Connected component tree construction for embedded systems

Petr Matas

University of West Bohemia
Faculty of Electrical Engineering
and
Université Paris-Est
École Doctorale MSTIC

THESIS

to obtain the Doctor of Philosophy degree of the University of West Bohemia
and Université Paris-Est with specialization in Computer Science

Connected Component Tree Construction for Embedded Systems

Petr Matas

presented on 30 June 2014

Composition of the Examination Committee:

Serge WEBER	Professor, University of Lorraine	Reviewer
Petr MATULA	Assoc. professor, Masaryk University Brno	Reviewer
Mohamed AKIL	Professor, ESIEE Paris	Director
Vjačeslav GEORGIEV	Assoc. professor, University of West Bohemia	Director
Václav MATOUŠEK	Professor, University of West Bohemia	**Chairman**
Eva DOKLÁDALOVÁ	Assoc. professor, ESIEE Paris	Examinator

Abstract

The aim of this work is to enable construction of embedded digital image processing systems, which are both flexible and powerful. The thesis proposal explores the possibility of using an image representation called *connected component tree* (CCT) as the basis for implementation of the entire image processing chain. This is possible, because the CCT is both simple and general, as CCT-based implementations of operators spanning from filtering to segmentation and recognition exist. A typical CCT-based image processing chain consists of CCT construction from an input image, a cascade of CCT transformations, which implement the individual operators, and image restitution, which generates the output image from the modified CCT. The most time-demanding step is the CCT construction and this work focuses on it.

It introduces the CCT and its possible representations in computer memory, shows some of its applications and analyzes existing CCT construction algorithms. A new parallel CCT construction algorithm producing the *parent point tree* representation of the CCT is proposed. The algorithm is suitable for an embedded system implementation due to its low memory requirements. The algorithm consists of many building and merging tasks. A building task constructs the CCT of a single image line, which is treated as a one-dimensional signal. Merging tasks fuse the CCTs together. Three different task scheduling strategies are developed and evaluated. Performance of the algorithm is evaluated on multiple parallel computers. A throughput 83 Mpx/s at speedup 13.3 is achieved on a 16-core machine with Opteron 885 CPUs.

Next, the new algorithm is further adapted for hardware implementation and implemented as a new parallel hardware architecture. The architecture contains 16 basic blocks, each dedicated to processing of an image partition and consisting of execution units and memory. A special interconnection switch is designed to allow some executions units to access memory in other basic blocks. The algorithm requires this for the final merging of the CCTs constructed by different basic blocks together. The architecture is implemented in VHDL and its functional simulation shows performance 145 Mpx/s at clock frequency 120 MHz.

Keywords

connected component tree, parent point tree, construction, graph, attributes, building, merging, parallel, concurrent, embedded, algorithm, scheduling, image processing, hardware, VHDL, FPGA

Résumé

L'objectif du travail présenté dans cette thèse est de proposer un avancement dans la construction des systèmes embarqués de traitement d'images numériques, flexibles et puissants. La proposition est d'explorer l'utilisation d'une représentation d'image particulière appelée « arbre des composantes connexes » (connected component tree – CCT) en tant que base pour la mise en œuvre de l'ensemble de la chaîne de traitement d'image. Cela est possible parce que la représentation par CCT est à la fois formelle et générale. De plus, les opérateurs déjà existants et basés sur CCT recouvrent tous les domaines de traitement d'image : du filtrage de base, passant par la segmentation jusqu'à la reconnaissance des objets. Une chaîne de traitement basée sur la représentation d'image par CCT est typiquement composée d'une cascade de transformations de CCT où chaque transformation représente un opérateur individuel. A la fin, une restitution d'image pour visualiser les résultats est nécessaire. Dans cette chaîne typique, c'est la construction du CCT qui représente la tâche nécessitant le plus de temps de calcul et de ressources matérielles. C'est pour cette raison que ce travail se concentre sur la problématique de la construction rapide de CCT.

Dans ce manuscrit, nous introduisons le CCT et ses représentations possibles dans la mémoire de l'ordinateur. Nous présentons une partie de ses applications et analysons les algorithmes existants de sa construction. Par la suite, nous proposons un nouvel algorithme de construction parallèle de CCT qui produit le « parent point tree » représentation de CCT. L'algorithme est conçu pour les systèmes embarqués, ainsi notre effort vise la minimisation de la mémoire occupée. L'algorithme en lui-même se compose d'un grand nombre de tâches de la « construction » et de la « fusion ». Une tâche de construction construit le CCT d'une seule ligne d'image, donc d'un signal à une dimension. Les tâches de fusion construisent progressivement le CCT de l'ensemble. Pour optimiser la gestion des ressources de calcul, trois différentes stratégies d'ordonnancement des tâches sont développées et évaluées. Également, les performances des implantations de l'algorithme sont évaluées sur plusieurs ordinateurs parallèles. Un débit de 83 Mpx/s pour une accélération de 13,3 est réalisé sur une machine 16-core avec Opteron 885 processeurs.

Les résultats obtenus nous ont encouragés pour procéder à une mise en œuvre d'une nouvelle implantation matérielle parallèle de l'algorithme. L'architecture proposée contient 16 blocs de base, chacun dédié à la transformation d'une partie de l'image et comprenant des unités de calcul et la mémoire. Un système spécial d'interconnexions est conçu pour permettre à certaines unités de calcul d'accéder à la mémoire partagée dans d'autres blocs de base. Ceci est nécessaire pour la fusion des CCT partiels. L'architecture a été implantée en VHDL et sa simulation fonctionnelle permet d'estimer une performance de 145 Mpx/s à fréquence d'horloge de 120 MHz.

Mots-clés

arbre des composantes connexes, parent point tree, construction, graphe, attributs, fusion, calculs parallèles, système embarqué, algorithme, programmation, traitement d'image, VHDL, FPGA

Anotace

Cílem této práce je umožnit konstrukci vestavěných systémů pro zpracování digitalizovaného obrazu, které jsou zároveň flexibilní a výkonné. Zkoumá se možnost použití reprezentace snímku zvané *strom souvislých komponent* (connected component tree, CCT) jako základu pro implementaci celého řetězce pro zpracování obrazu. Toto je možné, protože CCT je zároveň jednoduchý i obecný. Existují totiž na CCT založené implementace operátorů od filtrování až po segmentaci a rozpoznávání. Typický řetězec zpracování obrazu založený na CCT sestává z konstrukce CCT ze vstupního snímku, kaskády transformací CCT, které implementují jednotlivé operátory, a restituce obrazu, která generuje výstupní snímek z modifikovaného CCT. Časově nejnáročnějším krokem je konstrukce CCT a tato práce se na ni zaměřuje.

Práce představuje CCT a jeho možné reprezentace v počítačové paměti, ukazuje některé jeho aplikace a analyzuje existující algoritmy konstrukce CCT. Je navržen nový paralelní algoritmus konstrukce CCT, jehož výstupem je reprezentace CCT zvaná *parent point tree*. Tento algoritmus je vhodný k implementaci ve vestavěných systémech díky malým paměťovým nárokům. Algoritmus se skládá z mnoha úloh stavění a slučování. Z jednoho řádku snímku, se kterým se zachází jako s jednorozměrným signálem, stavění vytvoří CCT a slučování spojují tyto CCT dohromady. Tři různé strategie plánování úloh jsou vyvinuty a zhodnoceny. Výkonnost algoritmu je otestována na několika paralelních počítačích. Na 16jádrovém stroji s procesory Opteron 885 je dosaženo propustnosti 83 Mpx/s při 13,3násobném zrychlení paralelizací.

Následně je algoritmus dále adaptován pro hardwarovou implementaci a implementován jako nová paralelní hardwarová architektura. Ta obsahuje 16 základních bloků, z nichž každý zpracovává část snímku a skládá se z výkonných jednotek a pamětí. Je navržen speciální propojovací přepínač, aby některé výkonné jednotky mohly přistupovat k paměti v ostatních základních blocích. Algoritmus toto vyžaduje pro závěrečné slučování CCT vytvořených různými základními bloky dohromady. Architektura je implementována ve VHDL a její funkční simulace dává výkonnost 145 Mpx/s při frekvenci hodin 120 MHz.

Klíčová slova

strom souvislých komponent, parent point tree, konstrukce, graf, atributy, stavění, slučování, paralelní, konkurentní, zapouzdřený, vestavěný, algoritmus, plánování, zpracování obrazu, hardware, VHDL, FPGA

Contents

List of figures

List of tables

List of algorithms

Acknowledgements

In the first place I want to thank **God**, who accompanied me from the start and showed me His mercy and unconditional love, even though I did not want to know Him yet. He brought to my life many people, which made completing this work possible:

My biggest thanks go to my thesis directors and consultants for their continuous guidance and help, i.e. to **Vjačeslav Georgiev**, who did not let me run away from the unfinished work, **Mohamed Akil**, who enabled me to see things from another points of view, **Eva Dokládalová**, who pushed me straight forward and taught me how research is done, and **Martin Poupa**, who opened the FPGA domain to me.

I appreciate the kind acceptance and support from the **A3SI laboratory members**. I owe a lot also to other people who were helping me and supporting me during my work: **Harold Phelippeau**, **Václav Valenta**, **Jan Bartovský**, **Imran Taj**, **Ramzi Mahmoudi**, **David Hnilica**, **Petr Dokládal**, **David Menotti**, **Milan Dlauhý**, and others.

Thanks are also addressed to **Joëlle Delers** and **Samuel** from **acc&ss Paris-Est** (formerly BiCi – Bureau international des Chercheurs invités) for making my installation in France easy and fun. Additionally, my work would not have been possible without the generous financial support from the **French government**.

The access to computing and storage facilities owned by parties and projects contributing to the **MetaCentrum National Grid Infrastructure** [MetaCentrum], provided under the program "Projects of Large Infrastructure for Research, Development, and Innovations" (LM2010005) is also acknowledged.

Special thanks belong to my **parents** and **extended family members**, who accompanied me on the dark parts of the path. Substantial strength to go ahead originated also from the **members of the Christian meetings of university students in Pilsen**.

Abbreviations and symbols

1D, 2D, ...	1-dimensional, 2-dimensional, ...
C	Connected component
CCT	Connected component tree
D	Set of possible image sample values – image codomain
E	Pixel neighborhood relation and mapping
f	Image function
fps	Frames per second
G	Graph representation of an image
G	Number of grey levels of an image, $G = \|D\|$
H	Height of an image in pixels
HT	Hyper-Threading, the Intel's implementation of SMT
IPW	Inter-pixel watershed
$\log x$	Logarithm of x (base is not specified)
LSB	Least significant bit
$\max S$	Maximum of the elements of the set S
$\min S$	Minimum of the elements of the set S
Mpx/s	Megapixel per second
MSB	Most significant bit
N	Number of pixels of an image, $N = \|V\|$. For a 2D image, $N = W \cdot H$
$\mathbb{N}_0$	Set of non-negative integers $\{0, 1, 2, ...\}$
$\mathbb{N}_1$	Set of positive integers $\{1, 2, 3, ...\}$
O	Big O notation. $f(x) \in O(g(x))$ means that up to a multiplicative constant, $f(x)$ does not grow asymptotically faster than $g(x)$ as x goes to infinity, i.e. $\exists c \in \mathbb{R}, x_0 \in \mathbb{R} : \forall x > x_0 : \|f(x)\| \leq c \cdot \|g(x)\|$. For example, $2x \in O(x)$ and $x \in O(x^2)$
Θ	Big Theta notation. $f(x) \in \Theta(g(x))$ means that up to a multiplicative constant, $f(x)$ grows asymptotically as fast as $g(x)$ as x goes to infinity, i.e. $f(x) \in O(g(x)) \wedge g(x) \in O(f(x))$
px	Pixel, picture element, one sample of an iconic image of any dimension (1D, 2D, 3D, ...)
$\mathbb{R}$	Set of real numbers
$\mathbb{R}_+$	Set of positive real numbers
SMT	Simultaneous multithreading, a technique allowing multiple hardware threads per processor core
TW	Topological watershed
V	Set of points of an image – image domain
W	Width of an image in pixels
$\mathbb{Z}$	Set of integers

$P \wedge Q$	Logical conjunction of two predicates (P **and** Q)
$P \vee Q$	Logical disjunction of two predicates (P **or** Q)
$P \Rightarrow Q$	Implication between two predicates (**if** P, **then** Q)
$P \Leftrightarrow Q$	Equivalence of two predicates (P **precisely if** Q)
$x \in S$	x is an element of the set S
$x \notin S$	x is not contained in the set S
(a, b, c)	Ordered n-tuple given as a list of its elements
$\{a, b, c\}$	Set given as a list of its elements
$\{x : P\}$	Set of all elements x fulfilling the predicate P
$\{x \in S : P(x)\}$	Shorthand for $\{x : x \in S \wedge P(x)\}$, i.e. the set of all elements x from the set S fulfilling the predicate $P(x)$
$\emptyset$	Empty set
$A \cap B$	Intersection of the sets A and B
$A \cup B$	Union of the sets A and B
$\bigcup_{i \in S} A_i$	Union of all sets A_i, where $i \in S$
$\sum_{i \in S} x_i$	Sum of all values x_i, where $i \in S$
$A \setminus B$	Asymmetric difference of the sets A and B, i.e. the set $\{x \in A : x \notin B\}$
$A \subset B$	A is the subset of B, i.e. every element from A is contained also in B. Equality of the sets A and B is allowed
$A \subseteq B$	A is the subset of B. Equality of the two sets is explicitly allowed
$A \subsetneq B$	A is the subset of B, but they are not equal
$\|x\|$	Absolute value of the number x
$\|S\|$	Number of elements in the set S
$[S]$	The unique element of the single-element set S; undefined if $\|S\| \neq 1$
$\lfloor x \rfloor$	Floor function, i.e. the biggest integer not greater than x
2^S	Power set, i.e. the set of all subsets, of the set S
$\forall x : P(x)$	For all x, the predicate $P(x)$ is true
$\exists x : P(x)$	For at least one x, the predicate $P(x)$ is true
$f : V \to D$	f is a function (mapping) from domain V to codomain D
$f(x)$	Value of the function f at point x
$A \times B$	Cartesian product of the sets A and B, i.e. the set $\{(x, y) : x \in A \wedge y \in B\}$
$\perp$	Nothing. For example, if C has no parent, then $Parent(C) = \perp$
$x \leftarrow y$	Assign value y to variable x
$x{+}{+}$	Increment variable x by one and return its previous value
$x{-}{-}$	Decrement variable x by one and return its previous value

Chapter 1

Introduction

1.1 Embedded systems for vision

Nowadays, image sensors are becoming omnipresent and a huge amount of data is available from them. However, human capacity to handle these data is quite limited, and therefore majority of them has to be discarded without ever being processed. This is the opportunity for computer vision systems, which may extract the small amount of useful information contained in the data and sometimes even make decisions based on it and take actions without human intervention.

The computer vision systems are asked to give high performance and be flexible for a large variety of existing or possible applications. As an example, imagine just a small video surveillance system composed of five video cameras with a full-HD resolution (1920×1080 px, 24 bit/px, 25 fps) around a busy crossroad. Each second, such system produces over 6 Gb (gigabits) of data, which should enter a complex processing. This justifies the performance requirements. Next, let us imagine some of the many questions, which we may want to pose to this system, and the actions it could take. Some of them are in Figure 1.1, which demonstrates the flexibility requirements.

One of the global problems of the vision system design is how to achieve both high performance and flexibility simultaneously. It is because, a) many image-processing methods exist and each of them suits a different (usually small) subset of the applications, and b) each application requires cascades of multiple image-processing operators to produce the result.

If the high performance is achieved by an optimization effort, which means a kind of system specialization, it will (by definition) limit its flexibility. For example, a device optimized for FFT (Fast Fourier Transform) and similar computations is predestined for frequency domain image filtering and for image compression, but non-linear image processing operators and image segmentation may be quite inefficient with it.

Questions to the vision system:

- How many vehicles are standing in each lane?
- What are the speeds of the moving vehicles?
- What are the vehicles' registration numbers?
- Where and how many pedestrians are present?
- Is there any damage on the public transport vehicles?
- Is the crossroad obstructed?
- Is a traffic collision taking place?
- Is a fallen person lying in the field of view?
- Is a fight, assault or pocket-picking taking place? Who are the offenders?
- Has a given person crossed the field of view in a given time span? Which way did he go?
- What is the state of the road surface?
- What is the weather?
- What animal species live at the site?

Actions of the vision system:

- Traffic lights timing
- Traffic reporting
- Fine a speeding driver
- Alert police about location of a stolen vehicle or searched person
- Public transport dispatch
- Emergency services dispatch

Figure 1.1: Questions to and actions of the example computer vision system

1.2 Motivation for graph-based formalism

The image-processing operators may be usually classified into one of the following two categories:

- Low-level (local processing) operators: pre-processing, filtering
- High-level (image analysis) operators: segmentation, pattern recognition, motion extraction...

Simultaneous optimization of algorithms from both categories is particularly difficult using classic approaches, because the two categories' requirements are quite different from each other. Image-processing algorithms based on connected component tree (CCT) seem to be very promising from this point of view. They allow bridging the gap between low- and high-level processing elements. They have been used for image filtering [Salembier 1998, Wilkinson 2008], as well as the image analysis: motion extraction [Salembier 1998], watershed segmentation [Couprie 2005, Najman 2006, Mattes 2000], pattern recognition in astronomical imaging [Berger 2007, Jalba 2004], microscopic image analysis [Cuisenaire 1999], video-surveillance applications [Piscaglia 1999], image registration [Mattes 1999], or data visualization [Chiang 2005].

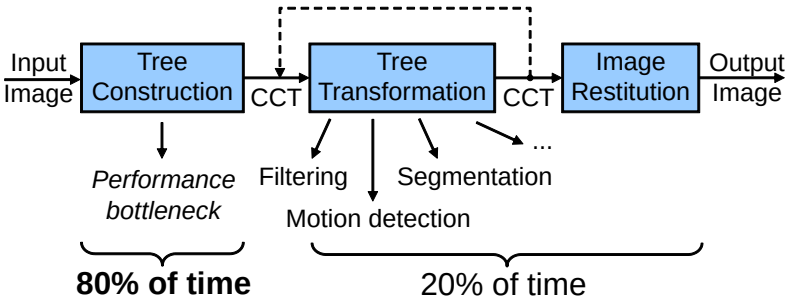


Figure 1.2: Stages of a typical CCT-based application with relative execution times

Another perspective for treating the flexibility problem are the data structures used for image analysis, which may be roughly classified into four levels going from the lowest to the highest level of abstraction [Sonka 2008]:

- **Iconic images** are matrices containing original data about pixel brightness or results of the image pre-processing.
- **Segmented images** join parts of the image into regions, which are likely to belong to the same objects or their borders.
- **Geometric representations** describe and quantify the shapes of the image regions.
- **Relational models** collect the information about relations among the objects in the image.

A flexible computer vision system will surely have to support data structures from all these four levels. The CCT is a unique data structure, which covers (at least partially) all of them: The iconic image is contained directly. Next, the CCT is composed of components – image regions – and the CCT can be therefore seen also as a segmented image. The component attributes allow characterization and quantification of many component properties including (but not limited to) the shape, which qualifies the CCT partly as a geometric representation. Finally, as components are organized into a hierarchic structure, the CCT is a simple relational model too.

Figure 1.2 shows typical stages of an application based on CCT. We can see another advantage of these methods: the processing is performed on the constructed CCT by graph transformation(s), and only one data structure is used from low-level to high-level processing.

On the other hand, the main bottleneck is the CCT construction, consuming about 80 % of the application execution time [Ngan 2011], which is penalizing for many practical applications. However, once the CCT has been constructed, the rest of the application executes quite fast. Therefore, this work focuses on acceleration of the CCT construction.

The key concept used by the CCT and the CCT-based algorithms is connectivity, i.e. existence of a contiguous path between two pixels of a given subset of the image domain. This is best described when we treat the image as a weighted graph, introduced in Section 2.1.1, which allows the CCT-based algorithms to be applied to images of any dimension (1D, 2D, 3D...) and connectivity. The CCT itself is a graph too, as tree is a special graph. The graph representation is also the most easily understandable form to imagine how

a structure works and what to do to make it working in our way. Note that many other components of intelligent vision systems are organized around the graphs (for example databases, knowledge bases and environment models). Graphs are widely used not only in image processing, but also in other disciplines such as digital signal processing and economy.

Chapter 2

Connected component tree and its applications

As stated in the introductory chapter, the connected component tree (CCT) is a representation of digital images, which allows efficient implementations of many image-processing operators from image filtering through watershed segmentation to data visualization.

This chapter introduces the graph-based digital image representation, explains the CCT, and presents its memory representations and some application examples.

2.1 Mathematical background

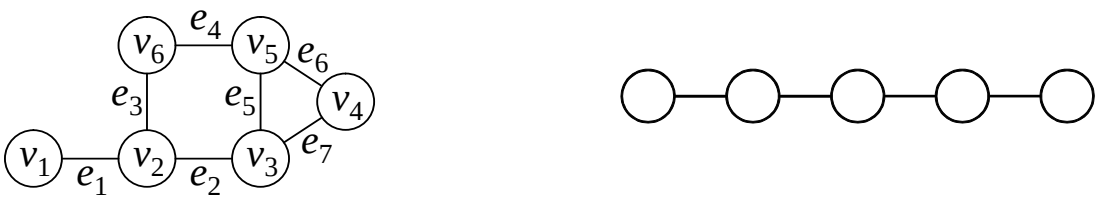
This section defines the mathematical terms that will be used later on. The majority of the definitions are stated in pairs:

The definition using intuitive wording and allowing fast reading comes first.

> The mathematically precise formal definition follows and is indented. Skipping it will not hamper general understanding, but this definition resolves any ambiguities, that the intuitive definition cannot easily avoid, and introduces the symbolic notation.

2.1.1 Digital image as a weighted graph

A *graph* is a mathematical representation of a set of objects where some of them are connected by links. The interconnected objects are called *vertices* and the links between them are called *edges*. Two examples are shown in Figure 2.1. Both are *undirected loopless graphs*. Such graphs will be used for digital image representation.



(a) A general graph. The set of vertices $V = \{v_1, \ldots, v_6\}$, the set of (undirected) edges $E = \{e_1, \ldots, e_7\}$

(b) A linear graph

Figure 2.1: Undirected loopless graph examples

The *undirected loopless graph* is a pair $G = (V, E)$, where

- V is a finite set of *vertices*,
- $E \subset V \times V$ is a binary *neighborhood relation* (the set of *edges*), which is
  - symmetric (ensures undirectedness): $\forall x, y : (x, y) \in E \Leftrightarrow (y, x) \in E$,
  - and anti-reflexive (ensures looplessness): $\forall x : (x, x) \notin E$

[Najman 2006, Wilkinson 2008].

An undirected edge notation $\{x, y\}$ may be used as a shortcut for the pair of edges (x, y) and (y, x).

The neighborhood relation E may be used also in the form of neighborhood mapping $E : V \to 2^V$, where $E(x) = \{y : (x, y) \in E\}$.

The graph is *connected* if there exists a *path* between any pair of its vertices. For example, both graphs in Figure 2.1 are connected.

The *path* from vertex $u \in V$ to vertex $v \in V$ is a vertex sequence $(x_0, \ldots, x_n)$, such that $n \geq 0$, $x_0 = u$, $x_n = v$, and $\forall i \in \{1, \ldots, n\} : (x_{i-1}, x_i) \in E$.

The *simple path* is a path, which does not repeat any vertices.

The graph is *connected* precisely if for every $u, v \in V$ there exists a path from u to v.

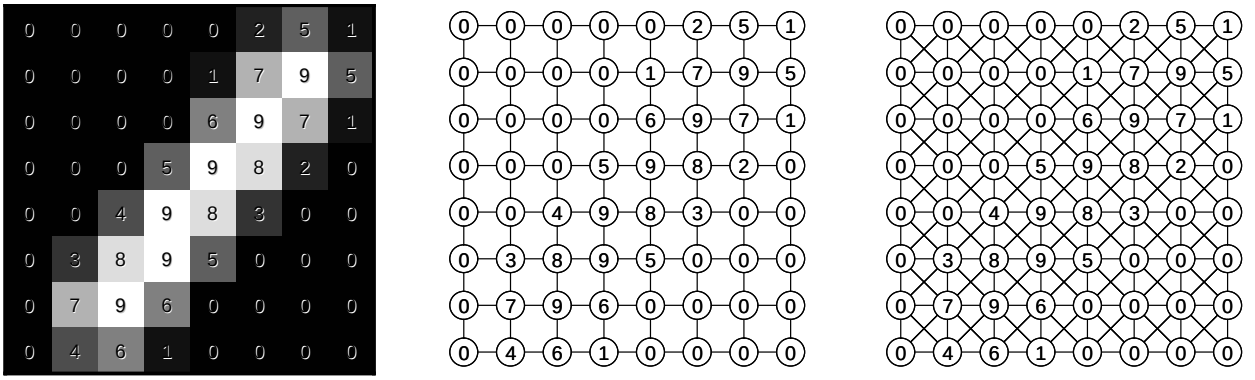
A set $X \subset V$ is connected in the (possibly not connected) graph $G = (V, E)$ precisely if the *subgraph* $G_X = (X, E \cap (X \times X))$ *induced by* X is connected [Wilkinson 2008].

A special connected graph type is a *linear graph* (Figure 2.1b) consisting of just a simple path. It will be used in the following chapters.

If a value is added to each vertex of the graph, a *vertex-weighted graph* is obtained.

The *vertex-weighted graph* is a triplet $G = (V, E, f)$, where (V, E) is a graph and $f : V \to D$ is a vertex weight function [Najman 2006]. D can be any totally ordered set, like the set of real numbers for example.

The **connected undirected loopless vertex-weighted graphs** can be used to represent digital (iconic) images [Najman 2006]: The graph vertices correspond to image sample (pixel, point) positions, the set of which (V) is called the *image domain* [Wilkinson 2008]. The set of edges E describes the neighborhood (or connectivity) relation among samples. The connectivity relation can be chosen arbitrarily and it does not



(a) An example image

(b) The graph representation of the image with 4-connectivity

(c) The graph representation of the image with 8-connectivity

Figure 2.2: Graph representation of an image. The values in the circles are graph vertex weights

depend on the actual image data (sample values). The sample values are given by the vertex weight function $f : V \to D$. The set of possible sample values D is called the *image codomain*. For grayscale images, $D \subset \mathbb{R}$. An example of such image representation is in Figure 2.2.

The advantage of representing the image with the graph is that it is both simple and general: Image processing algorithms, which treat the image as the graph, can be used for images of any dimension (1D, 2D, 3D...), with any connectivity, and for any sample grid arrangement (square, hexagonal, etc.).

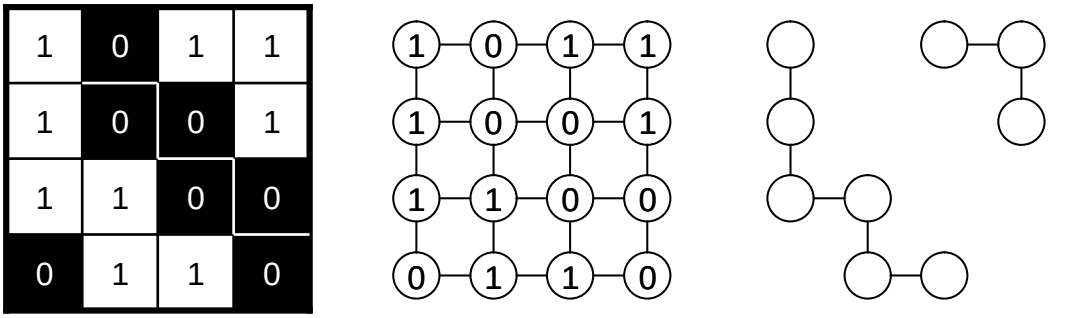
A binary image, which is a special case of the digital image with only two possible sample values, may be described in the following two ways, which are shown in Figure 2.3:

- as the weighted graph $G = (V, E, f)$ described above with the vertex weight function $f : V \to \{0, 1\}$. Vertices with weight 0 are called background pixels and vertices with weight 1 are called foreground pixels.
- as a non-weighted subgraph $G_X = (X, E_X)$ induced by the set of foreground pixels: $X = \{v \in V : f(v) = 1\}$, $E_X = E \cap (X \times X)$. Note that information about the background regions is dropped in this case.

2.1.2 Connected component tree (CCT)

A *connected component* C of the binary image is a maximal connected subset of its foreground pixels [Wilkinson 2008]. For example, the binary image shown in Figure 2.3 contains two connected components.

> The non-empty subset $C \subset X$ is a *connected component* of graph $G_X = (X, E_X)$ precisely if C is connected in G_X and there exists no vertex $x \in X \setminus C$ such that $C \cup \{x\}$ is also connected in G_X [Wilkinson 2008].



(a) An example binary image **(b)** The weighted graph **(c)** The non-weighted subgraph

Figure 2.3: Two possible descriptions of a binary image

The connected components of a grayscale image (also called *peak components* or *domes* [Wilkinson 2008]) are the connected components of all thresholdings of the image [Wilkinson 2008].

> The non-empty subset $C \subset V$ is a connected component of the image $G = (V, E, f)$ precisely if C is connected in G and there exists no vertex $x \in V \setminus C$ such that $f(x) \geq \min\{f(v) : v \in C\}$ and $C \cup \{x\}$ is also connected in G.
>
> The set of all connected components of the grayscale image is denoted by $CC \subset 2^V$.

The value $min(C) = \min\{f(v) : v \in C\}$ is called the minimum or level [Salembier 1998] or altitude [Najman 2006] of component C.

An inclusion relation organizes all connected components of the grayscale image into a rooted tree structure called the *connected component tree* (CCT), where the parent of component C_2 is the smallest component C_1 that is larger than C_2 and contains C_2. Figure 2.4 shows an example.

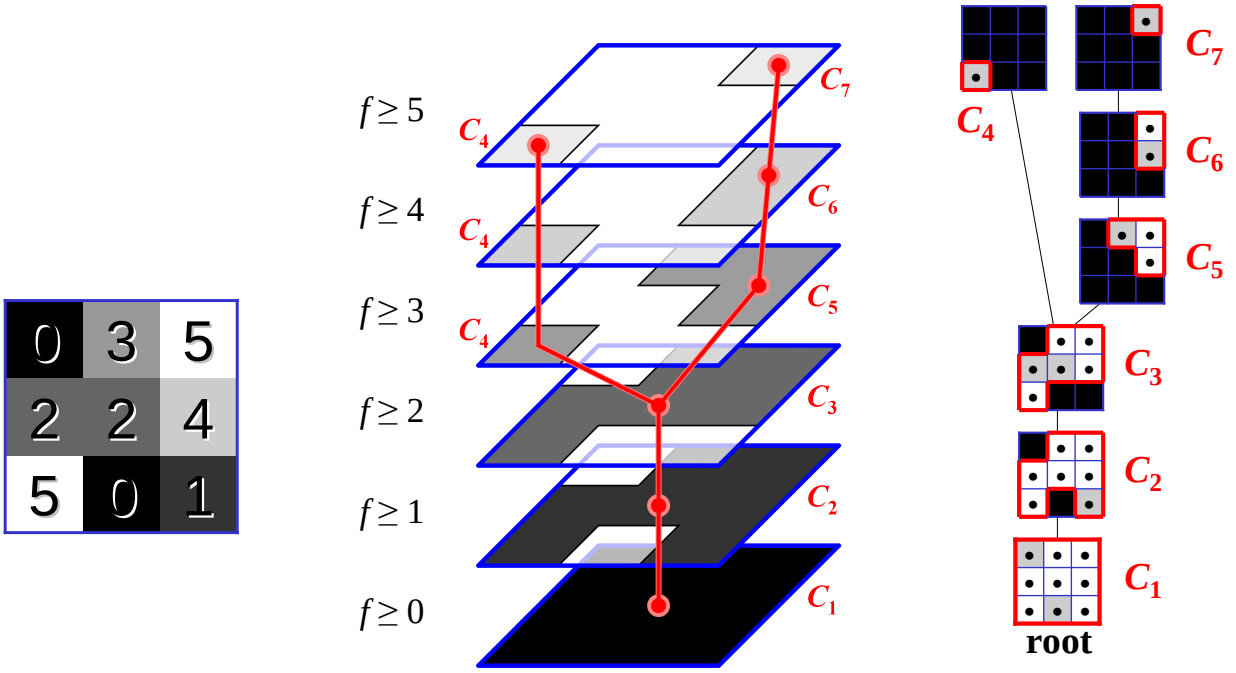
> The component C_1 is the parent of C_2 precisely if $C_2 \subsetneq C_1$ and there is no other component C_3 such that $C_2 \subsetneq C_3 \subsetneq C_1$. We write $C_1 = Parent(C_2)$. Parent of the root component is undefined.

