Bachelorarbeit

Chandrakant Swaneet Kumar Sahoo

Nested Data Parallelism for Image Processing Algorithms

Optimisations in Functional Programming Languages

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Chandrakant Swaneet Kumar Sahoo

Nested Data Parallelism for Image Processing Algorithms

Optimisations in Functional Programming Languages

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Michael Köhler-Bußmeier
Zweitgutachter: Prof. Dr.-Ing. Andreas Meisel

Eingereicht am: 23. Juli 2015

Chandrakant Swaneet Kumar Sahoo

Thema der Arbeit

Nested Data Parallelism for Image Processing Algorithms - Optimisations in Functional Programming Languages

Stichworte

Nested Data Parallelism, Haskell, Bewertung, Vectorization, Fusion, Histogrammausgleich, Funktionale Programmierung, Parallele Programmierung, Bildverarbeitung

Kurzzusammenfassung

Nested Data Parallelism ermöglicht den prägnanten und treffenden Ausdruck irregulär-paralleler Programme und erreicht trotzdessen eine Performance vergleichbar zu Flat Data Parallelism. Dies wird durch eine Programmtransformation ('Vectorization') erreicht. Verschachtelte Funktionen und Datenstrukturen werden dabei auf flache Funktionen und Datenstrukturen reduziert. Diese Arbeit verlgeicht und bewertet die Effektivität von Nested Data Parallelism und manueller Parallelisierung. Es werden vier Implementierungen des Histogramausgleichs erstellt - eine Sequentielle, eine Manuell-parallelisierte, Eine die Nested-Data-Parallelism verwendet und die davon Vektorisierte. Diese werden bezüglich Arbeitsaufwand, Ähnlichkeit zum Algorithmus, Komplexitätklassen und mehr bewertet. Dies gechieht in Haskell als Example für funktionale Programmiersprachen. Die Arbeit zieht den Schluss, dass Nested Data Parallelism effektiv im allgemeinen Verlgeich ist. Jede Implementierung hat Stärken und Schwächen. Es gibt allerdings, keine beste Implementierung.

Chandrakant Swaneet Kumar Sahoo

Title of the paper

Nested Data Parallelism for Image Processing Algorithms

Keywords

Nested Data Parallelism, Haskell, Evaluation, Vectorization, Fusion, Histogram Balancing, Functional Programming, Parallel Programming, Image Processing

Abstract

Nested Data Parallelism enables the concise expression of irregularly parallel programs while still being nearly as performing as flat data parallel programs. This is achived by a program transformation ('Vectorizaiton') . It flattens complex nesting of data structures and functions. This thesis is a comparison and evaluation of Nested Data Parallelism and manual parallelism of

irregular image processing algorithms such as Histogram Balancing in functional programming languages by the example of Haskell. Four implementations for Histogram Balancing are created and evaluated on constant factors, running time complexity, human workload and more. They are: sequential, manually-parallelized, nested data parallel and finally a vectorized thereof. The thesis comes to the conclusion, that Nested Data Parallelism compares well to the other approaches. Though, every implementation has its advantages and drawbacks - none is the best.

Contents

List of Tables

List of Figures

1 Introduction

The exponentially increasing number of processors and machines available in current hardware are a great opportunity to speed-up programs. This is desired in image processing and is frequently expressed with Data Parallelism.

Data-parallel programs express parallelism via bulk-primitives over parallel data structures - e.g. `map/reduce`. They perform well on flat data and with regular non-recursive program flow. However, most applications in need of parallelism have irregular behaviour. Their algorithms work with nested arrays (e.g. images or matrices), graphs, trees or complex recursion (e.g. clustering or machine learning). Direct parallel implementation of such algorithms in flat data parallelism results in a performance penalty. Programming parallel algorithms then mostly becomes manual work and utterly complex.

In contrast, Nested Data Parallelism Blelloch (1996) lifts these limits and enables the concise expression of irregularly-parallel programs - while still compiling to efficient machine code. It uses a non-trivial program transformation called 'Vectorization' (or 'Flattening') where the original program is transformed into an equivalent flat data-parallel program.

Due to properties like Immutability and Referential Transparency, functional programming languages like Haskell are fruitful grounds for sophisticated program transformations and high-level expressiveness. They are key building blocks for effective Nested Data Parallelism. Programming in Haskell is guided by important principles - conciseness of expression and "let the compiler do the work for you". Nested Data Parallelism in Haskell is an excellent embodiment of these principles.

All in all, this thesis aims to give an evaluation on Nested Data Parallelism and manual parallelization in the functional programming language Haskell for application in image processing.

1.1 Aim and Methodology

The question on how Nested Data Parallelism (NDP) compares to conventional parallelism is tackled by the following methodology: An image processing algorithm with irregular behaviour

is implemented in four variations. These variations are then compared on performance, human workload and other aspects. Finally a conclusion is drawn.

A_C := Histogram Balancing, a conceptual image processing algorithm
P_{seq} := A direct sequential implementation of A_C
P_{man} := A manually-parallelized implementation of A_C
P_{nest} := A nested data-parallel implementation of A_C
P_{vect} := The vectorized implementation of A_C from P_{nest}

The sequential implementation P_{seq} serves as a control-implementation for the parallel programs. The manual implementation P_{man} will be compared against the nested data parallel ones. It is also distinguished between the original program P_{nest} and its vectorization P_{vect}. This distinctions enables to pin-point the advantages created by Vectorization.

Given the generalisations of NDP into Haskell Jones (2008), Haskell is an optimal choice for the implementation. The results of this thesis however can be easily adopted to related functional programming languages.

Contrary to any expectations, the thesis does not present directly executable programs. NDP is Haskell is still in development and the program transformation is not yet completely implemented. The vectorization - usually done by the compiler - is done manually here. This leads to a theoretical analysis of all programs instead of benchmarking and statistical analysis.

Histogram Balancing was chosen for A_C because of its simplicity and limited-irregularity. It suits for three implementations and manual vectorization.

1.2 Structure

This thesis begins with the basics in chapter 2. It presents Haskell and Nested Data Parallelism with Vectorization. Parallel complexity measures and Histogram Balancing are introduced either. The subsequent four chapters present and analyse four implementations. First, the sequential P_{seq} is presented (chapter 3). Second, comes the manually parallelized P_{man} (chapter 4). Third, the nested data parallel P_{nest} is given (chapter 5). Fourth, its vectorization P_{nest} is examined (chapter 6). The thesis then compares and evaluates the programs in chapter 7. Finally, chapter 8 summarizes the thesis, references to related work and gives an outlook.

2 Basics

"Auf den Schultern von Riesen"

Johannes von Salisbury

This chapter will give an introduction into the basics needed to understand this thesis. First, an introduction into functional programming in Haskell is given - it is to ease the understanding of the code involved. Then Nested Data Parallelism (NDP) is covered where key insights are made and concepts that exploit them are presented. Afterwards an introduction into parallel complexity measures is given. The concept of work and depth complexities are presented - together with definitions to calculate them. Finally, Histogram Balancing is presented. Its problem is described and the algorithm is formulated.

2.1 Haskell

Haskell is a general-purpose strongly-typed purely-functional programming language. It has full type-inference, referential transparency, higher-order-functions and more features. Functional programming in Haskell and similar languages is guided by two ideas - "write what you mean" and "let the compiler do the work for you". The former means the use of high abstractions to write concise code - while the latter means the use of sophisticated compilers [1] to optimise the abstractions to efficient machine code.

A few of the main features needed in this thesis are presented now:

Syntax Consider this function definition:

```
1 foo :: Double -> Int
2 foo x = round (max 5 x)
```

`foo` takes the maximum of 5 and the argument number `x` and rounds it to the nearest integer. The function is given a type signature in the first line. `f :: t` denotes that `f` is of type `t`. The type of `foo` describes a function from a double floating number to an integer. Multiple argument

[1] some which - like the Glasgow Haskell Compiler - are written in Haskell themselves

functions are typed with multiple arrows - e.g. `max :: Double -> Double -> Double`. In contrast to many programming languages parentheses `()` are not used for function application (as in `f(a)`) but instead are used to specify precedence/ordering. Function application is expressed with spaces - as in ' '. E.g. `round 5.2` applies the built-in floating point rounding function to return the integer `5`. Multiple-argument functions are be applied with multiple spaces. E.g. `max 5 x` applies `max` on `5` and `x`.

The function can be written alternatively as:

```
foo :: Double -> Int
foo x = let a = max 5 x
            b = round a
        in  b
```

`let`-statements bind expressions to variables. The expression after the `in`-keyword is the return value of the entire expression.

The function can be written without the explicit use of its argument `x`.

```
foo = (round) . (max 5)
foo = round . max 5
```

This is called Currying and function composition. One can omit the last argument of a function and define the function `foo` by a series of compositions. Function composition is denoted by a dot (`.`). In this case `foo` is the composition of `x ↦ max 5 x` and `x ↦ round x`. Both lines are equivalent and show, that function application ' ' binds stronger than composition (`.`).

In functional programming one often ends up nesting multiple functions:

```
myFunc x = (bar a b (baz c (foo d x)))
```

One can write the same function in various different ways - depending on space and visual ease.

```
myFunc x = (bar a b . baz c . foo d) $ x
myFunc x = bar a b . baz c . foo d $ x -- single-line, explicit argument
myFunc = bar a b . baz c . foo d -- single-line, implicit argument
myFunc x = bar a b           -- multi-line, explicit argument
           . baz c
           . foo d
           $ x
myFunc = bar a b             -- multi-line, implicit argument
         . baz c
         . foo d
```

The `$` operator denotes function application either. However, it has very low precedence. Such that the functions are composed first before the argument is applied into the composition (line

1). The reader is noted, that the multi-line definitions apply the functions from bottom to up. First foo, then baz and finally bar is applied - even though foo is written last.

Naming Conventions Higher order functions are denoted with `f`,`g` or `h`. Variables are denoted with `x`,`y`,`z`,`a`,`b`,`c` or with longer names like `divisor` or `multWith2`. Two closely related variables are given primes. E.g. they are named `x` and `x'`. A prime is a part of the variable name. Finally, a collection of `x` will be referred to by `xs`. E.g. `xs` is an array, `xss` is a two dimensional array, etc.

Strong typing and Type Inference Every expression has a type. A program containing ill-typed expressions, e.g. `3 + False`, is rejected by the compiler. Types can be (almost always) be inferred from the context and then don't need to be specified.

Type Synonyms Type synonyms enable one to refer to a complicated type by a simpler name. E.g. `type Foo a = (Int,a)` defines a type synonym - such that `Foo Double` is replaced by `(Int,Double)` during compilation.

Referential Transparency and Immutability Functions and expressions are referentially transparent in Haskell. Like their inspiring equivalent in Mathematics, they always return the same result when evaluated (with the same arguments). A consequence is Immutability - variables and data structures cannot be mutated. Functions over data structures return new copies instead of mutating the argument. This enables powerful optimization techniques such as Inlining and Rewrite Rules. They are explored in section 2.2.

Record Syntax Records combine multiple values into a single complex value. The following snippet creates a record of type `Person` with the fields `name` and `age`.

```
1 Person {
2   name = "Mark"
3   age = 23
4 }
```

Polymorphism Types can be polymorphic. E.g. one can write the following general function without nesessarily fixing to concrete types.

```
1 foo :: a -> b -> a
2 foo x y = x
```

Lower case letters in the type signature denote type variables. This function is well-defined for both arguments as long as the return type is same as the type of `x`. This restriction is reflected in the type signature - the first and the last type variables are equal.

Similarly, one can define functions over polymorphic data types. E.g. `List a` denotes a list of elements of type a. `List Int` is by that definition a list of integers.

Polymorphisim is for example useful in `map :: (a -> b) -> List a -> List b`.

Higher Order Functions Functions are values and can be taken or passed as arguments to other functions. `map` is an example.

```
map :: (a -> b) -> List a -> List b
```

Given a function from `a` to `b` and a List of `a`-typed values, `map` applies the function on every element and produces a List of `b`-typed values.

Lambdas Functions can be written literally using Lambda expressions. For example, the function `f x = 2*(x+1)` can be written directly as `f = λx -> 2*(x+1)` or `f = \x -> 2*(x+1)`. The backslash \ is used inside code snippets - λ is used outside of them. [2]

Identity, Flip and Function Composition Three commonly used functions are:

```
id :: a -> a
id x = x
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
(.) :: (b -> c) -> (a -> b) -> a -> c
g . f = \x -> g (f x)
```

`id` is the identity function and `flip` exchanges a functions first two arguments. The identity function is the neutral element of function composition `(.)` and the following law holds: `∀f: f . id = id . f = f`. This is useful for Rewrite Rules.

A brief overview has been given. An introduction to Nested Data Parallelism is given next.

2.2 Nested Data Parallelism

In the ground breaking work Blelloch (1996) major contributions to parallel programming in functional programming languages were made. The paper presented his earlier work Blelloch

[2] The \ is used because of its similarity to λ.

et al. (1993) on NESL - a programming language specifically designed for expressing parallelism in functional programming languages. Its ideas and insights were adapted to various languages - one of them being Haskell. Multitude years of research generalized the results of the special purpose language NESL to the widely used general purpose language Haskell. [3] This project is called Nested Data Parallel Haskell and this section will give an overview of its key concepts.

