EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Feasibility of multi-core programming frameworks in image processing pipelines of wide format printers

Vo, N.

Award date:
2014

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Eindhoven University of Technology
Océ Technologies B.V.

Master's Thesis

Feasibility of Multi-core Programming Frameworks in Image Processing Pipelines of Wide Format Printers

Author:
Nguyen Vo

Supervisor:
Dr. Lou Somers (TU/e)
Dr. ir. Reinier Dankers (Océ)

A thesis submitted in fulfilment of the requirements
for the degree of Master of Science in Embedded Systems

in the

Mathematics and Computer Science Department

August 2014

Abstract

Processors have become faster, smaller, cheaper over the last few decades. Single core processors were still a key solution between 1980 and 2000 when it has more or less followed Moore's law. However, it has lost its pace, starting from the beginning of the last decade. Multi-core processors have been introduced as a better solution. Exploitation of multiprocessor platforms to improve performance is still a hot research area up till now. Research in the printing industry is not an exception.
This project is conducted in Océ; a printing company which produces high speed cut sheet printers, wide format printers and also related printing service software. A print head in a printer consists of many nozzles which jet ink on the printing media such as paper, plastic, etc. To achieve that, the printer needs to transform an input image into the type of printing data which is a list of nozzles to fire at every fire moment. The datapath in the printer is the image processing pipeline that is responsible to perform that transformation. Because the print head is built from a number of arrays of nozzles for each color, the relation between the positions of nozzles and the position of pixels in the input image can be rather awkward and the algorithms for manipulating the images are often not trivial. Thus, the datapath is a time consuming process and it becomes critical to speed it up to improve the performance of the printers. One possible solution is to parallelize the datapath on multi-core platforms using a multi-core programming framework.
The purpose of this project is to evaluate the feasibility of multi-core programming frameworks for the datapath in wide format printers. In this thesis, we define a search space which is a list of multi-core programming frameworks. Then the size of the search space is reduced based on some pre-defined criteria. We also present the implementation, measurement and analysis to explore the characteristics of the frameworks in the reduced search space with the datapath application and trade-offs among them. Collectively, these construct our conclusion about the suitable parallel programming framework for the datapath of the wide format printers which need to handle arbitrarily large input images with limited available hardware resources.

Acknowledgements

This thesis is the result of a six months graduation project performed in Research and Development Department at Océ Technologies B.V. It would be very challenging or perhaps impossible to complete this work without guidance and support from several people. I would like to express my deeply gratitude to all of them.

First of all, I would like to thank Lou Somers, my graduation supervisor for guiding me through the project. His working and coaching attitude never fail to impress me. He has spent a lot time to help me even though he was very busy with his work. I would like to express my gratitude to Reinier Dankers, my supervisor at Océ for his fruitful discussions and supports. His warm guidance makes me feel like working at home despite the fact that I am thousands of miles from my hometown. I would like to thank Alexander Lint for providing important information and documents. He has helped me with implementing the new thread scheduler for IP4. I want to thank Harro Haan for helping me with the ARM platform and the Linux environment. I also appreciate many people at Océ and Embedded System Institute, whom I could not mention all in this short paragraph, for supporting me with their ideas, feedbacks and comments during my time at Océ.

I would like to thank my carpooling friends Barath, Ketan, Kemal and Umar for sharing every trip with me to Venlo. Especially, Barath has spent his time to review my report. I would like to mention Twan Basten and Marc Geilen for joining my examination committee. Their comments during my preparation and mid-term presentations are really valuable and helpful for me.

Last but not least, I have to express my love to my family and Sally for always being with me. Physically we are thousands of miles apart but you are always in my heart!

Contents

Abstract **i**

Acknowledgements **ii**

Contents **iii**

List of Figures **vi**

List of Tables **viii**

Abbreviations **ix**

1 Introduction **1**
- 1.1 Background . . . 1
  - 1.1.1 Océ . . . 1
  - 1.1.2 Wide Format Printers . . . 1
- 1.2 Problem Description . . . 2
  - 1.2.1 Problem . . . 2
  - 1.2.2 Approach . . . 3
- 1.3 Report organization . . . 4

2 State of The Art **5**
- 2.1 Multi-core Processors . . . 5
- 2.2 Parallel Programming Frameworks . . . 6
- 2.3 Performance Modeling . . . 7
  - 2.3.1 Amdahl's law . . . 7
  - 2.3.2 Overheads . . . 8

3 Datapath **12**
- 3.1 The Datapath in Wide Format Printers . . . 12
- 3.2 Describing The Datapath with SDF . . . 15
- 3.3 Chapter Summary . . . 16

4 Alternatives of The Datapath Parallelization **17**
- 4.1 Alternatives'Description . . . 17
  - 4.1.1 Sequential code . . . 18
  - 4.1.2 Native Threading Packages . . . 18

4.1.3 OpenMP . . . 19
4.1.4 SDF3 . . . 19
4.1.5 IP4 . . . 19
4.1.6 Threading Building Blocks . . . 19
4.2 Comparison Criteria . . . 20
4.2.1 Execution Time . . . 20
4.2.2 Portability . . . 21
4.2.3 Flexibility . . . 22
4.2.4 Code Size . . . 22
4.2.5 Complexity . . . 23
4.2.6 Memory Usage . . . 23
4.2.7 Maintainability . . . 23
4.2.8 Cost . . . 24
4.2.9 Data Flow Control Support . . . 24
4.3 Alternatives Comparison . . . 24
4.4 Chapter Summary . . . 26

5 IP4 27
5.1 Introduction . . . 27
5.2 IP4 with The Datapath . . . 32
5.3 Experiments . . . 32
5.3.1 Experiment: the startup overhead of IP4 . . . 34
5.3.2 Experiment: IP4 on the multi-core platform . . . 34
5.3.3 Experiment: Flexibility of the datapath in IP4 . . . 36
5.3.4 Experiment: Different input data sizes in IP4 . . . 37
5.4 Thread Scheduling in IP4 . . . 38
5.5 Chapter Summary . . . 40

6 OpenMP 41
6.1 Introduction . . . 41
6.2 OpenMP with The Datapath . . . 43
6.3 Experiments . . . 44
6.3.1 Experiment: the startup overhead of OpenMP . . . 44
6.3.2 Experiment: OpenMP on the multi-core platform with the different loop schedulers . . . 44
6.3.3 Experiment: OpenMP with the different chunk sizes in the loop scheduler . . . 48
6.3.4 Experiment: the speedup of the tasks of the datapath with OpenMP on different platforms . . . 51
6.3.5 Experiment: Different input data sizes in OpenMP . . . 54
6.3.6 Experiment: Compatibility with the development environment . . 55
6.4 Chapter Summary . . . 57

7 Combined IP4 and OpenMP 58

8 Conclusion and Future Work 60
8.1 Conclusion . . . 60
8.2 Future Work . . . 62

A The Datapath in Wide Format Printers Description **63**

Bibliography **64**

List of Figures

1.1 Océ Arizona 350XT [1] 2
1.2 Project approach 3

2.1 The maximum speedup according to the Amdahl's law [2] 8
2.2 Deconstruction framework [3] 10

3.1 Swaths in wide format printing 13
3.2 The datapath of wide format printers 13
3.3 Scheduling tasks of the datapath in the real printing case 14
3.4 An example of required swaths for NFC 14
3.5 Synchronous data flow graph of the datapath of WFP 16

5.1 IP4 components 27
5.2 Logical view of IP4 28
5.3 An IP4 script example 29
5.4 An example of IP4 pipeline 29
5.5 IP4 pipeline types 30
5.6 Parallel pipeline division in IP4 31
5.7 An example of IP4 execution unit which consists of task and operations . 31
5.8 IP4 implementation graph of the datapath in WFP 32
5.9 The execution time of the RO task in IP4 on a different number of cores of the platform I 35
5.10 The speedup of the RO task in IP4 with different swath sizes 37
5.11 Thread scheduling in the test program 39
5.12 New thread scheduling in IP4 39

6.1 The OpenMP Fork-Join model [4] 42
6.2 The execution time of the RO task with the different OpenMP loop schedulers with a different number of threads on 4 cores of platform I 46
6.3 The execution time of the RO task with the different OpenMP loop schedulers on a different number of cores of platform I 46
6.4 The execution of threads on 4 cores 47
6.5 The execution of 4 threads on a different number of cores 48
6.6 The execution time of the RO task with different chunk sizes in the OpenMP dynamic loop scheduler 49
6.7 The execution time of the MA task with different chunk sizes in the OpenMP dynamic loop scheduler 50
6.8 The execution time of the MA task with different chunk sizes in the OpenMP dynamic loop scheduler (larger range) 51

6.9 The speedup of the tasks of the datapath in OpenMP on the platform I . 52
6.10 The speedup of the tasks of the datapath in OpenMP on the platform A . 52
6.11 The speedup of the RO of the datapath in OpenMP and IP4 on the platform I 53
6.12 The speedup of the RO task in OpenMP with different swath sizes 54
6.13 The speedup of the MA task in OpenMP with different swath sizes on 4 cores 55
6.14 OpenMP in RSARTE 56

7.1 The combined framework of IP4 and OpenMP 59

List of Tables

4.1 Parallel programming frameworks comparison 25

Abbreviations

WFP	**W**ide **F**ormat **P**rinters
SDF	**S**ynchronous **D**ata **F**low
IP4	**I**mage **P**rocessing **P**ipline **P**rallel **P**rogram
TBB	**T**hreading **B**uilding **B**locks
RSARTE	**R**ational **S**oftware **A**rchitect **R**eal **T**ime **E**dition

Chapter 1

Introduction

1.1 Background

1.1.1 Océ

Océ - Technologies B.V. is located in Venlo, the Netherlands. Océ was founded by Lodewijk van der Grinten and started as a producer of butter coloring in Venlo. After nearly hundred years, Océ has become a world leading company in printing and document management [1]. Océ produces high speed cut sheet printers (up to A3 paper size) as well as wide format printers (above A3 paper size) and related printing software applications. Océ was acquired by Canon in 2010 and became a company of the Canon Group.

1.1.2 Wide Format Printers

Wide format printers (WFP) are used to print large size images and often operate on a continuous sheet of printing media. In the WFP, the print head travels across the paper and consists of many nozzles which jet ink on the printing media. To perform that operation, the printer needs to transform an input file into the type of printing data which is a list of nozzles to fire at every fire moment. The datapath in the printers is the image processing pipeline that is responsible to perform that transformation. Because the print head is built from a number of arrays of nozzles for each color, the relation between the position of nozzles and the positions of pixels in the input image can be

rather awkward and the algorithms for manipulating the images are often not trivial. Figure 1.1 shows the printer Océ Arizona 350XT as an example of WFP.

FIGURE 1.1: Océ Arizona 350XT [1]

1.2 Problem Description

1.2.1 Problem

The datapath is a time consuming process so it is crucial to reduce its execution time to improve the performance of printers, especially for wide format printers where input images are large. Some methods can be applied to solve this problem like optimizing the datapath code or parallelizing the code on multi-core platforms to improve its performance. This thesis focuses on exploring the parallelization on the multi-core platforms. We are looking for a multi-core programming framework, which can help to parallelize the datapath of wide format printers on the multi-core platforms to improve its performance. The framework should be generic for wide format printers.

Problem statement: *"To explore suitable multi-core programming frameworks for the image processing pipeline in wide format printers"*

1.2.2 Approach

Given the defined problem in the previous section, this section explains about the approach which is employed in this project. The approach starts with defining the search space which consists of some multi-core programming frameworks. Based on previous works, available documents, the datapath and pre-defined criteria, the comparison among the frameworks is performed to reduce the size of the search space. From the reduced search space, the next stage proceeds with checking the feasibility by implementing, measuring and analyzing. From the analyzed result, some updates may be needed. Finally, the suitable framework is selected based on the results of the previous steps. Figure 1.2 shows the summary of the approach. The scope of this project is limited to the datapath of wide format printers.

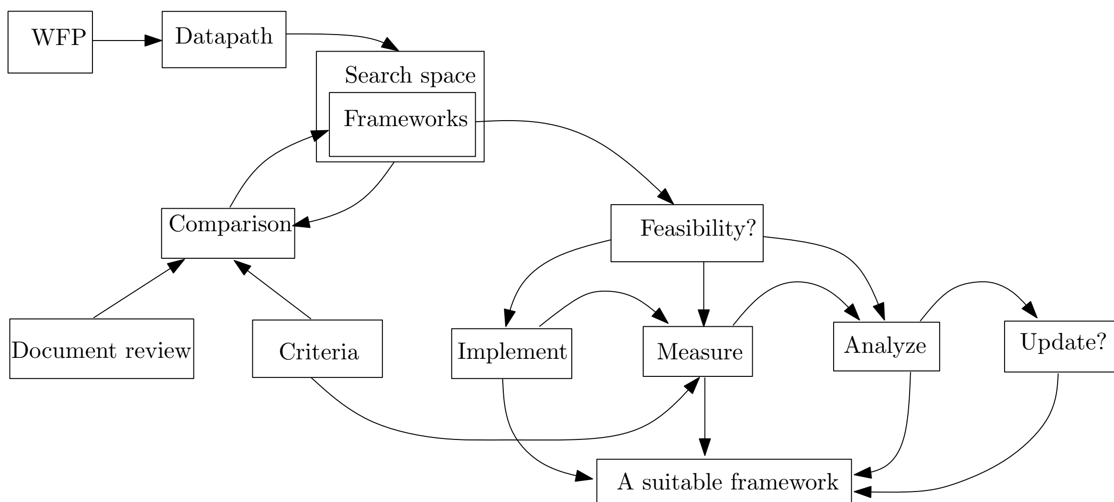

FIGURE 1.2: Project approach

1.3 Report organization

This report is the result of the master graduation project. Chapter 1 gives readers the overview about the project with the background information and problem statement as well as the approach which is used to solve the problem. Chapter 2 presents the state of the art of parallel programming frameworks. Chapter 3 describes the datapath in printers and the datapath in wide format printers specifically. Chapter 4 is about the first part of the approach, when all the alternatives are evaluated by reviewing available documents based on some pre-defined criteria. Chapter 5 and 6 evaluate the two selected alternative frameworks. The conclusion in chapter 7 explains the main findings of the project, what can be learnt from it and the suggested future work.

Chapter 2

State of The Art

"A dwarf on a giant's shoulders sees farther of the two"

George Herbert

2.1 Multi-core Processors

Single core processors were still a key solution between 1980 and 2000 when it has more or less followed Moore's law, which states that "the number of transistors that can be placed inexpensively on an integrated circuit double approximately every two years" [5]. However, it has lost its pace, starting from the beginning of the 21st century. From 1986 to 2002 the performance of microprocessors increased 50% per year on average, but it has dropped to 20% since 2002 [6]. Multi-core processors have been introduced as a better solution to improve the performance of processors and they have become popular in the market. In fact, Intel has 95% of all the processors which it ships by 2011; contain a multi-core design [5]. On the other hand, multi-core processors raise a challenge to program parallel applications which can utilize their resources effectively. Parallel programming is much more difficult to design, implement, debug and maintain than sequential programming. Fortunately, some parallel programming frameworks or models have been developed to make programmers' life easier.

2.2 Parallel Programming Frameworks

With the development of multi-core processors, parallel programming frameworks or models, which support for the software development on multi-core platforms, have also been developed and studied. OpenMP [7] was implemented and evaluated on a high performance embedded multi-core MPSoC in Chapman et al's work [8]. Intel Threading Blocks (TBB) [9] was studied and tested on both a real hardware and software simulation and its overheads are characterized in Contreras et al's work [10]. Cilk++ [11] was studied as a concurrency platform in Leiseron's work [12].

Besides studying the parallel programming frameworks separately, there are also some research works which compare the frameworks and conclude their advantages and disadvantages. For example, Message Passing Interface (MPI), Unified Parallel C (UPC) and OpenMP were investigated and evaluated with the NAS Parallel Benchmarks (NPB) in Mallon et al's work[13]. Another similar work is the comparison of OpenMP, Pthreads and Grand Central Dispatch (GCD) to parallelize face detection and automatic speech recognition in the paper of Deepak et al [14]. Wahlén compared some different parallel programming models for multi-core processors like OpenMP, Cilk++ and Pthreads in his thesis [15] with his pre-defined criteria and benchmark applications. Nevertheless, Wahlén did not specify the reason why these models are chosen. In many existing works, only a small set of choices is introduced and usually at the same level of parallelism. However, in the practical world, software designers are not limited to a small set of choices. Focusing on a small set of choices can cause the software designers make an inaccurate decision. Jarp et al. also used the parallel frameworks (OpenMP, Cilk Plus and TBB) to achieve the coarse-grained parallelism in vectorization and parallelization of a maximum likelihood data analysis application in [16]. However, their work emphasize on the algorithm optimization instead of the parallelization. At the design level, there are also tools which support Synchronous Data Flow (SDF) graph analysis and mapping tasks on the multi-core platforms like SDF3 [17].

In reality, the decision of choosing the most suitable parallel programming framework for a specific application is not a trivial task. The parallel programming frameworks have their own advantages and disadvantages and they all co-exist as Voss mentioned in [18]. Most of the existing works are conducted on a small set of the parallel programming frameworks with a generic benchmark code and mostly focused on the execution time and the speedup. The closest work is the work of Wahlén [15]; however, it still overlooks

the restriction of the hardware which is one of the most important characteristics of embedded platforms.

In this work, a more practical approach is proposed to choose the most suitable parallel programming framework for a specific application from a large pool of choices based on literature review, implementation and analysis. Other criteria beside the execution time, including the restriction of the real implementation hardware, are also taken into account in this assignment, which is rarely emphasized in the existing works. This work is conducted in the image processing pipeline of wide format printers' context, but it can be applied across the contexts in the parallelization on multi-core platforms domain. Moreover, one of the parallel programming frameworks, which is studied in this work is self-developed and used in the company for a specific application. Therefore, it is also interesting to evaluate whether a specific framework in one company scope is better than more generic frameworks. This aspect is also novel compared to the existing works.

2.3 Performance Modeling

2.3.1 Amdahl's law

Modeling the performance of a parallel application is an interesting topic which attracts many researchers. By modeling the performance, we can understand more about the behavior of the parallel application. The speedup is often used to assess the performance of the parallelization. In other words, it shows how much performance it gains after the parallelization. The speedup is calculated by Formula 2.1 below:

$$S = \frac{T_s}{T_p} \tag{2.1}$$

Where S is the speed up, T_s is the execution time of the sequential code, T_p is the execution time of the parallel code.

