Analysis of Sensor Data with the PACT Programming Model

Studienarbeit

Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

eingereicht von: Fabian Fier
geboren am: 12.01.84
in: Rheinfelden

Tutor: Dipl.-Inf. Matthias Sax

eingereicht am: 31. Mai 2012
# Contents

1 Introduction 1

2 Basics 3

2.1 Humboldt Wireless Lab 3

2.1.1 High Performance Wireless Sensor Networks 3

2.2 Stratosphere 4

2.2.1 Map-Reduce 5

2.2.2 Shortcomings of Map-Reduce 7

2.2.3 PACT 7

2.2.4 Nephele 10

2.3 Problem Statement 11

3 Seismographic Analysis in ClickWatch 13

4 Porting SeismoBusAnalysis to Stratosphere 17

4.1 Naive Approach 17

4.2 Approach 2: "vertical" data redundancy 18

4.3 Approach 3: "vertical" data redundancy without using timestamps 20

4.4 Approach 4: "horizontal" data redundancy 21

5 Performance 23

6 Conclusions 26
List of Figures

2.1 The ClickWatch experimentation workflow. ........................................... 4
2.2 Compiling and running a program with PACT and Nephele. ................. 5
2.3 Input and output of the Map-Reduce user interface. ............................. 5
2.4 Example for the usage of Map-Reduce. .................................................. 6
2.5 Parallelization Contract. ......................................................................... 8
2.6 The PU split assigned by the Map second-order function. ..................... 8
2.7 The PU split and grouping assigned by the Reduce second-order function. 8
2.8 The PU split and grouping assigned by the Cross second-order function. .... 9
2.9 The PU split and grouping assigned by the CoGroup second-order function. 10
2.10 Match. .................................................................................................. 10
2.11 Example of connection between a PACT program, JobGraph and Execu-
tionGraph. ................................................................................................. 11
3.1 Example plot of SeismoBusAnalysis result. ............................................. 16
3.2 Effects of a passing bus on seismographic sensors located in a building. .... 16
4.1 PACT program of naive approach (not parallelizable). ............................ 18
4.2 PACT program 2. .................................................................................... 19
4.3 Data flow example for Approach 2 and 3. .............................................. 19
4.4 DelimitedInputFormat. ............................................................................ 20
4.5 SerialDelimitedInputFormat. ................................................................. 20
4.6 PACT program 3. .................................................................................... 21
4.7 PACT program 4. .................................................................................... 22
4.8 Data flow example for Approach 4. ....................................................... 22
5.1 Output performance. ................................................................................. 24
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Input example</td>
<td>14</td>
</tr>
<tr>
<td>5.1</td>
<td>Output performance on different amount of calculation nodes and different</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>parallelization degrees</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Wireless communication networks are a highly discussed research topic. Dedicated routing
protocols are developed that enable a set of wireless networking nodes to act as mesh
network: each network node acts as relay for other nodes. In the case of unreachability of
one node, all other network nodes within the network are still able to establish connections
to each other.

Wireless mesh networks can be used for traditional communication purposes. An
application is the provision of urban communication networks: each participant operates
one or more networking nodes in his or her premises. Ideally, the nodes of all participants
are physically close enough to one or more neighbors’ nodes, connect to them and form a
mesh network.

Besides wireless networking nodes that solely consist of radio hardware, there are
special-purpose networking nodes that also feature sensors. These networks form Wire-
less Sensor Networks (WSNs). They can be used for the measurement of environmental
variables. One example for sensors used in wireless nodes are acceleration sensors, also
called seismographic sensors. The data collected by nodes equipped with seismographic
sensors allow for the forecast of earthquakes or Structural Health Monitoring: Changes in
the natural frequencies of bridges or buildings indicate changes of their condition.

Within the Humboldt Wireless Lab (HWL) project, experiments are conducted with
High Performance Wireless Sensor Networks (HP-WSNs). Compared to traditional WSN
nodes, the used networking nodes are powered by wire. This allows for more power
consumption of the hardware and the software: more physical variables can be measured
at once and at higher frequencies than using battery-powered devices. This results in
higher data rates and bigger amounts of data.

In this work, we focus on the pre-existing HWL ”SeismoBusAnalysis” experiment.
Within the experiment, seismographic data is measured by the HP-WSN network nodes.
The measured data is saved and analyzed by the HWL ClickWatch framework in a subse-
quent step. The analyses involve computing intensive tasks like Fourier Transformations.
Together with the huge amount of measured data, the analyses run very long on the given
non-distributed ClickWatch framework. In order to speed up the analyses, we employ the
parallelization framework Stratosphere. It enables the analyses to be run in a distributed fashion on a cluster.