In Flat Data Parallelism, the programmer is provided with parallel mapping primitives to express parallelism. Such a function might be `map` such that `map f xs` applies `f` on each of the elements in the array `xs` in parallel.

This function has a fairly intuitive parallel implementation: First distribute the input array evenly across the processing units (PUs), then compute each local chunk of the array with its PU and finally join all elements together. The inner function `f` is run sequentially.

Statements like `map incr` [4] can be perfectly parallelized. However, it reaches its limit when considering statements like `map (map incr)` or `map (map (map incr))`. Within Flat Data Parallelism only the outer-most level is run in parallel. That can lead to unbalanced work distribution. The sub-arrays can be of very different length. Figure 2.1 shows an example. Running the inner elements in parallel requires a different approach in dividing up the workload.

Nested Data Parallelism lifts these limitations. In NDP `f` can be a parallel operation either. All levels of nesting can be executed in parallel - hence the name: **Nested** Data Parallelism. It achieves this by transforming the source program into a functionally equivalent flat data-parallel program. This (non-trivial) transformation is called 'Flattening' or 'Vectorization' and includes subsequent compiler optimisations.

The entire program transformation can be broken down in three steps. Each step introduces its new data types. They are presented below:

1. *Vectorization* - The nested arrays `[:a:]` which are used by the programmer are converted to flat type-dependent array representations - namely `PA a`. The flattening is also applied to nested parallel functions like those mentioned in the previous paragraph.

2. *Communication Fusioning* - By inlining the definitions of the parallel functions and using semantics-preserving rewrite rules one can eliminate specific synchronisation points and create tight processing pipelines. It uses `Dist (PA a)` to denote distributed chunks of the global array `PA a`.

[3] Papers on NDP in Haskell: Jones (2008), Chakravarty et al. (2007), Leshchinskiy (2005), Leshchinskiy et al. (2006), Keller and Chakravarty (1999) and Lippmeier et al. (2012).

[4] `incr :: Int -> Int` increments an integer.

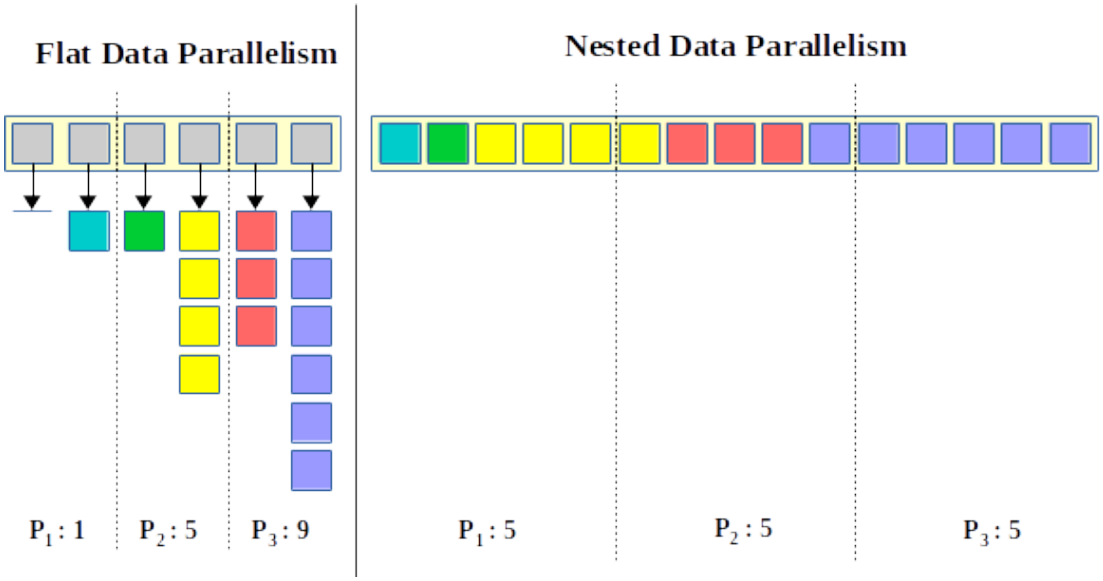

Figure 2.1: Workload distribution in Flat and Nested Data Parallelism

3. *Stream Fusioning* - Sequential functions local to each PU can further be optimised to reduce the number of intermediate arrays and to create tight loops. It uses `Vector a` and `Stream a`[5] to implement the local array chunks and to fuse loops, respectively.

A few functions are predefined for parallel arrays. They correspond to functions used in conventional functional programming. Their names and their types are given in 2.1.

2.2.1 Vectorization

Vectorization involves two major steps - *Type dependent representation* and *Lifting*. Both will be explained in detail below.

Type dependent representation

Type dependent representation enables the programmer to work with arrays containing complex data structures without scarifying. It is described in Chakravarty and Keller (2003) and Chakravarty et al. (2005).

[5] `Stream` is a data structure designed to specifically enable non-recursive definitions of fusing functions (e.g. `mapP`) by moving the recursion from the function to a step-wise stream-ful iteration. See Mainland et al. (2013).

Table 2.1: Parallel functions in NDP

function	type	description
(!:),indexP	`[:a:] -> Int -> a`	indexes the array; zero-based
lengthP	`[:a:] -> Int`	returns the length of the array
headP	`[:a:] -> a`	returns the first element
lastP	`[:a:] -> a`	returns the last element
mapP	`(a -> b) -> [:a:] -> [:b:]`	applies a function on all elements
zipWithP	`(a -> b -> c)` `-> [:a:] -> [:b:] -> [:c:]`	zip together pairs of elements by `f` and return an array of results [6]
sortP	`[:Int:] -> [:Int:]`	sorts an array
sumP	`[:Int:] -> Int`	sums the arrays elements
concatP	`[:[:a:]:] -> [:a:]`	removes a level of nesting
unconcatP	`[:[:a::]] -> [:b:] -> [:[:b:]:]`	exposes a structure to a flat array
replP	`Int -> a -> [:a:]`	replicates an element

Consider for example `[:Int:]` and `[:[:Int:]:]`. The former can be easily implemented using a contiguous region of memory and inserting the bytes. The latter however cannot be implemented equally. The sub-arrays could be of different size. To allocate a block of memory, one needs to know the size of all arrays beforehand. Implementing nested array naively would use an array of pointers to to arrays of integers. Pointers however are undesirable, since they decrease Cache Locality. CPU Caching is crucial for performance and is directly linked to Cache Locality. Denning (2005).

Nested arrays need a different approach for representation. That approach is the separation of data and structure. The following example describes how a nested array like `[[1,2,3],[4,5],[],[6]] :: [:[:Int:]:]` can be implemented to increase Cache Locality: [7]

```
1 array :: PA (PA Int)
2 array = AArr {
3   data = [# 1,2,3,4,5,6 #],
4   indices = [# 0,3,5,5 #]
5   lengths = [# 3,2,0,1 #]
6 }
```

All data is packed together into a data field - regardless of nesting. All structural information is divided into two arrays of integers. The indices-array contains - for every subarray in the

[6] e.g. `zipWithP (+) [:1,2,3:] [:1,3,4:] = [:1+1,2+3,3+4:]`. The arrays as assumed to be of equal size.

[7] `[#1,2,3#]` is the notation used for a contiguous-memory byte-array.

original array - the index of the first element if it were to be indexed into the `data` field. The lengths describe the lengths of each sub-array. Each sub-array corresponds to a pair of index and length. For example, the second sub-array (`[4,5]`) corresponds to the second index (3) and second length (2). Extracting 2 elements starting at index 3 in the `data` array will return exactly these two elements.

This flat representation uses the new type `PA (PA Int)` instead of the old `[:[:Int:]:]`. The functions `concatP` and `unconcatP` become constant time operations. Removing a level of nesting is as simple as discarding the segment descriptors. Adding a nesting structure of an existing array to a flat array is implemented by adding the segment descriptor of the nested array to the flat array.

The flat representation of irregularly nested arrays and the constant time (un-)flattening operations are crucial insights in NDP.

Implementation for all other types - such as arrays of ints, arrays of doubles, are also given. For example `[:1,2,3:] :: [:Int:]` can be simply represented as:

```
1 array :: PA Int
2 array = AInt [# 1,2,3 #]
```

Finally, during flattening the compiler transforms every parallel function (like `fooP`) to their predefined scalar counterpart (like `fooPS`). The scalar functions work directly on the flat representation and are designed to exploit them for efficiency. `concatPS` and `unconcatPS` are examples thereof. In fact, functions like `fooP` have no implementation because they are entirely replaced by their `fooPS` counterparts. They are solely used to simplify the programmer view.

Lifting

During lifting, all occurrences of `mapPS f` are replaced by `fL`. `fL` is the lifted version of the original function. Compared to the original scalar function `f :: a -> b`, the lifted function applies the mapping over arrays. It is given by `fL :: PA a -> PA b`. For user-defined functions, the lifted function is defined by lifting the definition of `f` recursively. The key is now the definition of the lifted functions for the built-in functions. The most important one among them is lifted `mapPS` - namely `mapPL`. To `mapPS` of type `(a -> b) -> PA a -> PA b`, `mapPL` is of type `(a -> b) -> PA (PA a) -> PA (PA b)` [8] . To overcome the limitations of flat data parallelism, its implementation has to map over all elements at once (instead of mapping over the outer-nesting level only). The following implementation does that.

[8] Its type actually is `PA (a :-> b) -> PA (PA a) -> PA (PA a)` and works with a parallel array of functions as the mapping argument. However, that is a detail independent of the conclusions of this thesis.

```
1 mapPL :: (a -> b) -> PA (PA a) -> PA (PA b)
2 mapPL f xss =
3   unconcatPS xss
4   . fL
5   . concatPS
6   $ xss
```

The definition of `mapPL` is curcial. First it flattens the array (line 5), then it applies the flat data-parallel operation (line 4) using `fL` and finally the array is un-flattened to the original structure (line 3). [9] Figure 2.2 shall visually aid the understanding using the example `mapP (mapP incr)`. The call to `incrL` is divided among the processors evenly as shown in figure 2.1.

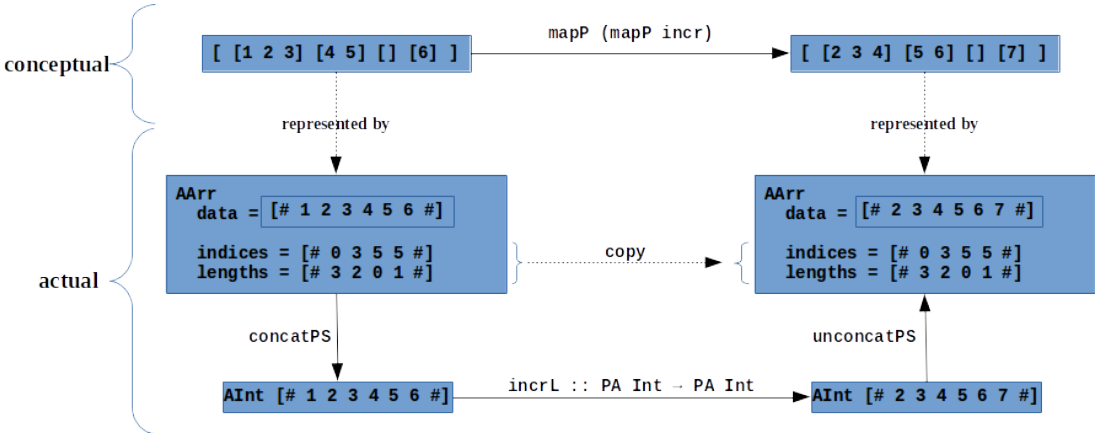

Figure 2.2: Conceptual view and actual implementation of `mapP (mapP incr)` [10]

This is the key insight in NDP. Nested parallel functions are transformed into calls of `mapPL` which avoid the nesting of the arrays entirely and map over the array is a single pass. Since the (un-)flattening functions are constant time operations, there is only the cost involved in mapping over all the elements. Using this procedure, one can transform nested data parallel programs to flat data parallel programs. The former is easier to write in and flexible - while the latter can be implemented directly and is efficient.

[9] This implementation is different from the original in one important aspect - namely the handling of pre-supplied arguments in the function. Our implementation only handles functions without pre-supplied arguments - but it is simpler to convey than the original in in Leshchinskiy (2005) and Leshchinskiy et al. (2006). More accurately, one has an additional expression `expandPS xss fEnv`. Introducing it wouldn't affect the overall situation in this thesis. Therefore, it was omitted.

[10] The commas in-between the elements have been omitted for visual clarity. This thesis will sometimes use such spaces for separation instead of commas.

Vectorization itself is a complex transformation and many details were omitted. More details can be found in Jones (2008).

2.2.2 Communication Fusion

During Communication Fusion [11] lifted functions like `incrL` are inlined. They generally have following form:

```
1 incrL :: PA Int -> PA Int
2 incrL = joinD . mapD incrS . splitD
```

The types of the new functions are explained in table 2.2. Essentially, the lifted functions are implemented by explicitly splitting the array across all PUs (`splitD`), applying the original function (in this case `incrS`) on each local chunk sequentially (`mapD`) and finally joining all chunks `joinD`.

The type `Dist a` denotes a distributed value of type a. Distributable values can for example be arrays or linked lists. In case of arrays, the array is chunked and distributed evenly. In case of linked lists, the lists is chunked and inter-PU pointers are used when each chunk reaches its end.