We call the *core* C' of the component C the set of its pixels with level equal to the minimum of the component.

$$C' = \{v \in C : f(v) = min(C)\}.$$

It is obvious that the points of the component's core belong to none of the component's descendants and therefore each point of the image belongs to precisely one component core. The component is the union of its core and of the cores of all its descendants. All practical CCT representations for storage in computer memory exploit this property by storing only the cores of the components.

The leaves of the connected component tree described above contain local maxima of the image. The tree is therefore called a *max-tree*. By duality, if all comparisons concerning the weight function f are reversed and the minimum and maximum operators on f are swapped, a *min-tree* will be obtained whose leaves contain local minima. Unless stated otherwise, the following text concerns with the max-tree variant of the CCT, but the appropriate dual form of all statements applies to the min-tree as well.



(a) A grayscale image (pixel values taken from [Ngan 2007])

(b) Thresholdings of the image at all possible levels with connected components labeled

(c) The connected component tree. Component core pixels are gray

Figure 2.4: A connected component tree example

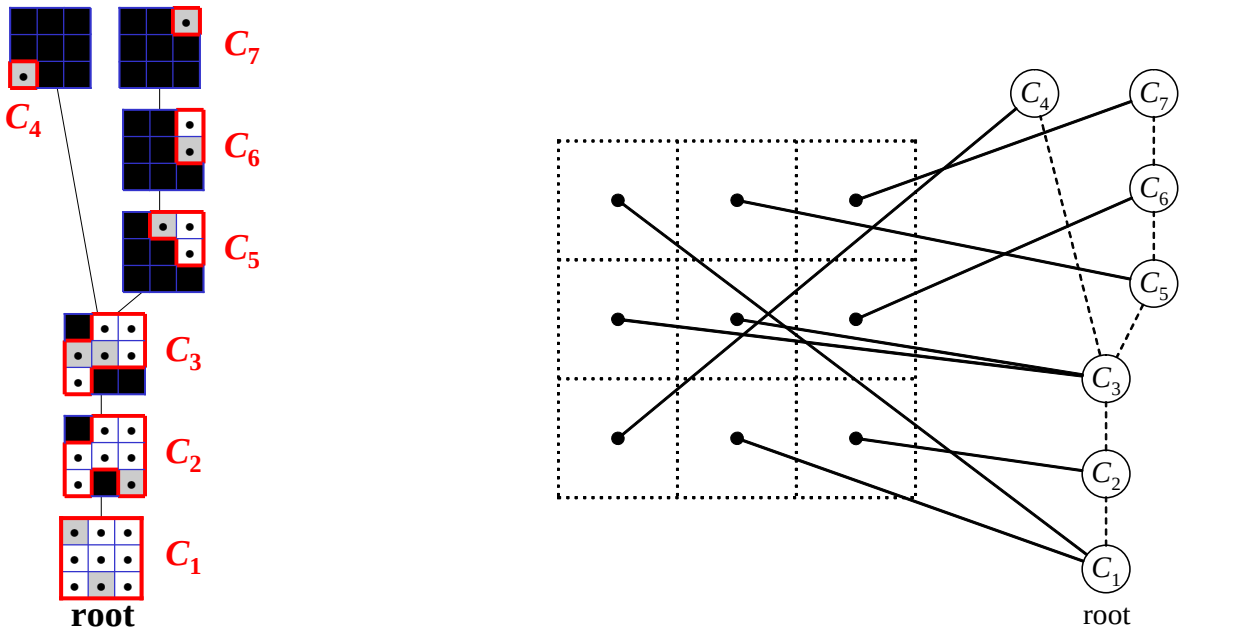
2.2 CCT representations

The information contained in the CCT consists of two parts:

- Parent/child relationships among the components of the CCT
- The shape of each component, that is the set of pixels contained in it

Figure 2.5 shows the classic and straightforward representation of the CCT. It is a tree, in which each node represents one connected component of the image, with parent/child relations among the nodes reflecting the parent/child relations among the components. The shapes of the components are represented by an injective relation between the tree nodes and the points of the image: Each tree node is in relation with the points of the corresponding component core. The inverse relation is called *component mapping* $M : V \to CC$ [Najman 2006]. This relation can be stored in two ways: Each tree node contains the list of the core's pixels and/or each pixel holds a reference to the corresponding tree node.

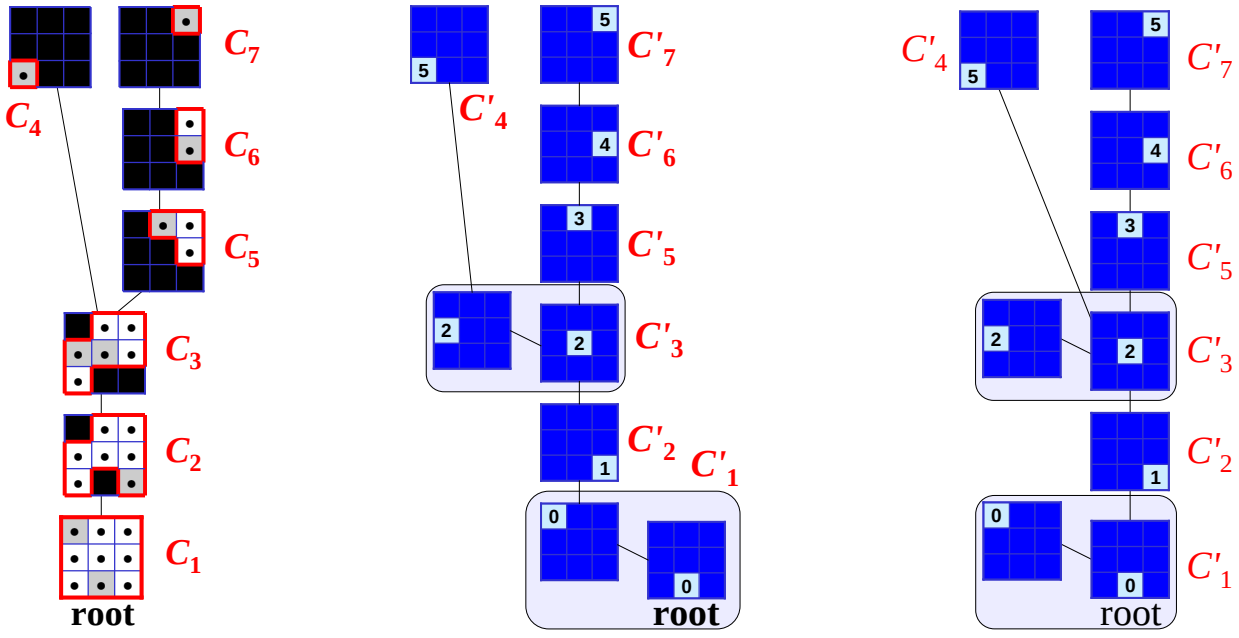
There is also another CCT representation called *point tree* [Meijster PhD, Wilkinson 2008] which is inspired by the data structure used in the Tarjan's union-find algorithm [Tarjan 1975]. It is shown in Figure 2.6b. As the name suggests, the pixels of the image themselves are organized into a tree, which is constructed in such a way that both the parent/child relationships among the components and the component shapes are retained:



(a) The CCT

(b) Its classic representation. Solid lines – the component mapping. Dashed lines – the parent/child relationships among the components

Figure 2.5: The classic representation of the CCT



(a) The CCT

(b) A general point tree

(c) The canonical point tree

Figure 2.6: The point tree. Each tree node in (b) and (c) shows the position of the pixel and its intensity value in the image. For the canonical point tree, the rule to select the level roots was "select the rightmost pixel of the bottommost pixels"

- Points of one component core form a tree whose shape is irrelevant and whose root is called the component's *level root*. The choice of the component's level root is also irrelevant.
- The level root's parent is an arbitrary point of the parent component's core. The level root of the root component is the root of the point tree.

The point tree definition allows some freedom and therefore a given CCT may be represented correctly by different point trees. Working with trees will be usually faster, if the average depth of the nodes is small. It is therefore good if parents of the tree nodes are level roots. If this is fulfilled for all nodes (except for the root which has no parent), then we say that the point tree is *perfectly compressed*.

A perfectly compressed point tree, in which the choice of the component level roots has been fixed using some deterministic rule (for example, "select the point with the highest address"), is unique for a given CCT. Such point tree is called *canonical* [Berger 2007] and it is shown in Figure 2.6c.

Both the classic and the point tree CCT-representations use a rooted tree. The parent/child relationships in this tree have to be stored as references from each tree node to its parent node and/or to its child nodes. This suggests that there are two possible directions of these references:

- If parents point to their children, it will be possible to browse the entire tree from a single starting node, the root node. However, the number of children of a given node is not known a priori. Therefore, the references have to be stored in dynamic lists, which are complex and require a significant amount of memory.
- If children point to their parents, it will be possible to access ancestor nodes of a given node. Because every node except the root has exactly one parent, only one reference per node is stored which is very memory-efficient.

The third, most memory-demanding possibility is to store the references in both directions, which will allow browsing the entire tree from any starting node.

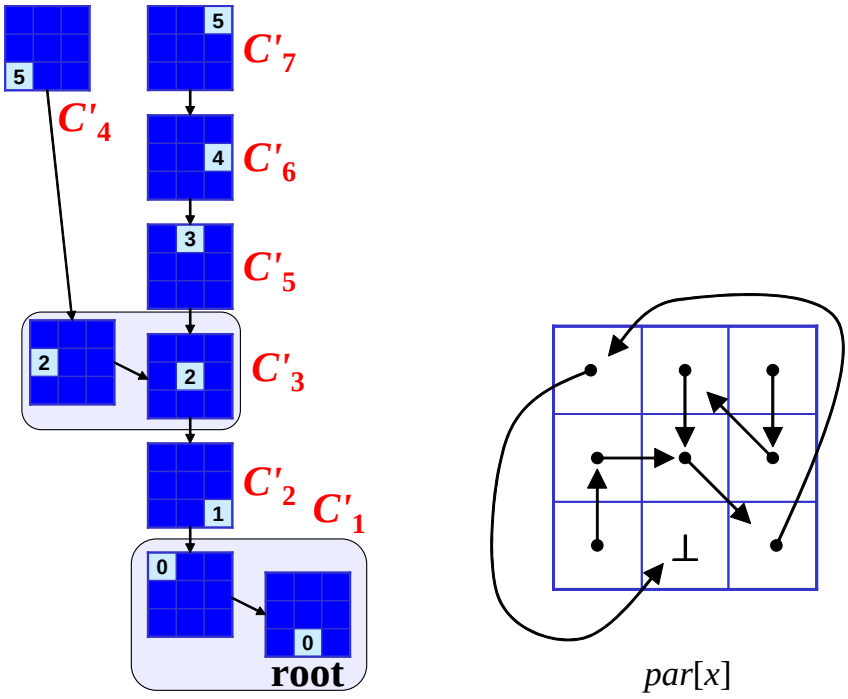
Any CCT representation described here may be converted to any other in linear time.

In this work, the CCT will be represented with the point tree with the references in the direction from children to their parents – a parent point tree that is used in many recent works [Meijster PhD, Berger 2007, Wilkinson 2008]. It is shown in Figure 2.7. It is the least memory demanding solution and it will allow an efficient implementation of tree merging, which will be necessary for parallelization of tree construction.

2.3 Attributes

The CCT on its own may be sufficient for some image transformations like watershed segmentation. Other operations on the image, especially filtering, require some aggregate knowledge about each component. This knowledge is expressed by component attributes. These are some numeric values characterizing each component.

Let us consider a connected component $C \subset V$ of an image $G = (V, E, f)$. The following list mentions many simple attributes of the component C and gives a precise formal definition for majority of them. Beware however, that literature often presents identically named attribute definitions, which are not equivalent.



(a) Parent point tree **(b)** Array of parent pointers

Figure 2.7: The parent point tree and its storage in the array of parent pointers

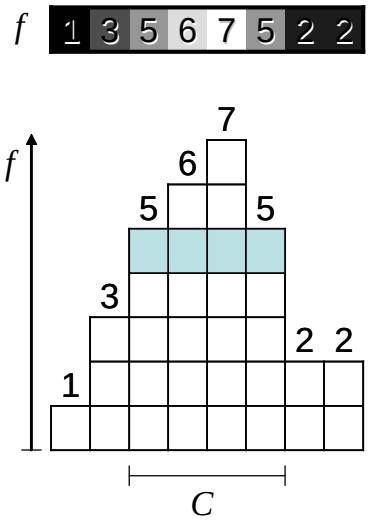
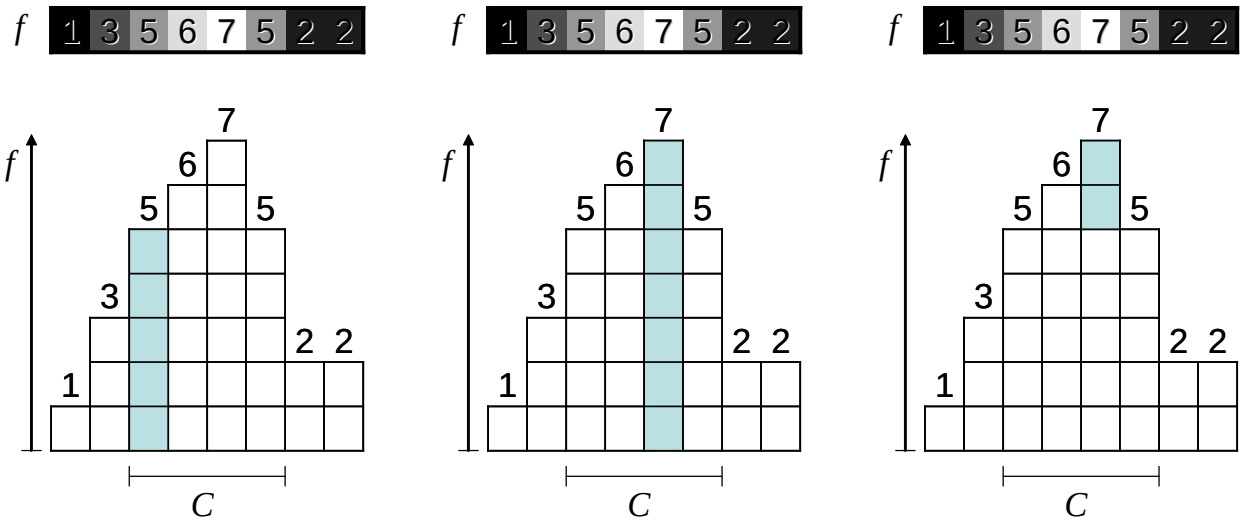


Figure 2.8: Illustration of the attribute Area on a 1D signal. $a(C) = |C| = 4$

- **Area:** The number of pixels of the component [Couprie 1997].
  $a(C) = |C|$
  This attribute is sometimes called "volume" in the context of 3D images [Wilkinson 2008], but it is different from the Volume definition used in this document.



(a) Minimum: $min(C) = 5$ **(b)** Maximum: $max(C) = 7$ **(c)** Range: $R_h(C) = 2$

Figure 2.9: Illustrations of attributes Minimum, Maximum, and Range

- **Minimum:** The minimum of the component's pixel intensities. This attribute is also called Level h [Salembier 1998, Wilkinson 2008] or Altitude [Najman 2006] in the literature.
  $min(C) = \min\{f(v) : v \in C\}$
- **Maximum** [Deloison 2007]: The maximum of the component's pixel intensities.
  $max(C) = \max\{f(v) : v \in C\}$
- **Range:** The difference between the maximum and minimum of the component. Sometimes it is also called Height [Couprie 1997]. This term will be avoided because it is sometimes used to refer to the Minimum attribute.
  $R_h(C) = max(C) - min(C)$
- **Parent's minimum:** The minimum of the parent component. The parent of the root component does not exist, so the value of this attribute has to be defined explicitly for the root component.
  $min_p(C) = min(Parent(C))$ if C is not the root component, $min_p(C) = min(C) - 1$ if C is the root component.
- **Level difference:** The difference of the minimum to the parent's minimum.
  $\Delta min(C) = min(C) - min_p(C)$
- **Core area:** The number of pixels of the component's core.
  $a_c(C) = |C'| = |\{v \in C : f(v) = min(C)\}|$
- **Area difference for a given branch** (inspired by [Sonka 2008]): The difference of the area of component C to the area of its son in the branch given by the component $C_l \subseteq C$.
  $\Delta a(C, C_l) = |C \setminus [\{C_s : C_s \text{ is a component of } G \wedge Parent(C_s) = C \wedge C_l \subseteq C_s\}]|$ if $C_l \subsetneq C$,
  $\Delta a(C, C_l) = |C|$ if $C_l = C$.

- **Area difference to the parent** (inspired by [Sonka 2008]): The difference of the area to the parent's area. Again, special treatment of the root component is needed.
  $\Delta a(C) = a(Parent(C)) - a(C)$ if C is not the root component, $\Delta a(C) = 0$ if C is the root component.

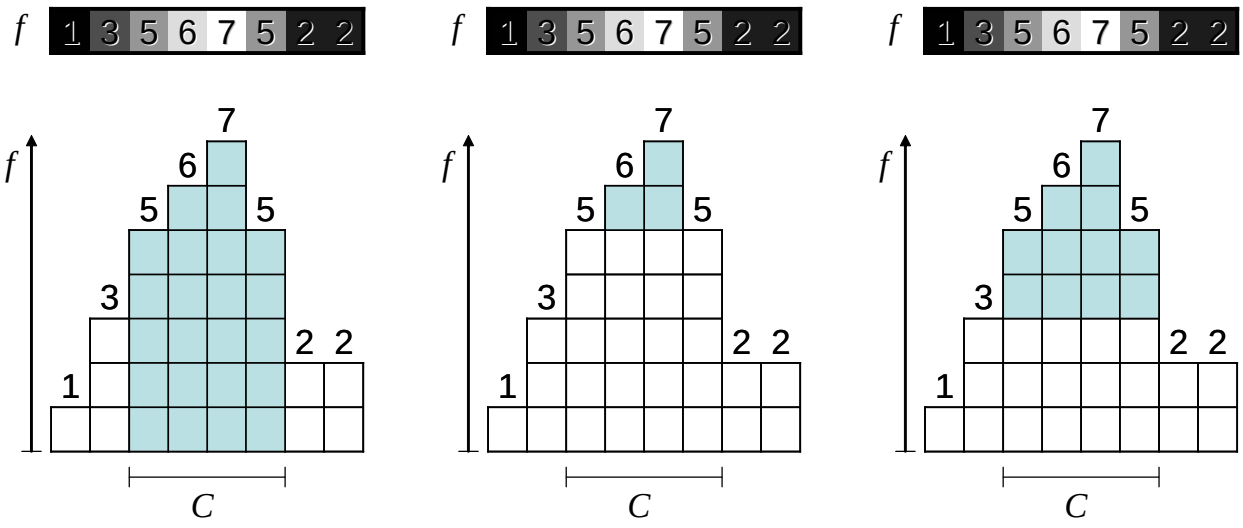
- **Raw volume:** The sum of the component's pixel intensities.
  $V_r(C) = \sum_{v \in C} f(v)$
  Note that the term "volume" is sometimes used to refer to the number of pixels of a 3D image component [Wilkinson 2008]. That attribute is called Area in this document.

- **Volume above the component prism:** The sum of differences of the component's pixel intensities to the component minimum. It is the "volume" attribute from [Couprie 1997].
  $V_d(C) = \sum_{v \in C}(f(v) - min(C)) = V_r(C) - |C| \cdot min(C)$

- **Proper volume:** The sum of differences of the component's pixel intensities to the parent's minimum.
  $V_p(C) = \sum_{v \in C}(f(v) - min_p(C)) = V_r(C) - |C| \cdot min_p(C)$

- **Perimeter** [Deloison 2007]: The number of edges between the component and its complement.
  $P(C) = |E \cap (C \times (V \setminus C))|$

- **Roundness** [Deloison 2007]: The ratio between the area and an appropriate power $k \in \mathbb{R}_+$ of the perimeter. For a d-dimensional image, the choice $k = d/(d-1)$ makes the attribute invariant to scaling. This attribute may be used as an assessment of **compactness** of the component.
  $R(C) = |C|/(P(C))^k$



(a) Raw volume: $V_r(C) = 23$ **(b)** Volume above: $V_d(C) = 3$ **(c)** Proper volume: $V_p(C) = 11$

Figure 2.10: Illustrations of three definitions of the attribute Volume

- **Moment of inertia** [Wilkinson 2001]: Another measure component's (non-)compactness. This one is based on pixel coordinates and it comes from physics.

  If the pixels are treated as points with zero size, the following formula applies:

  $I(C) = \sum_{v \in C} ||v - \bar{v}||^2$, where the points v are treated as positional vectors, $\bar{v} = \sum_{v \in C} v/|C|$ is the center of mass, and $||v||$ is the norm of the vector v.

  If the pixels are treated as squares or cubes with edge size equal to 1, then each pixel itself has some moment of inertia, which has to be added [Wilkinson 2001]:

  $I(C) = |C| \cdot d/12 + \sum_{v \in C} ||v - \bar{v}||^2$, where d is the dimension of the image.

- **Elongation** [Wilkinson 2008]: A ratio between the Moment of inertia and Area, which is invariant to scaling.

  $\phi_1(C) = I(C)/|C|^{(d+2)/d}$, where d is the dimension of the image.

- **Histogram entropy** [Deloison 2007]: A statistic on the histogram of the component's pixel intensities.

  $H(C) = -\sum_{h \in \mathbb{R}} p(h) \log p(h)$, where $p(h) = |\{v \in C : f(v) = h\}|/|C|$ is the percentage of the component's pixels having the intensity h, and $0 \cdot \log 0 = 0$.

- **Depth** [Deloison 2007]: The distance between the component and the root in the component tree. In other words, it is the number of the component's ancestors.

- **Movement** [Deloison 2007]: In a sequence of images, it is an assessment of how much the component has changed across different images of the sequence.

- **Contrast:** The standard deviation of the component's pixel intensities. This attribute is called "SNR" in [Deloison 2007].

- **Rate of change of the area with respect to the intensity** (inspired by [Sonka 2008]): This attribute indicates how fast the component's area changes with change of the threshold value. It is not defined for the root component.

  $a'(C) = \Delta a(C)/\Delta min(C)$

- **Number of holes** [Deloison 2007]: In the component's complement $V \backslash C$, it is the number of connected components with no pixels at the image border.

- **Orientation** (inspired by [Wilkinson 2008] and [Deloison 2007]): Another statistic based on pixel coordinates. This one indicates the angle or the directional vector of the direction, in which the component is elongated.

- **Gradient energy** [Deloison 2007]: The sum of the component's pixel gradient magnitudes.

Many of these attributes are defined as an accumulation of some value over the pixels of the component or they can be derived from one or more such attributes. These attributes can be computed efficiently during the construction of the CCT without drastic increase of the execution time. Some other attributes may be computed efficiently too, but not before the complete tree has been constructed. There are also some attributes, which cannot be implemented efficiently, but it is often possible to find other, efficiently implementable attributes, which provide similar information.

An important property of each attribute is whether it is monotonic, that is whether the value of the attribute changes monotonically from a leaf to the root of the tree: An attribute $A : 2^V \to \mathbb{R}$ is *increasing* precisely if in every image $G = (V, E, f)$, for every two components $C_1 \subseteq C_2 \subseteq V$, the inequality $A(C_1) \leq A(C_2)$ holds. An attribute fulfilling the reversed inequality is *decreasing*. An attribute is *monotonic* precisely if it is increasing or decreasing.

Here are some examples of increasing, decreasing, and non-monotonic attributes:

Increasing attributes:

- Area
- Maximum
- Range
- Volume
- Moment of inertia
- Gradient energy

Decreasing attributes:

- Minimum
- Depth

Non-monotonic attributes:

- Perimeter
- Roundness
- Elongation
- Histogram entropy
- Number of holes

2.4 Examples of applications

2.4.1 Attribute filters

Image filtering is a typical example of a low-level operator, which aims at removing of noise from the image or at emphasizing of important features in it. The CCT-based filters work on the connected component tree of the image and delete unwanted nodes or branches from it. The resulting output image is reconstructed from the modified tree. A big advantage of these filters is that they do not introduce any new contours to the image. This is because each contour corresponds to a connected component, no new components are created in the process and the shapes of the components are preserved. The shapes in the original image are therefore either preserved or removed, but never modified.

The decision whether to keep or not each connected component is based on some *criterion* built upon the attributes whose form is usually $A(C) \geq \lambda$, where $A : 2^V \to \mathbb{R}$ is some attribute and $\lambda \in \mathbb{R}$ is the *criterion threshold*. The *direct decision rule* is that the components, which fulfill the criterion, are kept, and those, which do not, are deleted.

This direct decision rule works well for criterions built upon increasing attributes, so called *increasing criterions*. In such criterions, if a given component is removed, then all of its descendants are removed too. The image restitution from such tree is straightforward.

On the other hand, if the criterion is not increasing, then the direct decision rule may leave some components without their parents, in which case the nearest kept ancestor of the component will become its new parent. We say that the direct decision rule does not *prune*. A question how to reconstruct the image from such tree arises. Intuition says that some properties of the kept components should be conserved. These properties are described by attributes and a choice which attribute should conserve its values has to be made.

The classical choice is to conserve the Minimum attribute, in which case the intensity of each pixel in the image is lowered to the Minimum of the smallest kept component, which contains the pixel. The resulting filter is called a "direct filter". It conserves well the absolute intensity levels in the image, but it produces steep jumps of intensity around the components, which are missing many ancestors in a row.

Conserving the Level difference attribute instead avoids this behavior. Such filter effectively subtracts the Level difference of each removed component from intensities of all image pixels belonging to the component. That is why the combination of the direct decision rule with the Level difference conservation is called a "subtractive filter".

A few more attributes can be chosen for conservation, like the Raw volume for example.

To avoid leaving some components without their parents, other, pruning decision rules may be used. The "Min" decision rule keeps the component if it and all its ancestors fulfill the criterion. The "Max" decision rule keeps the component if it or any of its descendants fulfill the criterion. The "Optimal" decision rule [Salembier 1998] prunes the tree at the places, which lead to the lowest number of kept nodes not fulfilling the criterion and vice versa. This rule is also able to work with criterions formulated as a certainty that a given node should be preserved instead of a strict true/false assessment. The filter variants are shown in Figure 2.11.

Other modifications of the method are possible. One example is taking the residue of the filtration (the difference between the input and filtered images) as the result. Another example is keeping of n most important tree branches in which case the criterion threshold λ is set automatically.

An example attribute filtering result taken from [Wilkinson 2008] is shown in Figure 2.12.

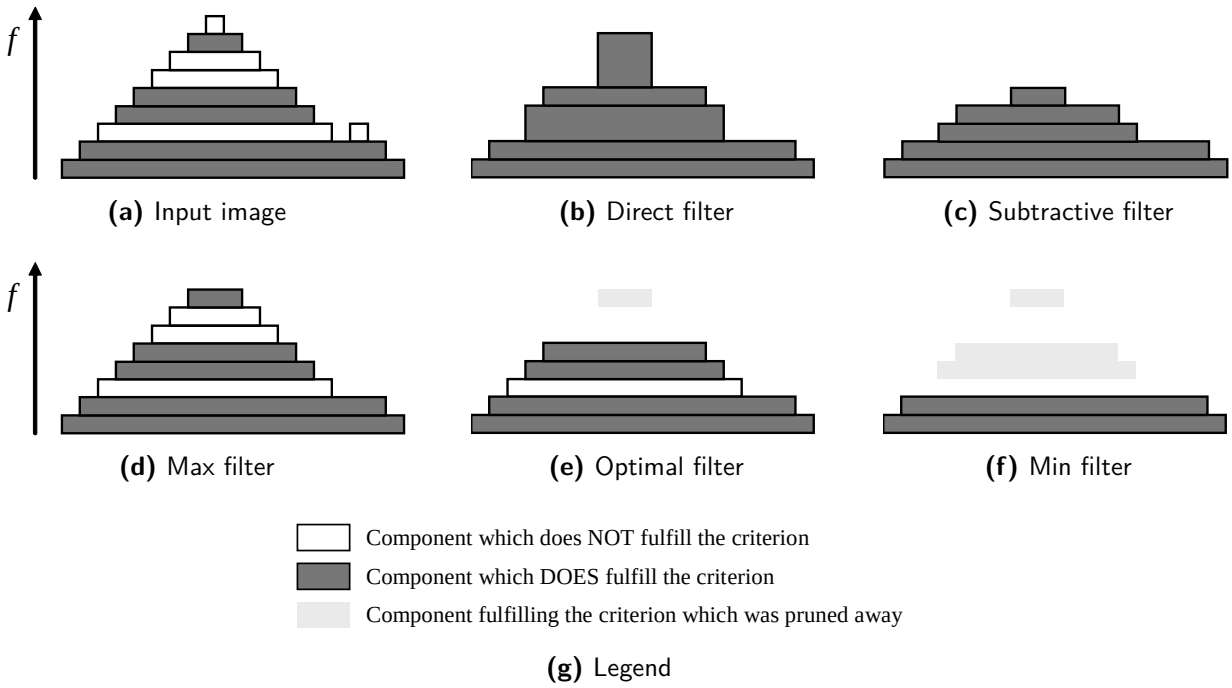
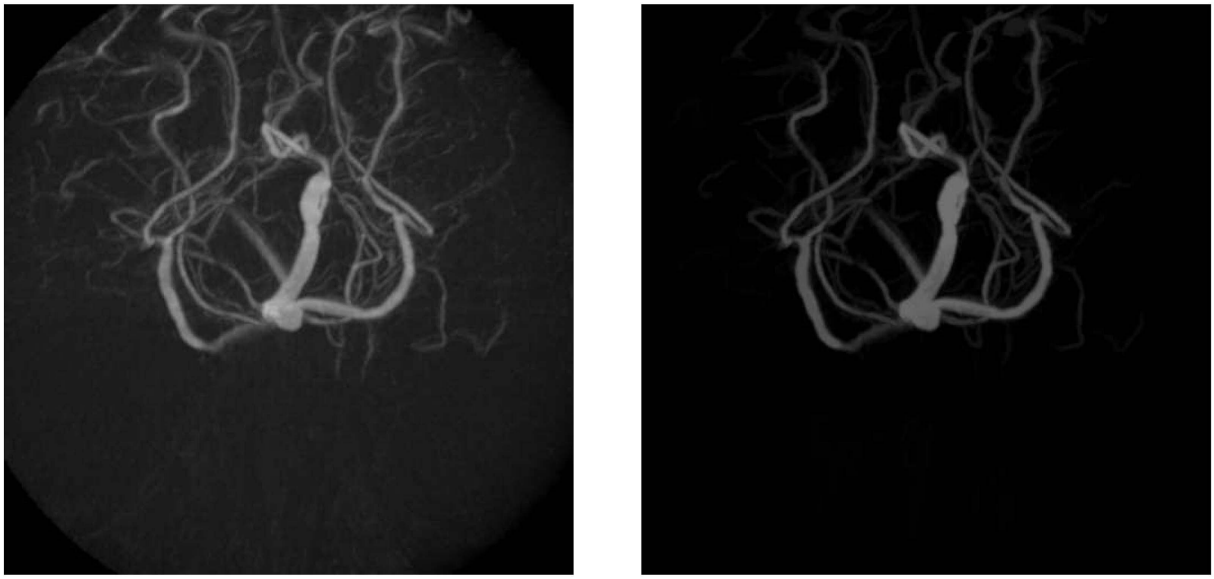


Figure 2.11: Different variants of attribute filters. (b, c) are non-pruning filters, (d, e, f) are pruning filters.