A primary goal of Stratosphere is to support the analyses of huge amounts of data. It is designed for analyses on texts or whole logfiles. In the given "SeismoBusAnalysis" continuous data is analyzed. Each atomic measurement comes with a subsequent timestamp. The analyzing algorithms do not run on the whole set of data, but instead on sliding windows: subsequent, overlapping data subsets of equal size.

In this seminar paper we are interested in answering the following questions by porting the HWL "SeismoBusAnalysis" to Stratosphere: Is Stratosphere suitable for the analyses of continuous data? How well can sliding window semantics be represented in PACT?

In the first chapter, we introduce the HWL project and Stratosphere. We motivate the use of Stratosphere in this context. In the second chapter, we introduce the SeismoBusAnalysis experiment. We discuss different Stratosphere implementation approaches and point out Stratosphere’s strengths and weaknesses regarding the SeismoBusAnalysis use case. Finally, we propose additional sliding window support for Stratosphere operators to overcome the identified weaknesses regarding continuous data processing.
Chapter 2
Basics

2.1 Humboldt Wireless Lab

The Humboldt Wireless Lab (HWL) project aims to build up a large-scale research mesh network (testbed) on the Adlershof campus of Humboldt-Universität zu Berlin [HWL12].

In wireless mesh networks (WMNs) each networking node is able to act as relay for other nodes: "In WMNs, nodes are comprised of mesh routers and mesh clients. Each node operates not only as a host but also as a router, forwarding packets on behalf of other nodes that may not be within direct wireless transmission range of their destinations. A WMN is dynamically self-organized and self-configured, with the nodes in the network automatically establishing and maintaining mesh connectivity among themselves (creating, in effect, an ad hoc network)" [AWW05].

The HWL software framework ClickWatch offers a convenient way to conduct experiments on the testbed. It supports an experimentation workflow similar to agile software engineering: It is assumed that hypotheses can only be created after having conducted experiments. "Research becomes a cyclic execution of experiments, where each cycle consists of four tasks: experimentation, observation, analysis, and adaption" [SZS12a].

The workflow is illustrated in Figure 2.1: Within ClickWatch, an experiment is developed and deployed to the network nodes (1). The network nodes collect data (2) that is statistically analyzed and visualized (3) in ClickWatch. Based on the analyses, the user can decide on restarting the experiment (4). Ideally, the analyses run very fast to allow for a rapid experimentation workflow.

2.1.1 High Performance Wireless Sensor Networks

In addition to the networking hardware, the HWL network nodes are equipped with a variety of sensors. The sensors include an acceleration sensor. It enables the measurement of seismographic data in three dimensions, also referred to as channels. In this seminar paper we follow up with seismographic data. Considering the built-in sensors, the HWL mesh network forms a distributed Wireless Sensor Network (WSN) as defined in [EGPS01].

Traditionally, WSN nodes are battery powered to enable them to be used in field environments that do not provide any infrastructure. The hardware and software is designed
to be very energy efficient. On the software side energy efficiency is achieved by low sensor sample rates, measurements over short periods of time, or the limitation to the measurement of only one single physical variable at a time. Obviously, this puts limitations on what can be measured by WSN nodes.

To overcome the shortcomings of traditional WSN networks, so-called High Performance WSNs (HP-WSNs) are introduced in HWL [SZS12b]. Compared to traditional WSNs the nodes are equipped with larger batteries or require a wired power connection. Being equipped with a fast wireless network connection (802.11n) and more computing power the used HP-WSNs allow for higher sample rates and the measurement of several physical variables at a time. Hence, they produce large amounts of data in a short period of time: Seismographic sensor data of 100 nodes can produce data up to 2.7 MB/s as stated in [SZS12b].

### 2.2 Stratosphere

Stratosphere is a research project on "Information Management in the Cloud". It is a joint project of Humboldt-Universität zu Berlin, the Hasso-Plattner-Institut and the Technische Universität Berlin [Str12]. Its aim is to advance the way large unstructured or semi-structured data is processed in parallel on distributed systems. A new database-inspired approach is developed to run analyses, aggregations, and queries on this kind of data in parallel on cluster and cloud architectures. The research focuses on three parts:
parallel programming models, parallel data processing engines and optimizations of data flow programs.

Within the project, the programming model PACT and the parallel execution engine Nephele are developed and implemented. From now, we refer to "Stratosphere" as the implemented framework rather than the project. Stratosphere is written in Java and available under an Open Source licence. The current official release is version 0.1.1. In this work, the development version 0.2 is discussed.

PACT, that is an abbreviation for parallelization contract, is an extension of the Map-Reduce programming model introduced by Jeffrey and Dean from Google [DG04]. With PACT, it is possible to write parallelizable programs in a standardized and flexible way. Stratosphere compiles PACT programs into directed acyclic graphs (DAGs) and executes them with Nephele (Figure 2.2).

As a motivation for Stratosphere, we first introduce Map-Reduce. PACT and Nephele are described subsequently.

