributed among the nodes. These files are distributed among the nodes so that each node has equal amount of row data and column data.

3.2.1 Co-clustering Algorithm and Generalized Reductions

This section discusses how the processing in the co-clustering algorithm can be viewed as a series of generalized reductions.

Initially, we start with the preprocessing phase of the algorithm. Consider the preprocessing part of the sequential Co-clustering algorithm shown in Figure 3.2. Since the arrays $pX$ and $pY$ need to be normalized by the total sum of all the elements in the array, we need to wait till all the chunks of data have been processed by all the nodes before we can normalize $pX$ and $pY$. Each node computes the $pX$ and $pY$ values corresponding to the row and column data it has. The partial sum from each chunk of data and the corresponding
part of $pX$ and $pY$ are updated in the reduction object. After all the partial sums are accumulated in the finalize part of the reduction loop, all values of $pX$, $pY$ and total sum are obtained and the values of $pX$ and $pY$ are normalized. Since the values for $xnorm$ and $ynorm$ require the value of total sum (normalized input value), this cannot be done in the first iteration of reduction and so preprocessing itself takes two iterations of reduction.

The next step is the calculation of the compressed matrix which is used to determine convergence. This compressed matrix is of dimension $\#rowclusters \times \#columnclusters$ and has the sum of all the values belonging to each row cluster across each column cluster. A local compressed matrix is built for each chunk of data. This matrix needs to be a part of the reduction object as well, and gets accumulated. Since the required compressed matrix is the sum of all the local compressed matrices, the reduction object, after all reduction operations, contains the compressed matrix. The overall reduction object is shown in Table 3.1.

The next step is the calculation of row cluster centroids based on the current redistribution of the row clusters. This is calculated after all the $rowCL$ values are accumulated across the nodes.
The objective function calculates a loglikelihood value from the compressed matrix, which is the objective value used in determining convergence. Since this requires the accumulated compressed matrix, and is the same function for both row clustering and column clustering, it can be performed as part of the global combination of the reduction loop. Particularly, all row cluster values can be accumulated across the nodes, and all nodes now have a replicated copy of the updated row clusters. The next value to be accumulated is the compressed matrix, which in turn is followed by calculation of row cluster centroids. The objective function mentioned earlier, is then calculated.

These accumulated values are used in the next iteration, which will be the column iteration. It is to be noted that both row iterations and column iterations have the same functions. Therefore, column iterations can be parallelized just like row iterations. To summarize, the parallel loops in the Co-clustering algorithm match the structure of generalized reductions and FREERIDE is well suited for parallelizing this algorithm.

### 3.2.2 Distributed Memory Parallelization

In view of the discussion above, it is easy to see how the Co-clustering algorithm can be parallelized on distributed memory machines.

The input data instances are distributed between the nodes. The initial row and column clusters are replicated on all nodes. The row clustering step is carried out on all nodes, using the set of data instances they have. After such local processing, a *global combination* is done. In this step, all nodes communicate their values of row clusters, compressed matrix and row centroids to one node, which aggregates these values and broadcasts them to all nodes.
Next, each node computes new values of column clusters, using the updated value of row clusters. Another round of global combination is done to accumulate the column clusters.

Finally, each node computes the objective value and each node independently decides if another iteration is needed. If another iteration is needed, the updated values of row clusters, column clusters and the compressed matrix are used in the successive iteration.

The above steps can be implemented easily using the FREERIDE interface. Two different local processing functions are created, corresponding to the processing in row and column clustering steps, respectively. There is one global combination function associated with each of these. The iterator is responsible for checking the terminal conditions, and
performing all computations that are replicated on all the nodes. The overall processing is shown in Figure 3.5.

Some other important details of implementation are as follows. Initially the rows are assigned clusters in a round-robin fashion. The arrays row\text{CL} and col\text{CL} are replicated in all the nodes. All nodes do this assignment locally. Since the assignment of initial clusters is in a round-robin fashion, this ensures that the initial clustering remains same across different nodes and cores. The size of each row cluster and column cluster can also be thus computed locally as a part of preprocessing.