(a) Original image **(b)** Filtered image

Figure 2.12: An attribute filtering example. Maximum intensity projection of a 512^3px CT angiogram, and the same volume filtered using the direct filter with the elongation criterion $\phi_1(C) \geq \lambda$. Note the distinct suppression of the background while retaining the vessel structure [Wilkinson 2008]

2.4.2 Segmentation 1: Detection of maximally stable extremal regions

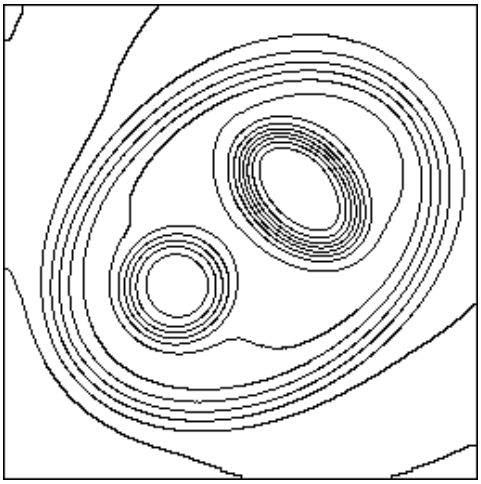
Image segmentation is a high-level image processing operation, which aims at separating of objects from the background and/or of the individual objects from each other and/or at detecting of the boundaries between them. Section 5.3.11 of [Sonka 2008] mentions a segmentation method, whose description follows. It segments the image directly from the CCT, although the author does not refer to the CCT explicitly. This method detects significant bright (or dark, by duality) connected regions, which may be nested and may not cover the image entirely.

We have seen that we can construct the CCT from all thresholdings of the image. Let us imagine a movie consisting of such thresholded images, running from the highest to the lowest threshold. We would see the components emerge from the background, gradually grow, sometimes two or more components would touch and merge into one component, and the movie would end with a single component covering the entire image.

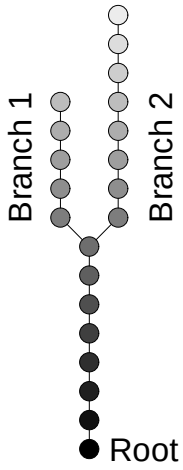
In some ranges of threshold values, there may be components, whose areas respond very slowly to the threshold change, that means slower than in surrounding ranges. Such components are called *maximally stable extremal regions* (MSERs). They are interesting, because each such component is significantly brighter than its surroundings and therefore it is likely to correspond to some bright object in the registered scene.

Figure 2.13 illustrates the principles of this method's implementation, which is quite straightforward: Figure 2.13a shows an input image, Figure 2.13b its level lines, and Figure 2.13c its connected component tree. Although Figures 2.13b and c show only 16 levels for simplicity, the actual CCT contains all 256 levels

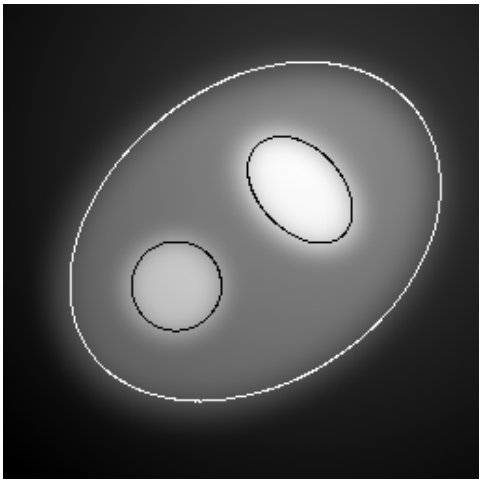
(a) A grayscale image



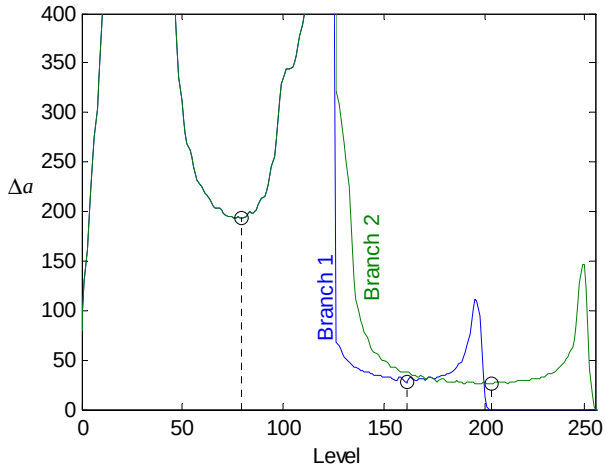
(b) Level lines



(c) CCT



(d) Maximally stable extremal regions



(e) Area difference function smoothed by a 6-tap weighted moving average. Significant local minima are marked

Figure 2.13: A maximally stable extremal regions example

present in the input image. Now for each leaf C_l of the CCT, we compute the area difference attributes $\Delta a(C, C_l)$ for all components C in the corresponding CCT branch as a function of level h:

$\Delta a(h, C_l) = \Delta a([\{C : C \text{ is a component of } G \wedge C_l \subseteq C \wedge min(C) = h\}], C_l)$ if such component C exists, $\Delta a(h, C_l) = 0$ otherwise.

Then we perform some smoothing of this function and we seek for its significant local minima in each branch (Figure 2.13e). The components corresponding to such minima are the desired MSERs, i.e. the result of the segmentation (Figure 2.13d).

The area difference functions are practically histograms of the image parts corresponding to the individual branches of the CCT. The segmentation using MSERs is therefore quite similar to multi-thresholding using histogram analysis. However, the MSERs method analyzes each part of the image separately, which is its advantage.

2.4.3 Segmentation 2: Topological watershed

In many cases, image segmentation is a difficult task and the segmentation methods usually need to be tuned for a given image using one or more parameters. The *watershed* segmentation is one of the few methods, which do not need them.

The watershed segmentation is applied to an image, in which boundaries between objects are expected to go through the locations with high pixel values f. It can be either an intensity image, if the areas separating the objects are bright, or a gradient image in which high pixel values correspond to rapid intensity changes often found at the object borders. The watershed segmentation partitions the image completely into disjoint regions and focuses on the borders between them. This contrasts to the MSERs segmentation, which focuses on individual significant regions of the image.

One of several existing watershed segmentation variants is the *topological watershed* (TW) [Couprie 2005, Neerbos 2011], which is intuitively defined as follows: Let us consider the image as an altitude map of a topographic surface with dark areas becoming valleys and bright areas becoming hills. Let us imagine that there is a hole in each local minimum and that the surface is slowly immersed in water. The water fills the basins and dams are built at points where waters coming from different local minima would meet. As a result, the image is partitioned into regions separated by dams called watershed lines.

Algorithm 2.1 gives the exact definition of the operator [Neerbos 2011]. It is valid for the image codomain $D = \mathbb{Z}$ (integers), but an extension valid for any image codomain is obvious.

Algorithm 2.1: Topological watershed definition

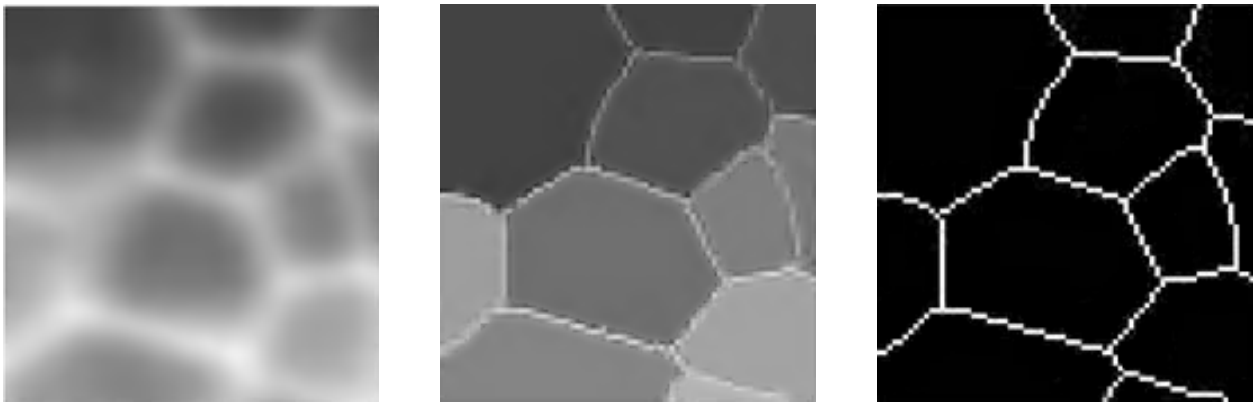
Input: Image domain V (set of image points)
Input: Pixel neighborhood relation $E \subset V \times V$
Input: Integral image function $f : V \to \mathbb{Z}$
Output: Modified f

```
while a destructible point exists in V do
    x ← any destructible point from V
    f(x) ← f(x) − 1
```

Note: Point x is *destructible* precisely if the operation $f(x) \leftarrow f(x) - 1$ does not change the connected components number at any level of the min-tree.

A naive implementation of this definition would yield an algorithm with a quite poor time complexity, so optimized algorithms are used in practice. Nevertheless, their outputs conform to this easily understandable definition too.

Note that the destructible point selection order may be arbitrary and therefore a given image may have many different TWs. Some of them may be even quite strange. For example, rectangular boundaries may



(a) Input image **(b)** Topological watershed **(c)** Watershed lines

Figure 2.14: Topological watershed results demonstration [Couprie 2005]

appear even though there is no support for them in the original image. Only additional requirements on the resulting TW may eliminate this ambiguity. It is clear, that every real implementation of the algorithm produces just one result. Which one of those many possible correct TWs it is, stays however implementation dependent.

Figure 2.14, taken from [Couprie 2005], demonstrates the method results. In the resulting image (b), the minimum altitude of a region is assigned to all pixels of the region, and the altitude of the entire dam is set to the minimum altitude k separating two regions. Thanks to this, the significance or "strength" of each watershed line can be determined easily. The value $(k+1)$ is called *connection value* in that article.

To determine whether a dam should be built at a given point of the image, it is necessary to know whether the neighboring points belong to the same basin. This information can be obtained efficiently from the min-tree CCT of the image by testing whether there is an ancestor-descendant relationship between two components.

Because each local minimum produces one segmented region, a filter suppressing the insignificant local minima is usually applied beforehand to avoid over-segmentation. This can be done by just filtering the CCT used for the segmentation.

Another widely used over-segmentation reduction technique is the use of markers. This means that selected sets of pixels, called markers, are labeled beforehand with different labels. Then the regions are merged in such a way that each marker establishes one output region and only the most significant watershed lines are retained.

2.4.4 Segmentation without CCT: Inter-pixel watershed

The topological watershed (TW) definition contains some ambiguity coming from the fact that the direction, in which the dams are being built during the immersion, is not explicitly specified, which in turn is because the destructible point selection order may be arbitrary. This leads to many quite different segmentations being a correct TW of a single input image.

Another definition of the watershed transformation exists: the *inter-pixel watershed* (IPW) [Watershed wiki], which is defined using Algorithm 2.2.

Algorithm 2.2: Inter-pixel watershed definition

Input: Image domain V (set of image points)
Input: Pixel neighborhood relation $E \subset V \times V$
Input: Image function $f : V \to \mathbb{R}$
Output: Region mapping $labels : V \to \mathbb{N}_1$

```
1 Label each local minimum of the image with a distinct label and attribute this label to all points of
  the local minimum
2 S ← set of all labeled points
3 while S ≠ ∅ do
4     x ← Extract and remove from S a point with minimal altitude, i.e. f(x) = min{f(y) : y ∈ S}
5     for each non-labeled neighbor y of point x do
6         Attribute the label of point x to point y
7         Insert point y into S
```

Its properties are expected to be quite similar to the TW. However, the IPW removes most of the TW's ambiguity by following the physical watershed definition, which examines the approximate direction, in which a drop of water would flow. Additionally, the IPW produces thin (i.e. zero width) watershed lines. On the other hand, the original IPW algorithm does not provide the connection values [Couprie 2005] between minima of the segmented regions of the image, as the TW does, but they can be obtained by an extension of the algorithm, proposed in Appendix A. The algorithm uses a priority queue for S to further constrain the shape of watershed lines, which divide plateaus. The priority queue could be seen as a bottleneck, but an important property of the algorithm is that the priority of the elements removed from the queue never increases, which allows an efficient priority queue implementation.

2.5 Conclusions

An image processing chain usually consists of multiple operators. The implementation of each operator can be often decomposed into two steps: 1) preprocessing step – scanning of the image and gathering information about its contents, 2) output step – computation of the operator result using the gathered information. Sometimes it is possible to perform these steps at the same time, sometimes it is not.

The connected component tree (CCT) is a powerful representation of image data, which allows efficient implementations of both low-level and high-level image processing operators. The most interesting thing is that the CCT-based operators share the preprocessing step (the CCT construction) and that the information entering into the output step of all operators is in the same format, the CCT. This will allow many components of a CCT-based image processing system to be shared among different operators, which is a big advantage for hardware implementation.

Furthermore, for many CCT-based operators it is natural to produce the output in the form of a CCT too. This may allow the implementation of the next operator in the chain to skip the preprocessing step entirely, speeding up the computation.

Chapter 3

Algorithm architecture matching

The previous chapter introduced the CCT image representation and presented some of its well-known applications. This chapter concerns with existing algorithms for construction of the CCT from the image, which is always the first step in any CCT-based operator implementation. Performance of this step determines the overall speed of the application: [Ngan 2011] states that the CCT construction accounts for as much as 80 % of execution time in existing applications.

3.1 Discussion, evaluation and comparison of existing CCT construction algorithms

In the past, several algorithms have been proposed in order to solve this problem. In majority of cases, they remain sequential and the improvement relies on fast data structures (FIFO-like) [Salembier 1998] or on optimization of computational complexity [Najman 2006]. Some parallelization effort has been done on shared-memory computers [Meijster PhD, Wilkinson 2008]. However, although the obtained performances are interesting, they are insufficient for real-time and not adapted to the embedded systems [Ngan 2007].

3.1.1 CCT construction algorithm classes

Four main classes of CCT construction algorithms exist in literature:

1. *Flooding-based algorithms* [Salembier 1998]: processing starts from the image pixel having the lowest level. A depth-first traversal of tree components, similar to flood-fill, is performed. In general, the flooding process relies on the use of ordered data structures: Either on hierarchical queues, unusable for floating-point pixel representation, or on priority queues, inefficient from the time and memory point of view.

2. *Emerging-based algorithms*: image pixels are processed in decreasing order of luminosity. It requires prior pixel sorting which could be done efficiently. The emerging components are processed as disjoint sets of pixels, based on Tarjan's Union-Find algorithm [Tarjan 1975]. In [Najman 2006], both total path compression and weighting are used to speedup the disjoint set algorithm and the algorithm complexity

is quasilinear. [Berger 2007], [Ngan 2007] and [Berger 2005] use only total path compression in order to save memory.

3. *Merging-based algorithms* [Meijster PhD] are based on the divide-and-conquer principle. Provided that the image has been split into two non-overlapping partitions and a separate CCT for each partition has been constructed, this method modifies the two trees in such a way that the CCT of the entire image results. Recursive application of this principle ends up with single-pixel partitions, for which the CCT construction is trivial. The merging-based algorithms are inherently parallel because the two trees entering into the merging can be constructed independently. However, their computational complexity is high.

4. *1D algorithms*: it is a special category where the CCT is built on 1D signal. Thus, the pixel processing ordering is unnecessary and the tree can be built in linear time [Menotti 2007]. These algorithms are extremely fast, but they cannot process 2D data. However, if tree merging is added, they are usable for any dimension.

The Salembier's flooding algorithm with slight modifications mainly for better readability is described below. The descriptions of other algorithms can be found in the literature.

3.1.2 Flooding algorithm description

Algorithm 3.1 constructs the classic CCT with parent pointers and the component map for an integer-valued image. It is inspired by the flood fill algorithm. It starts from an arbitrarily chosen seed pixel and follows the pixel neighborhood links to flood the entire image in a special order. The found but not yet processed pixels are stored in a hierarchical queue. At the beginning, the queue contains just the seed pixel.

The core of the algorithm is the procedure flood, whose detailed description follows. In each step, a pixel with the highest level present in the queue is retrieved. Its neighbors, which have not been found yet, are inserted into the queue. Each time a pixel with higher level than the retrieved one is found, the procedure flood is invoked recursively to constrain the processing to pixels having the level not lower than the newly found one. Finally, the retrieved pixel is processed, which involves a definition of the component mapping for it, and it is removed from the queue. The used pixel-retrieval order together with the recursive flood invocation ensures that all pixels belonging to a given component are processed consecutively. Once there are no more pixels of the given level in the queue, it means that all pixels of the corresponding component have been processed and the link between the component and its parent has to be defined. To do this, the highest non-empty level of the hierarchical queue is found. If it does not exist, then the component is the root component, which has no parent. After that, the appropriate component-counter *number_nodes* is incremented and the flood procedure returns so that processing of the parent component's pixels can be resumed.

As compared to the original [Salembier 1998] code, there are some minor improvements here, mainly for better readability:

- A few lines of code have been extracted into procedure `process()` to avoid code duplication.
- The loop included through the procedure call at line 4 permits starting of the algorithm from any image point instead of a point with the lowest level.

Algorithm 3.1: CCT construction by flooding

Input: Image domain V (set of image points)
Input: Pixel neighborhood mapping $E : V \to 2^V$
Input: Integral image function $f : V \to D$, $D = \{0, \ldots, f_{max}\}$
Output: Number of connected components at each level $number\_nodes : D \to \mathbb{N}_0$, which defines the set of connected components $CC = \bigcup_{i \in D} \{C_{i,0}, \ldots, C_{i,number\_nodes(i)-1}\}$
Output: CCT parent pointers $parent : CC \to CC \cup \{\perp\}$ ($\perp$ means that the component has no parent which is the case for the root component)
Output: Component mapping $M : V \to CC$

Global: $status : V \to \{\text{NOT\_ANALYZED, IN\_THE\_QUEUE, PROCESSED}\}$
Global: $queue_i$ for up to $|\{v \in V : f(v) = i\}|$ elements of V for each $i \in D$

initialize all elements of *status* to NOT_ANALYZED
initialize all elements of *number_nodes* to 0
choose a seed pixel $q \in V$
`process(`q, 0`)`

procedure `process(`$q \in V$, $level \in D$`)`
    $m \leftarrow f(q)$
    insert q into $queue_m$
    $status(q) \leftarrow$ IN_THE_QUEUE
    while $m > level$ **do**
        `flood(`m`)`

procedure `flood(`**in out** $level \in D \cup \{-1\}$`)`
    while $queue_{level}$ is not empty **do**
        $p \leftarrow$ top of $queue_{level}$ (without removing from the queue)
        for each neighbor $q \in E(p)$ **do**
            if $status(q) =$ NOT_ANALYZED **then**
                `process(`q, $level$`)`
        $M(p) \leftarrow C_{level,number\_nodes(level)}$
        $status(p) \leftarrow$ PROCESSED
        remove p from $queue_{level}$
    $m \leftarrow level - 1$
    while $m \geq 0$ and $queue_m$ is empty **do**
        $m \leftarrow m - 1$
    if $m \geq 0$ **then**
        $parent(C_{level,number\_nodes(level)}) \leftarrow C_{m,number\_nodes(m)}$
    else
        $parent(C_{level,number\_nodes(level)}) \leftarrow \perp$
    $number\_nodes(level) \leftarrow number\_nodes(level) + 1$
    $level \leftarrow m$

- Lines 17–19, which were originally before line 14, have been placed after the loop here. This change has no impact on the algorithm behavior, but permits getting rid of the *node-at-level* array (which was originally required at line 21) by testing the emptiness of the queue instead.
- Line 27, which was originally before line 20, has been moved down to improve readability of the code in-between.

Computational resource requirement analysis

The procedures `process()` and `flood()` may be recursively invoked up to $|D|$ times, which requires that some stack frame space for some of their local variables (minimally for the counter of the for loop at line 14 and for m) is reserved for each invocation. Efficient implementation of the hierarchical queue requires a cumulative histogram of the image which can be computed in linear time $O(|V|+|D|)$ and space $|D|$. The array *status* may share the space with array M, so no extra space is needed for it.

Beyond the memory space required for the input image and the output CCT, the algorithm needs:

- $|V|+2|D|$ integers for the hierarchical queue and the histogram
- $|D|$ integers for the *number_nodes* array
- $2|D|$ integers for the recursion stack

Total memory requirements are therefore $|V|+5|D|$ plus the space for the input and the output. The queue can also be implemented as a linked list in the array M. That would reduce memory requirements by $|V|$.

3.1.3 Past parallelization efforts

In order to improve the execution time, [Meijster PhD] studies the first parallel implementation of CCT construction on shared memory machines. The algorithm is based solely on tree merging which permits the division of the image into an arbitrary number of partitions to parallelize the CCT construction. Although merging of the partitions together is an inevitable step, the CCT construction for the individual partitions is better done by other algorithms than merging, which is slow.

Recently, a modification [Wilkinson 2008] of the Meijster's approach with a computation performance demonstration on 3D data was published. The principle consists of image division into regular domains. A modification of the algorithm from [Salembier 1998] is used to build a CCT of each domain independently. Then, the trees of the domains are merged in a binary-tree fashion.

See Table 3.1, which summarizes the resource requirements and other properties of the existing CCT construction algorithms (both sequential and parallel).

3.2 Parallel programming: Hardware threads, software threads, scheduling and performance

Before proceeding to description of a new parallel algorithm, some reflections on the parallel programming need to be made. This section describes the difference between hardware threads and software threads,

Table 3.1: Complexity analysis of existing CCT construction algorithms. A new algorithm, that will be introduced in the next chapter, is also mentioned for comparison. $N = |V|$ is the total number of pixels in the image, $G = |D|$ is the number of grey levels of the image, α is a very slowly growing "diagonal inverse" of the Ackermann's function, $\alpha(10^{80}) \approx 4$, P is the number of processes

	Algorithm	Time complexity	Memory needs calculation hints	Input data types	Output CCT representation
SEQUENTIAL	**Flooding** [Salembier 1998]	$O(NG)$	$4N + 3G +$ $+$ stack	small int	Classic with parent ptrs + component map
	Emerging [Najman 2006]	$O(N\alpha(N))$	$7N + G +$ $+$ stack	int/float	Classic with bidir. ptrs + component map
	Emerging [Berger 2007]	$O(N \log N)$	$4N$ + stack	int/float	Canonical parent point tree
	Merging [Meijster PhD]	$O(NG \log N)$	$2N$ + stack	int/float	Parent point tree
	1D [Menotti 2007]	$O(N)$	$3N + G +$ $+$ stack	int/float	Classic with bidir. ptrs + component map
	1D [Levillain 2006]	$O(N)$	$2N$	int/float	Parent point tree
	1D + Merging [Levillain 2006]	$O(NG \log N)$	$2N$ + stack	int/float	Parent point tree
PARALLEL	**Flooding + Merging, 3D** [Wilkinson 2008]	$O(NG/P +$ $+ N^{2/3}G \cdot$ $\cdot \log N \log P)$	$3N + 3G +$ $+ P$ stacks	small int	Parent point tree
	1D + Merging (Chapter 4)	$O([N/P +$ $+ \sqrt{N} \log P] \cdot$ $\cdot G \log N)$	$2N +$ $+ P$ stacks	int/float	Parent point tree

scheduling of the software threads into the hardware threads and its impact on the performance, and presents some parallel programming recommendations.

A *hardware thread* is a flow of instructions executed by a processor. A simple processor has one hardware thread. Complex systems, like multi-processor computers, multi-core processors and multi-threading cores (SMT, like Intel's Hyper-Threading – HT), have multiple hardware threads. The number of hardware threads of most systems is fixed and cannot be increased by software means.

Multi-tasking operating systems allow *pseudo-parallel* execution of arbitrary number of *software threads* thanks to *time-division multiplexing* controlled by a *scheduler* [Robison 2007]. Software threads are entities of the operating system and their number is theoretically unlimited, but no more than one software thread at a time may be running on one hardware thread.

Each software thread may be in one of three states:

- *Running:* The software thread is being executed on some hardware thread.
- *Ready:* The software thread wishes to run and there are no obstacles to execution of the software thread except that the scheduler has not assigned a hardware thread to it, perhaps because there is no free hardware thread, and therefore the thread is not running at the moment.
- *Blocked:* The software thread cannot be assigned to any hardware thread, because it is waiting for some event like notification from another software thread or completion of a network or hard disk transfer. Note that such data transfer can be triggered even by a page fault caused by a software thread's access to data, which are not present in the physical memory (RAM), in which case the thread also becomes blocked until the data transfer completes.

In one possible implementation, at any moment each hardware thread executes either the scheduler or some software thread. At some moment, when the scheduler is being executed on some hardware thread, the scheduler may decide to *schedule* some ready software thread to the hardware thread. The scheduler accomplishes this by restoring the hardware thread to the state at which the software thread was *descheduled* (*suspended*) last time and passes control to it, so that it can run.

If there are more non-blocked (i.e. running or ready) software threads than hardware threads, then it is not possible that all of them run simultaneously: They have to take turns. This is called *pseudo-parallel* execution. In this situation all hardware threads are occupied by running software threads and there are some software threads which are ready, but not running. These threads can be scheduled by three ways in principle:

- A running software thread becomes blocked because of waiting for some event. Thus, it deliberately passes control on its hardware thread to the scheduler, which saves the software thread's state and schedules another software thread.
- A running software thread switches to the ready state by voluntarily *yielding* the hardware thread. Again, it deliberately passes control to the scheduler and the rest is the same.
- A running software thread is *preempted* by an interrupt, which overrides execution in the hardware thread to the scheduler. The rest is the same again.

If some running software threads become blocked, it is the opportunity for the ready software threads to take their place, which prevents the hardware thread from being idle and thus improves performance. The number of software threads has to be greater than the number of hardware threads in this case.

As long as everything the software thread works with is in the physical memory and the thread does not have to wait for other threads, the thread will not block. This may be the case for programs, which simply load data from disk into memory, do some processing and finally store the results to the disk. Now if the number of software threads equals to the number of hardware threads, then all hardware threads will be utilized to 100 % until one of the threads completes all its work. If we use more software threads, the scheduler will exchange them on the hardware threads preemptively “against their will" which will lead to unnecessary overhead due to context switching.

3.3 Parallel program performance assessment quantities

This section defines the quantities that will be used for the new algorithm performance assessment.

3.3.1 Speedup

In literature one can usually find a definition of *speedup* like the following one [Herlihy 2008]:

Let T_1 be the time (measured in computation steps) needed to execute a program on one processor; let T_P be the minimum time needed to execute the same program on a system of P processors. Then speedup of the program on P processors is $S_P = T_1/T_P$.

A practical problem with this definition is that it does not say how many software threads should be actually used for program's execution. If we try to adjust the definition to suit real implementations (using POSIX threads for example), we could start like this:

Let $T_{P,M}$ be the wall clock time needed to execute a program on a system of P processors using M software threads.

Now there are basically two ways of dealing with the varying number of software threads M:

- Keep the number of software threads M fixed. Then speedup would be $S_{P,M} = T_{1,M}/T_{P,M}$, where $T_{1,M}$ is the execution time on one processor. But as we have seen before, the optimal number of software threads, which allows to obtain the best time, depends on the number of processors of a given system. One cause is the overhead of context switching. Therefore, the second definition will be prefered:
- For each number of processors P, use the optimal number of threads α_P to obtain the best time $T_P = T_{P,\alpha_P} = \min\{T_{P,M} : M \in \mathbb{N}_1\}$. Then speedup would be $S_P = T_1/T_P$.

3.3.2 Performance improvement

We call *performance improvement* $I_{P,M}$ of a program on a system of P processors using M software threads the ratio $I_{P,M} = T_{P,1}/T_{P,M}$, where $T_{P,1}$ is the execution time with one software thread.

It can be observed that if there are more processors (P) than threads (M), then some processors will not be used and the system will behave like if it had only M processors: $P \geq M \Rightarrow T_{P,M} = T_{M,M}$.

If the optimal number of threads α_M is equal to the number of processors M, then the previous observation gives ($\alpha_M = M$ and $P \geq M$) $\Rightarrow T_M = T_{M,M} = T_{P,M}$. This means that speedup of programs that fulfill the condition $\alpha_M = M$ can be directly measured by changing the number of threads M (which must not exceed the number of processors P in the testing system) because speedup S_M equals to performance improvement: ($P \geq M$ and $\alpha_M = M$) $\Rightarrow S_M = I_{P,M}$.

3.3.3 Efficiency

Efficiency is defined as $E_P = S_P/P$. [Galaghba, cs.cf.ac.uk]

3.3.4 Hardware thread utilization

We call *hardware thread utilization* the average percentage of time when the hardware threads are doing useful work. Some cache events (cache misses) may significantly decrease performance, but these events are considered as useful work too, because they make part of execution of instructions which access memory.

The efficiency is expected to be tightly bound to hardware thread utilization. Therefore, the goal is to maximize hardware thread utilization.

3.3.5 Software thread utilization

We call *software thread utilization* the average percentage of time when software threads are active. It is the time when software threads are doing useful work and they are either running or ready due to preemption. During the period from unblocking to resuming, a software thread is considered inactive. Figure 3.1 shows when a thread is considered active and when not.

When the number of software threads equals to the number of hardware threads, then there will be no need for preemption. In this case if a software thread is active then it is running. A consequence is that the number of utilized hardware threads equals to the number of active software threads and hardware and software thread utilizations are therefore equal.

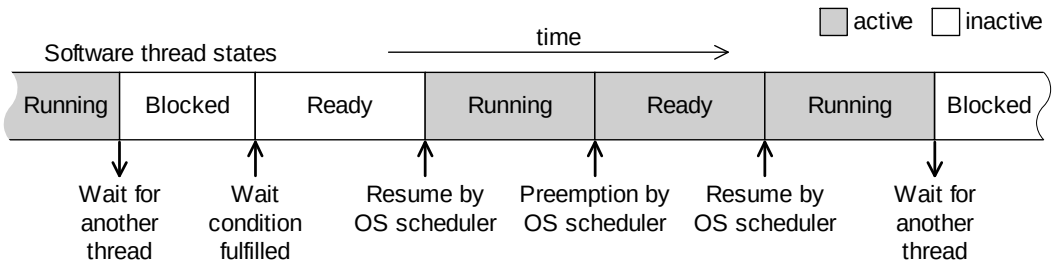


Figure 3.1: Evolution of a software thread in time

This observation will be used for the computation of hardware thread utilization, which is the more difficult one to measure.

3.4 Conclusions

The theoretical analysis of the existing CCT construction methods shows that the tree merging introduced in [Meijster PhD] is the key operation, which permits the parallelization. This merging needs the CCT to be stored as the parent point tree, which was introduced in Section 2.2 together with other CCT representations. Although different algorithms investigated in this chapter use different representations of the resulting CCT, it is always possible to either adapt them to produce any of the representations or at least convert their result in $O(n)$ time and space. For example, [Wilkinson 2008] adapted the [Salembier 1998] algorithm to produce directly the parent point tree.