2.2.1 Map-Reduce

The Map-Reduce programming model was introduced by Jeffrey and Dean in 2004 [DG04]. It is an approach to ease the way, parallel programs are written. Formerly, programmers had to implement details of parallelization in every single program that is intended to run in parallel. Map-Reduce separates common parallelization details from the user code. It provides a user interface, into which the programmer adds code.

A Map-Reduce program has one Map step that takes a key-value pair as input and generates one or more intermediate key-value pairs. After the Map step, the pairs are sorted and grouped by their key automatically. The subsequent Reducer takes the sorted groups of key-value pairs as input and generates the final output. The Map and Reduce functions consist of a predefined second-order function each. The user implements custom code within the first-order functions, one for Map and one for Reduce. The signatures and output of the first-order functions are illustrated in Figure 2.3.

\[
\text{map(key, value)} \rightarrow \text{list(key2, value2)} \\
\text{reduce(key2, list(values2))} \rightarrow \text{list(value3)}
\]

Figure 2.3: Input and output of the Map-Reduce user interface.

Figure 2.4 illustrates a simple Map-Reduce program. It processes web server logfiles: The logs are filtered by the occurrence of the word "error". All filtered log lines are grouped...
by the name of the application that threw the error. The corresponding user-implemented first-order functions are shown in Algorithm 1. When the program is executed, the second-order function for Map calls the first-order function for each key-value pair. The key is the name of a logfile (not used in this algorithm) and the value is its text content. For each line containing “error”, an intermediate key-value pair is emitted. This intermediate key-value pair comprises of the application name representing the key and the the line, which is the value. Subsequently, the second-order function of Reduce groups all intermediate key-value pairs by their keys. For every key, it calls the user-implemented first-order function with a multi-set (an iterator in this example) of all values existing for this particular key. The implemented first-order function concatenates all values and emits the result.

Algorithm 1 Map-Reduce: logfile processing example

1: procedure MAP(String key, String value)
2:     for each line in value do
3:         if line contains ”error” then
4:             application = EXTRACTAPPLICATIONNAME(line)
5:             EMITINTERMEDIATE(application, line)
6:         end if
7:     end for
8: end procedure
9: procedure REDUCE(String key, Iterator values)
10:    result = EMPTYSTRING
11:     for each v in values do
12:         result = CONCATENATE(result, v)
13:     end for
14:     EMIT(result)
15: end procedure

Both the Mapper and the Reducer can potentially be executed on subsets of data: The Mapper takes documents as input. Each document forms an independent subset of data that can be executed on different computing instances in parallel. The Reducer takes all
values for one key as input. For each key, one instance of the Reducer can be executed independently.

This concept is called *data parallelization*. The main idea behind data parallelization is that certain data processing tasks only require a part of a given data set for execution. These independent subsets of data are called *parallelization units* (PUs). For Map and Reduce, the maximum number of parallelization units is determined by the used keys.

### 2.2.2 Shortcomings of Map-Reduce

A major drawback of the Map-Reduce programming model is that the Map and Reduce functions “alone are not sufficient to express many data processing tasks both naturally and efficiently” [BEH+10]. Furthermore, Map-Reduce “ties a program to a single fixed execution strategy, which is robust but highly suboptimal for many tasks” [BEH+10]. In the cited paper there are examples shown that are not easily representable in Map-Reduce. One example is a simplification of a TPC-H query that includes a join and a “GROUP BY”-partition. In the paper, a Map-Reduce implementation of the query is presented that includes two subsequent jobs: the first job joins the two given relations, the second one calculates the partitions.

The implementation is rather unnatural, because there are two subsequent Map-Reduce jobs to be implemented for one query. The result of the first job has to be materialized to serve as input for the second job. The execution of the query as a whole can not be optimized: by design, the second job can not reuse pre-existing partitioning of the output of the first job. This leads to intermediate results being partitioned by key two times for the Reducers (“shuffling phase”).

The PACT programming model was introduced as a generalization of Map-Reduce. It overcomes shortcomings of the Map-Reduce programming model. It is described in the following.

### 2.2.3 PACT

As Map-Reduce, Stratosphere employs the concept of data parallelism to achieve parallelization [BEH+10]. In addition to a key, *Input Contracts* are used in Stratosphere to describe how to generate PUs. From a user’s perspective, Input Contracts are pre-existing second-order functions like Map and Reduce. The user implements the corresponding first-order functions as ”black boxes” for the underlying Stratosphere system. Both together, the first- and the second-order functions, form a PACT (Figure 2.5). Since Input Contracts are a main feature of PACTs, we also refer to them as PACTs.

In Stratosphere, PACTs can be interconnected to form jobs. This offers more flexibility compared to Map-Reduce: Given the TPC-H example of [BEH+10], the query can be expressed in a single Stratosphere job instead of two Map-Reduce jobs [BEH+10].