It is to be noted that we have the centroid of each row and the centroid of all of the clusters locally within each node and core, and allocating rows to clusters requires just these values. The process of reassigning row cluster and column cluster fits well into the reduction structure. As each chunk is processed, each row is assigned to the cluster for which the Kullback-Leibler divergence of the row is minimum. After each chunk is processed, the reduction object is updated with the new row clusters by zeroing out parts other than the current chunk’s rows. Since in a distributed system, it is not possible to determine which part of the original array the current chunk belongs to, all the data chunks have a chunk id as their first value. This chunk id is used to calculate a value called the row\text{Start} or col\text{Start} which determines the starting index of the chunk in the original dataset. Thus, we have

\[
row\text{Start} = \text{Chunk}\_id \times (\text{NUM\_ROWS}/\text{NUM\_FILES})
\]

and

\[
col\text{Start} = \text{Chunk}\_id \times (\text{NUM\_COLS}/\text{NUM\_FILES})
\]
The `rowStart` value is used to update the row cluster values. The range of values to be updated for each chunk will thus be `rowStart` to 

\[(\text{Chunk}_i + 1) \times \frac{\text{NUM\_ROWS}}{\text{NUM\_FILES}}\].

### 3.2.3 Shared Memory Parallelization

If multiple CPUs are available within a node, each of these can be used for local processing. A set of data instances can be assigned to each thread, which can perform the computation associated with these data instances. The main challenge in maintaining correctness arises because of race conditions when multiple threads may try to update the same array. Earlier, several techniques that are supported in FREERIDE to support shared memory parallelization were listed.

Among these techniques, cache-sensitive locking was used for co-clustering. This approach keeps the memory overheads quite low, and is scalable to increasing number of cores.
within a node. The overall parallelization structure, combining both shared and distributed memory processing, is shown in Figure 3.6.

3.3 Experimental Results

In this section, the performance of the parallel implementation of the co-clustering algorithm is evaluated. The primary goals of the experimental study was to study the parallel scalability of the implementation, in both distributed and shared memory settings. Scalability experiments were performed on two different clusters of multi-core machines. Each multi-core machine on the first cluster contains an Intel Xeon CPU E5345, comprising a quad-core CPU. Each CPU core has a clock frequency of 2.33 GHz and each node has a 6 GB main memory. Eight such nodes in the cluster are connected by Infiniband. This environment is referred to as \textit{env1} throughout this section. On the second cluster, each node contains AMD opteron 8350 CPU with 8 cores. The amount of main memory on this machine is 16 GB. Four such nodes in the cluster are connected by Infiniband. This environment is referred to as \textit{env2} throughout this section.

Experiments were conducted on datasets of two different sizes. The datasets used are 2-dimensional square matrices of size 1 GB and 4 GB. Specifically, the data matrices had a row and column configuration of $16K \times 16K$ and $32K \times 32K$, respectively. The number of row and column clusters were set to 4 in both the cases. The dataset was split into 32 files based on row partitioning. The transpose of this matrix was also precomputed and stored in another set of 32 files.
Figure 3.7: Parallel Performance on a Shared Memory Platform - 1 GB dataset (both environments)

Figure 3.8: Parallel Performance on a Distributed Memory Platform - 1 GB dataset (both environments)
Figure 3.9: Parallel Performance With Hybrid Parallelization - 1 GB dataset (env1)

Figure 3.10: Parallel Performance With Hybrid Parallelization - 1 GB dataset (env2)
3.3.1 Performance from Shared and Distributed Memory Parallelization

In this subsection, the speedups achieved from shared memory parallelization mode (1 multi-core node) and distributed memory mode (multiple nodes with 1-core each) are evaluated separately, to understand the scalability for each mode of parallelization. A dataset of size 1 GB (16K × 16K) was used for the experiments in this subsection.