The 1D algorithm class looks very promising for the hardware implementation, because its members are simple and extremely fast. They also have minimal memory requirements, partly thanks to independent processing of small data chunks, which will be essentially individual lines of an image. On the other hand, an extensive amount of time will be spent by the slow tree merging, which will be needed not only for parallelization, but also to make the 1D algorithm usable for multidimensional signals. Experiments, which will be presented in the next chapter, will show, how serious this disadvantage is.

Chapter 4

Parallel implementation of CCT construction

As stated in the previous chapters, the CCT-based applications usually spend most execution time in the CCT construction step. Parallelization of this step is therefore needed to improve performance significantly.

This chapter presents new parallel algorithm for construction of the CCT. The algorithm constructs a separate tree for each line of the image using a new 1D algorithm inspired by [Menotti 2007]. Then the partial trees are progressively merged using an algorithm from [Meijster PhD].

The algorithm uses a memory-aware data structure (the parent point tree described in Section 2.2), so it is suitable for an embedded system implementation. The 1D CCT construction is extremely fast and therefore the majority of time is spent by the merging steps.

The inherent parallelism of the algorithm is very high and its parallelism granularity is quite fine, so a new adaptive scheduling strategy is needed to unleash its full performance. This chapter presents and evaluates three scheduling strategies: 1) adaptive fine scheduling, 2) fixed scheduling, and 3) adaptive coarse scheduling. The algorithm and the first two scheduling strategies were already presented in [Matas 2008].

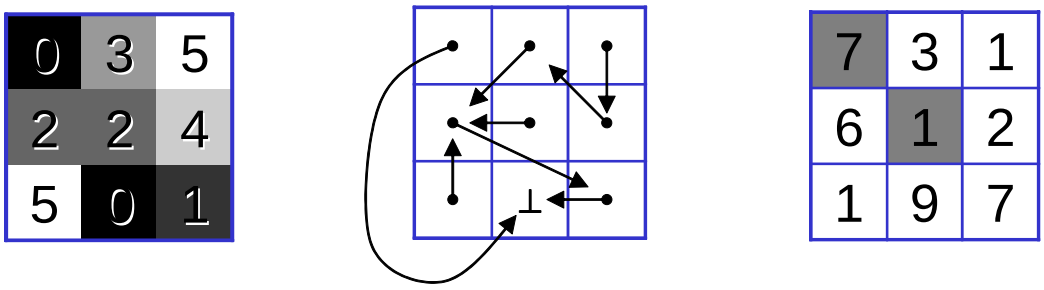
4.1 New parallel algorithm description

4.1.1 High-level description of the algorithm

The input to the algorithm is an iconic image $f : V \to \mathbb{R}$ with a predefined neighborhood relation $E \subset V \times V$. The output from the algorithm (Figure 4.1) consists of:

- the unchanged input image function $f : V \to \mathbb{R}$,
- the array of parent pointers $par : V \to V \cup \{\perp\}$,
- for each attribute computed, an array of attribute values $A_i : V \to$ any set.

The parallel algorithm can be described by a tree-shaped data dependency graph (Figure 4.2).



$f(x)$ $par(x)$ $a(x)$

(a) Unchanged image **(b)** Array of parent pointers **(c)** An attribute: Area

Figure 4.1: Output from the new parallel algorithm. The grey attribute values have no meaning, because they do not correspond to level roots

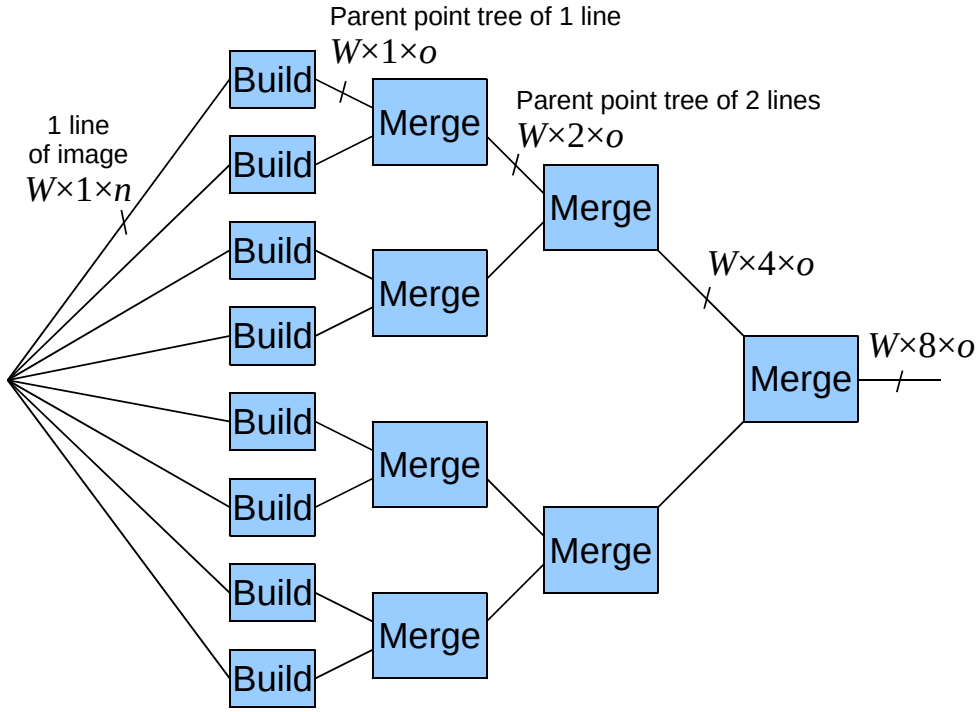


Figure 4.2: Data dependency graph of the new parallel algorithm. Image size in this example is $W \times 8$ px, W is the image width in pixels, n is the number of bits per image pixel, $o = n + p + \sum_i a_i$ is the number of output bits per pixel, p is the number of bits per parent pointer, a_i is the number of bits per value of attribute i

For a 2D input image of $W \times H$ pixels, the algorithm consists of H Build operations and $(H-1)$ Merge operations. The Build operation constructs a tree of one line of the image independently from other lines. The Merge operation takes two adjacent trees and modifies them slightly so that they become a single tree.

Note that the data dependency graph is binary-tree shaped. Generally, this binary tree is not necessarily complete (i.e. not for every node, the sizes of its two subtrees have to be equal), for two reasons: a) the height of the image may not be a power of 2, and b) the parallelization may require that the data dependency graph contains subtrees of specific sizes. However, best performance is obtained when trees of similar sizes are merged, because the execution time of the merging strongly depends on the total size of the two trees being merged.

Before merging, the two trees have to be stored in a single continuous block of memory because the data is not copied during merging. Because the algorithm was implemented on shared-memory parallel machines, where all threads have direct access to all memory, the whole tree can be placed in a single continuous block of memory from the beginning.

4.1.2 Build: 1D partial point tree construction

The original Menotti's 1D CCT construction algorithm produces the classic representation of the CCT, which is not directly usable in the CCT merging step. This subsection presents its modification, which produces the canonical parent point tree, which is suitable for merging and parallelization.

The algorithm accepts one line of the image, i.e. $W \times 1$ pixels, n bits per pixel as the input and produces the point tree of the same size, m bits per parent pointer.

The algorithm processes the points of the line in a single linear scan from left to right and outputs the available parent pointers on the fly. The leftmost point of each component core is chosen as its level root. As to the memory requirements, the algorithm uses only a stack (LIFO) to store the points, whose parent pointers could not be determined yet. The stack supports these operations:

- `StackPush(`x`)` : Adds the point x to the top of the stack.
- `StackPop()` : Removes one point from the top of the stack.
- `StackLast()` : Returns the point at the top of the stack without stack modification or $\perp$ if the stack is empty.
- `StackEmpty()` : Returns true if the stack is empty.

The algorithm is described in Algorithm 4.1, which proceeds as follows. The first point of the line is a level root, because it is surely the leftmost point of some component core. Variable r is initialized to this first point and the scan starts from the second point of the line.

The following invariant holds during the scan: before and after each iteration of the scan, i) the variable r holds the level root of the last processed point and ii) the stack contains all level root ancestors of r encountered so far, ordered by their levels, the highest level on the top of the stack.

The parent pointer of each point is assigned exactly once, just before the point is dropped from all the memory used during the scan: from the stack, from the variable r and from the variable p. The parent pointer is always set to point to a level root, so a perfectly compressed point tree is produced.

Algorithm 4.1: Parent point tree construction for a linear graph

Input: Non-repeating sequence of image points $V = (v_0, \ldots, v_{n-1})$
Input: Image function $f : V \to \mathbb{R}$
Output: Parent point tree $par : V \to V \cup \{\perp\}$

```
1  procedure build1D(V)
2      r ← v_0
3      for p ← v_1 ... v_{n-1} do
4          if f(r) < f(p) then
5              StackPush(r)
6              r ← p
7          else if f(r) = f(p) then
8              par[p] ← r
9          else f(r) > f(p) :
10             loop
11                 q ← StackLast()
12                 if f(q) < f(p) then
13                     par[r] ← p , r ← p
14                     break
15                 else if f(q) = f(p) then
16                     par[r] ← q , par[p] ← q , r ← q
17                     StackPop() , break
18                 else f(q) > f(p) :
19                     par[r] ← q , r ← q
20                     StackPop()
21     while StackEmpty() = false do
22         par[r] ← StackLast() , r ← StackLast()
23         StackPop()
24     par[r] ← ⊥
```

A new point p is processed in each iteration of the scan. Its level $f(p)$ is compared to the level $f(r)$. There are three possibilities:

- $f(r) < f(p)$: point p is the leftmost point of a new component core, so it is a level root. Point r is a (possibly indirect) ancestor of p, because r is the level root of p's left neighbor. Current r is pushed to the stack and p is set as the new value of r. No point is dropped, so no parent pointer is set.
- $f(r) = f(p)$: point p belongs to the same component core as the last processed point and r is its level root, so r is assigned to $par[p]$ and p is dropped.
- $f(r) > f(p)$: the component represented by point r is completed and its parent has to be determined. Let q be the point on the top of the stack. Either p or q is the parent of r, depending on their levels, so $f(p)$ is compared to $f(q)$. Again, there are three possibilities:

– The stack is empty or $f(q) < f(p)$: point p is the leftmost point of a new component core, so it is a level root. It is also the parent of r, so p is assigned to $par[r]$ and r is dropped. Point p is set as the new value of r.

– $f(q) = f(p)$: the points q and p belong to the same component core. The point q is its level root and it is also the parent of r, so q is assigned to both $par[p]$ and $par[r]$ and both p and r are dropped. Point q is removed from the stack and set as the new value of r.

– $f(q) > f(p)$: point q is the parent of r, so it is assigned to $par[r]$ and r is dropped. Point q is removed from the stack and set as the new value of r. But now, $f(r)$ is still greater than $f(p)$. This means that the component represented by the new value of r is completed and its parent has to be determined now. This is done by repeating the decision process with the same p, the new value of r and the new state of the stack.

After the scan has finished, r contains the level root of the last point of the line and all its ancestors are in the stack. They are removed from the stack one by one and parent pointers are set accordingly. The last point removed from the stack is the tree root.

4.1.3 Merging of partial point trees

The parent point trees of individual lines are independent after the building; there are no parent pointer links between the nodes of different lines. To create a complete point tree of the image, which consists of H lines, each two adjacent lines have to be *merged* using Algorithm 4.2. Two adjacent point trees are taken as the input and their parent pointers are modified to create a single point tree. At the start of the merging, the two originally independent point trees become connected and a larger point tree is formed. After that, the process continues with modifications of the tree. The process working on the two trees being merged requires an exclusive access to them to avoid race conditions.

The merging of the two point trees is done by procedure `merge`, which just executes procedure `connect`(x, y) for each pair of points x and y, where x is in the first point tree, y is in the second point tree and x and y are neighbors according to the chosen connectivity. The procedure `connect` follows the parent pointer paths from x and y respectively to the root of the tree and changes the parent pointers to form a single path. The new path includes nodes visited along both paths in correct order of levels. When the two parent pointer paths meet, the procedure terminates. The principle is the same as used in [Wilkinson 2008], with some simplifications.

4.2 Scheduling strategies

4.2.1 Inter-frame parallelism

The parallelization offering the highest throughput is the simplest one: Assigning each image from the input image sequence to one thread. A great advantage of this approach is that there are no dependencies or data

transfers among the threads, which would allow achieving of parallelization efficiencies very close to 100 % for any number of processors. It has two serious drawbacks however:

- Each thread requires memory for processing of an entire image. As an embedded implementation is the aim, such memory requirements are unacceptable.
- This approach does not reduce the latency (the time from input to the corresponding output) of the sequential implementation. High latency may be inhibiting for applications where a fast feedback loop is required.

For these reasons, the inter-frame parallelism was abandoned in favor of speeding up of the single image processing.

Algorithm 4.2: Merging of two parent point trees

Input: Domains of the two trees V_1, $V_2 \subset V$ $(V_1 \cap V_2 = \emptyset)$
$V_{12} = V_1 \cup V_2$ is the domain of the resulting tree
Input: Pixel neighborhood relation $E \subset V_{12} \times V_{12}$
Input: Image function $f : V_{12} \to \mathbb{R}$
Input: Parent point tree $par : V_{12} \to V_{12} \cup \{\perp\}$
Output: Modified par

```
1  procedure merge(V1, V2 ⊂ V)
2      for each pair of points (x, y) from E ∩ (V1 × V2) do
3          connect(x, y)
4  procedure connect(x, y ∈ V12)
5      x ← levroot(x) ,  y ← levroot(y)
6      if f(y) > f(x) then swap(x, y)
7      while x ≠ y do
8          if par[x] = ⊥ then
9              par[x] ← y ,  x ← y
10         else
11             z ← levroot(par[x])
12             if f(z) > f(y) then
13                 x ← z
14             else
15                 par[x] ← y ,  x ← y ,  y ← z
16 procedure levroot(x ∈ V12)
17     if f(x) = f(par[x]) then
18         par[x] ← levroot(par[x]) ,  return par[x]
19     else
20         return x
```

4.2.2 Intra-frame parallelism

Parallelism of the new algorithm is very large, at least at the beginning of processing: the number of Build operations, which could theoretically be executed concurrently, usually greatly exceeds the number of available hardware threads. Some scheduling of the operations is therefore needed to be able to execute the algorithm efficiently. Three scheduling strategies will be proposed in this section.

The target platform for the evaluation of the algorithm is a shared memory parallel machine with Pentium-like processors. The most easily implementable scheduling on such systems is to create one software thread per Build operation and to let the operating system schedule the software threads into the hardware threads automatically. However, this solution has several drawbacks coming from the fact that the number of software threads usually greatly exceeds the number of hardware threads:

- Thread creation and destruction overhead is large.
- Preemption is applied to let all software threads progress. This leads to overhead from context switching.
- Each time a software thread is preempted, the next software thread scheduled into the same hardware thread evicts the previous software thread's data from the CPU cache. This decreases memory subsystem performance.

A better solution is to create as many software threads as hardware threads and to distribute the operations among them in such a way that each software thread executes some of them. The execution of software threads is concurrent (truly parallel), but inside of each of them the operations are executed sequentially. Note that this solution involves two layers of scheduling:

- Scheduling of software threads into hardware threads by operating system scheduler. The assignment between hardware and software threads will stay unchanged for long periods of time because preemption is used only to let run other processes in the system.
- Scheduling of operations into software threads by user level scheduler.

Aside from the direct programmatic control of the software threads and their workload used in this implementation, language extensions and software libraries exist which perform the scheduling automatically, such as *OpenMP* or *Intel Threading Building Blocks (TBB)* [Mahmoudi PhD]. The principle of using these technologies is that the programmer just marks the portions of the code, which can be executed concurrently. The system then decides automatically (based on the available resources) how many software threads should be used, creates them, and assigns the workload to them. However, this possibility was not expored, because the final goal of this work is a hardware implementation of the algorithm, where scheduling has to be done explicitly.

Let us return to the programmatic scheduling now. Obviously, the programmer can affect performance mainly through behavior of the user level scheduling, so three different scheduling strategies were developed to compare their performance.

An obvious property of any implementation of the algorithm is that when a Build operation completes, its result is in the CPU cache. A common goal of all of the scheduling strategies is to minimize data transfers between processors and main memory. Therefore, in all three strategies, if both inputs of the Merge operation

following the completed Build operation are ready, then the thread, which completed the Build operation, executes immediately the Merge operation to take advantage of the cached data. The same procedure is taken when that Merge operation completes and so on. In other words, when any thread completes a Build operation, it tries to continue immediately with the cascade of Merges, until some Merge, whose other input is not yet available, is encountered. In that moment, the thread leaves the cascade of Merges and continues with the next Build operation.

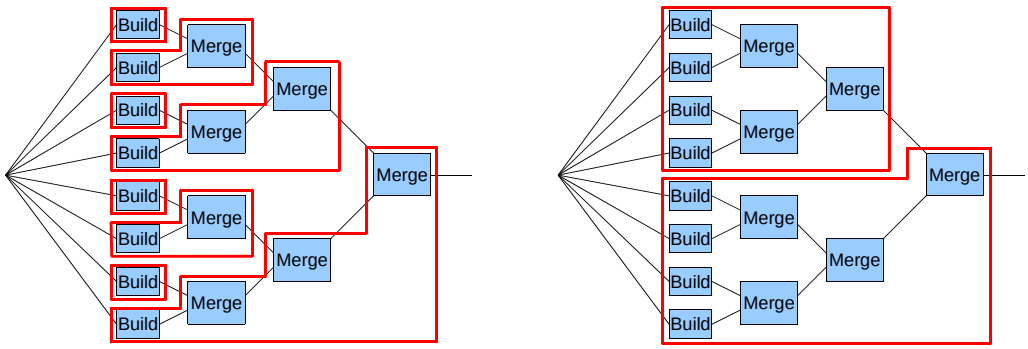
The strategies differ by the way of distribution of the Build operations among the threads.

4.2.3 Strategy 1: Adaptive fine scheduling

The data dependency graph for this strategy is constructed in such a way that only its subtrees containing the last line of the image may be incomplete.

This scheduling strategy puts the Build operations into a queue, which is common to all threads, so that the Build operations are started in the order from top to bottom of the image. Figure 4.3a shows which operations are done by a single thread. This strategy ensures that the threads advance uniformly even if their processing speeds are not equal. This may be important for performance because the speeds of the Merge operations depend strongly on the image contents. On the other hand, the data, which were produced in different hardware threads, are transferred between corresponding caches during the Merge operations. These transfers may reduce performance, especially if the hardware threads are not placed in the same CPU chip because communication among different chips is quite slower than inside of a single chip.

Strategy 1 is implemented by Algorithm 4.3, whose detailed description follows: The algorithm is executed in $P \in \mathbb{N}_1$ independent parallel processes (threads). Numbered lines in a queue are fetched by the individual threads. The queue is represented by a shared *next_line* counter. The counter has to be read first and then incremented on each fetch. This is done by the atomic procedure `get_next_line`. The atomicity means that the procedure has to be executed as a single operation to avoid race conditions. When a line is fetched by the thread, the thread performs the building operation. Then the thread proceeds to the cascade of merging



(a) Adaptive fine scheduling **(b)** Fixed scheduling

Figure 4.3: Distribution of the Build and Merge operations among threads. The enclosing polygons show which operations are done by a single thread

Algorithm 4.3: Concurrent parent point tree construction with Strategy 1: Adaptive fine scheduling

Input: Number of processes $P \in \mathbb{N}_1$
Input: Image width W and height $H \in \mathbb{N}_1$
$V = \{0, ..., W-1\} \times \{0, ..., H-1\}$ is the image domain (set of image points)
Input: Pixel neighborhood relation $E \subset V \times V$
Input: Image function $f : V \to \mathbb{R}$
Output: Parent point tree $par : V \to V \cup \{\perp\}$

Global: $next\_line \leftarrow 0$
Global: $block\_ready[1 \; ... \; H-1] \leftarrow$ false

```
1  for i ∈ {1, ..., P} do concurrently
2      while (line ← get_next_line()) ≠ −1 do
3          build1D((0, ..., W − 1) × (line))   // Build partial tree
4          block_no ← line
5          block_size ← 1
6          while block_size < H do   // Merge as deep as possible
7              block_no ← ⌊block_no/2⌋
8              block_size ← block_size · 2
9              first ← block_no · block_size
10             border ← (block_no + 1/2) · block_size
11             last ← min{H, (block_no + 1) · block_size} − 1
12             if border < H then
13                 if can_merge(border) = true then
14                     merge({0, ..., W − 1} × {first, ..., border − 1}, {0, ..., W − 1} × {border, ..., last})
15                 else
16                     break

17 atomic procedure get_next_line()
18     if next_line < H then
19         return next_line++
20     else
21         return −1

22 atomic procedure can_merge(border ∈ {1, ..., H − 1})
23     result ← block_ready[border]
24     block_ready[border] ← true
25     return result
```

operations following the path from the build job to the right in the data dependency graph. The thread performs as many merging operations as possible without waiting. Each merging job requires two inputs (two point trees); one tree (ending at line $border - 1$) just above the other (starting at line $border$). The meaning of the variable *border* is also illustrated by Figure 4.4. One of the inputs is carried by the thread which is about to perform the merging job. The other input has to be supplied by another thread. The boolean array *block_ready* is used to keep track of merging jobs which are ready to be done. An element of *block_ready* is true if the corresponding merging job has at least one input ready. The thread, which attempts the merging, calls the atomic procedure `can_merge`. This procedure reads the previous value of the corresponding element of *block_ready*, which it will return, and sets it to true. If the read value is true (the thread is the second one to attempt this merging job), it means that the other input of the merging job is ready and the job is performed by the thread. If the read value is false (the thread is the first one to attempt the merging job), the other input is not ready yet, so the thread leaves the cascade of merging operations and fetches next unassigned line to be built. We can see that each building job will be done exactly once, because each of them is fetched by one thread. Also each merging job will be done exactly once, because exactly one thread attempts each merging job as the second. If a thread attempts to fetch next unassigned line, but there are no more lines left, the thread exits. The algorithm ends when all threads exit.

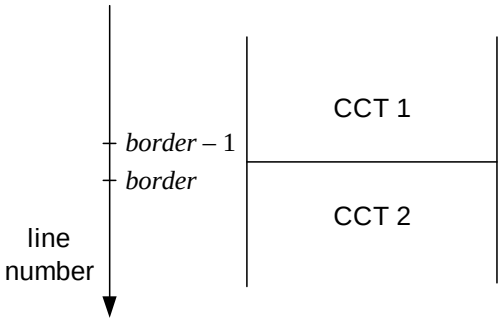


Figure 4.4: Meaning of the variable *border* (the merging boundary line number)

4.2.4 Strategy 2: Fixed scheduling

The second scheduling strategy divides the image into approximately equally sized blocks whose number equals to the number of threads. The data dependency graph is constructed in such a way that each block forms a subtree of the graph. The Build operations of each block are assigned to one thread, so all operations (Build and Merge) in a block are performed by a single thread. This way the data transfers among caches corresponding to different threads are avoided until inter-block merging starts. On the other hand, if some threads finish their work earlier than others, then they will become idle, because the workload assignment is fixed.

The details of the implementation of Strategy 2 follow: The algorithm is executed in $P \in \mathbb{N}_1$ independent parallel processes (threads). The entire image is divided into P roughly equally sized blocks of approximately (H/P) lines (the numbers of lines per block have to be integers). The data dependency graph is constructed in such a way that each block forms a subtree (Figure 4.3b). Each block is assigned to one thread. The thread builds and merges all lines of the block into the point tree of the block, just like the Algorithm 4.3 running in a single thread does. It means that after building of a line, as many merging steps as possible

are done. Synchronization is no more necessary, because the job execution order is known in advance. After the block was built and merged, it is merged with the other blocks. If we consider processing of the block as a single building operation, the data dependency graph applies to the merging of the blocks as for the Algorithm 4.3. One of the threads, which reach a merging operation, terminates and the other thread proceeds with merging. The final point tree is available when last two blocks are merged.

4.2.5 Strategy 3: Adaptive coarse scheduling

The last developed scheduling strategy divides the image into blocks too, but their number can be set by the user. Parallelism granularity can therefore be chosen independently from the number of software threads. Typically the number of blocks is $B = 2^k M \leq H$, where M is the number of threads, H is the number of lines of the image and k is the largest natural number fulfilling the relation $B \leq H$. The blocks are numbered from top to bottom of the image and they are put into a common queue ordered by a bit-reversed representation of their numbers. After a block taken from the queue has been processed by a thread, this thread continues by processing the following blocks according to the normal (not bit-reversed) order. Whenever an already taken block is encountered, the thread takes another block from the queue. This way each thread processes a large contiguous region of the image independently, but if some threads finish earlier than the others, the remaining regions can be split and redistributed.

4.2.6 Theoretical comparison of the strategies

The three scheduling strategies are compared in Table 4.1.

All three strategies share a problem with a long final cascade of Merges: After some thread completes the last Build operation, other threads either are idle already or will become idle soon. This means that the final cascade of Merges is performed sequentially, which reduces the performance. A solution to this problem by deferring a few highest levels of Merging to the end of processing was tried, but this did not lead to better execution times.

Having equal numbers of hardware and software threads is optimal for strategies with workload balancing, because at every moment, if there are enough operations which can be executed in parallel, then no software thread will be idle, so all hardware threads will be utilized. If we added more software threads in this situation, hardware thread utilization would not increase, because it was already at 100 %, but software

Table 4.1: Features of proposed scheduling strategies. Sequential input means that the lines of the image are processed in the order from top to bottom of the image.

	Memory access optimized for	**Workload balancing**	**Sequential input**
Strategy 1	Single CPU chip	Yes	Yes
Strategy 2	Multiple CPU chips	No	No
Strategy 3	Multiple CPU chips	Yes	No

thread management overhead would increase. Some software threads can become idle only if there are no more operations which can be executed concurrently. Decomposition of the algorithm into operations is practically independent from the number of threads, so if we added more software threads in this situation, they would be idle too, because there would be no operations which could be assigned to the new threads.

4.3 Performance evaluation

4.3.1 Time complexity and memory requirements

The time complexity of the building operation (procedure `build1D`) was already analyzed in [Menotti 2007]; its execution time is always $\Theta(W)$, where W is the width of the image in pixels.

For its memory requirements, let us consider that: no point can be inserted twice into the stack; no two points simultaneously present in the stack can have the same level; the last point is never inserted into the stack; the points with the highest level are never inserted into the stack. Thus, the maximum stack size is $\min\{W-1, G-1\}$, where $G = |D|$ is the number of grey levels of the image.

The merging operation time complexity was already analyzed by Wilkinson [Wilkinson 2008] and its worst-case execution time is $O(WG \log N)$, where $N = |V| = WH$ is the number of pixels of the image. It is clear that the total time complexity of the overall algorithm is dominated by the merging operations.

The merging operation does not need any additional memory except input and output buffers. The recursive procedure `levroot` needs a stack to store the points for path compression. It is possible to avoid the need for stack by implementing this procedure without recursion, but the parent pointers have to be read twice.

Each thread has to process H/P lines, which means to perform H/P Builds and $H/P - 1$ Merges. The worst-case execution time of this step is therefore $O(NG/P \cdot \log N)$. After that, some threads have to go through the final cascade of Merges, whose depth is $\Theta(\log P)$, so the worst-case execution time of this step is $O(WG \log P \log N)$. The total worst-case execution time for the whole image using P threads is $O([N/P + W \log P] \cdot G \log N)$. Using the approximation $W \in \Theta(\sqrt{N})$, it can be simplified to $O([N/P + \sqrt{N} \log P] \cdot G \log N)$.

The total memory requirements are input buffer of N pixels, output buffer of N memory pointers and P stacks of size $\min\{W-1, G-1\}$ each.

4.3.2 Execution time measurement method

Execution time is the main quantity used for algorithm performance assessment. Most systems support precise time measurements with fine resolution, but special system-dependent APIs have to be used to get them. For this reason, a small C/C++ multi-platform library called `Timer` was created for measurements of time. It contains implementations for Windows and for Linux:

- The Windows implementation uses Windows Performance Timer API
- The Linux implementation uses `gettimeofday()` function

The library contains these functions:

- `void reset_performance_timer(void);` This function sets the timer to zero.
- `double get_performance_timer(void);` This function returns the time in seconds since the last timer reset as a floating-point number.

The library will be used for majority of software implementation execution time measurements.

The following subsections test the resolution and faithfulness of the time measurements that the library offers.

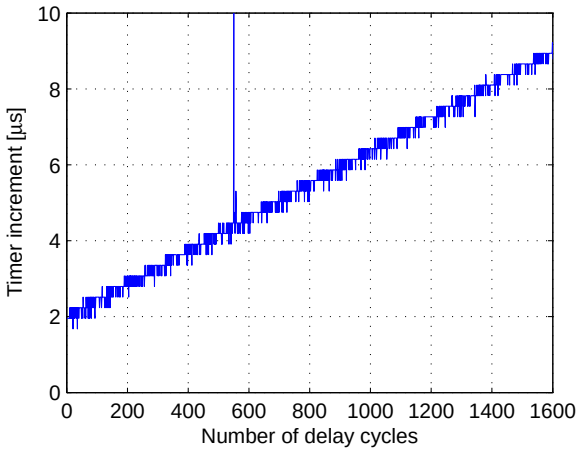
4.3.3 Time measurement resolution assessment experiments

The library was tested on a laptop with Pentium Dual-Core. There was a fresh installation of Windows XP SP3 with chipset drivers only. The following loop was used to measure resolution of the timer:

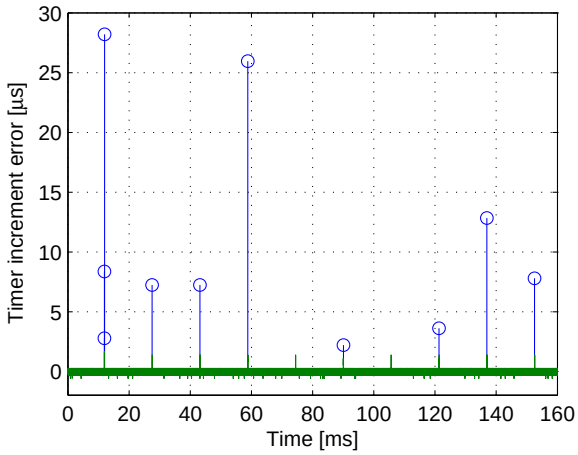
```
for (i = 0; i < RESOLUTION_RUNS; i++) {
  resolution_values[i] = get_performance_timer();
  for (counter = i; counter > 0; counter--) ;
}
```

In this code, each iteration of the outer loop is delayed by a gradually increasing number of iterations of the empty inner loop. Figure 4.5a shows differences between consecutive readings from the timer (i.e. timer increments). It can be seen that each access to the timer costs about 2 microseconds, but its resolution is well below one microsecond.

The second test serves to reveal any "jumps" or discontinuities of the timer. Here the timer was read as frequently as possible, with no delay between `get_performance_timer()` calls. Figure 4.5b shows differences



(a) Resolution. The 1 600 samples shown in the graph make 8.79 ms in total



(b) Regularity. The graph contains 81 037 samples

Figure 4.5: Timer evaluation results

(errors) of timer increments from usual value of timer increment. The error was decomposed into normal noise, which is always present, and rare major delays which are displayed as stems. One such major delay is visible also in Figure 4.5a. These delays are probably caused by invocations of routines of other processes during regular interrupts in the system.

Conclusion from the experiments on Windows XP: If we tolerate unpredictable delays in program execution caused by multi-tasking environment then the time measurements offered by the Timer library can be used for wall clock time measurements with precision about one microsecond. Programmers have to keep in mind that each access to the timer costs some time.