Another aspect to note is the following distinction: The use of `PA a` means, that the entire array is processed by a single PU locally and then redistributed globally to all PUs. The use of `Dist a` means, that each PU is locally processing its chunk of the distributed value.

After these introducing words, one can take a look at an example of the expressions Communication Fusion is designed to optimise. Here is an example.

```
1 myFunc :: PA Int -> PA Int
2 myFunc = mult2L . incrL
```

`mult2L` is a function which doubles each element in the array. Currently, the function works in two steps - applying each operations individually. First, it splits the array, increments its elements and joins the array (`incrL`). Second, it splits the array again, doubles its elements and rejoins the array (`mult2L`). A more efficient alternative would execute both operations locally in a single pass rather than two. The compiler however can optimise this.

Effective optimisation strategies in referentially transparent programming languages like Haskell are Inlining [12] and Rewrites Rules [13] . Inlining refers to the inlining of definitions of functions and variables. Applying that on the previous snippet of code gives the following:

[11] as described in Keller and Chakravarty (1999) and Chakravarty and Keller (2000)
[12] Jones and Marlow (2002)
[13] Peyton Jones et al. (2001)

```
1 myFunc :: PA Int -> PA Int
2 myFunc = joinD . mapD mult2S . splitD . joinD . mapD incrS . splitD
```

Then rewrite rules are used. They allow the specification of general semantic-preserving laws and allow the compiler to rewrite parts of the code according to them. They are specified by humans and table 2.3 introduces a few of them. Among them is "splitD/joinD". It states the general law, that joining and re-splitting an array does not change the array at all. This is clear for humans - but not for the compiler. By specifying such rules the compiler is aided in optimising the code. In our case the compiler finds the splitD/joinD pair and applies the rule to get:

```
1 myFunc :: PA Int -> PA Int
2 myFunc = joinD . mapD mult2S . mapD incrS . splitD
```

The compiler eliminated communication in-between two phases of computation. This is a step towards more efficient evaluation. Using another rule, namely "mapD/mapD", the compiler can further optimise to.

```
1 myFunc :: PA Int -> PA Int
2 myFunc = joinD . mapD (mult2S . incrS) . splitD
```

Communication Fusion did not only reduce the communication but also packed together consecutive operations. This is desirable behaviour. Now the compiler can further optimise local operations using Stream Fusion.

2.2.3 Stream Fusion

Stream Fusion concerns the optimisation of recursive composed functions into a single loop. It is a complex topic [14] and this section will give a broad overview. For the purposes of this thesis it is sufficient to state the following:

- Stream Fusion is applied similarly to Communication Fusion.
- Instead of inlining `fooL` functions, the `fooS` functions are inlined.
- Instead of mergeing "splitD/joinD", pairs of "unstream/stream" are merged.
- Instead of joining "mapD/mapD", pairs of "mapS/mapS" are joined.
- `Stream a` is a special stream-full data structure. Functions over streams are implemented non-recursively. They are crucial for Stream Fusion.

[14] A few papers on Stream Fusion: Mainland et al. (2013), Coutts et al. (2007), Chakravarty and Keller (2001) and Leshchinskiy et al. (2002)

- Any streams left over after optimisations are converted back into. `Vector a`. Streams are only used for optimisation, but not as a collection.

Applying Stream Fusion creates the following code:

```
1 myFunc :: PA Int -> PA Int
2 myFunc = joinD . mapD (mapS (mult2 . incr)) . splitD
```

In contrast to the prior - two step - distributed computation, the new function has been optimised to reduce communication and applies both operations in a single loop per PU.

Functions and Rewrite Rules

The transformation in chapter 6 is going to make use of many functions. Most common ones are explained in table 2.2. The table also holds for other functions with same suffix - e.g. the `mapP` row similarly applies to `scanlP`, `groupP` etc.

Most of the functions are simply inlined during optimisation. They do not exist at runtime anymore and therefore they are marked 'not-executed'. Besides these functions, the compiler needs rewrite rules. Most important ones are described in table 2.3.

A word on accuracy

The project of NDP in Haskell is - even after 15 years - still in *work in progress*. Due to frequent changes, the source papers use conflicting notation and refer to different statuses of progress. Inconsistent literature and a project still in work is a problem for a thesis on that topic. It is not simple to use the original ideas from NESL directly on Haskell as there are great differences in-between them (not mentioning the 15 years of research already in it).

Therefore, in this thesis, the author has improvised on a few conflicting or missing details to create an overall consistent view on NDP. The author specifically used implementations which could really have been used in NDP.[15] The reader is hereby noted that the details mentioned here are implementable - but not necessarily an accurate representation of the current state of progress.

After this introduction to NDP, the next sections will explain parallel complexitiy measures and Histogram Balancing.

[15] E.g. `groupP` as introduced in chapter 5 is not included in NDP currently. However, its implementation described there is perfectly possible.

Table 2.2: Overview of Functions and Phases in NDP

function	first appearance/ execution context	type/ description
mapP f xs	Programmer-view not executed	`(a -> b) -> [:a:] -> [:b:]` **P**arallel array functions the programer uses
mapPS f xs	Vectorization not executed	`(a -> b) -> PA a -> PA b` **P**arallel **S**calar functions over flat arrays
indexPL is xs	Vectorization not executed	`PA Int -> PA (PA a) -> PA a` **P**arallel **L**ifted indexing on flat arrays. It's scalar function is `indexPS :: Int -> PA a -> a`
mapD f x	Communication Fusion mapD: all PUs f: local per PU	`(Vector a -> Vector b) -> Dist (PA a) -> Dist (PA b)` Each PU applies `f` on it's chunk of the **d**istributed value. True parallelism here!
fooS f xs	Communication Fusion local per PU	`Vector a -> Vector b` Applies a function **s**equentially. `Vector` is the Haskell implementation of conventional PU-local in-memory arrays.
splitD	Communication Fusion all PUs	`PA -> Dist (PA a)` splits a glboal array into chunks and distributes them to each of the PUs
joinD	Communication Fusion all PUs	`Dist (PA a) -> PA a` join a distributed array into a global array
replD	Communication Fusion local per PU	`Int -> a -> Dist (PA a)` create the local chunk of a distributed replication directly
stream	Stream Fusion not executed	`Vector a -> Stream a` convert from an array to stream-ful data
unstream	Stream Fusion not executed	`Stream a -> Vector a` convert back to an array
mapSt f xs	Stream Fusion not executed	`(a -> b) -> Stream a -> Stream b` Applies a function on a **st**ream of values

2.3 Parallel Complexity Measures

In Parallel Computing, the notion of time complexity has to be revisited. The time complexity becomes dependent on the number of processors available.

Two key measures of an algorithm are *work* complexity and *depth* complexity. Work is defined to be the time (counted in number of operations) the algorithms needs, if it were executed on a single processor. Depth is defined as the longest chain of sequential data

Table 2.3: Rewrite Rules in NDP

name	rewrite	description
splitD/joinD	splitD . joinD = id	Joining and splitting an distributed array is equivalent to a no-op.
mapD/mapD	mapD g . mapD f = mapD (g . f)	Two consecutive mappings are equivalent to a single mapping with both of the functions
mapD/replD	mapD f . replD n = replD n . f	Replicating a value and mapping all values is equivalent to directly applying the function and subsequently replicating it.
splitD/replPS	splitD . replPS n = replD n	Replicating and splitting a value is equivalent to creating the local chunks directly. ReplD implements this and knows which chunk its PU is responsible for.
ZipReplSplit	zipWithD f (replD a) . splitD = mapD (f a)	Zipping with a replicated value already in scope - is equivalent to - applying the mapping with the value curried into the function.
mapD/zipWithD	mapD f . zipWithD g xs = zipWithD (λx y -> f (g x y)) xs	A map operation after a zip operation is equivalent to a single zip operation that applies both `f` and `g`.
unstream/stream	unstream . stream = id	Converting back and fourth is equivalent to doing nothing.

dependency. It is also the time needed if one had an unlimited number of processors - or if one had as many processors as the length of the longest intermediate array.

There are limited ways to calculate work and depth. Both are defined recursively over work and depth of their sub-expressions. Generally, work is the sum of operations involved in all of the sub-expressions - while depth is the maximum of number of operations involved in any particular chain of sub-expressions. In this thesis, the functions $\mathrm{W}(\cdot)$ and $\mathrm{D}(\cdot)$ denote work and depth, respectively. In some cases, syntactic constructs like $\mathrm{W}(f, x)$ are used to denote the work involved in *applying* `f` on `x`. It does not include the work involved in calculating `x` in the first place. Sometimes, the thesis denotes $\mathrm{W}(n)$ or $\mathrm{D}(n, gmax)$ to emphasize, that the work and depth is dependent of the length of the input or of the parameter $gmax$. In these cases, the function referred to is clear from the context. Table 2.4 gives work and depth complexities for the parallel primitives of table 2.1. Landau-Notation $O(\cdot)$ is used for the complexity classes.

Table 2.4: Work and Depth - Definitions and Complexities

function	work	depth
g . f	$\mathrm{W}(g) + \mathrm{W}(f)$	$\mathrm{D}(g) + \mathrm{D}(f)$
mapP f xs	1 + $\mathrm{W}(xs) + \sum_{x \in xs} \mathrm{W}(f, x)$	1+ $\mathrm{D}(xs) + \max_{x \in xs} \mathrm{D}(f, x)$
zipWith f as bs	1 + $\mathrm{W}(as) + \mathrm{W}(bs)$ $+ \sum_{pairs(x,y) \in (xs,ys)} \mathrm{W}(f, x, y)$	1 + $\mathrm{D}(as) + \mathrm{D}(bs)$ $+ \max_{pairs(x,y) \in (xs,ys)} \mathrm{D}(f, x, y)$
sortP	$O(n \log n)$	$O(\log n)$
sumP	n	log n
replP	n	1
(!:),indexP	1	1
lengthP	1	1
headP	1	1
lastP	1	1
concatP	1	1
unconcatP	1	1

The work and depth of `mapP` embodies the idea, that both are the sum or maximum of the sub-expressions. Work and depth of a function composition are the sum of the work and depths of each function.

An example is given now. Given the expression `ys = mapP inrc (replP n 1)`, one can apply the formulas to calculate its work and depth complexity.

$$
\begin{aligned}
\mathrm{W}(ys) &= 1 + \mathrm{W}(replP, n, 1) + \sum_{x \in xs} \mathrm{W}(incr, x) \\
&= 1 + n + n \\
&\in O(n) \\
\mathrm{D}(ys) &= 1 + \mathrm{D}(replP, n, 1) + \max_{x \in xs} \mathrm{D}(incr, x) \\
&= 1 + 1 + 1 \\
&\in O(1)
\end{aligned}
$$

xs is the intermediate array created by `replP`. The expression has constant depth - and therefore is expected to be executable in constant time if given enough processors. If one had as many PUs as the array is in size - one could assign each PU one element of the array and process the elements locally. Each PU would execute a constant number of operations This is indeed constant time in total.

These measures, as introduced in Blelloch (1996), work neatly within Nested Data Parallelism. Consider for example the expression `mapP (mapP incr) ass` where the nested array `ass` has dimensions $w \times h$. Within NDP, the nested parallel operation is flattened and the array is only once mapped over. Therefore, work of $O(w \cdot h)$ and depth of $O(1)$ are expected. This is indeed the result.

$$
\begin{aligned}
\mathrm{W}(mapP, mapP, incr) &= 1 + \sum_{as \in ass} \mathrm{W}(mapP, incr, as) \\
&= 1 + \sum_{as \in ass} (1 + \sum_{a \in as} \mathrm{W}(incr, a)) \\
&= 1 + \sum_{as \in ass} (1 + \sum_{a \in as} 1) \\
&= 1 + \sum_{as \in ass} (1 + h) \\
&= 1 + w \cdot (1 + h) \\
&\in O(w \cdot h) \\
\mathrm{D}(mapP, mapP, incr) &= 1 + \max_{as \in ass} \mathrm{D}(mapP, incr, as) \\
&= 1 + \max_{as \in ass} (1 + \max_{a \in as} \mathrm{D}(incr, a)) \\
&= 1 + \max_{as \in ass} (1 + \max_{a \in as} 1) \\
&= 1 + \max_{as \in ass} 2 \\
&= 1 + 2 \\
&\in O(1)
\end{aligned}
$$

Work and depth are an useful tool in measuring programs complexity. In the remaining thesis, the work and depth complexities will be presented mostly without explicit derivation.

2.4 A_C: Histogram Balancing

This section will introduce Histogram Balancing Masters (1999). It will state the problem and give an overview of its solution.

Problem

Suppose an $w \times h$-8-bit-gray-tone image with low contrast. [16]

Figure 2.3: An image with low contrast

The goal is to make details more visible to the human viewer. Inspecting the images histogram reveals new insights. [17]

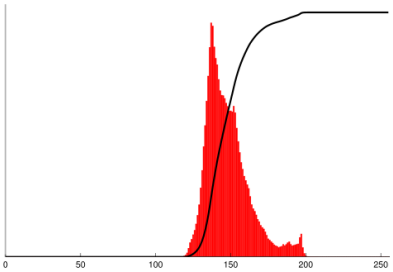

Figure 2.4: The histogram of the original image with absolute(red) and accumulative(black) count.