Predicting the speedup of the parallelization as S(N) = N, where N is the number of cores, is too optimistic for most of the cases. It is dependent on many factors, which make the modeling work more difficult. G. M. Amdahl presented his concern about the negative factors which can affect the parallel computation in the AFIPS conference in 1967 [19]. One part of his work is formulated as Amdahl's law which shows in Formula

2.2. Amdahl's law is used to predict the maximum speedup of the parallel application.

$$S_a(N) = \frac{T_s}{T_p} = \frac{T_s}{(1-f) * T_s + \frac{f}{N} * T_s} = \frac{1}{(1-f) + \frac{f}{N}} \tag{2.2}$$

Where $S_a(N)$ is the Amdahl's law speedup with N processors, f is the fraction of the program which is indefinitely parallelizable.

When $f = 1$, the speedup is N, which is the perfect speedup. When $N \to \infty$, we have $S(N) = \frac{1}{(1-f)}$, which is the maximum speedup in Amdahl's law. According to this, the speedup of the parallelized program cannot exceed $\frac{1}{(1-f)}$, regardless how many processors are used. Figure 2.1 shows the maximum speedups according to the Amdahl's law.

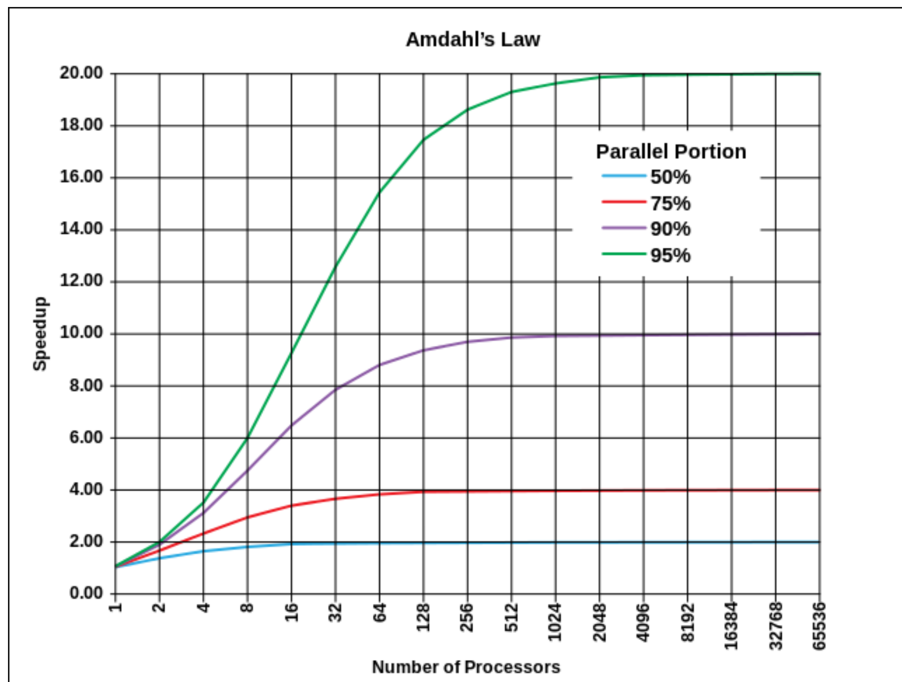

FIGURE 2.1: The maximum speedup according to the Amdahl's law [2]

2.3.2 Overheads

Amdahl assumed that if the part of the code is parallelizable, it can be parallelized perfectly. It makes Amdahl's law still optimistic for most of the real cases. The fact that it is too optimistic, is due to the overhead of the parallelization. There are research works which measure and analyze the overhead of the parallelization. J. M. Bull presents the measurement and analysis of synchronization and scheduling overheads in OpenMP in

his work [20]. He concludes that the overheads are significantly dependent on the used parallel programming directives and the platforms.

Other works focus on analyzing the overheads by dividing it into smaller categories. K. Fürlinger and M. Gerndt have categorized the overheads of OpenMP applications into four categories: synchronization, load imbalance, limited parallelism and thread management [21]. The synchronization overhead occurs when threads need to communicate with each other's. The load imbalance overhead is due to the different amount of work allocated to threads which creates idle waiting time. The limited parallelism overhead is from the critical section in the code where only thread can execute. Thread management overhead arises when threads need to be created and destroyed. Another similar but more generic work has been done by M. Roth et al. In their paper [3], they stated the performance of parallel applications depending on three factors: work, distribution and delay. The work is defined as the time for executing the actual work. The distribution is the time spent on distributing the workload across the cores and any related overheads. The distribution is divided into three smaller sub-categories: scheduling overhead, serialization and load imbalance. The scheduling overhead is identified as creating threads, mapping tasks to threads and allocating the workload across threads. The serialization overhead results from serial sections in the code. Load imbalance is described as the idle time because the unbalanced workload distribution. The performance of parallel applications is also affected by the availability of the resources, which is expressed in delay overheads. The delay is divided into two sub-categories: software and hardware. The software delay results from the time spent in synchronization, e.g. lock, or re-executing aborted software transactions. The hardware delay may come from the contention for resources in hyper-threading, cache misses or communication latencies which is defined as memory subsystem. Figure 2.2 shows the deconstruction framework in M. Roth et al's work. M. Roth et al are able to describe the overheads into more details compared to the work of K. Fürlinger and M. Gerndt and take into account the hardware dependency like memory access, cache sharing... However, these works do not give any model to support quantitative analysis of the overheads. They also do not mention about the overhead which is related to the parallel programming framework itself like calling functions in external libraries, loading Dynamic-Link Libraries (DLLs), parsing configuration files, etc.

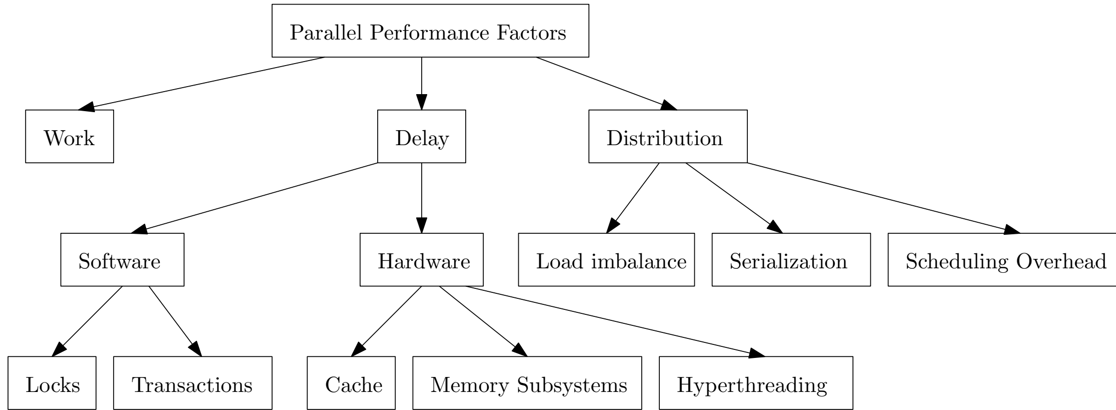

FIGURE 2.2: Deconstruction framework [3]

There are some works which try to build models for estimating overheads of parallelization on multi-cores platforms. N. Larsgård has proposed a model to estimate the overheads of OpenMP loops which only focus on the communication cost between threads on different processors [22]. The result of this work is limited since the overheads contain more factors than just the communication cost. The model of Larsgård is shown in the formulas 2.3:

$$\begin{aligned} T_o &= T_{comm} + T_{threads} \\ T_{comn} &= \frac{i}{c}(T_s + \beta * c * sizeof(datatype)) \end{aligned} \tag{2.3}$$

Where T_{comm} is the communication overhead, $T_{threads}$ is the overhead from spawning, destroying and switching between active threads. i is the number of iterations, c is the chunk size, T_s is the latency of the respective cache level or memory, β is the bandwidth between the processors.

To be able to model the speedup accurately, the overhead of the parallelization needs to be added into the Amdahl's function. According to M. Roth's categories, the serial section in the code is included into the distribution overhead. However, there are two types of serial sections: one is the serial part of the code which is not parallelizable and one is the critical section in the parallelizable part of the code but it can be executed by only one thread. The first type of serial section is reflected in the Amdahl's law as the fraction $(1 - f)$. Thus, the overhead should only include the second type of serial sections.

To accomplish a quantitative model, the overhead is broken down into small parts based on their quantitative characteristics. There are some overheads which increase when

the number of threads increases. For example, the more threads it has, the more time it spends to create and destroy the threads. The synchronization between threads also increases when the number of threads increases. In the critical section, the more threads have to wait if the number of threads increases. We assume these overheads are linearly related to the number of threads. There are other overheads which do not vary with the number of threads. For an instant, the static scheduling which allocates a fixed workload for each thread before executing them or the startup time of the parallelization tool, is likely constant. Besides that, the overhead is also dependent on the number of available cores to execute parallel applications. The overhead increases when there are more threads running on a core due to the resource contention. In this case, we only consider when the number of threads is equal or more than number of cores. If the number of threads is smaller than the number of cores, it is considered as equal to the number of cores.

We propose the new formula to express the parallelization overhead:

$$T_o(N_c, N_t) = (a * N_t + b + c * max(0, N_t - N_c)) * T_s \tag{2.4}$$

Where T_o is the overhead of the parallelization with N_t threads on N_c processors, T_s is the execution time of the sequential code; a, b, c are coefficients which are dependent on the application.

So the speedup with the overhead can be expressed in Formula 2.5:

$$S(N_c, N_t) = \frac{1}{(1 - f) + \frac{f}{N_c} + (a * N_t + b + c * max(0, N_t - N_c))} \tag{2.5}$$

The verification step is to evaluate the correctness of the hypothesis with the real results from the experiments. The closer the hypothesis results to the real result are, the more accurate the hypothesis is. Matlab provides the Curve Fitting toolbox which can determine the coefficient in the function to achieve the best fit with input values. By utilizing this toolbox, the coefficients a, b, c in Formula 2.5 can be determined by the experimental results. The parallel factor f can be determined by measuring the code outside the parallel code section. The coefficient of determination R-squared (R^2) [23] can be used to evaluate the correctness of the hypothesis function. R-squared ranges from 0 to 1 and R^2 of 1 indicates the perfect fit.

Chapter 3

Datapath

The datapath in printers is the image processing pipeline which performs the transformation from an input file (e.g., a postscript file) into the printing data which is a list of nozzles to fire at every fire moment. This chapter explains about the datapath in wide format printers where the input images are large and the print head has to travel across the printing media. Understanding the datapath is necessary to parallelize it on multi-core platforms with any multi-core programming framework.

3.1 The Datapath in Wide Format Printers

Due to the large size of the input image, it is printed swath by swath in WFP. One swath is one movement of the print head in the horizontal direction. Figure 3.1 shows a simplified image of the print head, nozzles and swaths. The printing data of the swath must be ready before it can be printed, i.e. the datapath must finish its process for the swath before printing it.

The input image, e.g., a postscript or a portable document format (pdf) file, is first transformed into a bitmap file. However, that part of the datapath is out of the scope. In this project, only the part of the datapath from the bitmap file to the swath data is studied. After the swath data, there are some other steps, but they are omitted in this project. The related part of the datapath consists of a number of tasks that we work on: SE, MG, MA, NFC, YC, RO, SC. Each step operates on a bitmap or a swath and transforms the data in a specific way. Figure 3.2 shows the datapath and the part which

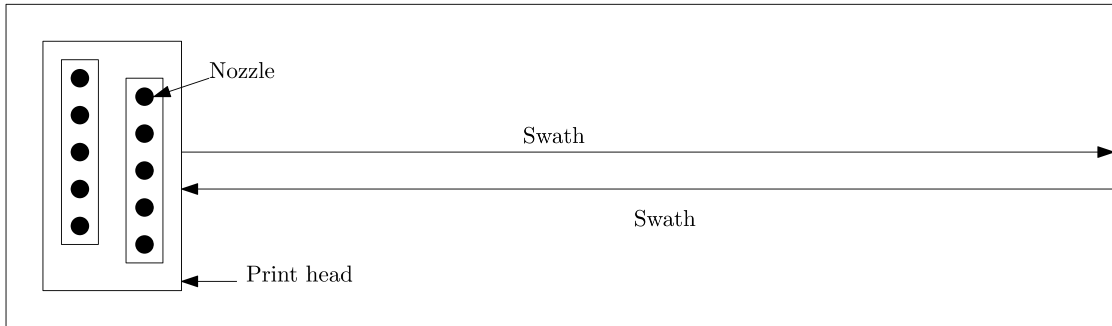

FIGURE 3.1: Swaths in wide format printing

is studied in this project. From this point for convenience, the datapath mentioned in this report refers to only the related part in the whole datapath. The details of the datapath tasks are described in Appendix A.

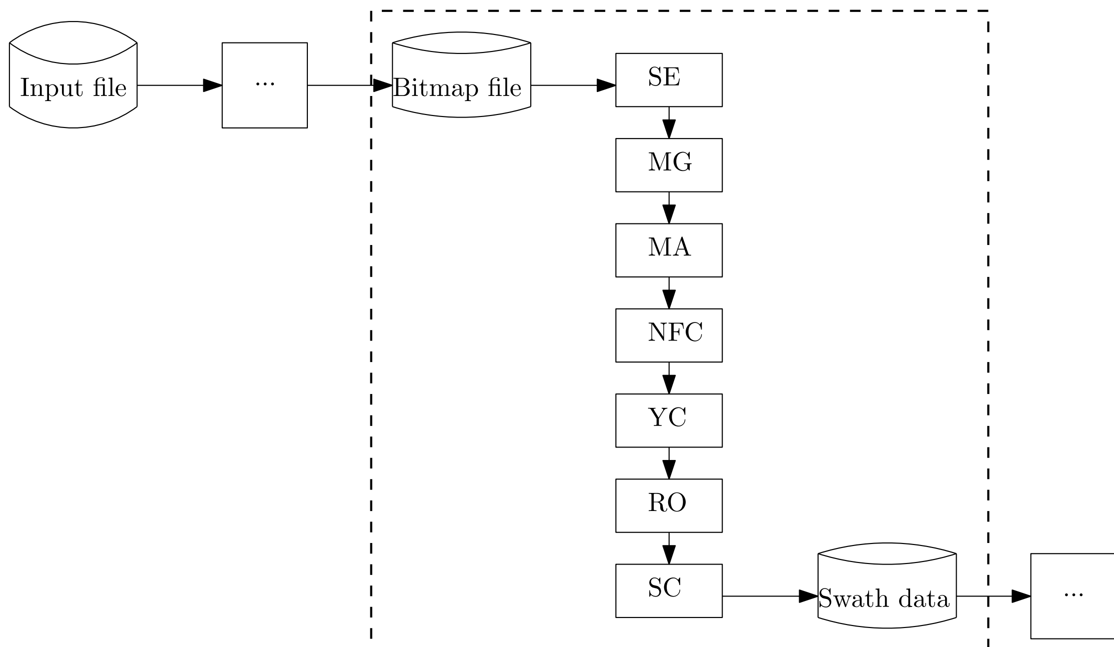

FIGURE 3.2: The datapath of wide format printers

In real printing cases, the SE task, the MG task and the MA task can be done for the next swath while printing the current swath. However, the rest of the tasks can only be done after printing the current swath, i.e., during the print head turning time because the NFC task can only get its input after printing the current swath. Figure 3.3 illustrates this printing process. SE_i, M_i, NFC_i, YC_i, RO_i, SC_i are the SE task, the MG task and the MA task, the NFC task, the YC task, the RO task, the SC task for swath i respectively.

Therefore, parallelizing at the swath level, i.e. processing a number of swaths at the

T_p T_t T_p T_t

SE2 M2 NFC2 YC2 RO2 SC2 SE3 M3 NFC3 YC3 RO3 SC3

Printing swath 1 Print head turns around Printing swath 2 Print head turns around

T_p is the printing time T_t is the turning time

FIGURE 3.3: Scheduling tasks of the datapath in the real printing case

same time, is not beneficial in this case. The SE task, the MG task and the MA task can be processed for future swaths but it does not improve the throughput since the NFC task needs to wait until the previous swath has been printed. The only benefit of processing these tasks for the future swaths is at the beginning of the printing process when it needs to wait to have enough swaths to process the NFC task for the first swath. The NFC task is the most complex task in the datapath. It does not only require the data from current swath, but also from some future swaths. For example, NFC may need the current swath and 4 future swaths to process the compensation in the current swath. So at the beginning, it needs to wait to have 5 swaths ($N = 5$) and then after that it only need receive one more swath to process the next swath. Figure 3.4 shows an example of the required swath for NFC in each swath, i.e., NFC in swath 1 requires swath 1 to 5.

NFC in swath 3
NFC in swath 2
NFC in swath 1
1 2 3 4 5 6 7 8 9

FIGURE 3.4: An example of required swaths for NFC

Parallelizing at the tasks level, i.e. processing the same task on different parts of the swath at the same time, is more beneficial. The printing time is much longer than the turning time, so reducing the execution time of the tasks in the turning time is more

critical. The RO task is the most time consuming task so parallelizing this task has the highest priority.

3.2 Describing The Datapath with SDF

The datapath diagram which is shown in Figure 3.2 can only express the tasks and the order of execution. However, it cannot illustrate the dependency among tasks. To be able to show the dependency in the datapath, a better modeling approach is chosen. Synchronous data flow (SDF) can be used to model the datapath. SDF graph consists of nodes which are called actors and edges which are called channels. The actors correspond to the tasks of the application. In this example, the functional steps of the datapath can be modeled as actors in SDF graph. The channels correspond to the data dependency and execution order. The numbers on the channels specifies the number of tokens each actor needs to consume before firing and the number of tokens it produces after firing. The channels can carry an infinite number of tokens.

As shown in the SDF graph of the datapath in Figure 3.5, the MA actor is dependent on the SE actor and the MG actor. The MA actor must receive one token from the SE actor and one from the MG actor to produce one token at the output channel. To model the NFC task, one self-loop is introduced on the NFC actor. NFC actor produces N-1 tokens to the self-loop and consumes N-1 tokens from the self-loop by each firing. N is the number of swaths which NFC requires. In the self-loop channel, there are N-1 initial tokens which are necessary to start. The channel from the SC actor to the NFC actor expresses that the NFC task can only be done when the previous swath has completed the scrambling task. However, another condition to execute the NFC task is the completion of printing the previous swath. Thus, a timer actor is added into this SDF graph. Timer actor fires every time after printing one swath. This SDF graph does not express the start of the process when the NFC task has to wait for first N swaths from the MA task to start processing the first swath.

