Stellingen
behorende bij het proefschrift

Embedding data and task parallelism
in image processing applications

Cristina Soviany
26 mei 2003

1. De hoeveelheid parallelisme die in de praktijk wordt verkregen hangt nog steeds af van de inspanningen van de applicatie programmeur. [Hoofdstuk 2 van dit proefschrift]

2. De huidige processor en communicatie technologie in aanmerking nemend, kunnen low-level image processing operatoren tegenwoordig ook efficiënt op MIMD systemen parallel uitgevoerd worden. [Hoofdstuk 3 van dit proefschrift]

3. Algoritmische Skeletten zijn parallelle ontwerp patronen die een zelfde impact hebben op het gebied van parallel rekenen als de klassieke ontwerp patronen op het gebied van de object georiënteerde software. [Hoofdstuk 4 van dit proefschrift]

4. De grens tot waaraan het nog zinvol is om programmeer abstracties te bouwen wordt gedicteerd door de onderliggende hardware. [Hoofdstuk 5 van dit proefschrift]

5. Het tegelijkertijd exploiteren van data en taak parallelisme verhoogt de prestatie van applicaties daar waar een toename van slechts taak parallelisme de prestatie laat afnemen. [Hoofdstuk 5 van dit proefschrift]

6. Het lezen van een artikel is het lezen van een hiërarchie van artikelen.

7. Wat sommigen waarnemen, wordt door anderen ontdekt.

8. Er is meer tijd nodig om het verhaal te vertellen dan om het mee te maken.


10. In het echte leven wordt een grotere voortgang verkregen door energie en zelfvertrouwen.

Deze stellingen worden verdedigbaar geacht en zijn als zodanig goedgekeurd door de promotoren:
Prof.dr.ir. H.J. Sips
Prof.dr.ir. L.J. van Vliet
Dr.ir. P.P. Jonker
Propositions
together with this thesis

Embedding data and task parallelism
in image processing applications

Cristina Soviany
May 26, 2003

1. The amount of parallelism obtained in practice depends on the amount of effort spent by the application programmer. [Chapter 2 of this thesis]

2. Today's processor and communication technology enable the efficient parallel execution of low-level image processing operators on MIMD architectures too. [Chapter 3 of this thesis]

3. Algorithmic skeletons are parallel design patterns that have an impact in the field of parallel computing similar to the classical design patterns in the field of object-oriented software. [Chapter 4 of this thesis]

4. The level at which it is practical to build programming abstractions is dictated by the underlying hardware. [Chapter 5 of this thesis]

5. Simultaneous exploitation of data and task parallelism improves the performance of an application in cases where a further increase of the degree of data parallelism alone reduces the performance. [Chapter 5 of this thesis]

6. Reading a paper is reading a hierarchy of papers.

7. What some people observe others discover.

8. It takes more time telling the story than living it.

9. Beautiful places should be visited only once.

10. In real life speed-up is gained by energy and self-confidence.

These propositions are considered defendable and as such have been approved by the supervisors:
Prof.dr.ir. H.J. Sips
Prof.dr.ir. L.J. van Vliet
Dr.ir. P.P. Jonker.
Embedding
Data and Task Parallelism in Image Processing Applications

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof.dr.ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op maandag 26 mei 2003 te 10:30 uur

door

Cristina Soviany

inginer
Universitatea Politehnica București
geboren te Boekarest, Roemenië
The work presented in this thesis was supported by the Netherlands Organization for Scientific Research (NWO) under grant 612-11-000.

Copyright © Cristina Soviany 2003

Proefschrift Technische Universiteit Delft
ISBN: 90-9016878-8
Keywords: Image Processing, Data Parallelism, Task Parallelism
Devoted to my children,
Andrei and Răzvan
E pluribus unum

Augustinus, Confessiones
The picture from the cover is a fresco painted by Benozzo Gozzoli in 1465, and shows a scene from the life of Augustinus, his parents introducing him to the school master.
## Contents

1 Introduction
   1.1 Levels of parallel programming .................................. 11
   1.2 Contributions ...................................................... 14
   1.3 Outline of the thesis ............................................... 16

2 Image processing algorithms
   2.1 Introduction ......................................................... 17
   2.2 Types of image pixels and image data structures .................. 17
   2.3 Low-level image processing ........................................ 21
   2.4 Intermediate level image processing ............................... 22
   2.5 High level image processing ....................................... 23
   2.6 A classification of low-level image operators ..................... 23
   2.7 Conclusions ......................................................... 31

3 Some parallel architectures and programming models for image processing
   3.1 Introduction ......................................................... 33
   3.2 Conceptual models of parallel computation ......................... 33
      3.2.1 Data Parallel Model .......................................... 34
      3.2.2 Task Parallel Model .......................................... 34
      3.2.3 Granularity and speed-up - basic performance considerations . 34
      3.2.4 Discussion .................................................... 37
   3.3 System level models of parallel computation ....................... 37
      3.3.1 System level classification .................................. 38
      3.3.2 MIMD systems .................................................. 38
      3.3.3 SIMD systems .................................................. 39
3.3.4 Discussion .................................................. 50
3.4 Implementation level models ............................... 51
  3.4.1 Message passing in imaging ......................... 51
  3.4.2 Shared address space mechanism ..................... 53
  3.4.3 Discussion ............................................... 55
3.5 Parallel design patterns .................................. 55
  3.5.1 Distributed Bucket Processing ....................... 56
  3.5.2 Discussion ............................................... 58
3.6 Parallel programming languages .......................... 59
  3.6.1 Automatic parallelization ............................ 60
  3.6.2 Explicit parallelization .............................. 65
  3.6.3 Annotation based parallelism ......................... 67
  3.6.4 Discussion ............................................... 69
3.7 Experiments .................................................. 70
  3.7.1 Message passing experiments ......................... 71
  3.7.2 Explicit parallel languages (CC++) versus automatic (SUN) 82
  3.7.3 Experiments on SIMD systems ......................... 84
  3.7.4 Discussion ............................................... 93
3.8 Conclusions .................................................. 94

4 Data Parallelism in Image Processing ........................ 97
  4.1 Introduction ............................................... 97
  4.2 Algorithmic skeletons .................................. 97
  4.3 Related work ............................................. 98
  4.4 Data mapping ............................................. 99
  4.5 Simultaneous exploitation of data and task parallelism 101
  4.6 Algorithmic skeletons .................................. 102
  4.7 The data parallel framework ............................ 106
  4.8 Skeletons for low-level image processing ................ 108
  4.9 Skeleton approach discussion ......................... 109
  4.10 Experiments ............................................... 114
  4.11 Conclusions ............................................... 118
5 Mixed Data and Task Parallelism in Image Processing 121
5.1 Introduction .......................... 121
5.2 Dependency Analysis .................. 123
5.3 The Image Application Task Graph Model .......... 127
5.4 Mathematical Cost Models ................. 131
  5.4.1 Processing cost model ................ 131
  5.4.2 Communication cost model ............ 133
  5.4.3 Communication optimization .......... 135
  5.4.4 IATG cost properties ............... 136
5.5 Mapping and Scheduling Algorithms .......... 138
  5.5.1 Definitions ......................... 138
  5.5.2 Pure Task Scheduling (TASK) .......... 141
  5.5.3 Pure Data Scheduling (DATA) .......... 141
  5.5.4 Two Level Scheduling(TwoL) .......... 141
  5.5.5 Two Step Allocation and Scheduling (TSAS) .... 143
  5.5.6 The Critical Path Reduction (CPR) Algorithm .... 145
  5.5.7 A measure of performance .............. 146
  5.5.8 Order analysis of the parallelization approaches .... 147
5.6 Experiments .......................... 149
  5.6.1 Multi-baseline stereo vision .......... 149
  5.6.2 Optical flow ........................ 156
  5.6.3 Artificial Task Graphs ............... 166
5.7 Conclusions .......................... 166

6 Exploiting data and task parallelism in a parallel programming lan-
guage 169
6.1 Introduction .......................... 169
6.2 Spar language extensions .................. 169
6.3 Program annotations ..................... 170
6.4 Supported pragmas ....................... 170
6.5 Implementing image processing applications in Spar .......... 172
6.6 Conclusions .......................... 174

7 Conclusions ......................... 175

Bibliography .......................... 177
A Posynomial functions and convex programming 189
  A.1 Convex sets ......................................... 189
  A.2 Convex functions ..................................... 189
  A.3 The convex programming problem .................. 190
  A.4 Posynomial functions ................................. 190
    A.4.1 Properties of posynomial functions ............ 191

B Estimation of processing and communication cost 193
  B.1 Processing cost estimation .......................... 193
  B.2 Communication cost estimation ...................... 195

Summary 199

Samenvatting 201

Acknowledgements 203

Curriculum Vitae 205
Chapter 1

Introduction

1.1 Levels of parallel programming

In this thesis we address the problem of building a high level system for parallelizing image processing applications on parallel hardware architectures, like distributed memory systems. Distributed memory systems consist of a collection of processors, each having its own memory. Parallel hardware architectures can be programmed using two conceptual models of parallel programming: data parallel and task parallel. In data parallel models, the data is partitioned and distributed among the processors. In image processing, data consist of images, so the images are split and each part or sub-image is processed by a different processor. In task parallel models, one can group the instructions into tasks and assign each task to a different processor. More details about parallel architectures and conceptual models of parallelism can be found in Chapter 3.

In building such a high level system for parallel image processing we mix the paradigms of parallel programming mentioned above: data decomposition and task decomposition. So our research aims at the development of tools that make the effort of implementing parallel image processing applications easier. Our first target is to focus on industrial - possibly embedded - image processing applications. Our parallel programming tools should provide:

- A high processing speed of the entire application at moderate machine cost.
- Ease of programming and ease of software maintenance at moderate machine cost.
- Portability on a variation of platforms: assembled from commodity components, but including heterogenous architectures.
The task of designing and implementing any part of a computer system is essentially a process of abstraction. The facilities provided on a lower level are used to construct an implementation of a higher level of abstraction, each with its own characteristics. In a complete system, many such levels are involved. Each abstraction allows us to sacrifice a certain degree of freedom, in return for a more useful and appropriate set of resources. In practice, this often results in performance loss of applications designed using higher level systems, over that which could be achieved by an implementation at a much lower level. Thus, a high level program subjected to a series of automatic compilations cannot be expected to run as quickly as an alternative solution, written directly in assembler code for the same machine. But, on the other hand, in the latter approach, the original problem might become much more complex. In this way, the abstraction process makes application programming easier and it can be regarded as a way to make a wider range of parallel solutions accessible in practice. Furthermore, these higher level solutions come with significant gains in portability and clarity of the code.

The historical trend in hardware technology has been towards a dramatical increase in miniaturization, speed and reliability. Processing elements are nowadays available, in the same extent as were memory cells traditionally. It is now quite achievable to build computers in which thousands of processors operate concurrently to solve a single problem. Forty years ago this was only an idea, but Gorn [39] already recognized the challenges that would be posed to the system designer by the new parallel computers:

"But visualize what it would be like to program for such a machine! If a thousand subroutines were in progress simultaneously, one must program the recognition that they have all finished, if one is to use their results together. The programmer must not only synchronize his subroutines, but schedule all his machine units, unless he is willing to have most of them sitting idle most of the time. Not only would programming techniques be vastly different from the serial ones in the serial languages we now use, but they would be humanly impossible without machine intervention."

Parallel hardware poses new questions. To what extent should the new found concurrency at the lower levels be reflected in the abstractions built on top of them? And, how should the parallelism be presented?

A variety of techniques are currently being used to address the problem of building higher level programming systems upon existing parallel hardware. When less important details are filtered out, the resulting systems fall into three categories.

- The level of abstraction of the systems designed in this first category is high. Programmers of these machines are not required to deal with parallelism at all and need to have no knowledge of the implementation to make use of the system. These systems are characterized by the isolation of the abstract (or idealized) design space, as can be seen by the programmer, from the parallel, distributed implementation. More concrete, the systems in this category fall in the group of automatic parallelization tools and compilers.

- In the second category the degree of abstraction is reduced. Here, the programmers are required to present the parallel solutions explicitly. However, in doing
1.1 Levels of parallel programming

so they also obtain some freedom not offered directly by the hardware. The systems from this category are those in which the programmer's world includes explicit parallelism. The programmer is now responsible for decomposing the solution into a collection of concurrent processes.

- The third category contains systems in which the programmer is required to specify solutions which use parallelism in a style very close to the present hardware. To be able to program these systems the user must have an extensive and detailed knowledge of the parallel architectures and parallel programming.

Our research was mainly focused to study and develop parallel image processing programming systems belonging to the second category, and hence focused on application programmers with moderate, yet sufficient knowledge of parallel computing.

The most fundamental and obvious principle in developing parallel systems, is program decomposition which can be used for the identification of parallelism. Processes must be identified and described that can operate concurrently. The term process here implies both data and task intensive processes.

The second idea is that of distribution, the exploitation of the potential parallelism identified by the decomposition phase by distributing them over processors. We must specify a mapping of the tasks that can be executed concurrently on the available processors.

Task and data parallelism are often considered to be mutually exclusive approaches to parallel programming. Yet many applications can benefit from both forms of parallelism. This is also the case in image processing. Image processing is widely used in many fields and applications, including the film and entertainment industry, medical imaging, industrial manufacturing and inspection, weather forecasting etc. In some of these areas the size of the images is very large, yet the processing time has to be very short. Sometimes real-time processing (keeping up with the frame rate of the camera) is required. Therefore, during the last decade there has been an increasing interest in the development and the use of parallel algorithms in image processing. Many algorithms have been developed for parallelizing various image operators on different parallel architectures. Most of these parallel image processing algorithms are either architecture dependent, or specifically developed for a dedicated application. They are often very difficult to be implemented by an image processing application programmer without sufficient knowledge of parallel computing.

The research described in this thesis is part of the PILE project. The two parts of this project involve firstly the development of a software architecture that allows image processing application programmers, with knowledge of sequential computing only, to automatically implement data parallel applications on distributed memory systems. Whereas, secondly, it aims at the design of a programming environment for the data and task parallel programming of applications using an annotated sequential language concept. The focus of this thesis is on the latter part. The two methods are supplementary.

\(^1\)Parallel Imaging Library and language Environment
In this thesis we study the benefits of exploiting both data and task parallelism in image processing applications using algorithmic skeletons [11, 15, 97] and an Image Application Task Graph (IATG). Skeletons are algorithmic abstractions common to a series of applications, which can be implemented in parallel. Skeletons are embedded in a sequential host language, thus being the only source of parallelism in a program. Based on skeletons, we have developed a data parallel image processing framework/system which does not hide the entire parallelization process but it is still easy to use by an image processing application programmer.

Recently, it has been shown that exploiting both task and data parallelism in a program to solve very large computational problems yields better speedups compared to either pure task parallelism or pure data parallelism [85, 103]. The main reason is that the speedup for either data or task parallelism are limited, and therefore using only one of them limits the achievable performance. Exploiting mixed task and data parallelism has emerged as a natural solution. In many applications from the field of signal and image processing, data set sizes are limited by physical constraints and cannot be easily increased. In such cases the amount of available data parallelism is limited. For example, if the size of an image is determined by a camera and a high throughput is required, then - depending on the interconnection speed of the processors - adding more processors for data parallelism will at a certain amount not be beneficial anymore while additional benefit may come from task parallelism. Note, that in every application this breakeven point may be different. Moreover, sometimes the number of available processors is limited, and in these cases we can increase the performance of the application by considering also the amount of task parallelism present in an application.

1.2 Contributions

In general, computer vision tasks consists of image processing algorithms of different complexity, which can be grouped in 3 large classes:

- Low-level image processing operators which operate at the pixel (picture element) level.
- Intermediate-level image processing operators which operate on abstractions derived from image pixels.
- High-level image processing operators which operate on higher abstractions, usually derived by intermediate-level operators.

Because in some applications the size of the images to be processed is increasing, and new and complex algorithms are being developed, the computation need increases, which requires faster processing. In contrast with a decade ago, clusters of workstations are now often available and their mutual interconnection can be made of Gigabit-speed networks. This makes that the necessity for special hardware systems for image processing is cornered to very high speed and demanding applications. Two other
phenomena are observed: the common availability of multi-processor boards (shared memory systems) and the dawn of special multi-media instruction sets in the current processors.

Some key points in designing higher abstraction level systems for the easy development of image processing applications on parallel hardware are:

- How can we make the parallelization of image processing applications user-friendly? Due to the target area of our proposed tools - professional image processing application designers - we focus on the development of parallelization tools from the first two categories presented in the previous section: semi-automated parallelization and specification of parallelism in a machine independent, abstract way.

- What paradigm of parallel programming is more suitable to be used, data parallelism, task parallelism or is it possible to combine both?

The main contributions presented in this thesis are as follows:

- The main reason for studying SIMD architectures and its way of (massively) data parallel programming, is that the SIMD paradigm is very well suited for low-level image processing. In the past, before the availability of cheap commodity clusters of workstations, many stand-alone SIMD machines performed outstanding on demanding image processing tasks. Also, new SIMD-like processor technologies such as MMX or SSE have been investigated with respect to their usability in image processing. The purpose of this part of our research was to investigate how incorporation of either an additional SIMD processor board, thus forming a heterogenous system, or the use of multi-media SIMD instructions such as MMX, would contribute to our speed requirements, and at what cost in the sense of programmability: i.e., how do they contribute to our list of application characteristics above.

- We have designed a system based on algorithmic skeletons for executing low-level image processing tasks in a data parallel way on a distributed memory system. By embedding the image processing operators within the algorithmic skeletons, the application programmer does not have to deal with the low-level parallelization details. However, the application programmer should be aware of some aspects of the parallelization process, as it will be concluded in Chapters 4 and 5.

- We introduce a general framework which enables the application programmer to exploit both data and task parallelism in image processing applications. The framework consists of a data parallel level based on algorithmic skeletons and a task parallel level built on top of the data parallel framework. The task parallel level is based on the image application task graph. We have studied the effect of the simultaneous exploitation of data parallelism and task parallelism with respect to the performance of low-level image processing applications.
1.3 Outline of the thesis

This thesis proposes a data and task parallel image processing environment, suitable for a large set of image processing applications. The thesis is organized as follows.

Chapter 2 presents an introduction to image processing and a classification of the image processing tasks, necessary for the skeleton concept developed in Chapter 4.

Chapter 3 introduces parallel architectures, programming models and languages suitable for image processing. The data and task parallel programming paradigms are presented, and also parallel programming languages that are suitable for image processing, such as SPAR.

Chapter 4 presents the data parallel framework of the parallel environment, based on so called algorithmic skeletons. Skeletons are algorithmic abstractions which hide the implementation details from the application user. Several skeletons are presented, in accordance with the classification of low-level image processing operators.

Chapter 5 presents the task parallel framework of the parallel environment. Task parallelism is based on extracting an Image Application Task Graph (IATG) which is the data dependency graph of the tasks composing the application.

Chapter 6 presents experimental results using the proposed parallel image processing environment to parallelize some applications and Chapter 7 draws conclusions.
Chapter 2

Image processing algorithms

2.1 Introduction

This chapter serves as a brief introduction for the non-expert in this field and is a concise inventory of the aspects that may play a role in the parallelization of these algorithms.

Image processing tasks can be categorized in three main groups, consisting of: low-level image processing operators, intermediate-level image processing operators and high-level image processing operators. Low level image processing operators usually convert image data into image data, e.g. contrast enhancement, noise reduction, filter transformations, or calculates characteristics of the input image (contours, histograms). Intermediate level image processing operations are more complex operations which derive abstractions from the image pixels (region labeling, object tracking). High level image processing is knowledge-based processing and concerns the interpretation of the information extracted from the intermediate level processing (classification, object recognition).

2.2 Types of image pixels and image data structures

The most common categories of pixels are: binary pixels, grey scale pixels and color pixels. Binary pixels can only have values TRUE (black) and FALSE (white), and require a single bit for storage. Gray scale pixels may cover a range of values like 0 to 255 (if they are represented by unsigned bytes), -128 to 127 (for signed bytes), 0 to 65535 (for unsigned shorts), -32768 to 32767 (for signed shorts) and a large dynamic range for floating point representations. Color pixels require three components, one for red, green and blue, each having the same dynamic range as the gray scale values. Therefore, color pixels need at least three bytes. There are different models for representing color pixels, such as RGB, YUV, HIS.

We distinguish the following image data:
• Iconic images: images containing the original data, usually integer matrices with data on pixel brightness. Neighboring pixels are partially correlated due to the spatial extent of the point-spread function of the imaging system in raw images or due to the local neighborhood operations that were applied to the images for highlighting some aspects of the image that are important for further treatment.

• Segmented images: parts of an image are joined into groups with identical grey values (labels) that represent the objects in the image.

• Geometric images: hold features on 2D and 3D shapes. The quantification of a shape is difficult but important.

• Relational models - give the ability to treat data more efficiently and at a higher level of abstraction. A priori knowledge on the case being solved is usually used in this kind of processing.

Image data structures are not only used for the direct representation of the acquired image, but also form the basis for more complex hierarchical methods of image segmentation. Examples of data structures are matrices, chains, graphs, lists of object properties, relational databases.

Figure 2.1: Matrix data structure

Figure 2.2: An example of a chain data structure; the start or reference pixel is marked with an arrow. The chain is 0007765555566000006444444 2221111112234445652211

• Matrices - are the most common data structures for low level image representation. The elements of the matrix are usually integer numbers obtained as output from the image capturing device, see Figure 2.1.

• Chains - are used for the description of image borders. The symbols in a chain usually correspond to the neighborhood of primitives in the image. Freeman codes [33] are often used for the description of object borders, as shown in Figure 2.2.
2.2 Types of image pixels and image data structures

Figure 2.3: Run length coding: the code is ((11144)(214)(52355)).

Figure 2.4: An example of a region adjacency graph

Figure 2.5: Description of objects using a relational structure

- Run length coding is often used to represent strings of symbols in an image matrix (for example, fax machines use it). In binary images, run length coding records only areas that belong to an object in the image; the objects are then represented by a list of lists. Each row of the image is described by a sublist, the first element of which is the row number. Subsequent terms are co-ordinate pairs; the first element of a pair is the beginning of a run and the second is the end. An example is shown in Figure 2.3. Run length coding can be used for an image with multiple brightness levels as well - in the sublist, sequence brightness must also be recorded. Pixel tables [116] (or pxy-tables, as they are called in 2-D) are link-lists of runlengths. They are used to encode multi-dimensional binary objects. Each run length consists of a n-D coordinate and the length of the run along the X dimension. Another example is the interval coding based methodology which can be applied to mathematical morphology [79, 80].

- Topological data structures - the image is described as a set of elements and their relations. An example, is the region adjacency graph model in which the
image is represented as a set of regions and each region can have its subregions, see Figure 2.4.

- Relational data structures - image information is concentrated in relations between image objects (resulting from a segmentation step), for example as shown in Figure 2.5. These data structures are used in high level image understanding algorithms.

<table>
<thead>
<tr>
<th>No.</th>
<th>Object name</th>
<th>Color</th>
<th>Min. row</th>
<th>Min. col.</th>
<th>Inside</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>sun</td>
<td>white</td>
<td>5</td>
<td>40</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>sky</td>
<td>blue</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>cloud</td>
<td>grey</td>
<td>20</td>
<td>180</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>tree trunk</td>
<td>brown</td>
<td>95</td>
<td>75</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>tree crown</td>
<td>green</td>
<td>53</td>
<td>63</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>hill</td>
<td>light green</td>
<td>97</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>pond</td>
<td>blue</td>
<td>100</td>
<td>160</td>
<td>6</td>
</tr>
</tbody>
</table>

- Hierarchical data structures - computer vision is by its nature very computationally expensive, mostly because of the huge amount of data to be processed. Besides using a parallel architecture for computation, another solution is to reduce the amount of data to be processed. This is done by hierarchical data structures which work at the finest resolution. Only those parts of the image that are necessary for processing are considered. Two typical hierarchical data structures are pyramids and quadtrees.

![T-Pyramid](image.png)

Figure 2.6: T-Pyramid.

![Quadtree](image.png)

Figure 2.7: Quadtree

1. Pyramids - A pyramid image is a sequence $M_L, M_{L-1}, ..., M_0$ of images where $M_L$ has the same dimensions and elements as the original image, and image $M_{i-1}$ is derived from image $M_i$ by reducing the resolution by a factor two. For this kind of representation only square matrices with dimensions equal to powers of two are used. $M_0$ corresponds to one pixel only. M-pyramids are used when it is necessary to work with an image at different resolutions simultaneously. An image having a one step lower resolution in a pyramid contains four times less data, so that it is processed approximately four times faster. Often it is advantageous to use several resolutions simultaneously rather than to choose just one image from the M-pyramid. Such
images can be represented using tree pyramids, T-pyramids, as shown in Figure 2.6. A T-pyramid is a tree, where every node of the T-pyramid has 4 child nodes. The number of image pixels used by an M-pyramid for storing all matrices is \( N^2 \left(1 + \frac{1}{4} + \frac{1}{16} + \ldots\right) \approx 1.33N^2 \).

2. Quadtree - are modifications of T-pyramids in which every node of the tree except the leaves has four children (NW: north-western, NE: north-eastern, SW: south-western, SE: south-eastern), see Figure 2.7. Similarly to T-pyramids, the image is divided into four quadrants at each hierarchical level. However it is not necessary to keep nodes at all levels.

The previous overview of the realm of image processing is to show the inherent complexity in parallelizing image-processing software. As a large quantity of routines in the field of low and intermediate level image processing is based on the matrix data structure and as the main focus of our research is in the combination of data and task parallelism, we will focus in the sequel on 2D gray scale images and the matrix data structure only. We inherently assume that the extension to other contents of images and their possible data structures are extensions that can be added - hopefully easily - to our framework in a later stage. Below we elaborate on the characteristics of each of the image processing levels and indicate their influence on the parallelization of image processing applications.

### 2.3 Low-level image processing

Low-level image processing operates directly on an image, to process or enhance it for human perception, or to prepare it for automatic analysis, measurement, or interpretation. Low level image processing is used in many applications, such as medical diagnosis, fingerprint analysis, industrial inspection, and the film industry. Images are first captured by cameras or other sensors and stored digitally. The stored image usually consists of a two-dimensional array of pixels (picture elements). Many low-level image processing operations assume monochromatic images and refer to pixels as image elements having gray level values or intensities. Typically, the gray scale is normalized to start at zero for black and 255 for white, which requires an 8-bit pixel. Color images typically use three values, one for each of the primary colors (red, green, and blue), or use the pixel to point to a look-up table (indexed color image).

Many different operations can be performed on an acquired image and one usually begins with low-level processing. Low-level processing uses the individual pixel values to modify the image in some way. For example, an image typically has noise, unwanted variations produced by the sensors or environment, which alter the actual pixel values. Ideally, this noise should be removed, leaving only the required image. The operations for doing this are named noise cleaning or noise reduction. Another example is the detection of edges in an image as, e.g. an aid for image segmentation and subsequent object recognition. An edge is a significant change in intensity. In edge detection, the changes in intensity in the image are highlighted in the output image. Other low-level operations include labeling pixels as belonging to a specific object and object
matching, in which an object is compared in some way with a known object. A simple form of matching is template matching, whereby the image is compared with a stored template.

Sometimes pixels that form straight lines or curves need to be localized. An efficient algorithm for this is the Hough transform [48]. The Hough transform uses pixel coordinates to identify the parameters of the equations of lines. It transforms a pixel \((x, y)\) matrix of an image into a \((\text{angle}, \text{offset})\) binning matrix in which votes are collected. High peak values in the binning matrix give a high probability that those binned pixels form a line. As both input and output space are image matrices (the neighboring elements are correlated), we will characterize the Hough transform as low-level image processing.

It may be advantageous in some applications to transform an image from the original spatial domain into the frequency domain. Such transformations operate upon all pixel values to produce a new set of values relating to the frequency content of the digitized image. Examples are the Fourier transform, Hartley transform, cosine/sine transforms [6].

As can be induced, in low-level image processing in many situations the operations on the pixels can be characterized by the statements: “for all pixels do” or “for all pixels with their neighborhoods do”. The processing is highly regular and the data on which is operated has a fine granularity. This indicates that (massively) data-parallelism will probably be beneficial.

2.4 Intermediate level image processing

Intermediate level image processing operations are symbolic processing operations. These operations work on the pixels of the objects in the image and produce more compact data structures such as lists. Examples are: region labelling, object tracking, image measurement, perceptual grouping.

Perceptual grouping refers to the human ability to derive relevant groupings or structures from input images. Recently, techniques based on perceptual grouping have produced acceptable results in many vision applications [47, 62] and have shown the promise of being extendable to more complex cases. Even though these grouping techniques result in acceptable accuracy, the computational speed on current workstations is unacceptable. Perceptual grouping is a problem with irregular computation and communication since the irregularity of the computation depends on the input image. To achieve large speedups in parallelizing such irregular problems, dynamic techniques should be used at run time. Some dynamic load balancing algorithms (task partitioning and task migration) were proposed for parallelizing perceptual grouping on distributed memory machines [83]. The experimental results were encouraging, i.e., given \(8K\) extracted line segments (detected from a \(1K \times 1K\) image), two grouping steps were completed in 0.644s and 0.585s using a task migration algorithm on a 32-node SP2 and a 32-node T3D, respectively. The same grouping steps were performed in 0.809s on a 32-node SP2 and in 0.716s on a 32-node T3D, using a task partitioning algorithm. A serial implementation of the same grouping steps required 10.550s and
10.023s on single nodes of SP2 and T3D, respectively. The observed speed-ups were above 16 for the task migration algorithm executed on 32 processors, and around 10 for the task partitioning algorithm executed on 32 processors.

Compared with low-level image processing operations, several issues must be considered in parallelizing intermediate-level algorithms, such as perceptual grouping:

- The computations are performed on symbolic data (as points, lines or areas), so the data on which is operated has medium size granularity.

- The operations are highly data dependent (i.e. the size and shape of the search is different for each symbolic datum).

- The operations may require non-neighboring pixel data.

## 2.5 High level image processing

High level image processing is knowledge-based processing and concern the interpretation of the high level information extracted from the intermediate level processing. The operations work on graphs and lists and can lead to the decision flow in an application. The available parallelism is coarse grain, and it can usually be described with a data flow graph and is highly application dependent. An example of a high-level operation is scene analysis.

Dynamic load balancing algorithms [83] offer a general framework for parallelizing many intermediate and high level vision tasks with unbalanced workloads.

## 2.6 A classification of low-level image operators

In this thesis we study methods for parallelizing image processing applications starting from the most simple level, i.e. applications which consist mainly of low-level image processing operators. Extensions to intermediate-level and high-level applications might be added in the future. Therefore, we will present in this section a classification of low-level image processing operators, without claiming that we cover the whole range or discuss all properties of them.

Low-level image processing operators use the values of the image pixels to modify the image in some way. They can be divided into point operators, local neighborhood operators, recursive neighborhood operators and global operators [38, 81], see Figures 2.8, 2.9, 2.10 and 2.11. Point operators produce an image in which each pixel only depends on the value of the corresponding pixel in the input images. The parallelization is simple and embarrassingly parallel. Neighborhood operators produce an image in which the output pixels depend on a group of neighboring pixels around the corresponding pixel in the input image. Operations like smoothing, sharpening, noise reduction, or edge detection are examples that can be parallelized by using the data decomposition paradigm. A special class are the recursive neighborhood operators which depend on
the values of a group of neighboring pixels from the input image as well as a group of neighboring pixels from the output image, which were just updated. An example is the distance transform. Global operators depend on all the pixels of the input image and they can be parallelized efficiently. On a distributed memory system, the entire image is distributed to the processors and each of them process only part of the image.

Figure 2.8: Point operators  
Figure 2.9: Neighborhood operators

Figure 2.10: Global operators  
Figure 2.11: Recursive neighborhood operators
1. **Point operators (PO)**

Point operators are the simplest image operators with regard to complexity and computing time. They require no data from other pixels to process an input pixel into an output pixel, but simply calculate a single-parameter function with the value of the input pixel as the parameter, see Figure 2.8. A large group of operators falls in this category. Point operators are used to copy an image from one memory location to another, in arithmetic, logical, and table lookup operations as well as in image composition. We will discuss in detail some arithmetic and logic operators, classifying them from the point of view of the number of images involved, as this is an important issue in the development of the corresponding skeletons.

**Arithmetic and Logic Operations**

Image Arithmetic and Logic Operations (ALO) are fundamental operations needed in almost all imaging applications for a variety of reasons. We refer to operations between a single input image and a constant as *monadic* operations, operations between two input images as *dyadic* operations and operations involving three input images as *triadic* operations.

- **Monadic image operations**

  Monadic image operators are ALO operators between a single input image and a constant. Examples are operations like addition, subtraction, division, multiplication, OR, AND, XOR between an image and a constant, or mathematical functions such as absolute value, square root, logarithm, applied to all values of an image. In Table 2.1 we present the threshold and highlight operators, in which \( a(i,j) \) and \( c(i,j) \) are the source and destination pixel values at location \( (i,j) \), respectively, and \( K1, K2, K3 \) and \( W \) are constant values.

<table>
<thead>
<tr>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
</table>
| **Threshold** | \[
  c(i,j) = \begin{cases} 
  W & : K1 < a(i,j) < K2 \\
  0 & : otherwise 
  \end{cases}
  \] |
| **Highlight** | \[
  c(i,j) = \begin{cases} 
  K3 & : K1 < a(i,j) < K2 \\
  a(i,j) & : otherwise 
  \end{cases}
  \] |
Monadic operations are useful in many situations. For instance, they can be used to add or subtract an offset value to make a picture brighter or darker.

- **Dyadic image operators**

  Dyadic image operators are arithmetic and logic functions between the corresponding pixels of two source images producing one destination image. Examples are operations like addition, subtraction, multiplication, division, minimum, maximum between the two corresponding pixel values from the input images.

  Dyadic operators have many uses in image processing. For example, the subtraction of one image from another is useful for studying the flow of blood in digital subtraction angiography. Addition of images is a useful step in many iterative imaging algorithms such as anisotropic diffusion.

- **Triadic image operators**

  Triadic operators use three input images for the computation of one output image. An example of such an operation is *alpha blending*. Image compositing is a useful function for both graphics and computer imaging. In graphics, compositing is used to combine several images into one. For example, the images may be rendered separately, possibly using different types of rendering hardware for different algorithms. In image processing, compositing is needed for any product that needs to merge multiple pictures into one final image. All image editing programs, as well as programs that combine synthetically generated images with scanned images, need this function.

  In computer imaging, the term *alpha blend* can be defined using two source images $a$ and $b$, an alpha image $\alpha$ and a destination image $c$, see Table 2.2.

  Another example of a triadic operator is the squared difference between a reference image and two shifted images, an operator used in the multi-baseline stereo vision application, described in Chapter 5.

<table>
<thead>
<tr>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha blend</td>
<td>$c(i,j) = (1 - \alpha(i,j)) \ast a(i,j) + \alpha(i,j) \ast b(i,j)$</td>
</tr>
<tr>
<td>Squared diff</td>
<td>$c(i,j) = (\text{ref}(i,j) - m1(i,j))^2 + (\text{ref}(i,j) - m2(i,j))^2$</td>
</tr>
</tbody>
</table>

2. **Local neighborhood operators (LNO)**

Local neighborhood operators (filters) create a destination pixel based on the criteria that depend on the source pixel and the value of pixels in the *neighborhood* surrounding it, see Figure 2.9. Neighborhood operators are widely used in computer imaging. They are used for edge detection, enhancement and changing the appearance of images by sharpening, blurring, crispening the edges, and
2.6 A classification of low-level image operators

noise suppression. They are also useful in image processing applications as object recognition, image restoration, and image data compression. We may distinguish two groups: **linear** and **nonlinear** filters.

**Linear filtering using two-dimensional discrete convolution**

Generally speaking, a filter in imaging refers to a transformation that produces a destination image from a source image, using some filter coefficients. A **linear** filter has the property that applying the filter to a weighted sum of input images is identical to computing the weighted sum of filtered input images.

In imaging, a two-dimensional convolution is the most common way to implement a linear filter. The operation is performed between a source image and a two-dimensional convolution kernel to produce a destination image. The convolution kernel is typically much smaller than the source image.

Mathematically, a two-dimensional discrete convolution is defined as a double summation. Given an $M \times N$ image $f(i, j)$ and $K \times L$ convolution kernel $h(k, l)$, we define the origin of each to be at the top left corner. We assume that $f(i, j)$ is much larger than $h(k, l)$. Then, the result of convolving $f(i, j)$ by $h(k, l)$ is the image $g(i, j)$ given by Formula 2.1:

$$g(i, j) = \sum_{x=0}^{K-1} \sum_{y=0}^{L-1} f(i + \frac{K-1}{2} - x, j + \frac{L-1}{2} - y)h(x, y)$$

(2.1)

$$0 \leq i \leq M - 1, \quad 0 \leq j \leq N - 1$$

In the above formula we assume that $K$ and $L$ are odd numbers and we extend the image by $(K-1)/2$ lines in each vertical direction and by $(L-1)/2$ columns in each horizontal direction. The sequential time complexity of this operation is $O(MNKL)$. As it can be observed, this is a time consuming operation, very well suited to the data parallel approach.

Generally, a convolution is defined as in the Equation 2.2.

$$c = K1 \quad (a \ast w)$$

(2.2)

The characteristic equation for a $3 \times 3$ window operator is presented in Formula 2.3.

$$c[i, j] = K1 \cdot (a[i - 1, j - 1] \cdot W1 + a[i - 1, j] \cdot W2 + a[i - 1, j + 1] \cdot W3 + a[i, j - 1] \cdot W4 + a[i, j] \cdot W5 + a[i, j + 1] \cdot W6 + a[i + 1, j - 1] \cdot W7 + a[i + 1, j] \cdot W8 + a[i + 1, j + 1] \cdot W9)$$

(2.3)
where W1, W2,...,W9 are weights or filter coefficients, which may be positive, negative or zero, and K1 is a normalization constant.

The matrix presented in Formula 2.4 is called the weight matrix or filter kernel and is important, because it fully determines the characteristics of the filter.

\[
\begin{bmatrix}
  W1 & W2 & W3 \\
  W4 & W5 & W6 \\
  W7 & W8 & W9 
\end{bmatrix}
\]  
(2.4)

The convolution operator is commutative, associative, and distributive, see Formulas 2.5, 2.6, and 2.7.

\[
a * w = w * a
\]  
(2.5)

\[
a * (p * q) = (a * p) * q
\]  
(2.6)

\[
a * (p + q) = a * p + a * q
\]  
(2.7)

The following rules summarize the behavior of this type of operator:

- If all weights are either positive or zero and form a compact kernel, the operator will blur the input image.
  Blurring is referred to as low-pass filtering. Subtracting a blurred image from the original results in the extraction of those points where the intensity is changing rapidly and is termed high-pass filtering. A high-pass filter can also be obtained by a filter composed of both positive and negative weights. If the sum of the weights is zero, the average value of the output image is zero.