A histogram shows the gray tone distribution of the image. The x-axis denotes the gray tone and the y-axis denotes the number of pixels with a gray tone x (red). The black curve denotes the total number of pixels with a gray tone g such that $g \leq x$.

Details are difficult to recognize in the original image due to the tight packing in the histogram. The entire image limits its values to the range [120..205]. Histogram Balancing solves this problem by defining a mapping $f : Graytone \rightarrow Graytone$, such that the accumulating

[16] Photo by Phillip Capper. Source: https://en.wikipedia.org/wiki/File:Unequalized_Hawkes_Bay_NZ.jpg

[17] made by Wikipedia user Jarekt. Source: https://en.wikipedia.org/wiki/File:Unequalized_Histogram.svg

count of the resulting image increases as uniformly as possible from 0 to 255. The histogram 2.5 [18] is the envisioned goal.

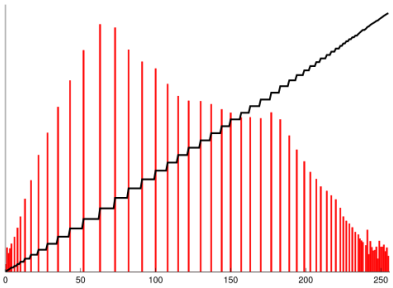

Figure 2.5: The images balanced histogram

This mapping can be defined by spreading out the gray tones such that more frequent gray tones get a larger range to occupy. This idea is explained detailed next.

Algorithm

To describe the image transformation $hbalance : Image \to Image$ a few definitions are necessary:

$$\begin{array}{ll} Image & = (Width, Height, Width \times Height \to Graytone) \\ Histogram_a & : Graytone \to a \\ gmax & : Graytone \end{array}$$

An $Image$ assigns a gray tone to each pair of coordinates. It is bounded in width and height. A $Histogram_a$ assigns a value of type a to each gray tone. For $a = Int$, this becomes is an (accumulated) histogram. For $a = Double$, one can describe normalised histograms. $gmax$ is the maximum gray tone for the gray tones in use (e.g. 255 for 8-bit gray tones).

The method is broken down into the following functions:

1. $hist : Image \to Histogram_{Int}$
   It calculates the histogram of an image.

2. $accu : Histogram_{Int} \to Histogram_{Int}$
   It calculates the accumulating histogram from the original histogram.

[18] made by Wikipedia user `Jarekt`. Source: https://en.wikipedia.org/wiki/File:Equalized_Histogram.svg

3. $normalize : Int \times Int \times Histogram_{Int} \to Histogram_{Double}$
   It normalizes the accumulated histogram to a range from 0 to 1. The arguments are denoted $a0$ and $agmax$. $a0$ denotes the number of pixels having the lowest gray tone. $agmax$ denotes the total number of pixels in the image. Normalisation is defined in mapping the histogram values by $x \mapsto \frac{(x-a0)}{agmax-a0}$.

4. $scale : Graytone \times Histogram_{Double} \to Histogram_{Int}$
   It scales the normalized values to the maximum gray tone (gmax) and rounds down to the nearest integer. Scaling is defined in mapping the histogram values by $x \mapsto \lfloor x \cdot gmax \rfloor$.

5. $apply : Histogram_{Int} \times Image \to Image$
   It maps each gray tone to its new value as dictated by the histogram in the first argument.

Given these functions $hbalance : Image \to Image$ can be defined as:

$$\begin{aligned} h &:= hist(img) \\ a &:= accu(h) \\ n &:= normalize(a(0), a(gmax), a) \\ gs &:= scale(gmax, n) \\ hbalance(img) &:= apply(gs, img) \end{aligned}$$

Concrete implementations are given in the coming chapters. Applying the algorithm to the example image gives 2.6 [19] . Details are indeed more distinguishable in the balanced image.

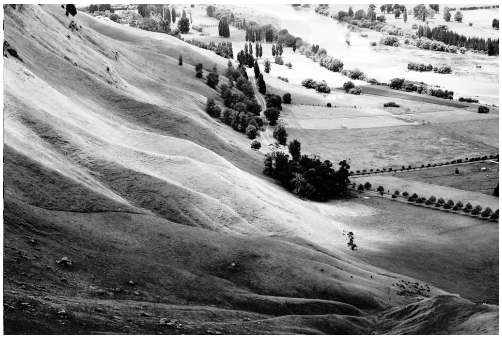

Figure 2.6: The balanced image

[19] Photo by Phillip Capper. Source: https://en.wikipedia.org/wiki/File:Equalized_Hawkes_Bay_NZ.jpg

Histogram Balancing is a frequently used algorithm in image processing. It is one of first methods applied on images to decrease their complexity. These balanced images are often then forwarded to sophisticated image processing algorithms.

The introduction is over and the programs P_{seq}, P_{man} P_{nest} and P_{vect} are presented and analysed next.

3 Sequential: P_{seq}

"Only one who devotes himself to a cause with his whole strength and soul can be a true master. For this reason mastery demands all of a person.

Albert Einstein

This chapter introduces P_{seq}- a sequential implementation of Histogram Balancing. The implementation is given first. Then its work and depth complexities - as introduced in section 2.3 - are given.

3.1 Implementation

P_{seq} is a direct implemenation of A_C as described in 2.4. First the data structures used are presented. Then each component of the implementation is given. Afterwards the components are assembled to P_{seq}.

Data Structures The implementation uses two data types.

```
type Image = PtrVector (PtrVector Int)
type Hist a = TreeMap Int a
```

`PtrVector a` is a pointer-based array holding values of type `a`. The use of pointers enables them to be nested. Thus they can be directly used to represent two-dimensional images. `TreeMap k a` is a binary search tree indexed by keys of type `k` and containing values of type `a`. They are used for the representation of a histogram. [1] Functions over `PtrVector` are suffixed `-V`. Functions over `TreeMap` are suffixed `-M`.

Histogram Calculation The steps for the creation of the initial histogram are given below:

[1] Due to Immutability, arrays are not an option. They require the replication of the entire array to change a single value.

```
hist :: Image -> Hist Int
hist = foldrV (\i -> insertWithM (+) i 1) emptyM . concatV
```

`hist` proceeds in two steps. First the image is flatten into an one-dimensional array. Then, a `TreeMap` is created counting the number of occurrences of each gray tone. `foldrV` is linear in the size of the flattened image array. `insertWithM` is logarithmic in the number elements inserted into the map. Its size is bounded by the number of gray tones - namely `gmax+1`.

Accumulation Calculating the accumulated histogram can be implemented by a prefix sum over values. It is linear in the size of the map.

```
accu :: Hist Int -> Hist Int
accu = scanlM (+) 0
```

Normalisation After accumulation, one has to normalise the histogram. The normalisation is a direct implementation of its formula in 2.4.

```
normalize :: Int -> Int -> Hist Int -> Hist Double
normalize a0' agmax' as =
   let a0 = fromIntegral a0'
       agmax = fromIntegral agmax'
       divisor = agmax - a0
   in  mapM (\freq' -> (fromIntegral freq' - a0) / divisor) as
```

It applies the mapping over the tree-map using `mapM`. `fromIntegral` explicitly converts from `Int` to `Double` since Haskell clearly distinguishes them. Variable names with a prime (') denote values of type `Int`. Variable names without a prime denote `Doubles`. This naming convention is equally used in P_{man}, P_{nest} and P_{vect}.

Scaling Scaling occurs similar to normalisation. It is implemented by a mapping over all values in the histogram.

```
scale :: Int -> Hist Double -> Hist Int
scale gmax = mapM (\d -> floor (d * fromIntegral gmax))
```

Apply The application of the gray tone mapping to the images pixels is implemented by a nested `mapV` over the image. It uses `lookupLessEqualM` to lookup the values for the histogram. (It reverts back to a lower gray tone, if the gray tone is not found in the map.)

```
apply :: Hist Int -> Image -> Image
apply as img = mapV (mapV (lookupLessEqualM as)) img
```

Histogram Balancing Having defined the components, one can now directly define P_{seq}:

```
1 hbalance :: Image -> Image
2 hbalance img =
3  let h = hist img
4      a = accu h
5      a0 = firstM a
6      agmax = lastM a
7      n = normalize a0 agmax a
8      gs = scale gmax n
9      img' = apply gs img
10 in  img'
```

First the histogram is created (line 3). Then it is accumulated (line 4). After that it is normalised (line 5 to 7) and scaled(line 8). And finally, the gray tone mapping is applied and returned (line 9 to 10). It is defined exactly as previously envisioned.

3.2 Complexities

For sequential algorithms, work and depth fall together with their time complexity class. This is true for the purely sequential program P_{seq}.

Before the time complexity is calculated, a few variables are introduced:

n: the number of pixels in the image

h: the height of the image and equally the the number of sub-arrays in the nested array

w: the width of the image and equally the length of the sub-arrays

The complexity for the functions involved in P_{seq} are given in table 3.1. The functions are grouped together by their context. Each component of P_{seq} (e.g. `hist`, `accu`) is given a group with its each of its sub-functions and the components complexity. For example, `hist` uses three functions. `concatV` has linear complexity in the number of total elements in the nested array - that is n. `insertWithM` is an logarithmic time insertion operation into a tree. `foldrV` traverses each pixel, starting with an empty map `emptyM` and adds each pixel into the tree-map using `insertWithM`. There are n pixels in the array and and each insertion takes at-most time logarithmic to the maximum gray tone `gmax`. Therefore its time complexity is $O(n \log gmax)$. Further analysis of the other functions reveals the complexities given in the table.

Finally, `hbalance` uses `hist` ($O(n \log gmax)$) and a few other functions ($O(gmax)$). Therefore P_{seq} has a complexity of $O(n \log gmax + gmax)$.

Table 3.1: Complexities for P_{seq}

function or variable	$O(...)$
hbalance	$n \log gmax + gmax$
firstM	1
lastM	1
hist	$n \log gmax$
concatV	n
emptyM	1
insertWithM	$\log gmax$
foldrV	$n \log gmax$
accu	gmax
scanlM	gmax
normalize	gmax
scale	gmax
mapM	gmax
apply	$n \log gmax = w \cdot h \cdot \log gmax$
lookupLessEqualM	$\log gmax$

The next chapter introduces the first parallel implementation P_{man}.

4 Manually parallelized: P_{man}

"When we had no computers, we had no programming problem either. When we had a few computers, we had a mild programming problem. Confronted with machines a million times as powerful, we are faced with a gigantic programming problem. "

Dijkstra, 1998

This chapter will give an implementation of Histogram Balancing that cannot make use of NDP. This is the case when one is only given a few parallel primitives that don't support nesting of operations (at least not without falling back to sequential evaluation). Table 4.1 shows a few of these primitives.

Table 4.1: Flat Data-Parallel Primitives

function	type
parMap	`(a -> b) -> Vector a -> Vector b`
parZipWith	`(a -> b -> c) -> Vector a -> Vector b -> Vector c`
parReplicate	`Int -> a -> Vector a`
parGenerate	`Int -> (Int -> a) -> Vector a`

They are analogous to the parallel functions in NDP. `parGenerate` is a function such that `parGenerate size f` creates a new array of size `size` and uses the generator function `f` to create the elements by their indices. E.g. `parGenerate 5 (λi -> i*i) = [0,1,4,9,16]` . These primitives all have work $O(n)$ and depth $O(1)$.

4.1 Parallel histogram accumulation

To implement Histogram Balancing in parallel, one has to revisit the sequential implementation. The parallel creation of the accumulated histogram is a difficult task. The goal is to try to come up with an low complexity algorithm for the histogram calculation. After a few tries, one can give the following implementation:

```
1 accuHist :: Image -> Hist
2 accuHist [] = parReplicate gmax 0
3 accuHist [x] = parGenerate gmax (\i -> if (i >= x) then 1 else 0)
4 accuHist xs = let (left,right) = splitMid xs
5                   [leftRes,rightRes] = parMap accuHist [left,right]
6               in  parZipWith (+) leftRes rightRes
```

The general idea is to merge accumulation and histogram creation into a single tree-like reduction. To each array of pixels, `accuHist` returns the accumulated histogram of its gray tones. The algorithm can be broken down into two edge-cases and one recursive case . Figure 4.1 gives an example of its evaluation.