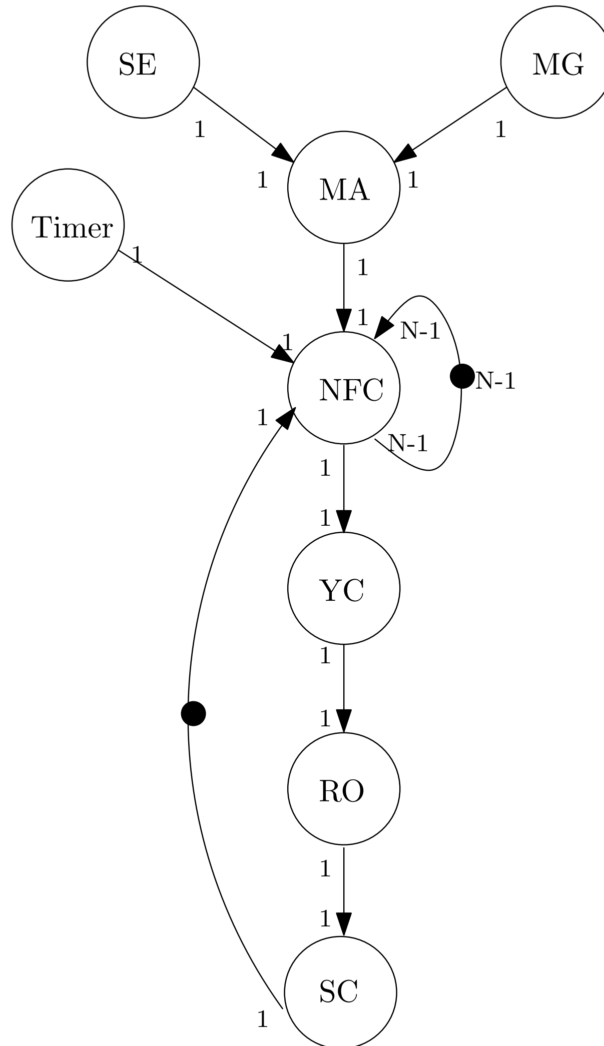

Figure 3.5: Synchronous data flow graph of the datapath of WFP

3.3 Chapter Summary

In this chapter, the datapath in WFP has been introduced and explained in detail. It gives an inside view of the application and the idea of which part we should focus on in the parallelization. Furthermore, the SDF graph of the datapath is able to describe more information such as the dependency among tasks. The datapath mentioned in this chapter is the generic datapath for WFP. In the next chapters, the MG task is only executed one time then the output is used for all swaths so it is omitted in the pipeline. The SC task is combined with the RO task and only referred as the RO task.

Chapter 4

Alternatives of The Datapath Parallelization

In this chapter, several multi-core programming frameworks are introduced and compared among each other based on their available documents. The strengths and weaknesses of each framework can be revealed by comparing it with other alternatives according to some criteria. Based on this comparison, the more suitable tools are selected to be implemented and analyzed in the next step. This is the search space reduction step in the approach.

4.1 Alternatives'Description

Selecting the alternatives for the preliminary comparison is based on three conditions. Firstly, the interest of the company in using the tools needs to be considered. For example in this case, IP4 draws more attention since it is an internal tool and can be reused from other projects. Secondly, the alternatives need to cover a broad range of parallel programming frameworks. To be able to form a collection of parallel programming frameworks, which covers a broad range, we have to understand its categorization. Ajkunic et al classified parallel programming frameworks into three categories: threading models, directive based models and tasking models [24]. Threading models are based on low level threading libraries. Directive based models provide high level compiler directives for parallelizing applications. Tasking models focus on specifying tasks instead

of controlling threads. The last condition is the alternatives should be available, mature and well-documented to be able to directly use to implement our application. Based on these conditions, the following collection of the parallel programming frameworks is chosen, which consists of native threading packages, OpenMP, SDF3, IP4 and Threads Building Blocks (TBB). Native threading packages are categorized as threading models, OpenMP as a directive based model, SDF3, IP4 and TBB as tasking models. SDF3, IP4 and TBB are in the same category but at different levels of implementation. While SDF3 focuses on the design level by supporting synchronous data flow graph analysis and task mapping, IP4 provides its templates to design and develop image processing pipelines on multi-core platforms and TBB provides a library for parallelizing applications at the task level. All the selected alternatives are explained in details below.

4.1.1 Sequential code

This is the sequential code of the datapath, which is only executed on a single core. It is still considered as an alternative since if there is no suitable framework to parallelize the code, the sequential code is the better way to implement the datapath. The datapath code is implemented from the provided algorithms. The algorithms are written in Matlab and the implemented code is written in C++. The implementation is not straightforward since it has to convert the operations in Matlab which work at the array level to the operation in C++ which work at the bit level. In other words, a pixel is implemented as an element in an array in Matlab but as a bit in an integer in C++ code. It is also a task in this project to implement the datapath in C++ code.

4.1.2 Native Threading Packages

Native threading packages like Pthreads or Windows threads are categorized as threading models. The native threading packages provide libraries for parallel programming. However, they are at a low level of implementation where programmers have to take care of thread creation, thread synchronization, termination and other threading operations manually. Using native threading packages gives programmers more flexibility but also the responsibility to handle the threading operations themselves. It is relatively difficult to implement with the native threading packages when parallel applications are complex.

4.1.3 OpenMP

OpenMP is a specification for a set of compiler directives, library routines, and environment variables which supports shared memory multiprocessing programming in C, C++ and FORTRAN on multi-processors platform. OpenMP is developed and maintained by the OpenMP Architecture Review Board (OpenMP ARB) [7]. OpenMP helps developers to focus on parallelizing the application instead of taking care of low level threads creation and synchronization. OpenMP focuses on parallelizing at the code level so it keeps the code modification for parallelization at a low level.

4.1.4 SDF3

SDF3 is a tool which can generate SDF graphs and also perform the analysis on SDF graphs. It also can map application tasks on a target platform and be a part of the design flow. SDF3 is developed by the Electronic Systems (ES) group at Eindhoven University of Technology (TU/e), the Netherlands.

4.1.5 IP4

Image Processing Pipeline Parallel Program (IP4) is a prototype parallel programming framework which has been developed in Océ. IP4 has been developed in another context to support the implementation of image processing algorithms on multi-core platforms. IP4 is developed as templates which allow developers fill in their code and define their own input and output data types. IP4 also provides proprietary scripts which specify the user-defined input values and the execution order of tasks in a pipeline. In contrast to OpenMP, IP4 focuses on the data parallelism at the task level instead of the code level.

4.1.6 Threading Building Blocks

Threading Building Blocks (TBB) is a library using standard ISO C++ code which helps programmers to write parallel C++ programs and exploit the scalable parallelism on shared memory multi-core platforms. TBB has been developed and maintained by Intel. Similar to OpenMP, TBB tries to avoid the disadvantages of native threading

packages where threads are created, synchronized and destroyed manually. TBB is not dependent on compilers. TBB is categorized as a tasking model in which tasks need to be specified instead of threads and mapping tasks onto threads effectively is the job of TBB.

4.2 Comparison Criteria

Most of the researches on parallel programming frameworks focus on the execution time and the speedup. However, other criteria like complexity, flexibility, portability, etc. are also important and can be crucial points to decide whether the frameworks are suitable. In [15], Wahlén compares three parallel programming models which are Pthreads, OpenMP and Cilk++ based on his pre-defined criteria like run time, efficiency, portability, complexity, modification, etc. However, my thesis is looking at the different perspective. Wahlén compares those tools with a generic view where there is no specific application. He chooses the applications himself, which serves the best the purpose of comparison. While in this thesis, the datapath in WFP is the pre-defined application. The restriction of the platform, which is overlooked in Wahlén's work, is also taken into account at the implementation step in this project. Pthreads is chosen as a representative of native threading packages to have specific information in this section.

4.2.1 Execution Time

The execution time is defined as the wall-clock time between the start and the end of processing one swath in a whole or a part of the datapath. Wall-clock time is the actual time to complete the task, which includes not only the CPU processing time of the task but all the communication time and interrupts. The execution time should be evaluated in different test cases in which some tests can reflect the restriction of the hardware, i.e. limited number of cores.

The sequential code usually has longer execution time when running on multi-core platforms than the parallel code since it does not use all the cores. The native threading packages can give a good improvement in the execution time if they are used effectively since they give developers more flexibility to optimize the parallel code. It is shown in Ravela's work [25] when he executes the Matrix-Matrix multiplication, Jacobi Iteration

and Laplace Heat Distribution applications in Pthreads, OpenMP and TBB. Pthreads gives a better speedup although it takes much more effort to implement. Ravela also explores the trade-off between the performance and the development effort among the parallel programming frameworks. The development effort in term of time and human resource is not in the scope of this project but the complexity of the tool, i.e. how difficult to implement a parallel application with the tool, is one of the criteria. Deepak et al [14] also show the same result when they execute the face detection and automatic speed recognition in Pthreads and OpenMP. IP4 does not have any previous work which compares with other frameworks but it shows a good performance in other existing projects. Most of the documents of SDF3 focus on analyzing SDFG, exploring trade-offs, optimizing resources and mapping tasks on platform but not the exact execution time on a target platform. For example, in [26] Stuijk et al prove that SDF3 is used in the design flow which can reduce the resource requirements of an MPEG-4 decoder by 66% compared to a state-of-the-art design flow.

4.2.2 Portability

The portability of the tool is the ability to run on different environments, e.g. compilers, OS, target platforms. Obviously, the sequential code does not depend on the compilers and OS, i.e. it can be compiled by any C++ compiler on any OS. Native threading packages does not require the compiler support but the cross OS support [18]. For example, Pthreads is available on many Unix-like OS but not directly available on Windows OS. OpenMP is compiler dependent but it is available for many compilers on different OS [7] such as GCC, Visual Studio 2008-2010 C++, Intel C/C++/Fortran, etc. Wahlén shows in his work [15] that OpenMP has better portability than Pthreads. IP4 is tested with the Visual Studio compiler, Intel compiler on Windows and being developed for Linux environment. TBB does not requirement compiler support and cross OS support. SDF3 is available for both Windows and Linux but it is dependent on the tools which convert the output of the task mapping to the target code. These tools are available in the ES group at TU/e but they are highly target specific and only support some platforms, e.g. CompSoC platform, at this moment.

4.2.3 Flexibility

The flexibility in the datapath is the ability to update without spending a lot of effort when the datapath changes, e.g., updating some tasks or changing the order of the tasks in the pipeline without changing the whole code. The parallel code using the native threading packages may require a lot of effort to be updated, especially when the threading operations need to be changed. OpenMP and TBB are at a higher level of abstraction, where programmers do not need to care about the basic threading operations. Thus, it may take less effort to update the parallel code. IP4 with its structure and script can help to update the datapath with little effort. For an instant, the configuration of the datapath and the order of the tasks can be changed easily in the IP4 script if input and output data types are defined correctly. The SDF graph can be easily updated when the datapath changes and the mapping can be done again automatically with SDF3. However, the conversion from XML output file to the target code is not automatically done by SDF3 for all platforms.

4.2.4 Code Size

The code size in this case is defined as the number of extra lines of code (LOC) which needs to be implemented using the parallel programming frameworks. The sequential code is the base code which does not have any extra code. Native threading packages require more modification than IP4 and TBB since the developers need to handle all the threading operations by themselves. Ajkunic et al [24] show that Pthreads requires 33 extra LOC while TBB needs 12 extra LOC and OpenMP needs only 1 extra LOC to converting the Matrix Multiplication code (30 LOC) to a parallel version. In another work, Ravela [26] implements a parallel version of Jacobi Iteration code with Pthreads (217 LOC), OpenMP (118 LOC), TBB (155 LOC) and Laplace Heat Distribution code with Pthreads (346 LOC), OpenMP (219 LOC), TBB (212 LOC). To use IP4, we need to declare data types of inputs and outputs, to create scripts and some operations to handle the queues. So the code size will increase significantly when implementing a parallel application in IP4. The code size when using SDF3 is not determined since it is dependent on the target code generation tools.

4.2.5 Complexity

The complexity is defined as the required knowledge to implement the datapath; given the datapath definition. Programmers only need to know C++ to implement the sequential code of the datapath. For native threading packages, programmers need to understand multi-threading operations, native threading libraries and also implementation platforms besides C++ programming. In the work of Deepak [14], Ravela [25], Ajkunic et al [24], Pthreads is always more complicated to implement. Ravela states in his work that it takes 90 developing hours and 58 developing hours to implement the Laplace Heat Distribution parallel code with Pthreads and OpenMP respectively. For OpenMP, IP4 and TBB, programmers need to learn how to use their libraries or templates. For SDF3, the knowledge of SDF graph and task mapping for target platform is required.

4.2.6 Memory Usage

The memory usage is defined as the amount of required memory to process the same input image. It includes the necessary memory for the datapath and any extra memory which is required by the framework itself. The sequential code does not have to use extra memory to support any framework. With the native threading packages, programmers have the flexibility to optimize the memory usage. There are no documents in about memory usage when using OpenMP, TBB or IP4 (in the scope of this project). SDF3 take into account the memory optimization. It is stated in [26] that the design flow using SDF3 can reduces the memory usage of an MP3 decoder to 21% compared to other design flows.

4.2.7 Maintainability

The maintainability is defined as the required effort to maintain a parallel application, due to the application itself or the updates of the frameworks. The maintenance cost is proportional to the frequency of changes and the effort required updating the programs due to the changes. The change of the application at this stage is not possible to check. The update of the frameworks can be checked with the number of stable releases. OpenMP has 5 releases in 16 years, TBB has 9 releases in 8 years and SDF3 has 8

releases in 7 years, IP4 just has 1 release. However, some updates may not affect the application.

Sufficient and well-organized documentation is also the advantage for maintenance since it can help to reduce the effort to update the program, especially when it is maintained by different person other than the developer. Pthreads, OpenMP, TBB and SDF3 have good documentation. Wahlén also mentions in his work [15] that the information about Pthreads and OpenMP are well documented. IP4 has less documentation but it is an internal tool so it has the support from its developers.

4.2.8 Cost

The cost is the license cost of the framework if there is any. The license term is also taken into account. All of the alternatives do not have license cost. TBB required a commercial license to get the support from Intel but it has the same features with the free version. IP4 is an internal tool so it does not require a license.

4.2.9 Data Flow Control Support

The data flow control is the control of the input and output of each task in the datapath pipeline. This criterion is very specific for this project. For example, the NFC task needs the data from at least N swaths to execute and the data control flow is responsible to handle that condition. This criterion is used to judge the ability of the framework to support the data flow control implementation. IP4 provides a queue mechanism handling the data transfer, which can support the data flow control implementation. SDF3 with the SDFG, which expresses the data transfer as tokens, can be useful for implementing the data flow control. The rest of the alternatives do not support the data flow control.

4.3 Alternatives Comparison

The comparison is shown in details in Table 4.1. The information in the table is retrieved from the documentation of the tools. The criteria are not judged equally, i.e. some criteria are more important. For instant, the sequential code version is small in code

size, simple to implement, easy to maintain but it has long execution time, which is one of the most important criteria.

Tools	Sequential code	Native threading	OpenMP	IP4	SDF3	TBB
Execution time	-	+	o	o	ND	o
Portability	+	-	o	o	-	+
Flexibility	o	-	o	+	o	o
Source code size	+	-	+	-	ND	o
Complexity	+	-	o	o	-	o
Memory usage	+	+	ND	ND	+	ND
Maintainability	+	o	o	+	o	o
Cost	+	+	+	+	+	+o
Data flow control	-	-	-	+	+	-

Table 4.1: Parallel programming frameworks comparison
+: More suitable, o: Neutral, -: Less suitable, ND: Non-Determined

The sequential code version is basically a sequential C++ code of the datapath. Thus, it is necessary to implement this version before utilizing any parallel programming frameworks. Moreover, it can be used as a base reference to compare the performance of different frameworks.

IP4 has higher complexity level compared to OpenMP or TBB. It requires more code modification when converting from the sequential code to the parallel code and it has some extra files like scripts. However, it has an important advantage which is the fact that it offers more flexibility to update the datapath. The datapath may need to be changed for each printing job. Updating the tasks in the datapath pipeline or changing the order of the tasks is straightforward by using the IP4 templates and scripts if the interfaces between the tasks are compatible. For example, if the order of the tasks in the pipeline changes, only the scripts need to be updated if the IP4 tasks have the compatible input and output data types. In another side, OpenMP or TBB focuses on parallelizing at the code level by adding the keywords into the source code. Therefore, the OpenMP or TBB code needs to be updated accordingly when the pipeline changes. SDF3 also has that flexibility but in the last step it cannot generate the target code automatically for all of the platforms. The last step for the code generation has been done on some specific platforms like the CompSoc platform in the Electronic Systems group, TU/e but not for any specific available platform in Océ. Therefore, even though SDF3 has a strong analysis feature, it is unlikely a suitable choice in this case.

Both OpenMP and TBB require little code modification from the sequential code to

the parallel code. They focus on parallelizing the existing code with directives or APIs while IP4 needs to arrange the whole code into its own structure. Choosing between OpenMP and TBB or other parallel programming tools is not a trivial problem. There is no single solution that fits all needs and developing environments [18]. That is also the reason why all the tools co-exist. Native threads like POSIX threads, Windows threads or Boost threads still exist even though they have disadvantages compared to TBB or OpenMP. If the application is written in C++, TBB may fit better since it is the C++ template which is highly object oriented and easier to develop the code. However, if the application contains a lot if array processing and loops, OpenMP may be a better choice. While OpenMP focuses more on loop parallelism, TBB has more generic parallelism techniques. All in all, OpenMP is a better choice for the datapath of WFP which contains a lot of array processing and loops. OpenMP has a disadvantage, which is the fact that it is compiler dependent but it supports many compilers on different OS.
The parallel code using the native threading packages may be customized to yield high performance but it is relatively difficult to implement, debug and maintain the parallel code. Thus, the native threading packages are not a suitable choice in this case.
From this comparison, three options are chosen to implement in the next phase; which are plain C++, IP4 and OpenMP.