- If the weight matrix, w, can be reduced to a convolution of the form \( P * Q \), where

\[
P = \begin{bmatrix}
  0 & 0 & 0 \\
  P4 & P5 & P6 \\
  0 & 0 & 0 
\end{bmatrix}
\]  
(2.8)

and

\[
Q = \begin{bmatrix}
  0 & Q1 & 0 \\
  0 & Q2 & 0 \\
  0 & Q3 & 0 
\end{bmatrix}
\]  
(2.9)
then the convolution $w$ is separable into a filter along the rows followed by a filter along the columns. The importance of this is that it is possible to apply two simpler operators in succession, respectively with weight matrices $P$ and $Q$, in order to obtain the same effect as that produced by the weight matrix $w$.

Nonlinear filtering

In contrast to linear filters, nonlinear filters are somewhat more difficult to characterize. This is because the output of the filter for a given input cannot be predicted by the impulse response. Nonlinear filters behave differently for different inputs.

A few examples of nonlinear filters are:

- Largest intensity neighborhood function $[\text{lnb}]$
  \[ \text{MAX}(A, B, C, D, E, F, G, H, I) \]
  where $A..I$ are the top-left and bottom right kernel pixels.
  This operator has the effect of expanding (dilating) bright regions and contracting (eroding) dark ones.

- Median filter $[\text{mdf}(5)]$
  \[ \text{FIFTH\_LARGEST}(A, B, C, D, E, F, G, H, I) \]
  This filter is particularly useful for reducing the level of noise in an image. Noise inevitably arises in all types of cameras and can be a nuisance if it is not eliminated by hardware or software filtering.

3. Global operators (GO)

Global operators create a destination pixel based on the entire input image, see Figure 2.10. A representative example of an operator within this class is the Discrete Fourier Transform (DFT). The Discrete Fourier Transform converts an input data set from the temporal/spatial domain to the frequency domain, and vice versa. It has a lot of applications in image processing, being used for image enhancement, restoration, and compression.

In image processing the input $f$ is a set of pixels forming a two-dimensional function that is already discrete. The formulae for the output pixel $X_{uv}$ is the following:

\[
X(u, v) = \frac{1}{NM} \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} f(x, y) e^{-2\pi i \left( \frac{ux}{N} + \frac{vy}{M} \right)} \tag{2.10}
\]
where \( j \) and \( k \) are column coordinates, \( 0 \leq x \leq N - 1 \) and \( 0 \leq y \leq M - 1 \).

The convolution theorem states that the Fourier transform of a convolution of two functions (images) is equal to the product of the Fourier transforms of each function (image) taken separately, i.e. \( \mathcal{F}(f(x, y) * g(x, y)) = F(u, v)G(u, v) \), where \( F(u, v) \) is the Fourier transform of \( f(x, y) \) and \( G(u, v) \) is the Fourier transform of \( g(x, y) \). The reversed statement holds also.

We also include in the class of global operators, operators like the histogram transform, which do not have an image as output, but another data structure.

4. Recursive neighborhood operators (RNO)

A special class of LNOs are the Recursive Neighborhood Operators (RNO) for which the update of each pixel also depends on the neighborhood pixels which have already been updated, see Figure 2.11. Usually, there are constraints in the pixel updating order. However, as the constraints are in the most cases static they can be expressed as a recursive mask such as those shown in Figure 2.12. A and B are the typical recursive masks for respectively top-left to bottom-right and bottom-right to top-left raster scan operation, the most frequently used RNOs. Examples of recursive neighborhood operators are the distance transform [5] and the recursive median filter [104].

\[ \begin{array}{c|c}
A & B \\
\hline
1 & 0 & 5 \\
2 & 6 & 7 \\
3 & 8 & \\
\end{array} \]

Figure 2.12: Recursive masks examples: pixel 0 is updated by using the value of pixels A) 1,2,3,4; B) 5,6,7; which were updated in the previous iteration

Besides data parallelism, another paradigm for parallel processing of recursive neighborhood operators is the *distributed bucket processing paradigm* [73]. A lot of RNOs can also be written as iterative algorithms that require multiple scans over the image, with each scan applying the kernel mask to every pixel in the image. An example is the erosion operation, conditionally and iteratively applied in thinning algorithms or in image skeletonization algorithms. Consider the example shown in Figure 2.13. Here we have a 4-connected erosion mask applied to an image of size \( 20 \times 20 \). The black pixels in the image are the object pixels, the white pixels are background pixels and the grey pixels are the object pixels that are eroded in the current iteration. Since the kernel operations on each pixel are identical and independent of one another, the data parallel approach can be used to parallelize the intended image operation task. As it can be observed in Figure 2.13, at each iteration only a limited number of pixels
2.7 Conclusions

change, i.e. the outer pixels of the object in the image, indicated by the grey pixels. Furthermore, the pixels that change in one iteration are neighboring pixels of the ones that changed in the previous iteration (or will be changed in the next iteration). Using a strategy of processing only the right region in an image will improve performance due to a significant reduction in the data to process. This principle was previously used by Young et al [116] and Piper et al [79, 80] in run-length coding. Van Vliet and Verwer [110] proposed queues for storing contours and faster processing of algorithms like erosion, dilation, propagation, and image skeletonization. Furthermore, the principle was applied to distance transform [111]. [73] extended this method for (heterogeneous) parallel machines.

![Figure 2.13: Multiple 4-connected erosions of a 20 × 20 image](image)

2.7 Conclusions

In this chapter, we presented a classification of the image processing operators, with respect to the type of data involved. We split the image processing operators in three large groups: low-level, intermediate-level and high-level operators. Usually, an image processing application is described as a sequence of image operators, starting from the low-level and ending with the intermediate-level or high-level operators. We refined the classification of the low-level image operators, according to the region of pixels involved in their computation. In the next chapters, we present data parallel solutions for parallelizing image processing applications consisting of low-level operators. Furthermore, solutions for parallelizing image processing applications using the task parallel paradigm are also presented. These solutions can be also applied to image processing applications composed of intermediate-level and high-level image operators.
Chapter 3

Some parallel architectures and programming models for image processing

3.1 Introduction

This chapter presents various aspects of parallelism, such as: conceptual models of parallelism, different types of parallel hardware architectures, software tools for parallelization, parallel software design patterns, and parallel programming languages. The purpose of this chapter is twofold. First to give the unfamiliar reader an overview of the realm of parallel processing in its main aspects, and, secondly, to establish the strong and weak properties of every item, so that one can prioritize them using the requirements of our application area (Chapter 1) and the requirements of image processing applications (Chapter 2). This chapter does not intend to present a complete survey of all possible paradigms in parallel hardware and software systems. It merely presents an overview of the state of the art of parallel hardware architectures and parallel programming languages related to them, with an accent on the SIMD and MIMD hardware architectures suitable for image processing, and it presents some parallel programming paradigms. Within the course of this survey various programming experiments have been performed to get a good feeling for the specific problems related with each paradigm and hardware architecture.

3.2 Conceptual models of parallel computation

Assuming that we have a parallel system composed by a set of processors we are interested in defining the parallel programming paradigms that can be used to execute a parallel program on such a system.
Parallel programming models are usually divided into two categories: data parallel and task parallel. In data parallel models, one can obtain parallelism by partitioning and distributing the data among the processors. In image processing data mean images, so the images are split and each part or sub-image is processed by a different processor. In task parallel models, one can group the instructions into tasks and assign each task to a different processor. Some features of each model are given below.

3.2.1 Data Parallel Model

The data parallel model can be characterized as follows:

- Data is distributed over the processors
- Each processor works on a different part of the same data structure

Programming with a data parallel model in a programming language is accomplished either by using message passing or by writing a program with data parallel constructs. The compiler converts the program into standard code and function calls to a message passing library for distributing the data to the processes.

3.2.2 Task Parallel Model

The task parallel model can be characterized as follows:

- The program is split into a number of tasks
- Each task is assigned to a specific processor
- The data necessary for each task is sent to the appropriate processor

3.2.3 Granularity and speed-up - basic performance considerations

At this point, it is necessary to introduce and explain the concept of a process together with its characteristics. The computation of a program (application), on a multiprocessor/multicomputer platform, can be divided into tasks that can be executed simultaneously. The size of a task can be described by its granularity. In coarse granularity, each task contains a large number of sequential instructions and takes a substantial amount of time to execute. In fine granularity, a task consists of a few instructions, or even one. Medium granularity tasks are between the previous two.

As we divide the problem into parallel tasks, at some point the communication time will dominate the overall execution time. The ratio:

\[ M_{\text{gran}} = \frac{\text{Computation time}}{\text{Communication time}} = \frac{T_{\text{comp}}}{T_{\text{comm}}} \]  \hspace{1cm} (3.1)
can be used as a granularity measure. It is desirable to maximize this ratio while maintaining sufficient parallelism.

Granularity depends on the number of processors used, when the program size remains fixed. For example, in domain decomposition, the size of a block of data used by a processor influences the granularity. In general we would like to design a parallel program in which it is very easy to vary the granularity: i.e. to design a scalable program. A classification of the granularity type of a program can be as follows:

- If $T_{\text{comp}} >> T_{\text{comm}}$, the task is considered to be coarse grained.
- If $T_{\text{comp}} > T_{\text{comm}}$ (but not tremendously so) then the task is medium grained.
- If $T_{\text{comp}} <= T_{\text{comm}}$ then the task is fine grained.

Granularity is typically controllable. Fast networks, scaling of computation to communication as a function of problem size, and CPU speed are all factors that influence the granularity of a task. Usually, fine grained tasks behave bad if scaled to many processors, while coarse grained tasks behave well.

In low-level operations, the processing is highly regular and the data on which they operate has a fine granularity. This indicates that (massive) data-parallelism will be highly beneficial, and task parallelism will probably be less beneficial to obtain a higher processing speed and a high efficiency.

In intermediate-level operations, the processing is irregular and the data on which they operate has a medium granularity. This indicates that data-parallelism may be beneficial, but also that task parallelism may contribute to a higher processing speed and a high efficiency.

In high-level operations, the processing is irregular and the data on which they operate has a coarse granularity. This indicates that data-parallelism will probably not be beneficial, and task parallelism may highly contribute to a high processing speed and high efficiency.

A measure of relative performance of a computation on a multiprocessor system versus a single processor system is the speedup factor, defined as

$$ SP(n) = \frac{\text{Execution time on a single processor}}{\text{Execution time on n processors}} = \frac{T_{\text{exec}}(1)}{T_{\text{exec}}(n)} \quad (3.2) $$

For comparing a parallel solution with a sequential solution, we will use the fastest known sequential algorithm for execution on a single processor. The underlying algorithm for the parallel implementation might be (and usually is) different. The maximum speedup is $n$ when using $n$ processors (linear speedup).

The behavior of a parallel program can be observed with visualization tools that display space-time or other forms of utilization diagrams.
The linear (ideal) speedup may be achieved when the computation can be divided in equal duration tasks (uniform load balancing), with each task mapped onto one processor, with no overhead:

\[
SP(n) = \frac{T_{\text{exec}}(1)}{T_{\text{exec}}(1)/n} = n \tag{3.3}
\]

It is reasonable to expect that some parts of the computation cannot be divided equally among the processors, and must be performed serially. If the fraction of the computation that cannot be executed by concurrent tasks is \( f \), and no overhead (interprocessor communication) appears during computation, the time to perform the computation using \( n \) processors is given by \( f \cdot T_{\text{exec}}(1) + (1-f) \cdot T_{\text{exec}}(1)/n \), as illustrated in Figure 3.1. Hence, the speedup factor is given by

\[
SP(n) = \frac{T_{\text{exec}}(1)}{f \cdot T_{\text{exec}}(1) + (1-f) \cdot T_{\text{exec}}(1)/n} = \frac{n}{1 + (n-1)f} \tag{3.4}
\]

Equation 3.4 is known as Amdahl's law and it is valid for fixed program sizes. The speedup of a program executed on \( n \) processors is limited. Even with an infinite number of processors, the maximum speedup is limited to \( 1/f \), see Formula 3.5.

\[
SP(n)_{n \to \infty} = \frac{1}{f} \tag{3.5}
\]

The efficiency, \( EF \), is defined as
\[ EF = \frac{T_{exec}(1)}{T_{exec}(n) \times n} = \frac{SP(n)}{n} \times 100\% \] (3.6)

3.2.4 Discussion

Low-level image processing, and to a certain extent intermediate-level image processing exhibit a fine granularity. The processing can be characterized by sequences of simple operations on a massive number of pixels (and their neighborhoods). In contrast, high-level image processing, and to a certain extent intermediate-level image processing exhibit a coarse granularity, as it can be characterized by a modest number of complex operations on complex data structures. So the granularity increases from fine grain to coarse grain, when the level of image processing varies from low-level to high-level.

Given a certain fixed amount of processing power, this means that the demand on the communication speed is the highest in case of low-level image processing and the lowest for high-level image processing. In principle, the parallelism in a data parallel approach applied to images can be as fine grain as a single processor per pixel. However, then, the communication speed has to match the processing speed. Data parallel image processing involves the scattering of image chunks over processors. The chunk size will hence depend on the available communication speed between the processors: the higher the communication speed, the smaller the possible chunk size.

3.3 System level models of parallel computation

The definition of a parallel computer can be confusing, but most people would agree that a parallel machine is one composed of several independent processors. Flynn [25] gives a coarse classification of parallel machines, where he analyzes the machines by the number of data streams and the number of different program streams processed at a given time. Four possible categories can be identified: *Single Instruction-stream Single Data-stream* (SISD), *Multiple Instruction-stream Multiple Data-stream* (MIMD), *Single Instruction-stream Multiple Data-stream* (SIMD) and *Multiple Instruction-stream Single Data-stream* (MISD).

Of interest to our research goal are the single-instruction stream, multiple-data stream (SIMD) and the multiple-instruction stream, multiple-data stream (MIMD) architectures and even a combination of these two architectures. In this section we will only present some SIMD and MIMD architectures, as the MISD architectures are not generally accepted as a distinct category. Some authors regard vector processors as MISD machines; others state that there is no such thing as an MISD machine, and that these machines are just a variant of SIMD machines. Still others say that they are not really parallel machines at all. Examples of vector processors are the Cray C90 and the NEC SX4. The big advantage of vector processors is that extremely good vectorizing compilers were developed for them, offering a relatively easy way to write programs that yield high performance.
3.3.1 System level classification

The above classification has been done from the hardware point of view. We are now interested in a classification from the systems point of view. At system level, we can identify elements as data spaces, programs, and communication and synchronization methods for defining the bindings between the data and the program [98]. Taking the data space and program into consideration, we are interested to find a correspondence between the Flynn [25] taxonomy and the following elements: Single Data space (SD), Multiple Data spaces (MD), Single Program (SP) and Multiple Programs (MP).

Within the MIMD classification each processor will have its own program to be executed. This could be described as a multiple program multiple data spaces (MPMD) structure. Multiple source programs exist and each processor executes its own program mapped in its own data space.

Another programming structure we may use is the single program multiple data spaces (SPMD) structure. In this structure, a single source program is written and each processor will execute its personal copy of this program in a certain data space. Usually, the SIMD architectures (as CM-2 [43] and IMAP-Vision [23]) are the hardware architectures corresponding to this model. But it should be mentioned that the SPMD model is also applicable to MIMD architectures, where we can execute the same program on each processor on a different data space.

The simplest model is, of course, the single program single data space (SPSD) and represents the sequential programming model.

The multiple program single data space (MPSD) model create several programs executed (on a single processor) in a single data space.

3.3.2 MIMD systems

MIMD systems are parallel computer structures composed of multiple independent processors. MIMD machines are the most common type of parallel machines in use today. In this architecture, each processor has its own copy of a program and it
can work on different data streams. One factor that can have great impact on the performance of parallel systems is the interconnection of the processors. Because the processors need to interact, some way to share information must be implemented. The study of the topological properties of the the communication network and the way processors share common resources, as, for example, the memory, generates further division in parallel computer architectural classification. From the memory point of view, the MIMD systems can be divided in two classes: distributed memory and shared memory systems. The shared memory architectures must solve the problem of avoiding conflicts in memory access but they have the advantage that the image is stored in the common memory and can be accessed by any processor at any time. In a distributed memory MIMD architecture the Interconnection Network (IN) connects processors to processors, each processor having its own memory. Because these systems lack a shared memory, any data needed by more than one processor must be explicitly sent from processor to processor through the IN. When the IN does not directly connect two processors that share data, this message-sending process will require intermediary processors to receive and forward the data (for example, a Myrinet switch in a Myrinet SAN network [69]). Many distributed memory architectures are geared to exploit medium-grained parallelism, because they often have more processors than task-level parallelism can effectively utilize, and because message-passing overhead would likely outweigh their execution speed-up for very fine-grained parallelism. In medium grain parallelism, the granularity of parallel execution is a few executable statements or a small subroutine. Because medium-grain parallelism requires significant message-passing among processes, the processor to processor interconnection network design is critical.

![Figure 3.3: MIMD shared memory architecture](image1)

![Figure 3.4: MIMD distributed memory architecture](image2)

Examples of distributed memory systems are network of workstations (NOW) and Beowulf-class systems like clusters of commercial off-the-shelf (COTS) PCs [2].

### 3.3.3 SIMD systems

SIMD machines were among the first parallel architectures that have been successful in digital image processing. They consist of large arrays of simple Processing Elements (PE). Machines of this type are sometimes also known as processor-array machines [7]. Because the processors of these machines operate in lock-step, i.e., all processors execute the same instruction at the same time (but on different data items), no software
CHAPTER 3. SOME PARALLEL ARCHITECTURES AND PROGRAMMING MODELS FOR IMAGE PROCESSING

synchronization between processors is required. This greatly simplifies the design of such systems. A control processor issues the instructions that are to be executed by the processors in the processor array. The SIMD architectures are very suitable for low-level image processing like point or neighborhood operators. In Figure 3.2 we show a general scheme for SIMD systems. They consist of a number of processors (usually a linear or two dimensional array), an equal number of memory modules, an interconnection network, and a Control Processor (CP).

SIMD computers were the first systems to be implemented with a massive amount of processors, and were among the first systems to provide computational power in the GFLOP range. A machine in which many data items are being operated upon simultaneously by the same instruction is thus classified as an SIMD machine. This rather general classification encompasses a wide variety of systems, all of which have some operational elements that synchronously execute identical instructions on different data items. Many current supercomputers utilize the SIMD principle of operation in some form. To efficiently process vectors and matrices, they employ vector processing units in which multiple arithmetic units operate on a large number of data items simultaneously.

The most important architectural aspect of an SIMD machine is the organization of the processor array. One such aspect is the processing element to processing element organization. In this configuration, \( N \) processing elements are connected via an interconnection network. Each processing element (PE) is a processor with local memory. The PEs execute the instructions that are distributed to the PEs by the CP via a broadcast bus. Each PE then operates on data stored in its own memory, and on data broadcast by the CP. Data is exchanged among PEs via a unidirectional interconnection network, and the I/O bus is used to transfer data from PEs to the I/O interface and vice versa. To transfer results from particular PEs to the CP, the result bus is used. Because local memory can be employed, the hardware that is used in such a machine can be constructed efficiently. In many algorithms, communication is mostly local, e.g., among the nearest neighbors.

To provide a convenient user interface for program development, debugging, and graphical display, most SIMD systems utilize a conventional computer, e.g., a UNIX workstation, as a front end. For execution, programs are downloaded from the front end to the CP, and data may be downloaded via a separate I/O device, e.g., a disk array or a camera, to the processor array. The CP then begins program execution and broadcasts SIMD instructions for the processors via an instruction broadcast bus.

SIMD is a processor structure in which a single instruction manipulates an entire data structure. High performance on such machines requires rewriting conventional algorithms to manipulate many data simultaneously by means of instructions broadcast to all processors. Although programming for these machines can be difficult in principle, in the ideal case, a serial algorithm can be converted to a SIMD algorithm by replacing each inner loop with a single broadcast instruction that implements the complete loop. The fact that an important, but limited, class of problems fits this model extremely well has provided the need for the design and construction of these machines.

A taxonomy of SIMD implementations can be based on the topology of the system [24]. The basic topology classes are linear array, mesh, tree, pyramid, and hypercube.
3.3 System level models of parallel computation

Classical SIMD systems such as the Connection Machine [43], MasPar, and IMAP-Vision are presented below.

The Connection Machine

One example of such a machine is the Connection Machine (CM-1, CM-2 and CM-200) [43], built by Thinking Machines Corporation. The CM-2 can have as many as 64K 1-bit processors (processing cells). The individual processing cells of the Connection Machine are extremely simple. The basic operation of the processing element is to read two bits from the external memory and one flag, combine them according to a specified logical operation producing two bits of results, and to write the resulting bits into the external memory and an internal flag, respectively. Even though one could specify floating point operations using the processing cells, for efficiency a floating point processor is attached to each set of 32 processing cells. Each processing cell has its own memory.

The Connection Machine is hosted by a front-end computer (usually a SUN-4). The host talks to the processing cells through a micro-controller that acts as a bandwidth amplifier between the host and the processing cells. Because the processing cells are able to execute instructions at a higher rate than the host is able to specify, the host specifies higher-level macro-instructions, that are interpreted by the micro-controller to produce the nanoinstructions for the processing cells. Upon receiving an instruction a processing unit can choose to execute it or not, depending on the current state of its flags.

Several languages have been ported to the Connection Machine. Among these, LISP and C have been extended to LISP* and C* [14] which allow parallel constructs. C* allows data structures to be spread over a set of processors, so operations on the data structures can be done concurrently, as in the case for matrix summation. C* has proved to be so useful for parallel programming with its high-level constructs. One of its most useful features is to allow the programmer to create a generic matrix machine with an arbitrary number of processors.

The MasPar MP-2

The MasPar MP-2 is the successor of the MasPar MP-1 [71], a massively parallel, distributed memory SIMD architecture. The MP-1 has $2^{14}$ PEs organized in a $128 \times 128$ mesh. The MP-2 comes with 1k, 2k, 4k and 8k processors in the PE array. At MP-1, each PE is a 4-bit processor and is connected with its eight nearest neighbors via an Xnet local neighborhood network. At MP-2, each PE is a 32-bit processor which has 16k of local memory (on a 4k MP-2). A global router provides different communication patterns. The PE array is controlled by the Array Control Unit (ACU), which can broadcast the data to all the PEs. Every PE receives the same instruction from the ACU simultaneously. The front end is usually a DEC Station running ULTRIX and provides user access, a programming environment and communication with DPU (Data Parallel Unit). The DPU, see Figure 3.5, consists of an ACU and the PE array.
The ACU executes non-parallel instructions that operate on singular data and feeds instructions for plural data to each PE.

![Diagram of MasPar architecture](image)

**Figure 3.5: MasPar architecture**

**The IMAP-Vision architecture**

The IMAP-Vision [23] is a Linear Processor Array (LPA) implemented on a PCI board. It is a parallel architecture for real-time image processing, that includes a high level language for data parallel programming: 1DC (1 Dimensional C) [107]. The IMAP-Vision consists of 256 8-bit Processing Elements (PE), controlled by a 16-bit Control Processor (CP) and it has a 10 GIPS (Giga Instruction Per Second) peak performance.

Although the IMAP chip has high levels of computational ability by integrating SIMD PE’s on a single chip, its on-chip memory is sometimes insufficient for flexible execution of complex algorithms. For this reason the IMAP-Vision board has a high bandwidth external memory interface equipped with an efficient data transfer mechanism. A bus controller allows the host PC to access data in on-chip memory and in external memory without collisions, even when a real-time vision computation is running. This mechanism not only allows the host PC to collect the results created by the processor array, but also allows the host PC to easily change the parameters in run-time.

In Figure 3.6 we show the IMAP-Vision chip configuration that integrates 32 8-bit processors and 32 8-Kbit SRAMs. Processors are connected in series to configure a one-dimensional SIMD processor array. Instructions for the processors are given from a dedicated external control processor, which is an adapted NEC microprocessor architecture.

In the IMAP-Vision chip each processor has only 1 Kbyte on-chip memory and can access 64 Byte/processor external memory in a 16 Mbit Synchronous DRAM. Data is
transferred in a 32 byte block (1 byte/processor) with a 160 Mbyte/s bandwidth. The data transfer can be issued asynchronously and concurrently with the computations. An IMAP-Vision board integrates eight IMAP-Vision chips, eight 16-Mbit synchronous DRAMs (one for each IMAP-Vision chip), a control processor, video interface, and a PCI interface LSI in a single slot PCI bus board. The board can be used with a PC or a workstation that has a PCI bus slot. Thus the IMAP-Vision board constitutes a compact system.

1DC is a data parallel language designed as an enhanced C language for a one-dimensional SIMD processor array system. The enhancement of 1DC from C is straightforward and constitutes of the following extensions:

- extended declaration of entities which are associated to the one-dimensional processor array
- extended constructs for selecting active processor groups
- extended operators for manipulating data on the processor array

An example image is shown in Figure 3.7. Entities are declared as either sep (or separate) or scalar (a scalar is the same as the declarations in the C language) as
shown on top of Figure 3.7. The sep entities are associated with the processor array and stored in the on-chip memory or the external memory, while the scalar entities are associated with the control processor and stored either in the data memory, the on-chip memory or the external memory. Each sep entity represents a linear array of scalar data where each element resides in the local memory of the corresponding processor. A sep variable may have a different value on each processor without the need for global consistency. In Figure 3.7a the sep variable $b$ is assigned the sep variable $a$.

Extended constructs for selecting active processors in the processor array, group the processors into two sets. The first set is composed of the processors that satisfy the predicate of the construct, and the second set is composed of all other remaining processors. These constructs are given a preceding $m$ for their notation such as $\text{mif}[\text{else}...]$, $\text{mwhile}...$, and $\text{mfor}(\ldots;\ldots;\ldots)$, where the only difference from standard C is that the predicate must be a sep expression. In Figure 3.7b the sep variable $c$ is assigned the sep variable $b$ only for that processors which hold a non zero value of the sep variable $a$.

```
int d;
sep char a,b,c;

b = a;

(a) mif(a) c = b;

b
\[
\begin{array}{ccccccc}
\text{1} & 0 & 0 & 1 & 0 & 1 & 0
\end{array}
\]

(b)

```

```
d = a[2];

d scalar data is stored on the control processor

(c)
```

```
a = => a; (a = < a;)

(d)

```

```
d = a[1]; (d = & & a;)

(e)
```

Figure 3.7: 1DC language extensions

Extended operators are given a preceding colon for their notation. Assume that $c_0,\ldots,c_n$ are constants, $E_{sep}$ and $E_{sca}$ are a sep and a scalar variable (expression), respectively, then $:(c_0,\ldots,c_n :)$ represents a sep constant with value $c_0,\ldots,c_n$ on the $0th,\ldots,nth$ PE, counting from the leftmost PE in the LPA. $E_{sep}:[E_{sca}:]$ extracts the scalar element of $E_{sep}$ on the $E_{sca}th$ PE, making it globally available. In Figure 3.7c the scalar variable $d$ is assigned the value of sep variable $a$ on PE number 2. The operators for accessing variables on other processors, $:<$ and $>::$, can be used in two
ways. As a unary operator, $E_{sep} =:< E_{sep}$; will rotate the values of $E_{sep}$ 1 processor to the left. In Figure 3.7d the value of sep variable $a$ is rotated 1 processor to the right. As a binary operator $E_{sep} = E_{sep} :< n$; will rotate the values of $E_{sep}$ $n$ processors to the left. The same holds for the $>: E_{sep}$, but then the rotation takes place to the right. Finally, $&E_{sep}$ and $||E_{sep}$ respectively produces a scalar entity whose values are the logical AND and OR of all the scalar elements of $E_{sep}$. In Figure 3.7e the scalar variable $d$ is assigned the logical OR of all the PE values of the sep variable $a$.

Figure 3.8 shows the current programming environment for IMAPVision systems using 1DC. The compiler not only generates code that runs on the IMAPVision system and an IMAPVision simulator, but also can produce C source code for running 1DC on native PCs and workstations.

![Diagram of IMAP-Vision programming environment]

The IMAP-Vision board comes with a MS Windows or X window (e.g. Linux) programming environment based on 1DC, which makes this parallel board a powerful tool. The X window based debuggers provide not only assembly and source level debugging facilities, but also functionalities such as interactive adjustment of variables and constants, which is useful for parameter tuning in real-time imaging applications.
IMAP-CE

The IMAP-CE [58] is the fourth generation of a series of SIMD linear processor arrays based on the IMAP (Integrated Memory Array Processor) architecture. The aim of IMAP-CE is to provide a compact, cost effective and yet high performance solution for various embedded real-time vision applications, especially for vision based driving assistance applications in the ITS (Intelligent Transportation System) fields. IMAP-CE integrates 128 VLIW processing elements, and a RISC control processor which provides the single instruction stream for the processor array. The peak performance of IMAP-CE is up to 51.2 GOPS operating at 100 Mhz.

Each PE of the processor array consists of a 2K byte local memory, 24 8-bit registers, two 1-bit mask registers, an 8-bit ALU and a multiplier. PEs are connected in series to configure a one-dimensional SIMD processor array, to which, instructions are provided by the internal Control Processor (CP), which is itself a 16-bit RISC processing unit. IMAP-CE supports the connection of sixteen chips at maximum, which forms a 2048 PE system with a peak performance of 819.2 GOPS.

Refinement of the instruction set of the IMAP-CE for accelerating SIMD operations has been done. Code patterns which frequently appear are identified, and were checked on the possibility of replacement by wired-logic instructions. As a result, special wired-logic instructions for PE indexed address generation, PE conditional branch and label propagation, are reinforced upon the existing conventional RISC instructions.

Current trends in microprocessor technology

In the last few years, new developments in the multimedia processing domain have generated a continuous request for faster processing. Several microprocessor manufacturers have enhanced their general-purpose processors with new features dedicated to the efficient handling of multimedia (audio and video) data [13, 4]. In order to exploit the high intrinsic data parallelism of multimedia processing, a SIMD solution has been adopted. Intel Corporation has developed the MMX/SSE technology [13] for the x86 architecture, while Motorola has developed the AltiVec [1] technology. Other examples are Hewlett Packard with MAX (Multimedia Accelerated eXtensions) [59] technology for the PA-RISC processor architecture and Sun with VIS [113] for the SPARC processor. Below, we present some features of the two most predominant technologies, the MMX/SSE technology and the AltiVec technology.

The MMX technology

MMX technology is designed to accelerate applications in multimedia and communication. The technology includes new instructions and data types that allow applications to achieve a new level of performance. It exploits the parallelism inherently present in many multimedia and communications algorithms.
3.3 System level models of parallel computation

Intel's MMX Technology adds several new data types and 57 new instructions specifically designed to manipulate and process video, audio and graphical data more efficiently. These types of applications often use repetitive loops that, while occupying 10 percent or less of the overall application code, can account for up to 90 percent of the execution time.

The new data types allow the handling of 64-bit data. This is accomplished by reassigning 64 bits of each of the eight 80-bit floating-point registers as MMX register. The 64-bit registers may be thought as eight 8-bit bytes, four 16-bit words, two 32-bit double words, or one 64-bit quadword [13].

The highlights of the technology are:

- Single Instruction, Multiple Data (SIMD) technique
- 57 new instructions
- Eight 64-bit wide MMX technology registers
- Four new data types

MMX technology provides several new features to the existing processor architecture:

1. It supports a new arithmetic capability known as saturation arithmetic. Saturation provides a useful feature of avoiding wraparound artifacts. One important point of MMX is the support of saturated operations. If an operation would cause an overflow, the value stays at the highest or lowest possible value for the data type: If one uses byte values one gets normally 250+12=6. This is very annoying when doing color manipulations or changing audio samples, when one has to do a word add and check if the value is greater than 255. The solution is saturation: 250+12 gives 255. Saturated operations are supported by the MMX unit.

2. The new 57 MMX instructions allow parallel processing of multiple data. These instructions may be grouped into 7 instruction categories: Arithmetic, Comparison, Conversion, Logical, Shift, Data Transfer and Empty MMX state. All MMX instructions start with a 0xFH byte and work on different data types.

3. The new 8 MMX registers are physically assigned to the Floating Point (FP) registers. Switching to MMX mode is done automatically when an MMX instruction is executed. Switching back to FP mode requires an execution of the EMMS (MMX mode exit instruction) instruction. Switching between the two modes is expensive and it is recommended not to intermix MMX and FP instructions in the same piece of code. Frequent transitions may result in significant performance degradation. It is recommended to stay in the MMX mode as long as possible during the application run. Before using the FP arithmetic the FP/MMX mode must be switched to FP, by executing EMMS, to avoid FP errors. It should also be mentioned that MMX registers cannot be used to address memory.
In [49] the reader may find a detailed description of MMX features, data types and all MMX instructions.

Intel SSE

The SSE (Streaming SIMD Extensions) [4], introduced with the Pentium III processor, enhances the Intel x86 architecture with:

- 8 new 128-bit SIMD floating-point registers.
- 50 new instructions that work on packed floating-point data.
- 8 new instructions designed to control cacheability of all MMX and 32-bit x86 data types, including the ability to stream data to memory without polluting the caches, and to prefetch data before it is actually used.
- 12 new instructions that extend the MMX instruction set.

These new features enable the programmer to develop algorithms that can mix packed, single-precision floating-point and integer using both SSE and MMX instructions respectively. Since SIMD floating-point registers are a separate register file, MMX or floating-point instructions can be mixed with SSE instructions without execution of a special instruction such as EMMS.

The current generation of x86 processors, the Pentium IV, has new SIMD instructions added to the previous MMX and SSE instruction sets. The new SSE2 instruction set adds new data types:

- 128-bit packed double-precision floating-point (two double-precision floating-point values packed into a double quadword).
- 64-bit (quadword) integer (signed and unsigned).
- 128-bit packed byte (word, doubleword, quadword) integers (signed and unsigned).

These enhancements attempt to close the gap between Intel's and Motorola's SIMD integer performance, by allowing operations on 128-bit data.

The AltiVec technology

Motorola's AltiVec technology [1] expands the current PowerPC architecture through the addition of a 128-bit vector execution unit, which operates concurrently with the existing integer and floating point units. This new engine provides highly parallel operations, allowing for the simultaneous execution of up to 16 operations in a single clock cycle.
AltiVec technology is a short vector parallel architecture. Depending on data size, vectors are 4, 8 or 16 elements long. This in contrast with the long vector architectures of supercomputers that were popular in the 1980s. Vector sizes for those machines ranged to hundreds of elements. The long vector approach of supercomputers, while useful for scientific calculations, is not optimal for the communications, multimedia and other performance-driven applications targeted by Motorola with its AltiVec technology.

AltiVec technology operations are performed on multiple data elements by a single instruction. This is often referred to as SIMD (single instructions, multiple data) parallel processing. AltiVec technology offers support for:

- 16-way parallelism for 8-bit signed and unsigned integers and characters
- 8-way parallelism for 16-bit signed and unsigned integers
- 4-way parallelism for 32-bit signed and unsigned integers and IEEE floating-point numbers

MMX has less capabilities than AltiVec. First of all, it is not a separate processing unit. MMX and SSE (which is MMX2) are both extra instructions that cause mode switches in the Floating Point registers of the P3 & P4 and then use those registers for their calculations. Granted, SSE is a much better implementation than the original MMX because it takes fewer clock cycles to perform the mode switching, but it is still not a separate processing unit.

The AltiVec unit is a dedicated vector processing unit. Its design is very much inline with the CPU’s in Cray systems. The fact that AltiVec is a separate processing unit provides it with several clear cut capabilities that MMX and SSE cannot beat. The AltiVec unit is made up of 128-bit registers, and as a result of that can operate on greater chunks of data at a time. Because MMX and SSE are tied to the 64-bit FPU of the chip, MMX and SSE are also limited to 64-bit execution.

AltiVec is a better multimedia co-processor implementation than MMX or SSE. MMX and SSE were tagged on to the P2, P3, and P4. AltiVec was designed in advance, and was coming to the PowerPC architecture whether or not Apple decided to pick it up in its product line.

To summarize, the characteristics of AltiVec and MMX are given in Table3.1

Apple claims that on average the G4 is up to three times faster than a PIII/600Mhz when using one of Intel’s own benchmarking applications - the Intel Signal Processing Library. Performance measurements [82] showed that a Pentium III/600Mhz is on average 66% slower than a G4/500, or put in another way, the G4 is three times faster.

To summarize, we can conclude that different microprocessor manufacturers have found different solutions for enhancing the microporcessor features, which do not converge to a stable architecture model. This is the reason why very little effort has been
Table 3.1: MMX - Altivec features

<table>
<thead>
<tr>
<th>MMX</th>
<th>Altivec</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 x 128 Bit FP Registers</td>
<td>32 x 128 Bit FP Registers</td>
</tr>
<tr>
<td>8 X 64 Bit Int Registers</td>
<td>32 X 128 Bit Int Registers</td>
</tr>
<tr>
<td>Fewer data types</td>
<td>More data types</td>
</tr>
<tr>
<td>Instructions destroy source</td>
<td>Instructions save source</td>
</tr>
<tr>
<td>Some reordering/resizing</td>
<td>Full permute</td>
</tr>
<tr>
<td>Limited cache control</td>
<td>Larger Cache, more control</td>
</tr>
<tr>
<td>Narrower Datapath</td>
<td>Wider data path</td>
</tr>
<tr>
<td>Fewer Instructions</td>
<td>More Instructions + modifiers</td>
</tr>
<tr>
<td>Older instruction set design</td>
<td>Specialized DSP instructions</td>
</tr>
<tr>
<td>More heat, cost, power</td>
<td>Less heat, cost, power</td>
</tr>
<tr>
<td>Larger Package</td>
<td>Smaller package</td>
</tr>
<tr>
<td>Single-sourced closed design</td>
<td>Multi-sourced open design</td>
</tr>
<tr>
<td>Worse tools</td>
<td>Better tools</td>
</tr>
<tr>
<td>Significantly Slower design</td>
<td>Far faster design</td>
</tr>
</tbody>
</table>

put in developing software, specific tools and vectorization compilers for these microprocessor architectures. Because of the lack of these kind of tools the use of these new technologies is not very large. Still, the MMX like technology appears to gain wide acceptance. Since the programming effort is very big due to manual coding of MMX assembly, Intel has released a suite of libraries addressing frequently occurring problems in several fields: image processing, pattern recognition, JPEG coding and decoding, signal processing. Some efforts are made in introducing the MMX technology into mainstream programming by designing vectorizations that can turn scalar C code into parallel MMX assembly code without user intervention.