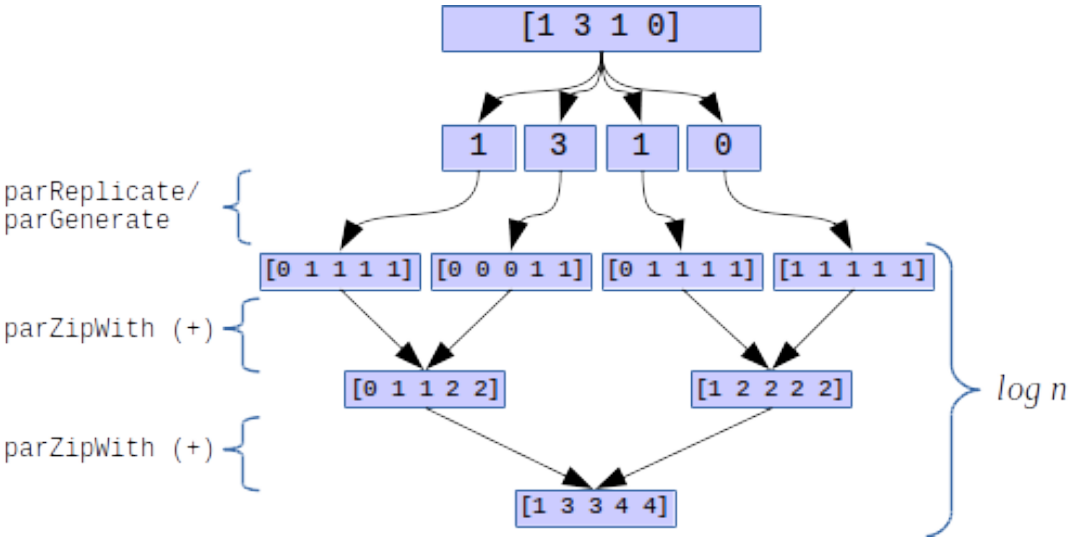

Figure 4.1: Evaluation of `accuHist [1,3,1,0]`

Suppose `gmax = 4` then the algorithm returns `[0 0 0 0 0]` if the input array was empty. If the image only contained a single gray tone x, then it creates its accumulated histogram `[0 0 ... 0 1 ... 1 1]` such that x is the index of the first `1`. For example, for `x = 2` the array `[0 0 1 1 1]` is returned. This is implemented using the `parGenerate` function. Finally,

one has the recursive case. In this case, the the calculation of larger images is broken down by splitting the array into half [1] and applying `accuHist` recursively. Finally, the histograms are combined by element-wise addition with `zipWith(+)` (histogram merging).

This algorithm was carefully constructed after many failed approaches on such an algorithm. Alternatives were considered, but they did not yield an acceptable complexity when compared to `accuHist`.

4.2 Implementation

Given the introduction, the manually parallelized P_{man} can finally be implemented. The parallel primitives can be integrated well into normalisation and scaling. However, `accuHist` and `apply` need to be adapted. The code is given below. `(!)` is used for indexing.

```
type Image = Vector Int
type Hist = Vector Int

hbalance :: Image -> Image
hbalance img =
 let as = accuHist img
    a0 = as ! 0
    agmax = as ! gmax

    sclNrm x = floor ( (x-a0)/(agmax - a0)*gmax )
    gs = parMap sclNrm as

    apply gs = parMap (\i -> gs ! i) img
    img' = apply gs img

 in img'
```

As explained in the previous section, the parallel histogram accumulation has been implemented in `accuHist`. For the gray tone mapping (`apply`) to work, nested arrays cannot be used [2] . One needs to change the entire image representation to a flat array manually. To retrieve a specific pixel one needs to calculate the offset using the image's width. Fortunately, for Histogram Balancing, indexed retrieval of pixels is not needed. However, any subsequent algorithms in the pipeline of image processing would have to cope with the flat image representation directly.

[1] splitting is a constant time operation for these view-based arrays
[2] as they would result into an array of pointers to sub-arrays. This is undesirable due to Cache Locality.

4.3 Complexities

In this section, complexity measures for the functions involved in P_{man} will be given. To calculate work and depth of P_{man}, one needs the measures of all sub-functions and sub-expressions. `accuHist` is not a built-in function - and so needs an individual analysis first.

`accuHist` The recursive case of `accuHist` involves functions of work $O(gmax)$ and depth $O(1)$ - namely `parZipWith (+)` and `splitMid`. The exceptions are the two recursive calls (packed together into a `parMap`).

One can formulate the work of `accuHist` as a recursive function.

$$\mathrm{W}(n, gmax) = \begin{cases} gmax & \text{if } n \leq 1 \\ 2\mathrm{W}(\frac{n}{2}) + gmax & \text{else} \end{cases}$$

where the edge-cases and recursive-cases correspond one-to-one to the definition of `accuHist`. Such a recurrence relation can be resolved by tying the knot or using the Master Theorem's first case [3] [4] . The following equations shall tie the knot:

$$\begin{aligned} \mathrm{W}(n, gmax) &= \begin{cases} gmax & \text{if } n \leq 1 \\ 2^0 gmax + 2^1 \mathrm{W}(\frac{n}{2}) & \text{if } n = 2 \\ 2^0 gmax + 2^1 gmax + 2^2 \mathrm{W}(\frac{n}{4}) & \text{if } n = 3 \\ 2^0 gmax + 2^1 gmax + \ldots + 2^{\log n - 1} gmax + 2^{\log n} \mathrm{W}(1) & \text{else} \end{cases} \\ &= \text{(... tying the knot ...)} \\ &= gmax \sum_{i=0}^{\log n} 2^i \\ &= gmax(2^{\log n + 1} - 1) \\ &= gmax(2n - 1) \\ &\in O(n \cdot gmax) \end{aligned}$$

The work involved is an product of the number of gray tones and the number of pixels $O(g \cdot gmax)$. The reduction tree has height logarithmic in the size of the input array. The input array is the image and has size n. Therefore, one can conclude $\mathrm{D}(n, gmax) \in O(\log n)$.

[3] Master theorem: Cormen et al. (2001)

[4] However, the Master Theorem does not apply directly because it treats `gmax` as a constant - and not as a variable parameter. The Master Theorem gives $O(n)$ whereas tying the knot gives the more accurate class $O(n \cdot gmax)$.

Putting it together Given the code for P_{man}, one can now give work and depth complexities. These complexities are given in table 4.2. The table summarises the work and depths of each of the calls. It is calculated by applying the formulas from section 2.3.

Table 4.2: Complexities for P_{man}

function or variable	W $\in O(...)$	D $\in O(...)$
hbalance	n · gmax	log n
apply	n	1
parMap sclNrm	gmax	1
accuHist	n · gmax	log n
accuHist	n · gmax	log n
splitMid	1	1
parZipWith	gmax	1
parReplicate	gmax	1
parGenerate	gmax	1
arr ! i	1	1

The outer-most work and depth of P_{man} is given below:

$$\begin{aligned}
\mathrm{W}(n, gmax) &= \mathrm{W}(accuHist) + \mathrm{W}(parMap, sclNrm) + \mathrm{W}(apply, gs) \\
&= n \cdot gmax + gmax + n \\
&\in O(n \cdot gmax) \\
\mathrm{D}(n, gmax) &= \mathrm{D}(accuHist) + \mathrm{D}(parMap, sclNrm) + \mathrm{D}(apply, gs) \\
&= \max\{\log n, 1, 1\} \\
&\in O(\log n)
\end{aligned}$$

One can note, how work and depth of P_{man} is entirely bounded by the complexities of `accuHist`. Improvements to `accuHist` complexities will improve P_{man} either.

Before moving to the next chapter - one shall be reminded that P_{man} involved much manual work. It was not a direct translation of the algorithms description. It, especially requires the subsequent algorithms to use a flat image representation.

The next chapter covers P_{nest}- an implementation using NDP.

5 Nested-Data-Parallel: P_{nest}

> "Yields falsehood when
> preceded by its quotation"
> yields falsehood when
> preceded by its quotation
>
> Quine's paradox

This chapter describes an implementation of Histogram Balancing that uses Nested Data Parallelism in Haskell (as in Jones (2008)). First, a few predefined functions are presented to increase their the understanding of their operational behaviour. Then the implementation is presented and finally its complexities are calculated.

5.1 Utilities

Scanl

Parallel prefix sum has well studied efficient implementations. One of them is the following:

```
1 scanlP f z xs =
2 joinD
3 . mapD (\(as,a) -> mapS (f a) as)
4 . propagateD f z
5 . mapD (scanlS f z)
6 . splitD
7 $ xs
```

This implementation is designed to reduce communication and therefore increase efficiency. It works in three steps. First, each PU computes its local prefix sum (line 5). Second, the total sum of each of the PUs is propagated - adding up subsequent values (line 4). Third, the updated sum is used to increase the values of the local chunks (line 39). This approach is visualised in figure 5.1

In terms of depth and parallel complexity, the propagation is the bottleneck. However, since the propagation itself is structurally isomorphic to prefix summing itself, one can use

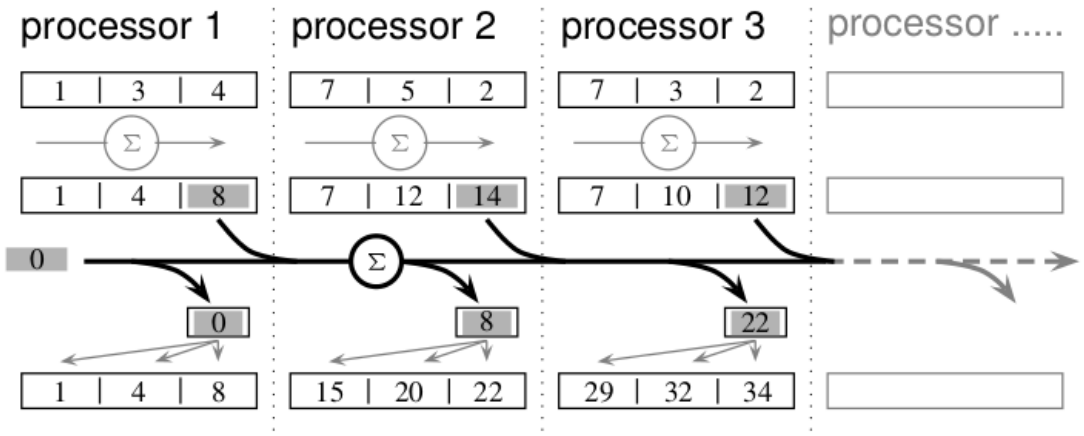

Figure 5.1: Parallel prefix sum in three steps (Figure from Keller and Chakravarty (1999))

a binary-tree-like scheme for propagation for the local values distributed on the PUs. This propagation runs in time logarithmic to the number of PUs. It an be derived from Ladner and Fischer (1980). For the purposes of this thesis, it is sufficient to state the following complexities for `scanlP (+) 0`: $\mathrm{W}(n) \in O(n)$ and $\mathrm{D}(n) \in O(\log n)$.

GroupP

`groupP` is a frequently used function in functional programming. It's type is `[:a:] -> [:[:a:]:]` and given an array it returns an array of arrays, where each subarray contains equal consecutive elements of the source array. For example `groupP [4,2,2,2,2,3,3,1]` becomes `[[4],[2,2,2,2],[3,3],[1]]`. In NDP, the latter is represented by

```
AArr {
  data = [# 4,2,2,2,3,3,1 #],
  indices = [# 0,1,5,7 #]
  lengths = [# 1,4,2,1 #]
}
```

An efficient parallel implementation of `groupP` relies on the following key insight: The `data` field in the nested array is the identical to the source array itself. To implement `groupP` one only needs to efficiently calculate the segment descriptor field and attach it to the source array. This is possible in depth logarithmic to the size of the input array.

As visualised in the figure 5.2, all elements are split onto the PUs first. Then each PU creates a local chunk of a linked list of `(Value,StartIdx,Count)`-Triplets to record its singleton. After that, each level of recursion merges two PUs by merging the last triplet of the left list

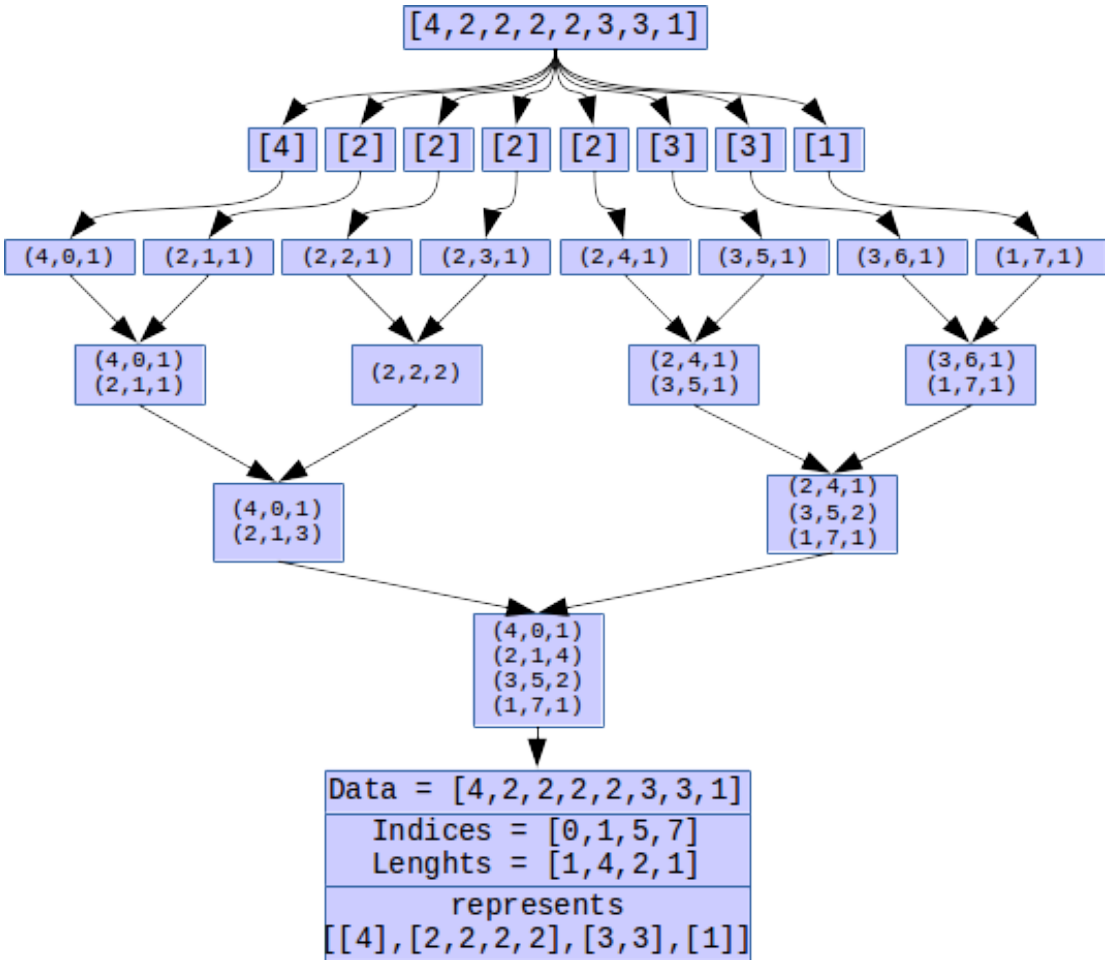

Figure 5.2: An example calculation of groupP. Each box is a PU.

with the first triplet of the right list. If both triplets correspond to the same value, then a new triplet with the total count and the left index is used. If both are unequal, then they are left unchanged. The implementation of `groupP` uses a sub-function `segdSplitMerge` to implement the splitting and merging. This function will be exposed later-on in the vectorization of P_{nest}. Further analysis reveals the complexities $\mathrm{W}(n) \in O(n)$ and $\mathrm{D}(n) \in O(\log n)$. All in all, `groupP` is an operation which can very well exploit the flat representation of nested arrays.