4.4 Chapter Summary

In this chapter, the first two steps of the approach have been presented: defining the search space of alternatives and reducing the size of the search space. Defining the search space is based on three conditions: the interest of the company, the broad range of search space, the availability and the maturity of the tools. The search space is formed with the sequential code, the native threading packages, OpenMP, TBB, IP4 and SDF3. Then reducing the size of the search space is based on some pre-defined criteria and the application. Finally, three candidates have been selected for the next steps: the sequential code, OpenMP and IP4. In the next two chapters, the details of the selected candidates will be explained.

Chapter 5

IP4

5.1 Introduction

Image Processing Pipeline Parallel Program (IP4) is a prototype parallel programming framework which supports the implementation of image processing pipelines. IP4 consists of three parts: IP4 scripts, IP4 core and IP4 tasks. The "task" mentioned here is a task in the pipeline. IP4 core is generic for all the pipelines. IP4 scripts and tasks are specific for each application. IP4 script files are written in a proprietary script language. The scripts contain the values of variables for initializing the tasks. It also defines the number of inputs and outputs of each task and the order of the tasks in the pipeline.

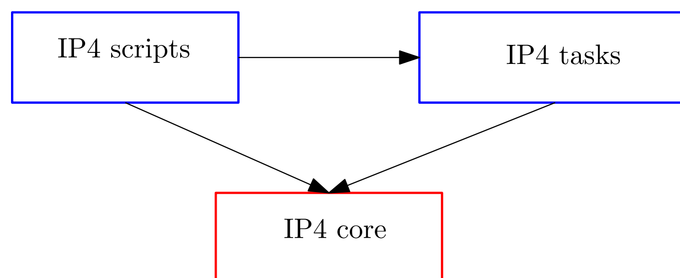

FIGURE 5.1: IP4 components

The logical view of IP4 is shown in Figure 5.2. The script files need to be loaded, parsed and analyzed to produce a valid intermediate structure. The tasks, defined in external modules (plug-ins) need to be loaded. The intermediate structures need to be transformed into an optimized structure for execution.

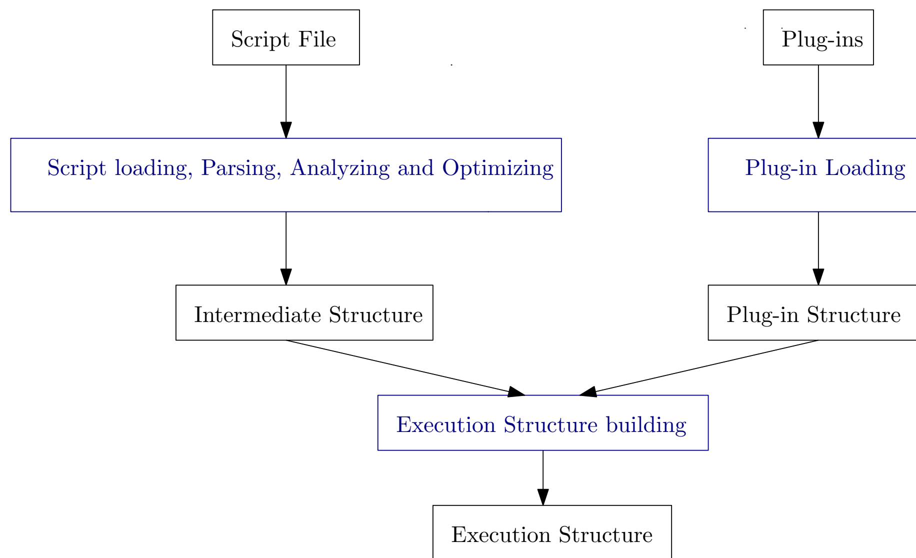

Figure 5.2: Logical view of IP4

Figure 5.3 shows an example IP4 script. The pipeline definition in the script is divided into two parts: the definition part and the execution part. In the definition part, the tasks are declared. In the execution part, it specifies the order in which the tasks are executed and the conditions to execute if there is any. The tasks need to be declared before they can be used. During the execution of the pipeline, the tasks read the input data, perform their operations and then produce the output data. The data transfer between the tasks is implemented as queues. To declare a task in IP4, the number of input and output queues, the task type and the task name need to be specified. For example, the task which is named *t2* is defined as the type *taskB* with 1 input queue and 4 output queues. The task types are declared in the tasks' code. The task *t2* has its variable *var1* with the assigned value 1 in this case. The variables are defined inside the tasks' code. The parameter $< fast >$ for the task *t3* is used to choose a specific implementation in the *taskC*. Variables and parameters are not compulsory to define a valid task in IP4. Moreover, IP4 supports to assign variables during executing the pipeline. In this example script, the variable X of the task *t2* is assigned the value of the variable Z of the task *t1*. It is useful when the assigned value can only be determined during executing the pipeline. It is also possible to set conditions in the pipeline, which allows altering the pipeline during the execution. For example, whether the task *t3* or *t3a* is executed is dependent on the condition $(t2.Y == 0)$.

```
Pipeline
{
#Definition part
    [00,01] taskA           t1;
        t1.input = "input1";
        t1.var2 = 2;
    [01,04] taskB           t2;
        t2.var1 = 1;
    [01,01] taskC <fast>    t3;
    [01,01] taskC <slow>    t3a;
    [01,01] taskD           t4;
    [04,01] taskE           t5;
    [01,00] taskF           t6;

#Execution part
    t1();
    t2.X = t1.Z;
    t2();
    if (t2.Y == 0)
    {
        t3();
    }
    else
    {
        t3a();
    }
        t4();
        t5();
        t6();
}
```

FIGURE 5.3: An IP4 script example

The graph in Figure 5.4 illustrates the pipeline in the example script in the case ($t2.Y == 0$).

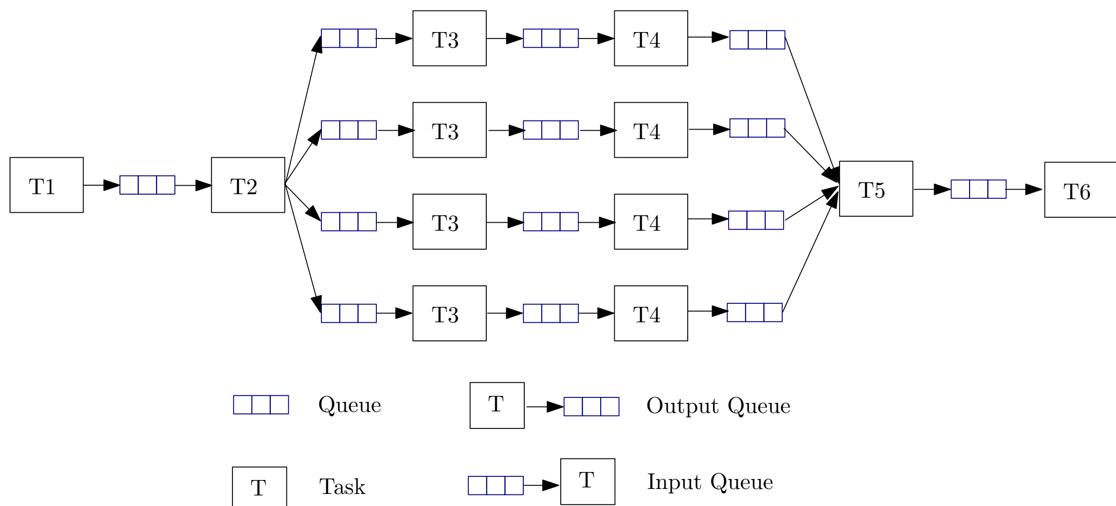

FIGURE 5.4: An example of IP4 pipeline

IP4 targets on the data parallelism in image processing pipelines. One advantage of the IP4 framework is the flexibility to update the pipeline. As mentioned, the datapath pipeline is not trivial for all of printers and even printing jobs. Thus, the ability to change the tasks without changing the whole pipeline is a desired feature. By using IP4 templates and scripts, the tasks in the pipeline or the order of the tasks can be easily changed.

There are four basic types of pipeline in IP4: create pipeline, extend pipeline, duplicate pipeline and join pipeline. Examples of each type of pipeline are shown in Figure 5.5. For extend pipelines, IP4 allows users to select which queue to output. For join pipeline, IP4 allows users to choose to process the task's code when all its inputs are available or at least one of its inputs is available.

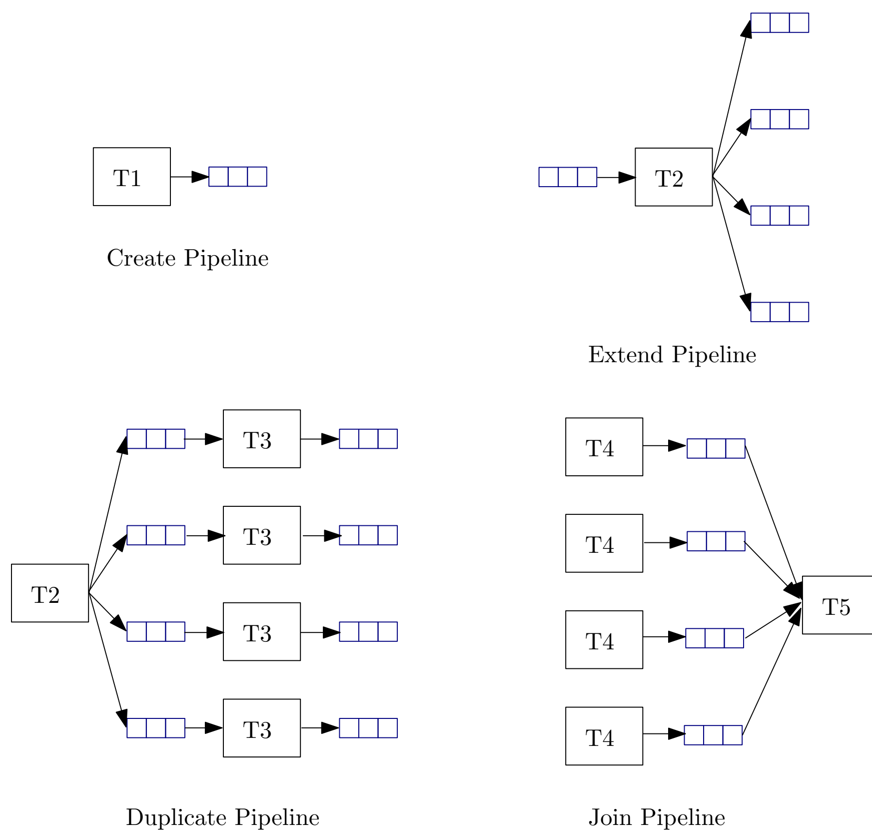

FIGURE 5.5: IP4 pipeline types

Figure 5.6 shows parallel pipelines which are on different threads. Threads are invisible for users. It is also the purpose of IP4 to keep developers away from threading operations and make them focused on their tasks.

The pipeline does not only contain the tasks, but also the operations. The operations are defined in the IP4 script. They can be variable assignments or conditions for the

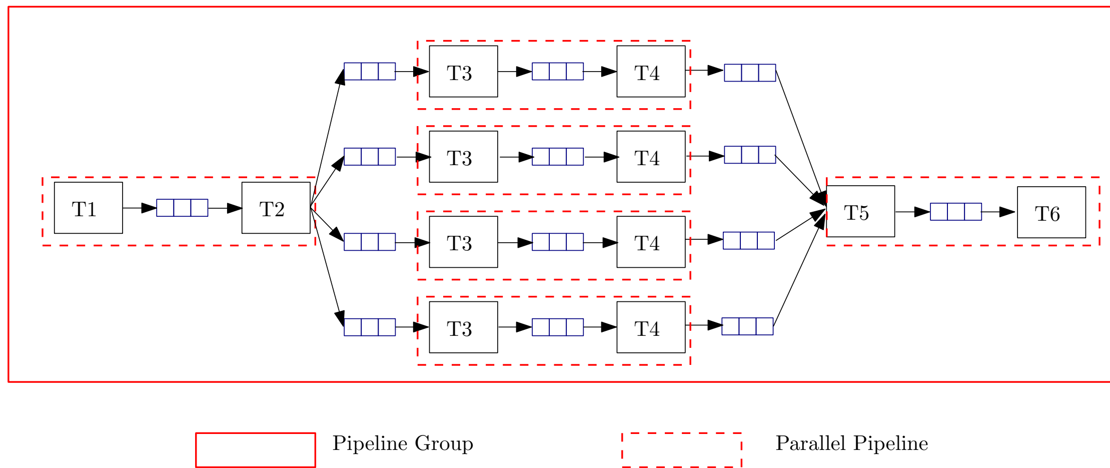

FIGURE 5.6: Parallel pipeline division in IP4

execution of tasks. The operations can depend on constants or variables. If they depend on the constants, they can be calculated before executing the tasks. However, if they depend on variables which are dependent on the execution of previous task, they have to be a part of the pipeline. The concept of execution unit is introduced to address the combination of task and operations. Basically, the execution unit consists of the task and the operations which depend on the previous task. The execution unit is illustrated in Figure 5.7.

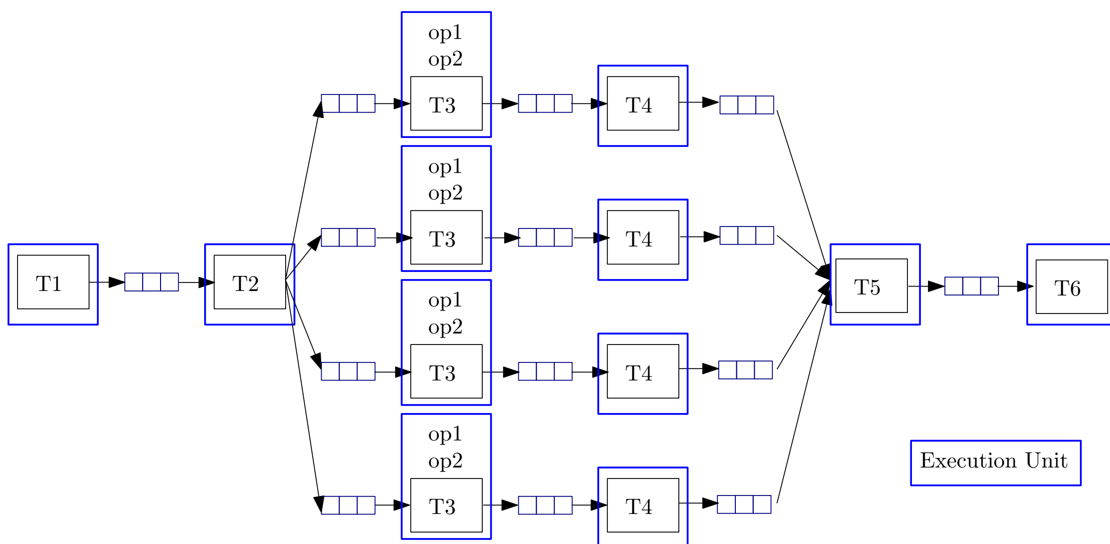

FIGURE 5.7: An example of IP4 execution unit which consists of task and operations

5.2 IP4 with The Datapath

The data in WFP can be implemented on IP4 by inserting the datapath code into IP4 templates. The tasks in the datapath are implemented as tasks in IP4. The code which is placed in IP4 tasks' template is similar to the serial code. However, the programmer himself has to take care of defining the input and output data types, deciding how to parallelize the code by creating output queues, allocating workload and handling data dependencies.
As mentioned in section 3.1, the RO task should be parallelized first to reduce the execution time during the head turning period. Figure 5.8 shows the IP4 implementation graph of the datapath in WFP. This graph does not show the dependency between tasks like the SDF graph in section 3.2 but it shows which part of the datapath is parallelized in IP4. The MG task is omitted in this implementation because only one mask is retrieved from a file and applicable for all the swaths. The SC task is also omitted because it is combined in the rotation task. The swath accumulation task is an additional task for IP4 which added at the end of the pipeline to collect the output from the RO tasks. Basically, each of RO task processes a different part of the swath.

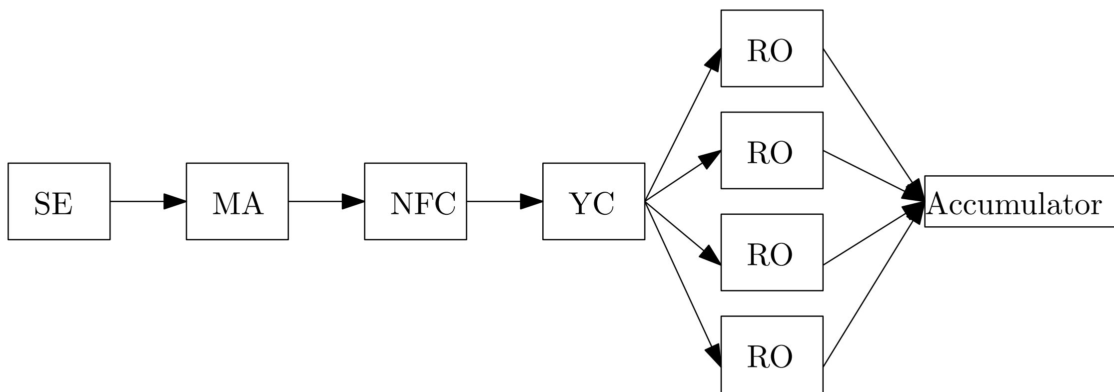

FIGURE 5.8: IP4 implementation graph of the datapath in WFP

5.3 Experiments

All the experiments in this project are conducted on two platforms:

- Intel Core i5-2400 CPU: It has 4 cores, 3.10GHz, 64KB L1 cache, 1MB L2 cache and 6MB L3 shared cache. This platform is referred as platform I.

- Freescale i.MX 6 Quad 1 GHz $ARM^{®}$ $Cortex^{TM}$ – A9 processor: It has 4 ARM cores, 1GHz, 32kb instruction and data caches, 1MB L2 shared cache. This platform is referred as platform A.

Platform I is a generic PC platform and platform A is an embedded platform. The number of cores which are available for executing the datapath on the platforms may be varied.
To get more accurate results, all the measurements are repeated several times and the final results are the average values, which are calculated by Formula 5.1:

$$\mu = \frac{\sum_{i=1}^{N} T_i}{N} \tag{5.1}$$

Where μ is the average value, N is the number of measurements and T_i is the value of each measurement. Besides that, the standard deviation is also calculated by Formula 5.2 to show the variation of the measurement. The standard deviation is plotted as the variation bars in the graphs, but sometimes it is too small to be recognized.