3.3.4 Discussion

MIMD systems can nowadays easily be built from off the shelf commodity products. Among the processor boards, boards can be found with two or four processors. However, these boards are still somewhat rare in comparison with the single processor boards and the software support for such a board (OS, compiler, run time system) is usually underdeveloped and not standard. For normal PCs, single processor boards with still increasing processor speed, are more common. At the same time, the interconnection speed between processor (in the range of Gigabits/s) is such that the granularity of the problems that can be tackled becomes finer than a few decades ago. Hence, one can conclude that a distributed memory system, based on off-the shelf processor boards and interconnection network (Clusters Of Workstations), is the best candidate to focus on for parallel image processing, even for data parallel processing of low and intermediate level image processing algorithms. The key point of success of such a system now lies with the communication software. This is treated in the next section.
For real-time applications, the speed of such systems might not be good enough. In the past, SIMD systems had been developed especially for image processing, as building clusters of workstations could not bring relief at that time. Currently, processor manufacturers have entered the SIMD design trajectory, by adding SIMD instructions, e.g., MMX, to the normal processor instructions with the speed-up of multi-media and computer graphics in mind. The IMAP-Vision add-on board and its successor, the IMAP-CE chip are examples of SIMD systems specially made for low-level image processing that could be used in conjunction with a normal PC for intermediate and high-level image processing. In contrast with the SIMD instructions for normal processors, the software environment for the IMAP-Vision (and IMAP-CE) is very mature, stable and focussed on image processing. On the other hand, manufacturers like Intel, have developed special dedicated libraries, such as an image processing library, to shield the SIMD instructions from the user. To answer the question if and how these developments can be incorporated into our parallel framework, we have performed some experiments, which are reported in Section 3.7.

3.4 Implementation level models

While processor and memory are the fundamental conceptual units of parallel hardware, the process (task) is the fundamental conceptual unit of parallel software. Roughly speaking, a process is an instance of a program that is executed on a physical processor.

The implementation of the SPMD or MPMD models on a distributed memory system, can be made possible by data communication and synchronization between processes. The most important methods for communicating data are using shared variables or by communicating variables. In shared variables communication, processes have direct access to shared variables which may reside also in the local memory of processors. Different mechanisms are used to prevent the concurrent update of shared variables.

3.4.1 Message passing in imaging

Message passing is the most commonly used method for communicating variables. It is mostly used for programming distributed-memory systems. In message passing, the processes coordinate their activities by explicitly sending and receiving messages. The message passing model is defined using the following items:

- The set of processes using only their local memory
- The processes communicate by sending and receiving messages
- The data transfer requires cooperative operations to be performed by each process

Programming with message passing is done by linking with and making calls to libraries which manage the data exchange between processors. Message passing libraries are available for most modern programming languages.
CHAPTER 3. SOME PARALLEL ARCHITECTURES AND PROGRAMMING MODELS FOR IMAGE PROCESSING

In the message-passing library approach to parallel programming, a collection of processes (tasks) executes programs written in a standard sequential language augmented with calls to a library of functions for sending and receiving messages. The set of communication operations that are allowed by an implementation of the message passing model form the components of a message passing library. Examples of message passing libraries include public domain packages that do not target a specific machine (PVM, CRL, P4, MPI, etc.) as well as machine dependent vendor implementations.

In this section, we outline the basic concepts in message passing computing. We explain the basic structure of message passing programs and how to specify message passing between processors. We present three specific systems, PVM (Parallel Virtual Machine), MPI (Message Passing Interface), and CRL (C Region Library). PVM is presented because of its rather widespread use. MPI is presented because it is a de-facto standard. CRL is presented because it is a simulated shared memory system library, designed for distributed memory systems. Finally, we present some experiments using these three message passing libraries.

PVM

In PVM, the programmer splits the problem into separate programs or tasks. Each program is written in C or Fortran and compiled to run on specific types of computers in the network. If the network is homogeneous (consists of computers of the same type), the programs have to be compiled only once for that type of computer. In a network of workstations, each workstation has access to the file system containing the compiled programs. It is necessary to ensure that the executables exist for the specific workstations that will execute the programs.

The set of computers used in a parallel program must be defined first. They form the Parallel Virtual Machine - PVM. The routing of messages between computers is done by the PVM daemon process installed by PVM on the computers which belong to the virtual machine.

Processes can be started dynamically. PVM programs are usually organized in a master-slave way, whereby a master program is first executed and spawns all slave processes. To start executing one or more identical processes (tasks), the PVM routine \texttt{pvm\_spawn()} is used. All PVM system calls start with the letters \texttt{pvm}. There are C and Fortran bindings of PVM. Programs communicate by message passing using PVM library routines such as \texttt{pvm\_send()} and \texttt{pvm\_recv()}, which are embedded into the programs. All PVM send routines are nonblocking (asynchronous) while PVM receive routines can be either blocking (synchronous) or nonblocking. All send and receive routines return information concerning the success of the calls. In PVM, broadcast, scatter, gather, and reduce operations, \texttt{pvm\_bcast()}, \texttt{pvm\_scatter()}, \texttt{pvm\_gather()}, and \texttt{pvm\_reduce()} can be used in a group of processors after that group is formed. A process can join a group by calling \texttt{pvm\_joingroup()}. 
3.4 Implementation level models

MPI

In the MPI programming model, a task consists of one or more processes that communicate by calling library routines to send messages to and to receive messages from other processes. In most MPI implementations, a fixed set of processes is created at program initialization, and one process is created per processor. However, these processes may execute different programs. Hence, the MPI programming model is sometimes referred to as a multiple program multiple data (MPMD) model to distinguish it from the SPMD model in which every processor executes the same program.

As PVM, MPI provides library routines for message passing. Although MPI has a large number of routines, we will discuss only a subset of them. MPI was developed to be as portable as possible. As with PVM, bindings to C and Fortran are available in MPI. All MPI routines start with the prefix MPI, and the next letter is capitalized. There are differences between PVM and MPI. A significant difference from PVM is that only static process creation is supported in MPI version 1. This means that all the processes must be defined prior to execution and started together. There is no equivalent to the PVM call *pvm.spawn(*). Initially, all processors are enrolled in a global communicator called *MPI.COMM_WORLD*, and each process is given a unique rank, a number from 0 to *P* − 1, where *P* is the number of processors. Other communicators can be established for groups of processors. A group is a collection of processors. Message passing is done via sending and receiving messages. There are several versions of send and receive, like blocking and non-blocking ones. A communication routine is blocking if the completion of the call is dependent on certain events. For sends, the data must be successfully sent or safely copied to system buffer space so that the application buffer that contained the data is available for reuse. For receives, the data must be safely stored in the receive buffer so that it is ready for use. A communication routine is nonblocking if the call returns without waiting for any communication events to complete (such as copying of message from user memory to system memory or arrival of message). It is not safe to modify or use the application buffer after completion of a non-blocking send. It is the programmer's responsibility to insure that the application buffer is free for reuse. Non-blocking communications are primarily used to overlap computation with communication to effect performance gain.

Processes can use point-to-point communication operations to send a message from one named process to another. A group of processes can call collective communication operations to perform global operations.

3.4.2 Shared address space mechanism

CRL

The C Region Library (CRL) [50] is an all-software Distributed Shared Memory (DMS) system intended for use on message-passing multicomputers and distributed systems. Parallel applications built on top of CRL share data through regions. Each region is an arbitrarily sized, contiguous area of memory. The programmer defines regions, and includes annotations to delimit accesses to those regions. Regions are cached in
the local memories of processors. Cached copies are kept consistent using a directory-based coherence protocol. Coherence is provided at the granularity of regions instead of at items such as memory pages, cache lines, or other arbitrarily chosen fixed-size units. This is the reason why CRL avoids the adjacent problems of false sharing, for coherence units that are too large, or inefficient use of bandwidth for coherence units that are too small.

Three key features distinguish CRL from other software DMS systems. First, CRL is system- and language-independent. Providing CRL functionality in programming languages other than C should require little work. Second, CRL is portable. By employing a region-based approach, CRL is implemented entirely as a library and requires no functionality from the underlying hardware, compiler, or operating system beyond that necessary to send and receive messages. Third, CRL is efficient. Very little software overhead is created between applications and the underlying message-passing mechanisms. While these features have occurred in isolation or in tandem in other software DMS systems, CRL is the first software DMS system to provide all three in a simple, coherent package [50].

Because shared address space or shared memory programming environments like CRL provide a uniform model for accessing all shared data, whether local or remote, they are relatively easy to use. In contrast, message-passing environments burden programmers with the task of orchestrating all interprocessor communication and synchronization through explicit message passing. While such coordination can be managed without adversely affecting performance for relatively simple applications (e.g., those that communicate infrequently or have relatively simple communication patterns), the task can be far more difficult for large, complex applications, particularly those in which data is shared at a fine granularity or according to irregular, dynamic communication patterns.

Applications using CRL can effect interprocessor communication and synchronization either through a relatively typical set of global operations (barrier, broadcast, reduce) or through operations on regions.

A region is an arbitrarily sized, contiguous area of memory identified by a unique region identifier. A region identifier is a portable and stable name for a region; in order to access a region, a processor node must know the region's identifier. New regions can be created dynamically (using rgn_create) and can later be removed when they are no longer needed. Blocks of memory comprising distinct regions (those with different region identifiers) are non-overlapping.

In order to access the memory associated with a region, it must be mapped into the local address space. To map a region, one calls rgn_map with the region identifier of the region to be mapped; the function returns a pointer to the base of the region's data area.

CRL also provides operations for sharing regions. Specifically, CRL provides library calls that can be used to delimit operations—portions of the code where regions will be read from or written to. The library calls initiating and terminating operations that are required to maintain the consistency of shared data. The coherence model provided by CRL considers entire operations on regions as indivisible units. From
this perspective, CRL provides sequential consistency for read and write operations in the same sense that a sequentially-consistent hardware-based DSM does for individual loads and stores.

The library has routines for initialization, basic region functions (for creating, destroying, identifying a region), functions for mapping and unmapping regions, region read and write operations and global synchronization functions.

### 3.4.3 Discussion

From the message passing communication libraries, PVM was the first standard available, later followed by MPI, which has a more restricted functionality. However, MPI is an industrial standard, most likely to be supported by computer manufacturers. In the MPI-2 standard the functionality is increased. This leads to a slight preference for MPI.

The shared address space mechanism CRL offers from the programming point of view a very nice solution. As with data caches in processors, it takes care of all data coherencies for the user, that can simply assume that his data is in memory and correct.

Our parallel framework will be built on top of existing communication libraries. Consequently, standardization and sustainability (how long will the product be supported on the market) and as much as possible ease of programming will be the main issues.

### 3.5 Parallel design patterns

Parallel design patterns represent for parallel computing what design patterns represent for object-oriented programming. A design pattern indentifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue. The main goal is to offer the user a general model for solving a particular class of problems in the parallel computing field, while keeping a good balance between the simplicity of the interface presented and the efficiency of their implementation. Parallel design patterns, often also called algorithmic skeletons are not restricted to object-oriented programming languages. Their principle can be applied in any environment. In the following, we will present an example of such a pattern, especially suited for image processing, called distributed bucket processing. A more general presentation of parallel design patterns for low-level image processing will follow in Chapter 4.

Large datasets, such as pixels and voxels in 2D and 3D images can usually be reduced during their processing to smaller subsets with less datapoints. Such subsets can be the objects in the image, features -edges or corners- or more general, regions of interest. For instance, the transformation from a set of datapoints representing an image, to one or more subsets of datapoints representing objects in the image, is due to a segmentation algorithm and may involve both the selection of datapoints as well as a change in datastructure. The massive number of pixels in the original image,
points to a data parallel approach, whereas the processing of the various objects in
the image is more suitable for task parallelism.

3.5.1 Distributed Bucket Processing

In this section we describe, within the context of a framework for parallel image
processing, the Distributed Bucket Processing paradigm (DBP) [73, 128] which focuses
on an array of buckets that can be distributed over a number of processors. Each
bucket contains pointers to the data. The benefit of this approach is that the processor
activity remains focussed on the datapoints that need processing and, moreover, that
the load can be distributed over many processors, even in a heterogeneous computer
architecture.

The idea behind Distributed Bucket Processing is to combine data reduction and data
parallelism strategies. It is geared towards iterative image processing algorithms where
only a subset of the image data is processed. Figure 3.9 shows the global steps in using
bucket processing. First a global scan over the image is performed to collect interesting
data. This data could for example contain the border pixels of the objects in an image
that needs to be eroded. Note that while collecting the interesting data in the image,
the first iteration of an operation on the image can be performed simultaneously.

1. scan image to collect data of interest
2. put data to process in a bucket
3. while bucket not empty
   - process data in bucket
   - put new(ly generated) data to process in bucket
   endwhile

Figure 3.9: Distributed bucket processing steps.

The data of interest are collected in one or more buckets. These are just data structures
to store, for instance, the subset of pixels of the image that needs to be processed.
During processing, new data can be generated and put in the bucket(s). Data parallel
processing of a bucket is possible since an ordering of elements in the bucket is neither
specified nor guaranteed and thus the bucket data structure can be distributed over
multiple processors.

Processing only a subset of the data can be done in parallel because the operations on
the pixels are independent. Figure 3.10 shows a bucket distributed over four processors.
A write paradigm is enforced which means that a processor can only read from its
own part of the bucket but can write to the whole bucket, also to parts mapped on a
different processor. The mapping function determines how the bucket is mapped on the
processors. Usually a bucket is mapped in the same way as the corresponding images
are mapped on the processors. So when writing data to a bucket, pixel coordinates
are used to determine the destination processor within the bucket.
In parallel processing it is desirable to map the image data differently on the processors depending on the image processing level as each level interprets the image data differently. For example at the intermediate and high-level processing levels it is generally not necessary to process all image pixels. In Figure 3.11 an image is initially mapped column-wise on the available region processors \( P_1 - P_5 \). After the low-level processing, it is convenient to have a different mapping where the different regions of the image are mapped to corresponding processors \( P'_1 - P'_5 \). Another example is shown in Figure 3.12. Here the low-level processors \( P_1 - P_5 \) are used to recognize two objects in the image. These two objects are then mapped to (higher level) processors \( P'_1 \) and \( P'_2 \) for further processing.

In [73] it is explained that low-level image processing fits best with SIMD architectures while intermediate and high-level image processing fits MIMD architectures.
The mapping problem described above is referred to as the SIMD-MIMD gap. The Distributed Bucket Processing (DBP) technique can be used for bridging this gap. The idea is to collect, at some stage during the low-level processing, interesting pixels (e.g. their values and image addresses, or even neighborhood pixels) that form a set of data that needs to be processed later. Collecting the interesting pixels, i.e. pixels that are likely to change in the next processing phase, is usually an iterative process where pixels are processed and new interesting pixels are created and added to the set. Finally, the set of pixels holds the desired subset of the full image, for example all pixels of a recognized object in the image.

With respect to architecture, there are a number of ways to apply the bucket processing concept to a combined SIMD-MIMD architecture in order to bridge the SIMD-MIMD gap. Bucket processing may be emulated in software on either SIMD or MIMD. Emulation of bucket processing on a SIMD machine has shown to offer an interesting way of handling irregular problems [53], e.g. the notorious bad behavior of if-statements in SIMD systems can be circumvented by processing both branches of an if-statement in parallel by putting the jobs to be done in then and else bucket sets. Another option of applying the bucket processing concept is to add special hardware to an SIMD or MIMD component, or add a separate bucket processing architecture component between SIMD and MIMD. The IMAP-CE SIMD architecture is optimized for bucket processing.

3.5.2 Discussion

Distributed Bucket Processing (DBP) can be implemented on MIMD systems as well as on SIMD systems, such as the IMAP-Vision or IMAP-CE, as they both are capable to indirectly address memory, allowing the implementation of queues or stacks. The special SIMD instructions of modern processors are not capable to implement DBP, as they lack the indirect addressing capability.
DBP can be implemented as a library for our parallel framework either for MIMD, SIMD or for a heterogenous MIMD/SIMD system.

3.6 Parallel programming languages

High performance computers have evolved rapidly over the past decade. During the last few years, the majority of HPC systems in use were distributed memory computers, ranging from small clusters of workstations or PCs to large-scale machines from IBM, Cray, Intel, and other vendors. Shared memory computers are wide-spread available; in particular, multiprocessor workstations and PCs are becoming commonplace, and are being used as components of distributed memory systems. Thus, computing platforms are now combining elements of both shared and distributed memory. Even more recently, the underlying infrastructure to support the creation of wide area networks of computers, including large-scale machines, has been developed. Research projects such as GLOBUS [37, 30], Legion [70], MOL [18] aim to provide a user with transparent access to such a "computational grid", which is essentially a distributed, heterogeneous collection of parallel computers. Such grids pose many new requirements with respect to programming models, compilation strategies, and execution environments.

During that same period, scientists at research laboratories were required to put a very large effort in porting codes to new architectures in an attempt to fully utilize their performance potential. Not only are they forced to struggle with immature hardware and system software, but the large performance gap between what is theoretically possible and what has actually been achieved has resulted in extensive efforts to re-code applications by-hand using machine-specific optimizations. While these efforts did narrow the performance gap, many major supercomputers have remained on the sideline. In anticipation of the increasing complexity and size of future applications, industrial users will not be able to afford such porting efforts every few years. Instead, in order to protect investments in code maintenance and development, technology must be developed to transform the parallelization process such that it 1) requires less time and effort, 2) generates code that performs well, 3) is able to handle a wide range of existing legacy applications, and 4) is portable to future machines.

Currently, there are three major approaches besides hand-coding for porting applications to parallel architectures. These include:

- Relying on vendor-supplied parallelizing compilers/tools.
- Annotating/rewriting application using data- and task-parallel directives/languages [115, 109, 41].
- Rewriting application codes using semi-custom building blocks or libraries [13, 59, 1, 113].

These approaches differ from one another in terms of relevant target applications, pertinent architectures, requirements for user involvement, as well as maturity. Therefore, the effectiveness of these approaches is a worthwhile study, especially for users
interested in modernizing large legacy applications onto evolving parallel computing architectures.

In this section we investigate the benefits and drawbacks of these approaches.

### 3.6.1 Automatic parallelization

From the user's point of view automatic parallelization is an important issue in parallel processing. If the applications can be efficiently parallelized by an automatic system then the user can focus on what is being computed rather than how it is being computed [36].

Implementing parallelism in applications requires specialized development and debugging tools. Some hardware vendors, like SUN, offer such tools - compilers, debuggers, and profilers - needed to support the development of multiprocessing and multithreaded programs. In this section, we present some performance issues of a parallelizing compiler, the SPARCompiler SC4.0

The automatic parallelization step is performed by the optimizer module of the compiler. The optimizer performs the following actions:

- Examine all loops in the program.

- Determine which loops have iterations that can be executed independent of each other.

- Transform the loop so that iterations are divided among multiple threads.

The compiler can use automatic parallelization to compile code to utilize a number of processors simultaneously in a shared-memory multiprocessor (SMP) machine. A SMP machine has a number of processors that share the machine's main memory and has a single operating system. Examples of SMP machines are the Sun Ultra 60 and the SGI O200 which can have up to two processors, while the Sun Enterprise 10000 can have up to 64 processors.

Automatic parallelization is targeted to code segments that perform most of their work in loops. The compiler splits a loop up so that a number of iterations of the loop can be executed on separate processors. For example consider the piece of code in Figure 3.13.

The diagram in Figure 3.14 shows conceptually how a compiler might parallelize this code to run on a four processor machine.

The code that is not in the loop is executed on a single processor while the code in the loop is executed across four processors. When the loop is complete the code runs on just one processor until the next loop, which might be executed in parallel, is encountered. Each part of the code that runs on a separate processor is called a thread, the creation and destruction of threads adds an overhead to the runtime of the code. Not all loops can be split up and run on separate processors. Examples of
such loops are illustrated below as well as procedures for modifying them so that they can be parallelized.

There is no guarantee that automatic parallelization will reduce the run time for a piece of code. For automatic parallelization to be efficient the code will need to perform most of its work within loops that the compiler can parallelize. Automatic parallelization can be used with C, FORTRAN or C++ depending on the particular compiler.

Setting the number of threads to use larger than the number of processors the machine has available will have a negative effect on the performance. The machine will suffer from the overhead of switching between the various threads that have been requested. Similarly, if there are a number of other jobs already running on the machine then specifying more threads than processors that are available will slow down all jobs.
Loops that cannot be parallelized

In order to illustrate how the automatic parallelization works and how efficient it is we will look at some sample pieces of real code.

Low trip counts

The first example is of a loop that counts only a few iterations. It is not parallelized because the overhead in creating a thread for it to run in parallel prevents any speed up. The loop is shown in Figure 3.15.

```c

{ 
    FOR (k = 1; k < 8; k++) 
   .buf4[k] = buf4[k]/nrc;

}

Figure 3.15: Automatic parallelization - fragment code 2
```

The compiler can decide that it is not profitable to parallelize this loop because it knows the number of iterations beforehand. In general, the compiler will not know the number of iterations of a loop until run time. For example, if the loop had been written as in Figure 3.16, were \( n \) is not known until run time, the compiler will parallelize the loop, but if \( n \) is small it may turn out to be inefficient.

```c

{ 
    FOR (k = 1; k < n; k++) 
   .buf4[k] = buf4[k]/nrc;

}

Figure 3.16: Automatic parallelization - fragment code 3
```

Loops with multiple exits

The code from Figure 3.17 is an example of a loop with multiple exits.

One simple test to check whether a loop can be parallelized is to verify if the iterations of the loop can be executed in any order, or if the statements inside the loop can be executed in any order. If the iterations of the loop can be executed in any order then they can run on separate processors simultaneously. The same holds when the statements inside the loop can be executed in any order. In code fragment presented in Figure 3.17, each iteration has to be executed in the specified order, because it is not possible to know when to jump out of the loop. Similarly, a while loop cannot be parallelized since the compiler cannot know when to exit the loop.
Loops containing function calls

The following loop, shown in Figure 3.18 cannot be parallelized by the compiler because it contains a call to a sub-routine.

```
FOR (i = 1; i < nmc; i++) {  
    ...........
    imgmdl2(h,x,nrow1,iseed,nse);
    ...........
}
```

Figure 3.18: Automatic parallelization - fragment code 5

The compiler cannot know if the sub-routines can be executed in parallel. Some functions can run in parallel such as the function to take the square root of a number. Taking the square root of one number will not affect taking the square root of another number. An example of a function that cannot be executed in parallel is the random number generator. Random number generators generally use sequences of numbers, they use a stored number to generate the next number in the random sequence. Each time it is called it updates the stored number. If the random number generator is called twice at the same time by separate threads they will both find the same stored number and return the same "random number". (For a detailed discussion on random number generators consult Numerical Recipes in Fortran [22]). One possible solution to parallelize loops with functions is to have the compiler replace the function call by inline code. In this way the compiler can analyse the code completely within the loop and decide whether it can be parallelized.

Flow dependency

The following example, shown in Figure 3.19, contains a flow dependency.

This is called a reduction since we are effectively reducing the array xy to the scalar quantity xyvar. The above loop cannot be executed in parallel because of the scalar quantity xyvar. If multiple iterations of the loop are executed at once they will try to access xyvar simultaneously. Consider two iterations of the loop being executed
simultaneously. When they execute the line \( xyvar = xyvar + xy[ik2+20][i2+20]^2 \) both will load the same value for \( xyvar \) from memory. Each will perform the addition \( xyvar + xy[ik2+20][i2+20]^2 \) and try to store the "new" value for \( xyvar \) in memory. But which value of \( xyvar \) will be stored? If one value is written first it will be lost because the second value will replace it. No matter in which order they are written, the value for \( xyvar \) will be incorrect. The value \( uxy \) in the first line of the loop does not cause a flow dependency, it remains constant for each iteration of the loop. It is effectively a "read only" value.

A possible solution is to change the loop as shown in Figure 3.20.

```
FOR (i2 = 1; i2 < ncol1; i2 + +)
FOR (ik2 = 1; ik2 < nrow1; ik2 + +)
    xy[ik2+20][i2+20] = xy[ik2+20][i2+20] - uxy;
    xy_temp[ik2+20][i2+20] = xy[ik2+20][i2+20]**2;

FOR (i2 = 1; i2 < ncol1; i2 + +)
FOR (ik2 = 1; ik2 < nrow1; ik2 + +)
    xyvar = xyvar + xy_temp[ik2+20][i2+20];
```

Figure 3.20: Automatic parallelization - fragment code 7

We did split the loop into two loops and added an extra array, \( xy_temp \), to hold the value of \( xy[ik2+20][i2+20] * * 2 \). The first loop can now be executed in parallel since there is no longer any scalar dependence. The second loop cannot run in parallel but we moved a lot of the computational load into the first loop. The drawback with this approach is the extra storage needed for the new array \( xy_temp \). (On some machines it may be more efficient to use \( xy[ik2+20][i2+20] * xy[ik2+20][i2+20] \) rather than \( xy[ik2+20][i2+20]^2 \), since a multiplication is generally faster than raising a number to a power.)

For another example of a flow dependency, see Figure 3.21. The scalar variable \( buf1 \) is causing a similar problem in using and writing its value when iterations are executed independently, as in the example of Figure 3.19. To be able to parallelize this loop we need to promote \( buf1 \) from a scalar to an array as follows.

In Figure 3.22, \( buf1 \) has been changed to an array \( buf1[i][j] \). Each iteration of the loop has its own copy of \( buf1 \) preventing any reading/writing conflicts. The drawback is the need for extra storage for the additional array.
3.6 Parallel programming languages

Figure 3.21: Automatic parallelization - fragment code 8

\[
\begin{align*}
\text{FOR} \ (j = 1; j < nrow; j++) \\
\text{FOR} \ (i = 1; i < nrow; i++) \\
\quad \text{buf1} &= \text{dble}((-1.0)**(i+j)); \\
\quad \text{ha}[i][j] &= \text{ha}[i][j]*\text{dcmplx}(\text{buf1,dble}(0.0)); \\
\quad \text{hb}[i][j] &= \text{hb}[i][j]*\text{dcmplx}(\text{buf1,dble}(0.0));
\end{align*}
\]

Figure 3.22: Automatic parallelization - fragment code 9

\[
\begin{align*}
\text{FOR} \ (j = 1; j < nrow; j++) \\
\text{FOR} \ (i = 1; i < nrow; i++) \\
\quad \text{buf1}[i][j] &= \text{dble}((-1.0)**(i+j)); \\
\quad \text{ha}[i][j] &= \text{ha}[i][j]*\text{dcmplx}(\text{buf1}(i,j),\text{dble}(0.0)); \\
\quad \text{hb}[i][j] &= \text{hb}[i][j]*\text{dcmplx}(\text{buf1}(i,j),\text{dble}(0.0));
\end{align*}
\]

Figure 3.23 shows another example of a flow dependency preventing parallelization. It is not possible to execute the iterations out of order, each iteration depends on the result of the previous iteration. There is no simple solution to parallelize this piece of code.

\[
\begin{align*}
\text{FOR} \ (i = 2; i < n; i++) \\
\quad \text{a}[i] &= \text{a}[i-1] \ast \text{c}[i];
\end{align*}
\]

Figure 3.23: Automatic parallelization - fragment code 11

Although automatic parallelization gives good results, it cannot always be used and, furthermore, it cannot be used for coarse grained parallelism, the one we are most interested in for image processing applications. Nevertheless, automatic parallelization offers, when possible, the easiest way for parallelizing an image processing application. Because of the small number of successful demonstrations, it is now widely believed that fully automatic techniques cannot solve all parallel programming problems. In such cases, the programmer must be involved in the parallel design, but the programmer’s effort should be minimized as much as possible.

3.6.2 Explicit parallelization

Some parallel programming languages provide explicit parallelism by using specific constructs for specifying concurrent execution, for managing locality, and for communication.

In this section, we briefly describe OpenMP [63] and Compositional C++ (CC++) [7, 115], a small set of extensions to C++ for parallel programming. OpenMP is an emerging standard for parallel programming on shared memory multiprocessors. The reason
of considering CC++ is that this parallel programming language includes extended constructs for allowing the use of explicit task parallelism within an application.

**OpenMP**

OpenMP [63] defines a set of program directives and a library for run-time support that augment standard C/C++ and Fortran 77/90. In contrast to MPI, OpenMP facilitates an incremental approach to the parallelization of sequential programs. In other words, the programmer can add a parallelization directive to one loop or subroutine of the program at a time.

The OpenMP API [27, 26] defines a set of program directives that enable the user to annotate a sequential program to indicate how it should be executed in parallel. OpenMP is based on a fork-join model of parallel execution. The sequential code sections are executed by a single thread, called the *master thread*. The parallel code sections are executed by all threads, including the master thread. OpenMp provides three kinds of directives: parallelism/work sharing, data environment, and synchronization.

The fundamental directive for expressing parallelism is the *parallel* directive. It defines a *parallel region* of the program, that is executed by multiple threads. All of the threads perform the same computation, unless a *work sharing* directive is specified within the parallel region. Work sharing directives, such as *for*, divide the computation among the threads. For example, the *for* directive specifies that the iterations of the associated loop should be divided among the threads so that each iteration is performed by a single thread. Shorthand forms as *parallel for* represent a parallel region that contains a single *for* directive.

The data environment directives control the sharing of program variables that are defined outside of a parallel region. The data environment directives include: *shared*, *private*, *reduction*. Variables defined as *shared* are shared among all the threads in a parallel region. A *private* variable has a separate copy per thread. Its value is undefined when entering or exiting a parallel region. The *reduction* directive identifies reduction variables.

The synchronization directives include *barrier* and *critical*. A *barrier* directive causes the thread to wait until all of the other threads in the parallel region have reached this point. A *critical* directive restricts access to the enclosed code to only one thread at a time. When a thread enters a critical section, it is guaranteed to see all modifications made by all the threads that entered the critical section earlier.

**CC++**

CC++ is a parallel programming language comprising all of C++ plus six new keywords. It is a strict superset of the C++ language, so any valid C or C++ program that does not use a CC++ keyword, can be considered also a valid CC++ program. CC++ contains the following six extensions:
1. The *processor object* is a mechanism for controlling the locality. A computation may comprise of one or more processor objects. Within a processor object, sequential C++ code can be executed without modification. The keyword *global* identifies a processor object class and the predefined class *proc.t* controls processor object placement.

2. The *global pointer*, identified by the type modifier *global*, is a mechanism for linking together processor objects. A global pointer must be used to access a data structure or to perform a computation (using a remote procedure call - RPC) in another processor object.

3. The *thread* is a mechanism for specifying concurrent execution. Threads are created independently from processor objects, and more than one thread can be executed in a processor object. The *par*, *parfor*, and *spawn* statements create threads.

4. The *sync variable*, specified by the keyword *sync*, is used to synchronize thread execution.

5. The *atomic function*, specified by the keyword *atomic*, is a mechanism used to control the interleaving of threads, executed in the same processor object.

6. *Transfer functions*, with predefined type *CVoid*, allows arbitrary data structures to be transferred between processor objects as arguments to remote procedure calls.

A C++ program, like a C++ program, runs initially as a single thread of control (task). However, a C++ program can use *par*, *parfor*, and *spawn* to create additional threads.

The C++ parallel statements introduce explicit task parallelism [7, 115] and they do not require too much understanding or knowledge of parallel programming from a programmer of image processing applications. It is the most basic mechanism for creating parallel threads of control in C++. A parallel block looks just like a compound statement in C or C++ and is preceded by the keyword *par*. Except for a few cases, the statements inside a parallel block can be any legal C, C++, or C++ statement. A parallel block differs from a normal C block in that the order in which the statements from a block are executed is not defined: an execution of a parallel block is an interleaved or possibly concurrent execution of the statements within the block.

The execution of a parallel block is finished after all the statements within the block have finished execution. Consequently, statements after a parallel block will not start to execute until all the statements within the parallel block are terminated.

### 3.6.3 Annotation based parallelism

Other data parallel programming languages provide specific directives for specifying data distribution across processors. These directives enable the programmer to specify
CHAPTER 3. SOME PARALLEL ARCHITECTURES AND PROGRAMMING MODELS FOR IMAGE PROCESSING

```
par {
  statement_1;
  statement_2;
  ...
  statement_N
}
```

Figure 3.24: The use of par statement

a layout scheme for the arrays in a program. Researchers have also proposed extensions for the specification of task parallelism [41, 109]. In this section, we will discuss two parallel programming languages based on annotations, namely High Performance Fortran (HPF) [41] and Spar [109].

High Performance Fortran

High Performance Fortran is the result of a research effort to create a standard for data parallelism in Fortran [54]. HPF provide a rich set of directives for specifying the distribution of data across processors. These `DISTRIBUTE` directives can be used for assigning blocks of data to processors or for assigning elements of an array in a cyclic way. Arrays may be distributed along all of any of their dimensions. An `ALIGN` directive is also provided and allows an array to be aligned relative to another array whose distribution was already specified. Initially, the HPF standard did not provide directives for expressing task parallelism. Meanwhile, researchers have proposed directives for expressing task parallelism in HPF programs.

The `INDEPENDENT` directive specifies that the iteration of the following do-loop may be executed independently and in any order.

Foster et al. [28] have proposed the `PROCESSES` statement as an extension to HPF. The body of the `PROCESSES` statement contains independent calls to a set of data parallel HPF subroutines. The call to each HPF subroutine must be annotated by the number of processors to be used for the routine (allocation). Scheduling of the annotated subroutines must be done explicitly using a `CHANNEL` statement. This statement establishes a virtual communication interface between a pair of routines resulting in a precedence relationship between the routines.

HPF does not adequately address task parallelism. Examples of applications that are not easily expressed using HPF alone include complex applications where different modules represent distinct scientific applications and image processing applications. Such applications must exploit task parallelism for efficient execution on multicomputers. Yet they may incorporate significant data-parallel structures. These issues have motivated the attempt for integration of data and task parallelism. Besides compiler-based and language-based approaches, an alternative approach [31] was proposed, based on specialized coordination libraries designed to be called from data-parallel programs. It uses an HPF binding to MPI. A programmer initiates a computation
request where a certain number of tasks are created. Each task executes a specified HPF program on a specified number of processors. The tasks can call MPI functions to exchange data with other tasks.

Gross, O'Halloran and Subhlok [41] have proposed the notion of parallel sections in a given HPF application. These sections are demarcated using the PARALLEL directive. A parallel section can contain HPF data parallel subroutine calls using the IN and OUT directives which provide the compiler information about the side effects of the subroutines calls and are helpful in deciding on precedence constraints between subroutine calls. Allocation of processors to HPF data parallel routines can be explicitly done using the PROCESSOR and ORIGIN directives. Scheduling is based on the found precedence constraints between subroutine calls and is done implicitly by the compiler.

Spar

Spar [109, 108] is a programming language for high-performance computing, including parallel programming. Since high performance computing often means computations on arrays, there is special support in the language for arrays, including a "toolkit" to build support for more specialized arrays. Due to its support for arrays, the Spar language becomes appropriate for image processing tasks.

The Spar programming language is designed as a modern programming language for parallel computing systems. It provides all the conveniences of modern programming languages, plus good support for parallel programming, while remaining sufficiently close to the implementation level to allow compilation to efficient code. Spar is a superset of Java. For a detailed overview of the differences, see [108].

The most user-friendly form of parallelization is not to provide any special parallelization constructs at all, but to leave parallelization work entirely to the compiler. From the previous section concerning automatic parallelization issues, we must accept that writing a program for a parallel computer requires the programmer to express parallelism more or less explicitly, unless we provide an environment for a smooth parallelization of image processing applications, as presented in Chapters 4 and 5. The other extreme is when the user has to be involved in the lowest details of parallelization, like with MPI or PVM message passing libraries. However, that method is difficult and error-prone. These are the extremes and Spar [109, 108] provides parallelization constructs that are in between: the user must provide some additional information to the compiler, but the generation of parallel code is explicitly left to the compiler.

Chapter 6 presents a detailed overview and ways to embed data and task parallelism in image processing applications implemented using the Spar language.

3.6.4 Discussion

Writing parallel programs would ideally be supported by tools such as parallelizing compilers. However, due to the small number of successful demonstrations, it is at this
time widely believed that fully automatic techniques cannot always solve the parallel programming problems. Currently, only very fine grain parallelism, mostly at loop level, can be exploited. As we focus on image processing applications, explicit parallel languages would be a second best option, as we assume that the programmer has a good knowledge on the inherent parallelism in his application. A negative point for explicit parallel languages is, however, that the current installed sequential software must either be re-programmed or treated in a different way. Ideally, a software program should run without re-programming, efficiently on both any sequential machine as well on any parallel machine. As such, a sequential language augmented with pragmas - parallelization annotations that suggest the (pre-) compiler to insert parallel constructions -, are a better choice. In Section 3.7 experiments are reported that have been performed to reveal the programming and processing efficiency of several parallelization tools.

3.7 Experiments

This section reports the outcome of several experiments that were performed to obtain more insight in the questions that are raised in the previous sections.

- Message passing experiments on a distributed memory system:
  - In a first experiment, we tried to verify our assumption that data parallel processing of low-level operations is beneficial on distributed memory systems with the current speed of interconnection networks. We applied MPI for a data parallel implementation of a simple problem: a geometric mean filter.
  - Assuming that the difference in performance between PVM and MPI would be small, we tried out both PVM and MPI on a more complex algorithm, the 2D and 3D watershed segmentation of a stack of CT-images of a human skull. The outcome of this experiment is to give us insight in: the differences in ease of programming in PVM and MPI, how beneficial a data parallel approach is if the image chunks cannot be evenly spread over the processors, and finally, the influence of the network hardware and device driver software. For the latter we used PVM over Fast Ethernet and MPI using Myrinet.

- Explicit parallel languages versus automatic parallelization on a shared memory system.
  - In this experiment we first tried to verify in which case parallel processing of low-level operations is beneficial on shared memory systems, and secondly we tried to find out the differences in ease of programming and in parallelization. We used the explicit parallel language (CC++) and a fully automatic parallelizing compiler, the SUN SPARCompiler 4.0. Both use (internally) the same SUN multi-threading library.
3.7 Experiments

- Experiments with SIMD and MMX.
  
  - In this experiment we first tried to verify the differences in ease of pro-
    gramming, and secondly we tried to obtain figures on the speed-up that
    can be achieved using an SIMD co-processor board or an SIMD instruction
    set. The reason behind this is, that for applications with severe real-time
    constraints, parallel processing on a cluster of workstations might not offer
    enough speed to meet real-time and/or latency constraints. Moreover, other
    requirements of the application, such as compactness, low-power, embed-
    ding the system, and cost/performance might indicate that a non-standard
    solution must be chosen, thus sacrificing the maintainability and sustain-
    ability.

3.7.1 Message passing experiments

We shall now present two sample applications illustrating the way parallel image pro-
cessing applications can be implemented using message passing libraries. In the follow-
ing, we present a framework to add data parallelism to a sequential image processing
library.

For effective processing of digital images it is essential to compute the data using a
variety of techniques such as filtering, enhancement, or segmentation. Thus, there is a
great need for a collection of image processing routines which can easily and effectively
be applied to a variety of data. In our research we have used a simple version of the
image processing library DIPLIB (Delft Image Processing LIBrary) [76]. It provides
a basic set of image handling and processing routines and a framework for expanding
a set of image processing routines.

While the library provides the necessary functionality and flexibility required for solv-
ing various image processing applications, it is clear that for real-time processing,
many important image processing tasks are too slow. For example, a median filter
operation which removes noise from a 1024 × 1024 pixel image requires several min-
utes to complete on a common desktop workstation. This is unreasonable in real-time
image processing. A method to speedup the execution is to use an existing cluster of
workstations for parallelization [8, 101, 81]. We have developed a parallel extension
to the library, on top of MPI [99] as a message passing layer. As a consequence, the
code remains portable to a large number of parallel computer platforms.