SortP

General parallel sorting can be as simple as a parallel implementation of merge-sort Cole (1988) where the recursive calls are executed in parallel. `sortP` of type `[:Int:] -> [:Int:]` implements this, and has complexities of $\mathrm{W}(n) \in O(n \log n)$ and $\mathrm{D}(n) \in O(\log n)$. [1]

Histogram calculation

Conventional functional programming often enables one to write consise and correct implementations by composing independent and general functions. For example - consider the problem of run length encoding. Given a (possibly long) array, the goal is to create a sparse array that compresses elements by encoding consecutive equal elements with a tuple of the element and its number of occurrences. E.g. it transforms `[4,2,2,2,2,3,3,1]` to `[(4,1),(2,4),(3,2),(1,1)]`. An implementation can be formulated directly.

```
1 runLengthEncode :: [:a:] -> [:(a,Int):]
2 runLengthEncode = mapP (\xs -> (headP xs, lengthP xs)) . groupP
```

This function uses `groupP` to first create sub-arrays of equal consecutive elements. Then it transforms sub-arrays like `[2,2,2,2]` to tuples like `(2,4)`. This is the second step of run length encoding.

Run length encoding is an useful intermediate step in creating the histogram. It implements the "counting"-property of a histogram. To make create a full histogram, the input image has to be flattened first (unconcatP), then sorted(`sortP`) and finally applied `runLengthEncode` on. The sorting is necessary, as it groups together pixels regardless of their position in the original image. The result is an array of type `[:(Int,Int):]` where each element describes a gray tone (first integer in tuple) and its number of occurrences (second integer in tuple). Using this approach, one has a parallel implementation of the histogram calculation. Comping up with this approach is easier than creating an entire reduction as was the case in P_{man}.

5.2 Implementation

One can now give the implementation of P_{nest}. The code implements Histogram Balancing using Nested Data Parallelism in Haskell and is given below:

[1] There exists other sorting mechanism like *Batcher's Bitonic Sort* Batcher (1968) with $O(n \log^2 n)$ work and $O(\log^2 n)$ depth. The author did not use its implementation as it is sophisticated and the workload of its vectorization were expected to be enormous (on top of the all the other work). Retrospectively, the thesis does not inline and expose the internals of `sortP` in P_{vect}. But this was not clear beforehand.

```
type Image = [:[:Int:]:]
type Hist a = [:a:]

hbalance :: Image -> Image
hbalance img =
 let h = hist img
     a = accu h
     a0 = headP a
     agmax = lastP a
     n = normalize a0 agmax a
     gs = scale gmax n
     img' = apply gs img
 in img'

hist :: Image -> Hist Int
hist = sparseToDenseP (gmax+1) 0
      . mapP (\g -> (headP g,lengthP g))
      . groupP
      . sortP
      . concatP

accu :: Hist Int -> Hist Int
accu = scanlP (+) 0

normalize :: Int -> Int -> Hist Int -> Hist Double
normalize a0' agmax' as =
 let a0 = fromIntegral a0'
     agmax = fromIntegral agmax'
     divisor = agmax - a0
 in mapP ( \freq' -> (fromIntegral freq' - a0) / divisor ) as

scale :: Int -> Hist Double -> Hist Int
scale gmax as = mapP ( \a -> floor (a * fromIntegral gmax) ) as

apply :: Hist Int -> Image -> Image
apply as img = mapP (mapP (as !:)) img
```

The first two lines describe the data structure used to encode an image - namely nested data-parallel arrays. As described in the basics chapter - in NDP nested arrays are converted to flat data and a segment descriptor during vectorization. Therefore nesting can be used from the programmer point of view to simplify the programming and to support the intuition - without sacrificing performance.

P_{nest}, like the description of Histogram Balancing, is split up in histogram calculation (lines 20 - 16) ,histogram accumulation (line 23), normalisation (lines 26 - 30), scaling (line 33) and the final gray tone mapping (line 36). The significant different between P_{nest} and P_{seq} is the calculation of the histogram. [2] In P_{nest}, it is implemented based on the approach explained in the previous section. First, it flattens the array, then it sorts it and afterwards it uses run length encoding.

At the end, however, it uses `sparseToDenseP`[3]. The expression `sparseToDenseP size z` to converts the sparse array (typed `[:(Int,Int):]`) to its corresponding dense array (typed `[:Int:]`). The dense array has size `size` and inserts `z` for elements not specified in the sparse array [4] . Using this approach, `hist` can create an array where the element at the index `i` is the number of occurrences of the gray tone `i` in the original image. This is a simpler format for the histogram. [5]

The function `apply` uses a nested application of parallel functions. Within NDP, it will be subject to flattening in P_{vect}.

5.3 Complexities

Using the work and depth measures as introduced in section 2.3, one can assign the following measures to the functions involved. Let n be the number of pixels in an $w \times h$ image, then the complexities are given in table 5.1. For the depths, `hist` and `accu` get logarithmic bounds by either the size of the image and the maximum gray tone, respectively. All in all, P_{nest} has $\mathrm{W} \in O(n \log n + gmax)$ and $\mathrm{D} \in O(\log n + \log gmax)$.

P_{nest} is mostly a direct translation of P_{seq} and Histogram Balancing. A parallel implementation of histogram calculation is given by using only common purely functional processing operations (such as `grouP` and `sortP`). All in all, it involves less workload for the programer then it was the case in P_{man}.

In the next chapter P_{nest} is going to be transformed into P_{vect} manually. It will show, what the compiler would do automatically.

[2] Aside from the the less significant fact, that P_{nest} uses an array for its histogram while P_{seq} uses a binary-search-tree.
[3] Its type is `Int -> a -> PA (Int,a) -> PA a`
[4] E.g. `sparseToDenseP 8 0 [: (1,5),(2,4),(6,7) :] => [: 0,5,4,0,0,0,7,0 :]`
[5] Keeping sparse arrays for the representation of the histogram is a viable option yielding possibly different (maybe better?) complexities.

Table 5.1: Complexities for P_{nest}

function or variable	W $\in O(...)$	D $\in O(...)$
hbalance	$n \log n + gmax$	$\log n + \log gmax$
hist	$n \log n + gmax$	log n
sparseToDenseP	gmax	1
groupP	n	log n
sortP	n log n	log n
concatP	1	1
accu	gmax	log gmax
scanlP	gmax	log gmax
normalize	gmax	1
scale	gmax	1
apply	$n = w \cdot h$	1

6 Vectorized Nested-Data-Parallel: P_{vect}

"Now I will have less distraction."

Leonhard Euler, after loosing
his right eyesight

This chapter will apply the core transformation and optimisations offered by Nested Data Parallelism in Haskell. This chapter will go through the transformations first and then present the final program P_{vect}. The program would have been the results of the compilers *automatic* optimisations before it would be translated into machine code and finally executed. At the end of this chapter, complexity measures for P_{vect} are given.

6.1 Transformations

As presented in section 2.2, the compiler applies three phases of program transformations:

1. *Vectorization* - Flattening of array representations and flattening of nested parallel functions.
2. *Communication Fusioning* - Inlining of parallel functions and the use of Rewrite Rules to eliminate communication.
3. *Stream Fusioning* - Inlining of local traversals and the use of Rewrite Rules to reduce the number of traversals.

The compiler begins with Vectorization.

6.1.1 Vectorization

Applying the vectorization procedure as described in Jones (2008) yields the following code:

```
1 hbalance img :: PA (PA Int)
2 hbalance img =
3 let a = scanlPS plusInt 0
```

```
4          . sparseToDensePS (plusInt gmax 1) 0
5          . (\g -> ATup2 (headPL g) (lengthPL g))
6          . groupPS
7          . sortPS
8          . concatPS
9          $ img
10    n = lengthPS a
11    gs = floorDoubleL
12         . multDoubleL (int2DoubleL (replPS n gmax))
13         . divL
14            (minusL
15             (int2DoubleL a)
16             (replPS n (int2Double (headPS a)))
17            )
18         . replPS n
19         $ minusDouble (int2Double (lastPS a)) (int2Double (headPS a))
20 in unconcatPS img
21     . indexPL (expandPS img gs)
22     . concatPS
23     $ img
```

Various functions (like `hist` and `accu`) have been inlined and are tightly packed together here. The program also replaced the nested `[:a:]` by flat `PA a`. It also replaced polymorphic functions like `fromIntegral` by specific monomorphic primitive machine functions like `int2Double`.

Starting, lines 9 to 4 describe the calculation of the histogram. It's only difference is the use of vectorized scalar functions (e.g.`groupPS`). These functions operate of the efficient flat representation instead of the nested representation.

After that in line 4, the accumulated histogram is calculated.

Lines 19 to 13 and 21 to 11 describe the normalisation and scaling of the gray tones respectively. The vectorized code uses lifted arithmetic functions (like `floorDoubleL`) that operate over arrays. The normalisation constants `gmax'`,`a0` and `divisor` have been inlined and are replicated[1] to the length of the gray tone array before the lifted arithmetic operations are applied.

Finally, lines 32 to 20 describe the mapping of the images gray tones. The nested parallel operation `mapP (mapP (!a))` - formerly a part of `apply` - has now been flatten to use a lifted parallel operation, namely `indexPL`, over a **flat** array of the pixels of the image. This is an embodiment of the key insight in Nested Data Parallelism.

[1] `replPS n x` creates an array of length `n` - all containing the element `x`. It has the type `Int -> a -> PA a`

The compiler however had to add an expression `expandPS img gs` for it to be correct. That is an artefact of the actual flattening procedure. It is responsible for the redistribution of the `gs` array. It does not affect the overall situation (in terms of work and depth) and therefore it is not further discussed.

In total, the program has a few smaller Constant Factors now. This is mainly due to the elimination of nested data structures and operations. However, there is still much room for improvement.

6.1.2 Communication Fusioning

Communication Fusioning consists of inlining definitions of parallel functions and using rewrite rules to eliminate unnecessary communication. Applying this transformation changes the definition of the histogram `a` and the gray tones `gs`. They are separately discussed below.

Histogram calculation

The new definition of the accumulated histogram calculation is given below:

```
1  let a = joinD                  -- scanlPS ends
2        . mapD (\(as,a) -> mapS (plusInt a) as)
3        . propagateD plusInt 0
4        . mapD (scanlS plusInt 0) -- scanlPS begins; fused
5        . sparseToDenseD (plusInt gmax 1) 0 -- sparseToDensePS ends
6        . splitSparseD (plusInt gmax 1)
7        . joinD                    -- sparseTodensePS begins
8        . mapD tripletToATup2      -- fused lambda and groupPS
9        . segdSplitMerge 0         -- workhorse of groupPS
10       . sortPS
11       . concatPS
12       $ img
```

One can firstly observe the occurrence of distributed functions (with a `-D` suffix). They operate either on each PU locally (as with `mapD`) or implement some specific inter-PU calculation (as does `propagateD`). These functions are the result of inlining various parallel functions and eliminating communication. The correspondence to their original functions is given as comments in the code. Only sortPS and concatPS are unchanged. A real compiler would have inlined their definitions and looked for optimisations.

Aside from them, inlining `scanlPS` and `sparseToDensePS` [2] exposes their internals. In-between both of the functions, there was a composition of the distribution primitives - namely

[2] It's definition is given in the appendix 8.4.

`splitD . joinD`. Applying the rewrite rule "splitD/joinD" eliminated a point of synchronisation.

This leaves `propagateD` as the only inter-PU communication in-between the splitting of the sparse array and joining the histogram at the end (line 1).

The expression `mapP (λg -> (headP g,lengthP g)) . groupPS` was involved in a rather special fusion. Essentially, the lambda expression was was applied to the result of groupPS. This enabled further communication fusion and created the local operation `tripletToATup2` [3] . It creates the local chunks of the sparse-array directly. `segdSplitMerge` does the actual work of the distributed grouping[4].