4.3.4 Results

The algorithm described earlier in this chapter was implemented in the C programming language and some measurements of its performance were made. Figure 4.6 shows the time measurement results for the algorithm using the following configuration:

- **Measured quantities:** wall clock time, performance improvement, software thread utilization
- **Image size:** 2816×2816 px (7.93 Mpx), 8 bits per pixel (grayscale)
- **Data sizes:** input image: 1 byte per pixel; output tree: 4 bytes per pixel
- **Number of threads:** 1 – 64
- **Scheduling strategies:**
  - Strategy 1: Adaptive fine scheduling
  - Strategy 2: Fixed scheduling
  - Strategy 3: Adaptive coarse scheduling
- **Computer:** 8× Dual-Core AMD Opteron 885 (2.6 GHz) – manwe1 from [MetaCentrum]
- **Operating system:** SUSE Linux, kernel version 2.6.18-smp
- **Compiler:** GCC version 4.1.2

Figure 4.7 shows the performance improvement for two different image sizes on a computer with two Hyper-Threading processors.

Appendix B contains more timing results using more computers with different processor configurations.

Measurement of L2 data cache miss count during execution of the algorithm on ikaros (computer description is in Table B.1) using PerfSuite [PerfSuite] under Linux for the small image (784×576, 8 bits) and optimal number of threads (2) was also done for each strategy. The results are shown in Table 4.2.

4.3.5 Discussion of the results

All implementations of the new algorithm contain a user-level scheduler and they completely avoid dependencies among software threads. Therefore, the preemptive behavior of the operating system scheduler in case of more software threads than hardware threads hampers the effort of the user-level scheduler. Software thread utilization quickly decreases in that case, which confirms that a system cannot achieve bigger true

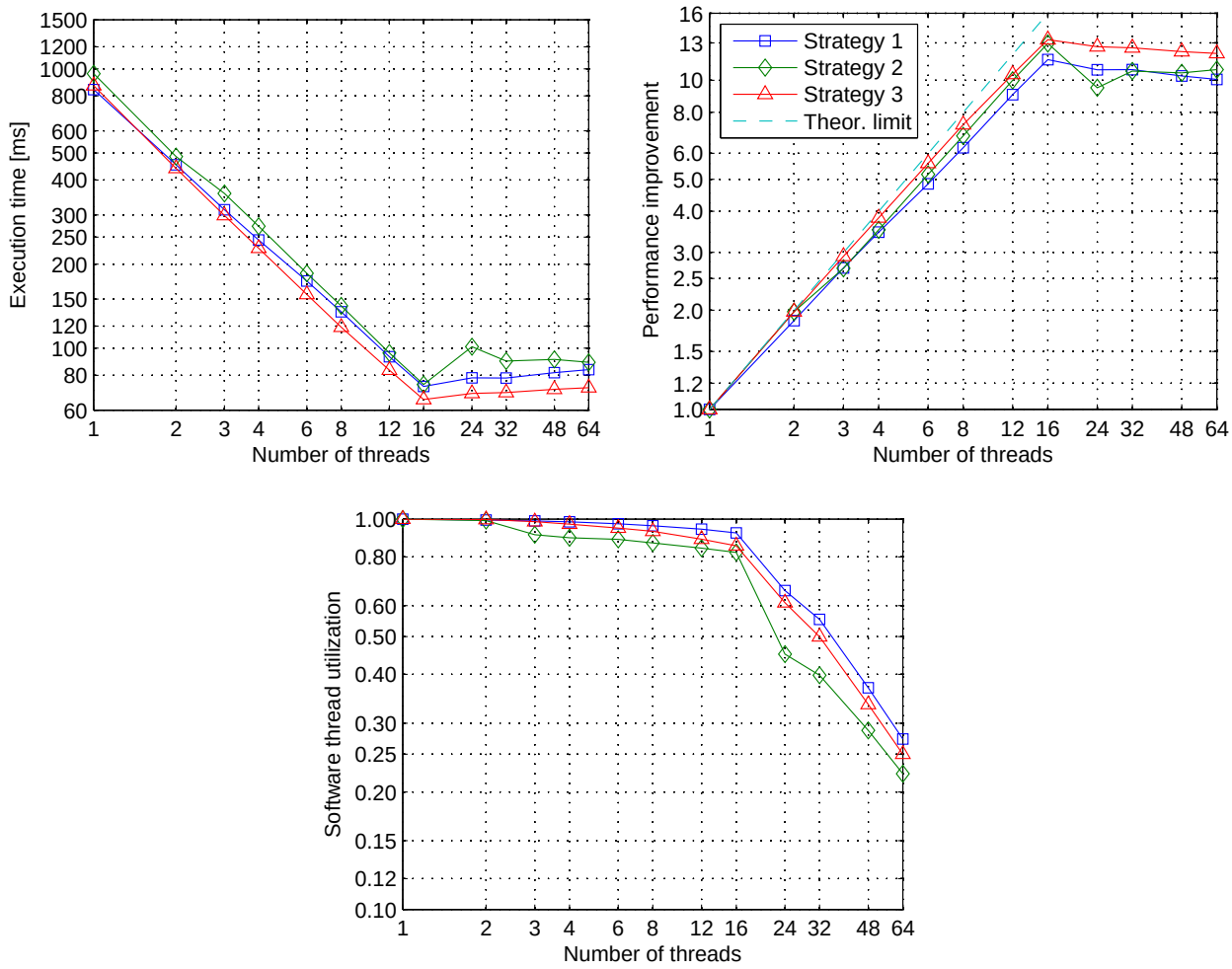
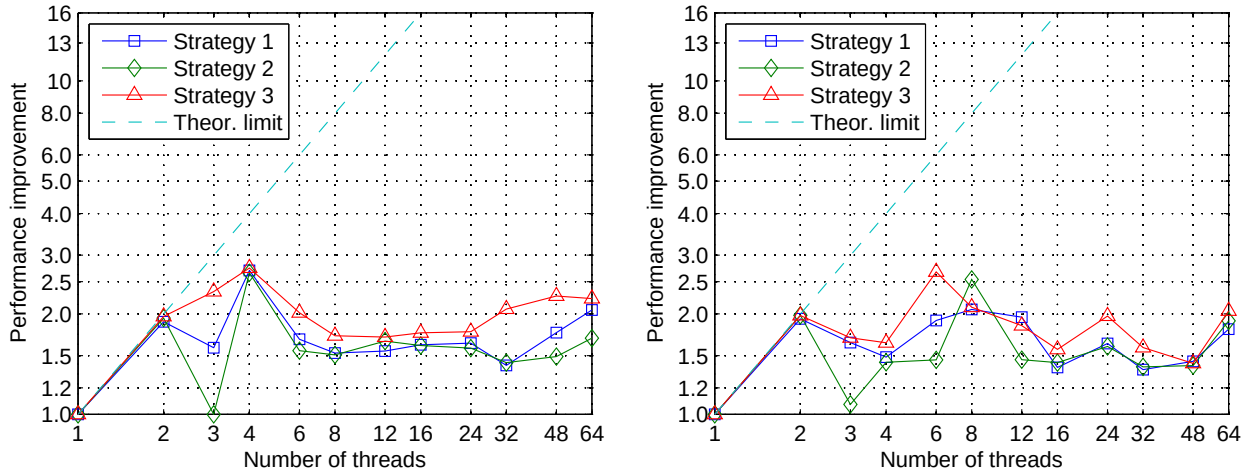


Figure 4.6: Timing results of the new algorithm on 16 cores. Both axes use a logarithmic scale. The line labeled "Theor. limit" shows the theoretical limit: efficiency equal to 1

Table 4.2: Cache miss count measurement results for the small image (784×576, 8 bits)

	L2 data cache miss count
Strategy 1	43 416
Strategy 2	51 316
Strategy 3	52 461



(a) 784×576 px **(b)** 2816×2816 px

Figure 4.7: Timing results of the new algorithm on 2 Hyper-Threading (HT) cores for two different-sized 8-bit images

parallelism than it corresponds to the number of its hardware threads. This explains why there is no speed gain after the number of software threads reaches the number of hardware threads. The only exception are some results from the Hyper-Threading machine which show very erratic behavior, so they will be discussed separately.

The performance improvement graphs show a good agreement on systems with different number of cores if the number of software threads M is not greater than the number of cores P. This is because if the number of threads M is lower than the number of cores P, then only M cores are employed and the system behaves like if it had only M cores.

Best performance improvement is obtained when the number of software threads equals to the number of hardware threads ($\alpha_P = P$), which confirms that the new algorithm has the property mentioned in Section 3.3.2. This means that the rising parts of performance improvements are also speedups. It does not apply to the Hyper-Threading machine, because when one hardware thread is not used, then performance of the other hardware thread of the same core significantly increases.

The aim of Hyper-Threading and other SMT technologies is better utilization of core's execution units, because one thread can take advantage of units which are not used by the other thread at the moment. It is therefore expected that when both hardware threads of the core are utilized, the performance of one hardware thread should not drop below 50 % and the overall performance should rise.

Some results of the experiments confirm this and the best performance improvement on the Hyper-Threading machine (2 cores HT) reaches about 2.7, which means that Hyper-Threading boosted overall performance by about 40 % in this case. However, the behavior of this machine is very erratic and most of the time the speedup is below 2, which shows that employment of Hyper-Threading actually decreased performance. Probably this is because the two threads sharing the core compete for its resources (particularly the L1

cache) in such a way that none of them can “win". It can be concluded that to profit from Hyper-Threading the program has to be specifically designed so that the two threads tightly cooperate.

As to the size of the image, the speedup is better for the larger image, because the last steps of merging, which have to be done sequentially, constitute a relatively smaller portion of computation. Software thread utilization, which is bigger for the larger image, confirms it. User-level scheduler overhead is also relatively lower.

4.3.6 Comparison of the scheduling strategies

Strategies 2 and 3 show better speedup than Strategy 1 on multi-CPU machines, because Strategy 1 requires big amounts of data to be transferred among caches of the CPUs. It is because each line of image can be assigned to different thread. Strategies 2 and 3 assign a big block to each thread, so that these transfers are not necessary. Speedup of Strategy 1 is lower in spite of better thread utilization, which occurs in some cases.

Strategy 1 is slightly better than the other two strategies on the machine with a single CPU chip (ikaros). It is because the threads share the L2 cache and it is therefore better if they work with the same data which is the case of Strategy 1. This saves some data transfers between the CPU and RAM chips. Measurements of cache miss counts (Table 4.2) prove it.

The performance of Strategy 2 on the Linux machines is about 15 % worse than performance of the other strategies, even for only one thread; the cause is unknown.

Strategy 2 shows a significant performance drop when the number of threads is slightly higher than the number of hardware threads. It is because the initial workload distribution is not good in this case and, unlike the other strategies, this strategy provides no workload balancing mechanism. This leads to a significant drop in thread utilization.

4.4 Conclusions

A new parallel algorithm for CCT construction was designed. The choice of its routines was driven by the aim to obtain a simple algorithm with low memory requirements, which are the requirements for an embedded hardware implementation. The new algorithm needs no more memory than the input and output buffers and a stack. It does not require pixel ordering or complex data structures like hierarchical queues. Thus, it fulfills both mentioned requirements.

Additionally, the new algorithm is inherently heavily parallel, because it is based on the divide-and-conquer principle. The parallelization is therefore straightforward, but good scheduling strategies had to be designed to unleash the full performance of the algorithm. Unlike scheduling by the operating system, the designed strategies are tailored for the given algorithm and they may therefore achieve lower overhead and higher efficiency.

The strategies are designed to occupy each hardware thread by one software thread and to keep it busy as long as possible. This avoids preemptive behavior of the operating system scheduler and it is also an

advantage for future hardware implementation, which will not need scheduling of software threads into hardware threads.

Threads of Strategy 1 cooperate on processing of the same region of image at given time. This strategy is therefore well suited for systems where communication among hardware threads is significantly faster than communication with RAM chips, like computers equipped with a single multi-core processor. Lines of the image are input into the computation in sequential order, which allows processing of the image while it is being transferred from a sensor.

Threads of Strategies 2 and 3 process separate regions of image as long as possible, communication among hardware threads is therefore minimized. These two strategies are well suited for systems where communication among hardware threads is *not* significantly faster than communication with RAM chips, like computers with multiple processor chips.

Execution speed depends strongly on image contents, workload balancing may therefore be important for performance. Strategies 1 and 3 provide a workload balancing mechanism, Strategy 2 does not.

Speed measurements confirm that optimal number of software threads for the new algorithm equals to the number of hardware threads. Condition from Section 3.3.2 is therefore fulfilled and speedup of the new algorithm can be measured as performance improvement by changing the number of software threads. Top speedup was 13.3 on the 16-core system, which gives efficiency $E_{16} = 83$ %.

Now let us compare the properties of the new algorithm to the fastest formerly known concurrent algorithm [Wilkinson 2008]. Its main difference to the new algorithm is that, for the building step, it uses a flooding-class algorithm, which requires a hierarchical queue (increase of memory requirements) and computation of an image histogram (algorithm complication). The new algorithm is therefore better suited for embedded implementation.

The flooding algorithm is also slow when the difference of neighbor levels in the image is large. For this reason, it is not able to process floating-point images, where the number of available levels is too high. On the other hand, the new algorithm does not depend on absolute levels, so it is able to process any pixel data type (including floating-point representation) without large performance loss.

Additionally, the Wilkinson's algorithm has to use many software threads to achieve the best execution speed, even on a system with one hardware thread. The obtained performance improvement (72 %) cannot come from parallelism in this case: Even a single software thread is sufficient to fully utilize one hardware thread, because it does not have to wait for any external events. Wilkinson himself states that this performance improvement comes from better cache utilization. Generally, this means that the computation is more efficient when it is divided into smaller tasks, even if they are still executed sequentially. Necessity to use big number of threads to reach top performance is therefore not an advantage. We can also say that Wilkinson's algorithm uses big number of threads to optimize behavior of each thread, but threads of the new algorithm already behave optimally with any number of threads from this point of view.

According to definitions from Section 3.3, the value labeled in Wilkinson's paper [Wilkinson 2008] as speedup is performance improvement. Condition from Section 3.3.2 is not fulfilled for the Wilkinson's algorithm, which means that its speedup is lower than performance improvement which cannot be directly compared to speedup of the new algorithm.

Chapter 5

Hardware architecture for CCT construction

Variety of computer vision systems were developed in recent years. All of them are more or less fitted to certain group of applications. The challenge is to design an embedded system covering a large variety of existing or possible applications. In the previous chapters, we have seen that image-processing algorithms based on the connected component tree (CCT) provide a wide palette of image operators. Their hardware implementation could therefore accomplish that.

However, the CCT construction itself is the bottleneck of these methods and represents about 80 % of the processing time when processed sequentially [Ngan 2011]. Obviously, real-time performance requires parallel CCT construction. This has been done with only limited efficiency on the general purpose processor based parallel computers [Meijster PhD, Wilkinson 2008, Matas 2008]. Moreover, such systems are not well suited for embedded applications. A special hardware architecture is therefore needed. According to the author's knowledge, the unique FPGA implementation of CCT construction has been published in [Ngan 2007] and [Ngan 2011] and relies on a sequential algorithm.

In the previous chapter, a new fast concurrent CCT construction algorithm was presented, which is well suited for hardware implementation. This parallel algorithm is based on building of 1D trees for individual image lines and their progressive merging.

This chapter presents a new parallel hardware implementation of its slight modification presented also in [Matas 2010]. The image is divided into independent partitions, which are processed concurrently, but merging of these partitions requires access to all partitions. A special interconnection switch is designed to solve this problem.

5.1 Algorithm modifications for hardware implementation

The new parallel implementation is based on division of the image of size $W \times H$ into P non-overlapping equally sized partitions of size $(W/P) \times H$, which cover the entire image. Then the building algorithm from the previous chapter (Algorithm 4.1, inspired by [Menotti 2007]) is used to construct partial point trees of image lines inside each partition.

The merging algorithm (Algorithm 4.2), [Wilkinson 2008, Meijster PhD] combines the partial trees of lines into the partial tree of the entire partition. The partitions are processed independently and in parallel. The same merging is used to combine the trees of the partitions into a resulting global tree.

In this section, both algorithms will be further adapted in order to enable their hardware implementation.

5.1.1 1D CCT construction algorithm

Algorithm 5.1 constructs the parent point tree of a weighted linear graph. It reads the input pixels sequentially, compares their values to previously read pixels, and pushes them onto a stack (LIFO). At appropriate

Algorithm 5.1: Parent point tree construction for a linear graph. Adapted for a hardware implementation

Input: Non-repeating sequence of image points $V = (v_0, \ldots, v_{n-1})$
Input: Image function $f : V \to \mathbb{R}$
Input: Point attribute function $A : V \to M$
Output: Parent point tree $par : V \to V$
Output: Accumulated attributes $attr : V \to M$

```
1  procedure build1D(V)
2      next_parent ← NONE
3      i ← 0 , attr[v_i] ← A(v_i)
4      while i < n do
5          if StackEmpty() ∨ f(x) < f(v_i) then
6              if next_parent = X_OR_P then par[z] ← v_i , attr[v_i] ← attr[v_i] + attr[z]
7              else if next_parent = X_ONLY then par[z] ← x , attr[x] ← attr[x] + attr[z]
8              StackPush(x) , x ← v_i , i ← i + 1 , attr[v_i] ← A(v_i)
9              next_parent ← NONE
10         else if f(x) = f(v_i) then
11             if next_parent ≠ NONE then par[z] ← x , attr[x] ← attr[x] + attr[z]
12             z ← v_i , i ← i + 1
13             next_parent ← X_ONLY
14         else f(x) > f(v_i) :
15             if next_parent ≠ NONE then par[z] ← x , attr[x] ← attr[x] + attr[z]
16             z ← x , x ← StackPop()
17             next_parent ← X_OR_P
18     if next_parent = NONE then
19         z ← x , x ← StackPop()
20     while StackEmpty() = false do
21         par[z] ← x , attr[x] ← attr[x] + attr[z]
22         z ← x , x ← StackPop()
23     par[z] ← z
```

Algorithm 5.2: Merging procedure `connect` adapted for a hardware implementation

Input: Nodes to connect $u, v \in V$
Input: Image function $f : V \to \mathbb{R}$
Input: Parent point tree $par : V \to V$
Input: Accumulated attributes $attr : V \to M$
Output: Modified par and $attr$

```
1  procedure connect(u, v ∈ V)
2      cor ← 0 ,  z ← u
3      while is_levroot(z) = false do
4          z ← par[z]
5      y ← z
6      z ← v
7      while is_levroot(z) = false do
8          w ← par[z]
9          if f(z) = f(y) then par[z] ← y
10         z ← w
11     if       f(z) > f(y) then  x ← z ,  is_levroot(x) ← true
12     else if  f(z) = f(y) then  x ← z ,  is_levroot(x) ← false
13     else     f(z) < f(y) :     x ← y ,  is_levroot(x) ← true ,  y ← z
14     z ← par[x]
15     while x ≠ y ∧ par[x] ≠ x do
16         while is_levroot(z) = false  do
17             w ← par[z]
18             if f(z) = f(y) then par[z] ← y
19             z ← w
20         if       f(z) > f(y) then  par[x] ← z,  attr[x] ← attr[x] + cor,  x ← z,  is_levroot(x) ← true
21         else if  f(z) = f(y) then  par[x] ← y,  attr[x] ← attr[x] + cor,  x ← z,  is_levroot(x) ← false
22         else     f(z) < f(y) :
23             par[x] ← y
24             new_cor ← attr[x] ,  attr[x] ← attr[x] + cor ,  cor ← new_cor
25             x ← y ,  is_levroot(x) ← true ,  y ← z
26         z ← par[x]
27     if x ≠ y then
28         par[x] ← y
29         new_cor ← attr[x] ,  attr[x] ← attr[x] + cor ,  cor ← new_cor
30         x ← y ,  is_levroot(x) ← true ,  z ← par[x]
31         while par[x] ≠ x do
32             if is_levroot(z) then  par[x] ← z,  attr[x] ← attr[x] + cor,  x ← z,  is_levroot(x) ← true
33             z ← par[z]
34         attr[x] ← attr[x] + cor
```

moments, the pixels are removed from the top of the stack and a parent pointer is defined for them. This algorithm is used to build trees of image partition lines in the overall algorithm. This operation is called "Build".

5.1.2 Progressive CCT fusion algorithm

After the parent point trees of individual lines have been placed into memory side by side, they are separate trees because they respect only horizontal connectivity. It is therefore necessary to modify some parent pointers to reflect also the vertical (generally other) connectivity. This means to perform a Connect(u, v) operation for each pair of neighboring nodes $\{u, v\}$ where each node u, v was originally in a different tree. Performing all Connect operations for a pair of trees is called "Merge".

The Connect operation is described in Algorithm 5.2. It follows the two parent pointer chains starting in u and v up to the point where they meet or to the roots and modifies the parent pointers on the way so that they become a single chain.

5.1.3 Dataflow representation of overall CCT construction algorithm

The overall parent point tree construction algorithm can be efficiently described by a dataflow diagram shown in Figure 5.1. It shows a generic division of a very small image into $P = 4$ partitions for parallel processing, each consisting of 4 lines. It also shows a mapping of the Build and Merge operations to hardware execution units and to RAM blocks. These hardware components will be described in the following section. Processing phases are also labeled.

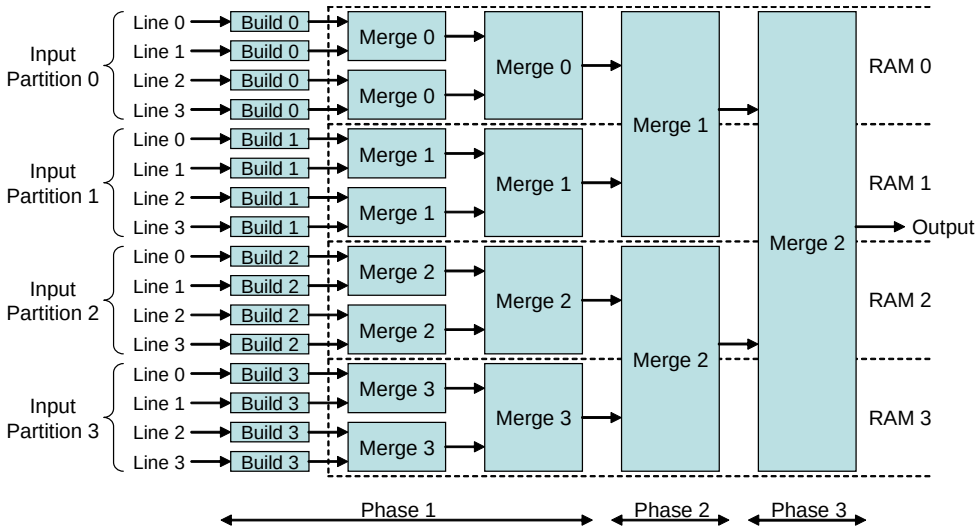


Figure 5.1: CCT construction dataflow diagram

5.2 Original parallel hardware architecture description

5.2.1 Overall architecture

The tree merging operates on large data structures, especially in the late stages, but in most cases it accesses only tree nodes "close" to the partition boundary. It is therefore undesirable to transfer the trees from one unit to another, because tree merging cannot take place during such data transfers. For this reason, the trees are kept in the same memories during all stages of merging and different Merge units are allowed to access them as needed.

The architecture is shown in Figure 5.2 and consists of P basic blocks, of a common interconnection network, and of a global controller. The basic block is an elementary structure, which performs the processing. Scaling of the design is done by repeating of the basic blocks. The basic block is composed of execution units (one Build unit and one Merge unit), memories (two FIFOs and a CCT RAM block, which stores the image, the parent point tree and attributes for the corresponding partition), and intra-block interconnections.

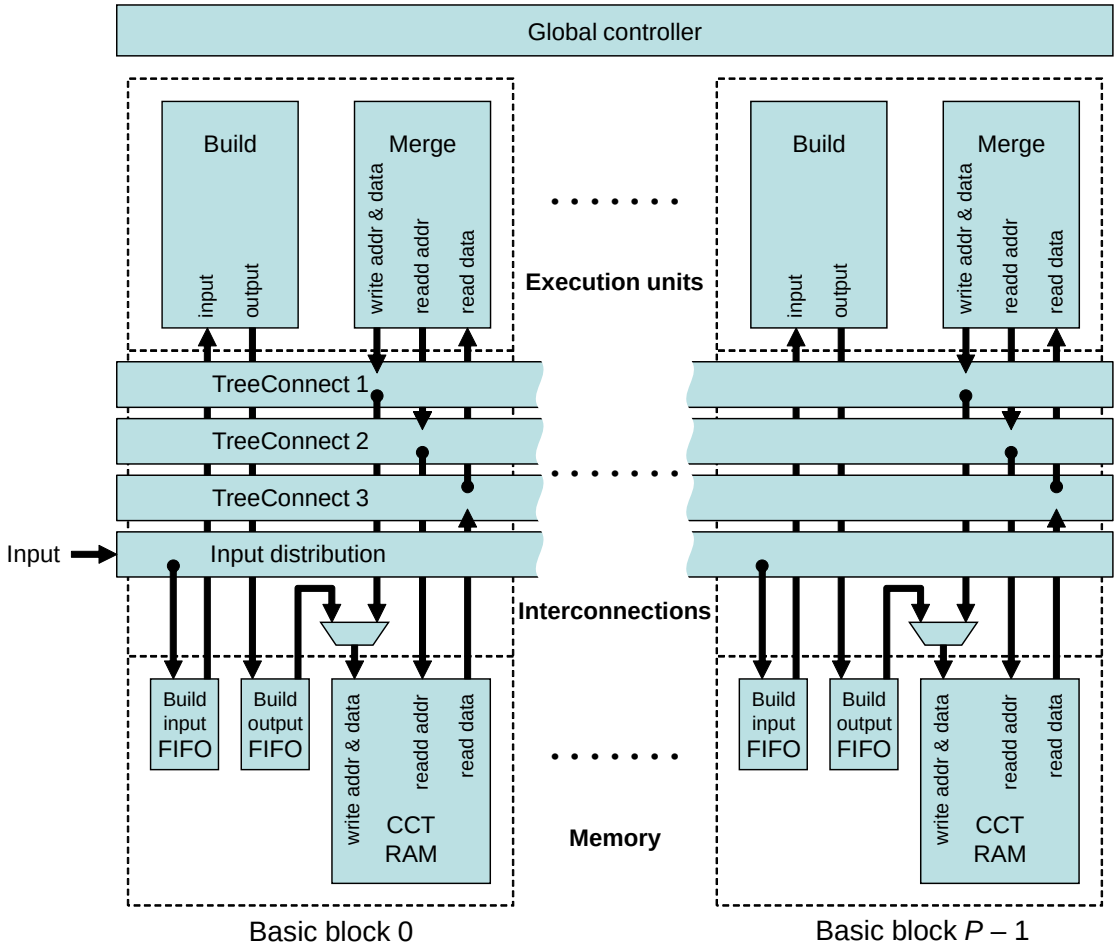


Figure 5.2: Overall structure of the new CCT construction architecture

Signals used by the Merge unit to access the CCT RAM in its basic block are routed through the common interconnection network to allow some Merge units to access also the CCT RAMs in other basic blocks.

The common interconnection network consists of three TreeConnect switches described in Section 5.2.5 and of an input distribution demultiplexer. The Merge unit interfaces to the CCT RAM through three groups of signals: 1) writing address and data, 2) reading address, and 3) read data. Each of these signal groups is routed through a different TreeConnect switch because datapaths for different groups of signals are configured independently and they are used at the same time. Writing address and data are routed through TreeConnect 1, reading address through TreeConnect 2, and read data through TreeConnect 3.

The global controller tells each unit what it should do at a given time. Mainly it ensures that the next merging phase does not start before all merging units completed the current phase.

The input image has to be divided into P narrow partitions (columns) to be able to process the partitions concurrently, each by a different basic block. This is done by the input distribution demultiplexer, which breaks the lines of the input image into line fragments corresponding to the image partitions and feeds them into the Build input FIFOs of the corresponding blocks.

The Build unit described in Section 5.2.2 reads intensity values in a stream from the Build input FIFO and processes them. It outputs the tree nodes in a data dependent order, so the tree nodes are stored together with their addresses in the Build output FIFO. They are transferred from there to the CCT RAM during cycles not used by the Merge unit for writing.

The merging process is started as soon as the first two lines have been stored in the CCT RAM. The Merge unit described in Section 5.2.3 reads the tree nodes needed by its algorithm from the CCT RAM using the reading signals and writes back tree modifications using the writing signals.

When all lines inside all partitions have been merged together then the first phase of merging is complete. During the second phase (see again the example in Figure 5.1), a half of the Merge units is deactivated and partition pairs are merged by the remaining $P/2$ units. Each of these $P/2$ units needs to access both CCT RAMs of the corresponding partition pair. The next merging phase is performed by $P/4$ units (each of them accesses the CCT RAMs of 4 consecutive basic blocks) and so on. There are $(1 + \log_2 P)$ phases of merging in total. In each phase, each CCT RAM block is accessible by precisely one Merge unit.

Generally said, in merging phase $(i + 1)$, where $i \in \{0, ..., \log_2 P\}$, for each $j \in \{0, ..., P/2^i - 1\}$ there is a single Merge unit, which has access to partitions $2^i j$ through $2^i(j + 1) - 1$. The choice of the particular Merge unit, which accesses each partition group, is shown in Table 5.1. It was made with the aim of making the TreeConnect switches as simple as possible.

Table 5.1: Which Merging units are active in each merging phase (for 16 basic blocks). Each rectangle represents a group of partitions; the number inside indicates which unit does the merging

Phase 1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Phase 2		1	2			5	6			9	10			13	14	
Phase 3			2				6				10				14	
Phase 4					4								12			
Phase 5									8							

5.2.2 Build

The internal structure of the Build unit is shown in Figure 5.3. The address counter corresponds to the algorithm variable i. The registers x, p, z correspond to the variables x, v_i, z respectively. The core of the unit is a comparator of $f(x)$ and $f(v_i)$. The result of the comparison decides the flow of data throughout the entire unit.

The Build unit is controlled by a small Mealy machine whose state covers the program counter and the *next_parent* variable components of the algorithm state. A mapping of the algorithm states to the controller states is shown in Table 5.2. The unit executes multiple lines of the algorithm pseudocode in one clock cycle and therefore not all program counter values correspond to a particular controller state.

The controller's state transition diagram is shown in Figure 5.4. In a given state, the controller tests the indicated condition and if it is fulfilled, it performs the corresponding state transition on the next clock.