In the following, we present some experimental results obtained using some of the par-
allel extensions of the image processing library described above. We have parallelized
two image processing algorithms on a distributed memory system, i.e. the geometric
mean filter and the watershed algorithm. The first algorithm belongs to the class of
low-level image processing operators, while the second algorithm belongs to the class
of intermediate-level image processing. The geometric mean filter is a spatial mean
filter (local neighborhood operator) which is quite computationally intensive, so it is a
good candidate for data parallelization. The watershed transform is an intermediate-
level image processing algorithm used to segment an image. The watershed transform
can be classified as a region-based segmentation approach.
Geometric Mean Filter

Spatial filters can be effectively used to remove various types of noise in digital images. These spatial filters typically operate on small neighborhoods, like $3 \times 3$ to $15 \times 15$, and some can be implemented as convolution masks. The two main categories of spatial filters for noise removal are rank order filters and mean filters. The rank order filters are implemented by sorting the neighborhood pixels, ranging from the smallest to the largest gray-level value and then select a rank value. A mean filter determines the average value. Mean filters work best for gaussian or uniform noise, and the rank order filters work best with salt-and-pepper, negative exponential, or Rayleigh noise. Mean filters have the disadvantage of blurring the lines, edges and other details: they are essentially lowpass filters. The rank order filters are nonlinear, so the results are sometimes unpredictable. In general, there is a tradeoff between preservation of image detail and noise elimination.

To reduce the development time for new image processing operators, the DIPLIB library supplies several framework functions. One of the available frameworks is responsible for processing different types of image filters (neighborhood image processing operators). This framework is intended for image processing operations that filter the image with an arbitrary filter shape. A description of the framework is presented in Figure 3.25. Each neighborhood operator calls a framework function which sequentially computes each line of the image. We parallelize the framework approach by data decomposition on a distributed-memory system and in this way we obtain parallelization on all image processing operators (i.e. filters) that are using the framework. So, given a neighborhood image processing operator, the operator calls the framework function on the master processor. The image is distributed by the master processor row-stripe or column-stripe wise across processors, each processor is computing its part of the image and then the master processor gathers the image back, see Figure 3.26.

![Figure 3.25: Library framework function](image)

The geometric mean filter is used to remove Gaussian noise in an image [38, 104]. It is defined as the product of the pixel values within a window, raised to the $1/k^2$ power.
The definition of the geometric mean filter is given in formula 3.7:

\[ GeometricMean = \prod_{(r,c) \in W} [I(r,c)]^{1/k} \]  (3.7)

where \((r,c)\) are the image pixel coordinates in window \(W\), \(I(r,c)\) is the pixel value (grey level) and \(k \times k\) is the size of the window.

The geometric mean filter is an example of a separable filter, i.e., the computation can be performed by two consecutive \(k \times 1\) filter operations on the whole \(n \times n\) image, one in the horizontal direction and the other in the vertical direction. To perform each filter operation on \(P\) processors, we have to scatter the image using a row-stripe distribution, perform on each processor a \(k \times 1\) filter operation, gather the image back, scatter the image again using a column-stripe distribution, apply on each processor a \(k \times 1\) filter and then gather the image back.

The filter is applied to an input image of respectively size \(256 \times 256\) and \(1024 \times 1024\) and the application is executed on a parallel computer with up to 24 processors. The filter is implemented on top of MPI (Message Passing Interface) [99] and runs on a cluster of Pentium Pro/200Mhz PCs with 64MB RAM under Linux [17]. The run times in milliseconds using a \(256 \times 256\) size image are tabulated in Table 3.2. The run times in seconds using a \(1024 \times 1024\) size image are tabulated in Table 3.3.

The relative speedups are plotted in Figures 3.27 and 3.28 for each image and window sizes of \(3 \times 3\), \(9 \times 9\) and \(15 \times 15\). One may observe that a better performance was obtained with larger window sizes. Thus, the lowest speedup was obtained for a \(3 \times 3\) size window, while the highest speedup was obtained for a \(15 \times 15\) size window, for both sizes of image. The reason is that the image operator granularity increases with the window size. As a consequence, communication time is less dominant compared to the computation time and a better performance is obtained.

To conclude, we have presented a way of adding parallelism to a sequential image processing library implemented using frameworks to allow processing of certain types of image operators. The approach is based on parallelizing the frameworks using message passing libraries such as MPI.
Table 3.2: Parallel geometric mean filtering execution time (ms) for a 256 × 256 size image

<table>
<thead>
<tr>
<th>N</th>
<th>3x3</th>
<th>9x9</th>
<th>15x15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>121.89</td>
<td>259.53</td>
<td>412.88</td>
</tr>
<tr>
<td>2</td>
<td>67.72</td>
<td>136.54</td>
<td>213.22</td>
</tr>
<tr>
<td>4</td>
<td>34.47</td>
<td>71.88</td>
<td>110.225</td>
</tr>
<tr>
<td>8</td>
<td>22.46</td>
<td>39.66</td>
<td>58.83</td>
</tr>
<tr>
<td>16</td>
<td>15.07</td>
<td>23.67</td>
<td>33.26</td>
</tr>
<tr>
<td>24</td>
<td>12.59</td>
<td>18.32</td>
<td>24.71</td>
</tr>
</tbody>
</table>

Table 3.3: Parallel geometric mean filtering execution time (sec) for a 1024 × 1024 size image

<table>
<thead>
<tr>
<th>N</th>
<th>3x3</th>
<th>9x9</th>
<th>15x15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.95</td>
<td>4.15</td>
<td>6.60</td>
</tr>
<tr>
<td>2</td>
<td>1.08</td>
<td>2.18</td>
<td>3.40</td>
</tr>
<tr>
<td>4</td>
<td>0.59</td>
<td>1.14</td>
<td>1.75</td>
</tr>
<tr>
<td>8</td>
<td>0.348</td>
<td>0.62</td>
<td>0.96</td>
</tr>
<tr>
<td>16</td>
<td>0.226</td>
<td>0.36</td>
<td>0.51</td>
</tr>
<tr>
<td>24</td>
<td>0.187</td>
<td>0.27</td>
<td>0.38</td>
</tr>
</tbody>
</table>

Watershed algorithms

In this section, we present a parallel implementation of an image segmentation transform - the watershed algorithm. The algorithm is implemented using the message passing paradigm, with both PVM and MPI and is applied to a specific application for the HPISIS project [46, 119]. The project aims at the development of a system that can monitor atrophic and traumatic brain changes in patients having Alzheimer disease, by means of high speed automated analysis of the brain/liquid ratio from cranial CT-scans. Our approach is to first segment a CT-scan (a 3D image) and then calculate the brain and liquid volume. A CT-scan of the human brain consists of a series (typically 20-30) of 2D images. The 2D images are consecutive slices of the brain, so the entire set can be seen as a 3D image. Because the sampling frequency in the z-direction (vertical when the patient is standing) is very poor, we used an interpolation method to obtain cubic voxels. To determine the brain/liquid ratio we must calculate the volume of each liquid enclosure in the brain. From each 2D slice the liquid/brain ratio can be calculated and accumulated to a total amount. In areas around the nose, ears, eyes and possibly ventricles the 2D segmentation results suffer from lack of context information in adjacent slices. Segmentation into 3D volumes is sometimes difficult if the context information related to the place of nose, ears and eyes is neglected. What seems to be an isolated cavity could in reality be a tunnel to a different object with a different label. Later in the HPISIS project, 3D segmentation algorithms were also investigated, to find improvements in cases were the 2D
Figure 3.27: Speedup of geometric mean filtering on a $256 \times 256$ size image for different window sizes

Figure 3.28: Speedup of geometric mean filtering on a $1024 \times 1024$ size image for different window sizes

segmentation was not optimal, such as in the neighborhood of nose and ears. This investigation did not prove to find much more accurate or faster results than the 2D
method, see [78].

Our watershed algorithm is applied to 2D images from cranial CT-scans to extract the liquid parts from them. The evaluation on a distributed memory system is included.

One of the morphological algorithms used to segment an image is the watershed filter. Image data may be interpreted as a topographic surface where the image gray-level represents altitude. In a gradient magnitude image, region edges correspond to the high watersheds and low-gradient region interiors correspond to catchment basins. Catchment basins of the topographic surface are homogeneous in the sense that all pixels belonging to the same catchment basin are connected with the basin’s region of minimum altitude (gray-level) by a simple path of connected pixels that have monotonically decreasing altitude (gray-level) along the path. Raw watershed segmentation produces a severely oversegmented image with hundreds or thousands of catchment basins. To overcome this problem, region markers and other approaches have been suggested to generate good segmentation.

First we implemented a sequential 2D and 3D version of the watershed algorithm to obtain a ground base to our speed-up figures. Determination of the brain/liquid ratio considering a CT scan of 20 slices as a set of 2D images, takes about 5 minutes on a Sparc20 and about 15 minutes if processed as a 3D image. To investigate speed-up by parallelization we have implemented and tested a parallel 2D watershed algorithm.

There are several methods to segment an image using a sequential watershed algorithm [112, 96]. In the algorithm of Vincent and Soille [112] all the pixels from the image are sorted in the order of increasing greylevel. The pixels with the lowest greylevel are taken as seeds for the basins and from that basins regions which include the pixels with the next greyvalue are grown in a breath-first manner. If the pixels from a propagation front (basin) do not touch pixels from another propagation front, they are assigned the same label. Otherwise, the pixels from the propagation front are labeled as watershed pixels. All the pixels from a greylevel that cannot be reached in this way are new local minima and become seeds for new basins and the process is repeated till all the pixels from the image have been assigned a label.

Parallel versions of the watershed algorithm have been proposed in [67, 68, 65, 66]. In the algorithm of Moga, Cramariuc, and Gabbouj[68] every node starts to compute the watershed of the sub-image assigned to it. A lot of communication is involved between nodes that process neighboring parts of the image. The algorithm of Meijster and Roerdink[66] consists of transforming the image in a directed valued graph \( f^* = (\mathcal{V}, \mathcal{E}) \). The vertices of the graph are maximal connected sets of pixels with the same greyvalue. If \( v, w \in \mathcal{V} \) then \((v, w) \in \mathcal{E}\) if \( v \) contains a pixel \( p \) that touches a pixel \( q \) in \( w \) and \( f(p) < f(q) \), where \( f(p) \) represents the greyvalue of pixel \( p \). Then, the watershed of the graph is calculated in a way similar to the sequential algorithm[112]. Finally, the graph is transformed back to an image. Since all the pixels having the same greyvalue belong to one node of the graph, the watershed calculation of the graph can be parallelized.

The watershed algorithm is a region growing algorithm. There are several methods to parallelize the region growing algorithms on a distributed memory system [65]. The methods can be classified as:
3.7 Experiments

- Sub-image parallel: The image is divided into sub-images which are distributed over the processors. Each processor processes the sub-image assigned to it. The processors have to exchange some data.

- Region parallel: The complete image is distributed to all processors. Each processor starts to grow regions from the seeds that are assigned to it. Since information assigned to one processor might be important to another, some communication between processors could be expected.

- Pixel parallel: All processors are assigned a disjunct number of pixels. The mapping of pixels to processors can vary from one algorithm to another. A lot of communication can be expected from this approach.

- Region and pixel parallel: In this method the region and pixel parallel methods are combined. A subset of processors is assigned to grow a certain region in a pixel-parallel manner.

- Distributed bucket processing (DBP) [73]: The sequential algorithm starts with a number of seeds, which are the starting points for growing regions. For growing the regions one can use wavefronts. During each step a wavefront of neighbouring candidate pixels is examined and evaluated to determine whether candidate pixels satisfy the region parameters. If yes, the candidate pixels are added to the region and the region parameters are updated. The propagation of the wavefronts over the image may be implemented by repetitive scanning of the image. Usually, the number of candidate pixels is far less than the number of pixels in the image. An approach for reducing the set of candidate pixels is to collect them in a bucket and then process them, possibly updating the region parameters. Two buckets are necessary. One bucket, called the write bucket is used to collect the candidate pixels. The other bucket, called the read bucket is initialized with the seeds. The read bucket is processed and each candidate pixel is examined. If the pixel is not already assigned to a region and it meets the requirements then the pixel is assigned to a region and a set of possible new candidate pixels is generated and added in the write bucket. When the read bucket is empty, the region parameters are updated and the role played by the read and write buckets is reversed. The parallel implementation of the above algorithm consists of mapping the image, for example row-stripe wise, on a number of processors. The buckets might be mapped on the same set of processors or on a different set. All pixels belonging to a certain region will end up in a bucket corresponding to the same processor and may be used in further processing.

The key problem in the parallelization of the watershed algorithm is how to distribute the image over processors to achieve low communication time and high speed. We use a variant of the algorithm of Vincent and Soille [112]. We implemented a sequential program and a region parallel program. The entire image is distributed (in a multicast) over the $n$ processors. In this way each processor has all the data needed to compute its own result (i.e. a part of the output image), so the communication time is minimized. Each node starts computing the watershed in a certain region assigned to it and then the computed image is sent back to the master processor which combines the results.
Because the watershed algorithm is a global operation it may be clear that to compute a certain part of the output image a node needs more than the information inside the corresponding part of the input image assigned to that node.

The main problem is how to find the overlap between the regions of the input image assigned to different processors in order to compute correctly the watershed of the image. An example is presented in Figure 3.29. If the two local minima from the lower part of the image are not partially included in the overlap with the upper part of the image, the dotted watershed will not be found and the watershed of the image will not be computed correctly. So, the overlap around a part of the input image assigned to a processor has to be large enough to contain all the seeds of the regions of image assigned for computing to neighboring processors. All the pixels that can be reached from the edge of that part of the image in a non-descending way are part of the overlap. We use this observation in our algorithm.

Figure 3.29: Determination of overlapping regions.

The behavior of the algorithm is studied on images of sizes 128x128, 256x256, 512x512 and 1024x1024. For MPI we also use 2 more images of sizes 257x257 and 384x384. An example of a test image is shown in Figure 3.30. As a preprocessing step a gradient magnitude filter (implemented using Gaussian derivatives) with \( \sigma = 1.5 \) is first applied to the image. The result of applying the watershed algorithm to the pre-processed image is shown in Figure 3.31. The severe oversegmentation is clearly visible. To reduce the oversegmentation we use the histogram of the CT-image. This has 3 large peaks. From light to dark these peaks come from the background, the brain and the skull. To extract the skull we use a value derived from the histogram. Since CT-systems are calibrated, the value of the liquid has a predefined value that is used to extract a liquid region.

To extract the liquid parts, the basins with a deepest greyvalue between 0 and 40 are joined. The result is presented in Figure 3.32. The image from Figure 3.33 shows the result of extracting the skull. This is done by joining basins with a lowest value between 170 and 255. It is quite easy to obtain the liquid areas inside the brain by combining the skull and liquid segmentation results. Figure 3.34 shows the final segmentation result. To compute the brain and liquid areas we count the number of pixels inside these regions as a measure of the surface. Figure 3.35 shows the skull segmentation in the 3D variant of the algorithm.

The parallel 2D watershed algorithm is implemented using the message passing paradigm on a cluster of PentiumPro/200MHz PCs with 64MB RAM running Linux, and connected through Myrinet [69]. Both PVM and MPI are used to show the speed-ups and efficiency in each case. In Figures 3.36 and 3.38 the speed-up and efficiency of
the parallel implementation of the 2D watershed algorithm using PVM are depicted for different image sizes. In Figures 3.37 and 3.39 the speed-up and efficiency of the parallel implementation using MPI are depicted. The speed-up and efficiency of the algorithm when using PVM are not very good. This is because on the distributed memory system used for tests, PVM was implemented on top of TCP/IP and fast Ethernet. With MPI-Panda on top of Myrinet we obtained better results, largely due to the fact that Myrinet is faster (2Gb/s) and is based on wormhole routing crossbar switches. The large difference between the speed-up for 256x256 image size compared to 257x257 image size is due to the MPI-Panda implementation. When more than
64kB are sent, another protocol is used. The selection of the protocol is done in the internal implementation of MPI-Panda.

The brain/liquid ratio can be determined in a very robust manner with a watershed segmentation and a basin-joining method based on peaks in the histogram. The segmentation results are very accurate, as is shown in Figure 3.34. The speed-up and efficiency of the parallel watershed algorithm are not very good when PVM is used due to the many collisions on Ethernet when the resulting images are gathered. The
speed-up for a CT slice is at most 3 on a 20 nodes machine while the efficiency is 20%. MPI is implemented on top of fast Myrinet and the results are better: a speed-up of 6 on 24 nodes with an efficiency of 30%.

The main conclusion to be drawn from these experiments is that the watershed transform remains a global operation, and therefore in the case of a parallel implementation, a modest speedup is to be expected.
3.7.2 Explicit parallel languages (CC++) versus automatic (SUN)

One of the methods [8, 101, 81, 55] for parallelizing an image processing application that uses a sequential image processing library is to use the help of parallelization tools (compilers). First, we have used the Compositional C++ (CC++) language to introduce task parallelism explicitly in our applications. Secondly, we automatically parallelized an image processing application using our SCIL-Image library (a sequential image processing library, see [105]) with the SPARCompiler SC4.0. We compared this result to the result obtained when using the same compiler without automatic parallelization options and the result of using the gcc compiler, which was the compiler used to design the SCIL-Image libraries. Other image processing applications using the same image library were parallelized introducing explicit parallelism by using the CC++ compiler, where the par statement was the only used parallel construct of CC++. All the experiments were performed on a SUN SPARC multiprocessing architecture - a shared memory system with 2 processors, to show the principles [117, 118].

We used our SCIL-Image library [105] to measure the time of reading/writing a number of input/output images, because one can always reduce the disk I/O time, whenever one needs to process more images. We have also applied some point operators (integer addition, subtraction, multiplication), neighborhood operators (convolution kernels) and global image operators on images to measure the run time.

Figures 3.40, 3.41 and 3.42 show the processing/reading/writing times of the sequential and parallel programs for a number of images. The CC++ language is used for task parallelism, introducing it with the help of the par statement. For a small number of image operations the process of creating parallel threads proved to be time consum-
3.7 Experiments

![Graph showing parallel and sequential processing times](image)

**Figure 3.40:** Parallel and sequential image *processing* times for a sequence of image processing functions (sobel, robinson, prewitt, roberts, fast-Fourier transform, fast-Hartley transform)

...ing and not much improvement can be obtained. As the number of image operations inside the *par* statement increases, the difference between the parallel and sequential times can clearly be seen. Figure 3.40 shows the execution time of a sequence of different image processing functions (sobel, robinson, prewitt, roberts, threshold, fast-Fourier transform and fast-Hartley transform) each of them applied to a different image of the same size. The sequence is applied first in parallel, then sequentially. In Figures 3.41 and 3.42 we measure the reading/writing time of a sequence of images in parallel and sequentially.

Figure 3.43 shows the execution time of the fast-Fourier transform applied to a number of images of the same size. The execution times of sequential and parallel programs are measured and it can be seen that the parallel execution time is about half the sequential time.

A method of parallelizing an image processing application using a sequential image processing library was described. Our approach demonstrates improvements in the execution time of an image processing application compared to a sequential execution for a two processor shared memory architecture to illustrate the principles.

The parallelization was performed both automatically (using the SPARCompiler SC4.0) and user controlled (using the CC++ par statement, see Section 3.5.1) with very few changes at the application source level. Thus, extensive knowledge of parallel programming paradigms are not always necessary to parallelize a sequential image
processing application.

3.7.3 Experiments on SIMD systems

1DC programming on the IMAP-Vision architecture

Writing a program in 1DC is not very difficult, but writing a good program is. A program that works is not necessary efficient. In order to write efficient programs, one should follow certain guidelines. The first is always try to exploit the application data parallelism to the maximum. If one has 256 available PEs, one should use all of them. The second is to avoid using nested loops. These two important recommendations often imply some extra programming effort [123, 107].

A simple but very expressive example is the histogram computation, which is a global image processing operation.

For histogram calculation the first step is trivial. Every processor makes a local histogram of the image column that it has in its own memory, see Figure 3.44. The variable named src contains the image which is distributed column-wise on the PE's. In a separate variable histo we will compute the histogram of the image, the histogram
result for the grey value $g$ being obtained on the PE whose PE number is $g$. The variable $tmp$ is a `sep` variable, a two dimensional array, with the number of rows equal to 256 and the number of columns equal to the number of PEs ($PENO$). The variable $tmp$ stores, for every processor, the local histogram of the image column it has in its memory. $PENO$ is the number of available processors.

The next step is to compute the global histogram $histo$ from the local histogram $tmp$. For computing the global histogram the alternatives are presented in Figures 3.45, 3.46, 3.47, and 3.48. The code in Figure 3.45 has a nested loop in which only 1 processor is working at a time. It calculates, for every grey level value $j$ (from 0 to 255), the sum of the components of each `sep` variable $tmp[j]$ computed in Figure 3.44, thus obtaining the total number of pixels from that grey level value.

A more efficient use of the available processors would be to do a `prefix addition`, a data parallel method described by [44], which is one of the most basic forms of parallel programming. The parallel prefix addition transforms a problem of size $n$ into smaller problems of size $n/2$. By combining the outputs of the smaller problems with the original inputs we get a time complexity of $O(\log n)$. In our case, the prefix addition uses only $\log(256) = 8$ steps, see Figure 3.46. Let us consider the grey level $i = 0$. After the first step of the inner loop $j$ we obtain in variable $tmp[0] : [0 :]$ the sum of its first two values (from PE 0 and 1). After the second step, $tmp[0] : [0 :]$ contains the sum of its values from the first 4 processors, and after the last step $tmp[0] : [0 :]$ contains the sum of all its values (assuming that we have 256 PE's), so the histogram value for grey level 0.
Since the local histogram values of each grey level are on each line of variable tmp computed in Figure 3.44, we can rotate the image over 90 degrees to get them on one column, and thus on one processor, see the code presented in Figure 3.47.

The code can be further improved. Every processor could add its local value and then pass it on to its neighbor, which will add its local value while the first one adds the next value, see the code presented in Figure 3.48. Note that $PENUM$ is a pre-defined $sep$ constant equal to $(1, 2, \ldots, PENO :)$.

The performance figures, obtained for the four versions of the code are given in Table 3.4. Compared to the sequential approach from Figure 3.45 the final implementa-
3.7 Experiments

Global histogram - sequential solution
BEGIN
\[ \text{FOR } (j = 0; j < 256; j ++) \]
\[ \text{FOR } (i = 0; i < \text{PENO}; i ++) \]
\[ \text{histo}[j] += \text{tmp}[j] : [i] \]
END

Figure 3.45: A first implementation

Global histogram - with prefix addition (
BEGIN
\[ \text{FOR } (i = 0; i < 256; i ++) \{ \]
\[ \text{FOR } (j = 0; j < 8; j ++) \]
\[ \text{tmp}[i] += \text{tmp}[i] : < (1 << j); \]
\[ \text{histo}[i] = \text{tmp}[i] : [i]; \]
\}
END

Figure 3.46: A second implementation

...tion is 100 times faster.

Table 3.4: Histogram algorithm performance

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential histogram</td>
<td>31 ms</td>
</tr>
<tr>
<td>With prefix addition</td>
<td>2.77 ms</td>
</tr>
<tr>
<td>With rotation</td>
<td>0.823 ms</td>
</tr>
<tr>
<td>Fastest algorithm</td>
<td>0.31 ms</td>
</tr>
</tbody>
</table>

A special class of low level image processing operators are the recursive neighborhood operators (RNO) [52]. Examples of RNOs are the morphological operators thinning and distance transform [5]. For updating each pixel, RNOs refer to the pixel value of neighborhood pixels that have already been updated. A parallel method for efficient implementation of RNOs is described in [51]. This method uses overlapping waves, where each wave is a line of activated PEs. PEs are activated successively in a fixed direction, while each activated PE updates the corresponding pixel after every fixed time interval. Using this method, a distance transform implementation in 1DC has a performance of 8ms on a 256 × 240 size image when using 256 PE.

The distributed bucket processing [74, 75], described also in this chapter, section 3.5.1, is a stack based method appropriate for implementing RNOs. It consists of two processing phases during which every PE of the LPA simulates a software stack in its local memory. In the first phase, all pixels are visited once in order to find points
Global histogram - with rotation ()
BEGIN
   rot90(tmp,0,256);
   FOR (i = 0; i < 256; i++)
      histo += tmp[i];
END

Figure 3.47: A third implementation

Global Histogram - final version ()
BEGIN
   FOR (i = 0; i < 256; i++)
      histo =:< (histo + tmp[(PENUM + i)&255]);
END

Figure 3.48: The fastest implementation

with specific features such as contour pixels or peak pixels and push them onto the stack. In the second phase, the pixels are popped and processed. In the processing phase, the pixels can be pushed back on the stack if they correspond for instance to an object contour (contour tracking operation) or they can be used to accumulate votes like in the Hough transform. The pixel processing continues until all the stacks are empty. A circular Hough transform implemented using this method takes 6 to 11ms on a 256 × 240 size image. This image transform is used to detect in real time the ball in a robot soccer game using a vision system [74].

MMX programming

In Figure 3.49 we show an approach of integrating the MMX features in a data parallel framework to parallelize a sequential image processing library. A master processor is distributing the image in a row-stripe way to the slave processors and each slave processor is computing its part of the image by applying a point operator to each line of that part of the image. If the slave processor is enabled with MMX technology we exploit the MMX features of processing more elements of a line in parallel. This part has to be coded using MMX instructions.

The best way to design an efficient MMX code is to write blocks in assembly language that later can be included into the C code. When writing the code using MMX instructions one should follow general guidelines considering the processor architecture, pipelining, and dynamic execution features. Some important issues that have to be considered are:

- The rules of pairing the instructions in U and V processor pipelines.
- Optimal memory and register access sequencing.
3.7 Experiments

Figure 3.49: Adding MMX features to an image processing library

- Memory caching and read/write operations.
- Branch prediction and dynamic execution.
- Alignment of data and stack.

The last item is very important for achieving high performance and we will discuss it in more detail.

When data access to a cacheable address misses the data cache, the entire line is brought into the cache from external memory. This is called a line fill. On Pentium and dynamic execution (P6-family) processors, data arrives in a burst composed of four 8-byte sections to match the cache line size of 32 bytes. On the P6 when a write occurs and the write misses the cache, the entire 32-byte cache line is fetched. On the Pentium processor, when the same write miss occurs, the write is simply sent out to memory. Therefore, when aligning data on a 32-byte boundary in memory, one may take advantage of matching the cache line size and avoiding multiple memory-cache transfers. The delay for a cache miss on the Pentium with MMX is 8 internal clock cycles and on P6 the minimum delay is 10 internal clock cycles. For cache issues specifically related to image processing algorithms see [90].

In order to align data on a 32-byte boundary, padding of data may be required. Many compilers allow the user to specify the alignment. However, one may use the following C code to guarantee the alignment, see Figure 3.50.

As a matter of convention, compilers allocate anything that is not static on the stack and it may be convenient to make use of 64-bit data quantities that are stored on the stack. Another reason for aligning the stack may be the following, if inside the MMX routine an argument passed to the routine is accessed several times, it is better to aligned with a boundary to avoid cache misses. Figure 3.51 shows code in the program prologue and epilogue that will make sure that the stack before calling the MMX routine is aligned.

In Figure 3.52 we show the code of adding two images with saturation, using MMX instructions.

An interesting problem is how to measure the execution time of the MMX routines. The information provided by RDTSC (Real Time Stamp Counter) may be used [12]. The measured timing is approximate and depends on many factors such as OS overheads, number of processes running, cache situation if MMX code contains read/write
Data_Alignment ()
BEGIN
  unsigned char *pad, *data; // — padding the data —
  pad = (unsigned char*) malloc(SIZE+31); // the pad pointer
data = (unsigned char*) (((unsigned int) pad) + 31) & (31));
  // the 32-byte aligned pointer
  ...
  the program ...
  ...
free(pad);
END

Figure 3.50: Padding of data

instructions etc. Various other conditions, such as cache warming prior to reading or writing from/to the same memory blocks, and a particular write strategy implemented in the processor and L2 cache can affect the performance. For that reasons we need to carefully consider the results and run multiple tests.

MMX technology can indeed be used to improve the performance of common digital image processing operations. If parallel processing of multiple data is applicable, on the average, the MMX application exhibits two to four times better performance than its ordinary counterpart.

Experiments with different hardware and operating systems showed that execution time is dependent on many factors, of which memory transfer characteristics being the most important. The exact time cannot be provided until the user performs his own test under his own conditions.

By large, considering various factors, the developer will benefit from using MMX technology in speeding up compute intensive loops and exploiting the parallelism in processing multiple data. The user benefits by avoiding additional expenses for specialized image processing hardware, because MMX software can provide comparable or even better performance compared to parallel systems, but currently at the cost of programming in assembler.

MMX versus IMAP

We have made a study of the impact of using SIMD techniques and architectures in low level image processing. Speedups obtained on the IMAP-Vision board and a single Intel MMX processor computer have been determined for different low-level image processing operators.

In Table 3.5 we present a comparison of the execution times between MMX code on a single Pentium II 200MHZ processor and 1DC on a 40 MHz IMAP-Vision system. We have used $256 \times 256$, 8 bits per pixel images in our measurements. The Intel Pentium II (200MHz) and the IMAP-Vision (40 MHz) where comparable technologies at the time we performed the experiment.
Stack Alignment ()
BEGIN
  asm volatile

  ( — stack alignment —

    mov %%%esp, %%%ebx // load ESP into EBX

    sub $4, %%%ebx // reserve space on stack for the old value of ESP

    and $-32, %%%ebx // align EBX along a 32 byte boundary

    mov %%%esp, (%%%ebx) // save old value of ESP in stack, behind the boundary

    mov %%%ebx, %%%esp // align ESP along a 32 byte boundary

    ::

  );

  ... the program with calls to MMX routines ...

  asm volatile

  ( // — restoring old stack —

    mov (%%%esp), %%%ebx // load the old value of ESP

    mov %%%ebx, %%%esp // restore the old value of ESP

    ::

  );
END

Figure 3.51: Stack alignment

Low level image processing performs very well on a single MMX processor architecture. The IMAP-Vision still performs, on average, about 10 times better, mostly because it has 256 PE’s. The drawback of the IMAP-Vision system is the 40Mhz operating frequency. Both systems share another major drawback. The IMAP-Vision system has no floating point operations, and within MMX code, we cannot use floating point (FP) instructions either. In fact we can, but the cost of switching between the FP mode and the MMX mode is 50 processor cycles.

While the IMAP-Vision system performs better because of the large number of processing elements, the price of the MMX processors make them good candidates for low level image processing.
CHAPTER 3. SOME PARALLEL ARCHITECTURES AND
PROGRAMMING MODELS FOR IMAGE PROCESSING

Image_add ()
BEGIN
    int ImgAdd(unsigned char *Src1, unsigned char *Src2, unsigned char *Dest, int l)
    {
        if (l < 8) return 0; // image size must be at least 8 bytes

        asm volatile
        {
            mov %2, %eax // load Src1 address into EAX
            mov %1, %ebx // load Src2 address into EBX
            mov %0, %edi // load Dest address into EDI
            mov %3, %ecx // load loop counter (SIZE) into ECX
            shr $3, %ecx // counter/8 (MMX loads 8 bytes at a time)
            .align 16 // 16 byte alignment of the loop entry
            .L1010:
            movq (%eax), %mm1 // load 8 bytes from Src1 into MM1
            paddusb (%ebx), %mm1 // MM1=Src1+Src2 (add 8 bytes with saturation)
            movq %mm1, (%edi) // store the result in Dest
            add $8, %eax // increase Src1, Src2 and Dest
            add $8, %ebx // register pointers by 8
            add $8, %edi
            dec %ecx // decrease the value of the loop counter
            jnz .L1010 // check loop termination, proceed if necessary
            emms // exit MMX state
        };
    
    return 1;
}
END

Figure 3.52: Image addition with saturation

Table 3.5: MMX versus IMAP-Vision timings

<table>
<thead>
<tr>
<th>Operator</th>
<th>MMX PII 300 Mhz</th>
<th>IMAP-Vision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image binarization</td>
<td>0.3ms</td>
<td>0.036 ms</td>
</tr>
<tr>
<td>Image addition</td>
<td>0.5ms</td>
<td>0.1 ms</td>
</tr>
<tr>
<td>Image mean</td>
<td>0.9ms</td>
<td>0.04ms</td>
</tr>
<tr>
<td>Image multiplication</td>
<td>1.1ms</td>
<td>0.09ms</td>
</tr>
<tr>
<td>Images bit and</td>
<td>0.8ms</td>
<td>0.07ms</td>
</tr>
<tr>
<td>Convolution 3 x 3 kernel</td>
<td>5.5ms</td>
<td>0.42ms</td>
</tr>
<tr>
<td>Sobel edge detection</td>
<td>2.4ms</td>
<td>0.4ms</td>
</tr>
<tr>
<td>Image Variance</td>
<td>14.8ms</td>
<td>0.65ms</td>
</tr>
<tr>
<td>Histogram</td>
<td>10.6ms</td>
<td>0.31ms</td>
</tr>
<tr>
<td>Dilation</td>
<td>12.4ms</td>
<td>0.31ms</td>
</tr>
</tbody>
</table>
A clear negative point is that programming MMX code is cumbersome, in contrast with the 1DC programming of the IMAP vision system.

### 3.7.4 Discussion

From the message passing experiments we observed that:

- Data parallel processing of low-level operations on distributed memory systems is, with the current network speeds, highly beneficial.

- CRL is easier to program, and therefore to be preferred over MPI. However, as an embedded library for parallelization tools on a higher abstraction level MPI is to be preferred, because it is maintained, widely used, has become an industry standard, and hence will probably exist longer.

From the explicit parallel language versus automatic parallelization on a shared memory system we observed that:

- Data parallel processing on a shared memory processor is not very beneficial, as the processors must wait for each other, even when accessing their own image chunk.

- Global, regular, fine grain image processing problems, such as FFT operations perform well on shared memory systems.

- In our experiments we have found not much difference in the performance of the automatic parallelizing compiler and the explicit parallel compiler CC++ (both based on the same thread-library).

From the experiments with SIMD and MMX we learned that:

- The use of MMX like instructions is hardly supported by compilers and source level debuggers. This is in contrast with a system such as the IMAP-Vision and the IMAP-CE which offer excellent software tools.

- The speedup of using the special multimedia instructions is limited, compared to a system such as the IMAP-Vision and the IMAP-CE which is a more up to date system for comparison.

- When necessary, the use of SIMD add-ons can be hidden in a data-parallel framework, and/or manufacturer supplied libraries.
CHAPTER 3. SOME PARALLEL ARCHITECTURES AND
PROGRAMMING MODELS FOR IMAGE PROCESSING

3.8 Conclusions

In this chapter we have evaluated parallelism on its various levels: conceptual models of parallelism, parallel architectures, software tools for parallelization, parallel software design patterns, and parallel programming languages. The purpose of this chapter was to establish the strong and weak properties in order to prioritize them using the requirements of our application area and the requirements of image processing.

We argued that granularity runs from fine grain to coarse grain, when the level of image processing runs from low-level to high-level. Given a certain fixed processing power this means that the demand on required communication speed is the highest in case of low-level image processing and the lowest on high-level image processing. Our experiments showed that the current speed of interconnection networks of distributed memory systems is high enough in comparison with the power of the processors. We also showed that data parallel processing of low-level image processing operations is beneficial, up to a high number of processors, i.e. small image chunk sizes. However, the watershed experiment showed that as soon as the data distribution over the processors becomes data dependent, data parallelism is less beneficial.

MIMD systems can nowadays easily be built from off the shelf commodity products. Currently, although still rare, shared memory processor boards can be found with two to four processors, however the software support for such a board is usually underdeveloped. Our experiments showed that for low-level kernel operations shared memory processor systems are not so beneficial. They are, however, beneficial for global low-level operations such as FFTs.

We concluded that a distributed memory system (Cluster Of Workstations) is a good solution for parallel image processing. However, there may be reasons induced by the application, to go for the adventure of either cumbersome programming in multimedia assembler instructions, or for investing in a co-processor board such as the IMAP-Vision (or in near future the IMAP-CE) that can easily be programmed in 1DC. In any case, the parallelization can be hidden in a special version of a data parallel library for low-level image processing. The same holds for Distributed Bucket Programming, which can be considered as a special case of data parallel processing for irregularly distributed data.

On the topic of software tools for parallel image processing, we concluded that a good choice for a software library as building block for a more abstract parallel framework is MPI, as it has shown similar communication speeds as PVM and CRL, while it will probably have the best chance of surviving.

Writing parallel programs would ideally be supported by tools such as parallelizing compilers, with explicit parallel languages as the second best option. A negative point for explicit parallel languages is, however, that the current installed sequential software must either be re-programmed or treated in a different way. As such, a sequential language augmented with pragmas - parallelization annotations - are a better choice. Moreover, in our experiments we have found not much difference in the performance of an automatic parallelizing compiler and an explicit parallel compiler.

Consequently, we will focus in this thesis on a combined data and task parallel framework for low- and intermediate-level image processing on a distributed memory system.
3.8 Conclusions

With a note that either multimedia instructions, or an SIMD subsystem can be incorporated in the framework. The use of shared memory processor boards is allowed.
Chapter 4

Data Parallelism in Image Processing

4.1 Introduction

One of the approaches to programming parallel systems is called data parallelism. In a data parallel computation, the same operation is performed on different data elements simultaneously (in parallel). Many algorithms from image processing can be cast in a data parallel form. In image processing, data parallelism is obtained by distributing the data structures (usually images) over the processes. This approach is extremely well suited to SIMD systems (see Chapter 3). However, it is also quite common to use it on MIMD systems. Data parallelism is intended for very regular structures and operations such as low-level image processing algorithms. In this chapter, we introduce as a parallel design pattern for image processing the concept of algorithmic skeletons, which are higher order functions that encapsulate the data parallelism inherently present in the low-level image processing applications. We propose a method for exploiting data parallelism in low-level image processing applications by means of skeletons. The model is designed to be used in image processing applications on distributed memory systems and the data parallelism is transparent to the image processing application programmer.

4.2 Algorithmic skeletons

Distributed memory systems such as the IBM SP-2 [9], the Intel Paragon [19], Thinking Machines CM-5 [20], Cray T3D and T3E line of architectures, IBM RS/6000 SP series, and the Distributed ASCI Supercomputer (DAS) [17] offer significant advantages in terms of cost and scalability. Unfortunately, the utilization of all the available computational power in these machines requires a tremendous programming effort on
the side of the image processing application programmer. This creates a need for sophisticated compiler and run-time support for these distributed memory systems. In this thesis, we present a new environment which simultaneously exploits the data and task parallelism in image processing applications. In this chapter we present the data parallel approach.