Normalisation and Scaling

The new code for normalisation and scaling is given below:

```
1  let n = lengthPS a
2      gs = joinD . mapD f . splitD $ a
3      f = (\gmax' divisor a0 x ->
4           floorDoubleS
5            (multDoubleS
6             (divDoubleS
7              (minusDoubleS
8               (int2DoubleS x)
9               a0)
10             divisor)
11            gmax')
12          )
13          $ ( replD n . int2Double $ gmax )
14          $ ( replD n
15             . minusDouble (int2Double (lastPS a))
16             . int2Double . headPS $ a )
17          $ ( replD n . int2Double . headPS $ a )
```

The definitions of the lifted functions are similar to these two examples:

```
1 floorDoubleL = joinD . mapD floorDoubleS . splitD
2 multDoubleL as = joinD . zipWithD multDoubleS (splitD as) . splitD
```

Inlining these functions created *five* pairs of `splitD . joinD` - which were then immediately eliminated using the "splitD/joinD" rule.

[3] `tripletToATup2 :: LinkedList (Int,Int,Int) -> PA (Int,Int)`.
[4] as explained in chapter 5

After that, a cascade of rewrite rules fired and propagated the normalisation and scaling constants into local computation. A sophisticated combination of the rules "mapD/zipWithD", "splitD/replPS", "mapD/replD" and "ZipReplSplit" resulted in the code given above.

Operationally, there is an important change in the normalisation and scaling. Although, constants (like `int2Double $ gmax`) are still being replicated into arrays before applying the arithmetic mapping - now the replication is only limited to the local PU. This is different than before, when the constants were replicated *globally* and subsequently split.

In terms of speed, communication fusion eliminated *six* points of synchronisation. It greatly reduced constant factors in its complexity by pushing replications from global redistributions into local PU operations. Stream Fusion further improves this.

6.1.3 Stream Fusioning

Stream Fusioning is the final step of optimisation. Applying it improves the normalisation and scaling only. The new code is given below: [5]

```
let a0  = int2Double . headPS $ a
    divisor = minusDouble (int2Double (lastPS a))
              . int2Double . headPS $ a
    gmax' = int2Double $ gmax
    normScale = floorDouble
                . (flip multDouble) gmax'
                . (flip divDouble) divisor
                . (flip minusDouble) a0
                . int2Double
    gs = joinD . mapD (mapS normScale) . splitD $ a
```

The replications have been removed entirely. Constants are calculated first and then used in `normScale` to apply the arithmetic calculation. The arithmetic functions also have been merged together into single function `normScale`. This function is now applied element-wise on each value in each of the local chunks of the entire histogram array.

The code is the result of inlining the local sequential functions (like `multDoubleS`) and subsequent stream fusion. For example, `multDoubleS` and `floorDoubleS` are defined as:

```
floorDoubleS :: Vector Double -> Vector Int
floorDoubleS = unstream . mapSt floorDouble . stream

multDoubleS :: Vector Double -> Vector Double -> Vector Double
multDoubleS as = unstream . zipWithSt multDouble (stream as) . stream
```

[5] `flip` were not actually inserted by the compiler. However, the nesting of functions is easier to display using `flip`s.

Inlining these definitions creates expressions of `unstream . stream`. Applying the "unstream/stream" rule and a few other rules analogous to Communication Fusioning finally propagates the constants outside of any traversals.

The transformation is over now. The next section will give an overview of the results.

6.2 Final Program

Summing up the steps, the compiler gives the following optimised code for P_{vect}:

```
type Image = PA (PA Int)
type Hist = PA Int

hbalance :: Image -> Image
hbalance img =
let a :: Hist
    a = joinD
        . mapD (\(as,a) -> mapS (plusInt a) as)
        . propagateD plusInt 0
        . mapD (scanlS plusInt 0)
        . sparseToDenseD (plusInt gmax 1) 0
        . splitSparseD (plusInt gmax 1)
        . joinD
        . mapD tripletToATup2
        . segdSplitMerge 0
        . sortPS
        . concatPS
        $ img
    n :: Int
    n = lengthPS a

    a0, divisor, gmax' :: Double
    a0    = int2Double . headPS $ a
    divisor = minusDouble (int2Double (lastPS a))
              . int2Double . headPS $ a
    gmax' = int2Double gmax

    normScale :: Int -> Int
    normScale = floorDouble
                . (flip multDouble) gmax'
                . (flip divDouble) divisor
                . (flip minusDouble) a
                . int2Double
```

```
34
35      gs :: Hist
36      gs = joinD . mapD (mapS normScale) . splitD $ a
37
38  in unconcatPS img
39       . indexPL (expandPS img gs)
40       . concatPS
41       $ img
```

On the surface, the algorithm works quite similar to a direct implementation of P_{nest}. First the histogram is calculated (lines 18 to 11) and accumulated (lines 10 - 7). Then the constants `a0`,`divisor` and `gmax'` are calculated globally and distributed to each PU (lines 22 to 33). After that, each PU applies the normalisation and scaling transformations (line 36). The mapping array `gs` is then finally used to map each gray tone to its new value (lines 41 to 38). The gray tone mapping is an example of the flattening of nested parallel functions in NDP.

All in all, P_{vect} offers a few advantages over P_{nest}:

- a decreased number of communication and synchronisation points
- flat data structures and flat operations further decrease constant factors
- Inlining and optimisation fused normalisation and scaling together - even though they were separated in P_{nest}. The programmer did not need to fuse them manually. This stands in contrast to the situation in P_{man}.
- After writing P_{nest}, there is no more work involved for the programmer in generating this optimised code.

Having transformed P_{nest} to P_{vect}, one is now ready to give a complexity analysis thereof.

6.3 Complexities

The complexity analysis for work and depths remains similar to that of P_{nest}. After all, both are the same algorithm. P_{vect} is only better at it's constant factors. Let n be the number of pixels in the image, then its complexity is given by: [6] [7]

$$\begin{aligned}
\mathrm{W}(w \times h, gmax) &= \mathrm{W}(hist) + \mathrm{W}(accu) + \mathrm{W}(gs) + \mathrm{W}(img') \\
&\in O((n \log n + gmax) + gmax + gmax + n) \\
&= O(n \log n + gmax) \\
\mathrm{D}(w \times h, gmax) &= \mathrm{D}(hist) + \mathrm{D}(accu) + \mathrm{D}(gs) + \mathrm{D}(img') \\
&\in O(\log n + \log gmax + 1 + 1) \\
&= O(\log n + \log gmax)
\end{aligned}$$

The use of the plentiful new functions does not change the overall situation compared to P_{nest}. P_{vect} has the same work and depth complexities as before. Being at $O(n \log n + gmax)$ in work the histogram calculation remains the most expensive when executed on a single processor. With increasing number of processors, the various logarithmic depth operations in the (accumulated) histogram calculation become the bottleneck. They require time logarithmic in the number of gray tones `gmax` and the number of pixels `n`. A overview of all the functions an their complexities can be found in the table 6.1.

The implementation and analysis of the programs P_{seq}, P_{man}, P_{nest} and P_{vect} have been given. The next chapter with the discussion of the results.

[6] The calculation of `a` is re-split into `hist` and `accu`.

[7] The reader might ask, why `propagateD` has higher depth than work - and why work is simply 1. That is, because the value increases logarithmically with the number of PUs. Since in 'work' only a single processor is used, there is no propagation of values and the time is constant.

Table 6.1: Complexities for P_{vect}

function or variable	W $\in O(...)$	D $\in O(...)$
hbalance	$n \log n + gmax$	$\log n + \log gmax$
hist	$n \log n + gmax$	$\log n$
concatPS	1	1
sortPS	$n \log n$	$\log n$
segdSplitMerge	$n \log n$	$\log n$
mapD tripletToATup2	gmax	1
joinD xs	gmax	1
accu	gmax	$\log gmax$
sparseToDenseD	gmax	1
splitSparseD	gmax	1
mapD scanlS	gmax	1
propagateD	1	$\log gmax$
mapD (mapS plusInt)	gmax	1
joinD	gmax	1
as	gmax	1
splitD	gmax	1
mapD normScale	gmax	1
joinD	gmax	1
expandPS	n	1
img'	n	1
concatPS	1	1
indexPL	n	1
unconcatPS	1	1

7 Results and Discussion

"Vertrauen ist gut, Kontrolle ist besser."

Redewendung

This chapter summarises results of the four implementations P_{seq}, P_{man}, P_{nest} and P_{vect}. First the complexities of the programs are given and discussed. Then, the pros and contras of each program are presented. They compare them on different aspects. Finally, the chapter ends with a conclusion on Nested Data Parallelism.

7.1 Complexity Analysis

The complexities of all four programs are summarised in table 7.1. Note that in most image processing applications $gmax < n$ holds. The number of pixels in an image is usually greater than the number of gray tones each pixel has available.

Table 7.1: Complexities for P_{seq}, P_{man}, P_{nest} and P_{vect}

program	W $\in O(...)$	D $\in O(...)$
P_{seq}	$n \log gmax + gmax$	$n \log gmax + gmax$
P_{man}	$n \cdot gmax$	$\log n$
P_{nest}	$n \log n + gmax$	$\log n + \log gmax$
P_{vect}	$n \log n + gmax$	$\log n + \log gmax$

A few observations can be made.

P_{nest} versus P_{vect}: P_{nest} and P_{vect} have equal work and depth complexities. Compiler optimisations can rarely optimise into a lower complexity class. However, the use of flat data structures, the optimisation for cache locality and the fusion of communication and loops greatly reduces the constants factors of P_{vect}.

P_{man} versus P_{vect}: Work and depth complexities of P_{man} and P_{vect} are competing. While the work of P_{man} is a product of the parameters n and $gmax$ - the work of P_{vect} (and P_{nest}) is only a sum. For larger parameters, and a limited number of processors P_{vect} will out-compete P_{man}. P_{vect} has a speed-up linear in $gmax$.

Regarding depth however, the opposite is the case. P_{man} grows logarithmic to n whereas P_{vect} grows logarithmic to n and $gmax$. For large values of $gmax$ and a high number of processors, P_{man} will out-compete P_{vect}. However, the speed-up is only a summand of $\log gmax$.

Sequential versus Parallel: Out of all programs, P_{seq} has the best bounds of work. This is due to its iteration-based histogram creation. Parallel programs cannot use this method because they then were to fall back to sequential traversal. P_{man} and P_{nest} instead used more advanced methods to implement parallel histogram creation.

On one hand, their methods have worse work complexities. If $gmax$ is treated as a constant, then P_{seq} grows linearly - whereas P_{man} and P_{vect} grow $O(n \log n)$. On the other hand, they greatly improve in depth complexities. While P_{seq} remains linear to n, P_{man} and P_{vect} are only logarithmic to n.

Parallel Programs have an overhead when the number of PUs is low. With an increasing number of PUs, they out-compete P_{seq}.

Parameter configuration: Different parametrization in n and $gmax$ leads to different programs being better and worse and others. As stated in paragraph 7.1, for large parameters and a limited number of PUs, P_{vect} performs better than P_{man}. However, for large $gmax$ and a high number of PUs the situation is reversed. Depending on the application context, different parameters $gmax$ and n are given. This makes some algorithms faster than the others.

7.2 Pro and Contra

Table 7.2 gives an extensive pro and contra analysis of the four programs. They compare the programs on their similarity to A_C, running time with variation of the number of PUs, amount of human workload involved and more.

Table 7.2: Pros and Contras

program	pro/contra
P_{seq}	+ It is a direct implementation of A_C. + It requires the least amount of human work. - It has semi-high constant factors due to pointer-based nesting. - It is purely sequential and cannot gain from parallelism.
P_{man}	+ It achieves time logarithmic to n for large number of PUs. + It has low constant factors. + It has a short implementation. +/- The programmer trades flexibility and comfort for fine grained control. - The work is product of n and $gmax$. It grows faster than all other complexities. - It is only a surface translation of A_C. - Normalisation, scaling and histogram creation were fused together manually. They are not separated steps anymore. - Much human work was necessary to parallelize the histogram creation. - Subsequent algorithms have to be coded to operate on flat images. [1]
P_{nest}	+ It achieves time logarithmic to n and $gmax$ for large number of PUs. + It is almost a direct implementation of A_C. + It implements parallel histogram creation using only high level function compositions. + It involves only limited workload for the human. - It has high constant factors (compared to P_{man} and P_{vect}) due to nesting and unoptimised communication and traversals.
P_{vect}	+ The programmer does not need to think about the flat representation of the image. + Flattening of arrays and nested functions reduce constant factors. + Communication Fusion and Stream Fusion reduce constant factors and automatically fuse normalisation and scaling. + The compiler optimises automatically. - If failed, compiler optimisations are difficult to guide. +/- The programmer trades fine grained control for flexibility and comfort. - Real code produced by the compiler is mostly incomprehensible for humans.

7.3 Conclusion

Given the prior analysis, a few conclusions on parallel functional programming with NDP can be drawn. Using Nested Data Parallelism, high-level flexible parallel programs can be written and efficiently executed. In NDP much work and burden is taken from the programmer to the compiler. Solving the same problem using conventional parallel programming methods requires more time and thought, but can lead to better performance and more control on the execution. For small numbers of PUs, sequential programming is faster and simpler than parallel programming. However, for large number of PUs parallel programs perform better.