Another advancement compared to Map-Reduce is the data model: PACTs operate on multi-sets (*bags*) of *Records*. Records are tuples that comprise of an arbitrary amount of *Attributes*. Records are more flexible than the key-value data model: *Keys* can be defined as parameters of a PACT. Previous PACTs that generate the input Records are agnostic
to the Keys used by subsequent PACTs. Furthermore, Keys can comprise of an arbitrary subset of Attributes. Together with the Input Contract, the Key defines PUs that are processed independently by instances of the user-implemented first-order function of the PACT.

In the following, we describe the existing Input Contracts. As in Map-Reduce, PACT also comprises of a Map and a Reduce Input Contract that operates on a single input bag. In addition, there are Input Contracts that operate on multiple input bags. They are called Match, CoGroup, and Cross. All PACTs have in common that they can emit an arbitrary amount of output Records.

**Single-Input Contracts**

Single-Input Contracts take a single bag of Records as input.

*Map:* Each input Record forms a separate PU that can be executed independently. In Figure 2.6 the independent PUs are illustrated by dotted squares, containing one Record each. From now, all dotted squares in operator figures refer to PUs. Map does not employ a Key function. This Input Contract can be used for filtering for example.
Reduce: Input Records for Reduce are grouped and sorted by their Key. In Figure 2.7, the colored left part of each square represents the Key of the Record. All Records with the same key form one PU, as depicted on the right. A Reducer can be used for partitioning: an SQL "GROUP BY" can be implemented by using a Reducer.

The Reducer can be enhanced with a Combiner. The Combiner is called by the Input Contract before the Reducer and accepts arbitrary "subbags" of the PU as input. It performs pre-calculations for the subsequently executed Reducer. A Combiner can be used for associative functions like add: When adding a set of values, any subset of values can be summed up without changing the result. For functions like the arithmetic mean, Combiners cannot be used. Calculating a mean over means of subsets leads to a different result than calculating a mean over a complete set of data.

Multiple-Input Contracts

Multiple-Input Contracts operate on two or more bags of input Records. In the following figures we employ only two bags of input Records for simplicity.

Figure 2.8: The PU split and grouping assigned by the Cross second-order function (adopted from Str12).

Cross forms the cartesian product over its input bags (Figure 2.8). In the case of two inputs, each Record pair over both inputs forms an independent PU. Cross does not employ Key functions. An application of Cross is an SQL cross join.

CoGroup partitions its input bags according to their Keys. Partitions of all input bags with the same key form one PU as depicted in Figure 2.9. Note that if a Key appears in only one input bag, the corresponding partition also forms one PU as indicated by the blue Key in the Figure. With the help of CoGroup, a full outer join can be implemented by combining all Records of different input bags.

Match In contrast to CoGroup, in Match (Figure 2.10) only combinations of Records sharing the same Key and originating from different input bags form one PU. Furthermore,
Match does not consolidate all Records sharing the same Key into one PU like it is the case for CoGroup. There has to be a matching Key in all bags of input: Otherwise a Record will be dropped like the blue one in the Figure. With Match, an equi-join can be realized.

PACT serves as the main user interface for Stratosphere. The user can assemble PACTs to Stratosphere programs. In order to execute PACT programs, Stratosphere uses Nephele. In the following we describe Nephele and how PACTs and Nephele are related.

### 2.2.4 Nephele

Nephele is the parallel execution engine of Stratosphere. It is agnostic about PACT: the user interface of Nephele are JobGraphs, directed acyclic graphs (DAGs). In order to execute PACT programs, Stratosphere compiles PACT programs into JobGraphs.
2.3. PROBLEM STATEMENT

In Figure 2.11 an example PACT program is depicted. It is compiled to a JobGraph consisting of vertices V1 to V4 and edges connecting them. The vertices represent Nephele-wrapped PACT programs: a PACT program consists of a second-order function like Map or Match and a first-order user function. The Nephele code handles the physical communication over channels defined by the edges. Channels can be for example in-memory or network channels. Given a JobGraph and informations on available hardware resources, Nephele spans the JobGraph to an ExecutionGraph. It distributes the program code and data among the computing nodes. The number of computing nodes can be either of fixed size (cluster) or of variable size (cloud). In the figure, two computing nodes are available. For the measurements we introduce at the end of Chapter 2, we use a cluster.

Figure 2.11: Example of connection between a PACT program, JobGraph and ExecutionGraph (adopted from [BEH +10]).

2.3 Problem Statement

In the following, the ”SeismoBusAnalysis” experiment of HWL is described. Within the experiment, huge amounts of data arise. An analysis involves complex computations of sensor-time signals. Using a non-distributed implementation for the analysis is not feasible. The computations run for hours or even days depending on the amount of measured variables, nodes, time, and the measured frequency. One way to accelerate these analyses is parallelization. The computations can be divided by nodes, dimension and/or by time: The data of one dimension of one node can be processed independently from other dimensions of the same node and data from other nodes. Within the data of one dimension of one node, data blocks of a certain time frame (sliding window) can be processed independently from each other. For example, the block of data ranging from time 0 to N can be processed in parallel to the data ranging from time 1 to N+1, from 2 to N+2 etc.