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(T_i - \mu)^2} \tag{5.2}$$

Each experiment is described in three parts: setup, results and analysis. In the setup part, the testing object, the platform and other details of the experiment are described. The result part shows the experiment result and important observations from it. The analysis part is used to explain about the reasons behind the observations. The timing in all the experiments is the wall time, which is the elapsed time between the start and end of the execution. To make sure the parallelization code correct, the output of the parallelized code version is written into a file and compared with the output of the sequential code version. Both of the outputs should be exactly the same to prove the functionality of the parallelized code version.

5.3.1 Experiment: the startup overhead of IP4

Setup:

- Test object: the datapath code in IP4 without any parallelization, i.e. executing on only one thread.
- Test input: one swath with 40,000,000 pixels.
- Test platform: Platform I.
- Test goal: To observe the startup overhead of IP4 by executing the datapath code in IP4 with only one thread and comparing the execution time with the sequential version of the datapath code. The startup overhead is the overhead which relates to the frameworks themselves like calling functions from external libraries, loading DLL files, parsing configuration information,etc.

Result: The IP4 version takes 364.54 ms to finish the job and the sequential version takes 356.76 ms to finish the job. The startup overhead of IP4 without any parallelization is about 2.18 %, which is calculated by using Formula 5.3

$$O(\%) = \frac{T_p - T_s}{T_s} * 100 \tag{5.3}$$

Where $O(\%)$ is the overhead percentage, T_s is the sequential version execution time, T_p is the IP4 version execution time.

Analysis: The overhead is not significant since IP4 loads only once all the needed files and parses the scripts at the beginning. In other words, it tries to execute as much as possible not during the run time.

5.3.2 Experiment: IP4 on the multi-core platform

Setup:

- Test object: the datapath code in IP4 with the parallelized RO task. The RO task is executed on 2 and 4 threads.

- Test input: one swath with 40,000,000 pixels.
- Test platform: Platform I. The number of cores which are available for executing the datapath code is limited to a specific number. The application can be set to run on a specific number of cores. To get accurate results, other running applications should be closed and the datapath code is executed at high priority.
- Test goal: to observe the performance of IP4 when the number of available cores is changed.

Result: Figure 5.9 shows the execution time of the RO task on IP4 with 2 and 4 threads on a different number of cores. IP4 shows the speedup when the number of cores increases. However, the execution time is much higher when it is forced to run on the smaller number of cores. It is even much slower than the sequential code which also runs on one core. Another observation is the variation of the execution time is large when it runs on a smaller number of cores. The variance becomes smaller when the number of core increases.

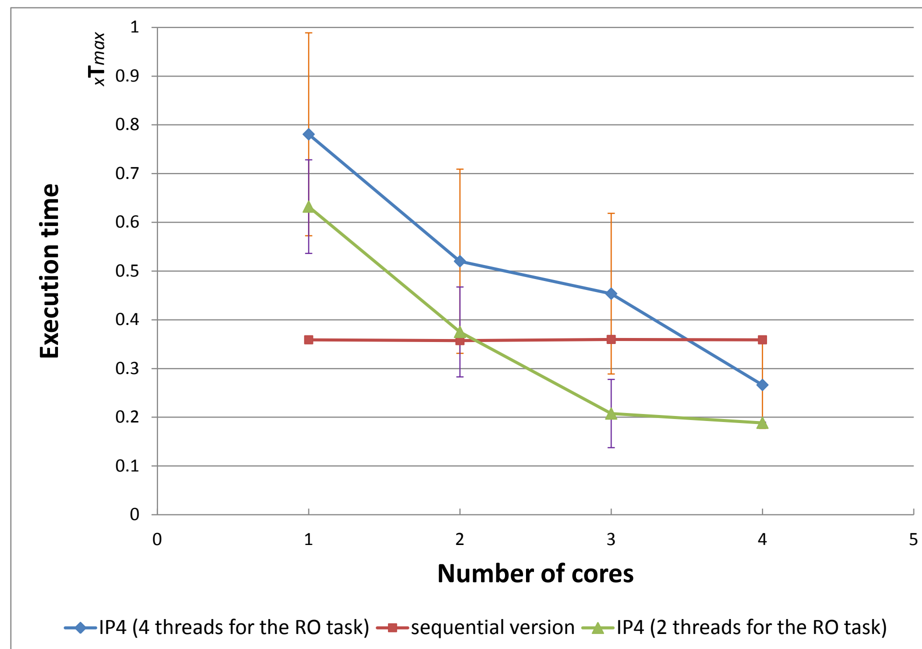

FIGURE 5.9: The execution time of the RO task in IP4 on a different number of cores of the platform I [1]

Analysis: Thread conflicting is a reason of the bad recorded performance of the datapath on a single core. IP4 creates the threads with an equal priority. Besides the threads

[1] $\mathbf{T}_{max}$ is the reference value of the execution time, which is used for all the execution time graphs in this thesis.

for the RO task, IP4 also creates other threads, i.e. one thread for the tasks before the RO task, one thread for the tasks after the RO task and one system thread. When all the threads have to run on the same core, they compete with each other to get the resource and it results in a lot of context switching. Furthermore, it causes the variance in the measured execution time. The variance is not significant when there are enough cores to execute the threads.

If the experiment speedups of the RO task on 4 threads are plugged into the formula 2.5 with $f = 0.99$, we get the coefficients $(a, b, c) = (0.1060, 0.0724, 0.2976)$ and the goodness of the fit $R^2 = 0.9678$. The coefficient c is dominant in this experiment so it shows the overhead is dependent heavily on the number of available cores. The overhead decreases significantly when the number of cores increases. It is also what we observe from this experiment result.

5.3.3 Experiment: Flexibility of the datapath in IP4

Setup:

- Test object: the datapath code in IP4 with a changeable order of the tasks.
- Test input: one swath with the different configurations like the swath sizes, the number of nozzles, the distance between nozzles, etc.
- Test platform: Platform I.
- Test goal: to check the flexibility of the datapath in IP4, i.e. verifying what needs to be updated when the configuration of the datapath changes.

Result & Analysis: The order of the tasks in the datapath can be changed in the IP4 script and the code does not have to be re-compiled. However, programmers need to take care of the compatibility between the input and output data types of IP4 tasks. It basically needs to satisfy the condition where the input data type of the task is the same with the output data type of the previous task, except the first task in the pipeline. The input data can also be changed by updating the IP4 script and the code does not need to be re-compiled, assuming that the variables of the input data are declared in the tasks' code and their values are retrieved from the script.

5.3.4 Experiment: Different input data sizes in IP4

Setup:

- Test object: the datapath code in IP4 with the parallelized RO task. The RO task is executed on 2 threads.
- Test input: one swath with the various swath sizes.
- Test platform: Platform I.
- Test goal: to check the effect of the input data size on the speedup.

Result & Analysis: According to the result as shown in Figure 5.10, the speedup of the RO task is not really affected by the swath size. When changing the swath size, the cache utilization may be changed. Technically, when the swath size is smaller, the processed data on each core is smaller and better fit in the caches. However, the RO task is more complicated and the cache utilization is not effective although the swath size is small. Thus, we do not observe a change in the speedup when the swath size changes.

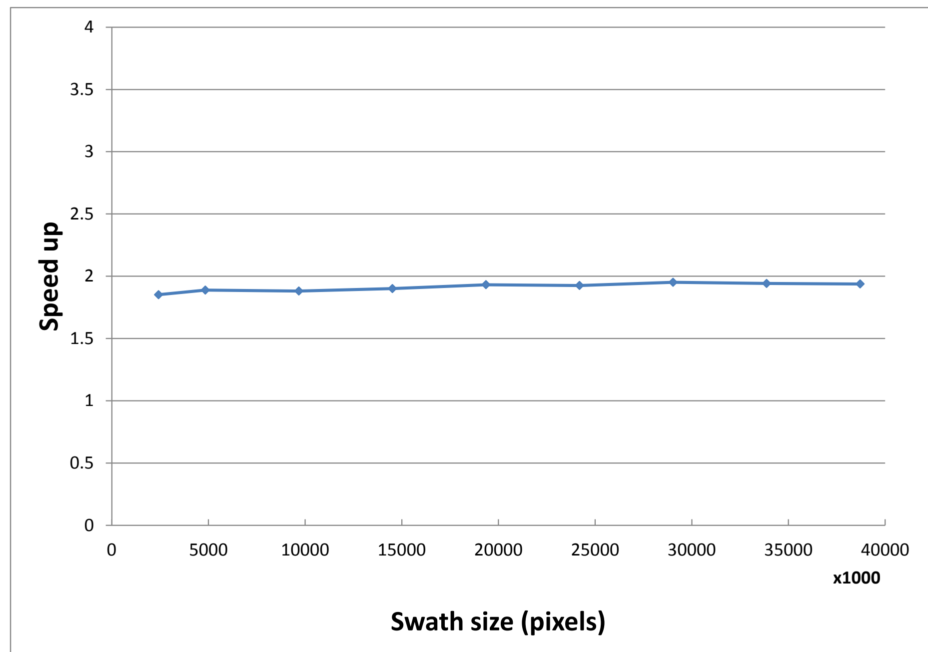

Figure 5.10: The speedup of the RO task in IP4 with different swath sizes

5.4 Thread Scheduling in IP4

The thread conflict problem which is observed in the experiment 5.3.2 can be solved by limiting the number of executing threads at the same time if the number of available cores is smaller than the number of threads. In this section, two methods are proposed to accomplish that. The first method is setting different priorities for threads. The second method is making the threads yield their time slices to other threads if necessary.
To understand the solutions, we need to understand the thread scheduling mechanism. When a thread is created, it is assigned a priority. When there is more than one thread, the system chooses the thread which has a higher priority to execute first. If the threads have the same priority, the system assigns time slices in a round-robin fashion to all the threads. The thread is preemptive, i.e. the system suspends the current thread and executes the higher priority thread which is ready to run. All the threads which are created by IP4 have the same priority. Other threads are also executed when the threads of the RO task are executed. Therefore, the system assigns time slices to the other threads as well. If the priority of the RO task threads is set higher than the priority of the other threads, the system only assigns time slices to the RO task threads. It is the first solution. Another solution is making the other threads yield their time slices to the RO task threads if the RO task threads are ready to run. To be able to verify these solutions, one test program is built with Windows C++11 threading library which is also used in IP4. In the test program, 6 threads are created and testing functions which contain loops are assigned to the threads. The purpose of this test program is to mimic the threading scheduling in IP4. The result of the test program is shown in Figure 5.11.

The similar problem as IP4 is observed when all the threads are set to the same priority. By adjusting the priority of 2 threads lower than the other 4 threads, the result is improved significantly. The yielding time slices solution also shows a good result but the adjusting priority solution is still better when the number of cores is 3 or 4.
The second solution has been implemented in IP4 and its performance is shown in Figure 5.12. The new scheduler also shows a good improvement compared to the original scheduler. Due to the time limit of this project, the first solution has not been implemented in IP4.

If the speedups of the new scheduler are plugged in Formula 2.5 with $f = 0.99$, we get the coefficients $(a, b, c) = (0.0315, 0.0682, 0.0278)$ and the goodness of the fit $R^2 = 0.9468$.

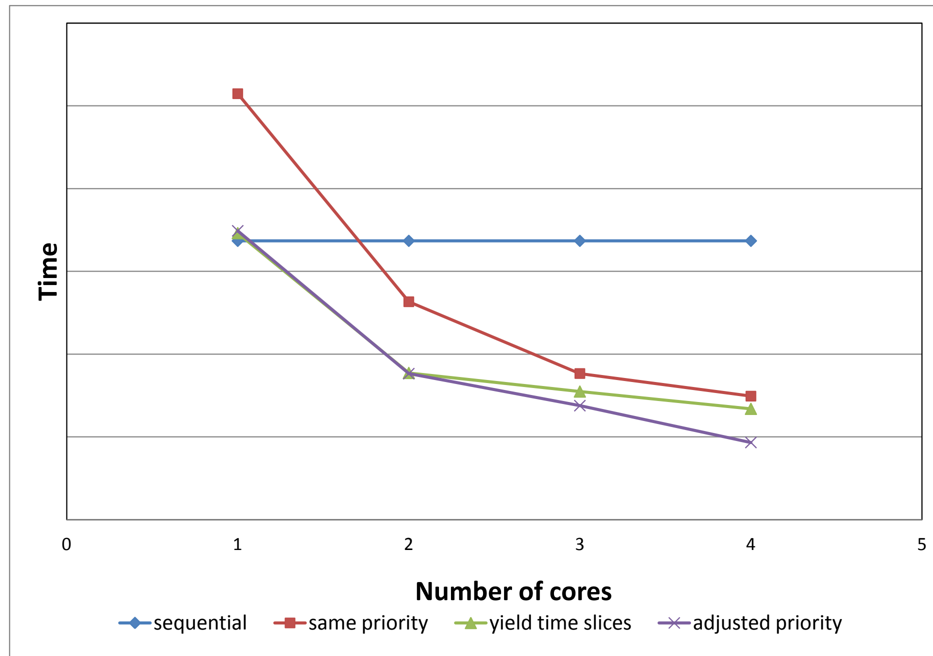

FIGURE 5.11: Thread scheduling in the test program

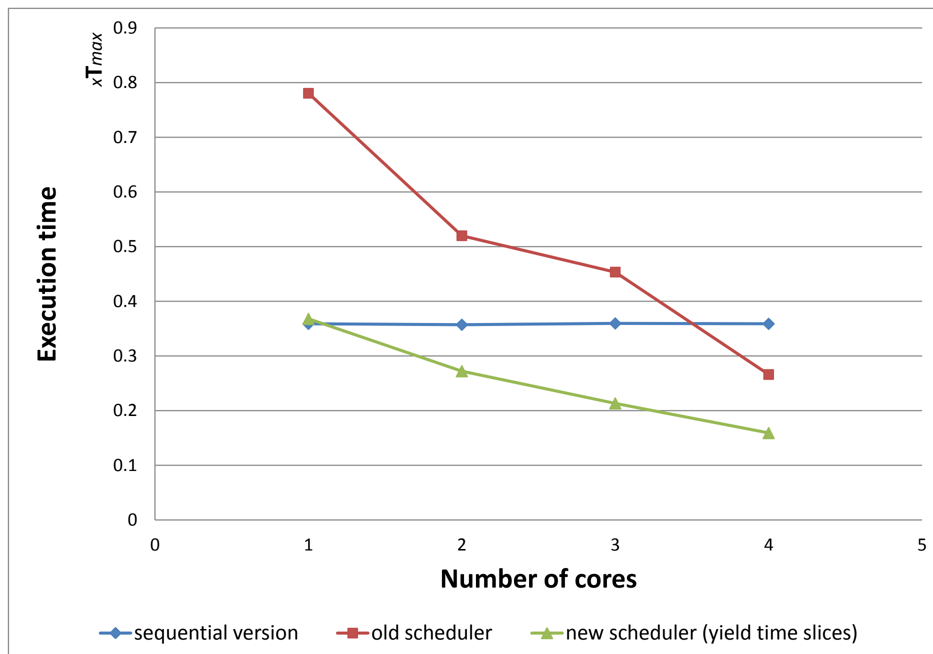

FIGURE 5.12: New thread scheduling in IP4

All the coefficients are smaller than the old scheduler case, especially the coefficient c. It proves the overhead is smaller and especially the resource contention overhead is reduced significantly.

5.5 Chapter Summary

In this chapter, the architecture of IP4 has been explained in detail. Some experiments have been performed to evaluate the performance of the datapath application in IP4. The startup overhead, the performance on the multi-core platforms, the flexibility of the datapath and the effect of the input size are checked. IP4 shows a small startup overhead when running without any parallelization in the experiment 5.3.1 but it has a significant overhead when running with many threads in the experiment 5.3.2. As verified in the experiment 5.3.1, the startup overhead of IP4 such as script parsing, DLL files loading, etc. is not significant since IP4 does most of these tasks when setting up the pipeline. The execution time reduces significantly when the number of cores increases and the variation of the measurements also reduces. Thus, the overhead in the experiment 5.3.2 is mostly from the thread conflicts when the threads with the same priority are forced to run on the smaller number of cores. In section 5.4, some solutions to the thread conflict in IP4 are proposed. IP4 provides its own scripts which can help to change the configuration of the image processing pipelines without re-compiling the code as shown in the experiment 5.3.3. The compatibility between IP4 and the developing environment, which is Rational Software Architect Real Time Edition (RSARTE) in this case, was investigated in another project. IP4 has been proved to be compatible with RSARTE by using an interface which has been developed separately.

Chapter 6

OpenMP

6.1 Introduction

During the second half of the 1990s, when multiprocessors became more and more popular, the OpenMP Architecture Review Board (OpenMP ARB) was established to provide a common framework for programming a broad range of Symmetric Multiprocessors (SMP) architectures. OpenMP ARB includes many well-known vendors such as AMD, HP, Intel, Fujitsu, Texas Instruments, etc. OpenMP is defined as a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level in Fortran and C/C++ program [7]. The first version of OpenMP for Fortran was published in 1997 and the OpenMP version for C/C++ was published in 2000. The current version OpenMP 4.0 was published in 2013. OpenMP is a widely adopted standard and it is well-maintained. Moreover, there are many researches on OpenMP with a large range of applications from embedded systems to super computers. The compilers need to support OpenMP to be able to compile the code with OpenMP APIs; however, OpenMP is supported by many compilers, including GCC, Microsoft Visual Studio, and Intel compilers.

The parallelized code can be created by inserting the OpenMP pragma directives in the sequential code. One advantage of this approach is keeping the modification level from the sequential code at a very low level. The OpenMP pragma has the common form as *# pragma omp name_of_directive [clauses]*[4]. Multithreading in OpenMP follows the Fork-join model which is shown in Figure 6.1. The master thread contains a series of instructions which need to be executed consecutively. At the beginning of the parallel

section, the master thread forks a number of slave threads and the workload is divided into the threads. At the end of the parallel section, the slave threads join the master thread.