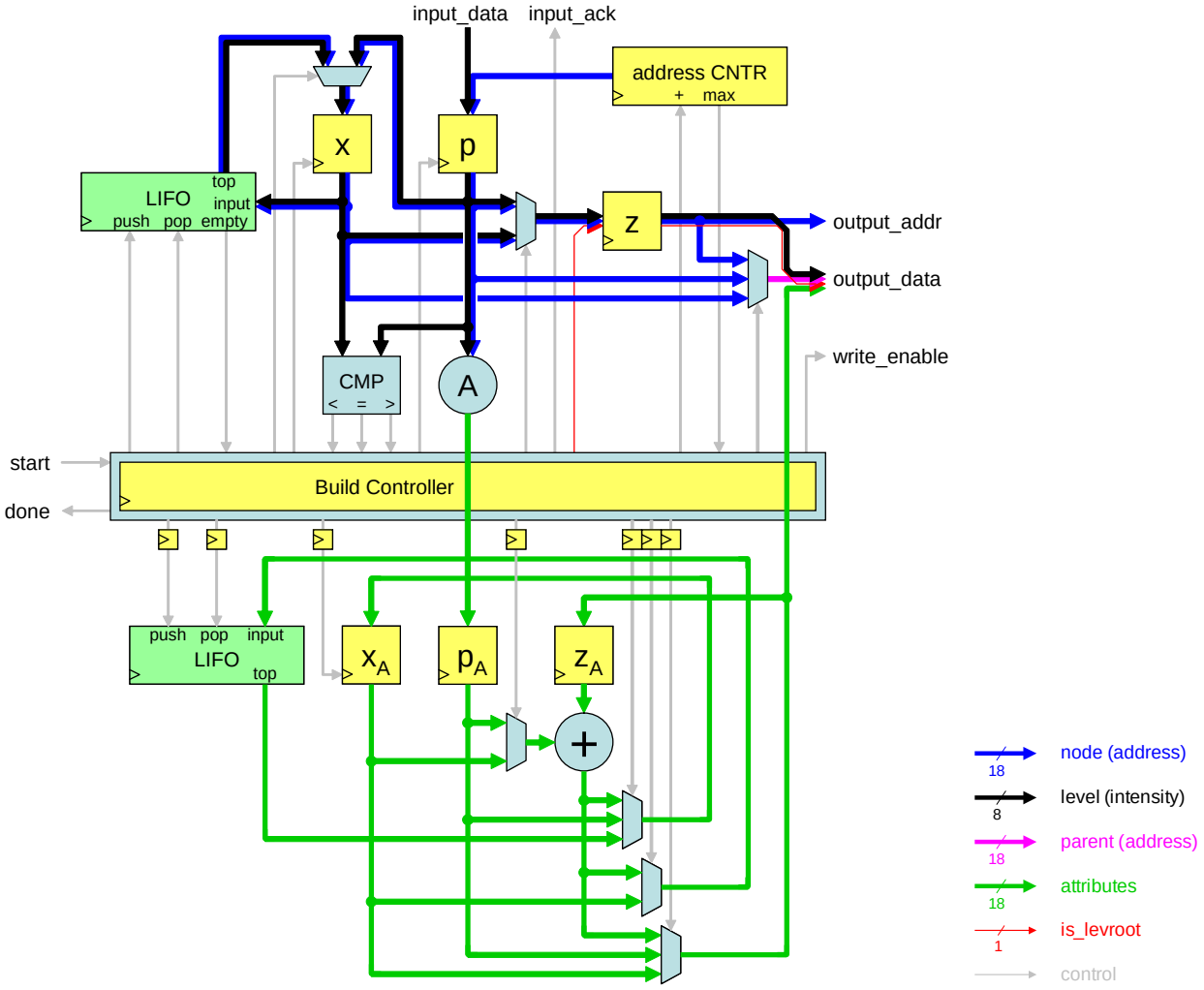


Figure 5.3: 1D CCT building unit RTL structure

Table 5.2: Building algorithm and controller states

Algorithm state		
Before line no.	*next_parent*	**Controller state**
3	NONE	IDLE
5	NONE	NONE
5	X_ONLY	X_ONLY
5	X_OR_P	X_OR_P
19	NONE	FLUSH1
20	*any value*	FLUSH2

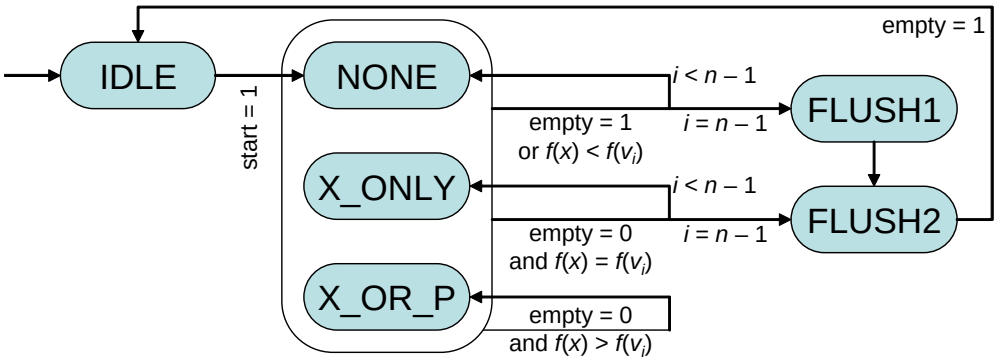


Figure 5.4: Build controller state transition diagram

5.2.3 Merge

The most important part of the Merge unit is shown in Figure 5.5. It is composed of registers, multiplexers, comparators and of a controller. It performs one Connect(u, v) operation at a time. The two nodes u, v to be connected are specified by corresponding address inputs.

The Merge unit reads the tree nodes that it needs from the CCT RAM using the reading signals and writes back tree updates using the writing signals. The registers x, y and the signal z correspond to respective variables. Note that the intensity values do not change and therefore need not to be written to the CCT RAM again. The corresponding data paths can therefore be disconnected from the RAMs and an HDL synthesizer will automatically remove all unneeded data paths and registers.

The Merge unit is controlled by another Mealy machine whose state corresponds to the program counter component of the algorithm state. The mapping of the algorithm states to the controller states is shown in Table 5.3 and the state transition diagram is shown in Figure 5.6.

A merging acceleration using a path compression [Wilkinson 2008] in visited chains of non-level-root nodes was not implemented yet to reduce the development time. Simulations show that its absence does not incur a significant performance loss. Nevertheless, the path compression may run concurrently with already

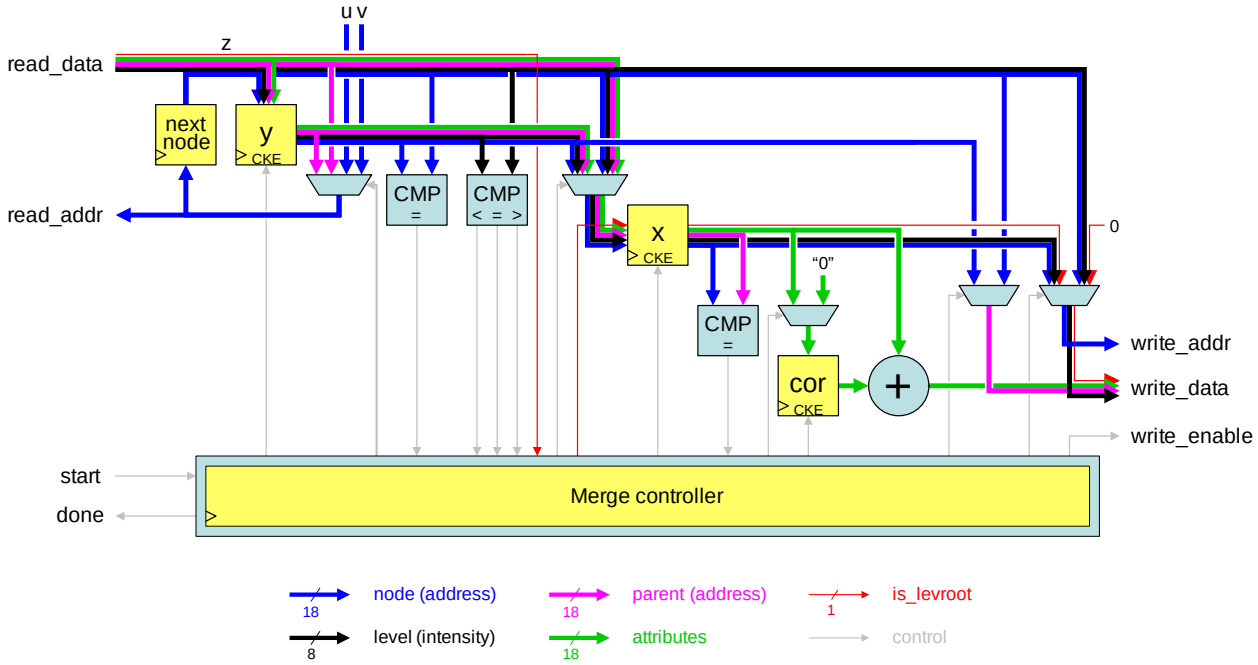


Figure 5.5: CCT merging unit RTL structure

Table 5.3: Merging algorithm and controller states

Algorithm state Before line no.	**Controller state**
2	IDLE
3	START1
7	START2
15	NEXT_CMP
16	FIND_LR

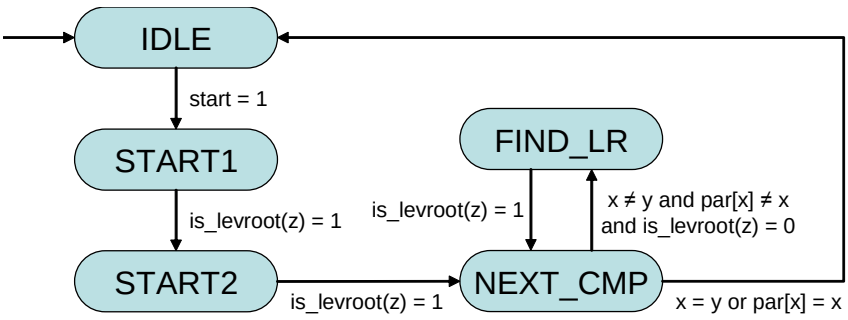


Figure 5.6: Merge controller state transition diagram

implemented main merging. Its write accesses to the CCT RAM will be multiplexed with main merging write accesses, which will have the higher priority. The path compression will therefore use only otherwise free memory cycles and its later implementation will not reduce the main merging performance.

The time-critical path, which limits the maximal frequency of this architecture, is the combinational path from the read_data input through the intensity comparator, controller and multiplexer to the read_addr output. The path is closed into a sequential loop through interconnections and the synchronous CCT RAM.

A cascade of counters (not shown in Figure 5.5) generates the u and v addresses for all Connect(u, v) operations required by a given merging phase. The line number components of these addresses have to be generated in a special order required by the CCT construction algorithm. This is done by a special counter described in Section 5.2.4.

5.2.4 Merging boundary line numbers generator

The merging units need coordinates of point pairs (u, v) to perform the `connect` operation on as an input. The merging is done in the order prescribed by Strategy 2. Inside of the blocks it is done according to Algorithm 4.3 with $P = 1$. Each iteration of the algorithm computes the variable *border* (the merging boundary line number, illustrated in Figure 4.4) and performs merging of the trees across the boundary between lines with numbers $border - 1$ and $border$. The purpose of the merging boundary line numbers generator is to produce these line numbers.

The generator is a special synchronous binary counter consisting of bits $border_{\log_2 H - 1}$ (MSB) to $border_0$ (LSB). Upon reset, the counter is initialized to decimal value 1, which is the first member of the counter's

Table 5.4: Merging boundary line numbers generator (4-bit) output sequence

	border	
index	dec	bin
0	1	0001
1	3	0011
2	2	0010
3	5	0101
4	7	0111
5	6	0110
6	4	0100
7	9	1001
8	11	1011
9	10	1010
10	13	1101
11	15	1111
12	14	1110
13	12	1100
14	8	1000

output sequence shown in Table 5.4. The next member $border'$ is given by the current member $border$ according to the following formulas:

$$border'_0 = \begin{cases} 0 & \text{if } border \text{ ends with “1}\underbrace{00\ldots0}_{\geq 0}\text{" (LSB on the right),} \\ 1 & \text{else.} \end{cases}$$

$$\text{For } i \geq 1,\ border'_i = \begin{cases} border_{i-1} & \text{if } \forall j \in \{0,\ldots,i-2\} : border_j = 0, \\ border_i & \text{else.} \end{cases}$$

5.2.5 Memory access network

The TreeConnect switches allow the Merge units to access the CCT RAM blocks that they need according to Table 5.1. TreeConnects 1 and 2 route the signals from the Merges to the RAMs. Their internal structure, which is identical for both units, is shown in Figure 5.7 for 16 basic blocks. TreeConnect 3 routes the read data from the RAMs to the Merges. Its internal structure, which is shown in Figure 5.8, is different from TreeConnects 1 and 2 because it routes signals in the opposite direction.

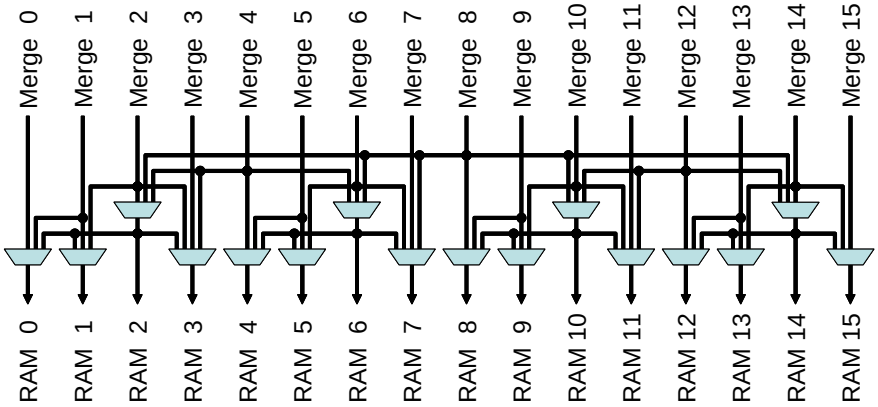


Figure 5.7: TreeConnect 1 and 2 for 16 basic blocks

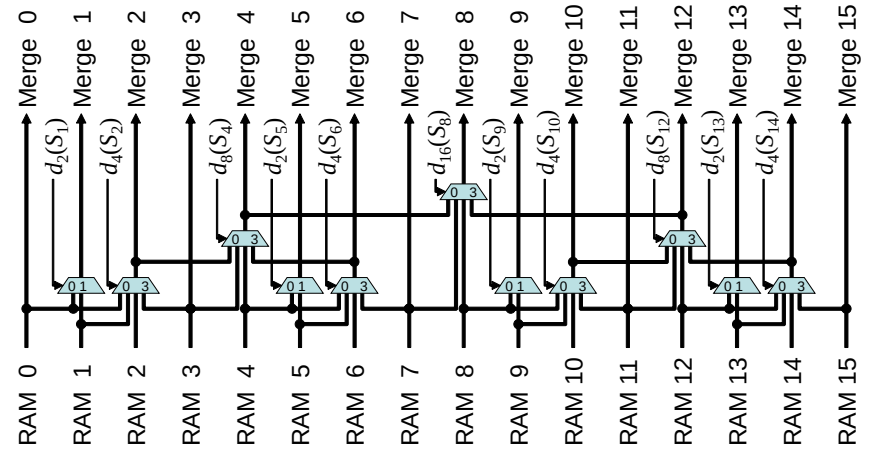


Figure 5.8: TreeConnect 3 for 16 basic blocks

Table 5.5: TreeConnect 1 and 2 multiplexers control

	Phase 1	**Phase 2**	**Phase 3**	**Phase 4**	**Phase 5**
RAM 0 =	Merge 0	Merge 1	RAM 2	RAM 2	RAM 2
RAM 1 =	Merge 1	Merge 1	Merge 2	RAM 2	RAM 2
RAM 2 =	Merge 2	Merge 2	Merge 2	Merge 4	Merge 8
RAM 3 =	Merge 3	Merge 2	Merge 2	Merge 4	RAM 2
RAM 4 =	Merge 4	Merge 5	RAM 6	Merge 4	RAM 6
RAM 5 =	Merge 5	Merge 5	Merge 6	RAM 6	RAM 6
RAM 6 =	Merge 6	Merge 6	Merge 6	Merge 4	Merge 8
RAM 7 =	Merge 7	Merge 6	Merge 6	RAM 6	Merge 8
RAM 8 =	Merge 8	Merge 9	RAM 10	RAM 10	Merge 8
RAM 9 =	Merge 9	Merge 9	Merge 10	RAM 10	RAM 10
RAM 10 =	Merge 10	Merge 10	Merge 10	Merge 12	Merge 8
RAM 11 =	Merge 11	Merge 10	Merge 10	Merge 12	RAM 10
RAM 12 =	Merge 12	Merge 13	RAM 14	Merge 12	RAM 14
RAM 13 =	Merge 13	Merge 13	Merge 14	RAM 14	RAM 14
RAM 14 =	Merge 14	Merge 14	Merge 14	Merge 12	Merge 8
RAM 15 =	Merge 15	Merge 14	Merge 14	RAM 14	RAM 14

Table 5.6: Definition of addresses S_i for TreeConnect 3 multiplexers control. A_i is the read address issued by Merge unit i in the previous clock cycle. The values labeled with * have no actual effect, because the outputs of the corresponding multiplexers are routed only to deactivated Merge units

	Phase 1	**Phase 2**	**Phase 3**	**Phase 4**	**Phase 5**
$S_1 =$	A_1	A_1	A_1*	A_1*	A_1*
$S_2 =$	A_2	A_2	A_2	A_4	A_8
$S_4 =$	A_4	A_4*	A_4*	A_4	A_8
$S_5 =$	A_5	A_5	A_5*	A_5*	A_5*
$S_6 =$	A_6	A_6	A_6	A_4	A_8
$S_8 =$	A_8	A_8*	A_8*	A_8*	A_8
$S_9 =$	A_9	A_9	A_9*	A_9*	A_9*
$S_{10} =$	A_{10}	A_{10}	A_{10}	A_{12}	A_8
$S_{12} =$	A_{12}	A_{12}*	A_{12}*	A_{12}	A_8
$S_{13} =$	A_{13}	A_{13}	A_{13}*	A_{13}*	A_{13}*
$S_{14} =$	A_{14}	A_{14}	A_{14}	A_{12}	A_8

The TreeConnects are configured differently for each merging phase, according to the set of active Merge units. An active Merge unit can select a different RAM block from the group of RAM blocks accessible to it in each clock cycle.

The multiplexers in TreeConnects 1 and 2 are controlled by the selected phase according to Table 5.5. The multiplexers in TreeConnect 3 are controlled by a value derived from the reading address issued in the previous clock cycle by the Merge unit given by the selected phase. The values of the addresses S_1 through S_{14} are defined by Table 5.6. They could however be taken also from the address components of TreeConnect 2 outputs delayed by one cycle. The functions d_2 through d_{16} are defined in Table 5.7, where the expression $partition(S)$ is the number of partition, which contains the address S.

Two objectives were achieved in the design of the TreeConnect switches: Low logic utilization and small logic depth. For 16 basic blocks, TreeConnects 1 and 2 contain 16 multiplexers each and TreeConnect 3 contains only 11 multiplexers, all with only 2 to 4 inputs. Maximum logic depth for TreeConnects 1 and 2 is 2 multiplexers, for TreeConnect 3 it is 3 multiplexers. Additionally, during every Merge operation, the two RAM blocks along a merging boundary, which are accessed the most often, are always accessed through only one multiplexer. This is important because it is the memory latency what limits the merging performance.

Table 5.7: Functions d_2, d_4, d_8, d_{16}

$d_2(S) = partition(S) \bmod 2$																
$d_4(S) = partition(S) \bmod 4$																
$partition(S) \bmod 8$	0	1	2	3	4	5	6	7								
$d_8(S)$	0	0	0	1	2	3	3	3								
$partition(S) \bmod 16$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$d_{16}(S)$	0	0	0	0	0	0	0	1	2	3	3	3	3	3	3	3

5.3 Scalability discussion

The new architecture was implemented with $P = 16$ basic blocks for the image size $W \times H = 512 \times 512$ px, $n = 8$ bits per pixel, with the area as a connected component attribute. However, it can be modified to change any of these parameters. This section discusses the resource utilization and performace trends with respect to these parameters.

The number of basic blocks P (which must be a power of 2) affects mainly the internal structure of the TreeConnect switches. There are always 3 TreeConnect switches. While the number of inputs per multiplexer does not exceed 4, the number of multiplexers would grow linearly with P. The maximum logic depth would grow logarithmically with P and thus the maximum clock frequency would decrease. However, the depth of the most frequently used data-paths is constant, equal to one multiplexer. If some logic is added, which delays the transfers through other paths by one or more cycles, it will be possible to ignore these paths during a timing analysis, and thus eliminate this performance degradation source for the most frequently used data-paths. The CCT RAM block size is proportional to $1/P$, as the pixels are distributed among the basic blocks. Other components of the architecture are not affected significantly by P.

The image width W and height H (which must be powers of 2 as well for this implementation) affect the total CCT RAM size and the address width, which equals to $p = \log_2(WH)$. The width a of the implemented attribute (i.e. area) is the same as of the address.

The image bit depth n gives the width of the corresponding buses and affects the size of all memories, because the input data are stored as well.

The total number of bits stored per pixel is therefore $o = n + p + a + 1 = n + 1 + 2\log_2(WH)$ and the total CCT RAM size equals to WHo bits.

5.4 Performance evaluation and implementation results

The architecture was implemented in VHDL with $P = 16$ basic blocks. It turned out that one Build unit is able to feed at least two Merge units with sufficient data, so only 8 Build units were used (one per two basic blocks). The design was synthesized into an FPGA Xilinx Virtex 5 XC5VTX240T-1. The measured performance coming from simulations is summarized in Table 5.8 for a natural photograph and for a synthetic image designed specifically to obtain a very long processing time, even though it is still not a worst case input.

Resource utilization in the target FPGA is shown in Table 5.9. It indicates the amount of resources occupied by a single Build and Merge unit and by all Build and Merge units and a total for the architecture including memories and interconnections. The Build unit occupies no block memory because its LIFO is so small that it can be efficiently implemented in slice LUTs. The Merge unit does not contain any memory at all.

Figure 5.9 shows the percentage of building and merging units, which are doing useful work at each moment of the natural image processing.

Table 5.8: Measured performance in FPGA

Image size	512×512 px, 8 bits	
Image type	natural	synthetic
Total cycles	216 014	928 006
Mean cycles per pixel	0.82	3.54
Mean Merge utilization	78 %	75 %
Maximum clock frequency[a]	120 MHz	
Mean performance[b]	145 Mpx/s	34 Mpx/s
Processing time[b]	1.8 ms	7.7 ms

(a) Post-synthesis timing estimate

(b) Derived from total cycles and maximum clock frequency

The following inequality gives an upper bound on the worst case number of cycles $T_{\max}$ required to process any input image with total size $W \times H$ on the new architecture with P basic blocks under the assumption that $W/P \geq 4$ and $H \geq 4$:

$$T_{\max} < \frac{2HW^2}{P^2} \log_2 H + 4H^2W\frac{P-1}{P}$$

Table 5.10 compares the performance of the new architecture to other CCT implementations. The Wilkinson's algorithm [Wilkinson 2008] was benchmarked on a concurrent elongation-attribute filtering of a 3D 8bit data set of 512^3 voxels and it ran on a 2× dual-core Opteron 280 machine. The new parallel software implementation from the previous chapter (described also in [Matas 2008]) constructed the CCT of

Table 5.9: FPGA resource utilization for a 512×512px image

	1 Build	**1 Merge**	**8 Builds**	**16 Merges**	**Total including interconnections**
Slice LUTs	106	500	849 (< 1 %)	8 003 (5 %)	11 928 (8 %)
Registers	56	289	448 (< 1 %)	4 620 (3 %)	5 752 (4 %)
Block memory	0	—	0 (0 %)	—	8 064 Kib (69 %)

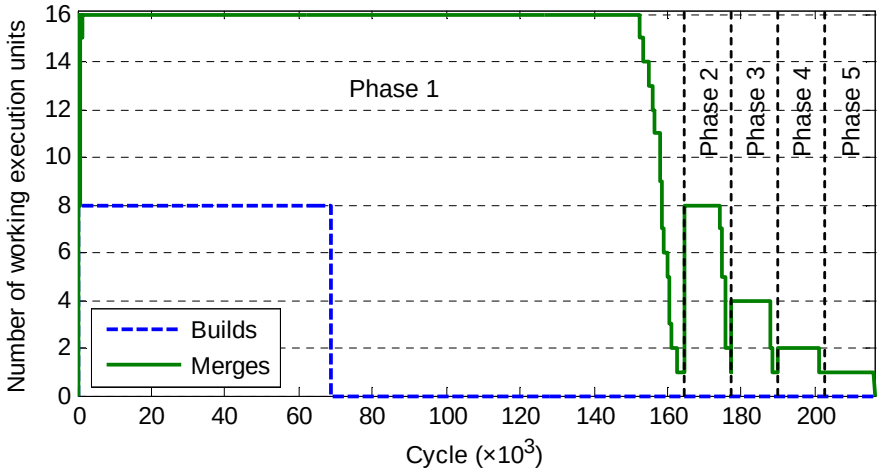


Figure 5.9: Execution units' utilization versus time (natural image)

Table 5.10: Performance comparison of CCT implementations

	Computation resources	**Performance**
[Wilkinson 2008]	4 CPU cores @ 2.4 GHz	9.8 Mpx/s
[Matas 2008]	16 CPU cores @ 2.6 GHz	83 Mpx/s
[Ngan 2007, Ngan 2011]	Sequential HW @ 50 MHz	2.7 Mpx/s
New architecture	**16 basic blocks @ 120 MHz**	**145** Mpx/s

a 784×576px 8bit image and it ran on an 8× dual-core Opteron 885 machine. The hardware architecture [Ngan 2007, Ngan 2011] performed a sequential attribute filtering of a 160×120px 8bit image.

5.5 Conclusions

This chapter presented the new parallel hardware architecture for the CCT construction. The architecture was implemented in the VHDL hardware description language and verified by a functional simulation. Its FPGA implementation with 16 basic blocks clocked at 120 MHz would give the performance of 145 Mpx/s on a 512×512px 8bit image. It is 70 % faster than the software implementation ([Matas 2008], introduced in the previous chapter) running on a 16-core machine and better suited for embedded applications. It is also 50 times faster than the only existing special hardware architecture for CCT [Ngan 2007, Ngan 2011] and it requires about 4 times less memory per pixel.

Future work should focus on further reduction of FPGA memory occupation using a hierarchical memory and on the acceleration of the merging.

Chapter 6

Conclusions and perspectives

This work suggested a new specific approach to computer vision system design. Chapter 2 introduced the data structure for digital image representation called connected component tree (CCT), which allows an efficient implementation of many low- and high-level image processing operators. The set of operators available using CCT computations is so large, that it can support complete image processing chains of many applications. This is a big advantage, because a hardware optimized for CCT computations can achieve high performance without losing its flexibility. It is therefore a promising way, which can lead to flexible and powerful embedded computer-vision systems.

A CCT-based application consists of the CCT construction, a cascade of CCT transformations, and image restitution. The CCT construction, which consumes about 80 % of the execution time in existing applications, is the performance bottleneck of existing applications and this work is focused on it. In Chapter 3 the existing CCT construction algorithms were inspected and classified. Most of them are sequential, require a big amount of memory, or both, so they are not well suited for hardware implementation.

Chapter 4 introduced the new parallel CCT construction algorithm, which constructs 1D tree of individual lines and then it merges them into the resulting global tree. The performance measurements show that it is even faster than the Wilkinson's parallel algorithm, which was the fastest one to date. Furthermore, it is better suited for hardware implementation thanks to its simplicity and low memory requirements.

In Chapter 5, this CCT construction algorithm was further adapted for hardware implementation and a new parallel hardware architecture for the CCT construction was created. The architecture core consists of 16 basic blocks, each containing memory and execution units for performing the tree construction. The basic blocks are connected by three special switches which allow some of the execution units to access memory in other basic blocks. The switches were designed carefully to keep their logic utilization and depth (and delay in turn) low, as the memory access latency has a critical impact on the architecture's performance. The hardware implementation further improved the algorithm performance by 70 % compared to the software implementation.

To create a complete application, further work should focus on CCT-based hardware implementation of some CCT-based operators, for example filtering and segmentation.

Publications

1. P. Matas, E. Dokladalova, M. Akil, T. Grandpierre, L. Najman, M. Poupa, V. Georgiev. Parallel algorithm for concurrent computation of connected component tree. In: *Advanced Concepts for Intelligent Vision Systems* – proceedings of the 10th International Conference, ACIVS 2008, Juan-les-Pins, France, October 20–24, Lecture Notes in Computer Science, LNCS 5259, pp. 230–241. Springer-Verlag Berlin Heidelberg, 2008.
2. R. Mahmoudi, M. Akil, P. Matas. Parallel image thinning through topological operators on shared memory parallel machines. In: *Signals, Systems and Computers, 2009 Conference Record of the Forty-Third Asilomar Conference on.* Pacific Grove, California. pp. 723–730. IEEE, 2009. ISBN 978-1-4244-5825-7.
3. P. Matas, E. Dokladalova, M. Akil, V. Georgiev, M. Poupa. Parallel Hardware Implementation of Connected Component Tree Computation. *Proceedings 2010 International Conference on Field Programmable Logic and Applications FPL 2010.* pp. 64–69. IEEE, 2010.

Presentations

1. P. Matas. Parallel Algorithm for Concurrent Computation of Connected Component Tree. Atelier doctorants. ESIEE Paris, 25 March 2010.

Bibliography

[Berger 2005] C. Berger, N. Widynski 2005. Using connected operators to manipulate image components. *Technical Report no. 0517*, revision 884. Laboratoire de Recherche et Développement de l'Epita, July 2005.

[Berger 2007] C. Berger, T. Géraud, R. Levillain, N. Widynski, A. Baillard, E. Bertin. Effective component tree computation with application to pattern recognition in astronomical imaging. *ICIP07*, vol. IV, pp. 41–44, 2007.

[Chiang 2005] Chiang, Y.-J., Lenz, T., Lu, X., and Rote, G., Simple and optimal output sensitive construction of contour trees using monotone paths. *Comp.Geometry: Theory and Applications*, 30(2):165–195, 2005.

[Couprie 1997] M. Couprie, G. Bertrand. Topological Grayscale Watershed Transformation. *SPIE Vision Geometry VI Proceedings*, Vol. 3168, pp. 136–146, 1997.

[Couprie 2005] M. Couprie, L. Najman, G. Bertrand. Quasi-linear algorithms for the topological watershed. *Journal of Mathematical Imaging and Vision*, Volume 22, Issue 2 - 3, May 2005, pp. 231–249.

[cs.cf.ac.uk] http://www.cs.cf.ac.uk/Parallel/Year2/section7.html, retrieved 26 March 2009.

[Cuisenaire 1999] O. Cuisenaire and E. Romero. Automatic segmentation and measurement of axones in microscopic images. In SPIE Medical Imaging, volume 3661 of Lecture Notes in Computer Science, pp. 920–929. IEEE, 1999.

[Deloison 2007] B. Deloison. Recherche et développement en traitement d'image : Utilisation de l'arbre des composantes pour la fusion d'images. Internship report, ESIEE Paris, 2007.

[Galaghba] http://carbon.cudenver.edu/~galaghba/speedup.htm, retrieved 26 March 2009.

[Herlihy 2008] Maurice Herlihy, Nir Shavit. The Art of Multiprocessor Programming. Elsevier, 2008. ISBN 978-0-12-370591-4.

[Jalba 2004] A.C. Jalba, M.H.F. Wilkinson, and J.B.T.M. Roerdink. Morphological hat-transform scale spaces and their use in pattern classification. Pattern Recognition, 37(5):901–915, May 2004.

[Levillain 2006] R. Levillain. Add a third version of the computation of a the max-tree based on Fiorio's and Gustedt's labelling algorithm, oln-0.10 10.256. *Olena-patches – patches for the Olena project*, August 29, 2006. Available at: https://www.lrde.epita.fr/pipermail/olena-patches/2006-August/000898.html. Accessed May 5, 2010.

[Mahmoudi PhD] Ramzi Mahmoudi. Common parallelization strategy of topological operators on SMP machines. PhD thesis, p. 22, Université Paris-Est, MSTIC, 2011.

[Matas 2008] P. Matas, E. Dokladalova, M. Akil, T. Grandpierre, L. Najman, M. Poupa, V. Georgiev. Parallel algorithm for concurrent computation of connected component tree. In: *Advanced Concepts for Intelligent Vision Systems* – proceedings of the 10th International Conference, ACIVS 2008, Juan-les-Pins, France, October 20–24, Lecture Notes in Computer Science, LNCS 5259, pp. 230–241. Springer-Verlag Berlin Heidelberg, 2008.

[Matas 2010] P. Matas, E. Dokladalova, M. Akil, V. Georgiev, M. Poupa. Parallel Hardware Implementation of Connected Component Tree Computation. *Proceedings 2010 International Conference on Field Programmable Logic and Applications FPL 2010*. pp. 64–69. IEEE, 2010.