In this seminar paper we utilize Stratosphere to parallelize the analysis of the ”SeismoBusAnalysis” experiment. We intend to determine the suitability of Stratosphere’s
PACT programming model and data parallelization approach concerning the given use case. We expect the computations to scale with the number of the used computation instances. Besides, we expect the time to rise linearly with the amount of seismographic data to be processed.

The algorithm for the analysis is used as-is for the research on the suitability of the problem to Stratosphere. The evaluation of the analysis results is out of the scope of this work.
Chapter 3

Seismographic Analysis in ClickWatch

In the following, we describe the pre-existing "SeismoBusAnalysis" experiment of HWL. Within the experiment, it is investigated whether collected seismographic data is suitable for Structural Health Monitoring (SHM). The intention of SHM is to measure natural frequencies of buildings or other civil structures to draw conclusions on their quality properties. "Changes in these frequencies indicate changes in structural health." [SZS12a] As a first step for this evaluation, passing buses are to be recognized by data analyses. This is just an introductory experiment to develop subsequent SHM analyses.

Following the general HWL experimentation workflow (Figure 2.1), seismographic data is measured by HWL network nodes. Afterwards, mathematical calculations are computed in order to highlight the occurrence of relevant frequencies and to gain a visual impression of the empirical distribution of the data. Both the measurement and the analysis are done within the ClickWatch framework.

The data is saved to HBase, a distributed scalable big data store ([HBa12]). It is used to obtain a higher write speed compared to relational databases. Distributed capabilities are not used yet.

Each input entity consists of node, channel, time and the measured value (example in Table 3.1).

In the following, we describe the pre-existing algorithm of the experiment. We start with a high-level overview on the algorithm and its most important parameters. In the remainder, the algorithm is defined by pseudocode and explained on a line level.

In a first step, the values are normalized by the subtraction of a moving average. The moving average is calculated as the mean over one window: a window includes the current value and preceding values. The total number of values in one window is determined by the parameter removeOffsetWindowSize. Afterwards, a Fourier Transformation is executed over a predefined window of normalized values. This window is set by the fftWindowSize parameter. Finally, the resulting array is divided into chunks of equal size. The number of chunks can be influenced with the parameter numberOfBins. This chunking procedure is called binning. The chunk of one pre-defined number is chosen (chosenBin). It refers
to a range of frequencies. The mean of this chunk divided by a standardization constant is returned. The constant is $2^{21}/1000$ and ensures that the values are returned in “milli g”: g is a unit for gravitation force.

The SeismoBusAnalysis algorithm is specified in Algorithm 2. The given input is ordered by time. To simplify the code, time is assumed to be represented by an increasing integer number starting at 0 and incremented by 1 for each subsequent measurement.

- In line 1 to 8 parameters for the algorithm are defined: `sampleRateInHz` specifies, how many measurements are done per second. The `fftWindowSizeInSeconds` is the window size of the Fast Fourier Transformation (FFT) in seconds. `fftWindowSizeOrig` holds the number of measurements contained in the window. The window size of the FFT is saved in `fftWindowSize`. `numberOfBins` is the number of chunks the FFT result is divided into. Of these chunks, the `chosenBin`th one is chosen (beginning at 0).

- From line 10 to 22, the input is normalized. The sum of measured values is calculated up to `removeOffsetWindowSize` values. The sum is divided by `removeOffsetWindowSize` and subtracted from the current value. The result is written to `normalizedValue`.

- In line 23, the FFT is calculated over the last `fftWindowSize` `normalizedValues`.

- In line 28, the result length is calculated. The FFT result is symmetric. In order to obtain the result only once, the first half of the FFT array is used.

- From line 29 to 34, the FFT result is binned: The FFT result is split into `numberOfBins` chunks. The `chosenBin`th chunk is chosen and all its values are saved into `fftResultArray`.