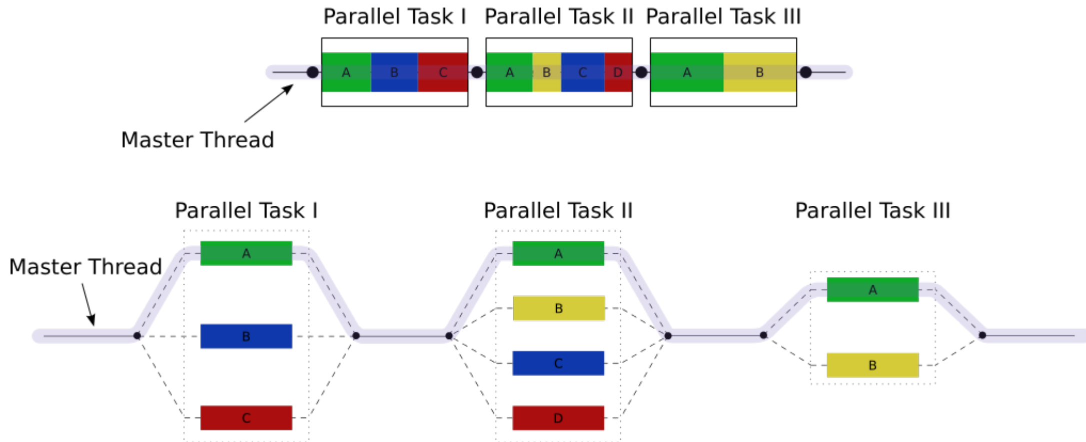

FIGURE 6.1: The OpenMP Fork-Join model [4]

Although OpenMP has simple pragma syntax, it is not trivial to deploy OpenMP directives into the sequential code. Some problems like race conditions and data dependencies need to be handled by the programmers themselves. An example of data dependencies is the dependency between iterations in loops. The programmers can solve this problem by rewriting the code and OpenMP also provides some feature to solve the data dependency for specific operations like the reduction feature for the operation increment. Race conditions occur when the correctness of the program depends on the sequence or timing of threads. They can cause invalid executions or unpredictable results. They can be solved by using OpenMP features like critical sections or locks.
Loop-level parallelism is one of the main features of OpenMP and it is very useful for application which has a lot of loop-level parallelism. The datapath code contains many loops which have many opportunities to be parallelized. OpenMP provides some loop schedulers which can be chosen by users. The types of OpenMP loop schedulers, which refer to the OpenMP specification 3.1 [27], are listed below:

- Static: the loop is divided into equal-sized chunks or as equal as possible then the chunks are assigned to the threads in a round-robin fashion in the order of the threads. By default, the chunk size is set to *(number_of_loop_iterations/ number_of_threads)*.

- Dynamic: the loop is also divided into equal-sized chunks or as equal as possible and the chunks are put into a queue. Each thread executes a chunk, then it requests for another chunk when it finishes that chunk. The process keeps going until there is no remaining chunk. The default chunk size is 1. The difference between static and dynamic scheduling is the number of loop iterations which is allocated to threads unequally. It can help to cope with the uneven distributed workload over the loop iterations.

- Guided: the guided scheduling has the same chunk division and chunk request mechanism like the dynamic scheduling. However, the initial chunk size is proportional to the number of unassigned iterations divided by the number of threads and then the chunk size decreasing to the user-defined minimum chunk size. The default minimum chunk size is 1. This scheduling mechanism can reduce the overhead of the dynamic scheduling but it limits the robustness of coping with the unevenly distributed workload.

- Auto: the auto scheduling is delegating the decision of choosing the scheduling types to the compiler and the runtime system. This scheduling type is only available from OpenMP version 3.0.

6.2 OpenMP with The Datapath

The OpenMP implementation version of the datapath in WFP has no significant difference with the serial version since OpenMP aims to keep the code modification small. The datapath contains a lot of loops so the loop parallel feature in OpenMP is very useful in this case. The syntax of the OpenMP parallel loop construct in C/C++ is shown as below:

```
#pragma omp parallel for [clause[[,] clause]\...]
        for-loop
```

However, it is not straightforward to insert the parallel loop pragma into the serial code. Programmers have to take care of data dependencies between iterations in the loops. If the data dependencies are not solved correctly, the program may have unexpected results due to data sharing problems or race conditions. OpenMP provides features to

tackle the data dependencies but it is still the programmers' responsibility to utilize them correctly. Workload allocation is done automatically by OpenMP and the programmers can choose the workload scheduling modes which were mentioned in the previous section. Nevertheless, what is the optimal workload scheduling mode and the chunk size is not a trivial question. Some experiments in the next section may help to answer this question.

6.3 Experiments

6.3.1 Experiment: the startup overhead of OpenMP

Setup:

- Test object: the datapath code in OpenMP without any parallelization, i.e. executing on only one thread.
- Test input: one swath with 40,000,000 pixels.
- Test platform: Platform I.
- Test goal: To observe the startup overhead of OpenMP by executing the datapath code on OpenMP with only one thread and comparing the execution time with the sequential version of the datapath code.

Result: The IP4 version takes 374.52 ms to finish the job and the sequential version takes 356.76 ms to finish the job. The startup overhead of OpenMP without any parallelization is about 4.97 %, which is calculated by using Formula 5.3.

6.3.2 Experiment: OpenMP on the multi-core platform with the different loop schedulers

Setup:

- Test object: the datapath code with the RO task which is parallelized by OpenMP. The number of threads on which the RO task is executed is varied.
- Test input: one swath with 40,000,000 pixels.

- Test platform: Platform I.
- Test goal: To observe the performance of the application with OpenMP on the multi-core platform with the different loop schedulers, i.e. static scheduler, dynamic scheduler and guided scheduler. Auto scheduler is only available for OpenMP 3.0 but Visual Studio compiler only supports OpenMP 2.0 so it is not selected in this case.

Result: Figure 6.2 shows the result when executing the RO task with the various numbers of threads on 4 cores of platform I. The execution time of the RO task with the static loop scheduling decreases when the number of threads increases from 1 to 4 then it fluctuates when the number of threads is more than 4. There are peaks in the execution time graph at 5, 9, and 13 threads. The execution time of the RO task with the dynamic loop scheduling does not have the fluctuation like the static scheduling. When the number of threads is more than the number of cores, i.e. more than 4 threads in this case, the execution time is kept the same as when the number of threads is equal to the number of cores. The execution time of the RO task with the guided loop scheduling is close to the dynamic loop scheduling case with small fluctuation.
Figure 6.3 shows the execution time of the RO task with four threads on the different number of cores. OpenMP has an overhead which can be recognized as the difference between the sequential version execution time and the OpenMP version execution time. OpenMP runs slower than the sequential code on a single core due to the startup overhead but it does not have a significant difference. In the static loop scheduling case, the execution time does not improve when the number of cores increases from 2 to 3. It does not happen for the dynamic and guided loop scheduling. The dynamic loop scheduling performs better than the guided loop scheduling when the number of cores is 3.

Analysis: When the number of threads increases, the execution time decreases because the workload is divided into more threads and can be parallelized more. However, when the number of threads is more than the number of cores, we experience the performance impact on the execution time. In our case, the maximum number of threads which can be executed concurrently is four. When there are five threads and the workload is divided equally, the last thread runs on one core and the other three cores are idle at that time. This behavior is known as load imbalance. With six threads, the performance is better than with five threads, since the last two threads can be executed on two different

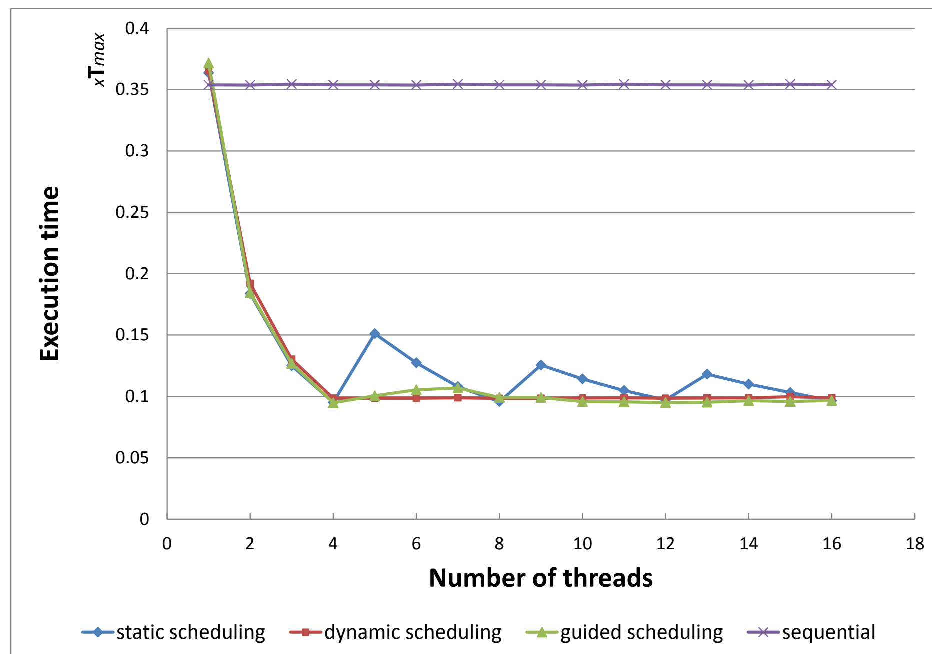

FIGURE 6.2: The execution time of the RO task with the different OpenMP loop schedulers with a different number of threads on 4 cores of platform I

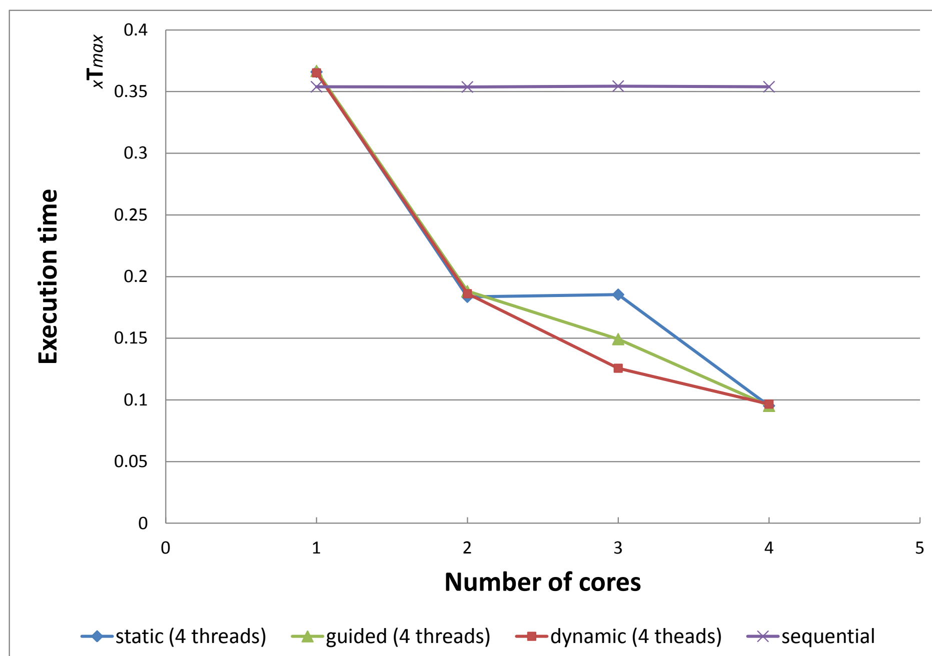

FIGURE 6.3: The execution time of the RO task with the different OpenMP loop schedulers on a different number of cores of platform I

cores and only two cores are idle at that time. Figure 6.4 illustrates the execution of threads on four cores.

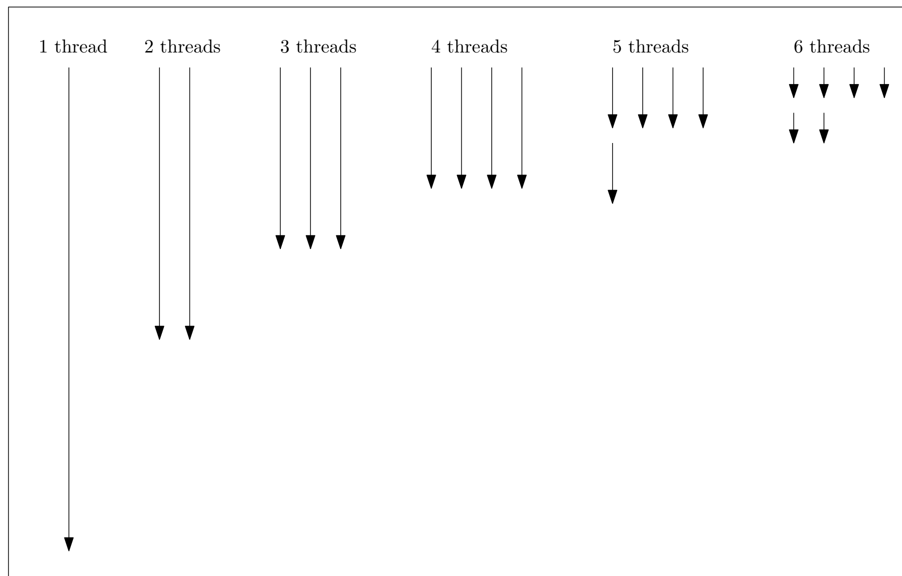

FIGURE 6.4: The execution of threads on 4 cores

The load imbalance behavior does not happen with the dynamic loop scheduling because the workload is allocated in real time. The threads can only process the next chunk of workload if they finish the current chunk. Thus, there is no case when the threads are allocated with a lot of workload but they have to wait for the available cores to process. The same approach is employed in the guided loop scheduling. However, the guided loop scheduler starts with a big chunk size and then the chunk size decease to adapt to the real time processing. The initial chunk size is proportional to the number of iterations divided by the number of threads. When the number of threads is small, the initial chunk size is big. Therefore, some threads have big initial chunks to process but there are no available cores for them. It results in the fluctuation of the execution time when the number of threads increases from four to eight in Figure 6.2. Similarly, the load imbalance is the cause of the problem of the static loop scheduling in Figure 6.3. When there are three cores, the last thread only runs on one core and other two cores are idle, so it does not help to improve the performance from two cores case. Figure 6.5 illustrates the distribution of four threads across a different number of cores.

In this case, the scheduling overhead of the dynamic scheduler and the static scheduler is not much different. The dynamic scheduler is expected to have more the scheduling overhead due to the real time scheduling. However, the number of iterations in the

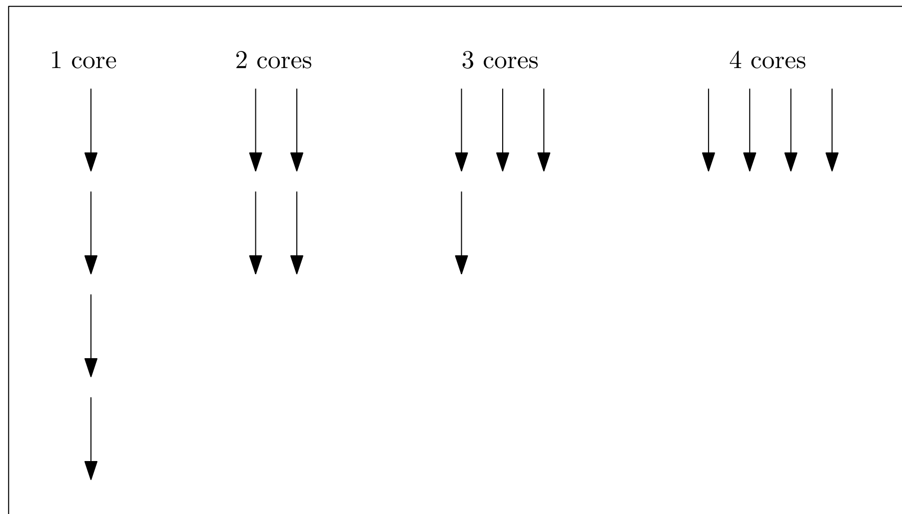

FIGURE 6.5: The execution of 4 threads on a different number of cores

rotation task is only 1182. Thus, the number of chunks is not many even though the chunk size is set to one. For the mask application task, there are more than 600,000 iterations and the processing time of each iteration is small, so the scheduling overhead between the dynamic scheduler and the static scheduler is significant. It takes 34.54 ms to complete the mask application task for one swath on 4 threads with the dynamic scheduler while it takes only 0.47 ms to complete the same work load with the static scheduler and 0.50 ms with the guided scheduler. In this experiment, the chunk size is set to default. The effect of changing the chunk size is verified in the next experiment.

6.3.3 Experiment: OpenMP with the different chunk sizes in the loop scheduler

Setup:

- Test object: the datapath code with the RO task which is parallelized by OpenMP.
- Test input: one swath with 40,000,000 pixels.
- Test platform: Platform I.
- Test goal: to observe the performance of OpenMP with different chunk sizes in the OpenMP dynamic loop scheduler.

Result: Figure 6.6 shows the result of this experiment. The execution of the RO task on 4 cores (with 4 threads) fluctuates with the highest peak at the chunk size 240 before

increasing linearly at the chunk size 300 and remaining constantly at the chunk size 1180. Similarly, the execution of the RO task on 3 cores (with 3 threads) fluctuates with the highest peak at the chunk size 300 before increasing linearly at the chunk size 400 and remaining constantly at the chunk size 1180.