3.3.2 Results from a Larger Dataset

Figure 3.11: Parallel Performance With Hybrid Parallelization - 4 GB dataset (env1)

Results from Shared Memory Mode: Figure 3.7 shows the scalability of the parallel implementation on both the shared memory platforms. In the first environment, the speedups were 1.99 for 2 threads and 3.59 for 4 threads. Similarly, in the env2, a speedup factor of 5.28 was achieved from 8 threads. The reasons for sub-optimal performance with
the increase of threads are as follows. It should be recollected that the shared memory parallelization technique used was cache-sensitive locking. While cache-sensitive locking reduces the memory requirements for the locks, the contention for the acquisition of the lock increases with increase in the number of threads. Thus, increase in contention leads to degradation of performance at 4 and 8 threads for two environments, respectively.

**Results from Distributed Memory Mode:** Figure 3.8 shows scalability of the parallel implementation in distributed memory platforms. In this case, only 1-core from each node is being used. The results show that a near-linear speedup is achieved on 8 and 4 nodes. On 8 nodes (env1), a speedup factor of 7.8 was achieved. On 4 nodes (env2), a speedup of 3.6 was achieved. It should be noted that on a distributed memory setup, each node has its own copy of reduction objects. Also, since there is only one thread within each node there is no contention for the reduction elements within a single node.
Figure 3.13: Performance of the Preprocessing Stage - 1 GB dataset (env1)

Figure 3.14: Performance of the Preprocessing Stage 1 GB dataset (env2)
Figure 3.15: Speedup Excluding Preprocessing Stage - 1 GB dataset (env1)

Figure 3.16: Speedup Excluding Preprocessing Stage - 1 GB dataset (env2)
Figure 3.17: Performance of Preprocessing Stage - 4GB dataset (env1)

Figure 3.18: Speedup of Preprocessing Stage - 4 GB dataset (env2)
Results from Hybrid Distributed-Shared Memory Platforms: The results from hybrid distributed-shared memory platform are shown in the Figures 3.9 and 3.10. In this set of experiments, both the number of nodes as well as the number of cores used within each node were varied. On 8 nodes with 4 threads (env1) each a speedup of 12.17 was achieved. On 4 nodes with 8 threads each (env2), a speedup of 9.38 was achieved.

The main observations are as follows. There is a consistent performance gain obtained with the increase in the number of nodes, however the scalability is sub-linear. When both the distributed and shared memory parallelizations are combined, the overheads from both the setups are introduced and limits the overall scalability of the application. While the conflicts increase with shared memory, the communications performed by different nodes also need to wait until all the threads in the nodes are synchronized.

The results from a dataset size of 4GB (32K × 32K) are reported with focus only on results from the hybrid distributed-shared memory parallelization. The experimental results are shown in Figures 3.11 and 3.12 for two different environments.

On 8 nodes with 4 threads each (env1), a speedup of about 20.7 was achieved. On 4 nodes with 8 threads each (env2), a speedup of 19.5 was achieved. In both the environments, the scalability has significantly increased when compared to the results with smaller dataset size. This is mainly because of the increase in computation with the increase in problem size.

3.3.3 Impact of Preprocessing on Scalability

As mentioned earlier in Section 3.2, the preprocessing stage of the algorithm requiring the calculation of totalsum, xnorm, and ynorm has been mapped into two different iterations of generalized reduction loops. However, the computations involved with these
two iterations are not very high, possibly creating a bottleneck in the speedup. Hence, the scalability of these two iterations was analyzed to understand its impact on the overall performance of the application.

Figures 3.13 and 3.14 report the results from time taken only for the preprocessing steps of the algorithm for the smaller dataset (1 GB) in both the environments. It can be clearly observed that preprocessing steps have good scalability only until the use of 2 nodes and 4 threads. Since, it is not compute-intensive, further addition of nodes (4 and 8 nodes) does not result in any noticeable performance benefit. Thus, these two preprocessing steps could turn out to be a bottleneck for the overall performance of the application. To support this argument, the results that report the execution times of the application without the preprocessing steps are presented in Figures 3.15 and 3.14. It can be observed from the results that the scalability obtained after removing the preprocessing, is much better than the one that includes preprocessing. In fact, on env1, for 4 nodes and 8 nodes, the speedup achieved increased to 18.75, from 12.17 for the entire application (i.e. with preprocessing).