- In line 35 to 40, the mean is calculated over all values in the `chosenBin`. The mean is divided by a pre-defined parameter and the result is emitted.
Algorithm 2 SeismoBusAnalysis Algorithm

1: `sampleRateInHz ← 100`
2: `fftWindowInSeconds ← 1`
3: `fftWindowSizeOrig ← fftWindowInSeconds * sampleRateInHz`
4: `fftWindowSize ← Integer.highestOneBit(fftWindowSizeOrig) * 2`
5: `removeOffsetWindowInSec ← 10`
6: `removeOffsetWindowSize ← removeOffsetWindowInSec * sampleRateInHz`
7: `numberOfBins ← 10`
8: `chosenBin ← 2`
9: `while line ← readInput() do`
10:    `node ← extractNodeFrom(line)`
11:    `dimension ← extractDimensionFrom(line)`
12:    `time ← extractTimeFrom(line)`
13:    `value[node][dimension][time] ← extractValueFrom(line)`
14:    `if sum[node][dimension] = undefined then`
15:       `sum[node][dimension] ← 0`
16:    `end if`
17:    `sum[node][dimension] ← sum[node][dimension] + value[node][dimension][time]`
18:    `movingAverage ← sum[node][dimension]/removeOffsetWindowSize`
19:    `normalizedValue[node][dimension][time] = value[node][dimension][time] − movingAverage`
20:    `complexFFTResultArray ← FourierTransformation(n)
21:       normalizedValue[node][dimension][time − fftWindowSize],
22:       normalizedValue[node][dimension][time − fftWindowSize + 1],
23:       ...,
24:       normalizedValue[node][dimension][time])`
25:    `resultLength ← (complexFFTResultArray.length/2) + 1`
26:    `startIndex ← (resultLength/numberOfBins) * chosenBin`
27:    `endIndex ← startIndex + (resultLength/numberOfBins)`
28:    `for i = startIndex → endIndex − 1 do`
29:       `fftResultArray[node][dimension][time][i] ← complexFFTResultArray[i].abs()`
30:       `fftWindowSizeOrig`
31:    `end for`
32:    `fftSum = 0`
33:    `for i = 0 → fftResultArray[node][dimension][time].length do`
34:       `fftSum ← fftSum + fftResultArray[node][dimension][time][i]`
35:    `end for`
36:    `fft[node][dimension][time] ← (fftSum/binnedFft[chosenBin].length)/(2^{21}/1000)`
37:    `EMIT(fft[node][dimension][time])`
38: `end while`
CHAPTER 3. SEISMOGRAPHIC ANALYSIS IN CLICKWATCH

Figure 3.1: Example plot of SeismoBusAnalysis result.

Figure 3.1 illustrates the result of the calculation for the measured data of one node. The measured time span is approximately 67 seconds and the input consists of roughly 2 million entities. Each color represents one of the three channels. The peaks are seismographic events.

Figure 3.2: Effects of a passing bus on seismographic sensors located in a building (from HWL).

Figure 3.2 shows how a passing bus can be recognized by seismographic sensors distributed in a building: the peaks are increasingly timely delayed the further away the corresponding node is placed from the street.
Chapter 4

Porting SeismoBusAnalysis to Stratosphere

In the following we describe approaches, how the HWL analysis can be realized in Stratosphere. First, a non-parallelizable naive approach is used to illustrate important features of Stratosphere. In the subsequent approaches, we employ ”vertical” and ”horizontal” data redundancy to achieve parallelizability.

4.1 Naive Approach

Stratosphere has certain properties which have to be regarded when implementing a PACT program. In order to highlight these properties, we employ a fictional ”naive” implementation of the SeismoBusAnalysis algorithm without regarding parallelization-specifics of Stratosphere.

Every atomic action of the algorithm is ported to one PACT (Figure 4.1): The calculation of the Moving Average and the Fast Fourier Transformation is performed by Reduce PACTs. The Normalization is represented by a Map PACT. Additionally, there is a Tokenizer (Map PACT) that splits the input and creates Records.

One single file would serve as input. It would contain data of several nodes and channels and be sorted by time in ascending order. We assumed that Stratosphere reads the input linearly, preserving time order. The used Key is the same for all PACTs and consists of node, dimension and time.

The Moving Average PACT and the FFT PACT would have to keep internal static value lists of the size of a window. Records would be required to keep their order when being passed from one PACT to a subsequent one.

Neither the first, nor the second prerequisite is met: The order of input entities and Records is not preserved within Stratosphere. A workaround could be to set the parallelization degree of all PACTs to 1, which would obviously a bottleneck and not provide better performance than a non-parallelized implementation. The stateful PACTs would rely on proper sorting of the input and intermediate Records. Stateful PACTs are not
desired by design \cite{Str12}. They could cause incorrect and nondeterministic results. All subsequent approaches take the described Stratosphere properties into account.

**Figure 4.1: PACT program of naive approach (not parallelizable).**

### 4.2 Approach 2: "vertical" data redundancy

In this approach (Figure 4.2), we apply "vertical" data redundancy in order to obtain parallelization: Each input entity is copied by the Tokenizer into a number of Records that is equal to the window size of the subsequent Normalization step ($removeOffsetWindowSize$).

The Key for the Normalizer consists of the copied timestamp, node, and dimension. Figure 4.3 illustrates this data redundancy for a normalization window size of 3. For simplicity, the Key is represented by an integer, followed by a value represented by a character. The tokenizer copies each input entity into three Records and applies the preceding 3 integers that are used as Keys. Additionally, it keeps the original Key as marked with parentheses.