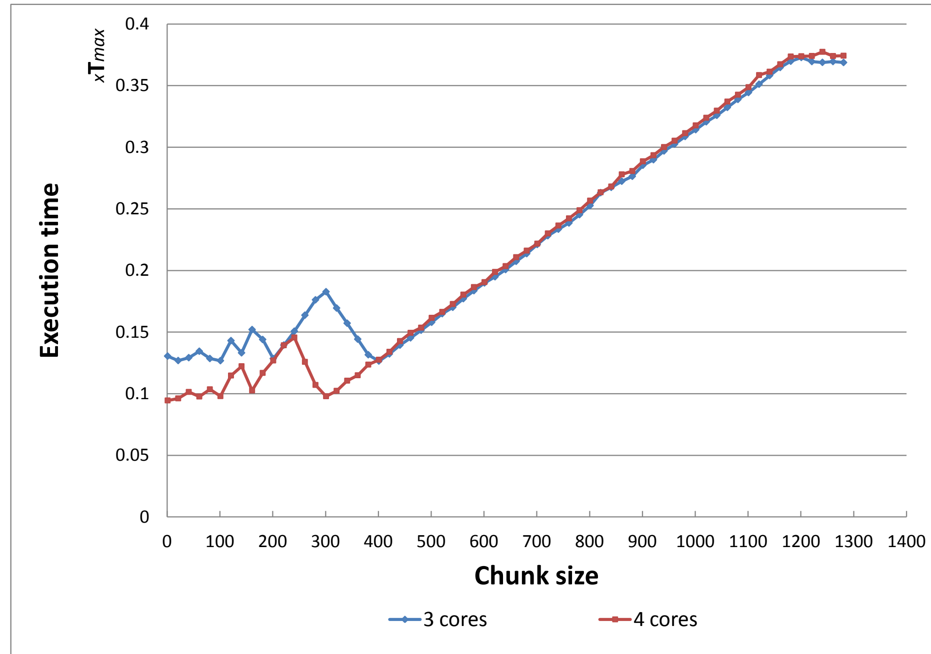

FIGURE 6.6: The execution time of the RO task with different chunk sizes in the OpenMP dynamic loop scheduler

Analysis: The larger chunk size can reduce the synchronization overhead of the scheduler but also make it less robust to the change of input. When the chunk size is 236 ($=\frac{1182}{5}$) for 4 cores case, the load imbalance among the cores is high. The first 4 chunks are allocated to 4 threads and the last chunk is allocated to the fastest thread which completes its chunk. It is the reason why there is a peak at the chunk size 240, i.e., the nearest tested chunk size to 236. Similarly, there is a peak at chunk size 300 ($\approx \frac{1182}{4}$) for 3 cores case. The execution starts increasing linearly at the chunk size 300 ($\approx \frac{1182}{4}$) for 4 cores and 400 ($\approx \frac{1182}{3}$) for 3 cores since the load imbalance overhead become larger and larger. There are only 1182 iterations so the execution stays constant when the chunk size is more than 1180 ($\approx$ 1182). The number of iterations is 1182, which is not big. The scheduling overhead is not high; therefore the load imbalance overhead is dominant in this case. Generally, the execution time has a peak when the chunk size is $\frac{N_i}{(N+1)}$ (with N_i is the number of iterations, N is the number of cores) and increases linearly from the chunk size $\frac{N_i}{N}$.

For the MA task, we can observe a huge improvement in the execution time when the

chunk size increases from 1 to 50 as shown in Figure 6.7. The MA task contains more than 600,000 iterations so its scheduling overhead is high when the chunk size is small.

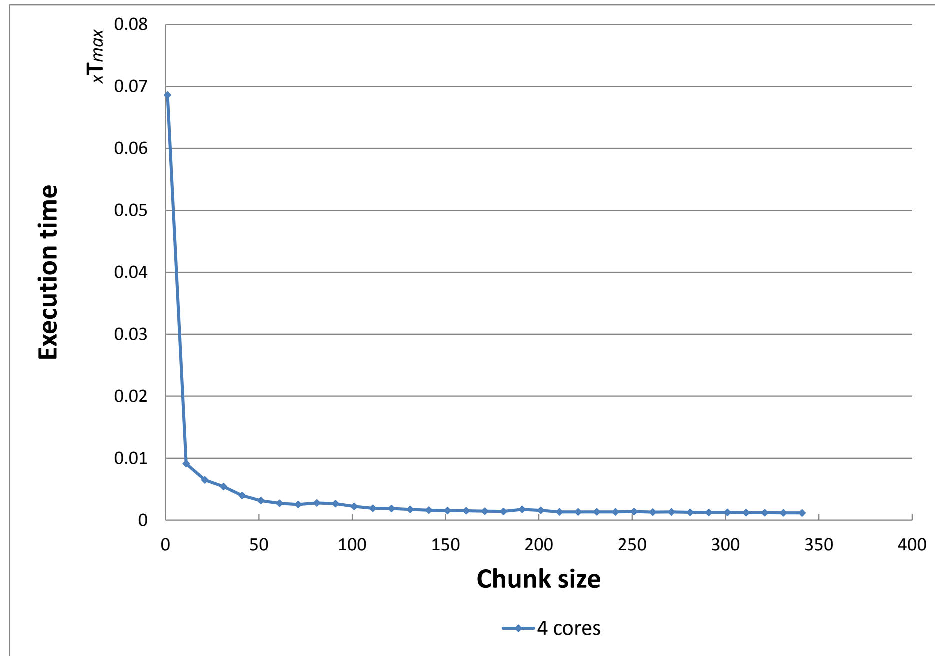

FIGURE 6.7: The execution time of the MA task with different chunk sizes in the OpenMP dynamic loop scheduler

If we look at a larger range of chunk size, the same behavior like the RO task can be observed. In Figure 6.8, the execution time of the MA task on 4 cores (with 4 threads) has a peak at the chunk size $\frac{N_i}{(N+1)} \approx \frac{600000}{(4+1)} = 120000$ and it starts increasing linearly at the chunk size $\frac{N_i}{N} \approx \frac{600000}{4} = 150000$.

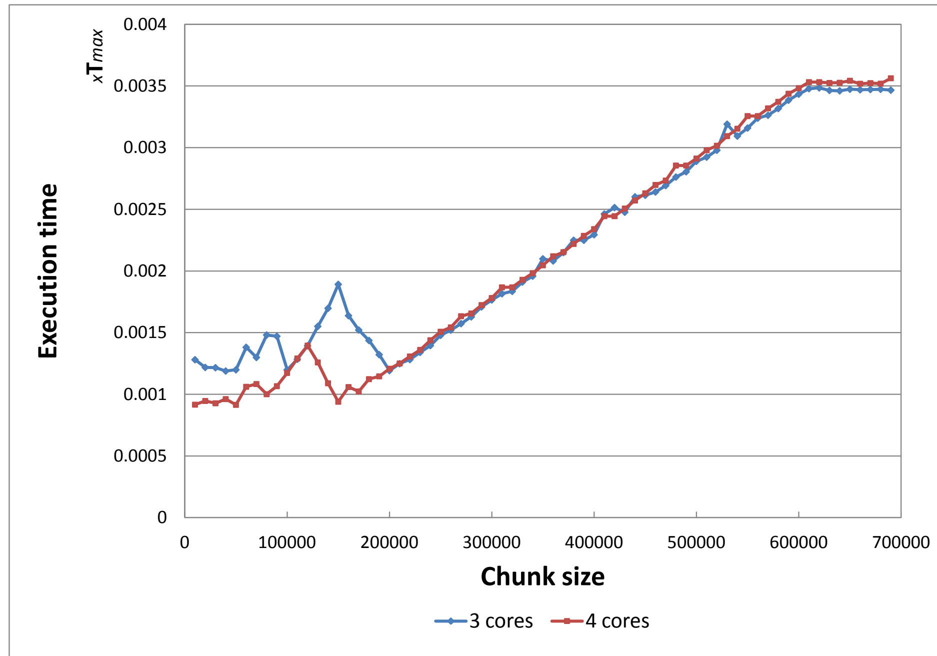

FIGURE 6.8: The execution time of the MA task with different chunk sizes in the OpenMP dynamic loop scheduler (larger range)

6.3.4 Experiment: the speedup of the tasks of the datapath with OpenMP on different platforms

Setup:

- Test object: the datapath code with the SE, YC, RO task which are parallelized by OpenMP with 4 threads and the dynamic loop scheduler with the chunk size 1.
- Test input: one swath with 40,000,000 pixels.
- Test platform: Platform I and A. The code with OpenMP is compiled by the Visual Studio compiler for platform I and the GCC cross compiler gcc-linaro-arm-linux for platform A. The number of cores which are available for executing the datapath code is varied.
- Test goal: to observe the speedup of the tasks of the datapath with OpenMP on different platforms when the number of available cores is varied.

The speedup is calculated by Formula 2.1 from the measured execution time.

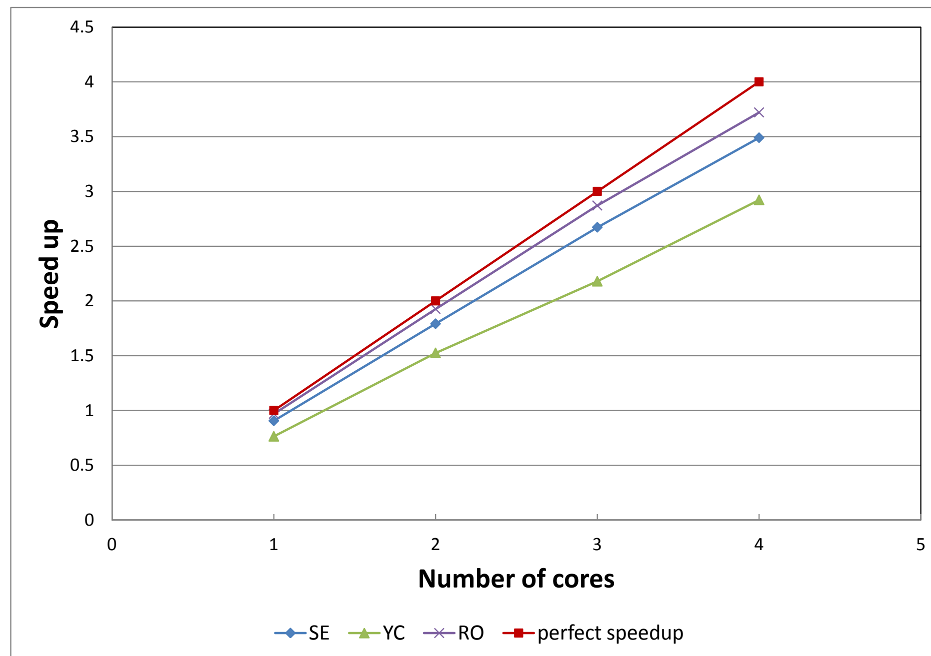

FIGURE 6.9: The speedup of the tasks of the datapath in OpenMP on the platform I

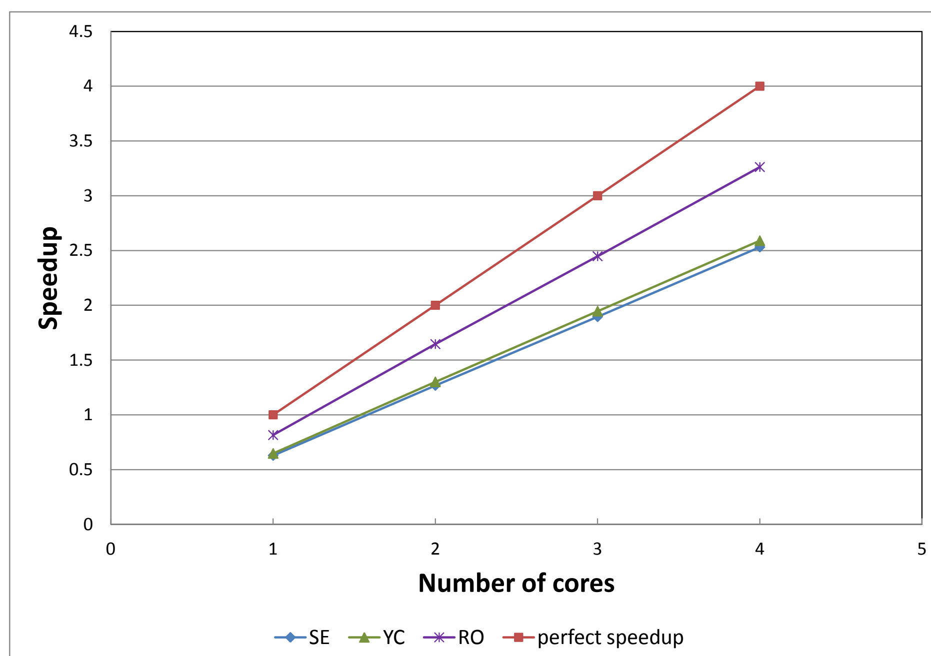

FIGURE 6.10: The speedup of the tasks of the datapath in OpenMP on the platform A

Result: Figure 6.9 shows the speedup of the datapath tasks in OpenMP on the platform I with the Visual Studio compiler. The speedups vary for different tasks but all of them are nearly linear. Figure 6.10 shows the speedup of the datapath tasks in OpenMP on the platform A with the GCC compiler. The speedups are almost linear. However, it has a different result on the platform I.

Analysis: OpenMP is dependent on the compilers and the platforms. Two platforms have different hardware specifications such as speed, cache size, etc. Moreover, the GCC cross compiler uses GCC version 4.8.3 with OpenMP 3.1 while Visual Studio compiler still uses OpenMP 2.0. These can be the reasons for the different speedups between two compilers. If the speedups of the RO task are plugged in the formula 2.5 with $f = 0.99$, we get the coefficients $(a, b, c) = (0.0019, 0.0021, 0.0021)$ and the goodness of the fit $R^2 \approx 1$ for the experiment on the platform I. We get the coefficients $(a, b, c) = (0.0096, 0.0094, 0.0264)$ and the goodness of the fit $R^2 = 0.9980$ for the experiment on the platform A. The coefficients in this experiment are smaller than in the experiment of IP4. It shows the speedups in OpenMP are higher than in IP4 for this datapath. It is indeed proved for the RO task in Figure 6.11. The coefficient c is much higher on the platform A than the platform I. The platform A is less powerful and has fewer resources,e.g. smaller caches, compared to the platform I. It can be a cause of the higher resource contention.

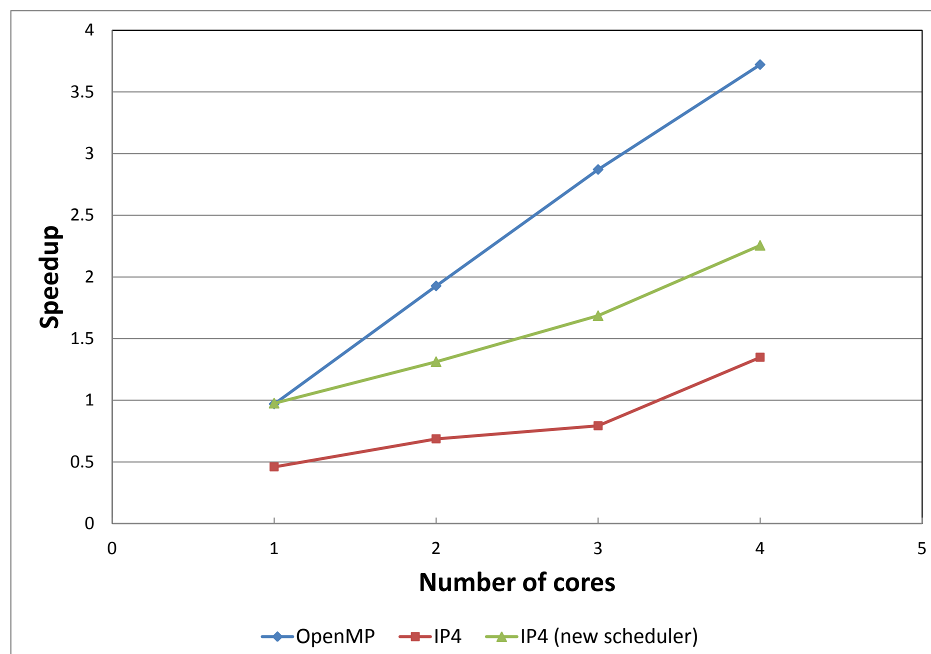

FIGURE 6.11: The speedup of the RO of the datapath in OpenMP and IP4 on the platform I

6.3.5 Experiment: Different input data sizes in OpenMP

Setup:

- Test object: the datapath code with the RO task and MA task, which are parallelized by OpenMP. The number of threads on which the tasks are executed is varied as same as the number of available cores.
- Test input: one swath with different swath sizes.
- Test platform: Platform I.
- Test goal: to observe the effect of the swath size on the speedup of the tasks in the datapath with OpenMP.

Result & Analysis: As shown in Figure 6.12, the speedup of the RO task does not vary significantly when the swath size changes. The experiment result is the same as IP4. The cache is not utilized effectively in the RO task so we do not see the effect on the speedup when the swath size changes.

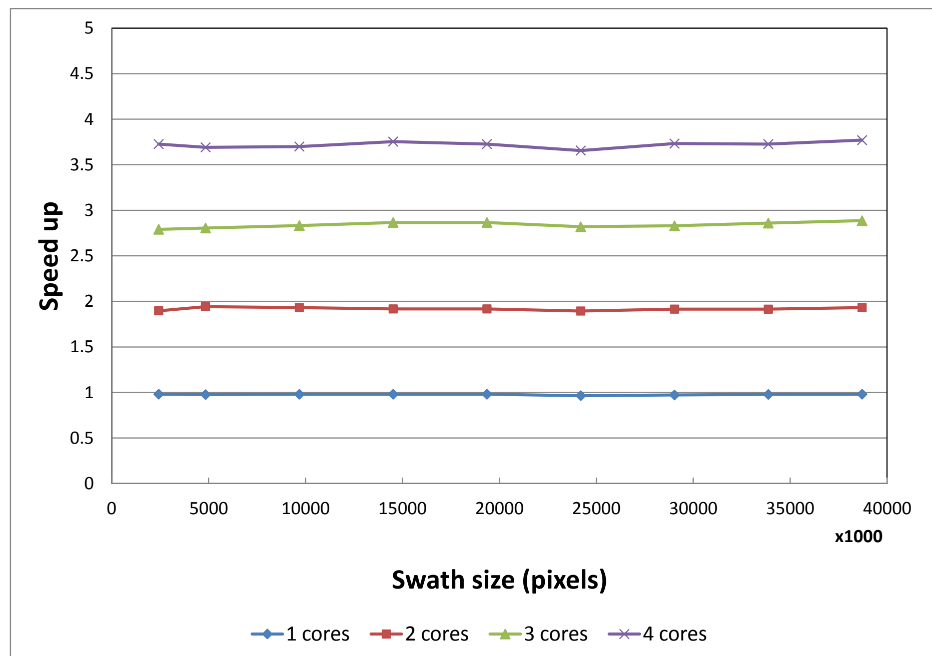

FIGURE 6.12: The speedup of the RO task in OpenMP with different swath sizes

In another test case, the effect on speedup can be observed in the MA task in Figure 6.13. The MA task is a simple task and has high locality, which is very suitable for the

cache utilization. Platform I has 64KB L1 cache, 1MB L2 cache and 6MB L3 cache. The parallel code can utilize all the caches in all the cores but the sequential code can only utilize the L1 can L2 cache on one core. At the beginning, the speedup increases since the sequential code starts using L2 cache but the parallel code still uses L1 cache. When the swath size keeps increasing, the speedup increases slower because the parallel code uses all the L1 caches on all the cores and starts using the L2 caches. When the sequential code starts using the L3 cache, the parallel code still has the available L2 caches. Thus, the speedup increases again. When the parallel code starts using the L3 caches, the speedup starts reducing. When the parallel code uses more and more L3 cache, the speedup keeps reducing.