[Mattes 1999] Julian Mattes, Mathieu Richard, and Jacques Demongeot. Tree representation for image matching and object recognition. In DCGI '99: Proceedings of the 8th International Conference on Discrete Geometry for Computer Imagery, pp. 298–312, London, UK, 1999. Springer-Verlag.

[Mattes 2000] Mattes, J., Demongeot, J., Efficient algorithms to implement the confinement tree. *In LNCS:*1953, pp. 392–405. Springer, 2000.

[Meijster PhD] A. Meijster. Efficient Sequential and Parallel Algorithms for Morphological Image Processing. PhD thesis, Rijksuniversiteit Groningen, 2004.

[Menotti 2007] D. Menotti, L. Najman, A. de Albuquerque Araújo. 1D Component Tree in Linear Time and Space and its Application to Gray-Level Image Multithresholding. *Proceedings of the 8th International Symposium on Mathematical Morphology*, Rio de Janeiro, Brazil, Oct. 10–13, 2007, MCT/INPE, v. 1, pp. 437–448.

[MetaCentrum] MetaCentrum National Grid Infrastructure. http://www.metacentrum.cz

[Najman 2006] L. Najman and M. Couprie. Building the component tree in quasi-linear time. *IEEE Transactions on Image Processing*, 15/11: 3531–3539, 2006.

[Neerbos 2011] J. van Neerbos, L. Najman, M.H.F. Wilkinson. Towards a Parallel Topological Watershed: First Results. In: *Mathematical Morphology and Its Applications to Image and Signal Processing.* 10th International Symposium, ISMM 2011, Verbania-Intra, Italy, July 6-8, 2011. Proceedings. Lecture Notes in Computer Science, Volume 6671. pp. 248–259. Springer Berlin Heidelberg, 2011.

[Ngan 2007] N. Ngan, F. Contou-Carrère, B. Marcon, S. Guérin, E. Dokládalová, M. Akil. Efficient hardware implementation of connected component tree algorithm. *Workshop on Design and Architectures for Signal and Image Processing, DASIP 2007*, Grenoble, France.

[Ngan 2011] N. Ngan, E. Dokladalova, M. Akil, F. Contou-Carrère. Fast and efficient FPGA implementation of connected operators. In: *Journal of Systems Architecture* 57, 8 (2011). pp. 778–789. Elsevier, 2011. DOI 10.1016/j.sysarc.2011.06.002.

[PerfSuite] PerfSuite performance analysis software. http://perfsuite.ncsa.uiuc.edu/

[Piscaglia 1999] Patrick Piscaglia, Andrea Cavallaro, Michel Bonnet, and Damien Douxchamps. High level description of video surveillance sequences. In EC-MAST '99: Proceedings of the 4th European Conference on Multimedia Applications, Services and Techniques, pp. 316–331, London, UK, 1999. Springer-Verlag.

[Robison 2007] Arch Robison: Why Too Many Threads Hurts Performance, and What to do About It; sample chapter, 6 April 2007. http://www.codeguru.com/cpp/sample_chapter/article.php/c13533, retrieved 14 April 2009.

[Salembier 1998] P. Salembier, A. Oliveras, L. Garrido. Anti-extensive connected operators for image and sequence processing. *IEEE Trans. on Image Proc.*, 7(4): 555–570, April 1998.

[Sonka 2008] M. Šonka, V. Hlaváč, R. Boyle. Image processing, analysis, and machine vision, 3rd ed. Toronto, 2008. Thomson.

[Tarjan 1975] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22: 215–225, 1975.

[Watershed wiki] Wikipedia contributors. Watershed (image processing), section Inter-pixel watershed. Wikipedia, The Free Encyclopedia. December 28, 2011, 15:03 UTC. Available at: http://en.wikipedia.org/w/index.php?title=Watershed_(image_processing)&oldid=468093320#Inter-pixel_watershed. Accessed February 16, 2012.

[Wilkinson 2001] M.H.F. Wilkinson, M.A. Westenberg. Shape Preserving Filament Enhancement Filtering. *MICCAI 2001*, LNCS 2208, pp. 770–777, 2001. Springer-Verlag.

[Wilkinson 2008] M.H.F. Wilkinson, H. Gao, W.H. Hesselink, J. Jonker, A. Meijster. Concurrent Computation of Attribute Filters on Shared Memory Parallel Machines. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 10, pp. 1800–1813, Oct. 2008.

Appendix A

Inter-pixel watershed algorithm

Algorithm A.1: Inter-pixel watershed

Input: Image domain V (set of image points)
Input: Pixel neighborhood mapping $E : V \to 2^V$
Input: Image function $f : V \to \mathbb{R}$
Output: Number of regions $count \in \mathbb{N}_1$
Output: Region mapping $labels : V \to \{1, ..., count\}$
Output: Region separation values $separation : \{1, ..., count\} \times \{1, ..., count\} \to \mathbb{R}$

Initialize elements of $labels$ to UNASSIGNED
$count \leftarrow 0$
$minima \leftarrow$ `FindLocalMinima`(V, E, f) // *Algorithm A.2*
$queue \leftarrow \bigcup_{floor \in minima} floor$
for each $floor \in minima$ **do**
    $count$++
    for each $p \in floor$ **do**
        $labels[p] \leftarrow count$
Initialize elements of $separation$ to $+\infty$
while $queue \neq \emptyset$ **do**
    $p \leftarrow$ Extract and remove from $queue$ the oldest element with minimal f
    for each $q \in E(p)$ **do**
        if $labels[q] =$ UNASSIGNED **then**
            $labels[q] \leftarrow labels[p]$
            Insert q into $queue$
        else if $labels[q] \neq labels[p] \wedge f(q) \geq f(p) \wedge separation[labels[p], labels[q]] > f(q)$ **then**
            $separation[labels[p], labels[q]] \leftarrow f(q)$
            $separation[labels[q], labels[p]] \leftarrow f(q)$

Algorithm A.2: Local minima of an image

Input: Image domain V (set of image points)
Input: Pixel neighborhood mapping $E : V \to 2^V$
Input: Image function $f : V \to \mathbb{R}$
Output: Set of disjoint local minima regions $minima \subset 2^V$

procedure `FindLocalMinima`(V, E, f)

```
1  minima ← ∅
2  visited ← ∅
3  for each p ∈ V do
4      if p ∈ visited then
5          continue
6      Insert p into visited
7      stack ← {p} ,  points ← {p} ,  level ← f(p)
8      while stack ≠ ∅ do
9          extracted ← Extract and remove element from stack
10         for each q ∈ E(extracted) do
11             if q ∉ visited then
12                 Insert q into visited
13                 if f(q) < level then
14                     stack ← {q} ,  points ← {q} ,  level ← f(q)
15                     continue while
16                 else if f(q) = level then
17                     Insert q into stack and into points
18             else if f(q) < level ∨ [f(q) = level ∧ q ∉ points] then
19                 continue for each p
20     minima ← minima ∪ {points}
21 return minima
```

Appendix B

Parallel CCT construction algorithm software implementation timings

The plots on the following pages show wall clock times, performance improvements and software thread utilizations for various numbers of threads on multiple machines. Table B.1 lists the computers used to make the measurements. Two different input images shown in Figure B.1 were used for measurements: A small image (Goldhill, 784×576 px, 8 bits), and a large image (Earth, 2816×2816 px, 8 bits).

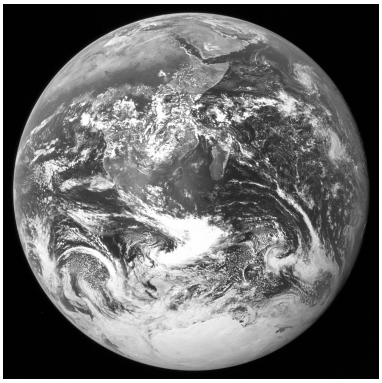
Plots in Figures B.2–B.8 are for the small image, Figures B.9–B.15 are for the large one.

Figures B.2, B.3, B.4, B.9, B.10, B.11 compare results from different machines,
Figures B.5, B.6, B.7, B.8, B.12, B.13, B.14, B.15 compare results from different scheduling strategies.

The line labeled “Theor. limit" shows the theoretical limit: efficiency equal to 1.

Data sizes for the computations were 1 byte per pixel for the input image and 4 bytes per pixel for the resulting parent point tree.

(a) Small image: Goldhill, 784×576 px, 8 bits



(b) Large image: Earth, 2816×2816 px, 8 bits

Figure B.1: Input images used for measurements

Table B.1: Testing systems

Name	CPUs	Cores	Hardware threads	System description
ikaros	1	2	2	Notebook with Intel Pentium Dual-Core T2330 (1.60 GHz), FSB 533 MHz, Cache: 32 KB instruction + 32 KB data 8-way L1, 1 MB 4-way L2 (full-speed), 64 bytes per line, 1 GB RAM 2R×16 8-bank PC2-5300S-555-12, Mobile Intel GM965 Express Chipset, Windows XP Professional, code compiled using Visual C++ 2005 Express Edition
skurut66	2	2	4	2× Intel Xeon 3.06 GHz (HT), 2 GB RAM, Linux version 2.6.20.1 (ruda@erebor.ics.muni.cz) (gcc version 3.3.5 (Debian 1:3.3.5-13)) #1 SMP Thu Mar 1 16:39:20 CET 2007, code compiled using gcc version 4.1.2 20061115 (prerelease) (Debian 4.1.1-21) [MetaCentrum]
konos30-1	2	4	4	2× Dual-Core AMD Opteron 270 (2 GHz), 4 GB RAM, Linux version 2.6.22.17-0.1-xen (geeko@buildhost) (gcc version 4.2.1 (SUSE Linux)) #4 SMP Mon Jul 21 14:00:22 CEST 2008, virtualization, code compiled using gcc version 4.1.2 20061115 (prerelease) (Debian 4.1.1-21) [MetaCentrum]
manwe1	8	16	16	8× Dual-Core AMD Opteron 885 (2.6 GHz), 64 GB RAM, Linux version 2.6.18-smp (root@manwe1) (gcc version 4.0.2 20050901 (prerelease) (SUSE Linux)) #4 SMP Tue Oct 17 11:09:17 CEST 2006, code compiled using gcc version 4.1.2 20061115 (prerelease) (Debian 4.1.1-21) [MetaCentrum]