For the Normalization, Records with the same Key are grouped automatically by Stratosphere. The Normalizer calculates the average of the values and subtracts it from the value with the newest original Key. Again the result value is copied into several Records in order to obtain a window of the size $fftWindowSize$ for the next calculation step. In the Figure, we used a small window size of 2 for demonstration.

The copying of records is performed independently on each entity without knowledge of the previous timestamps. A prerequisite of this approach is the equidistance of all timestamps. For real-world data this requirement is not met. In the example data provided by HWL, the time between two measurements varies between 9.770 ms and 11.020 ms. Obviously, not all time values can be normalized to 10 ms by rounding, because the spread is too big. This causes the number of grouped Records to vary in the Normalizer. The calculation delivers incorrect results.
4.2. APPROACH 2: "VERTICAL" DATA REDUNDANCY

Figure 4.2: PACT program 2.

Figure 4.3: Data flow example for Approach 2 and 3.
4.3 Approach 3: "vertical" data redundancy without using timestamps

For the third approach, we eliminate the problem of non-equidistant timestamps by replacing them with IDs. Originally, there are no IDs present in the data. In order to be able to calculate IDs, we change the way, Stratosphere processes input files by a SerialDelimiterInputFormatter. It prevents Stratosphere from partitioning input files. With the built-in DelimitedInputFormatter the content of input files is splitted arbitrarily between computing instances (Figure 4.4). This default behavior is suited well for applications that have completely independent input data within files. This strategy optimizes calculation node usage. For this particular application however, a partition of input data is undesired, because the data within one file is not independent: each atomic entity should be assigned a subsequent ID.

![Figure 4.4: DelimitedInputFormat.](image)

In order to apply an ID to each input entity that is unique together with the node and the dimension, we modify the data source to not partition input files (Figure 4.5). The input source holds a static variable that counts up the ID. It applies the IDs to the copied Records in the same way like the timestamps are applied in Approach 2.

Each input file exclusively contains the data of one channel of one node and is sorted by time ascending. All files can be processed in parallel. The maximum parallelization degree of reading the input is limited to the number of files. This is not a practical constraint since experiments include lots of rather small input files.

Every computing instance that processes the data source can be stateful and applies an auto-incremented ID to each input entity. The resulting PACT program 4.3 is equal to the one from Approach 2, but with a different data source function. Keys contain a line ID orig(inal) and a line ID copy instead of the timestamps.

The approach only works for extremely small sets of input data. Stratosphere by design assures that all Records of one key are present when handed to a Reducer. Stratosphere is agnostic to continuous data and sliding window semantics. It needs to collect and temporarily save all values until the complete input is read. Given the small set of 60 MB of input data, a Normalization window size of 1000 and disregarding object serialization overhead, the intermediate result is at least 60 MB * 1000 = 60 TB. Obviously, this can
4.4 Approach 4: "horizontal" data redundancy

For Approach 4, we moved the window necessary for the Normalizer and the Fourier Transformation into single Records. It makes use of the SerialDelimitedInputFormatter introduced in Approach 3. Again, the input files are split into nodes and channels and sorted by time ascending. We eliminated all Reducers in order to achieve pipelining without the necessity of saving potentially huge intermediate results.

As illustrated in Figure 4.8, the LineInFormatter keeps static internal lists of the last values up to the size of the needed overall window. It returns Records with value lists. The overall window size for the input formatter is equal to the size of the Normalization window removeOffsetWindowSize plus the size of the FFT window fftWindowSize minus one.

This approach appears to be a good solution to the problem, since it eliminates all previously stated problems. We reached pipelining without the need to read the whole input and save huge intermediate results. We eliminated the problem with non-equidistant timestamps.

Still it restricts the ability to further parallelize the input formatter. It uses data-redundant Records which causes overhead. The Normalization and Fourier Transformation...
is re-calculated on the same values many times. The serial data source is stateful which is not intended by design.

![Diagram](image)

Figure 4.7: PACT program 4.

![Table](image)

Figure 4.8: Data flow example for Approach 4.
Chapter 5

Performance

Regarding Approach 4, we are interested in whether the execution time for the program scales linearly with the size of input data (1) and the number of calculation instances (2).

We use a computation cluster for the performance measurements. The computation instances consist of different hardware configurations. They are equipped with 2 and 4 Opteron 280, 880 and 6168 processors. Each processor consists of at least 2 cores. The main memory varies between 4 and 32 GB. The hard disk space on each instance varies between 160 and 500 GB. All instances share one network file system and are interconnected by Ethernet and InfiniBand. The instances can not be reserved exclusively. All measurements are subject to interference by other running tasks and different hardware configurations.