In the last years there has been an increased interest in so-called skeleton-based parallel programming models in which the programmer’s task remains only to select and compose instances of pre-defined templates, chosen from a fixed repertoire, rather than being involved in low-level parallel constructs such as message passing, shared-memory access, or new constructs and annotations that are accompanying different parallel programming languages. The idea is that recurrent patterns of parallel computations can be encapsulated into higher-order program constructs or functions, and thus hiding all low-level, error-prone parallel implementation details which give so much problems from the application programmer. Skeletons are algorithmic abstractions which encapsulate different forms of parallelism, common to a series of applications. The aim is to obtain environments or languages that allow easy parallel programming, in which the user does not have to handle problems such as communication, synchronization, deadlocks or non-deterministic program runs. Usually, they are embedded in a sequential host language and they are used to code and hide the parallelism from the application programmer. Depending on the kind of parallelism used, skeletons can be classified into task parallel and data parallel ones. In the first case, the skeleton creates a series of tasks that run concurrently whereas in the second case it scatters and gathers the data to/from processors. In this chapter, we present some of the data parallel skeletons we have developed.

4.3 Related work

Many software architectures have been proposed for parallelizing image processing applications on parallel computers. In [56, 55], the authors propose a software architecture to support an application programmer in the design of image processing applications on parallel computers. The target hardware architecture is an MIMD system. The software architecture is based on abstract data types with a well-defined interface. On the application side, the interface offers the user a sequential programming model. On the hardware side, the interface is augmented with an efficient implementation on a parallel computer. Several data distributions are considered, among which the best one is selected, based on the performance characteristics of the image operations.

The concept of algorithmic skeletons is not new and much research has been done to demonstrate their usefulness in parallel programming. Most skeletons are polymorphic higher-order functions and can be defined in functional languages in a straightforward way. This is the reason why most skeletons are build upon a functional language [11, 15]. Some work has also been done in using skeletons in image processing. In [97] Serot et al. present SKiPPER, a programming environment dedicated to the fast prototyping of parallel vision algorithms on MIMD-DM (Distributed Memory) platforms. SKiPPER is based upon the concept of algorithmic skeletons. The source
4.4 Data mapping

The problem of data mapping consists of how to assign data elements to processors such that inter-processor communication is minimized. Two issues should be considered. First, the data locality issue is a critical one in the programming of distributed memory systems, because, in general, in such systems communication is much more expensive than computation. Care must be taken to avoid unnecessary communication. To accomplish this it is desirable to have the data locally available to each processor. The second issue, load balancing, is also very important. It would be desirable to assign the same amount of work to each processor, otherwise we waste some of the system’s resources. In this section, we will take a look at the mapping problem in low-level image processing: how to map a 2D grey value image to a collection of nodes in a distributed memory system.

Suppose we have an image $A$ of $n \times n$ elements

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix} \tag{4.1}$$

and a linear array of $p = (p_0, p_1, \ldots, p_{n-1})$ processors with $p < n$. Several data distributions are possible [57]. In Figures 4.1, 4.2, 4.3, 4.5, and 4.6 we present some important types of data distributions: row-stripe distribution, column-stripe distribution, block-wise distribution, row-cyclic distribution, window-wise distribution and helicoidal distribution.

1. **Row-stripe distribution.** If $p$ evenly divides $n$, then the first $n/p$ rows are assigned to processor 0, the next $n/p$ to processor 1, etc. If $p$ does not divide $n$
evenly, suppose that \( n = qp + r \), where \( 0 \leq r < p \), then we can assign \( \lfloor n/p \rfloor \) rows to the first \( r \) processors, and \( \lceil n/p \rceil \) rows to the remaining \( p - r \) processors, see Figure 4.1.

2. **Column-stripe distribution.** The description is the same as at the row-stripe distribution, but applied to columns.

3. **Block distribution.** If \( p \) evenly divides \( n \), then the first \( (n/p) \times (n/p) \) block is assigned to processor 0, the next to processor 1, etc. If \( p \) does not divide \( n \) evenly, suppose that \( n = qp + r \), where \( 0 \leq r < p \), then we can assign \( \lfloor n/p \rfloor \) blocks to the first \( r \) processors, and \( \lceil n/p \rceil \) blocks to the remaining \( p - r \) processors, see Figure 4.3.

4. **Row-cyclic distribution.** The first row is assigned to processor 0, the second row to processor 1, ..., the \( p - 1 \)th row to processor \( p - 1 \), the \( p \)th row is assigned to processor 0 again, etc., see Figure 4.4.

5. **Column-cyclic distribution.** Is the same as the row-cyclic distribution, but with respect to the image columns.

6. **Window-wise distribution.** The image is split into blocks of size \( \sqrt{P}/\sqrt{P} \) and the elements of each block are assigned to a processor, see Figure 4.5.

7. **Helicoidal distribution.** The elements of each row of the image are assigned cyclically to each of the available processors, starting with the first processor. The next row assigns the elements cyclically, starting with the second processor, etc., see Figure 4.6.

```
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4
```

```
1 1 2 2 3 3 4 4
1 1 2 2 3 3 4 4
1 1 2 2 3 3 4 4
1 1 2 2 3 3 4 4
1 1 2 2 3 3 4 4
1 1 2 2 3 3 4 4
1 1 2 2 3 3 4 4
1 1 2 2 3 3 4 4
```

Figure 4.1: Row-stripe data distribution

Figure 4.2: Column-stripe data distribution

To simplify the implementation of the data parallel framework we have considered the row-stripe data distribution of the images involved in an image processing application. The implementation can be extended with the other types of data distributions, enumerated above. The most interesting ones are the row-stripe, column-stripe and
Figure 4.3: Block-wise data distribution

Figure 4.4: Row-cyclic data distribution

Figure 4.5: Window-wise data distribution

Figure 4.6: Helicoidal data distribution

block-wise distributions. The other types of distributions involve too much communication overhead in image processing applications.

4.5 Simultaneous exploitation of data and task parallelism

The data parallel framework based on algorithmic skeletons is part of an environment designed for embedding the simultaneous exploitation of data and task parallelism in image processing applications. Chapter 5 will discuss in detail the task parallel framework which is based upon the data parallel framework. Figure 4.7 depicts the interconnection between the different modules of the proposed parallel environment. The source program is a "skeleton" specification of the image processing application,
in which all data parallelism is made explicit by means of composing instances of
the above-mentioned skeletons, each instance taking as parameters user-specific im-
age processing sequential functions written in C. A separate module called Syntactic
analyzer / Dependence analyzer (not yet implemented but explained in Chapter 5)
turns this specification into a task graph, the Image Application Task Graph (IATG),
presented in Chapter 5, with nodes associated to image processing functions (which
are executed on sets of processors from the available pool of processors), and edges
representing communication channels. On the extracted IATG we apply Mapping
and Scheduling Algorithms (named MSA) to find the minimum execution time of the
application. This module computes the set of processors and the corresponding execu-
tion time for each node (image processing operator) in the graph. The associated
communication times are also determined. The MSA's intermediate output is a list
of sets of processors on which the image processing operators (executed via skeletons)
have to be computed. The mapping and scheduling module needs information about
the execution costs of the image processing operators (executed via skeletons) on a
varying number of processors. This information is provided by the Cost estimation
/ Profiling module which communicates with the Skeletons library. This library con-
tains skeletons for low-level, intermediate-level, and high-level image processing. Only
the framework for low-level image processing is implemented in the module Distribute,
Compute, and Gather (DCG). There is also a skeleton Data Sending and Receiving
(DSR) which is responsible for communication of the images across the processors.

4.6 Algorithmic skeletons

Template-based systems employ a library of predefined solutions to the problems in-
curred by parallelization. Each solution in the library takes into account only a re-
stricted set of parallel computations that either share the same parallel paradigm or
share for the computation the same type of operators (as in the case of the image pro-
cessing operators). The solutions presented in the library are higher-order functions
called skeletons. In the template-based systems initially proposed in the literature,
the programmer is provided with a library of higher-order functions. We provide them
with a functional framework, or with high-level program "templates" in the context of
the conventional imperative languages. The language in which the various skeletons
are defined is called the host language.

Higher order functions are functions that accept other functions as arguments and
return new functions as results. The function which is returned is a structural composi-
tion of the argument functions and possibly some others. For clarity, let us consider
the integer square function, defined as in Formula 4.2.

\[ \text{sqr}(x) = x \times x \]  \hspace{1cm} (4.2)

The square function takes an integer as an argument and returns an integer as result,
and the signature of the function is denoted as in Formula 4.3.
Some simple functions are able to operate upon items of more than one data type because they do not inspect the details of the argument but only its structure. An example is the function head which takes a list of items of a particular type as its argument and return the first item on the list as its result. Clearly, the type of the items is of no consequence and such functions are called polymorphic (having many forms). Their type can be denoted using symbols to represent arbitrary types, as in Formula 4.4, where the square brackets denote “list of” and “a” stands for any type.
Higher order functions are slightly more complicated. Rather than taking and returning simple data items, these functions accept other functions as arguments and return new functions as results. The function returned is a structural composition of the argument functions and possibly some others. An example, is the function map. If $f$ is a function described by Formula 4.5 then the result of applying map to function $f$ is another function $map\ f$ defined in Formula 4.6, a function which has a list of elements of type $a$ as its argument and returns a list of elements of type $b$. The effect of $map\ f$ is to apply $f$ to every element in the argument list, placing the result in the corresponding position in the result list. We can rewrite the map function as in Formula 4.7.

$$f : a \rightarrow b \quad (4.5)$$

$$(map\ f) : [a] \rightarrow [b] \quad (4.6)$$

$$map : (a \rightarrow b) \rightarrow ([a] \rightarrow [b]) \quad (4.7)$$

Higher order functions are the basis of the algorithmic skeletons. They do not deal with the low level details of a particular problem. Instead, they capture the computational structure of a whole class of algorithms at a high level. Solutions to particular problems can be obtain by supplying the appropriate argument functions to the skeleton. Thus, we can equally well use map in conjunction with different functions to obtain different results, using a method which has the same overall structure. For example, given a function sqrt which returns the square root of a positive integer we can easily map sqrt to transform a list of positive integers in the same way, see Formula 4.8.

$$(map\ sqrt) : [unsigned\ int] \rightarrow [unsigned\ int] \quad (4.8)$$

The high-level algorithm is identical to that of map sqrt. The low-level argument function makes the difference. Being more explicit, the use of map ties down the essential structure of our computation, while its argument function produces a problem specific program which solves the problem at hand.

The programming model can now be described. In this model, the image processing application programmer is presented with a selection of specialized higher order functions (skeletons) from which one must be selected as the outermost function in the program. The programmer may then describe the lower level image processing
functions to produce an application specific final program. We will refer to the higher order functions as *algorithmic skeletons*.

Configuration skeletons model the partition of arrays (usually images) and the allocation of their components to processors. The images can be either sequential (residing on a single processor) or parallel (distributed across processors). Parallel images are the ones that can be used within data-parallel skeletons, while sequential images are to be used by sequential portions of the program. For example, a two dimensional sequential image is described by Formula 4.9.

\[
SeqImage = (index_1, index_2, \alpha)
\]  

(4.9)

Each image element (pixel) is of type \(\alpha\) and each index \(x\) is of range \(0 \div index_1 - 1\) and each index \(y\) is of range \(0 \div index_2 - 1\).

A two dimensional parallel image is described by the Formula 4.10, where \(index_1\), \(index_2\), and \(\alpha\) have the same meaning, and \(n\) is the number of processors on which the parallel image is distributed.

\[
ParImage = (index_1, index_2, \alpha, n)
\]  

(4.10)

In the following, a *ParImage* is denoted using \(<< im_1, ..., im_n >>\).

The *partition* skeleton divides a sequential image into a parallel image of sequential subimages. The skeleton is described by Formula 4.11.

\[
\text{partition} : \quad \text{Partition\_pattern} \quad \rightarrow \quad SeqImage = (idx_{1a}, idx_{2a}, \alpha) \\
\rightarrow \quad ParImage = (idx_{1p}, idx_{2p}, SeqImage, n)
\]  

(4.11)

where *Partition\_pattern* is a function mapping the index of a sequential image onto the corresponding index of a parallel image. Some commonly occurring partitions such as row/column block (*row\_block, col\_block, rowcol\_block*) and row/column cyclic (*row\_cyclic, col\_cyclic*) may be provided as built-in functions.

The *align* skeleton is presented in Formula 4.12 and pairs corresponding images in two distributed images together to form a configuration.

\[
\text{align} : \quad ParImage = (idx_1, idx_2, \alpha, n) \times ParImage = (idx_1, idx_2, \beta, n) \\
\rightarrow \quad ParImage = (idx_1, idx_2, (\alpha, \beta), n)
\]  

(4.12)

The *map* skeleton encompasses most of the regular, data-parallel strategies in which the input data is divided into a fixed number of partitions and each one is processed
by a different processor. The map skeleton is presented in Formula 4.13 and is defined as in Formula 4.14. The map skeleton takes a parallel image and process the value of each pixel of the parallel image, according to a function $f$.

\[
\begin{align*}
\text{map} &: (\alpha \rightarrow \beta) \\
&\rightarrow \ ParImage = (idx_1, idx_2, \alpha, n) \rightarrow \ ParImage = (idx_1, idx_2, \beta, n)
\end{align*}
\]  

(4.13)

(4.14)

The gather skeleton is used to gather a parallel image distributed across several processors into a sequential image which resides on a single processor. The gather skeleton is presented in Formula 4.15.

\[
\begin{align*}
\text{gather} &: \ Partition\_pattern \rightarrow \ ParImage = (idx_{1p}, idx_{2p}, SeqImage, n) \\
&\rightarrow \ SeqImage = (idx_{1s}, idx_{2s}, \alpha)
\end{align*}
\]  

(4.15)

The skeletons described in the following sections are based on the map, partition and gather skeletons. The parallel target hardware is a distributed memory system with the processors interconnected via an interconnection network which can have different topologies.

### 4.7 The data parallel framework

![DCG skeleton for point operators](image)

Figure 4.8: DCG skeleton for point operators

From the operator description given in the previous sections we conclude that point, neighborhood, and global image processing operators can be parallelized using the data parallel paradigm with a master-slave approach. A master processor is selected
for splitting and distributing the data to the slaves. The master also processes a part of the image. Each slave processes its part of the image and then the master gathers and assembles the output image. Different types of data distributions can be used, as
they have been presented in the previous section.

In Figures 4.8, 4.9, and 4.10 we present the data parallel paradigm with a master-slaves approach for point, neighborhood, and global operators. For global operators we send the entire image to the slaves but each slave will process only a certain part of the image. In order to avoid extra inter-processor communication for exchanging border information needed in neighborhood operators we extend and partition the image as depicted in Figure 4.9. In this way, each slave processor receives all the data needed for applying the neighborhood operator.

Based on the above observations, we identify a number of skeletons for parallel processing of low-level image processing operators. They are named after the type of low-level operator and the number of images involved in the operation. Headers of some skeletons are shown below. All of them are based on a "Distribute Compute and Gather"(DCG) main skeleton, a combination of the partition, map, and gather skeletons presented in the previous section, intended for applications that employ regular low-level image processing algorithms. The implementation of all the skeletons is based on the principles described in the previous paragraph, see Figures 4.8, 4.9, and 4.10.

Each skeleton can be executed on a set of processors. From this set of processors a host processor is selected to split and distribute the image(s) to the other nodes. Every other node from the set receives a part of the image(s) and the image operator which should be applied to it. Then the computation takes place and the result is sent back to the host processor. The language in which the various skeletons are implemented is called the host language. We have implemented the skeletons in C using the MPI-Panda library [99, 89] on a distributed memory system cluster [17] with up to 128 nodes, under the Linux operating system [61]. The implementation is transparent to the user. The programmer of the image processing application selects a skeleton from the skeleton library, after detecting the data parallelism present in an image processing operator. The selected skeleton will be the outermost function in the program. Then the programmer instantiates the various parts of the skeleton using the full power of the host language.

4.8 Skeletons for low-level image processing

We have developed algorithmic skeletons for various types of low-level operators (point, neighborhood, global) and for a different number of input/output images. The skeletons are presented in Table 4.1. With each algorithmic skeleton we associate a parameter which represents the task number corresponding to that skeleton. This is used by the task parallel framework, presented in the next chapter. Depending on the skeleton type, one or more identifiers of the images are given as parameters. The last argument is the operator for processing the image(s). So, each skeleton is used for a number of low-level image processing operators that work in a similar way (for instance all dyadic point operators take two input images, combine and process them depending on the operator type and then produce an output image). Depending on the operator type and the skeleton type, there might exist additional parameters necessary for the
image operator. For point operators we assigned the ImagePointDist skeletons, for neighborhood operators we assigned the ImageWindowDist skeletons, and for global operators we assigned the ImageGlobalDist skeletons. Some of the skeletons modify the input image (ImagePointDist.1I0, ImageWindowDist.1I0, ImageGlobalDist.1I0, so 1I0 stands for one Input/Output image), other skeletons take a number of input images and create a new output image, for example the ImagePointDist.2I1O skeleton for point operators takes two input images and creates a new output image. This skeleton is necessary for dyadic point operators (like addition, subtraction, etc., see Table 4.1) which create a new image by processing two input images. Similarly, the skeleton ImagePointDist.3I1O for point operators takes 3 input images and creates a new output image. An example of a low-level image processing operator suitable for this type of skeleton is the squared difference between one reference image and two disparity images, an operator used in the multi-baseline stereo vision application, see Table 4.1 and Section 4.4. Similar skeletons exist also for local neighborhood and global operators. ImagePointDist.1I0.C is a skeleton for monadic point operators which need a constant value as parameter, for processing the input image, see Table 4.1.

The parameters used and their meaning are given in Table 4.2.

Apart from the skeletons, the environment also contains a vector of distributed images. This allows us to compose a complete image from the parts that were created by or sent to different processors. This vector consists of the images created during the image processing application, see Figure 4.11. Each of the images im11, ..., imp2 needed in an image processing application which must be parallelized on a p processor distributed memory system, is mapped onto the processors where they were created or sent.

The skeletons library also contains the DSR module (see Figure 4.7) responsible for sending and receiving data between processors. This module includes SendImage and ReceiveImage skeletons.

In Figure 4.12 we present an example of using the skeletons to code a very simple image processing application in a data-parallel way. It is an image processing application for edge detection using the Sobel operator. First we read the input image and we create an output image and a 3 × 3 window. Then we apply the Sobel operator to the distributed image at num_nodes number of processors. num_nodes is the number of nodes on which the application is run and is detected on the first line of the partial code showed in Figure 4.12. image.in is the name of the input image given as input parameter to both skeletons and image.l, image.s are the output parameters (images) for each skeleton. We have used the ImageWindowDist.1I1O skeleton to perform the operator. The last two parameters are the window used (which contains information about the size and the data of the window) and the image operator that is applied via the skeleton.

### 4.9 Skeleton approach discussion

Having presented the method of the skeleton approach for exploiting data parallelism in image processing applications, we can discuss the major benefits and drawbacks of
Table 4.1: The skeletons library

```
... void ImagePointDist_1I0(unsigned int n, char *name, int p, list_proc set, void(*im_op)());
     // DCG skeleton for monadic point operators - one Input/Output

void ImagePointDist_1I0.C(unsigned int n, char *name, int p, list_proc set,
    void(*im_op)(), float ct);
     // DCG skeleton for monadic point operators which need a constant value as
     // parameter, one Input/Output

void ImagePointDist_1I0.O(unsigned int n, char *name1, char *name2, int p, list_proc set,
    void(*im_op)());
     // DCG skeleton for monadic/dyadic point operators - one Input and one Output

void ImagePointDist_1I0.1I(unsigned int n, char *name1, char *name2, int p, list_proc set,
    void(*im_op)());
     // DCG skeleton for monadic/dyadic point operators

void ImagePointDist_2I0.OO(unsigned int n, char *name1, char *name2, char *name3,
    int p, list_proc set, void(*im_op)());
     // DCG skeleton for dyadic - triadic point operators - 2 Inputs and one Output

void ImagePointDist_2I0.O(unsigned int n, char *name1, char *name2, char *name3,
    char *name4, int p, list_proc set, void(*im_op)());
     // DCG skeleton - 2 Inputs and 2 Outputs

void ImagePointDist_3I0.OO(unsigned int n, char *name1, char *name2, char *name3,
    char *name4, int p, list_proc set, void(*im_op)());
     // DCG skeleton for triadic point operators - 3 Inputs and one Output

void ImageWindowDist_1I0(unsigned int n, char *name, Window *win, int p,
    list_proc set, void(*im_op)());
     // DCG skeleton for neighborhood operators - one Input/Output

void ImageWindowDist_1I0.O(unsigned int n, char *name1, char *name2, Window *win,
    int p, list_proc set, void(*im_op)());
     // DCG skeleton for neighborhood operators - one Input and one Output

void ImageGlobalDist_1I0(unsigned int n, char *name, int p, list_proc set,
    void(*im_op)());
     // DCG skeleton for global operators - one Input/Output

void SendImage(unsigned int i, char *name, int t);
     // DSR skeleton used by the current processor which sends to processor i
     // the image "name", with tag t
```
### Table 4.2: Meaning of skeleton parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>The task number</td>
</tr>
<tr>
<td>name</td>
<td>The image name (more images can appear in the header)</td>
</tr>
<tr>
<td>ct</td>
<td>A constant value necessary for the image operator</td>
</tr>
<tr>
<td>win</td>
<td>The window size (for neighborhood operators)</td>
</tr>
<tr>
<td>p</td>
<td>The number of processors</td>
</tr>
<tr>
<td>set</td>
<td>The set of processors</td>
</tr>
<tr>
<td>im_op</td>
<td>The image processing operator</td>
</tr>
<tr>
<td>i</td>
<td>The processor number</td>
</tr>
</tbody>
</table>

![distributed vector of images](image)

Figure 4.11: The distributed vector of images

this approach.

The skeletons introduced in the previous sections are parallel design patterns for low-level image processing operations, that encapsulate the available data parallelism. The application programmer is presented with a selection of algorithmic skeletons which might be combined to code the application. Each type of skeleton can be applied only to a certain group of image processing operators. Usually, one would like to perform each skeleton on the maximum available number of processors, so the number of processors is usually hidden in the skeleton implementation. It is either the total
Example ()
BEGIN
    num_nodes=ImageInitPar(argc,argv);
    // initialize the parallel environment
    win=CreateWindow(3,3);
    // creates the 3x3 window
    im.i=ReadImage(256,256,"image_in",file,PAR_DOUBLE);
    // reads the input image from file
    im.s=CreateImage(256,256,"image.s",PAR_DOUBLE);
    // creates the output image
    ImageWindowDist_1110(num_nodes,"image_in","image.s",win,sobel.3x3);
    // applies the skeleton to execute the operator
END

Figure 4.12: A Straightforward Implementation using skeletons

number of processors, or a certain value which is the output of a scheduling algorithm (see next chapter). The skeletons that we have introduced can be executed on any number of processors and this number is not hidden from the skeleton interface. Each skeleton is executed on a set of processors from which a master processor is selected. The master processor has the role of scattering and gathering the image to the other processors from the set allocated for the image processing operator computation via the selected skeleton. The reason that the skeleton framework is designed in this way is for further use in combination with a task parallel approach. It should be mentioned that considering the algorithmic skeleton approach by itself, proves to be not the best one. The communication primitives extensively used by the skeleton method are the scatter and gather ones. An extensive list of communication primitives is presented below, see also Figure 4.13 which illustrates the data movement associated with each primitive.

- **Transfer**: a single source processor sends a message to a single destination processor.
- **OneToManyMulticast**: a single source processor sends a message to all other processors in the given group.
- **ManyToManyMulticast**: all processors in the group send data to all other processors in that group.
- **Scatter**: a single source processor sends different messages to all other processors in that group.
- **Gather**: all processors send messages to a single destination in the group.
- **Reduction**: reduction of data using a simple associative and commutative operation over all processors in a group.

Table 4.3 shows the time complexity of functions that correspond to these primitives on a hypercube architecture with p processors.
4.9 Skeleton approach discussion

![Diagram of communication primitives]

Figure 4.13: Types of communication primitives

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transfer(m)</td>
<td>(O(m))</td>
</tr>
<tr>
<td>OneToManyMulticast(m,p)</td>
<td>(O(m \times \log p))</td>
</tr>
<tr>
<td>ManyToManyMulticast(m,p)</td>
<td>(O(m \times \log p))</td>
</tr>
<tr>
<td>Scatter(m,p)</td>
<td>(O(m \times p))</td>
</tr>
<tr>
<td>Gather(m,p)</td>
<td>(O(m \times p))</td>
</tr>
<tr>
<td>Reduction(m,p)</td>
<td>(O(m \times \log p))</td>
</tr>
</tbody>
</table>

Table 4.3: Cost complexity of communication primitives on a hypercube

From Table 4.3 it should be noted that the costs of communication for the scatter/gather routines are amongst the highest, and repetitive use of them for the execution of every skeleton induces significant overhead. This is an important drawback. A method to reduce the communication overhead would be to use a “lazy parallelization approach” as described in [93]. This is a method for optimization of data parallel image processing applications, based on a finite state machine. It aims at reducing the redundancy based on the observation that, after a skeleton execution, the data that was distributed over the processors is often used in the following skeletons as well. As a consequence, a gather operation which is performed within one skeleton can be avoided in order to skip a following scatter operation performed in the next skeleton. The “lazy parallelization approach” assumes that all the communication operations
inserted are redundant, as long as the parallel code remains consistent. It must however be noted that this approach holds only in the case where the set of processors allocated to successive skeletons are identical.

4.10 Experiments

Multi-baseline stereo vision

We will illustrate the use of the skeleton approach with the multi-baseline stereo vision application. The input consists of three $n \times n$ images acquired from three horizontally aligned, equally spaced and parallel oriented cameras, see Figure 4.14. One image is the reference image, the other two are named the disparity images. For each of 16 disparities, $d = 1, ..., 16$, the first match image is shifted by $d$ pixels, the second match image is shifted by $2d$ pixels. A difference image is formed by computing the sum of squared differences between the corresponding pixels of the reference image and the shifted match images. Next, an error image is formed by replacing each pixel in the difference image with the sum of the pixels in a surrounding $13 \times 13$ window. A disparity image is then formed by finding, for each pixel, the disparity that minimizes the error. Finally, the depth of each pixel is displayed as a simple function of its disparity. The computation of difference images requires a point operator, while the computation of the error images requires neighborhood operators. The computation of the disparity image also requires a point operator.

![Figure 4.14: The alignment of the three cameras](image)

Figures 4.15, 4.16 and 4.17 show an example of a left image and a right image together with the extracted depth image.

In Figure 4.18 we present a standard sequential implementation of the application. In Figure 4.19 we present the data parallel implementation of the application based on the proposed skeletons. The implementation using algorithmic skeletons is only slightly different from a standard sequential implementation. Besides creating the images on the host processor, the code is nearly the same, only the function headers differ. The skeletons have as parameters the name of the images, the window and the image operator, whereas in the sequential version operator headers have as parameters the images and the window. The skeletons are implemented in C using MPI [99]. Coding
the application by just combining a number of skeletons does not require much effort from the application programmer, yet it parallelizes the application.

It should be noted that the code presented in Figure 4.19 is not optimal. The update of the two disparity images can be performed directly on the processors containing the distributed data (images) and using the “lazy parallelization approach” [93] adapted to our skeleton framework we can skip the repetitive scatter/gather operations which are preceding and ending the execution of each skeleton. Also, to stay consistent with the task parallel approach based on skeletons (see the next chapter), the initial image and the disparity images are created/updated on a master processor selected from the list of given processors. This explains the large difference in the performance compared with the use of the “lazy parallelization approach” in the same application [95].
Mb_Stereo_seq ()
BEGIN
ref=CreateImage(256,256,"ref",PAR_DOUBLE);
    // creates the reference image
InitImage.ij(ref);
    // initializes the reference image
m1=CreateImage(256,256,"m1",PAR_DOUBLE);
    // creates the first disparity image
InitImage.ij(m1);
    // initializes the first disparity image
m2=CreateImage(256,256,"m2",PAR_DOUBLE);
    // creates the second disparity image
InitImage.ij(m2);
    // initializes the second disparity image
im=CreateImage(256,256,"im",PAR_DOUBLE);
    // creates the output image
InitImageZero(im);
    // initializes the output image
FOR (d = 1; d <= 16; d ++) DO {
    // for all 16 disparities
DisparityImage(m1,d);
        // updates the first disparity image
DisparityImage(m2,d);
        // updates the second disparity image
Image2DiffSqr(im,ref,m1,m2);
        // applies the squared difference operator
ImageErr(im,win);
        // computes the error image
ImageMin(im,min);
        // computes the disparity image
}
END

Figure 4.18: Sequential code of multi-baseline stereo vision
Mb_Stereo_data_par ()
BEGIN
 n=ImageInitPar(argc,argv);
  // get the number of processors
 IF (My_Processor()==MASTER) {
   // on the master processor
   ref=CreateImage(256,256,"ref",PAR_DOUBLE);
   // creates the reference image
   InitImage_i(j)(ref);
   m1=CreateImage(256,256,"m1",PAR_DOUBLE);
   // creates the first disparity image
   InitImage_i(j)(m1);
   m2=CreateImage(256,256,"m2",PAR_DOUBLE);
   // creates the second disparity image
   InitImage_i(j)(m2);
   im=CreateImage(256,256,"im",PAR_DOUBLE);
   // creates the output image
   InitImageZero(im);
 }
 win=CreateWindow(13,13);
 FOR (d = 1; d <= 16; d += ) {
   IF (My_Processor()==MASTER) {
     // on the master processor
     DisparityImage(m1,d);
     // updates the first disparity image
     DisparityImage(m2,2d);
     // updates the second disparity image
   }
   ImagePointDist_3L1O(d,"im","ref","m1","m2",n,list_nodes,Image2DifSqr);
   // computes the squared difference operator via a skeleton
   ImageWindowDist_1IO(d,"im",n,list_nodes,win,ImageErr);
   // computes the error image via a skeleton
   ImagePointDist_1IO(d,"im",n,list_nodes,ImageMin);
   // computes the disparity image via a skeleton
 }
END

Figure 4.19: Parallel code of multi-baseline stereo vision
Figure 4.20: Speed-up for the data-parallel approach applied to multi-baseline stereo vision

In Figure 4.20 we present the speedup of the multi-baseline stereo vision application implemented with skeletons. From the graph we can observe that for normal image sizes of 256 x 256 the speedup results are not larger than 4. A kernel of 13 x 13 is considered rather large and by using more than 32 processors the speedups start to decrease, due to communication overhead caused by the border effects of the window used. Due to the kernel size the communication time needed to send the corresponding image stripe to a processor, is much larger than the computation time necessary for that processor to perform its task. As the image size increases, the speedup performance improves, but is still limited. This is because the communication time of our implementation is comparable with the computation time, due to repetitive scatter and gather of the images performed by each skeleton and because the complexities of the squared difference (point operator) and error (neighborhood operator) operators are not very high.

4.11 Conclusions

In this chapter we have introduced the concept of algorithmic skeletons and we have presented a data parallel image processing framework based on algorithmic skeletons. The framework is designed in such a way that allows easy implementation and utilization. The application programmer can select the skeletons from a skeleton library. The skeletons are designed for specific types of image processing operators like point operators (monadic, dyadic or triadic), neighborhood operators and global operators. Each skeleton can be executed on an arbitrary number of processors. The skeletons offer an easy way for a parallel implementation of an image processing application. Depending
on the image processing application, one must select the appropriate skeletons corresponding to the image operators involved in the image processing application. The data parallel framework described in this chapter was designed as part of a more general approach for simultaneous exploitation of data and task parallelism. This is the reason why the data parallel method based on skeletons and the results obtained by using pure data parallelism are not optimal, see Sections 4.9 and 4.10. The main lesson drawn from this chapter, however, was not on raw performance but on the effectiveness of the skeleton approach for writing data parallel image processing applications. In an image processing application that is parallelized using the skeleton approach, the images are usually created/loaded on one processor (named the master processor) and then using the appropriate skeletons, they are distributed to a set of processors for processing. To acquire a better efficiency the repetitive scatter and gather of the images should be restricted as much as possible, by applying some optimization methods [93, 95]. The combined data and task parallel approach for image processing is presented in the next chapter.
Chapter 5

Mixed Data and Task Parallelism in Image Processing

5.1 Introduction

Recently it has been shown that exploiting both task and data parallelism in a program to solve very large computational problems, may yield better speedups compared to either pure data parallelism or pure task parallelism [85, 103, 16]. The main reason is that both task and data parallelism are relatively limited, and therefore using only one of them bounds the achievable performance. Thus, exploiting mixed task and data parallelism has emerged as a natural solution. We show that applying both data and task parallelism can further improve the speedup at the application level.

There has been a considerable effort in adding task-parallel support to data-parallel languages, as in Fx [102], Fortran M [29], Paradigm HPF [85], or in adding data-parallel support to task-parallel languages, such as in Orca [42]. In order to fully exploit the potential advantage of the mixed task and data parallelism, efficient support for task and data parallelism is a critical issue. This can be done not only at the compiler level, but also at the application level and applications from the image processing field are very suitable for this technique.

Mixed task and data parallel techniques use a directed acyclic graph, in the literature also called a Macro Dataflow Graph (MDG) [85, 36, 35], in which data parallel tasks (in our case the image processing operators) are the nodes and the precedence relationships are the edges. In the following we name this graph the Image Application Task Graph (IATG).

Making the parallelism fully explicit is not so easy. We have to identify data dependencies (constraints that arise from the flow of data between tasks), i.e., which image operators must be executed before other ones to preserve the semantics of the original
program. When given the IATG, we are given the precedence constraints. These are edges of the graph, which the application programmer has to provide. Implementing a technique to automatically derive the edges of the graph (the precedence constraints) is not the subject of our work. Usually it is impossible to derive the dependencies exactly, so approximations are used. This is still a research area for automatic parallelization techniques which aim to extract parallelism out of a source program. This extraction is automatic, i.e., done by the compiler, and is architecture-independent.

To put it in an application framework, the idea is to process sequential code and to automatically generate a parallel program. Let us consider the Fortran programming language. Many researchers have proposed language extensions to Fortran in order to ease data parallel programming; most well-known is the High Performance Fortran (HPF) language standardization [54]. The most significant feature of HPF is the data distribution directives it provides. These directives enable the programmer to specify a layout scheme for all the arrays in the program. Given such a layout scheme, an HPF compiler generates a data parallel program for any given data architecture. Also extensions have been proposed to HPF for the specification of task parallelism [41, 28]. Figure 5.1 depicts whole chain of transformations needed to convert, for example, a Fortran program into an HPF program.

Several methods have been developed for extracting the task graph structure for a given application. See [85, 84] for a detailed description of extracting the MDG structure for a given extended HPF application. In our research we did not consider this part. We start from the assumption that we have the specification of the application’s IATG available.

The extraction of the IATG is not part of our research although it is easier to extract the IATG from an image processing application than to extract the IATG from arbitrary code. Using the IATG containing all the information regarding the precedence constraints, we have to apply scheduling algorithms in order to obtain the minimal execution time for the image processing application. Although the task graph extraction has not been part of our research, we present in the next section some considerations on dependency analysis. For a more detailed presentation and algorithms, see [84].

In Table 5.1, we summarize the notations and definitions used throughout this chapter.
Table 5.1: Notations and definitions introduced and used in this chapter

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G = (V, E, w, c)$</td>
<td>An IATG</td>
</tr>
<tr>
<td>$V$</td>
<td>The set of $n =</td>
</tr>
<tr>
<td>$E$</td>
<td>The set of $e =</td>
</tr>
<tr>
<td>$w$</td>
<td>The task cost function</td>
</tr>
<tr>
<td>$c$</td>
<td>The edge cost function</td>
</tr>
<tr>
<td>$P$</td>
<td>The number of processors in the system</td>
</tr>
<tr>
<td>$W(G)$</td>
<td>The task graph width</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>The schedule function</td>
</tr>
<tr>
<td>$\bar{M}S(\sigma, P)$</td>
<td>The makespan function</td>
</tr>
<tr>
<td>$Ph(\infty)$</td>
<td>The problem of determining a schedule for unlimited resources</td>
</tr>
<tr>
<td>$Ph(P)$</td>
<td>The problem of determining a schedule assuming $P$ processors</td>
</tr>
<tr>
<td>$SP(\sigma, P)$</td>
<td>The speedup of a schedule $\sigma$ on $p$ processors</td>
</tr>
<tr>
<td>$EF(\sigma, P)$</td>
<td>The efficiency</td>
</tr>
<tr>
<td>$T_{exec}(t, p(t))$</td>
<td>The processing cost of a task</td>
</tr>
<tr>
<td>$T_{comm}(i, j)$</td>
<td>The communication time from processor $i$ to processor $j$</td>
</tr>
<tr>
<td>$p(t)$</td>
<td>The set of processors on which task $t$ is scheduled</td>
</tr>
<tr>
<td>$CP(G)$</td>
<td>The critical path of a task graph</td>
</tr>
<tr>
<td>$A(G)$</td>
<td>The average area of a task graph</td>
</tr>
<tr>
<td>$T_b(t)$</td>
<td>The bottom level of a task $t$</td>
</tr>
<tr>
<td>$T_t(t)$</td>
<td>The top level of a task $t$</td>
</tr>
<tr>
<td>$T_s(t)$</td>
<td>The start time of a task $t$</td>
</tr>
<tr>
<td>$T_f(t)$</td>
<td>The finish time of a task $t$</td>
</tr>
<tr>
<td>$T_r(p)$</td>
<td>The processor ready time of a processor $p$</td>
</tr>
<tr>
<td>$p_T$</td>
<td>The processor becoming ready the earliest</td>
</tr>
<tr>
<td>$RPA_{alg2}$</td>
<td>Relative performance of $alg2$ with respect to $alg1$</td>
</tr>
<tr>
<td>$DATA$</td>
<td>Pure data scheduling</td>
</tr>
<tr>
<td>$TASK$</td>
<td>Pure task scheduling</td>
</tr>
<tr>
<td>$TSAS$</td>
<td>Two Step Allocation and Scheduling</td>
</tr>
<tr>
<td>$TwoL$</td>
<td>Two Level parallelism</td>
</tr>
<tr>
<td>$CPR$</td>
<td>Critical Path Reduction</td>
</tr>
</tbody>
</table>

5.2 Dependency Analysis

Dependency analysis is a vital step, prior to scheduling. This statement is valid for both sequential computers (in instruction scheduling) and parallel computers (in task scheduling). Dependency analysis determines the ordering relationships between instructions in a basic block or tasks in a task graph, that has to be satisfied for the code to execute correctly. Here, in this section, we are concerned only with the
dependency analysis problem which appears in task scheduling. Although, we are not covering this aspect, we are presenting an overview of the methods used to accomplish this step. More details are presented in [16].

To understand what the task graph is, consider the application of multi-baseline stereo vision. The input consists of three $n \times n$ images acquired from three horizontally aligned, equally spaced cameras. One image is the reference image, the other two are match images. For each of 16 disparities, $d = 0, \ldots, 15$, the first match image is shifted by $d$ pixels, the second image is shifted by $2d$ pixels. A disparity image is then formed by finding, for each pixel, the disparity that minimizes the error. Finally, the depth of each pixel is displayed as a simple function of its disparity. The computation of the difference images requires simple point image operators on the three input images. The computation of the error images is somewhat more interesting, being a local neighborhood image operator. The pseudo code of the application is presented in Figure 5.2.