All in all, the four programs, P_{seq}, P_{man}, P_{nest} and P_{vect}, have different strengths and weaknesses. No program is strictly better in all aspects than any other. However, the use of Nested Data Parallelism enabled a sufficiently high-level implementation with only modest performance drawbacks as compared to manual parallelization.

[1] Unless one wraps P_{man} with (un-)flattening operations. This approach however manually implements the flattening approach used in NDP.

8 Outlook

"I claim to be a simple individual
liable to err like any other fellow mortal.
I own, however, that I have humility
enough to confess my errors
and to retrace my steps.

Mahatma Gandhi

8.1 Summary

This thesis gave an evaluation of Nested Data Parallelism in the functional programming language Haskell by giving four implementations of Histogram Balancing and comparing them to each other. They were P_{seq}, P_{man}, P_{nest} and P_{vect}. The results suggest that no program was clearly better than all other programs. Each of them had its advantages and disadvantages. While P_{seq} performs the best on a few number of processors, it is out-competed by parallelized programs when the number rises. P_{man} has the lowest complexity for large number of processors - though it has the highest complexity for small numbers. However, P_{vect} is a good compromise between both extremes. It enables almost as much abstraction as does P_{seq} while still performing well on varying number of processors.

This leads to the conclusion, that Nested Data Parallelism is a very welcome programming model in functional programming.

8.2 Related Work

There is much active research in parallel programming in general and in the programming language Haskell.

Generally, there are two approaches on parallel programming. In the following a summary of these approaches is given. The comments after the semi-cola state properties of problems for which the approach is fruitful.

- Task Parallelism; irregular, non-deterministic
  - dependence graph-based: Streit et al. (2015), deterministic, medium-abstraction
  - futures-based: Cohen et al. (2012), simple integration, subtle-preconditions
  - thread-based: Butenhof (1997), high-control, low-abstraction
  - In Haskell: Marlow (2012)
    - Algorithm + Strategy = Parallelism: Trinder et al. (1998); task-based, deterministic, subtle-preconditions
    - The Par Monad; Marlow et al. (2011); dependency graph, in-concise, high-performing
    - STM: Harris et al. (2005); transactional, non-det. IO, safe, composable
- Data Parallelism: Hillis and Steele (1986); (ir-)regular, deterministic
  - Flat GPU Parallelism: CUDA Nickolls et al. (2008), OpenCL Stone et al. (2010); highly regular and high-performing, deterministic,
  - In Haskell: Marlow (2012)
    - Repa: Keller et al. (2010); nested arrays, regular, data-parallel, high-performing
    - Accelerate: McDonell et al. (2013); GPU, high-performing, data-parallel, regular
    - Nested Data Parallel: Jones (2008); type-dependent, irregular/flexible, concise

All related work on parallelism in Haskell (except for Marlow et al. (2011)) implement sophisticated optimisations strategies like NDP. In functional programming, these optimisations are more necessary than an option. Unoptimised functional programs are usually slower by a magnitude than their imperative counterparts.

8.3 Future Work

Based on the results on this thesis, further work can be of interest:

Brent's Equation

The work and depth complexities calculated in this thesis are mere complexitiy classes. Given more time and details on the implementation of the NDP functions, one could give an exact expression for the work and depth of each program. One could then use Brent's Equation Gustafson (2011):

$$\frac{W}{P} \leq T \leq \frac{W}{P} + L \cdot D$$

where P is the number of processors, L is the communication latency in number of machine instructions, and W and D are work and depth in number of instructions, respectively.

Brent's Equation gives upper and lower bounds on the time a program with work W and depth D requires given a latency of L and P processors.

If given exact numbers for work and depth (of table 7.1), one could do a more fruitful and exact analysis in the parameter-space spanned by n, $gmax$, P and L. For concrete parameters one could solve for the optimal algorithm to use.

Alternate Algorithms

Given the flexibility of NDP, alternate algorithms and data structures can be considered. The original papers Jones (2008) use the 'Banes-Hut-Algorithm' for their examples. It is an $O(n \log n)$ algorithm for the approximate of calculation of n-bodies moving in three-dimensional space under the force of gravity. A comparison with manual implementation is a relevant question - since the algorithm is known to be hard to parallelize. For image processing, one could similarly consider other algorithms such as 'Connected-Components-Labelling' or 'Face-Detection'.

Distributed NDP

The implementation of NDP in Haskell is much more general than its original implementation in NESL. Considering the explicit distinction of locality (e.g the use of `Dist`) used, there might be only limited work necessary to expand NDP from working on a single machine with many processors to working on multiple machines with multiple processors.

8.4 Final words

With this paragraph, the thesis comes to an end. The thesis gave a short evaluation of Nested Data Parallelism in Haskell for implementations of Histogram Balancing. The author thanks greatfully for your attention.

Appendix

SparseToDensePS

```
sparseToDensePS :: Int -> a -> PA (Int,a) -> PA a
sparseToDensePS size z ps =
 joinD
 . sparseToDenseD size z
 . splitSparseD size
 $ ps

splitSparseD :: Int -> PA (Int,a) -> Dist (PA (Int,a))
sparseToDenseD :: Int -> a -> Dist (PA (Int,a)) -> Dist (PA a)
```

SparseToDensePS converts a sparse array of index-value-pairs to a dense array, where the elements are inserted in the appropriate indices. Unspecified indices are given the default value `z`.

The function operates by first splitting/distributing the sparse array to the various PUs. It does that in such a way, that if all PUs were to hold a chunk of an array of lenght `size`, then each PU would get those index-value-pairs for which it would be responsible for on the dense array. This approach enables the second step to be purely local. The second step converts each local chunk of the sparse array to its corresponding local chunk of the dense array.

Further analysis reveals complexities $\mathrm{W}(z, ps) \in O(z + length(ps))$ and $\mathrm{D}(z, ps) \in O(1)$.

Bibliography

Batcher, K. E. (1968). Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA. ACM.

Blelloch, G. E. (1996). Programming parallel algorithms. *Commun. ACM*, 39(3):85–97.

Blelloch, G. E., Hardwick, J. C., Chatterjee, S., Sipelstein, J., and Zagha, M. (1993). Implementation of a portable nested data-parallel language. *SIGPLAN Not.*, 28(7):102–111.

Butenhof, D. R. (1997). *Programming with POSIX threads*. Addison-Wesley.

Chakravarty, M. and Keller, G. (2003). An approach to fast arrays in haskell. In Jeuring, J. and Jones, S., editors, *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 27–58. Springer Berlin Heidelberg.

Chakravarty, M. M. T. and Keller, G. (2000). More types for nested data parallel programming. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 94–105, New York, NY, USA. ACM.

Chakravarty, M. M. T. and Keller, G. (2001). Functional array fusion. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, volume 36 of *ICFP '01*, pages 205–216, New York, NY, USA. ACM.

Chakravarty, M. M. T., Keller, G., Jones, S. P., and Marlow, S. (2005). Associated types with class. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 1–13, New York, NY, USA. ACM.

Chakravarty, M. M. T., Leshchinskiy, R., Jones, S. P., Keller, G., and Marlow, S. (2007). Data parallel haskell: A status report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 10–18, New York, NY, USA. ACM.

Cohen, A., Gérard, L., and Pouzet, M. (2012). Programming parallelism with futures in lustre. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 197–206, New York, NY, USA. ACM.

Cole, R. (1988). Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms*, pages 65+. MIT Press and McGraw-Hill, second edition.

Coutts, D., Leshchinskiy, R., and Stewart, D. (2007). Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, volume 42 of *ICFP '07*, pages 315–326, New York, NY, USA. ACM.

Denning, P. J. (2005). The locality principle. *Commun. ACM*, 48(7):19–24.

Gustafson, J. (2011). Brent's theorem. In Padua, D., editor, *Encyclopedia of Parallel Computing*, pages 182–185. Springer US.

Harris, T., Marlow, S., Jones, S. P., and Herlihy, M. (2005). Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 48–60, New York, NY, USA. ACM.

Hillis, W. D. and Steele, G. L. (1986). Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183.

Jones, S. P. (2008). Harnessing the multicores: Nested data parallelism in haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, page 138, Berlin, Heidelberg. Springer-Verlag.

Jones, S. P. and Marlow, S. (2002). Secrets of the glasgow haskell compiler inliner. *J. Funct. Program.*, 12(5):393–434.

Keller, G. and Chakravarty, M. (1999). On the distributed implementation of aggregate data structures by program transformation. In Rolim, J., Mueller, F., Zomaya, A., Ercal, F., Olariu, S., Ravindran, B., Gustafsson, J., Takada, H., Olsson, R., Kale, L., Beckman, P., Haines, M., ElGindy, H., Caromel, D., Chaumette, S., Fox, G., Pan, Y., Li, K., Yang, T., Chiola, G., Conte, G., Mancini, L. V., Méry, D., Sanders, B., Bhatt, D., and Prasanna, V., editors, *Parallel and Distributed Processing*, volume 1586 of *Lecture Notes in Computer Science*, pages 108–122. Springer Berlin Heidelberg.

Keller, G., Chakravarty, M. M. T., Leshchinskiy, R., Jones, S. P., and Lippmeier, B. (2010). Regular, shape-polymorphic, parallel arrays in haskell. *SIGPLAN Not.*, 45(9):261–272.

Ladner, R. E. and Fischer, M. J. (1980). Parallel prefix computation. *J. ACM*, 27(4):831–838.

Leshchinskiy, R. (2005). *Higher order nested data parallelism: semantics and implementation.* PhD thesis, Berlin Institute of Technology.

Leshchinskiy, R., Chakravarty, M. M. T., and Keller, G. (2002). Costing nested array codes. *Parallel Process. Lett.*, 12(02):249–266.

Leshchinskiy, R., Chakravarty, M. M. T., and Keller, G. (2006). Higher order flattening. In *Proceedings of the 6th International Conference on Computational Science - Volume Part II*, ICCS'06, pages 920–928, Berlin, Heidelberg. Springer-Verlag.

Lippmeier, B., Chakravarty, M. M. T., Keller, G., Leshchinskiy, R., and Jones, S. P. (2012). Work efficient higher-order vectorisation. *SIGPLAN Not.*, 47(9):259–270.

Mainland, G., Leshchinskiy, R., and Jones, S. P. (2013). Exploiting vector instructions with generalized stream FusioN. *SIGPLAN Not.*, 48(9):37–48.

Marlow, S. (2012). Parallel and concurrent programming in haskell. In *Proceedings of the 4th Summer School Conference on Central European Functional Programming School*, CEFP'11, pages 339–401, Berlin, Heidelberg. Springer-Verlag.

Marlow, S., Newton, R., and Jones, S. P. (2011). A monad for deterministic parallelism. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 71–82, New York, NY, USA. ACM.

Masters, B. R. (1999). The image processing handbook. *Journal of Microscopy*, 196(1):79–80.

McDonell, T. L., Chakravarty, M. M. T., Keller, G., and Lippmeier, B. (2013). Optimising purely functional GPU programs. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 49–60, New York, NY, USA. ACM.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable parallel programming with CUDA. *Queue*, 6(2):40–53.

Peyton Jones, S., Tolmach, A., and Hoare, T. (2001). Playing by the rules: Rewriting as a practical optimization technique in GHC. In *Proceedings of the 2001 Haskell Workshop*, pages 203–233.

Stone, J. E., Gohara, D., and Shi, G. (2010). OpenCL: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73.

Streit, K., Doerfert, J., Hammacher, C., Zeller, A., and Hack, S. (2015). Generalized task parallelism. *ACM Trans. Archit. Code Optim.*, 12(1).

Trinder, P. W., Hammond, K., Loidl, H. W., and Jones, S. L. P. (1998). Algorithm + strategy = parallelism. *J. Funct. Program.*, 8(1):23–60.

Glossary

`Vector` is the Haskell implementation of conventional local in-memory arrays. They are used for the distributed chunks of `Dist (PA a)` , page 14

Cache Locality Use of contiguous-memory data over pointer-based data to create an execution behaviour that simplifies CPUs cache hierarchies to optimise for and greatly increases performance , page 9

Chunk An local array (or linked list) that is part of a globally distributed array (or linked list) , page 7

Constant Factors Constant factors increasing the running time of a program that are hidden inside Landau-Notation $O(\cdot)$. They are important for decreasing practical running time. , page 43

Flat Data Parallelism Expression of parallelism via parallel primitives over flat arrays. The primitives cannot be nested , page 7

Fusion The use of Inlining and semantic-preserving Rewrite Rules to find and eliminate communication (Communication Fusion) or intermediate data structures (Stream Fusion) , page 8

Nested Data Parallelism Expression of parallelism via parallel primitives over nested arrays. The primitives can be nested. It flattens the code before execution , page 7

PU A processing unit. Refers mostly to CPUs on a single machine. Can also be CPUs in a cluster or in a distributed setting , page 7

Vectorization A compiler transformation which flattens and uses Fusion a nested data parallel program to increase cache locality, decrease communication and traversals for higher performance , page 8

Work and Depth Language-based measures for the time necessary if given one processor (work) or infinitely many (depth) in number of instructions , page 16

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 23. Juli 2015 Chandrakant Swaneet Kumar Sahoo