For all measurements we disregarded the described hardware heterogeneity. We only used a part of the available resources: 500 MB of main memory and up to 2 parallel Nephele tasks on one physical instance. Limiting the resources for calculations, we assumed to reduce the influence of different hardware configurations and interference by other running tasks.

For input data we use 34 files that contain the measurement of one dimension of one HWL node each. The total size of all files is approximately 58 GB. The measured time span varies file by file. Thus, the file sizes vary. The reason for the variations is of technical nature: Some HWL nodes were interrupted during recording.

While conducting the performance measurements, we observed that the data output rate (GB/minute) stays nearly constant over time. We recognize that the amount of output data the program produces is solely dependent on the amount of input data and a constant factor. It is particularly not dependent on time or data semantics. Regarding this constant relationship between the amount of input and output data together with the measured constant data output per time, we conclude that the program scales linearly with the size of input data (1).

To determine the scalability regarding calculation instances, we measured the output per time on 2, 4, 8, and 16 calculation instances. Each measurement we conducted with a parallelization degree equal to the number of calculation instances.
Table 5.1: Output performance on different amount of calculation nodes and different parallelization degrees.

The results are shown in Table 5.1 and plotted in Figure 5.1. Considering the different hardware configurations of the given cluster, the output performance scales nearly linearly with the number of calculation instances (2). To verify whether the relationship is linear, a dedicated cluster with homogeneous hardware would be necessary.

Figure 5.1: Output performance.

We performed the entire calculation on 34 files, 17 calculation instances and a parallelization degree of 34. It took 87 minutes. We did not compare this result to the calculation time needed by the ClickWatch framework on the same data: Stratosphere reads the data from files, while ClickWatch reads it from a non-SQL database system. The used hardware is different than the cluster hardware used for the Stratosphere cal-
culation. Because of these fundamental differences, a comparison would not have been meaningful. Nevertheless, we have shown that the calculation time scales with the number of used computing nodes. By design, the ClickWatch framework does not scale. Thus, we assume that from a certain amount of input data, Stratosphere has an advantage regarding calculation time compared to using ClickWatch standalone.

In the following, we derive conclusions from the performance measurements and the suitability of Stratosphere to the given problem in general.
Chapter 6

Conclusions

We introduced the HWL project and Stratosphere, ported the use case "SeismoBusAnalysis" from HWL to Stratosphere and discussed several implementation approaches.

Stratosphere in its current development version is ready to run the calculations for the experiment. The calculation time scales linearly with the amount of input data. It also scales linearly with the amount of computation instances employed.

However, the automatic partitioning of input files restricts the suitability of Stratosphere to data that is independent within each file. For ID generation, input files must be read subsequently and as a whole: the data is not independent. To allow for interdependent data as input, the described SerialDelimitedInputFormatter is shown. It offers the possibility to disable the default input partitioning. It does not break parallelizability, but it puts limits on it. Using the SerialDelimitedInputFormatter, the maximum parallelization degree of the input is determined by the number of input files (or other chunks of data of which each chunk has to be processed subsequently as a whole). In the presented use case, this is not problematic because of the high number of input files.

Furthermore, the Reduce design in combination with the standard Stratosphere sorting strategy limits the maximum amount of input data for a Reducer: Stratosphere guarantees that all Records with the same Key are processed in one step. To achieve this, the whole intermediate result needs to be stored in RAM or in temporary hard disk files for sorting and grouping. This reduces performance or makes the calculation impossible if the given memory resources are not sufficient. For the presented approaches using data redundancy excessively, this is problematic.

For data that needs to be processed subsequently with predefined window sizes, this concrete problem could be overcome by a non-blocking Sliding Window Reducer that takes the window size as argument: It processes bags of Records with the same Key once their number reaches the window size. The user has to assure that no more Records with the same Key will follow. This is an advancement compared to the existing "NONE" sorting strategy of Stratosphere since it is optimized for weakly sorted input as occurring in Approach 3. Still "vertical" data redundancy would be required to use.

A more advanced sliding window extension in Stratosphere would be the support of overlapping windows without data redundancy. For this to work, the data flow between
PACT instances in the Nephele Execution Graph has to be modified. This would enable the use of stateful PACTs that are allowed to re-use previously seen input Records or previously calculated values by design. It enables pipelining in Approach 3 without the need to store huge intermediate results.

Besides Reduce, other Stratosphere operators could also benefit from sliding window support. When using CoGroup, Cross, and Match in Stratosphere programs using sliding windows on continuous data, similar restrictions like the mentioned ones for Reduce would arise. Comprehensive sliding window support would open the possibility for pipelined continuous data processing in Stratosphere. It would push Stratosphere towards an even more general purpose parallel data processing framework.
Bibliography


Statement of authorship

I declare that I completed this thesis on my own and that information which has been
directly or indirectly taken from other sources has been noted as such. Neither this nor a
similar work has been presented to an examination committee.

Berlin, May 9, 2012

........................