This section does not intend to present an exhaustive method to task precedence analysis, but uses an example to only sketch some of the techniques that may be used to extract the task graph of an application.

In the original program there is a total precedence order [16] between tasks. Let us write $t_1 <_{seq} t_2$ if task $t_1$ is executed before task $t_2$ in the original sequential code. We have then

$$t_{1,0} <_{seq} t_{2,0} <_{seq} t_{3,0} <_{seq} t_{4,0} <_{seq} t_{1,1} <_{seq} \ldots <_{seq} t_{4,15} <_{seq} t_{5} \quad (5.1)$$

However, some of these tasks are independent and can be executed in parallel, see Figure 5.3. Intuitively, independent tasks are tasks whose execution order can be interchanged without modifying the result of the program execution. A sufficient condition for this is that the tasks do not access the same variable. They can read the same variable but they cannot write into the same memory location. We can express this in a formal way. We associate to each task $t$ an input set $In(t)$ (read values) and an output set $Out(t)$ (written values). In our example:
5.2 Dependency Analysis

\[
\begin{align*}
\text{In}(t_{1,d}) &= \{m_1\} & \text{Out}(t_{3,d}) &= \{\text{diff}\} \\
\text{Out}(t_{1,d}) &= \{m_1\} & \text{In}(t_{4,d}) &= \{\text{diff}\} \\
\text{In}(t_{2,d}) &= \{m_2\} & \text{Out}(t_{4,d}) &= \{\text{err}(d)\} \\
\text{Out}(t_{2,d}) &= \{m_2\} & \text{In}(t_5) &= \{\text{err}_0, \text{err}_1, \ldots, \text{err}_{15}\} \\
\text{In}(t_3,d) &= \{\text{ref}, m_1, m_2\} & \text{Out}(t_5) &= \{\text{disp}\}
\end{align*}
\]

Two tasks \(t\) and \(t'\) are dependent (we write \(t \perp t'\)) if they share a variable that is an output variable for either one or both of the tasks:

\[
t \perp t' \iff \begin{cases} 
\text{In}(t) \cap \text{Out}(t') \neq 0 \\
\text{Out}(t) \cap \text{In}(t') \neq 0 \\
\text{Out}(t) \cap \text{Out}(t') \neq 0
\end{cases}
\]

These conditions are known as Bernstein's conditions [3, 16]. Tasks \(t_{1,d}\) and \(t_{2,d}\) which compute the new shifted images, are independent and must precede task \(t_{3,d}\), which reads those new images. Task \(t_{3,d}\) which computes the difference image must precede task \(t_{4,d}\) which reads the difference image and computes the error image. But tasks \(t_{1,d}\), \(t_{2,d}\), \(t_{3,d}\) and \(t_{4,d}\) are independent from tasks \(t_{1,d+1}\), \(t_{2,d+1}\), \(t_{3,d+1}\) and \(t_{4,d+1}\) for \(d = 0..14\). Task \(t_5\) must follow the tasks \(t_{4,d}\), \(d = 0..15\).

Given the dependency relation \(\perp\), we can extract a partial order relation from the total order relation \(\prec\) induced by the sequential execution of the program. If two tasks \(t\) and \(t'\) are dependent, i.e., \(t \perp t'\), we order them according to the sequential execution. We write \(t \prec t'\) if both \(t \perp t'\) and \(t \prec \) \(\prec\) \(t'\) are valid. The precedence relation \(\prec\) represents the dependencies that must be satisfied to preserve the semantics of the original program; if \(t \prec t'\) then \(t\) was executed before \(t'\) in the sequential code, and should also be executed before \(t'\) in the parallel program, because \(t\) and \(t'\) share a variable that is written by at least one of them.

To define \(\prec\) more accurately, in terms of order relations, we can write

\[
\prec \text{ equals } (\prec_{\text{seq}} \cap \perp)^+ 
\]

where \(^+\) denotes the transitive closure. In other words, we take the transitive closure of the intersection of \(\perp\) and \(\prec_{\text{seq}}\) to derive the set of all constraints that need to be satisfied to preserve the semantics of the sequential program. The original total ordering relation \(\prec_{\text{seq}}\) was to restrictive, only the partial ordering relation needs to be fulfilled. We can draw a directed acyclic graph to represent the dependency constrains that need to be satisfied. If these constraints are not enforced, we cannot guarantee that the semantics of the original problem are preserved. An edge \(e : t \rightarrow t'\) in the graph means that the execution of \(t'\) must begin only after the end of the execution of \(t\), whatever the number of available resources. We do not usually draw transitive edges on the graph, as they represent redundant information; if \(t \prec t'\) and \(t' \prec t''\) and
if there exists a dependency $t \perp t''$, then it will automatically be satisfied. We say that $t$ is a predecessor of $t'$ if $t < t'$ and if there is no task $t''$ in between, i.e., such that $t < t'$ and $t' < t''$. So far we have discussed data dependencies, i.e., constraints that arise from the flow of data between tasks. Besides data dependencies, there are also control dependencies. A control dependence is a constraint that arises from the control flow of the program. Control dependencies are generally omitted in a task graph, unless such a dependency is the only one that is connecting two nodes. There are four types of data dependencies:

- If $t < t'$ and the former sets a value that the latter uses, this is called a flow dependency or true dependency, which is a binary relation denoted $t \prec^f t'$.

- If $t < t'$, $t$ uses some variable’s value and $t'$ sets it, then we say that there is an anti-dependency between them, and we write $t \prec^a t'$.

- If $t < t'$ and both tasks set the value of some variable, we say there is an output dependency, and we write $t \prec^o t'$.

- Finally, if $t < t'$ and both tasks read the value of some variable, we say there is an input dependency between them, and we write $t \prec^i t'$.

A set of data dependency relationships may be represented by a directed graph originally called Macro Dataflow Graph (MDG) and renamed by us into Image Application Task Graph (IATG). In such a graph, the nodes represent the tasks (image processing operators) and the edges represent dependencies between them. Control dependencies are generally omitted in dependency graphs, unless such a dependency is the only one that connects two nodes. Examining the multi-baseline stereo vision application, we are led to the graph of Figure 5.3.

As we have seen in this section, dependency analysis is a tool vital in task scheduling, which is discussed in the forthcoming sections. For task scheduling, dependency analysis determines the ordering relationships between instructions that must be satisfied.
for the code to execute correctly, and so determines the freedom available to the task
scheduler to schedule the tasks on different processors in a way that improves the per-
formance of an application. We continue, in the following sections, with presenting the
Image Application Task Graph model and then the allocation and scheduling problem
within the context of image processing applications.

5.3 The Image Application Task Graph Model

Regular scientific applications, as the ones from image processing, can be formulated
in a task-oriented programming model. In this model, a parallel program consists of
a number of multiprocessor tasks (also called multitasks or M-tasks) which can be
executed in parallel if there is no data dependency between them. Each M-task can
be further executed in a data parallel way, by a subset of processors. The multitasks,
further denoted by tasks, and their dependencies can be represented by a task graph,
with the vertices standing for the tasks and the edges standing for the dependencies
between the tasks. A task parallel program consists of a collection of multiprocessor
tasks \( t_1, \ldots, t_w \) where each task can be executed on any number of processors. The
tasks do not need to be independent of each other but may be related by a precedence
relation \( \prec \) that specifies whether the tasks must be executed sequentially one after
another because of the data dependencies, i.e. \( t_i \prec t_k \) if \( t_i \) produces data required by
\( t_k \).

The execution of a task parallel program on a parallel architecture with \( P \) processors
has to guarantee that the input data required by a task \( t_k \) are available when the
execution of \( t_k \) starts, i.e.:

(i) The input data must have been produced by predecessor tasks \( t_i \) of \( t_j \) with
\( t_i \prec t_k \).

(ii) The input data must be available, i.e, each processor must have the data it needs
for its local computations.

Item (i) is satisfied when executing the tasks according to the precedence relation. To
satisfy (ii), a communication between tasks \( t_i \) and \( t_k \) with \( t_i \prec t_k \) might be necessary,
to send the data expected by task \( t_k \).

Definition 1 (MDG) A task parallel image processing program can be modeled by
a Macro Dataflow communication Graph [85], which is a directed acyclic graph \( G =
(V, E, w, c) \), where:

- \( V \) is the finite set of nodes which represents tasks (image processing operators)
- \( E \) is the set of directed edges which represent precedence constraints between
tasks: \( e = (t, t') \in E \) if \( t \prec t' \).
- \( w \) is the weight function \( w: V \rightarrow N^* \) which gives the weight (processing time)
of each node (task). Task weights are positive integers.
Figure 5.4: Task graph (IATG)  Figure 5.5: A well balanced IATG

- $c$ is the communication function $c : E \rightarrow N^*$ which gives the weight (communication time) of each edge. Communication weights are positive integers.

In the above definition, $N^*$ represents the set of natural numbers, without zero. This is not a restriction; task weights can be rational numbers. However, because there is a finite number of tasks, we can always approximate them to integers.

An Image processing Application Task Graph (IATG) [127] is, in fact, an MDG in which each node stands for an image processing operator and each edge stands for a precedence constraint between two adjacent operators. In this case, a node represents a larger entity than in the MDG, where a node can be any simple instruction from the program. Figure 5.4 represents an IATG with five tasks numbered from zero to four, where the node and edge weights are not given.

Some important properties of an IATG are:

- It is a weighted directed acyclic graph.
- Nodes represent image processing operators and edges represent precedence constraints between them.
- There are two distinguished nodes: START precedes all other nodes and STOP succeeds all other nodes.

We define a well-balanced IATG as an application task graph which has on each level the same type of tasks (image operators). Figure 5.5 would represent a well-balanced IATG if tasks 1 and 5 are identical, tasks 2, 3, 6 and 7 are identical, and tasks 4 and 8 are identical. An example is the IATG of the multi-baseline stereo vision application, described further in this chapter, and presented in Figure 5.3, which has as tasks on
5.3 The Image Application Task Graph Model

the first two levels the shift operator, on the third level the squared difference operator
applied to three images and on the last level the error operator. Moreover, the graph
edges form a regular pattern.

The basic types of nodes in an IATG are:

- **Image Library-Defined (I)** nodes that correspond to library-defined image
processing functions (operators) used in the given application. An I node corre-
sponds to a function body.

- **Loop (L)** nodes that correspond to loop constructs in the given application.
Loops can be FOR loops or WHILE loops. An L node contains the IATG
structure corresponding to the loop body.

- **Conditional (C)** nodes that correspond to conditional constructs in the given
application. A C node contains the IATG corresponding to the IF and ELSE
bodies.

The **I** nodes are the simplest entities in the IATG. L and C nodes may contain I
nodes.

The weights of nodes and edges in the IATG are based on the concepts of **processing**
and **communication** costs. Processing costs account for the computation and com-
unication costs of data parallel tasks (image processing operators corresponding to
nodes) and depend on the number of processors allocated to that node. Communi-
cation costs account for the costs of all data communication between nodes.

Our goal is that having an IATG that must be executed on a P processor system, to
find an optimal allocation of processors to the nodes of the graph (allocation) and an
optimal execution order of the tasks (scheduling) that will minimize the execution time
of the application. In the following, we will introduce the definitions and theorems of
allocation and scheduling, as they are used in [16]:

**Definition 2 (Mapping)** *Given a task graph IATG = \((V, E, w, c)\) and a target ar-
chitecture \(A\) the mapping problem consists of finding an allocation for each node (task)
of the graph onto a set of processors of \(A\) in order to minimize a suitable cost function.***

Typically, we want to minimize the execution time of a program, which is also known
as makespan and defined in Definition 4.

In the following sections we discuss mapping algorithms having a homogeneous dis-
tributed memory system as a target architecture.

A schedule \(\sigma\) of a task graph is a function that assigns a start time to each task. The
schedule must satisfy the conditions presented in Definition 3.

**Definition 3 (Schedule)** *A schedule of a task system IATG = \((V, E, w, c)\) is a func-
tion \(\sigma : V \rightarrow N^*\) such that:

\[
\forall e = (t, t') \in E, \left\{
\begin{array}{ll}
\sigma(t) + w(t) \leq \sigma(t') & \text{if alloc}(t) = \text{alloc}(t') \\
\sigma(t) + w(t) + c(t, t') \leq \sigma(t') & \text{otherwise}
\end{array}
\right.
\]
where \( t \) and \( t' \) are two successive tasks and \( \text{alloc}(t) \) denotes the processors executing task \( t \).

Typical questions for scheduling are the following:

1. What is the minimal execution time using \( p \) processors, where \( p \) is fixed?
2. What is the minimal execution time using an unlimited number of processors?
3. In both cases, how to assign to each task both a time-step at which its execution is started, and a processor number that is responsible for executing the task?

Answering these questions is the subject of this chapter.

**Definition 4 (Makespan, Pb(\( \infty \)), Pb(\( p \)))** Let \( G = (V, E, w, c) \) be an IATG.

1. Let \( \sigma \) be a schedule for \( G \). Assume \( \sigma \) uses at most \( P \) processors (let \( p = \infty \) if resources are unlimited). The makespan (or completion time) \( MS(\sigma, P) \) of \( \sigma \) is its total execution time:

\[
MS(\sigma, P) = \max_{t \in V} \{\sigma(t) + w(t)\} - \min_{t \in V} \{\sigma(t)\} \tag{5.4}
\]

2. \( Pb(\infty) \) is the problem of determining a schedule \( \sigma \) of minimal makespan \( MS(\sigma, \infty) \) assuming unlimited resources. Let \( MS_{\text{opt}}(\infty) \) be the value of the makespan of an optimal schedule with unlimited resources, in other words, \( MS_{\text{opt}}(\infty) = \min_\sigma MS(\sigma, \infty) \).

3. \( Pb(P) \) is the problem of determining a schedule \( \sigma \) of minimal makespan \( MS(\sigma, P) \) assuming \( P \) processors. Let \( MS_{\text{opt}}(P) \) be the value of the makespan of an optimal schedule with \( P \) processors.

**Definition 5 (Speedup, Efficiency)** Let \( G = (V, E, w, c) \) be a MDG, and \( \sigma \) a schedule for \( G \) with \( P \) processors:

1. The speedup is the ratio \( SP(\sigma, P) = \frac{\text{Seq}}{MS(\sigma, P)} \), where \( \text{Seq} = \sum_{t \in V} w(t) + \sum_{e \in E} c(e) \) is the sum of all task weights.

2. The efficiency is the ratio \( EF(\sigma, P) = \frac{SP(\sigma, P)}{P} = \frac{\text{Seq}}{p \times MS(\sigma, P)} \).

**Theorem 1** Let \( G = (V, E, w, c) \) be a task system. For any schedule \( \sigma \) with \( p \) processors,

\[
0 \leq EF(\sigma, p) \leq 1 \tag{5.5}
\]
Proof Let us consider the execution of a schedule $\sigma$. At any moment of the execution some processors are active and some processors are idle. At the end of the execution all tasks have been processed. Let $Idle$ denote the cumulated idle time of the $p$ processors during the whole execution. Because $Seq$ is the sum of all task weights, the quantity $Seq + Idle$ is equal to the area of the rectangle, i.e., the product of the number of processors by the makespan of the schedule:

$$Seq + Idle = p \times MS(\sigma, p)$$

(5.6)

Hence,

$$e(\sigma, p) = \frac{Seq}{p \times MS(\sigma, p)} \leq 1$$

(5.7)

Another way to express it more intuitively is: the speedup with $p$ processors is always bounded by $p$. No superlinear speedup can be obtained.

Theorem 2 Let $G = (V, E, w, c)$ be a task system. We have:

$$Seq = MS_{opt}(1) \geq MS_{opt}(p) \geq MS_{opt}(p + 1) \geq \cdots \geq MS_{opt}(\infty)$$

(5.8)

Proof

Let us consider an optimal schedule $\sigma$ with $p$ processors. We can view it as a schedule with $p + 1$ processors where the last processor is kept idle. Then, $MS(\sigma, p + 1) = MS(\sigma, p) = MS_{opt}(p)$, hence $MS_{opt}(p + 1) \leq MS_{opt}(p)$.

5.4 Mathematical Cost Models

This section presents important aspects of choosing appropriate functions to represent the processing and data transfer costs involved in an IATG. The cost functions we choose have to be accurate in practice and belong to the class of posynomial functions (see Appendix A). The processing cost function we use is an often used model. Also, the communication cost model is a known formula. For communication we distinguish between internal and external communication.

5.4.1 Processing cost model

A node in the IATG represents a processing task (an image processing operator applied via a DCG skeleton, as described in Section 4.9) that runs non-preemptively (without being interrupted) on any number of processors. Each task is assumed to have a computation cost $w$ that will be further denoted as $w(t, p)$, which is a function of the number of processors $n(t)$. The computation of the function parameters, i.e. $\alpha(t)$ and $\tau(t)$ can be obtained either by estimation or by profiling.
For being able to deduct the processor allocation for a certain application it is important to use a cost model which gives accurate results and, moreover, which satisfies the constraint of being a posynomial function. Such an estimation function is the Amdahl’s law. According to it, the execution time of the task \( t \) is:

\[
w(t, n(t)) = (\alpha(t) + \frac{1 - \alpha(t)}{n(t)}) \ast \tau(t)
\]  
(5.9)

where \( t \) is the task number, \( n(t) \) is the number of processors on which task \( t \) is executed, \( \tau(t) = w(t, 1) \) is the task’s execution time on a single processor and \( \alpha(t) \) is the fraction of the task that executes serially. It can be seen that

\[
0 \leq \alpha(t) \leq 1 \quad 0 \leq \tau(t)
\]  
(5.10)

In case we use estimation, the values of \( \alpha(t) \) and \( \tau(t) \) can be calculated for the data parallel image processing tasks and certain image sizes used in our benchmarks by actually measuring execution times for the tasks as a function of the number of processors they use and then using non-linear regression (for the fractional function presented in Formula 5.9 ) to fit the measured values to a function of the form shown above. From the regression procedure we can compute the values of \( \alpha(t) \) and \( \tau(t) \) for a certain task \( t \). Let us refine the processing cost function. From Formula 5.9 we can derived the simplified function presented in Formula 5.11, which depends only on the variable \( n \) which is the number of processors. For a given image processing task and a given data input (image size), the values \( \alpha \) and \( \tau \) are constant.

\[
f(n) = (\alpha + \frac{1 - \alpha}{n}) \times \tau
\]  
(5.11)

We can further simplify the function by substituting \( \alpha \) and \( \tau \) as in the Formula 5.12, where \( x \) is the number of processors, and \( \alpha = \frac{b}{a+b}, \tau = a + b. \)

\[
f(x) = \frac{a}{x} + b
\]  
(5.12)

The values of \( a \) and \( b \) from Formula 5.12 can be determined by using a regression method, presented in Appendix B.1.

In Table 5.2 we present the processing cost of the squared difference image processing operator, the actual execution time and the estimated execution time using the regression method, for 1024 × 1024 size images. The processing cost function for this image size is given in Formula 5.13.

\[
f(x) = \frac{0.737}{x} + 0.004
\]  
(5.13)
If we use profiling, the values of $\alpha$ and $\tau$ can be used directly through a table. When using profiling, we measure the execution times of the image processing operators used in the benchmarks, for all number of processors.

The cost function $w$ is a posynomial function with respect to $n(t)$. A detailed explanation of posynomial functions is presented in Appendix A. The most significant characteristic of posynomial functions is that it can be mapped onto a convex function through an elementary variable transformation. As a consequence, the critical path and average area, introduced in the next sections, are also posynomial functions with respect to $n(t)$. That means that the allocation problem can be reduced to a convex programming problem which can be solved in polynomial time [106]. More details are presented in Section 5.5.

**Lemma 1** $w(t, n(t))$ is a posynomial function with respect to $n(t)$.

**Lemma 2** $w(t, n(t)) \cdot n(t)$ is a posynomial function with respect to $n(t)$.

Proof of the previous Lemmas can be found in [84].

### 5.4.2 Communication cost model

In this section we consider the cost of communicating data (images) between the processors allocated to two successive tasks in the IATG.

We first reduce the complexity of the problem, by allowing only a single type of distribution scheme (row-stripe) of the data onto the processors allocated to a task, and secondly by transporting images between two processors only (the selected master processors from the two sets of processors), as shown in Figure 5.6. More complex communication models can be found in [94]. Our approach introduces overhead because after the execution of each image processing operator we have to gather the image back to the master processor, then send the image to the master processor of the set of processors assigned to the next (in the IATG) image processing operator, and then this master processor has to scatter the image to the other processors from the set etc. In this section we present the model for estimating the communication time of point-to-point operations, the ones that we use for sending data (images) between the two master processors. The time estimation for image scatter/gather is incorporated.
in the processing cost model. We think that this is a way to make the details of parallelism as transparent as possible in our semi-automatic framework. A possible improvement would be to directly transfer (in a transparent way) an image mapped on a set of processors corresponding to the current task, to the set of processors allocated to the subsequent task, see next section.

An edge $e = (u, v) \in E$ in an IATG corresponds to a precedence relationship between the corresponding tasks $t$ and $t'$ and has associated with it a communication cost denoted by $c(t, t')$ which depends on the network characteristics (latency, bandwidth) and the amount of data to be transferred. We can use either cost estimation or profiling to determine the communication time.

It should be emphasized that there are two types of communication times. First, we have the internal communication time which represents the time for internal transfer of data between the processors allocated to a task, see Figure 5.6. This quantity is part of the term $\alpha$ of the execution time associated to a node of the graph, as presented in Formula—5.9. Secondly, we have the external communication time which is the time of transferring data, i.e. images, between two processors. These two processors represent the master processors for the two associated image processing tasks (corresponding to the two adjacent graph nodes), see Figure 5.6. This quantity is actually the communication cost associated with an edge of the graph. Therefore, the communication cost associated with an edge of the graph is the cost of performing a point to point communication, between two processors, for sending the necessary data (image(s)).

Again, we can use either cost estimation or profiling to determine the communication time. In state-of-the-art distributed memory systems the time to send a message containing $L$ units of data from a processor $i$ to another processor $j$ can be modeled as:

$$c(i,j) = t_a + L \times t_b$$

(5.14)
where \( t_s, t_b \) are the startup costs and the costs per byte for point-to-point communication and \( L \) is the length of the message, in bytes. The estimation of the parameters \( t_s \) and \( t_b \) can be done in the same way as with the estimation of \( \alpha \) and \( \tau \) from the processing cost model. The linear regression method used in this case is also presented in Appendix B.2.

To validate the above formula, we ran an experiment on a distributed memory system which consists of a cluster of Pentium Pro/200Mhz PCs with 64MB RAM running Linux, and connected through Myrinet in a 3D-mesh topology, with dimension order routing [17]. Figure 5.7 shows the performance of point-to-point communication operations and the predicted communication time. The reported time is the minimum time obtained over 20 executions of the same code. It is reasonable to select the minimum value because of the possible interference caused by other users' traffic in the network. From these measurements we perform a linear regression method described in Appendix B.2, and we extract the communication parameters \( t_s \) and \( t_b \) in the same way as described in the previous section, for the processing cost model. In Figure 5.7 we see that the predicted communication time, based on the above formula, approximates the measured communication time very well.

![Figure 5.7: Performance of point-to-point communication on DAS](image)

5.4.3 Communication optimization

Data communication (data redistribution) is essential for implementing an execution scheme which uses both data and task parallelism. Individual tasks are executed in a data parallel fashion on subsets of processors and the data dependencies between tasks may necessitate not only changing the set of processors but also changing the distribution scheme. Figure 5.8 illustrates another approach of redistribution between a pair of tasks. Task \( \text{Task}_A \) is executed using seven processors and reads from distributed data \( D \). Task \( \text{Task}_B \) is executed using four processors and reads from the same data.
CHAPTER 5. MIXED DATA AND TASK PARALLELISM IN IMAGE PROCESSING

D. This necessitates the redistribution of data D from the seven processors executing task \textit{Task}_A to the four processors executing task \textit{Task}_B. In addition to changing the set of processors we could also change the distribution scheme of the data D. For instance, if D is a two dimensional image then \textit{Task}_A might use a block distribution for D, whereas \textit{Task}_B might use a row-stripe distribution.

![Data redistribution between two tasks](image)

Figure 5.8: Data redistribution between two tasks

This method can be used for optimizing our current model, where we use only one type of data distribution and only point to point communication between two processors for sending the necessary data from one task to the subsequent one. The implementation of the current skeletons, which stand for the tasks in the graph, should also be reconsidered in the sense that no further scatter/gather of the image on the master processor might be necessary, therefore reducing the overhead present in our current approach.

5.4.4 IATG cost properties

A task with no input edges is called an \textit{entry} task and a task with no output edges is called an \textit{exit} task. The length of a path from the graph is the sum of the computation and communication costs of all nodes and edges belonging to the path. We define the \textbf{Critical Path} [85] (CP) as the longest path in the graph.

\textbf{Definition 6 (Critical Path)} \textit{If we have a graph with N nodes, where N is the last node of the graph and } t_i \textit{ represents the finish time of node } i, w(i, n(i)) \textit{ is the execution time of task } i \textit{ on a set of } n(i) \textit{ nodes then the critical path is given by the formulas 5.15 and 5.16, where } PRED_i \textit{ is the set of immediate predecessor nodes of node } i.

\begin{equation}
CP = t_N 
\end{equation}

\begin{equation}
t_i = \max_{p \in \text{PRED}_i} (t_p + c(p, i)) + w(i, n(i))
\end{equation}
5.4 Mathematical Cost Models

Definition 7 (Average Area) We define the Average Area \[A\] of an \(N\) task IATG for a \(P\) processor system as in Formula 5.17, where \(n(i)\) is the number of processors allocated to task \(i\).

\[
A = \frac{1}{P} \sum_{i=1}^{N} w(i, n(i)) n(i)
\]  
(5.17)

The critical path represents the longest path in the IATG and the average area provides a measure of the processor-time area required by the IATG. Using the sum and max properties of posynomials and the two previous Lemmas 1 and 2, we can prove the critical path and average area to be posynomials as well. As shown in the next section, this is useful while making allocation and scheduling decisions.

In a two step allocation and scheduling algorithms, usually one has to solve the following minimization problem

\[
\phi = \min(\max(A, CP))
\]  
(5.18)

subject to

\[
1 \leq n(t) \leq P \quad \forall t = 1, n
\]  
(5.19)

where \(P\) is the total number of processors.

The intuition behind minimizing \(\phi\) in Equation 5.18 is that \(\phi\) represents a theoretical lower bound on the time required to execute the image processing application corresponding to the IATG. The execution time of the application can neither be smaller than the critical path of the graph nor be less than the average area of the graph. For a better understanding of these constraints we have pictured the processor system as a bin of width \(P\) where each node \(i\) of the IATG corresponds to a box of width \(n(i)\) (the set of processors allocated to the node) and height \(w(i)\) (the cost of execution, as described in Formula 5.9). Then, the execution time of the application is the height of the processor system bin which accommodates boxes corresponding to all the nodes in the IATG while preserving all the precedence constrains. Clearly, this height can neither be less than that of the tallest stack of boxes arising from the set of precedence constrains nor have area less than that required to accommodate all the boxes. The critical path of the IATG represents the former constraint and the average area of the IATG represents the latter constraint. Figures 5.9 and 5.10 illustrate the effect of the critical path and the average area on the execution time of an application. Each node of the IATG has associated a box: the width of the box represents the allocation \((n(i))\) for the node, and the height of the box is the cost \((w(i, n(i)))\) of the node.

Given the allocations and costs and ignoring the edge (communication) costs, we have computed the critical path and the average area for an eight processor system, using two allocation schemes. The first scheme has a higher critical path and the second scheme has a higher average area value. This means that the execution time for the
first scheme is constrained by the critical path and the second scheme by the average area. In terms of the processor system bin visualization, this translates to the height of the bin being constrained by the tallest stack of boxes in the first case and by the area required to accommodate all the boxes in the second case, as shown in Figures 5.11 and 5.12.

Figure 5.9: Dominating Critical Path

**Effect of allocation on Critical Path and Average Area**

The minimization problem can be solved using convex programming techniques, see Appendix A and Section 5.5.5.

### 5.5 Mapping and Scheduling Algorithms

In this section, we describe several scheduling algorithms designed for mixed task and data parallelization. The classical approach to scheduling in general, is the well-known list scheduling paradigm [40] introduced by Graham, which schedules single processor tasks (tasks running only on one processor). Scheduling is proved to be NP-complete for single processor tasks [60]. Since then, several other list scheduling algorithms were proposed, and the scheduling problem was also extended to multiple processor tasks (tasks that run non-preemptively on any number of processors) [85]. Multiple processor task scheduling is considerably harder than the list scheduling problem, therefore heuristic techniques need to be used.

#### 5.5.1 Definitions

In this section, we introduce the definitions and notation symbols that we use in presenting the allocation and scheduling algorithms. We assume that we have an
image processing application modeled by an IATG $G = (V, E, w, c)$, where $V$ is the set of $n = |V|$ tasks (vertices) and $E$ is the set of $e = |E|$ edges.

The weight of a task $t$, denoted $w(t, n(t))$, represents the computation and possible internal communication time of the task $t$ when executed on $n(t)$ processors. The task computation time is obtained either analytically using Formula 5.9, or through profiling, i.e., measuring the task execution time on different number of processors and tabulating the values obtained. The weight of an edge $(t, t')$ denoted $c(t, t')$ represents the external communication time between the two tasks. As for the computation time, the communication time is also obtained either analytically using Formula 5.14 or by profiling.

The graph width $W$ is the maximum number of tasks that are not connected through a path. This measure roughly indicates the amount of task parallelism that exists in an application. The higher $W$, the more task parallelism is present in the application.

A task is said to be ready if all its parent tasks have finished their execution. As mentioned before, we call a entry task a task with no input edges and a exit task a task with no output edges.

**Definition 8 (Entry Task, Exit Task, Top Level, Bottom Level)**: For $v \in V$, $PRED(v)$ denotes the set of all immediate predecessors of $v$, and $SUCC(v)$ the set of all its immediate successors.

$1$ $v \in V$ is an entry task if $PRED(v) = 0$. 
2 For \( t \in V \), the top level \( T_t(t) \) is the largest weight of a path from an entry task to \( t \), excluding the weight of \( t \). The length of a path is the sum of the costs of the tasks and edges belonging to that path. Recursively, this is written as

\[
T_t(t) = \max_{(t',t) \in E} \{T_t(t') + w(t,n(t')) + c(t', t)\}
\] (5.20)

3 For \( t \in V \), the bottom level \( T_b(t) \) is the largest weight of a path from \( t \) to an exit task, including the weight of \( t \). Recursively, this is written as

\[
T_b(t) = w(t,n(t)) + \max_{(t,t') \in E} \{T_b(t') + c(t,t')\}
\] (5.21)

A scheduled task \( t \) is associated with a number of processors \( n(t) \), the set of processors \( PROC(t) \subseteq P \) on which \( t \) is mapped (where \( P \) is the set of processors), a start time \( T_s(t) \), and a finish time \( T_f(t) \).

**Definition 9 (Processor Ready Time)** The processor ready time of a processor \( p \) on a partial schedule is defined as the finish time of the last task scheduled on that processor, where \( t \) are the tasks scheduled on that processor:

\[
T_r(p) = \max_{t \in V, p \in PROC(t)} T_f(t)
\] (5.22)

**Definition 10 (Processor becoming ready the earliest)** Given a partial schedule, we define the processor becoming ready the earliest (\( p_r \)) to be the processor with the minimum \( T_r \):

\[
T_r(p_r) = \min_{p \in P} T_r(p)
\] (5.23)

**Definition 11 (Last Message Arrival Time)** The last message arrival time of a ready task \( t \) is defined as:

\[
T_m(t) = \max_{(t',t) \in E} \{T_f(t') + c(t', t)\}
\] (5.24)

The priority \( \rho(t) \) of a task \( t \) depends on the scheduling algorithm. Usually, the last message arrival time, top level or bottom level are used as metrics.
5.5 Mapping and Scheduling Algorithms

```
LS()
BEGIN
    Compute the priorities of the entry tasks.
    WHILE NOT all tasks scheduled DO
        t ← ready task with the maximum ρ(t).
        // rho(t) is the priority of the task, and it may be the T_m(t) or the T_b(t).
        Schedule t on the first n(t) processors becoming idle.
        Compute the priorities ρ of the new become ready tasks.
    END WHILE
END
```

Figure 5.13: The List Scheduling Procedure

5.5.2 Pure Task Scheduling (TASK)

In pure task scheduling, we assign only one processor per task and for scheduling we use the classic list scheduling algorithm described in [40], see Figure 5.13. At each iteration, the task with the highest priority ρ(t) is scheduled. For computing the priority of a task, different algorithms can be used, as the Earliest Start Time (EST) or the bottom level, see formula 5.21. In this case, the time complexity of TASK is $O(E + N \log(N) + NP \log P)$ [88]. $N$ represents the total number of tasks, $E$ is the total number of edges in the IATG and $P$ is the number of processors.

5.5.3 Pure Data Scheduling (DATA)

In pure data scheduling, each task is assigned to be performed on the entire set of available processors. For this reason, all the IATG tasks are executed sequentially. Each of the image processing operators are applied via the suitable skeleton. In this case scheduling reduces to $O(1)$ time, since no decision has to be made.

5.5.4 Two Level Scheduling (TwoL)

In [87, 86] Rauber and Rünger study a restricted case of task graphs with a series-parallel (SP) topology. SP graphs consist of series and/or parallel node compositions, as the one shown in Figure 5.14. 7-8 is a serial node composition and 4-5 is a parallel node composition. The name of the algorithm, two level, comes from the observation that algorithms from scientific computing often exhibit two-level parallelism based on potential method parallelism and potential system parallelism. The two-level potential parallelism of algorithms is expressed in a specification consisting of an upper level hierarchy of multiprocessor tasks each of which has an internal structure of uniprocessor tasks.

Scheduling is performed hierarchically as follows. The tasks that form a series composition are assigned the same number of processors. The tasks that form a parallel composition are assigned equal numbers of processors from the set of available processors. The tasks are assigned in the order of their execution time on the processor
group becoming ready the earliest. Given a processor set, all the group sizes are tried, and the one with the best scheduling is selected.

TwoL starts with the higher-level node, which encapsulates the whole program, for which it assigns all the processors. It then proceeds hierarchically, scheduling the nodes according to their composition, down to the basic tasks.

The time complexity of TwoL is given by the scheduling complexities of the two compositions. Let $N$ be the total number of nodes. The series composition takes $O(N)$ time. The parallel composition takes $O(N \log N)$ time to sort the nodes, and $O(N \log K)$ to allocate the processors, assuming that there are $K$ processor groups, which for all processor group sizes is $O(NP \log P)$. There can be at most $V$ series of parallel compositions. As the sub-node allocations must be known for all values of $P$, there are $VP$ composition allocations to be computed. Considering also that there can be at most $V$ nodes forming a composition, the time complexity of scheduling all compositions is $O(V^2P^2 \log(P))$ [88].

```
TwoL ()
BEGIN
  np(START) = P
  FOR (k=1,...,c) {
    partition the processors in k subsets
    of size $p_k = p/k$
    sort $J_1, ..., J_c$ such that
    $T(J_1, p_k) \geq \cdots \geq T(J_c, p_k)$;
  } END FOR
END
```

Figure 5.15: The TwoL Algorithm
5.5.5 Two Step Allocation and Scheduling (TSAS)

In [85], the authors propose a two-step algorithm. The first step performs allocation based on a convex programming approach to find for each task of the graph the number of processors on which that task will be executed. In the next subparagraph we present a detailed description of the algorithm. Given an IATG and a $P$ processor system, the Two Step Allocation and Scheduling Algorithm (TSAS) consists of the following steps:

1. Allocate processors to the nodes of the IATG using the Convex Programming Allocation Algorithm (CPAA).
2. Schedule the allocated nodes using the Prioritized Scheduling Algorithm (PSA).

Convex Programming Allocation Algorithm (CPAA)

Based on the Formulas 5.15, 5.16 and 5.17, processors are allocated to tasks according to the results obtained by the following steps:

1. Obtain a real number solution for the following minimization problem

$$\phi = \min(\max(A, CP))$$  \hspace{1cm} (5.25)

subject to

$$1 \leq n(t) \leq P \quad \forall t = 1, n$$  \hspace{1cm} (5.26)

where, the quantities $A$ and $CP$ are the average area and the critical path for the IATG.

2. Set $n(t) = \lceil (n(t)) \rceil \quad \forall \ t = 1, N.$

3. Set $n(t) = \min(n(t), PB)$, where $PB$ is a function of $P$ and can be computed using Corrolary 3.

By obtaining an allocation (set of $n(i)$'s) that minimizes the lower bound $\phi$, the execution time for a given application can be minimized. However, the minimization process is carried out in a continuous domain, which results in $n(i)$ being real numbers rather than integers, thus necessitating the second step, to round-off the solution found in the first step. The reasons of formulating the problem in a continuous domain are

(i) The posynomial properties of the cost models make the minimization problem equivalent to a convex programming formulation [106], see Appendix A. This guarantees finding the optimal solution.

(ii) A convex programming formulation has been shown to be solvable in polynomial time [106], that means it finds an optimal solution in polynomial time.
Note that in step 1 of the CPAA algorithm the search step is not constrained in any form, tasks are allowed to use all the processors. This unconstrained minimization problem does not ensure "schedulability". By this term we mean that allowing a task to use any number of processors may result in the impossibility to ensure the concurrency, i.e., the existence of tasks that run concurrently. To improve the schedulability, a theoretical bound $PB$ is imposed in step 3 of the algorithm. Without going into further details we present the following Corollary which computes the optimum value for $PB$.

**Corollary 3** The optimum value of $PB$ to use for the TSAS algorithm given $P$ processors is the solution to the following minimization problem:

$$
\text{minimize} \left(1 + \frac{P}{P - PB + 1}\right) \cdot \left(\frac{2 \cdot P}{PB}\right) \quad \text{subject to} \quad 1 \leq PB \leq P \tag{5.27}
$$

The proof of the above Corollary can be found in [85].

**Prioritized Scheduling Algorithm (PSA)**

In the first step (CPAA) the number of processors assigned to each task of the graph was determined. In the second step of the algorithm (PSA) the tasks are scheduled on processors using a list scheduling algorithm [40]. The tasks are iteratively scheduled based on the Earliest Start Time $EST$ which is, in fact, the last message arrival time $T_m$, defined in Formula 5.24. Each task $t$ is scheduled on the first $n(t)$ processors that become available, where $n(t)$ is the number of processors allocated to $t$.

The steps of PSA are:

1. Place the **START** node of the IATG in a queue called the Ready queue and mark its Earliest Start Time ($EST_1$) as 0.

2. Pick a task (node) $t$ from the Ready queue that has the lowest value of the Earliest Start Time $EST_t$. Use the schedule built so far to check the time at which the processor requirement of the task can be met. This is called the Processor Satisfaction time $PST_t$ for the node. Schedule the node at $\max(EST_t, PST_t)$ and compute its finish time $FT_t$.