Next, the impact of the preprocessing stage on the performance achieved on the larger dataset, is considered. The performance of the preprocessing stage for the larger dataset is shown in Figures 3.17 and 3.18. The speedup of the preprocessing stage is similar to the overall performance, and thus, the preprocessing stage is no longer a bottleneck with the larger dataset.

Discussion: To summarize, the performance gains obtained from parallelizing the co-clustering algorithm are presented. The experiments were performed on two different environments, and on two different datasets, one with a size of 1 GB and the other with a size of 4 GB. With the smaller dataset, there is not much computation in the preprocessing stage,
and thus this stage turns out to be a bottleneck. However, with the larger dataset, preprocessing stages scales as well as the other computational steps and hence is not a bottleneck. Overall, on the first environment, with 8 nodes and 4 cores each, a speedup of 20.7x was achieved relative to the 1-thread version. On the second environment, with 4 nodes and 8 cores each, a speedup of 19.5 was achieved.

3.4 Summary

This chapter reports on the parallelization of a Co-clustering algorithm using the FREERIDE middleware. It is shown that the main processing loops in both the row and column clustering of the Co-clustering algorithm essentially involve a generalized reduction, and therefore, the algorithm can be parallelized using FREERIDE. Using the same high-level, reduction based API, both shared memory and distributed memory parallelization are achieved. A speedup of up to 21 is reported with 8 nodes having 4 cores each (i.e. 32 cores).
CHAPTER 4

CONCLUSION

This thesis aims at parallelizing data mining and data intensive algorithms on a reduction based framework. A middleware called FREERIDE (FRamework for Rapid Implementation of Datamining Engines) is used. FREERIDE offers a reduction based API, with many similarities to the map-reduce API. Previous work however, has shown a large performance advantage with FREERIDE for several data mining applications. FREERIDE is based upon an observation that the structure of parallel algorithms for the a lot of data mining problems essentially involves a generalized reduction.

Two applications have been parallelized using this framework. The first algorithm considered is the Wavelet Transform algorithm. Wavelet Transformation is used in various signal processing applications like noise reduction, fMRI analysis, image compression and so on. The Wavelet Transform under consideration, Haar Wavelet Transform, may be considered to simply pair up input values, storing their differences and passing their sum. This process is repeated where the differences in the sums is stored and their sum is passed over till we get $2^n-1$ differences and one final sum. The algorithm has been analyzed and it is shown that parallelizing the algorithm on a distributed environment can be very communication intensive. A novel method of parallelizing this application is proposed. It involves looking at the algorithm as a generalized reduction. It is shown that viewing this application as a
reduction reduces the communication overhead between the threads and nodes in shared and distributed environments. This algorithm is then implemented on FREERIDE.

The next algorithm considered is the Information Theoretic Co-clustering algorithm. Co-clustering considers clustering along two dimensions, i.e., original points can be viewed as being in a two dimensional array, and the rows are clustered into a specified number of row-clusters and the columns are clustered into a specified number of column clusters. This process repeats itself till a convergence point is reached, beyond which the row and column clustering does not change in successive iterations. It is shown that the information required to assign a row or a column to a cluster is relatively small and this can be represented as a reduction function. This algorithm is parallelized on FREERIDE.

The implementations were experimentally evaluated on a cluster of SMPs, using different datasets. Both the algorithms scale well with increasing number of threads and nodes. For the Parallel Wavelet Transform algorithm, a speedup of 42 is achieved using 8 nodes running 8 threads each. For the Parallel Co-clustering algorithm, a speedup of up to 21 is reported using 32 cores.
BIBLIOGRAPHY