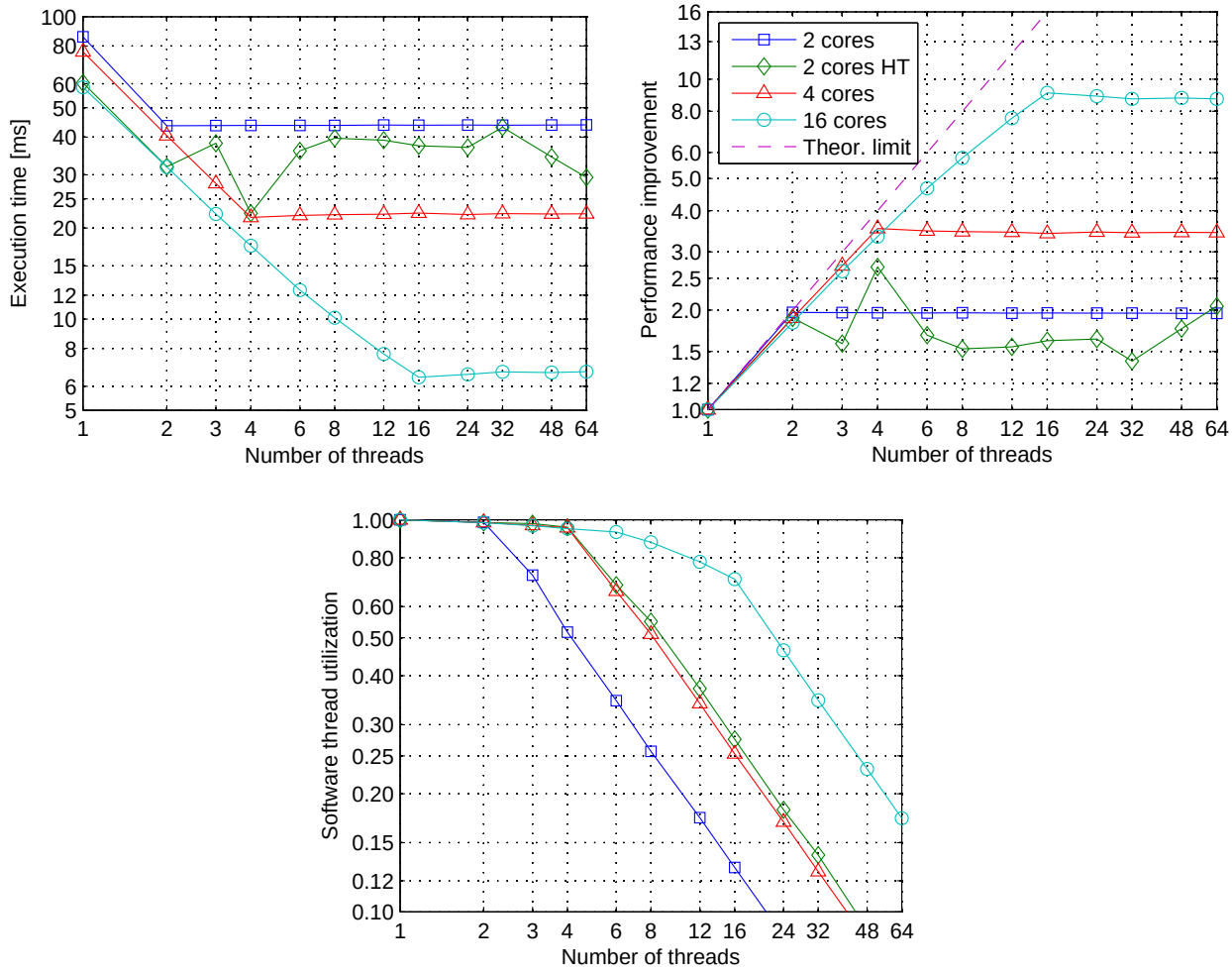


Figure B.2: Goldhill (784×576, 8 bits) – Strategy 1

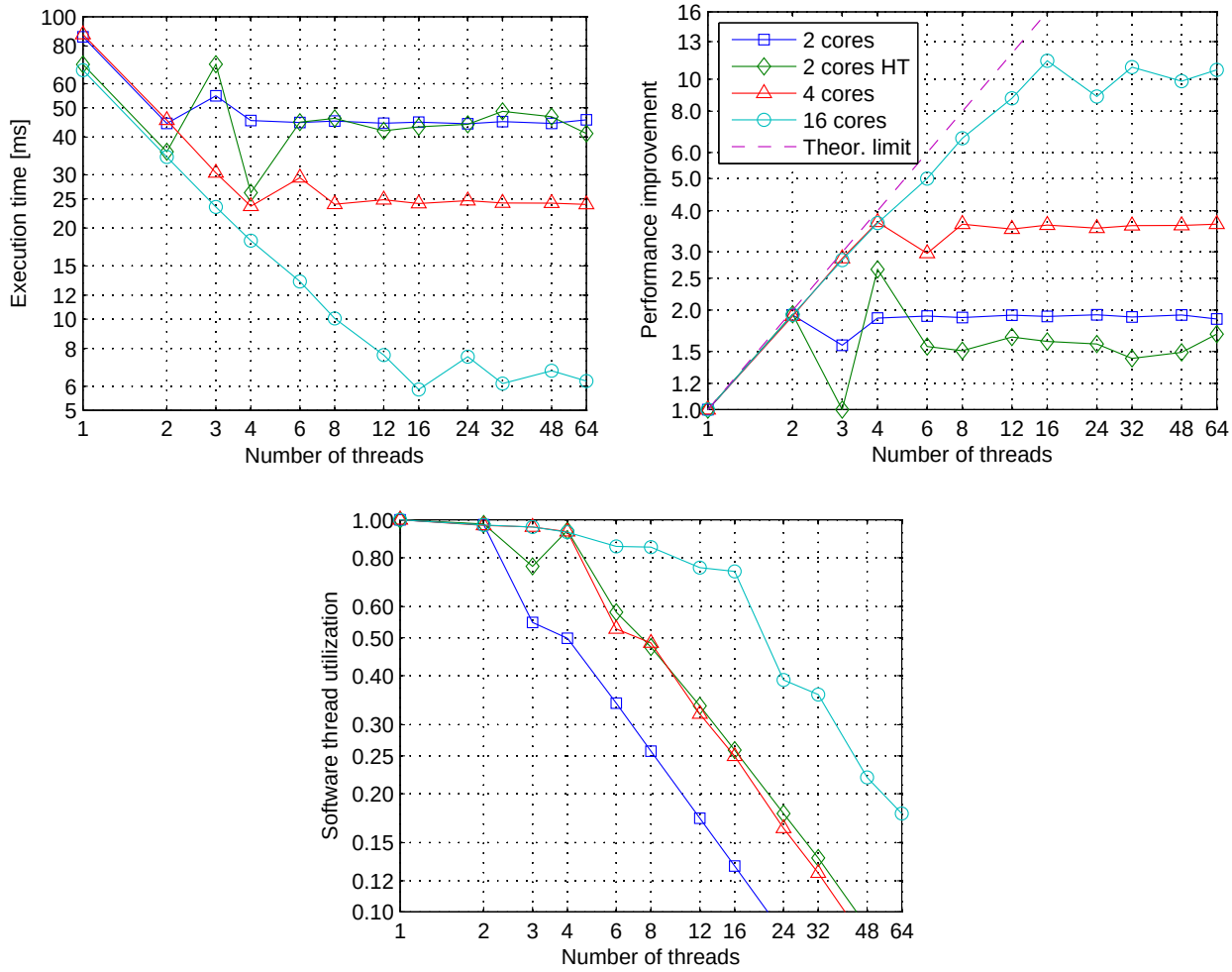


Figure B.3: Goldhill (784×576, 8 bits) – Strategy 2

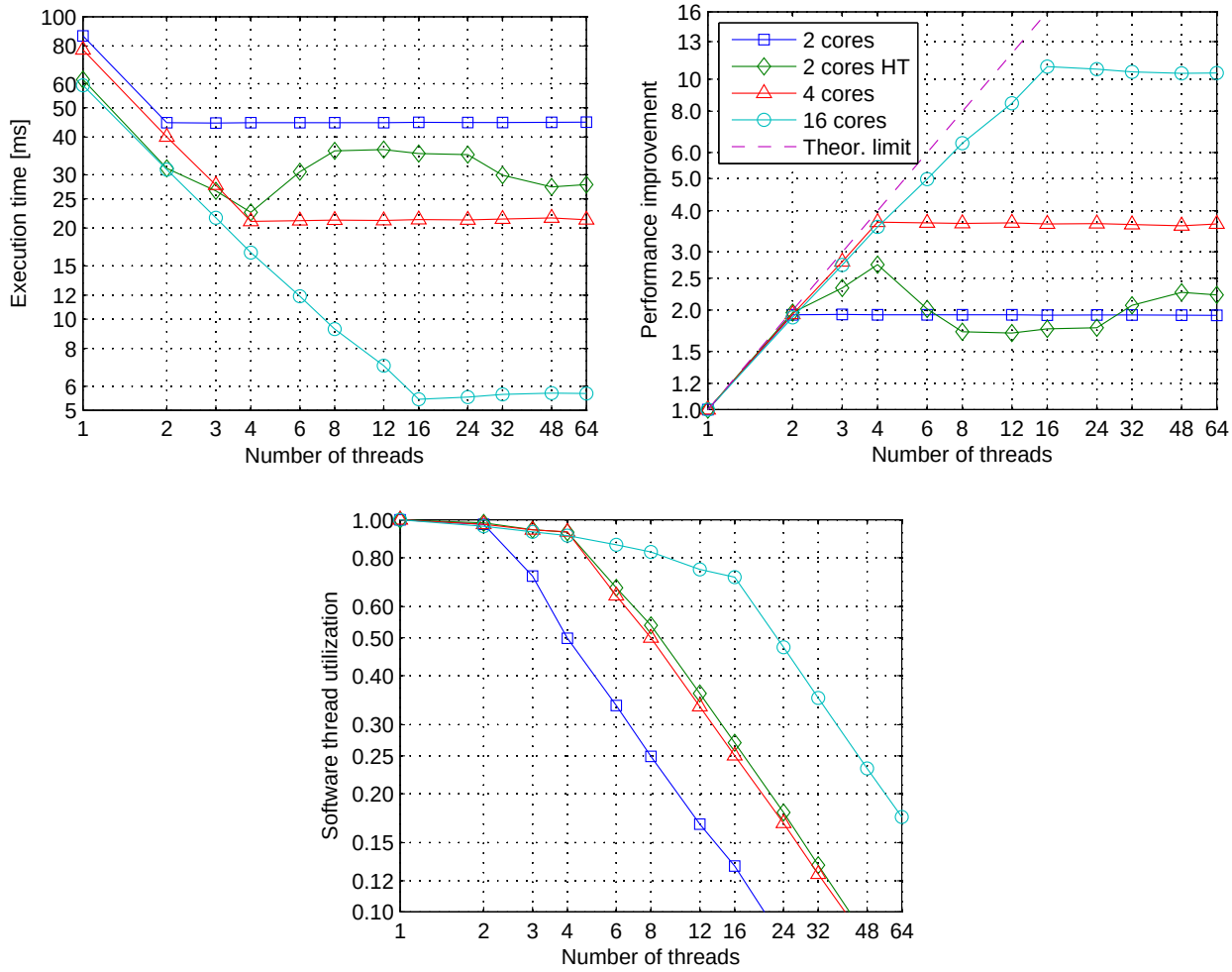


Figure B.4: Goldhill (784×576, 8 bits) – Strategy 3

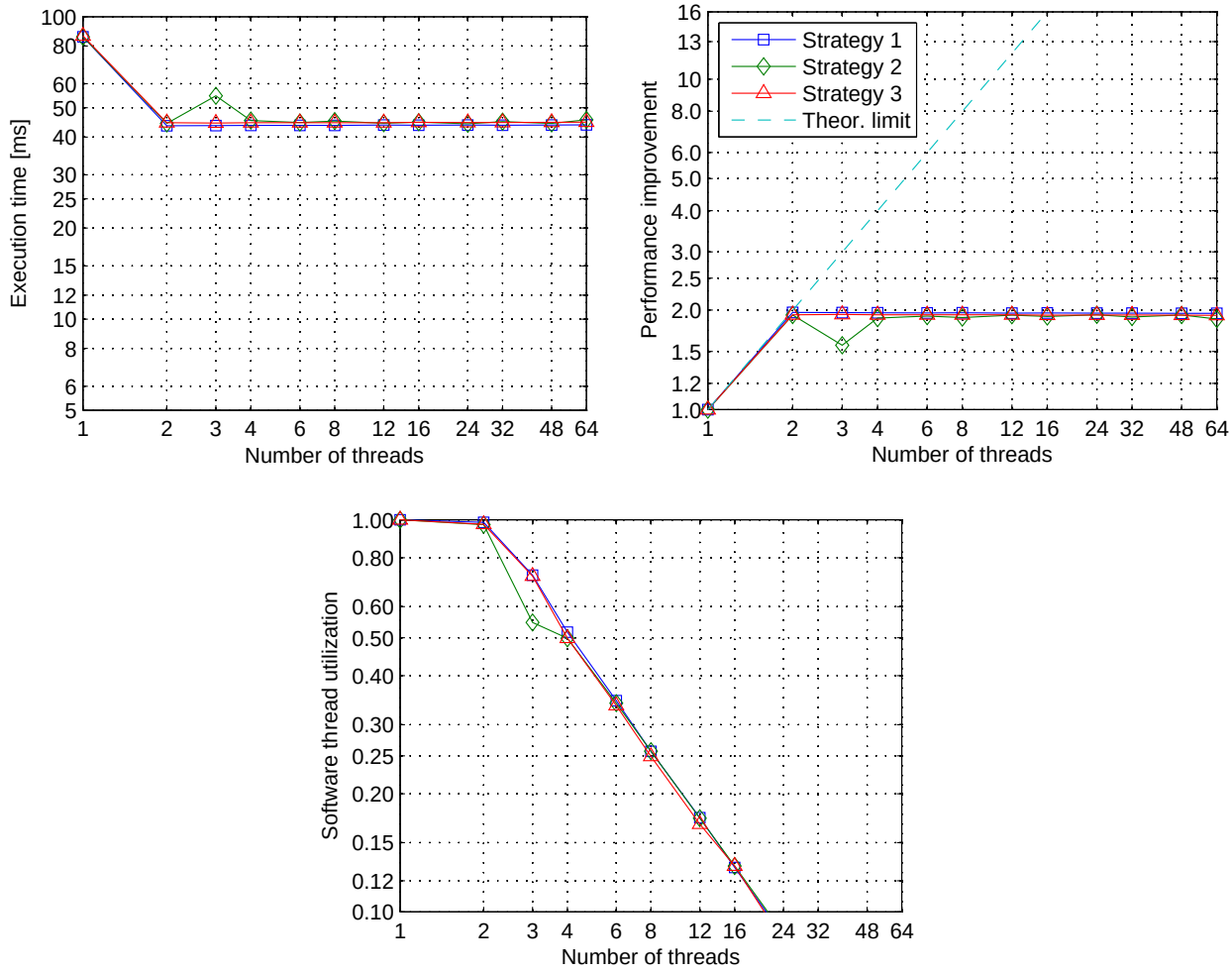


Figure B.5: Goldhill (784×576, 8 bits) – 2 cores

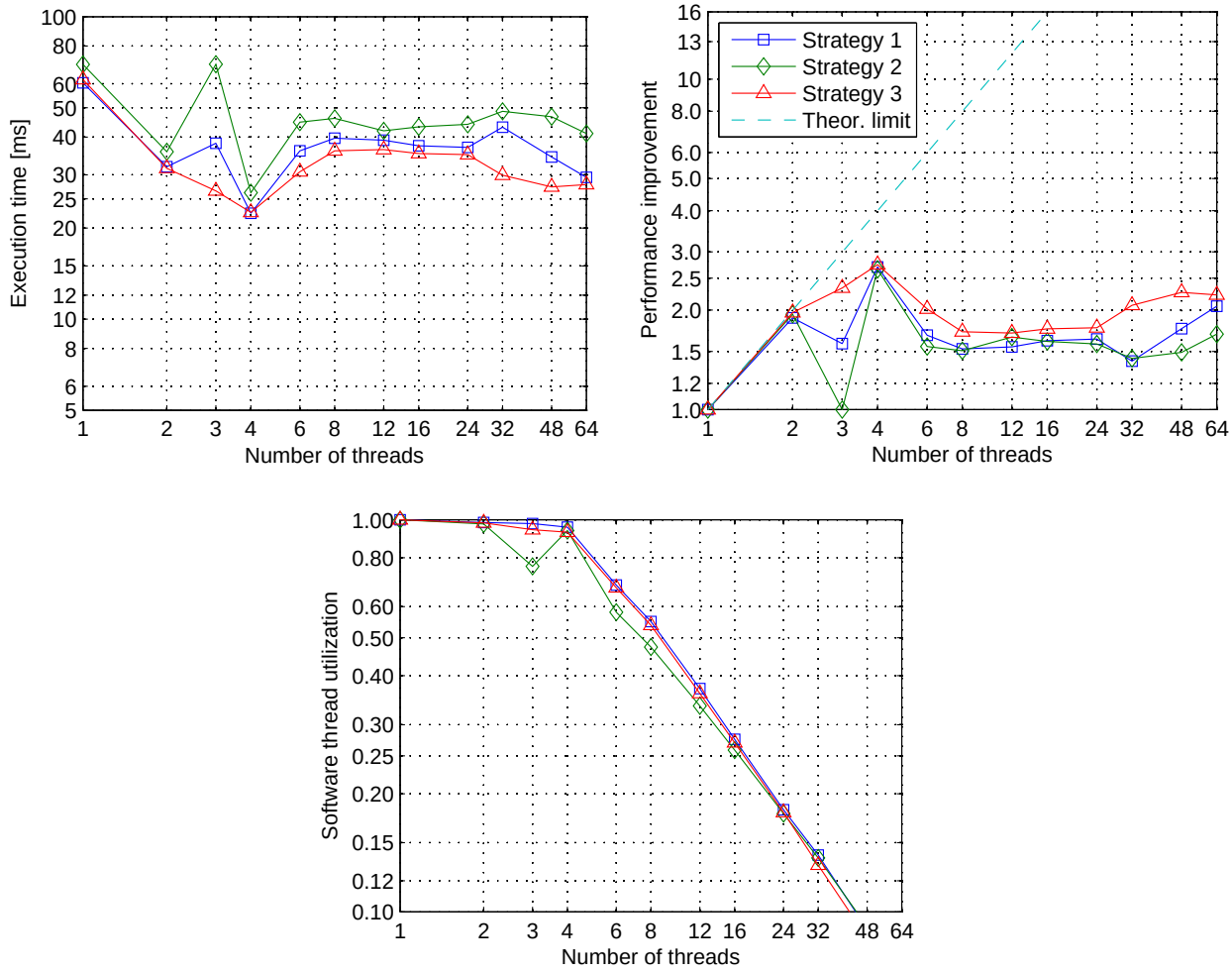


Figure B.6: Goldhill (784×576, 8 bits) – 2 cores with hyper-threading

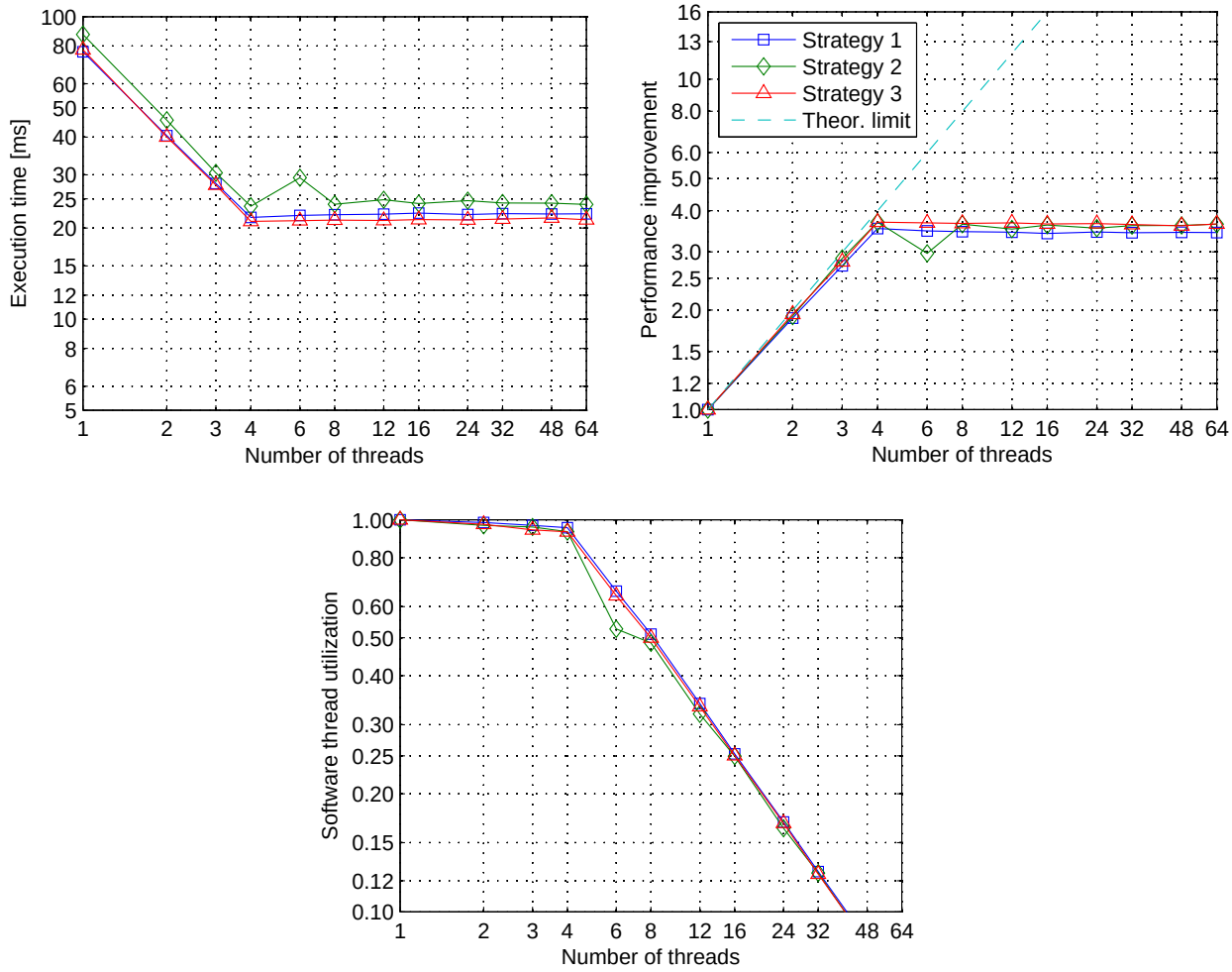


Figure B.7: Goldhill (784×576, 8 bits) – 4 cores

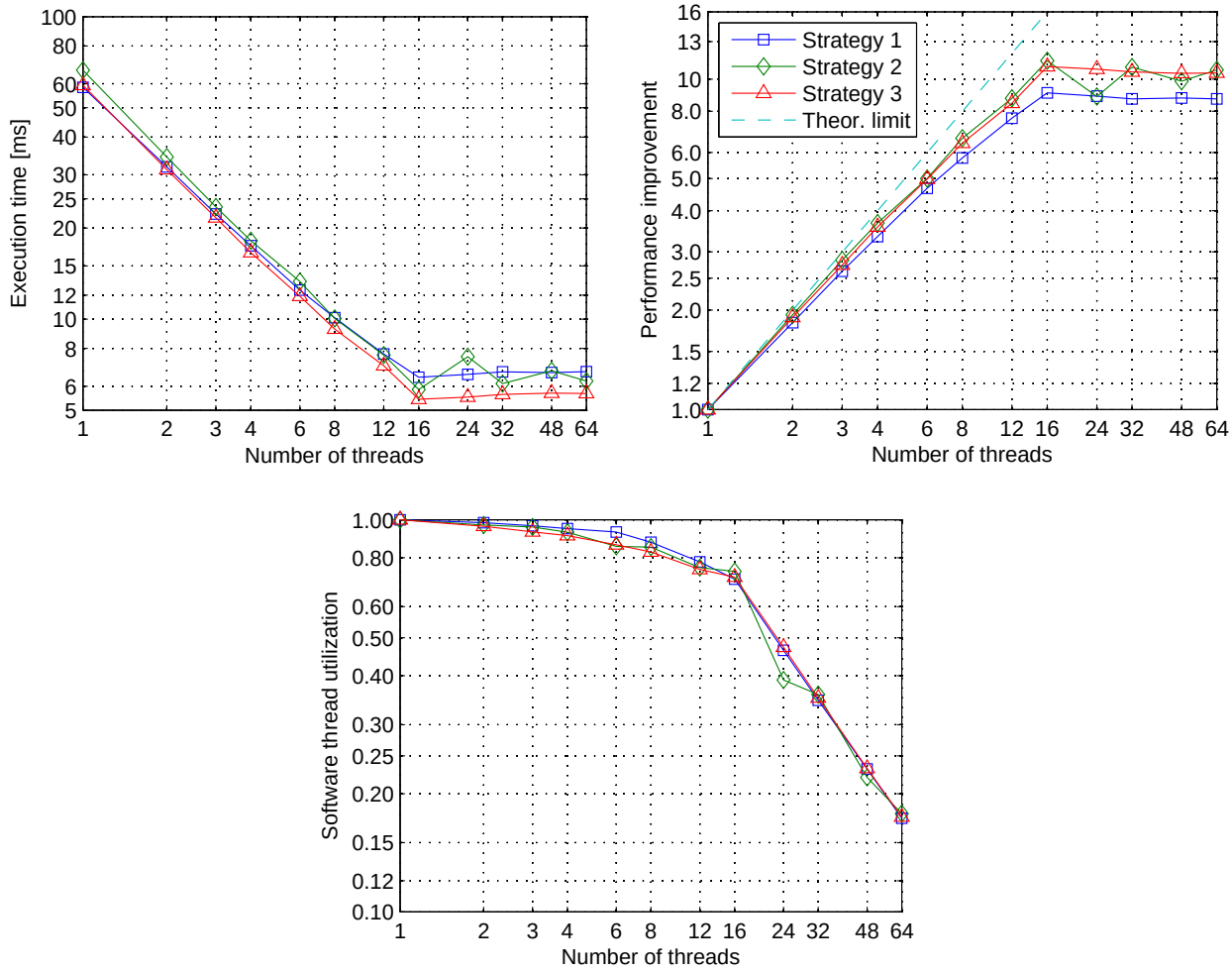


Figure B.8: Goldhill (784×576, 8 bits) – 16 cores

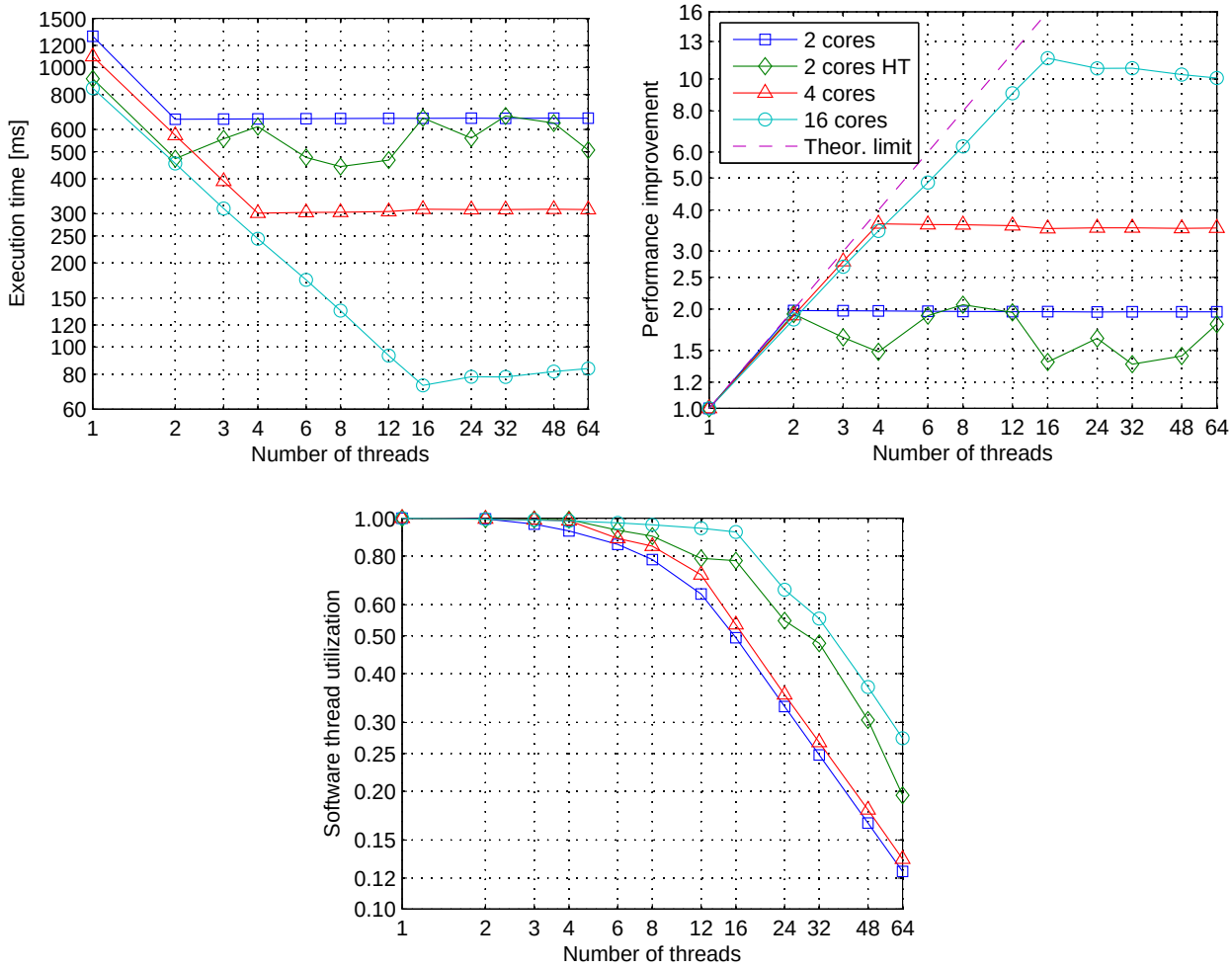


Figure B.9: Earth (2816×2816, 8 bits) – Strategy 1

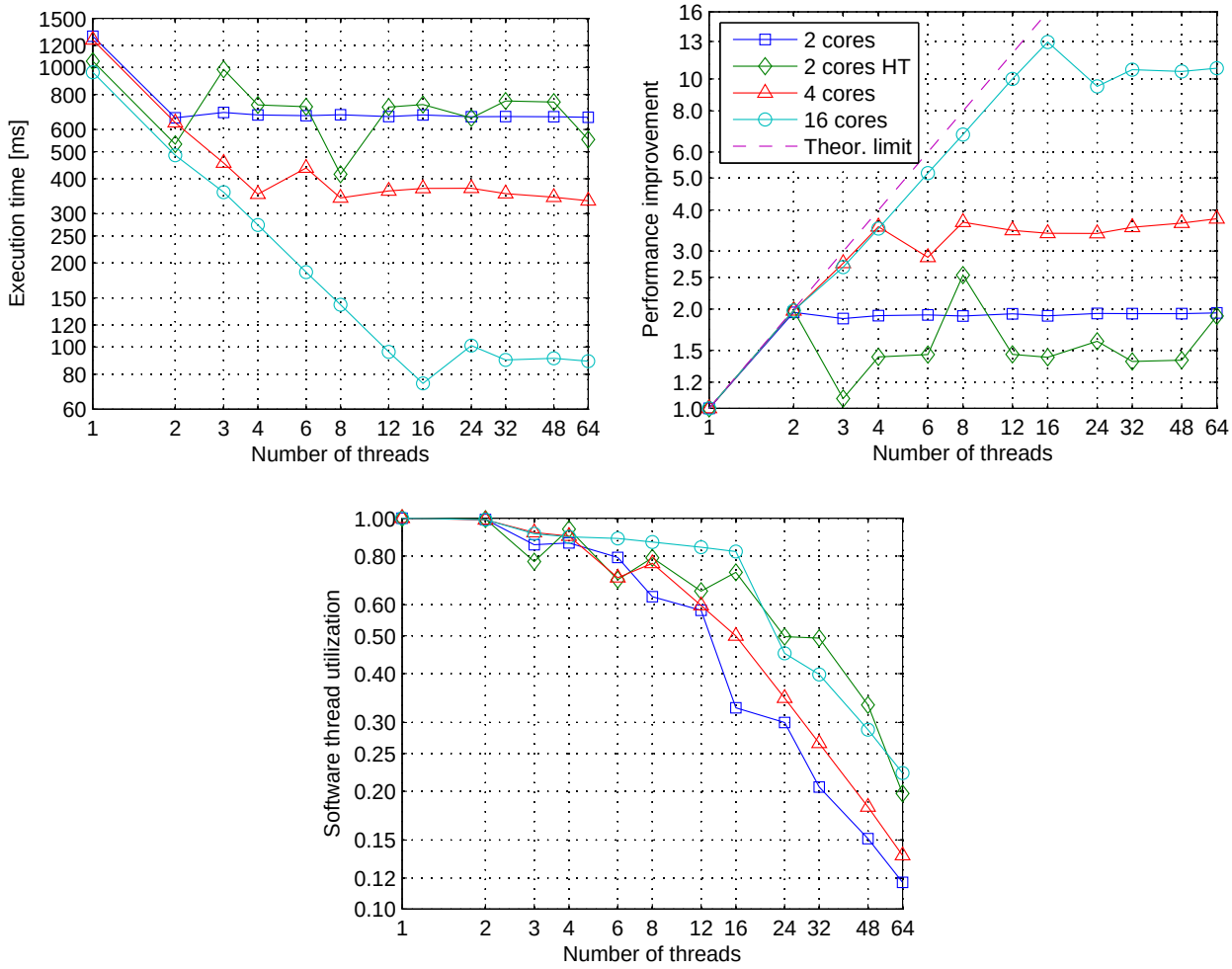


Figure B.10: Earth (2816×2816, 8 bits) – Strategy 2

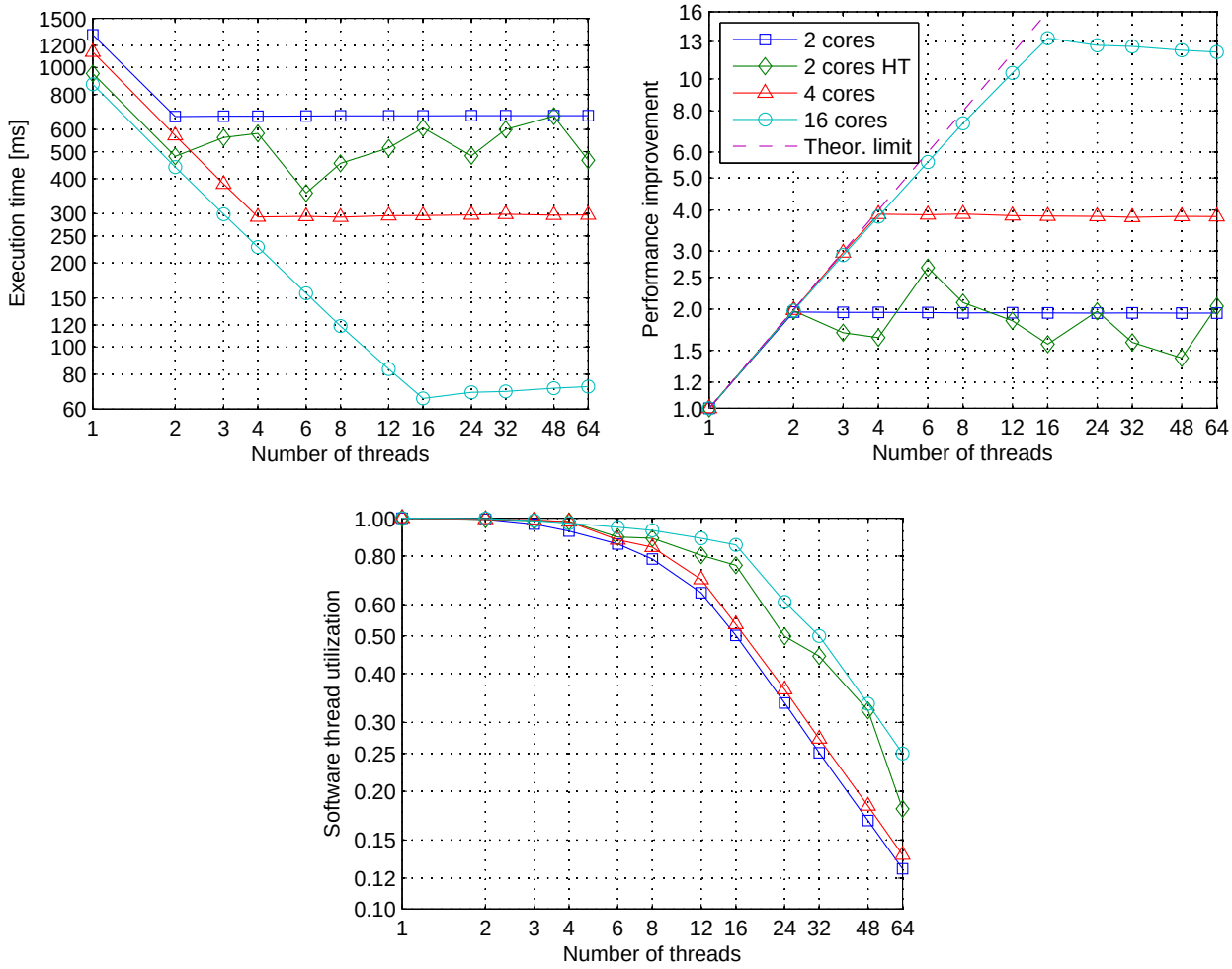


Figure B.11: Earth (2816×2816, 8 bits) – Strategy 3

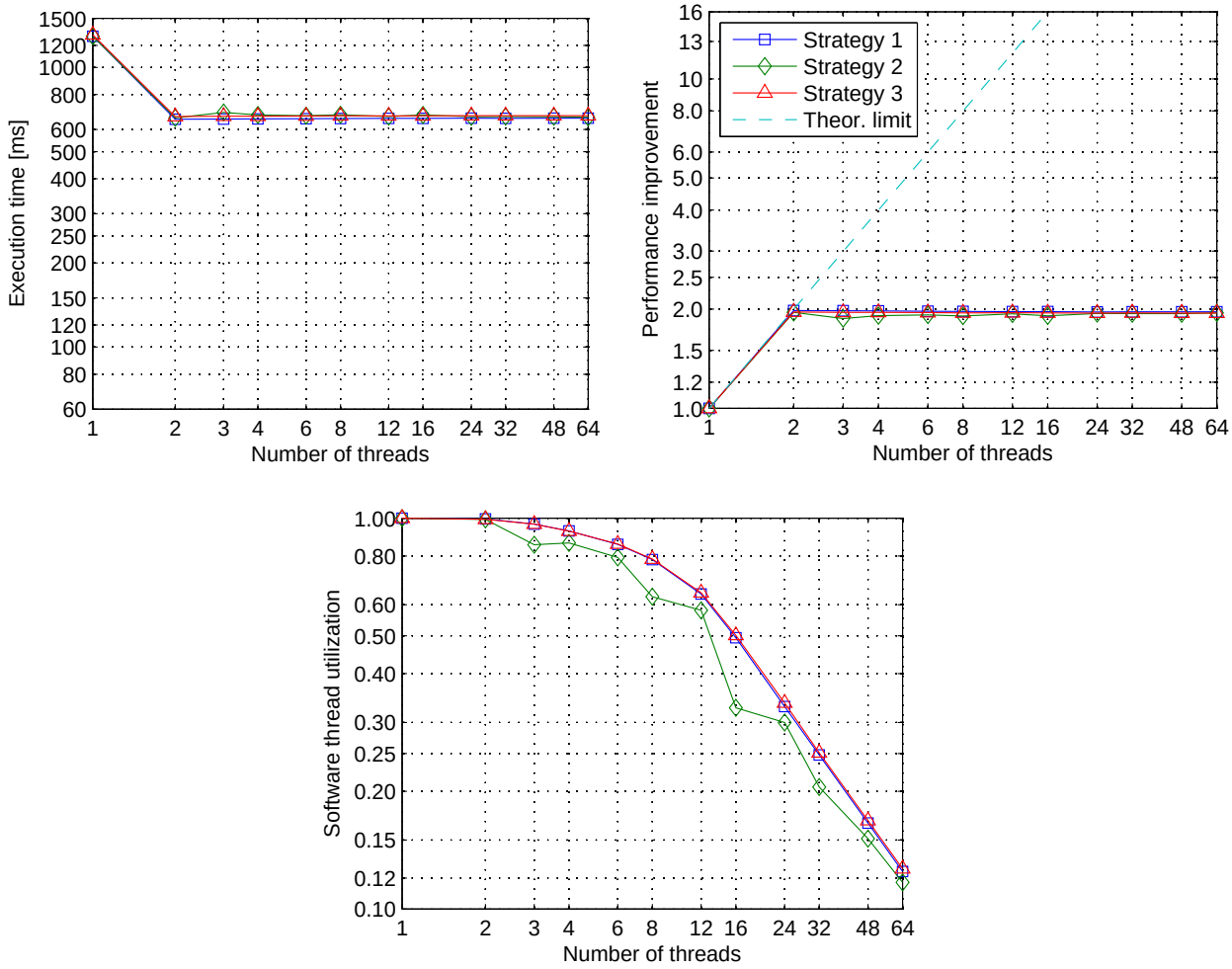


Figure B.12: Earth (2816×2816, 8 bits) – 2 cores

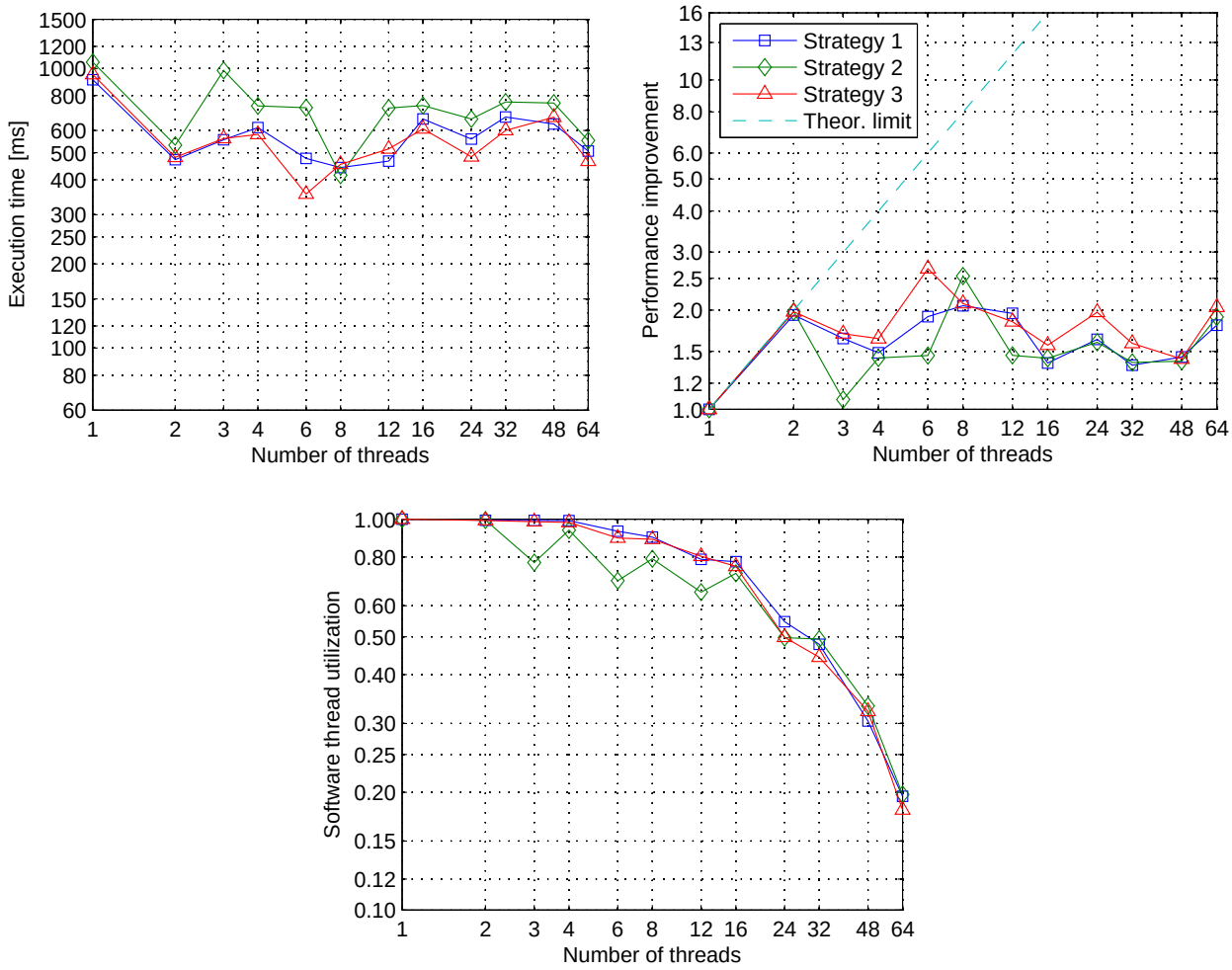


Figure B.13: Earth (2816×2816, 8 bits) – 2 cores with hyper-threading

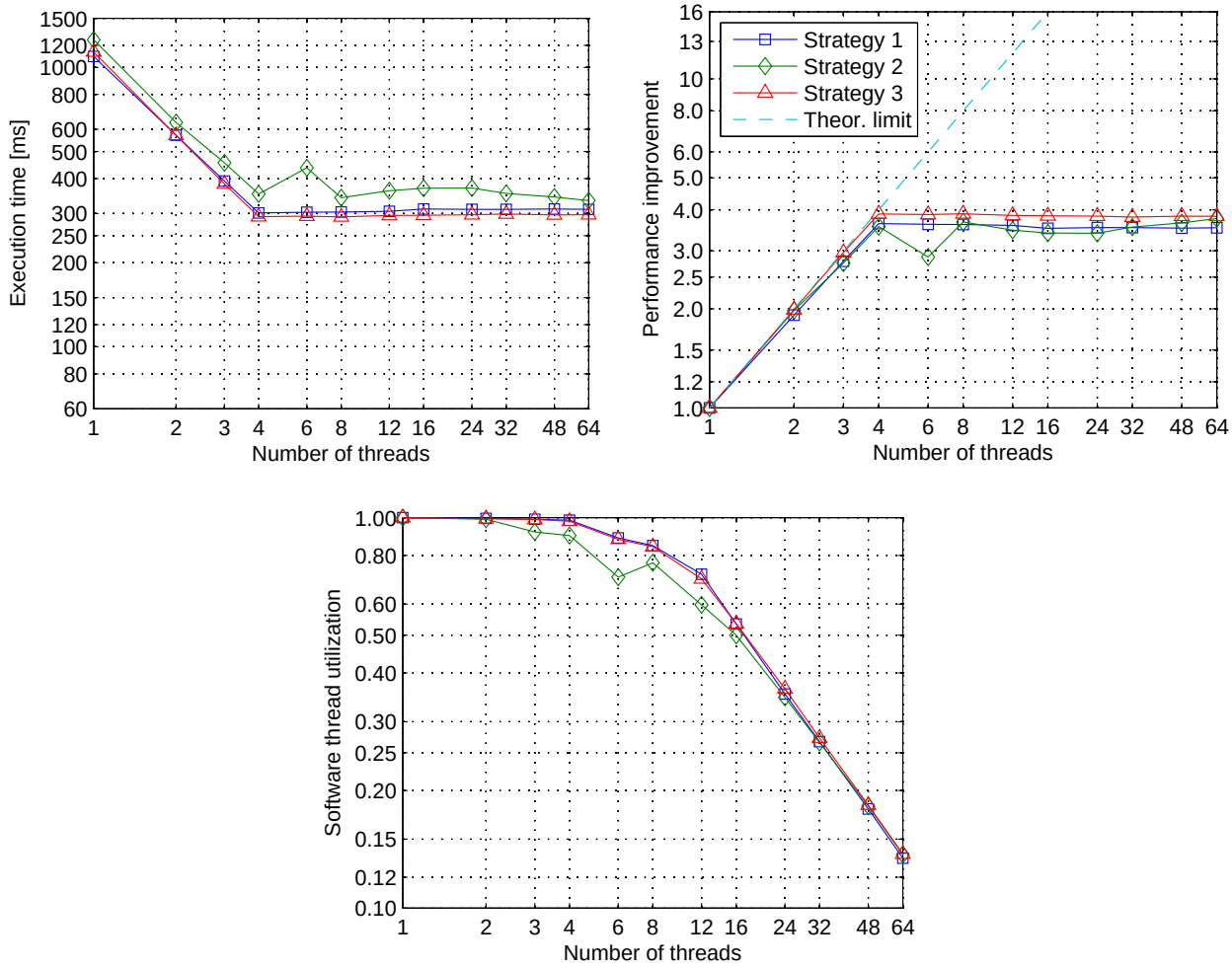


Figure B.14: Earth (2816×2816, 8 bits) – 4 cores

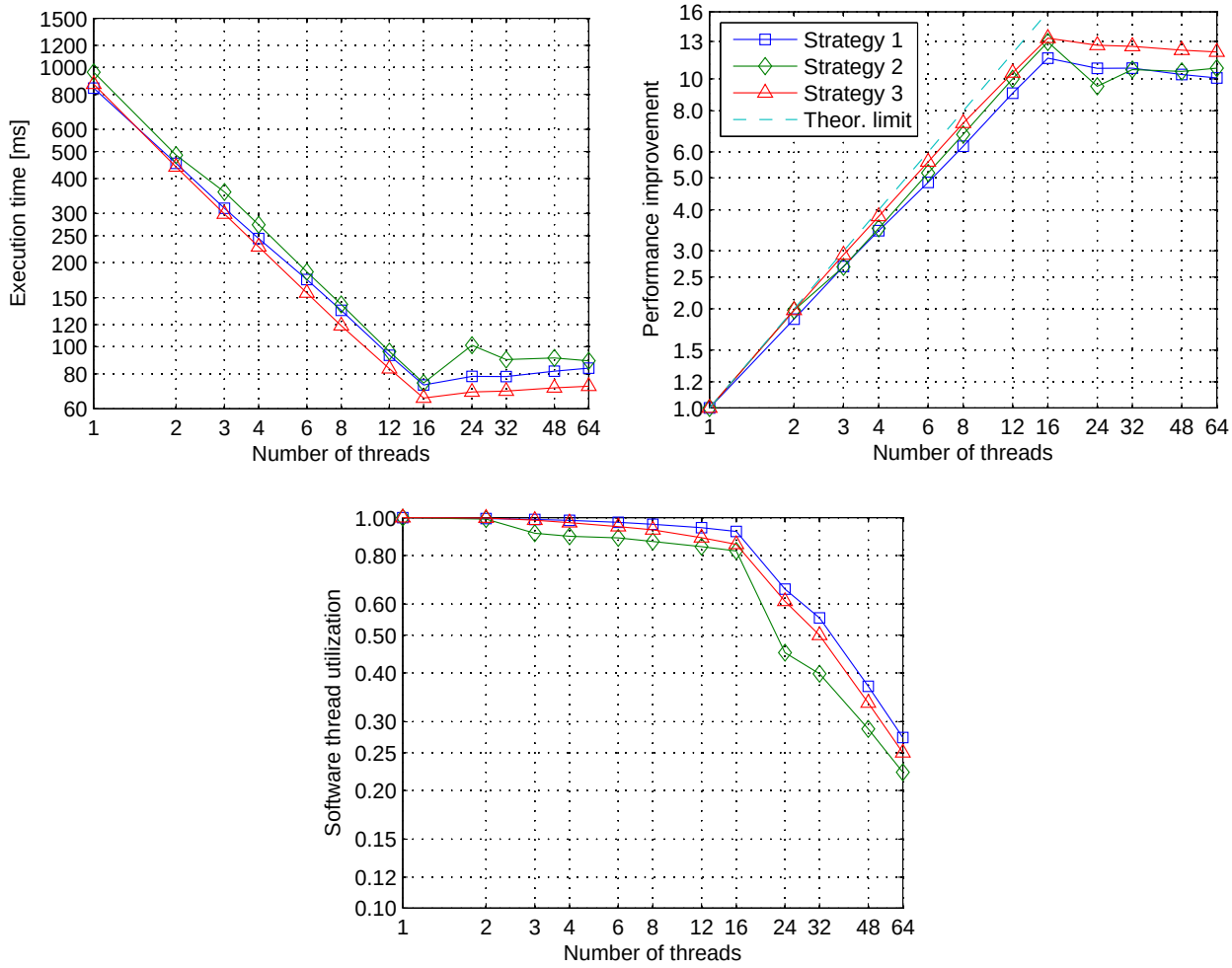


Figure B.15: Earth (2816×2816, 8 bits) – 16 cores