3. If the node scheduled in the previous step is **STOP**, terminate the scheduler with a finish time $FT_N$; otherwise, proceed to the next step.

4. Check all the successors of the node just scheduled to see if all the precedence constraints of any of them have been met, i.e., all their predecessors have been scheduled. Any successor node that meets this criteria is placed on the Ready queue with their Earliest Start time computed.

5. Repeat starting at Step 2.
PSA ()
BEGIN
Place the START node in the Ready queue and mark its EST\(_1\) = 0
WHILE Ready queue NOT EMPTY DO
\(t \leftarrow \) a task from the Ready queue with the lowest \(EST_i = T_m(t)\).
Check the time \(PST_i\) at which the processor requirements of the task can be met.
Schedule the node at \(\max(EST_i, PST_i)\) and compute its finish time \(FT_i = T_f(t)\).
IF the node scheduled in the previous step is STOP
terminate the scheduling with the finish time \(FT_N\)
ELSE
put all the successors of \(t\) which have the precedence constrained met
on the Ready queue, with the \(EST\) computed.
END WHILE
END

Figure 5.16: The PSA Algorithm

Note that picking the node with the lowest Earliest Start time in Step 2 of the algorithm creates a priority among nodes, hence the name for the algorithm. In Figure 5.16 we present the pseudo-code of the algorithm.

The time complexity of TSAS consists of the \(O(N^{2.5})\) complexity of the task allocation step [106], and the \(O((E+N \log(N)+NP \log(P))\) complexity of the list scheduling step. The \(P \log P\) term arises in Step 2 of the PSA where the processor finish times have to be sorted to determine a suitable set of processors for the next step. Consequently, the TSAS time complexity is \(O(N^{2.5} + NP \log(P))\).

5.5.6 The Critical Path Reduction (CPR) Algorithm

We consider now the Critical Path Reduction (CPR) algorithm that we have developed (as a joint work with A. Radulescu), see [88, 124]. Compared to TSAS, Critical Path Reduction (CPR) is a one step allocation and scheduling algorithm. This algorithm views allocation and scheduling as being done simultaneously in one single step and provides a solution to it. CPR (see Figure 5.17) is a one step greedy iterative algorithm that at each iteration attempts to improve the task allocation found in the previous step [124]. It starts with one processor allocated to each task of the graph. The key idea is that the schedule is improved if more parallelism can be exploited by adding idle processors to the allocation of tasks. As more processors are allocated to a task, the task execution time is shortened and as a consequence the whole program execution time will typically be shortened.

CPR iteratively increments the number of processors allocated to a particular task. To select the best candidate to allocate more processors, we observe that, as long as there are idle processors, the program execution time is given by the critical path of its task graph. Allocating one more processor to a task belonging to the critical path may have the consequence of reducing that task's execution time and, hence, reducing the program execution time. We compute the top level and bottom level for each task of the graph. We select the task \(t\) with the highest \(T_t(t) + T_b(t)\) where \(n_p(t) < P\).
We allocate one more processor to the selected task, and test whether the program schedule (performance) decreases. If it does, we commit this allocation. If not, we select the task with the next highest \( T_i(t) + T_b(t) \) and \( n_p(t) < P \). At each iteration, the critical path of the graph may change as a consequence of the new allocation. Therefore, the top and bottom levels of the tasks are recomputed to select the next task. CPR stops when there is no task where increasing the allocated processor number yields a performance gain.

\[
\begin{align*}
\text{CPR}() \\
\text{BEGIN} \\
\text{FORALL } t \in V \text{ DO} \\
\quad n(t) \leftarrow 1; \text{ denote this allocation with } \sigma_{cr} \\
\text{END FORALL} \\
T \leftarrow MS(\sigma_{cr}, P) \text{ of the current allocation using LS();} \\
\text{REPEAT} \\
\quad q_t \leftarrow \text{ all tasks } t \text{ such that } n(t) < P; \\
\text{REPEAT} \\
\quad t \leftarrow t \in q_t \text{ with max } \{T_i(t) + T_b(t)\}; \\
\quad n(t) \leftarrow n(t) + 1; \text{ current allocation is now } \sigma' \\
\quad T' \leftarrow MS(\sigma', P) \text{ of the current allocation } \sigma' \text{ computed using LS();} \\
\text{IF } T' < T \text{ THEN} \\
\quad T \leftarrow T'; \\
\text{ELSE} \\
\quad n(t) \leftarrow n(t) - 1; \text{ restore previous allocation } \sigma_{cr} \\
\quad \text{ Remove } t \text{ from } q_t; \\
\text{END IF} \\
\text{UNTIL } T \text{ was modified, or } q_t \text{ is empty;} \\
\text{UNTIL } T \text{ remains not modified; } \\
\text{END}
\end{align*}
\]

Figure 5.17: The CPR Algorithm

CPR consists of two nested loops. In the outer loop, there can be at most \( NP \) iterations, since there are \( N \) tasks and for each task at most \( P \) processors can be considered. In the inner loop the \( T_i \) and \( T_b \) values are computed for each task in \( O(E+N) \) time, and the tasks are sorted in \( O(N \log N) \). Then, the algorithm increments \( n_p(t) \), until the program execution is improved. In the worst case, all \( N \) tasks are considered. At each attempt, the \( LS() \) procedure is called to improve the task allocation, which has a \( O(E + N(\log(N) + P \log(P))) \) time complexity. As a consequence, the inner loop takes \( O(EN + N^2(\log(N)) + P \log(P)) \), and the total CPR complexity is \( O(EN^2P + N^3P(\log(N) + P \log(P))) \).

### 5.5.7 A measure of performance

To be able to make a fair comparison of the scheduling algorithms presented, we define a measure called the relative performance of the algorithm as the ratio between the performance of that scheduling algorithm and the performance of a base scheduling algorithm.
5.5 Mapping and Scheduling Algorithms

Definition 12 (Relative performance of a scheduling algorithm) The relative performance $RP_{alg_c}^{alg_b}$ of the scheduling algorithm $alg_c$ with respect to the base scheduling algorithm $alg_b$ is defined as

$$ RP_{alg_c}^{alg_b} = \frac{MS(alg_c, P)}{MS(alg_b, P)} $$ (5.28)

The lower the ratio (the minimum value being 1 for comparing with the base scheduling algorithm $alg_b$), the better the scheduling algorithm $alg_c$ performs with respect to the scheduling algorithm $alg_b$.

5.5.8 Order analysis of the parallelization approaches

A program in our model consists of a sequence of image processing tasks $t_1, t_2, t_3, ..., t_N$, where each task receives data from its predecessor, process it, and sends the output to its successor; the first task reads the external input data and the last task generates the final output data sets. Each task can be executed on one or more processors. The execution time of a task is a function of the number of processors and is denoted by $w(t, n(t))$, see Formula 5.9. For simplicity, we will replace $w(t, n(t))$ by $w_t$. If the program is executed in a data parallel way, then we have a chain of tasks with the following execution times, see 5.29.

$$ w_1, w_2, w_3, ..., w_N $$ (5.29)

A task can be executed on one or more processors. If the task is executed on more processors, then the execution time includes also the internal communication time needed to scatter/gather the data from the master processors to the other processors assigned to that task. A pair $(i, j)$ of adjacent tasks may need to communicate data (images). If this is the case, then the external communication time $c(i, j)$ (see 5.14) for the whole chain of tasks becomes as in 5.30.

$$ c(1, 2), c(2, 3), c(3, 4), ..., c(N - 1, N) $$ (5.30)

Intuitively, the data and task parallel approach has a better performance than the data parallel approach when the cost of the Critical Path CP(DATA+TASK), which consists of the execution and communication time of the tasks associated to it on a number of processors less than or equal to the total number of processors, is less than the Critical Path for the data parallel approach CP(DATA), which, in our case, contains all tasks of the IATG executed on all available processors, see Formulas 5.31 and 5.32.

$$ T_{exec}(DATA + TASK) < T_{exec}(DATA) $$ (5.31)
CHAPTER 5. MIXED DATA AND TASK PARALLELISM IN IMAGE PROCESSING

\[ CP(DATA + TASK) < CP(DATA) \]  \hspace{1cm} (5.32)

If \( N \) is the total number of program tasks, and the Critical Path contains the tasks numbered from 1 to \( k \), where \( k \geq 2 \) and at most \( k - 1 \) external communication operations, and \( P \) is the total number of processors, then inequality 5.32 becomes 5.33, where each task belonging to the Critical Path is executed, in the data parallel case, on all \( P \) processors, and in the data and task parallel case, on \( p_i \) processors:

\[ \sum_{i=1}^{k} (\alpha_i + \frac{1 - \alpha_i}{p_i}) \tau_i + \sum_{i=1}^{k-1} c(i, i+1) < \sum_{i=1}^{N} (\alpha_i + \frac{1 - \alpha_i}{P}) \tau_i \]  \hspace{1cm} (5.33)

If we assume that the size of the data involved in all external communication operations is the same and we replace \( c(i, i+1) \) by Formula 5.14, we obtain inequality 5.34, where \( p_i \leq P \):

\[ (k - 1) \cdot (t_s + L \cdot t_b) < \sum_{i=1}^{N} (\alpha_i + \frac{1 - \alpha_i}{P}) \tau_i - \sum_{i=1}^{k} (\alpha_i + \frac{1 - \alpha_i}{p_i}) \tau_i \]  \hspace{1cm} (5.34)

Let us denote by \( T_{\text{comm}} = t_s + L \cdot t_b \) the point-to-point communication time needed to send the data (image) between two processors. Then, we obtain Formula 5.35:

\[ T_{\text{comm}} < \frac{\sum_{i=1}^{N} (\alpha_i + \frac{1 - \alpha_i}{P}) \tau_i - \sum_{i=1}^{k} (\alpha_i + \frac{1 - \alpha_i}{p_i}) \tau_i}{k - 1} \]  \hspace{1cm} (5.35)

In the case of pure task parallel approach, the number of processors allocated to each task is one, \( p_i = 1 \), and the condition that should be satisfied in order to have a better data and task parallel performance than pure task parallel performance becomes 5.36:

\[ T_{\text{comm}} < \frac{\sum_{i=1}^{N} (\alpha_i + \frac{1 - \alpha_i}{P}) \tau_i - \sum_{i=1}^{k} \tau_i}{k - 1} \]  \hspace{1cm} (5.36)

Note that \( T_{\text{comm}} \) is the point-to-point communication time of data between two processors. The time to scatter and gather the data (images) from the master processor to the other processors allocated to that task is included in the terms \( \alpha_i \). Consequently, is the overhead for scattering and gathering is high, it becomes relatively (with respect to \( T_{\text{comm}} \) and the pure execution time) earlier beneficial to perform both data and task parallel processing than when this overhead is low.

The scheduling algorithms DATA, TASK and the different approaches of the combined data and task parallel (TSAS, TwoL, CPR) have different complexities which are analysed in the previous subsections.
5.6 Experiments

5.6.1 Multi-baseline stereo vision

To evaluate the benefits of the proposed data parallel framework based on skeletons and also of the task parallel framework based on the IATG, we compare the speed-ups obtained by applying only data parallelism to the application, with the speed-ups obtained with both data and task parallelism.

The multi-baseline stereo vision application uses an algorithm developed by Okutomi and Kanade [72] and described by Webb and al. [21, 114], that gives greater accuracy in depth through the use of more than two cameras. It can be observed that the computation of the difference images requires point operators, while the computation of the error images requires neighborhood operators. The computation of the disparity image requires also a point operator. The graph width for this task graph is 16, which represents the maximum available parallelism in this application, see Figure 5.3. All the 16 tasks which compute the difference and error images can be executed in parallel, as it can be deducted from the pseudo-code of the application, presented in Figure 5.2, at the beginning of this chapter.

In Table 5.3 we present the code of the application using mixed data and task parallelism. Data parallelism is applied via skeletons and for applying task parallelism we have used different scheduling algorithms (data, task, TwoL, TSAS and CPR).

Table 5.3: Mb_Stereo_data_task_parallel

```c
int main(int argc, char **argv)
{

    num_nodes=ImageInitPar(argc,argv); // initialize parallel stuff

    IF (My_Processor()==0) //
        t1=MPI_Wtime();
    win=CreateWindow(13,13);
    //read_and_extract_proc_mapping(proc_ban_new_16);
    read_and_extract_proc_mapping(proc_ban_new_4);
    //read_and_extract_proc_mapping(stereo_ban_8_1);
    //read_proc_mapping(proc_8);
    n_imag[My_Processor()]=0; // initialize number of created images on each processor
    enqueue_task(CREATE,Create); // put the tasks in the prio queues for each processor
    enqueue_task(S1,Image_shift); // image shift operator
    enqueue_task(S2,Image_shift);
    enqueue_task(S3,Image_shift);
    enqueue_task(S4,Image_shift);
    enqueue_task(S5,Image_shift);
    enqueue_task(S6,Image_shift);
    enqueue_task(S7,Image_shift);

    continued on next page
```
enqueue_task(S8, Image_shift);
enqueue_task(S9, Image_shift);
enqueue_task(S10, Image_shift);
enqueue_task(S11, Image_shift);
enqueue_task(S12, Image_shift);
enqueue_task(S13, Image_shift);
enqueue_task(S14, Image_shift);
enqueue_task(S15, Image_shift);
enqueue_task(S16, Image_shift);

enqueue_task(D1, Image_diff); // square difference operator
enqueue_task(D2, Image_diff);
enqueue_task(D3, Image_diff);
enqueue_task(D4, Image_diff);
enqueue_task(D5, Image_diff);
enqueue_task(D6, Image_diff);
enqueue_task(D7, Image_diff);
enqueue_task(D8, Image_diff);
enqueue_task(D9, Image_diff);
enqueue_task(D10, Image_diff);
enqueue_task(D11, Image_diff);
enqueue_task(D12, Image_diff);
enqueue_task(D13, Image_diff);
enqueue_task(D14, Image_diff);
enqueue_task(D15, Image_diff);
enqueue_task(D16, Image_diff);

enqueue_task(E1, Image_err);
enqueue_task(E2, Image_err);
enqueue_task(E3, Image_err);
enqueue_task(E4, Image_err);
enqueue_task(E5, Image_err);
enqueue_task(E6, Image_err);
enqueue_task(E7, Image_err);
enqueue_task(E8, Image_err);
enqueue_task(E9, Image_err);
enqueue_task(E10, Image_err);
enqueue_task(E11, Image_err);
enqueue_task(E12, Image_err);
enqueue_task(E13, Image_err);
enqueue_task(E14, Image_err);
enqueue_task(E15, Image_err);
enqueue_task(E16, Image_err);
enqueuel task(DISP, Disparity); // image disparity

execute_parallel(); // activate mixed data and task parallel execution

IF (My_Processor() == 0) {
    t2 = MPI_Wtime();
    total_time = t2 - t1;
    printf(time="%lf %lf %lf \n", total_time, t1, t2);
}

ImageExitPar(); // stops parallel stuff

RETURN 0;

}

void Create(task_t t)
{

int nimag;

v_imag[My_Processor()][nimag] = CreateImage(256, 256, red", PAR_INT);
InitImage(v_imag[My_Processor()][nimag], 0);
n_imag[My_Processor()]++;
nimag++;

v_imag[My_Processor()][nimag] = CreateImage(256, 256, m1, PAR_INT);
InitImage(v_imag[My_Processor()][nimag], 0);
n_imag[My_Processor()]++;
nimag++;

v_imag[My_Processor()][nimag] = CreateImage(256, 256, m2, PAR_INT);
InitImage(v_imag[My_Processor()][nimag], 0);
n_imag[My_Processor()]++;
nimag++;

v_imag[My_Processor()][nimag] = CreateImage(256, 256, im, PAR_INT);
InitImage(v_imag[My_Processor()][nimag], 0);
n_imag[My_Processor()]++;
nimag++;
}

RETURN;

}

void Image_shift(task_t t)
{
ImagePointDist_10(t, m1, m1_s, nr_proc[t], v[t], Image_shift);

continued from previous page
void Image_diff(task_t t)
{
    ImagePointDist_11O(t,m2.m2.s.nr.proc[t],v[t],Image.shift);
    RETURN ;
}

void Image.err(task_t t)
{
    ImageWindowDist_11O(t,im,ref,m1.s.m2.s.nr.proc[t],v[t],Image2DifSqr);
    return ;
}

void Disparity(task_t t)
{
    FOR (i = 0 ; i < 16 ; i++)
        ImagePointDist_11O(t,min.err,i,nr.proc[t],v[t],Min);
}

Table 5.4: Performance of multi-baseline stereo vision for different scheduling algorithms

<table>
<thead>
<tr>
<th>Processors</th>
<th>DATA</th>
<th>TASK</th>
<th>TSAS</th>
<th>TwoL</th>
<th>CPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>103.75</td>
<td>103.75</td>
<td>103.75</td>
<td>103.75</td>
<td>103.75</td>
</tr>
<tr>
<td>2</td>
<td>62.13</td>
<td>52.86</td>
<td>52.66</td>
<td>52.85</td>
<td>52.98</td>
</tr>
<tr>
<td>4</td>
<td>33.81</td>
<td>27.03</td>
<td>27.30</td>
<td>27.15</td>
<td>27.04</td>
</tr>
<tr>
<td>8</td>
<td>17.61</td>
<td>13.81</td>
<td>14.23</td>
<td>13.80</td>
<td>13.75</td>
</tr>
<tr>
<td>16</td>
<td>11.52</td>
<td>7.70</td>
<td>7.88</td>
<td>7.68</td>
<td>7.68</td>
</tr>
<tr>
<td>32</td>
<td>10.24</td>
<td>7.67</td>
<td>3.99</td>
<td>3.61</td>
<td>3.56</td>
</tr>
</tbody>
</table>

Table 5.5: Speedup of multi-baseline stereo vision for different scheduling algorithms

<table>
<thead>
<tr>
<th>Processors</th>
<th>DATA</th>
<th>TASK</th>
<th>TSAS</th>
<th>TwoL</th>
<th>CPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.66</td>
<td>1.96</td>
<td>1.97</td>
<td>1.96</td>
<td>1.99</td>
</tr>
<tr>
<td>4</td>
<td>3.00</td>
<td>3.83</td>
<td>3.85</td>
<td>3.82</td>
<td>3.83</td>
</tr>
<tr>
<td>8</td>
<td>6.18</td>
<td>7.51</td>
<td>7.29</td>
<td>7.51</td>
<td>7.54</td>
</tr>
<tr>
<td>16</td>
<td>9.05</td>
<td>13.47</td>
<td>13.16</td>
<td>13.50</td>
<td>13.50</td>
</tr>
<tr>
<td>32</td>
<td>10.13</td>
<td>13.52</td>
<td>26.00</td>
<td>28.73</td>
<td>29.14</td>
</tr>
</tbody>
</table>
5.6 Experiments

We have applied the scheduling algorithms presented in the previous section (DATA, TASK, TwoL, TSAS and CPR). The experiments have been performed on a distributed memory system which consists of a cluster of Pentium Pro/200Mhz PCs with 64MB RAM running Linux [17], and connected through Myrinet in a 3D-mesh topology, with dimension order routing. In the task parallel framework we use a special mechanism to register the images on the processors where they are first created. Moreover, each skeleton has associated the task number to which it corresponds. We use 1, 2, 4, 8, 16, 32 and 64 processing nodes in the pool. Three artificial reference images of sizes $256 \times 256$, $512 \times 512$ and $1024 \times 1024$ are used. The code is written in C and uses the MPI message passing library.

The multi-baseline stereo vision algorithm is an example of a regular well balanced application in which task parallelism can be applied without the need of an allocator of scheduler, when the number of processors is a multiple of the number of tasks. We have studied the performance of the application with different scheduling algorithms. As the TSAS's convex programming algorithm [85] for determining the number of processors for each task was not available for solving the previous min-max problem, we have used the nonlinear solver based on SNOPT [34] available on the Internet [64]. In Figure 5.18 we show the speed-ups obtained for the pure data parallel scheduling, for different image sizes. Figure 5.19 shows the speed-up of the same application using the TSAS scheduling algorithm, also for different image sizes. Tables 5.4 and 5.5 show the performance and the speedup of the presented scheduling algorithms on an image of size $1024 \times 1024$. The speedups were calculated with respect to the performance of a optimized sequential version of the application executed on a single processor, implemented without additional communication overhead. The performance loss of TSAS scheduling algorithm is mainly caused by the allocation step (the first step of the algorithm) which determines the number of processors that should be assigned to each task of the IATG. As the computation of the squared difference operators and the error images are identical tasks for each disparity value, the best performance would be obtained by mapping an equal number of processors to them, which is actually the behavior of the CPR and TwoL scheduling algorithms. However, TSAS does not map an equal number of processors to identical tasks, which results in a slightly lower performance of the algorithm. Also, in our measurements we have used a non-linear solver for dealing with the minimization problem, since the original algorithm was not available. We can observe that the speed-ups for the data-parallel approach become quickly saturated while the speed-ups for the data and task parallel approach perform very good. In fact, we have pure task parallelism up to 16 processors and data and task parallelism from 16 on. So, the pure task parallel speed-ups will become flattened from 16 processors on because at this type of application is better to first apply task parallelism and then to add data parallelism. Using both data and task parallelism is clearly more efficient than using only data parallelism for this image processing application.

Figures 5.20 and 5.21 show the relative performance (according to Equation 5.28) of the pure data, pure task, TwoL and TSAS scheduling algorithms to the CPR scheduling algorithm, for a $1024 \times 1024$ image size. The lower the value (which can be 1 at the minimum) the better the scheduling algorithm, in comparison to CPR.
Let us apply the order analysis presented in the previous section. For simplicity, let us suppose that we have a total number of processors $P$ which is a multiple of 16 (the width of the IATG) and that each task is executed, in the data and task parallel approach, on $P/16$ processors.

Equation 5.36 becomes for this specific application as in 5.37:
Figure 5.21: Relative performance at execution, with different scheduling alg. for multi-baseline stereo vision

\[ T_{\text{comm}} < \frac{\sum_{i=1}^{N}(\alpha_i + \frac{1-\alpha_i}{P})\tau_i - \sum_{i=1}^{k}(\alpha_i + \frac{1-\alpha_i}{P/16})\tau_i}{k - 1} \]  

(5.37)

Because, for this application, the total number of tasks is a multiple of the number of tasks belonging to the critical path, \(N = 16 \cdot k\), and task \(t_i = t_{i+4} = t_{i+8} = \ldots = t_{i+4 \times j}\), where \(j = 1...15\), so on each path tasks are the same, we obtain Formula 5.38:

\[ T_{\text{comm}} < \frac{16 \sum_{i=1}^{k}(\alpha_i + \frac{1-\alpha_i}{P})\tau_i - \sum_{i=1}^{k}(\alpha_i + 16\frac{1-\alpha_i}{P})\tau_i}{k - 1} \]  

(5.38)

After replacing \(k\) with its value 4 in the case of this application, and some easy simplification we obtain Formula 5.39:

\[ T_{\text{comm}} < \frac{15 \sum_{i=1}^{4} \alpha_i \tau_i}{3} \]  

(5.39)

The terms \(\alpha_i \tau_i\) can be replaced by \(b_i\) from Formula 5.12 and Equation 5.39 becomes:

\[ T_{\text{comm}} < 5(b_1 + b_2 + b_3 + b_4) \]  

(5.40)
The interpretation of Equation 5.40 is straightforward now. If the communication time of sending an image between two processors is less then five times the sum of b terms of the four tasks belonging to the Critical Path then it will be more efficient to use combined data and task parallelism than to use simple data parallelism, in case of the multi-baseline stereo vision application.

5.6.2 Optical flow

Image analysis may also study the dynamic aspects of a scene, as for example the movements of objects and camera. These can only be detected if we look at a time series of images (video sequences). Movement of an object relative to a camera usually causes movement of image patterns in the image plane. In this chapter, we present the theory, implementation, and experimental results for an algorithm which determines the displacement vector field in image patterns. The calculation of displacement vector fields is also called a calculation of optical flow. The image patterns are composed of a multitude of arbitrary small pixels, which are mobile within the image plane. In this case each pixel contains a gray value which is constant in time. Movement of gray value patterns in the image plane are as such considered as a flow of pixels in the image plane.

To evaluate the benefits of the proposed combined data and task parallel framework we compare the speed-ups obtained by applying only data parallelism, with the speed-ups obtained with both data and task parallelism for the computation of optical flow. The algorithm is a differential method proposed by Horn and Schunck in [45]. It takes a pair of gray scale images from an image sequence and determines the displacement of the gray value structures. The result of these calculations is known as a displacement vector or a displacement vector field. The starting point for the derivation is an image pair. We say that the first image was taken at time \( t \), the second at time \( t + dt \), where \( dt \) is infinitesimally small. Each pixel in the first image can also be found in the second image, but at a position which may have shifted in the image plane by displacement vector \((dx, dt)^T\). These facts can be formally expressed as in Formula 5.41

\[
E(x, y, t) = E(x + dx, y + dy, t + dt) \tag{5.41}
\]

\( E(x, y, t) \) is the function of the gray value in time and in the image plane. The displaced gray value image can be approximated by a first order Taylor series, as in Formula 5.42.

\[
E(x + dx, y + dy, t + dt) = E(x, y, t) + dx \frac{\delta E}{\delta x} + dy \frac{\delta E}{\delta y} + dt \frac{\delta E}{\delta t} + \epsilon \tag{5.42}
\]

When the higher order terms \( (\epsilon) \) are ignored, and substitute (5.42) into (5.41) we get the total differential of the gray value, see Formula 5.43.

\[
dx \frac{\delta E}{\delta x} + dy \frac{\delta E}{\delta y} + dt \frac{\delta E}{\delta t} = 0 \tag{5.43}
\]
5.6 Experiments

After a division by $dt$ we obtain the continuity equation of optical flow, as in Formula 5.44.

$$\frac{dx}{dt} \cdot \frac{\delta E}{\delta x} + \frac{dy}{dt} \cdot \frac{\delta E}{\delta y} + \frac{\delta E}{\delta t} = 0 \quad (5.44)$$

If the observed time interval $dt$ tends to 0, the relevance of Equation 5.44 becomes apparent. The quotients $\frac{dx}{dt}$ and $\frac{dy}{dt}$ then become components of the current velocity vector $(u, v)^T$ of a pixel in the image plane.

$$\lim_{dt \to 0} \left( \frac{dx}{dt} \right) = u \quad \lim_{dt \to 0} \left( \frac{dy}{dt} \right) = v \quad (5.45)$$

When the time interval between two images of a sequence is taken as unit-time, then $(u, v)^T$ can be interpreted directly as the desired displacement vector. The partial derivation of the gray value image with respect to $x$, $y$ and time can be calculated approximately from the gray scale image via convolution operators and difference calculations $(E_x, E_y, E_t)$. The equation to determine the components $u$ and $v$ of the desired displacement vector for each pixel $(x, y)$ as such, is an equation with two unknowns (Formula 5.46). To be able to evaluate the equation, further conditions for $u$ and $v$ need to be found.

$$E_x \cdot u + E_y \cdot v + E_t = 0 \quad (5.46)$$

To determine a unique solution to Equation 5.45, another restricting condition in the form of an equation is necessary for the variables $u$ and $v$. Horn and Schunck [45] introduce a smoothness constraint, a condition for $u$ and $v$ which requires the largest possible smoothness for the calculated displacement vector field. The introduction of the smoothness condition is justified by the assumption that all investigated image objects posses closed, smooth boundaries. In that case neighboring displacement vectors differ only minimally. The smoother the displacement vector field, the smaller are the terms $T_1$ and $T_2$ in Equation 5.47.

$$T_1 = \left( \frac{\delta u}{\delta x} \right)^2 + \left( \frac{\delta u}{\delta y} \right)^2 \quad T_2 = \left( \frac{\delta v}{\delta x} \right)^2 + \left( \frac{\delta v}{\delta y} \right)^2 \quad (5.47)$$

We can define an error function $\epsilon^2$, which punishes those results of the displacement vector calculation which do not satisfy the smoothness constraint, i.e. Equation 5.48.

$$\epsilon^2 = \left( \frac{\delta u}{\delta x} \right)^2 + \left( \frac{\delta u}{\delta y} \right)^2 + \left( \frac{\delta v}{\delta x} \right)^2 + \left( \frac{\delta v}{\delta y} \right)^2 \quad (5.48)$$
From Equation 5.46 another penalty function can be constructed. The error $\epsilon_b$ in Formula 5.49 shows to what extent the solution found satisfies this equation.

$$\epsilon_b = E_x \cdot u + E_y \cdot v + E_t$$  \hspace{1cm} (5.49)

We can now group both errors into a global error function $F$, presented in Formula 5.50. The factor $\alpha$ determines the influence of the smoothness condition on the result.

$$F = \epsilon_b^2 + \alpha^2 \cdot \epsilon_c^2 = (E_x \cdot u + E_y \cdot v + E_t)^2 + \alpha^2(u_x^2 + u_y^2 + v_x^2 + v_y^2)$$  \hspace{1cm} (5.50)

Now the calculation of the displacement vector field no longer consists of the explicit solution to a system of equations for $u$ and $v$, but in minimizing the global error $\epsilon^2$, which can be obtained over the entire image plane by integration of the local errors. The minimization problem can be solved via a variation calculation. It is assumed that the integral in Equation 5.51 is minimal when for each position $(x, y)$ in the image plane the displacement vector $(u, v)^T$ is zero. From this condition we can derive two Euler Equations 5.52 and 5.53 which describe the dependency of the error on a variation of the vector components.

$$\epsilon^2 = \int \int (\epsilon_b^2 + \alpha^2 \cdot \epsilon_c^2) dx dy = \int \int F dx dy$$  \hspace{1cm} (5.51)

$$2(E_x^2 u + E_x E_y v + E_x E_t) - 2\alpha^2 u_{xx} - 2\alpha^2 u_{yy} = 0$$  \hspace{1cm} (5.52)

$$2(E_y^2 v + E_x E_y u + E_y E_t) - 2\alpha^2 v_{xx} - 2\alpha^2 v_{yy} = 0$$  \hspace{1cm} (5.53)

The sums of the second derivatives correspond to the Laplace operator, see Equation 5.54. When substituted into Equations 5.52 and 5.53 we obtain Equations 5.55 and 5.56.

$$u_{xx} + u_{yy} = \nabla^2 u \hspace{1cm} v_{xx} + v_{yy} = \nabla^2 v$$  \hspace{1cm} (5.54)

$$E_x^2 u + E_x E_y v + E_x E_t - \alpha^2 \nabla^2 u = 0$$  \hspace{1cm} (5.55)

$$E_y^2 v + E_x E_y u + E_y E_t - \alpha^2 \nabla^2 v = 0$$  \hspace{1cm} (5.56)
The Laplace operator can be approximated via a convolution mask presented in Figure 5.22. For each pixel we determine the average value of the eight neighbors of the pixel, according to the value of the corresponding weights. In the following, these average values will be denoted by \( \bar{u} \) and \( \bar{v} \).

Hence the Laplace operator can be calculated from the difference between the value of the centre pixel and the average values of the neighborhood \((\bar{u}, \bar{v})\): \( \nabla^2 u \approx \bar{u} - u, \ \nabla^2 v \approx \bar{v} - v \). If we substitute these in Equations 5.55 and 5.56 we obtain Equations 5.57 and 5.58.

\[
(\alpha^2 + E_x^2)u + E_x E_y v = \alpha^2 \bar{u} - E_x E_t \quad (5.57)
\]

\[
E_x E_y u + (\alpha^2 + E_x^2)v = \alpha^2 \bar{v} - E_y E_t \quad (5.58)
\]

By solving for \( u \) and \( v \) the system of equations is brought into a form which can be solved by the Gauss-Seidel Iteration method:

\[
u^{(n+1)} = \frac{\alpha^2 \bar{u}^{(n)} - E_x E_t - E_x E_y \bar{v}^{(n)}}{\alpha^2 + E_x^2} \quad (5.59)
\]

\[
v^{(n+1)} = \frac{\alpha^2 \bar{v}^{(n)} - E_x E_t - E_y E_x \bar{u}^{(n)}}{\alpha^2 + E_y^2} \quad (5.60)
\]

The values in the exponent brackets denote the specific iteration. The equations can be rewritten as in 5.61 and 5.62, to reduce the amount of computation. Hence, the optical flow for a pair of images is calculated via the Formulas 5.61 and 5.62.

\[
u^{n+1} = \bar{u}^n - \frac{E_x (E_x \bar{u}^n + E_y \bar{v}^n + E_t)}{\alpha^2 + E_x^2 + E_y^2} \quad (5.61)
\]
\[ v^{n+1} = \bar{v} - \frac{E_y(E_x \bar{u}^n + E_y \bar{v}^n + E_t)}{\alpha^2 + E_x^2 + E_y^2} \]  \hfill (5.62)

where:

- \( u^{n+1}, v^{n+1}, u^n, v^n \) represents the displacement vectors at the iterations \( n+1 \) and \( n \)
- \( \bar{u}, \bar{v} \) represents the average values which can be calculated via the convolution mask presented in Figure 5.22.
- \( E_x, E_y, E_t \) are the first order spatial and temporal derivatives
- \( \alpha \) is a smoothness constraint

As a result of the iteration, the function returns for each pixel an associated displacement vector \((u, v)\), consisting of a component for the x- and y- direction. A typical value for parameter \( \alpha \) is 50. For larger values of \( \alpha \) the influence of the smoothness condition on the calculation of the displacement vectors increases. Sensible values of the number of iterations are in the range of 20 to 300.

**OF_pseudocode**()  
BEGIN  
Task t0: Input im1,im2, alpha,u(0),v(0)  
Task t1: Compute Ex  
Task t2: Compute Ey  
Task t3: Compute Et  
Task t4: Calculate b=alpha**2+Ex**2+Ey**2  
FOR iterCnt=0,ITERS  
    Task t5: Compute u(n)  
    Task t6: Compute v(n)  
    Task t7: Compute a=Ex*u(n)+Ey*v(n)+Et  
    Task t8: Compute u(n+1)=u(n)-(Ex*a)/b  
    Task t9: Compute v(n+1)=v(n)-(Ey*a)/b  
END FOR  
Task t10: Output u(n+1), v(n+1)  
END

Figure 5.23: Pseudocode of the optical flow algorithm according to Horn and Schunck

It can be seen that tasks t1, t2, t3 can be executed in parallel. The same holds for the tasks t5, t6, and t8, t9. The parallel execution of the tasks t5, t6 and t8, t9 is valid for all the iterations. The pseudo code of this application is given in Figure 5.23 and the IATG is presented in Figure 5.24. Figure 5.25 shows the speedup obtained for this application on the same distributed memory system, as for the stereo vision application. Tables 5.7 and 5.8 present the performance and speedup of optical
flow for different scheduling algorithms. The algorithm uses a pair of images of size 1024 × 1024 and the scheduling algorithm used in this case was TSAS. Almost the same results were obtained by the CPR scheduling algorithm. TwoL could not be applied in this case, because the application graph is not of a serial-parallel type. The results are compared with the ones obtained with pure data scheduling. Although for this application the width of the IATG is not larger then three we can still see an improvement in the performance, by using combined data and task parallelism. Figure 5.26 shows the relative performance of DATA, TASK and TSAS with respect to CPR.

![Diagram of optical flow computation]

Figure 5.24: IATG of the optical flow computation, according to Horn & Schunck

Table 5.6: Optical_flow_data_task_parallel

```c
//data+task parallel optical flow calculation (algorithm by Horn and Schunck)

int main(int argc,char **argv)
{
    double t1,total_time,t2;
    int iterCnt;

    num_nodes=ImageInitPar(argc,argv); // initialize parallel stuff

    if(My_Processor()==0)
    t1=MPI_Wtime();

    read_and_extract_proc_mapping(optical_flow_cpr_A); // CPR schedule for 4 processors
```

Continued on next page
n_imag[My_Processor()]=0; // initialize number of created images on each processor

enqueue_task(CREATE,Create); // put the tasks in the priority queues for each processor

enqueue_task(EX,gv.dx);
enqueue_task(EY,gv.dy);
enqueue_task(ET,gv.dt);
enqueue_task(B,b_part);

for(iterCnt=0;i<ITERS;i++) {
    enqueue_task(DX_ENVIR,u_n);
    enqueue_task(DY_ENVIR,v_n);
    enqueue_task(A,a_part);
    enqueue_task(FLOW_X,u_n_1);
    enqueue_task(FLOW_Y,v_n_1);
}

enqueue_task(OF,OpticalFlow);

execute_parallel(); // activate mixed data and task parallel execution

if(My_Processor()==0) {
    t2=MPI_Wtime();
    total_time=t2-t1;
    printf("time=%lf %lf %lf",total_time,t1,t2); }

ImageExitPar(); // stops parallel stuff

return 0;

} // Create - creates and initializes the images needed in the application

void Create(task.t t)
{
    int nimag;

    if(My_Processor()==0) {
        nimag=n_imag[My_Processor()];
        v_imag[My_Processor()][nimag]=CreateImage(256,256,im1,PAR_INT);
    }
continued from previous page

ReadImageFromFile(v._imag[My_Processor()][nimag], optical_1);
  n_imag[My_Processor()]++;
nimag++;

  v._imag[My_Processor()][nimag]=CreateImage(256,256,im2,PAR_INT);
  ReadImageFromFile(v._imag[My_Processor()][nimag], optical_2);
  n_imag[My_Processor()]++;
nimag++;

  alpha2=50;

  v._imag[My_Processor()][nimag]=CreateImage(256,256,flow_x,PAR_INT);
  InitImage(v._imag[My_Processor()][nimag],0);
  n_imag[My_Processor()]++;
nimag++;

  v._imag[My_Processor()][nimag]=CreateImage(256,256,flow_y,PAR_INT);
  InitImage(v._imag[My_Processor()][nimag],0);
  n_imag[My_Processor()]++;
nimag++;

}

return;

//computes spatial_gradient Ex
void gv_dx(task_t t)
{
  Window *win;
  win=CreateWindow(3,3);
  ImageWindowDist_2I_1O(t,gv_dx,im1,im2,nr_proc[t],v[t],win,spatial_grad_x);
  return;
}

//computes spatial_gradient Ey
void gv_dy(task_t t)
{
  Window *win;
  win=CreateWindow(3,3);
  ImageWindowDist_2I_1O(t,gv_dy,im1,im2,nr_proc[t],v[t],win,spatial_grad_y);
  return;
}

continued on next page
/computes temporal gradient Et
void gv_dt(task_t t)
{
    Window *win;
    win=CreateWindow(3,3);
    ImageWindowDist_21_1O(t,gv_dt,im1,im2,nr_proc[t],v[t],temporal_grad);
    return;
}

//computes term b of the formulas
void b_part(task_t t)
{
    ImagePointDist_21_1O_C(t,imb,gv_dx,gv_dy,nr_proc[t],v[t],Op_b,alpha);
    return;
}

void u_n(task_t t)
{
    Window *win;
    win=CreateWindow(3,3);
    ImageWindowDist_11O(t,flow_x,nr_proc[t],v[t],win,ModifiedLaplace_3x3);
    return;
}

void v_n(task_t t)
{
    Window *win;
    win=CreateWindow(3,3);
    ImageWindowDist_11O(t,flow_y,nr_proc[t],v[t],win,ModifiedLaplace_3x3);
    return;
}

void a_part(task_t t);
{
    ImagePointDist_51_1O(t,im_a,gv_dx,flow_x,gv_dy,flow_y,gv_dt,nr_proc[t],v[t],Op_a);
    return;
}

void u_n_1(task_t t)
{
    ImagePointDist_41_1O(t,flow_x,gv_dx,im_a,im_b,nr_proc[t],v[t],Op_flow_field);
    return;
}
5.6 Experiments

\[\text{Figure 5.25: Speedup of optical flow for 1024X1024 image size}\]

\[
\begin{align*}
\text{continued from previous page} \\
\text{void v.n.1(task.t t)} \\
\{ \\
\text{ImagePointDist.4L.1O(t,flow.y,gv.dy,im.a,im.b,nr.proc[t],v[t],Op.Flow.field);} \\
\text{return;} \\
\} \\
\text{void optical_flow(task.t t)} \\
\{ \\
\text{ImagePointDist.2L.1O(t,flow.flow.x,flow.y,nr.proc[t],v[t],DisplayImage);} \\
\text{return;} \\
\}
\]

<table>
<thead>
<tr>
<th>Processors</th>
<th>DATA</th>
<th>TASK</th>
<th>TwoL</th>
<th>CPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>468.21</td>
<td>468.21</td>
<td>468.21</td>
<td>468.21</td>
</tr>
<tr>
<td>2</td>
<td>242.59</td>
<td>267.54</td>
<td>240.10</td>
<td>238.88</td>
</tr>
<tr>
<td>4</td>
<td>145.85</td>
<td>217.77</td>
<td>120.36</td>
<td>119.74</td>
</tr>
<tr>
<td>8</td>
<td>86.38</td>
<td>217.77</td>
<td>78.95</td>
<td>75.15</td>
</tr>
<tr>
<td>16</td>
<td>78.69</td>
<td>217.77</td>
<td>61.85</td>
<td>58.45</td>
</tr>
<tr>
<td>32</td>
<td>69.67</td>
<td>217.77</td>
<td>52.78</td>
<td>50.72</td>
</tr>
</tbody>
</table>
Table 5.8: Speedup of optical flow for different scheduling algorithms

<table>
<thead>
<tr>
<th>Processors</th>
<th>DATA</th>
<th>TASK</th>
<th>TSAS</th>
<th>CPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.93</td>
<td>1.75</td>
<td>1.95</td>
<td>1.94</td>
</tr>
<tr>
<td>4</td>
<td>3.21</td>
<td>2.15</td>
<td>3.89</td>
<td>3.91</td>
</tr>
<tr>
<td>8</td>
<td>5.42</td>
<td>2.15</td>
<td>5.93</td>
<td>6.23</td>
</tr>
<tr>
<td>16</td>
<td>5.95</td>
<td>2.15</td>
<td>7.57</td>
<td>8.01</td>
</tr>
<tr>
<td>32</td>
<td>6.72</td>
<td>2.15</td>
<td>8.87</td>
<td>9.23</td>
</tr>
</tbody>
</table>

Figure 5.26: Relative performance at execution, with different scheduling algorithms for optical flow

5.6.3 Artificial Task Graphs

We have designed and studied 10 synthetic IATG's. In Figure 5.27 we show the relative performance of the scheduling algorithms relative to CPR, for the 10 synthetic task graphs (all of them being serial-parallel graphs). Because the figure contains information about all the 10 synthetic graphs, we show in the figure the average value of each measure of relative performance, and the interval of relative performance values, for each scheduling algorithm and number of processors. One can note that CPR performs best of the scheduling algorithms studied. TwoL is second best.