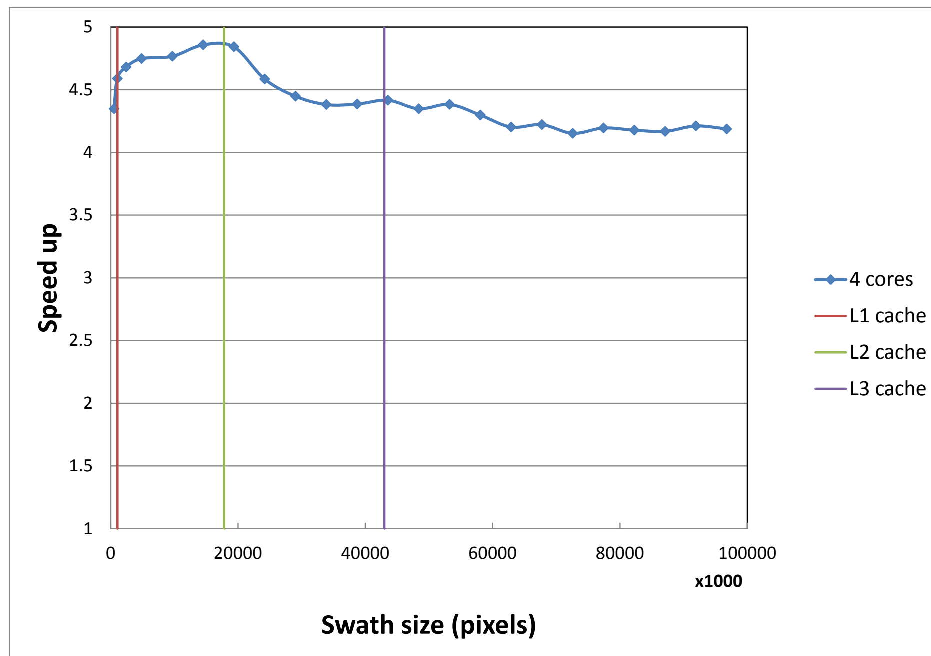

Figure 6.13: The speedup of the MA task in OpenMP with different swath sizes on 4 cores

6.3.6 Experiment: Compatibility with the development environment

Although the performance of the selected multi-core programming frameworks is checked in the previous chapters but they are evaluated in an independent environment. In reality, the datapath is just a part of the software system in the printers. Thus, if the datapath is implemented with the parallel programming frameworks, it is important that they are compatible with the rest of the software system. In this section, we evaluate the compatibility of OpenMP with the developing environment. The software

development environment in this case is Rational Software Architect Real Time Edition (RSARTE). Rational Software Architect (RSA) is a suit of modeling and development software applications using the Unified Modeling Language (UML). RSA is built on the Eclipse framework with a variety of plug-ins. RSARTE is an edition of RSA, which supports for developing real time software applications. The software applications are designed in RSARTE with UML diagrams which can be converted to the target code automatically.

While the compatibility between IP4 and RSARTE was evaluated in a different project as mentioned in section 5.5, the compatibility between OpenMP and RSARTE has not been evaluated. To be able to test with RSARTE, one simple model is created in RSARTE. A software application is designed in RSARTE as UML diagrams like state machine diagram. Figure 6.14 shows a simple state machine diagram in RSARTE. One timer is put in the transition from the initial state and the state S1. The OpenMP datapath code is put in the loop transition of the state S1. When the timer triggers, the datapath code is executed. The state machine diagram is converted to the C++ target code by RSARTE. The target code is then compiled by GCC-TMD compiler which has the activate OpenMP option and executed on the platform I. The deference between the execution time which we get from the execution of the RSARTE generated code and the execution time of the original OpenMP datapath code is not significant (below 1%). From this result, we can show that OpenMP is suitable to integrate into the RSARTE environment.

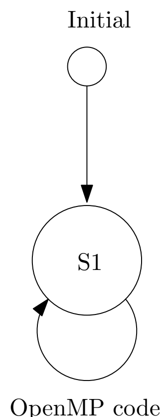

FIGURE 6.14: OpenMP in RSARTE

6.4 Chapter Summary

In this chapter, OpenMP has been explained in detail. The experiment 6.3.1 shows that the startup overhead of OpenMP is larger than IP4 but still not significantly large. Developers have to choose the type of loop scheduler and the chunk size by themselves. The dynamic scheduler has a high scheduling overhead but a low load imbalance overhead. The static scheduler has a low scheduling overhead but a high load imbalance overhead. The guided scheduler is in between the dynamic scheduler and the static scheduler. The experiment 6.3.2 shows the tradeoff between the schedulers. The dynamic scheduler is suitable for loops which contain not too many iterations and the execution time of each iteration is not too short. While the static scheduler is suitable for the loops which contains a lot of iterations and the execution time of each iteration is short. In the datapath application, most of the tasks are suitable to use the dynamic loop scheduler except the MA task. The chunk size is also has a significant impact on the execution time as shown in the experiment 6.3.3. A larger chunk size can help to reduce the scheduling overhead of the dynamic loop scheduler, however it can also increase the load imbalance overhead. OpenMP shows good scalability on the different compilers and platforms, i.e. Visual Studio compiler for Windows on the Intel platform and GCC cross compiler for Linux on the ARM platform, in the experiment 6.3.4. The size of the input data, i.e. the swath size does not affect the speedup of the parallel task, except when the task is simple and has high locality for utilizing caches like the MA task. OpenMP has been proved to be compatible with the software development environment RSARTE in the experiment 6.3.6.

Chapter 7

Combined IP4 and OpenMP

In this chapter, we evaluate the combination of IP4 and OpenMP. As shown in the previous two chapters, IP4 is flexible and target on developing image processing pipelines. However, it does not have an automatic workload allocation mechanism. While OpenMP is easy to parallelize an existing code and supports workload allocation. But it does not have templates for developing image processing pipeline like IP4. The idea of combining IP4 and OpenMP to combine the strengths of both frameworks is investigated in this chapter. To be able to evaluate the combined framework, we use the IP4 model with one thread like in the experiment 5.3.1 and replace the code in the RO task with OpenMP code. In other words, we keep the structure of IP4 but let OpenMP do the parallelization in the tasks. Thus, we still can utilize IP4 scripts and its structure to develop image processing pipelines as well as the OpenMP workload allocation feature. Both IP4 and OpenMP have the startup overhead so the first question is whether the combined framework has a significant overhead. When the number of threads in the RO task is set to one by OpenMP, the experiment 5.3.1 is repeated. The difference between the execution time in the combined framework and IP4 is negligible (below 1%). When the number of threads in the RO task is set to four by OpenMP, the RO task is parallelized on four threads. The execution time of the RO task in the combined framework is the same with OpenMP and better than IP4 as shown in Figure 7.1.

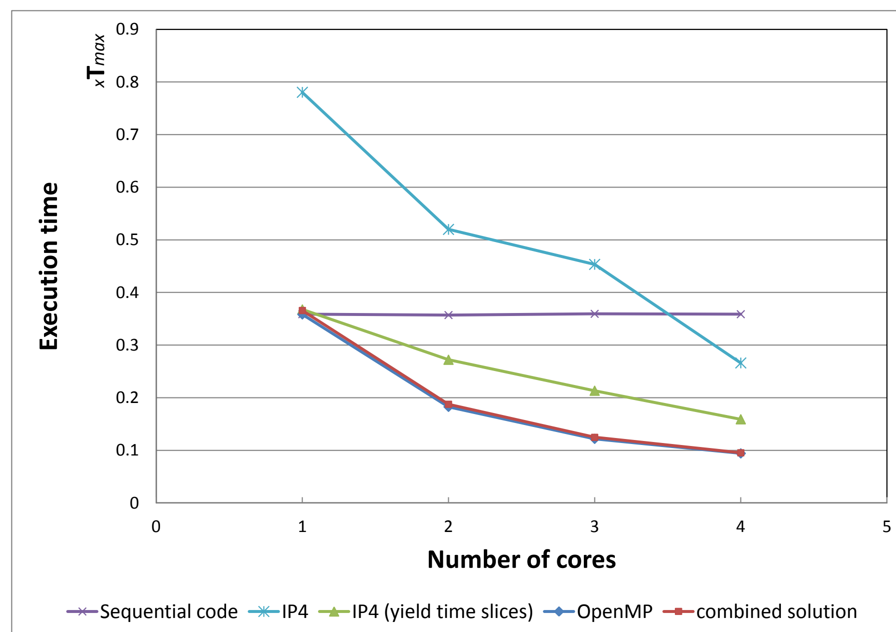

FIGURE 7.1: The combined framework of IP4 and OpenMP

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this thesis, we have evaluated the feasibility of multi-core programming frameworks for the datapath in wide format printers. The approach started with defining the search space based on the interest of the company, the broad range of choices, the availability and maturity of the tools. Then the search space size reduction was performed with the available documents and the pre-defined criteria. Within our scope, the search space was formed with the sequential code, the native threading package, OpenMP, IP4, TBB and SDF3. After reducing the size of the search space, there are 3 remaining candidates which are the sequential code, OpenMP and IP4. The details of this work were described in chapter 4. The selected candidates were evaluated their feasibility for the application by the experiments and analysis in chapter 5 and 6.

IP4 is built as templates which aim for image processing pipelines so it can be useful for developing the image processing algorithms from scratch. From software designers' point of view, it makes developers implement their code in modules which are more structural and easier to maintain. It also makes the designers keep in mind about the parallelism when they design the algorithm. Moreover, IP4 provides its own scripts which can help to change the configuration of the image processing pipelines without re-compile the code. In other hand, IP4 requires more effort and code modification when converting existing sequential code to parallel code. The significant degradation in performance is observed when running IP4 on a small number of cores due to the thread scheduling. In this work, some solutions have been suggested to solve that problem and they show

a good improvement compared to the current thread scheduling of IP4. In contrast, OpenMP does not have templates which support developing image processing pipelines, but it can be easily used for converting the existing sequential code to the parallel code with small code modification by inserting its directives into the sequential code. Especially, OpenMP has good support for the loop parallelization, thus it is suitable for the code which contains a lot of loops. Loop parallelization is supported by OpenMP loop schedulers that can divide the workload automatically. OpenMP also support the parallelization of any sections of code (not just only loops) but the workload division has to be done manually by using the thread identifiers. Therefore, it depends on the application development to choose between IP4 and OpenMP. For the datapath in WFP, parallelizing one part of the pipeline is more beneficial than parallelizing the pipeline at the swath level due to the dependency of the swaths. OpenMP is the better choice in this case since we need to parallelize one part of the existing code which contains a lot of loops.

Although IP4 or OpenMP provides their features to parallelize the code, it is still the responsibility of the developers to handle dependencies in loops and other threading problems like race condition or load imbalance. While IP4 does not have any workload allocation feature, OpenMP have a loop scheduling feature which can schedule the load for threads. However, selecting the suitable loop schedule and the chunk size is not a trivial problem. The static scheduling has a high load imbalance overhead but a low scheduling overhead since the scheduling is done before executing the threads. On the other hand, the dynamic scheduling has a low load imbalance overhead but a high scheduling overhead. The trade-off between the load imbalance overhead and the scheduling overhead needs to be considered by the developers. If the number of iterations is small and the execution time of each iteration is not too short, the dynamic scheduler is more suitable. If the number of iterations is big and the execution time of each of iteration is relatively short, the static scheduler is more suitable. The guided scheduler is in between the dynamic scheduler and the static scheduler. It has less scheduling overhead but it is less robust to load imbalance. In our application, the dynamic scheduler is suitable for most of the tasks except the MA task which contains many loop iterations. The chunk size is also has a significant impact on the execution time. The larger chunk size can help to reduce the scheduling overhead of the dynamic loop scheduler, however it can also increase the load imbalance overhead.

A combination of IP4 and OpenMP was also evaluated in chapter 7. The combination

solution shows a good performance and can capture the strengths of both frameworks. IP4 is currently only available for Windows at this moment. OpenMP is dependent on the compiler but it supports many compilers on different operating systems. OpenMP shows good scalability on different compilers and platforms. The different compilers and platforms affect the performance of the application in OpenMP.

8.2 Future Work

In this final section, we propose some possible extensions to the work described in this thesis. First of all, all the experiments have been performed in a "clean" environment where there are no other running applications beside the datapath. It is not the case in the real working environment where other applications are running concurrently with the datapath on the target platform. These applications can affect the performance of the datapath significantly so it is necessary to test the parallel datapath in that kind of environment.

Secondly, only one solution to the thread scheduling of IP4 is implemented in this thesis. The priority adjustment solution can be a good solution since it shows a good performance on the test program. Thus, it is worth to implement it in IP4 to verify. Testing IP4 with the ARM platform can be carried on after a stable version of IP4 for Linux is ready.

The parallel datapath is implemented in OpenMP and IP4 manually. All the decisions in the implementation like choosing schedulers, chunk sizes, etc. are made from experiments. It is a good suggestion to parallelize the code automatically. Especially, OpenMP has a low level of modification, which may help to build an automatic parallel code generator easier.

Appendix A

The Datapath in Wide Format Printers Description

This appendix chapter is confidential and only provided as a separate document with the permission from Océ.

Bibliography

[1] Océ. Océ company, 2014. URL http://global.oce.com/company/default.aspx.

[2] Wikipedia. Amdahl's law, 2014. URL http://en.wikipedia.org/wiki/Amdahl's_law.

[3] M. Roth, M. J. Best, C. Mustard, and A. Fedorova. Deconstructing the overhead in parallel applications. In *Workload Characterization (IISWC),IEEE International Symposium*, La Jolla, CA, USA, 2012.

[4] W. Dautermann. Programming with OpenMP, 2014. URL http://www.admin-magazine.com/HPC/Articles/Programming-with-OpenMP.

[5] Kenn R. Luecke. Software Development for Parallel and Multi-Core Processing. In Kiyofumi Tanaka, editor, *Embedded Systems - High Performance Systems, Applications and Projects*. 2012.

[6] Peter S. Pacheco. *An Introduction to Parallel Programming*. San Francisco: Morgan Kaufmann Publishers Inc, 1st edition, 2011.

[7] OpenMP. The OpenMP® API specification for parallel programming, 2014. URL http://openmp.org/wp/.

[8] Barbara Chapman, Lei Huang, Eric Biscondi, Eric Stotzer, Ashish Shrivastava, and Alan Gatherer. Implementing OpenMP on a high performance embedded multicore MPSoC. In *Parallel & Distributed Processing, IEEE International Symposium*, Rome, Italy, 2009.

[9] Intel. Intel® Threading Building Blocks (Intel® TBB), 2014. URL https://www.threadingbuildingblocks.org/.

[10] G. Contreras and M. Martonosi. Characterizing and Improving the Performance of Intel Threading Building Blocks. In *Workload Characterization, IEEE International Symposium*, Seattle, USA, 2008.

[11] MIT. The Cilk Project, 2014. URL http://supertech.csail.mit.edu/cilk/.

[12] C. E. Leiserson. The Cilk++ concurrency platform. In *46th Annual Design Automation Conference (DAC '09)*, New York, USA, 2009.

[13] D. A. Mallon, G. L. Taboada, C. Teijeiro, J. Tourino, B. B. Fraguela, A. Gomez, R. Doallo, and M. J. Carlos. Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures. In *16th European PVM/MPI Users' Group Meeting*, Espoo, Finland, 2009.

[14] T. Deepak, K. Varaganti, R. Suresh, R. Garg, and R. Ramamoorthy. Comparison of Parallel Programming Models for Multicore Architectures. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW),IEEE International Symposium*, Shanghai, China, 2011.

[15] N. Wahlen. A comparision of different parallel programming models for multicore processors. Bachelor's thesis, KTH Information and Comunication Technology, Stockholm, Sweden, 2010.

[16] S. Jarp, A. Lazzaro, A. Nowak, and L. Valsan. Comparison of software technologies for vectorization and parallelization. Technical report, CERN, 2012.

[17] S. Stuijk, M. Geilen, and T. Basten. SDF3: SDF For Free. In *Application of Concurrency to System Design, 6th International Conference, ACSD*, Turku,Finland, 2006.

[18] M. Voss. Intel® Threading Building Blocks, OpenMP, or native threads?, 2011. URL https://software.intel.com/en-us/intel-threading-building-blocks-openmp-or-native-threads.

[19] G. M. Amdahl. Validity of the single-processor approach to achieving large-scale computing capabilities. In *American Federation of Information Processing Societies Conference (AFIPS)*, California, USA, 1967.

[20] J. Bull. Measuring Synchronisation and Scheduling Overheads in OpenMP. In *Proceedings of First European Workshop on OpenMP*, 1999.

[21] K. F. Gerndt and M. Gerndt. Analyzing Overheads and Scalability Characteristics of OpenMP Applications. In *Proceedings of the Seventh International Meeting on High Performance Computing for Computational Science (VECPAR'06)*, Rio de Janeiro, Brazil, 2006.

[22] N. M. Larsgård. Parallelizing Particle-In-Cell Codes with OpenMP and MPI. Master's thesis, Department of Computer and Information Science, Norwegian University of Science and Technology, Trondheim, Norway, 2007.

[23] Wikipedia. Coefficient of determination, 2014. URL http://en.wikipedia.org/wiki/Coefficient_of_determination.

[24] E. Ajkunic, H. Fatkic, E. Omerovic, K. Talic, and N. Nosovic. A comparison of five parallel programming models for C++. In *MIPRO, Proceedings of the 35th International Convention*, Opatija, Croatia, 2012.

[25] S. C. Ravela. Comparison of Shared memory based parallel programming models. Master's thesis, School of Computing, Blekinge Institute of Technology, Sweden, 2010.

[26] S. Stuijk, M. Geilen, and T. Basten. A Predictable Multiprocessor Design Flow for Streaming Applications with Dynamic Behaviour. In *Digital System Design, 13th Euromicro Conference, DSD 10 Proceedings, IEEE Computer Society Press*, Los Alamitos, CA, USA, 2010.

[27] OpenMP_ARB. *OpenMP Application Program Interface Version 3.1*. OpenMP Architecture Review Board, 2011.