5.7 Conclusions

In this chapter we have presented a method for simultaneous exploitation of data and task parallelism in image processing applications. The data parallel approach, based on algorithmic skeletons, is presented in Chapter 4. Here, we have presented
the task parallel approach, based on the Image Application Task Graph, and the computation of the IATG communication and processing costs. We have studied the influence of different scheduling algorithms (DATA, TASK, TSAS, TwoL and CPR) on the performance of image processing applications. The multi-baseline stereo vision is an example of an image processing application which contains parallel tasks, each of the tasks being a very simple image point or neighborhood operator. Using both data and task parallelism is more efficient than using only data parallelism. Our code for the data and task parallel environment, written in C and using the MPI-Panda library [99, 89] can be easily ported to other parallel machines. The optical flow is an image processing application having an IATG which contains a lot of dependencies between tasks. The width of the graph is three, so the data parallelism dominates over the task parallelism. Exploiting both data and task parallelism gives better results, also in this case.
Chapter 6

Exploiting data and task parallelism in a parallel programming language

6.1 Introduction

In this chapter we present some implementation issues for exploiting the amount of parallelism present in image processing applications implemented in a data parallel programming language. We are interested in a way to express data parallelism, task parallelism or both, in order to parallelize an image processing application implemented in such a language. We have tried out the Spar language due to its feature of being a language for high performance computing.

The Spar language appeared as an alternative to the very popular Fortran language. Fortran is a very good programming language for scientific computations due to features like multidimensional arrays, complex numbers, and in the recent versions array expressions and data-parallel extensions. But it has some major drawbacks such as the lack of support for pointers or for object oriented programming. Spar [109] is a modern language designed as an extension to the Java language. It is designed to improve the expressiveness and performance of Java for scientific computation. Spar contains language extensions for the manipulation of multi-dimensional arrays. Furthermore, Spar is a data-parallel language which proves to be very efficient in parallelizing array-based algorithms.

6.2 Spar language extensions

Spar provides support for multidimensional arrays as a straightforward generalization of Java arrays. This feature is very useful for implementing image processing algo-
rithms. For example, the code from Figure 6.1 creates an image, and assigns to the first pixel a grey value.

Spar offers also support for complex numbers, by means of a new primitive type complex. A floating point literal with the suffix i represents an imaginary number.

6.3 Program annotations

Spar is designed for parallel programming of high performance computing applications, such as image processing. The user is required to identify fragments of code that can be executed in parallel. The Spar compiler does not automatically parallelize code.

Spar provides explicit support for parallel programming via a set of constructs and annotations.

Spar provides the each and foreach statements. The each statement is similar to the par statement from Compositional C++ [115]. Given an each statement such as:

```
each { s1; s2; }
```

the compiler may choose one of the execution orders s1; s2; or s2; s1;. It is guaranteed by the compiler that, even for compound statements, every statement is executed as a one state transition.

The foreach statement is a parameterized version of the each statement. As with the each statement, it is guaranteed that every iteration instance of a foreach statement is executed as one state transition. Thus iterations cannot influence each other during their execution. Parallel execution of the foreach statement is only allowed when there is no observable interference between iterations. In general, to parallelize a foreach statement, the compiler must perform a data dependency analysis, or the user must inform the compiler by using an annotation.

6.4 Supported pragmas

The each and foreach statements can only be executed when there is no observable interference between statements or iterations. The user must inform the compiler with an annotation. The Spar language provides a general mechanism to annotate
6.4 Supported pragmas

a program. Each annotation consists of a list of pragmas. These pragmas allow the user to give the compiler further information about the program, and give hints for efficient compilation. A pragma does not influence the behavior of a program; it only improves the efficiency of the program. To help the compiler with the parallelization of a program, the user may annotate a program with pragmas to specify the placement of data, or the place where a block of code is executed. The annotations can be placed to declarations, statements, expressions, types, formal parameters or the entire program.

1. The ProcessorType pragma is used to declare a processor type and to describe the processor characteristics and capabilities. ProcessorType pragmas must be global.

    <$ ProcessorType = ((Gpp '"Pentium3"')
    (Dsp '"Trimedia"')) $>

    The strings "Pentium3" and "Trimedia" refer to processor descriptions that are known to the compiler, for example through configuration files.

2. The Processors pragma is used to name and list the processors in a system. The Processors pragma must be a global pragma. Only a single Processors pragma is allowed in a program.

    Either a single processor can be declared, or an array of processors. The processor array can be multidimensional. For example, we can have a pragma like:

    <$ Processors = ((Gpp gpp1)
    (Dsp dsp1D[4]) (Dsp dsp2D[2,3])) $>

    The system described above consists of a single processor gpp1 of type Gpp, a one-dimensional processor array dsp1D of type Dsp, and a two-dimensional array dsp2D of type Dsp.

3. The on pragma is used to annotate a Spar program, allowing the placement of data or work on specific processors. The on pragma can annotate declarations, expressions, statements, and member functions.

    <$ on = Gpp $>
    <$ on = Dsp[0] $>
    <$ on = _all $>
    <$ on = Dsp[_] $>

    The special value _all is used to denote all processors and the value _ denotes an unspecified placement.

    With the on pragma it is allowed to use two special placement functions. The (block a * i + b m) placement function places index i onto processor p = (a · i + b)/m. The value of p is bounded by the index range allowed in the
Stereo_spar ()
BEGIN

...........
Image[] < $ on = (lambda (i) P[(cyclic i)]) $ > diff = new Image[16];

...........
foreach(d :- 1:16) < $ on = P[(cyclic @d)]$ >
{
diff[d] = st.DiffImage(d);
diff[d].ExtendImage(win.GetWinY());
diff[d].ErrorImage(win.GetWinY(),diff[d].GetX(),diff[d].GetY());
diff[d].RestoreImage(win.GetWinY());
}

...........
END

Figure 6.2: Implementation of the stereo vision application

corresponding dimension of the processor type. The (cyclic $a*i+b*m$) placement function places index $i$ onto processor $p = ((a \cdot i + b)/m) \mod P_{ext}$.

For data that is distributed by either block or cyclic function, the compiler is able to generate highly efficient code for the enumeration of local elements and the translation of global to local array indices.

6.5 Implementing image processing applications in Spar

To illustrate the use of program annotations and pragmas we present in Figure 6.2 a fragment of the multi-baseline stereo vision application implemented in the Spar language. In principle, any declaration, statement or expression can be annotated with an on pragma. The new expression is a special case, since this not only specifies the placement of the constructor execution (if not overridden by an annotation on the constructor), but also the placement of the newly constructed class or array instance. The on pragma which annotates the declaration of the "diff" (difference) images in Figure 6.2 specifies that the new images are constructed and distributed cyclically onto the processors. The on annotation used in combination with the cyclic mapping function distributes the array of images cyclically on the number of processors. By using the on annotation we are applying data-parallelism, but actually, for the processing of the 16 disparity images we use the foreach statement also associated with the on pragma, so the processing is done in a task-parallel way. In fact, we are using a data-parallel paradigm but the algorithm is executed in a task parallel way.

The annotated foreach statement tells the compiler to distribute the vector of images cyclically onto the processors. In this way the image diff[i] will be processed by processor number $i \mod P$, where $P$ is the total number of processors.
Table 6.1: Performance of multi-baseline stereo vision implemented in SPAR

<table>
<thead>
<tr>
<th>No. of processors</th>
<th>Time(sec) with Spar</th>
<th>Time(sec) with data/task parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12.15</td>
<td>11.46</td>
</tr>
<tr>
<td>2</td>
<td>6.12</td>
<td>5.79</td>
</tr>
<tr>
<td>4</td>
<td>3.07</td>
<td>2.88</td>
</tr>
<tr>
<td>8</td>
<td>1.57</td>
<td>1.46</td>
</tr>
<tr>
<td>16</td>
<td>0.81</td>
<td>0.75</td>
</tr>
<tr>
<td>32</td>
<td>0.81</td>
<td>0.45</td>
</tr>
</tbody>
</table>

In Table 6.1 we show the execution time of the application implemented in Spar versus the execution time of the application using the data and task parallel framework presented in the previous chapter. The applications were executed on a distributed memory system which consists of a cluster of Pentium Pro/200Mhz PCs with 64Mb RAM running Linux [17], and connected through Myrinet. We used 256 × 256 size image. Using Spar we were interested in a way to use task parallelism for this type of application. We have created an array of objects (images) and then using annotations we distributed the array on the available number of processors. One can note that in the current implementation, from 16 processors on, Spar cannot exploit the task parallelism feature of this application anymore, and because mixed data and task parallelism cannot be used yet, the performance of the multi-baseline stereo vision application remains constant. Allowing processors subsets and a mechanism to send data between processors would extend the parallelization expressiveness for this application, in a way similar to our proposed data and task parallel framework. A possibility to increase the parallelization efficiency even more, which was also discussed and should be reconsidered in our framework, is for Spar to provide a mechanism for efficient data redistribution between subsets of processors. In this way, one avoids the overhead caused by repetitive scatter/gather of the data and point-to-point data communication between processors, which is our current framework implementation. This application illustrates the need to represent task/data-parallel computations in a data-parallel language such as Spar. Basically, there are three approaches for the integration of task/data-parallel computations in a data parallel language such as Spar: implicit approaches based on extending the compiler support, explicit approaches based on extending the language annotations for task coordination and explicit task-parallel coordination libraries.

- **Compiler-based** approaches are based on the development of more sophisticated compilers capable of extracting task-parallel algorithms from data-parallel specifications. Usually, they use new directives to trigger the application of specific transformations. An example is the use of the PROCESSES directive [85] to demarcate a part of code that should be transformed in a task parallel algorithm. However, these approaches tend to increase the complexity of the mapping from the user program to executable code.

- **Language-based** approaches represent explicit ways that allow the programmers to specify the creation and coordination of tasks. Specific annotations are intro-
duced for tasks, as in [88] for the Spar language, currently only available for annotating single processor tasks. Other approaches use a graphical notation [10], channels [28] or pipeline notations [91] to connect data-parallel computations. There is as yet no consensus or standardization on which language constructs should be used.

- **Explicit task-parallel coordination libraries** represent a third alternative to the integration of task and data parallelism that avoids the difficulties associated with compiler-based and language-based approaches. For a data-parallel language we can design language bindings to communication libraries like PVM or MPI. In general, MPI and PVM implementations support only bindings to sequential languages such as C or Fortran. However, MPI or PVM functions can also be used for communication among data-parallel tasks. In this way we can implement data and task parallel applications, without the need for a new compiler technology or more language extensions. By using the on pragma for data decomposition combined with MPI calls for sending data between sets of processors we can obtain a very efficient data and task parallel integration within a data parallel language, see [32]. The drawback with this approach is that it requires explicit parallel implementation from the application programmer.

At this moment, the Spar compiler can schedule and generate code only for programs containing single processor tasks (tasks mapped only on one processor). Extending the compiler to built code for multiprocessor tasks is part of future work. A possibility for embedding data and task parallelism in the Spar language would be by providing MPI or PVM bindings for it.

### 6.6 Conclusions

In this chapter we have presented the Spar language, an extension of the Java language, for data-parallel computation. The language was specially designed for scientific computation, so the applications from the image processing area are very suitable for Spar implementation. Spar is an annotated parallel programming language in which an application programmer must explicitly interfere to express the parallelism using annotations and pragmas, in order to parallelize applications implemented in this language.

The language extensions allow the programmer to write parallel programs in a portable and understandable manner. The current compiler only supports data-parallel programming. The each and foreach statements together with the on pragma gives sufficient support for expressing task parallelism within an application, such as the multi-baseline stereo vision. However, the multi-baseline stereo vision application is an example of an application which can exploit both data and task parallelism, but for which Spar does not yet provide support, i.e. for expressing both paradigms by selecting processor subsets on which tasks must be executed.
Chapter 7

Conclusions

Computer vision tasks consist of image processing algorithms which can be classified in three large groups: low-level image processing algorithms, intermediate-level image processing algorithms, and high-level image processing algorithms. In the last few years the size of the images have increased and new algorithms have been developed, with high computational demands. At the same time there is a growing demand for real-time image processing. Despite the recent advances in computer technology, there is still a gap between the processing capabilities and the actual demands. This gap can be bridged by parallelizing computer vision applications on different types of parallel hardware architectures.

The aim of this research was to design a programming model for the development of time-constrained image processing applications on currently available parallel architectures, like a cluster of workstations. The goal was to bring the benefits of parallel computing to the image processing community at large, without requiring comprehensive skills needed to write a parallel program.

In this thesis, we have addressed the problem of parallelizing low-level image processing applications on distributed-memory systems (cluster of workstations), by simultaneous exploitation of data and task parallelism. Our approach uses the degree of task parallelism present in many image processing applications to control the degree of data parallelism used for individual tasks. In our view, a task represents an image processing operator. We have implemented our approach as a data and task parallel framework intended for low-level image processing applications. In order to exploit the available data parallelism we have used the concept of algorithmic skeletons. Skeletons are algorithmic abstractions that try to hide the parallelism from the application programmer. We have presented algorithmic skeletons which cover the whole range of low-level image processing operators. In order to exploit the available task parallelism, we use the Image Application Task Graph (IATG) representation. Given an IATG of an application, we use allocation and scheduling algorithms to present a suitable data and task parallel execution scheme. Allocation decides on the number of processors used by each data parallel task in the given application and scheduling decides on
an order of execution for the tasks. The allocation and scheduling problem is NP-complete and we have studied and developed different heuristic methods to solve it. The proposed framework can be further extended to intermediate and high-level image processing applications. We demonstrated that using both paradigms may yield better results than using only one of them. The main goal is to exploit both paradigms in image processing applications, while maintaining the environment as user friendly as possible.

The safest form of parallelization would be to have no special parallelization constructs at all, but to leave parallelization entirely to the compiler. Unfortunately, this is currently only feasible for a limited class of algorithms. At the other extreme, the user can write explicitly parallel programs using communication libraries such as PVM or MPI, and perhaps explicit threading such as provided by the Java Thread class. This method is difficult, error prone, and often results in programs that are not portable. Therefore, the latter approach is not appropriate for application programmers without enough knowledge of parallel computing. Since both these extremes are undesirable, we have presented a few alternative solutions for parallelizing applications from the image processing field. One solution is to design a user friendly environment which embeds both data and task parallel support. Another solution would be to use a parallel programming language, like Spar [109], which offers the application programmer the possibility to provide explicit additional information for parallelizing a program, while the generation of the parallel code is left to the compiler.

In addition, we have studied the benefits of using other types of parallel architectures, like the SIMD architectures, for implementing image processing applications. Usually, these architectures come with a dedicated data parallel programming language, like the 1DC language for the IMAP-Vision architecture, or the applications must be implemented in assembly language, as for the MMX and SSE like microprocessor architectures. In order for these systems, to fit nicely in our scheme, dedicated compilers such as 1DC should preferably be replaced by compilers for a sequential language augmented with pragmas (see Chapter 6).

In Chapter 3 we investigated the main two types of parallel hardware architectures, namely the SIMD and MIMD systems, and the way of parallelizing image processing applications on these systems.

In Chapter 4 we presented a framework, based on algorithmic skeletons, for parallelizing image processing applications using the data-parallel approach.

In Chapter 5 we presented a task parallel framework and the way of embedding data and task parallelism for parallelizing image processing applications.

In Chapter 6 we studied the possibility to embed the use of data and task parallelism in parallel programming languages which, usually, offer only data-parallel support. We have presented the Spar language and the approaches to extend a data-parallel language with task parallel capabilities.

There are some future improvements to be made to our work. One of the issues would be to improve the data redistribution scheme. Data redistribution is critical for implementing a task and data parallel execution scheme. Our implementation was very simple, the master processor gathers the data processed by the allocated processors
in a task and sends it to the master processor of the successor task in the graph. It would be more efficient if the processors allocated to a task can send the computed data directly to the processors allocated to the successor task in the graph, as it was proposed in [84]. Another extension would be performing the data dependency analysis of a given image processing application. In our research, we started from the assumption that we already have the information related to data dependencies in the form of the Image Application Task Graph. The "lazy programmers method", developed by [93] in the same research project as this thesis, is a run-time system that has similarities with a page fault mechanism for virtual memory: it only inserts inter-processor communication when data is missing or outdated on a certain processor. This method would be an excellent tool to replace our simple data redistribution scheme, yielding a system that has to be best of both worlds. However, the real impact of this fusion has to be proven in future.
Bibliography


List of publications


Appendix A

Posynomial functions and convex programming

A.1 Convex sets

Definition 13 (Convex Set) A set $C$ in $\mathbb{R}^n$ is said to be convex if, for every $x_1, x_2 \in C$, and every real number $\alpha$, $0 \leq \alpha \leq 1$, the point $\alpha x_1 + (1 - \alpha) x_2 \in C$.

Geometrically, this definition has a very simple illustration. A set is convex if, given two points in the set, every point on the line segment joining the two points is also a member of the set. In Figure A.1 we show examples of convex and nonconvex two dimensional sets.

A.2 Convex functions

Definition 14 (Convex Function) A function $f$ defined on a convex set $\omega$ is said to be convex if, for every $x_1, x_2 \in \omega$, and every $\alpha$, $0 \leq \alpha \leq 1$,
\[ f(\alpha x_1 + (1 - \alpha)x_2) \leq \alpha f(x_1) + (1 - \alpha)f(x_2) \quad (A.1) \]

\( f \) is said to be strictly convex if the above inequality is strict for \( 0 < \alpha < 1 \). Examples of convex and concave one dimensional functions are shown in Figure A.2. Geometrically, a function is convex if the line which joins two points of the function graph is always above the graph. A function is said to be concave if the line which joins two points is always below the graph. There are functions which are neither convex, nor concave.

### A.3 The convex programming problem

The formulation of the convex programming problem is

\[
\text{minimize}(f(x)) \quad (A.2)
\]

\text{such that} \quad x \in S \quad (A.3)

where \( f \) is a convex function and \( S \) is a convex set. The solution of the problem has the property that any local minimum of \( f \) over \( S \) is a global minimum. A convex programming formulation can be solved in polynomial time [106].

### A.4 Posynomial functions

**Definition 15 (Posynomial Function)** A function \( f \) of a positive variable \( x \in \mathbb{R}^n \) is said to be posynomial if it has the form
\[ g(x) = \sum_{j} \gamma_j \prod_{i=1}^{n} x_i^{\alpha_{ij}} \]  
(A.4)

where exponents \( \alpha_{ij} \in \mathbb{R} \) and the coefficients \( \gamma_j > 0 \).

A posynomial function is, in fact, a polynomial function with the following limitations and extensions:

- The coefficients \( \gamma_j \) must be positive.
- An exponent \( \alpha_{ij} \) could be any real number, and not necessarily a positive integer, as in the case of polynomials.

A posynomial has the property that it can be mapped onto a convex function through an elementary variable transformation

\[ x_i = e^{xi} \]  
(A.5)

Such a transformation maps the problem of minimizing a posynomial function to a convex programming problem (which can be solved in polynomial time).

A.4.1 Properties of posynomial functions

If \( f \) and \( g \) are convex functions on a convex set \( S \) then the following statements are valid:

- The function \( f + g \) is a convex function over \( S \).
- If \( c \) is a positive constant, then the function \( c \cdot f \) is a convex function over \( S \).
- The function \( \text{max}(f, g) \) is a convex function over \( S \).
- The function \( \text{min}(f, g) \) is a convex function over \( S \).

Taking into consideration the above statements we can deduct that the critical path and average area (defined in Chapter 5) are both posynomial (so convex) functions.
Appendix B

Estimation of processing and communication cost

B.1 Processing cost estimation

The estimation of the processing cost for each image processing operator is based on a set of measurements performed using a varying number of processors. In this appendix we present the method we used to estimate the processing cost of an image processing operator on a number of processors which is not part of the measured data. We are interested in how inaccuracies in the measurement data propagate to inaccuracies in the estimation of $w(t, n(t))$, for a $n(t)$ number of processors allocated to the image processing task $t$.

The processing cost model that we use is defined in Section 5.4.1 by Formula B.1, where $0 \leq \alpha(t) \leq 1$ and $\tau(t) \geq 0$.

$$w(t, n(t)) = (\alpha(t) + \frac{1 - \alpha(t)}{n(t)}) * \tau(t) \quad (B.1)$$

In Section 5.4 we showed that the processing cost model in the above formula can be reduced to a nonlinear function of the form presented in Formula B.2, where $a = (1 - \alpha)\tau$ and $b = \alpha\tau$.

$$w(x) = \frac{a}{x} + b \quad (B.2)$$

For the non-linear function showed in Formula B.2 we want to find the regression curve, see Figure B.1, i.e. the parameters $a$ and $b$ of the curve that represents approximately the shape of the pairs of measured values. In our formulas $x_1, x_2, ..., x_N$ are a set of
values for which the values $y_1, y_2, ..., y_N$ of the function are measured. We will find the model parameters by using the least squares principle. Because we only have inaccuracies in the vertical direction $y_i$, we calculate the error function as the sum of the squares of the vertical distances $v_i$, see Formula B.3.

$$\delta = \sum_{i=1}^{N} v_i^2 = \sum_{i=1}^{N} (y_i - w(x_i))^2$$  \hspace{1cm} (B.3)

Minimizing the error function means solving the system of equations $\partial \delta / \partial a = 0$ and $\partial \delta / \partial b = 0$. From these, the expressions for computing parameters $a$ and $b$ can be determined, see Formulas B.4 and B.5.

$$a = \frac{\sum_{i=1}^{N} \frac{y_i}{x_i} - \bar{y} \sum_{i=1}^{N} \frac{1}{x_i}}{\sum_{i=1}^{N} \frac{1}{x_i^2} - \frac{1}{N} (\sum_{i=1}^{N} \frac{1}{x_i})^2}$$  \hspace{1cm} (B.4)

$$b = \bar{y} - a \sum_{i=1}^{N} \frac{1}{x_i}$$  \hspace{1cm} (B.5)

Using the notations:

$$W_x = \sum_{i=1}^{N} \frac{1}{x_i} \quad W_y = \sum_{i=1}^{N} y_i \quad W_{xx} = \sum_{i=1}^{N} \frac{1}{x_i^2} \quad W_{xy} = \sum_{i=1}^{N} \frac{y_i}{x_i} \quad \bar{y} = \frac{P_y}{N}$$  \hspace{1cm} (B.6)
we obtain Formulas B.7 and B.8 for determining the parameters $a$ and $b$ of the nonlinear function.

$$a = \frac{NW_{xy} - W_x W_y}{NW_{xx} - W_x^2} \quad \text{(B.7)}$$

$$b = \frac{W_y W_{xx} - W_x W_{xy}}{NW_{xx} - W_x^2} \quad \text{(B.8)}$$

For determining the estimation inaccuracy of model parameters $a$ and $b$ we use the following formulas:

$$\sigma_a^2 \approx \sigma_y^2 \left( \frac{\partial a}{\partial y_i} \right)^2 = \frac{N\sigma_y^2}{NW_{xx} - W_x^2} \quad \text{(B.9)}$$

$$\sigma_b^2 \approx \sigma_y^2 \left( \frac{\partial b}{\partial y_i} \right)^2 = \frac{W_{xx}\sigma_y^2}{NW_{xx} - W_x^2} \quad \text{(B.10)}$$

The formulas for calculating the processing cost parameters $\alpha$ and $\tau$ are:

$$\tau = \frac{NW_{xy} - W_x W_y + W_y W_{xx} - W_x W_{xy}}{NW_{xx} - W_x^2} \quad \text{(B.11)}$$

$$\alpha = \frac{W_y W_{xx} - W_x W_{xy}}{NW_{xy} - W_x W_y + W_y W_{xx} - W_x W_{xy}} \quad \text{(B.12)}$$

### B.2 Communication cost estimation

The same reasoning can be applied to the estimation of the communication cost between two processors. The communication time is defined in Section 5.4.2, by the following formula, where $i$ is the processor which sends the message, $j$ is the processor which receives the message and $L$ is the message size.

$$c(i, j, L) = t_s + L * t_b \quad \text{(B.13)}$$

We are interested in the estimation of $t_s$ and $t_b$, given a set of measurements (communication times) for different sizes of data (images), see Figure B.2. Because the
communication time is a function of the message length, we can reduce the above formula to the following linear function, by substituting $L$ with $x$:

$$c(x) = t_b \times x + t_s$$  \hspace{1cm} \text{(B.14)}

Using the notations:

$$C_x = \sum_{i=1}^{N} x_i \hspace{0.5cm} C_y = \sum_{i=1}^{N} y_i \hspace{0.5cm} C_{zx} = \sum_{i=1}^{N} x_i^2 \hspace{0.5cm} C_{zy} = \sum_{i=1}^{N} x_i y_i \hspace{0.5cm} \bar{y} = \frac{C_y}{N} \hspace{0.5cm} \bar{x} = \frac{C_x}{N}$$  \hspace{1cm} \text{(B.15)}

we obtain the following expressions for parameters $t_s$ and $t_b$ and the estimation inaccuracies, when using the least squares principle of minimizing the error function defined as the sum of squares of vertical distances $v_i$, see Appendix B.1:

$$t_b = \frac{NC_{xy} - C_x C_y}{NC_{xx} - C_x^2}$$  \hspace{1cm} \text{(B.16)}

$$t_s = \frac{C_y C_{zx} - C_x C_{zy}}{NC_{xx} - C_x^2}$$  \hspace{1cm} \text{(B.17)}

$$\sigma^2_{t_b} \approx \frac{\sigma_y^2}{\sum_{i=1}^{N} \left( \frac{\partial t_b}{\partial y_i} \right)^2} = \frac{N\sigma_y^2}{NC_{xx} - C_x^2}$$  \hspace{1cm} \text{(B.18)}
\[ \sigma_{t_s}^2 \approx \sigma_y^2 \sum_{i=1}^{N} \left( \frac{\partial t_s}{\partial y_i} \right)^2 = \frac{C_{xx} \sigma_y^2}{NC_{xx} - C_x^2} \] (B.19)
Summary

Embedding data and task parallelism in image processing applications

Cristina Soviany

The focus of this thesis is on studying the benefits of exploiting both data and task parallelism in low-level image processing applications using algorithmic skeletons and the Image Application Task Graph (IATG). Task and data parallelism are often considered to be mutually exclusive approaches to parallel programming. Yet many applications from several fields can benefit from both forms of parallelism. This applies also to image processing. A lot of algorithms have been developed for parallelizing image processing algorithms on parallel architectures. Most of these algorithms are either specifically designed for a certain parallel hardware architecture or very difficult to implement by an image processing application programmer, novice in the parallel computing area.

Chapter 1 is an introduction to the topics of parallel computing. The concepts of data parallelism and task parallelism are introduced. Three categories of building high level systems for parallelizing applications are presented and the goal of this thesis, to address the systems from the second category, is introduced.

Chapter 2 is a brief introduction to the image processing field with a classification of the image processing algorithms in low-level, intermediate-level and high-level operators. Usually, an image processing application consists of a set of image operators, starting with low-level ones and ending with intermediate or possibly high-level operators. The data parallel concept explained in this thesis can be introduced for processing the low-level operators, while the task parallel concept can be applied to all three categories.

Chapter 3 gives an overview of the realm of parallel hardware architectures and programming paradigms and their applicability in image processing. MIMD and SIMD type hardware architectures were investigated, together with their specific parallel programming methods or languages. The survey is illustrated with experiments.

Chapter 4 introduces the concept of algorithmic skeletons and their applicability in designing data parallel image processing algorithms. Algorithmic skeletons are designed patterns which encapsulate and hide the parallelism from the user. The aim is
to obtain environments or languages that allow easy parallel programming, in which
the user does not have to handle with problems like communication, synchronization,
dead-locks or non-deterministic program runs.

Chapter 5 introduces the concept of Image Application Task Graph (IATG) and the
combined data and task parallel approach of parallelizing image processing applica-
tions. The data parallel approach is based on the algorithmic skeletons introduced in
Chapter 4. The combined data and task parallel approach is performed by extract-
ing the application task graph and performing the image processing tasks in parallel.
Each task is executed, by means of a skeleton, on one or more processors. Different
scheduling algorithms were investigated for their influence on the performance of some
image processing applications.

Chapter 6 presents some implementation issues for exploiting the amount of parallelism
present in image processing applications using Spar, a data parallel language.

The conclusion is that the image processing application programmers can benefit from
the simultaneous exploitation of both data and task parallelism, even for simple low-
level image processing applications. The data and task parallel environment is based
on algorithmic skeletons and the Image Application Task Graph.
Samenvatting

Embedding data and task parallelism in image processing applications

Cristina Soviany

Dit proefschrift richt zich op onderzoek naar de voordelen van het zowel data- als taak-parallel uitvoeren van image processing applicaties, gebruik makend van de concepten "algorithmic skeleton" en "Image Application Task Graph (IATG)". Taak en data parallelisme worden in het gebied van de parallelle programmering vaak afgeschilderd als elkaar uitsluitende aanpakken. Toch kunnen veel applicaties profiteren van het gelijktijdig gebruik maken van beide vormen van parallelisme. Dit geldt met name voor het bewerken van beelden, waarbij de segmentatie een groot aantal pixels transformeert moet worden naar een aantal onafhankelijke objecten. Er zijn vele parallel beeldbewerking algoritmen ontwikkeld voor parallelle architecturen. Echter de meeste algoritmen zijn ofwel speciaal ontworpen voor een specifieke parallelle architectuur, of zijn zeer moeilijk te implementeren door een beeldbewerking applicatieprogrammeur die weinig bekend is met parallel rekenen.

Hoofdstuk 1 is een introductie in de kernpunten van parallel rekenen. Het introduceert de concepten data parallelisme en taak parallelisme. Vervolgens worden drie uitgangspunten voor het ontwerp van programmeer systemen voor het paralleliseren van applicaties op bestaande parallelle hardware gepresenteerd. Als eerste het gezichtspunt van de puur sequentieel denkende programmeur, als tweede het gezichtspunt van een programmeur die informatie heeft over de mogelijke afhankelijkheden in zijn software en inzicht heeft in welke operaties mogelijkerwijs parallel kunnen worden uitgevoerd. En als derde gezichtspunt een programmeur die volledig inzicht heeft in de parallelle architectuur. Het doel van dit proefschrift, programmeersystemen voor het tweede gezichtspunt, wordt in dit hoofdstuk geïntroduceerd.

Hoofdstuk 2 is een korte introductie op het gebied van de beeldbewerking. Hierbij wordt een onderscheid gemaakt in algoritmes voor laag-, midden- en hoog-nivo bewerkingen op beelden. Veelal bestaat een beeldbewerking applicatie uit een set van operatoren op beelden, beginnend bij laag nivo operatoren en eindigend bij midden of hoog nivo operatoren. Het data-parallelle concept van dit proefschrift wordt geïntroduceerd
voor laag nivo operatoren, terwijl het taak-parallelle concept op all nivo’s gebruikt kan worden.

Hoofdstuk 3 geeft een overzicht van het rijk van de parallelle hardware architecturen en programmeerparadigma’s, als wel hun toepasbaarheid in de beeldbewerking. MIMD en SIMD type hardware architecturen zijn onderzocht, in samenhang met hun specifieke programmeermethoden en programmeertalen. Het overzicht is verduidelijkt met enkele experimenten.

Hoofdstuk 4 introduceert het concept algoritmische skeletten en hun toepassing in het ontwerp van data-parallelle beeldbewerkingalgoritmen. Algoritmische skeletten zijn voorgevormde patronen die het parallelisme inkapselen en verbergen voor de gebruiker. Door gebruik te maken van skeletten kan men programmeer-omgevingen en -talen maken die de gebruiker bevrijden van problemen met communicatie, synchronisatie, dead-lock en niet determineerbare programma aflopen.

Hoofdstuk 5 introduceert zowel het concept van de Image Application Task Graph (IATG), als wel de gecombineerde data- en taak parallel aanpak om beeldbewerkingapplicaties te paralleliseren. De data-parallelle aanpak is gebaseerd op de algoritmische skeletten, zoals geïntroduceerd in Hoofdstuk 4. De gecombineerde data en taak parallelle aanpak wordt gerealiseerd door de extractie van een applicatie taakgraaf en het vervolgens parallel executeren van de taken uit de graaf. Elke taak is vervat in een skeleton dat draait op een of meer processoren. Verscheidene schedulingsalgoritmen zijn onderzocht om hun invloed op de prestaties van de parallelle implementaties van een aantal beeldbewerkingapplicaties te onderzoeken.

Hoofdstuk 6 presenteert een aantal implementatieaspecten bij de exploitatie van parallelisme in de data-parallelle programmeertaal SPAR, aan de hand van enkele beeldbewerkingalgoritmen.

De conclusie van dit proefschrift is dat een beeldbewerking applicatie programmeur kan profiteren van het gelijktijdig gebruik maken van zowel data als taakparallelisme gebaseerd op de concepten “algorithmic skeletons” en “Image Application Task Graph”, zelfs bij simpele laag-nivo beeldbewerkingoperaties.
Acknowledgements

It is a pleasure to thank my advisor, Pieter Jonker, for the entire support, enthusiasm and advice that he gave to me during this work and for the comprehensive reviews of this thesis. I appreciate the several discussions and the always opened door whenever it was a question directly related to research or not.

I am also very pleased to thank all the people with whom I worked during the PhD years. Special thanks to all PILE project members, prof. Henk Sips, Pieter Jonker, Dennis Koelma and Frank Seiistra for the many interesting meetings, lunch meetings and discussions that we had during this research project and for the valuable criticism and advice given when reviewing this thesis. Prof. Henk Sips was always open to new ideas and gave us freedom to take certain directions in this project and suggested changes, corrections, and better ways to describe or explain some concepts in the thesis. Many thanks go to one of my best colleagues and friends, Andrei Rădulescu, with whom I wrote a paper awarded at the International Parallel and Distributed Processing Symposium in 2001. Frits Kuijlman and Kees van Reeuwijk gave me a lot of support when experimenting with the Spar compiler.

It is my pleasure to thank prof. Lucas van Vliet, one of my promotors, for his enthusiasm and promptitude in reading and structuring the ideas in my thesis. I also have to mention the time and effort spent by all my promotors and committee members in reading and improving the clarity of presentation.

I would also like to thank to all PH people, staff and aio's, for the friendly environment that I found there and for the nice time spent among them.

My friends, Emilia, Roger, Irina, Andrei Rădulescu, Oana, Stelian, Monica, Andrei Rares, made living in the beautiful Delft to be as enjoyable as possible and were always besides me.

I am also happy to thank the colleagues from Philips CFT for the support provided for improving the layout of this thesis.

I will end by thanking my family. My parents and my brother for their continuous love, support and understanding. Many thanks to a special lady, Cecilia, to whom I am deeply indebted for her dauntless support. I know that for my dearest children, Andrei and Răzvan, sharing me with research and thesis writing was not at all easy. I thank them so much for their tremendous patience and understanding and for making me so proud of them.
Curriculum Vitae

Cristina Soviany was born in Bucharest, Romania on the 16th of June, 1968. In 1986 she went to study computer science and control engineering at the Politechnics University of Bucharest and she graduated in 1991. Till 1992 she worked for the Institute of Control Engineering, in Bucharest. In 1992 she joined the Faculty of Computer Science and Control Engineering where she worked as a teaching and research assistant. In March 1998 Cristina Soviany joined the Pattern Recognition Group of Delft University of Technology as a PhD student. She was involved in the PILE (Parallel Imaging library and Language Environment) project working on embedding data and task parallelism in image processing applications. In February 2002 she joined the Industrial Vision Department of Philips CFT.