Programming with Explicit Dependencies

A Framework for Portable Parallel Programming

Programming with Explicit Dependencies

A Framework for Portable Parallel Programming

Eva Burrows

Dissertation for the degree of philosophiae doctor (PhD)
at the University of Bergen

March 2011

ISBN 978-82-308-1730-8

Published by the University of Bergen, Norway, 2011
Printed by AIT

MOTTO

Remember your Creator in the days of your youth,
Before the difficult days come, and the years
draw near when you say, "I have no pleasure in them" (. . .)
For man goes to his eternal home,
And the mourners go about the streets.

Remember your Creator before the silver cord is loosed,
Or the golden bowl is broken,
Or the pitcher shattered at the fountain,
Or the wheel broken at the well.
Then the dust will return to the earth as it was,
And the spirit will return to God who gave it.

"Vanity of vanities," says the Preacher,
"All is vanity." (. . .)

And further, my son, be admonished by these.
Of making many books *there is* no end,
and much study *is* wearisome to the flesh.

Let us hear the conclusion of the whole matter:
Fear God and keep His commandments,
For this is man's *all*.
For God will bring every work into judgment,
Including every secret thing,
Whether good or evil.

Ecclesiastes 12

Preface

When I set off on my Norwegian saga to obtain a PhD in computer science, I soon realised how little prepared I was for the challenges that would have to face on the long run, truly I was unprepared and naive. Making the decision to embark on a journey like this took several years, and it brought about a series of life-changing events, e.g., moving country, leaving behind family and friends, and quitting Koinónia – a Romanian publisher where I had spent most of my time working as IT Manager for more than a decade by then.

My return to academia began with a two years study leave from my employment to obtain a Master's Degree in Programming Theory from the Department of Informatics at the University of Bergen. Marc Bezem, who supervised me at the time, tried to entice me to enrol on a PhD training right away. I was, however, rather uncertain about this, and upon my graduation I duly returned home.

During the years of my master studies, I had the opportunity to get an insight into Magne Haveraaen's research agenda on algebraic software methodologies. In the first year, he was my professor for a course on algebraic program specification, and in the following year, he asked me to be his teaching assistant for the course.

When I made up my mind about my PhD, I contacted Magne inquiring about the prospect of working with him. His most positive reply, finally, removed all my remaining doubts.

The journey since then has been long, enriching but tiring, uplifting and depressing. Sometimes all in the same time. I have learnt many a thing about computer science, research, academia and life. This knowledge, I hope, will stay with me on the journey onward.

Acknowledgements

I remain indebted to Magne for his unceasing advice and support which provided me with the knowledge, determination and confidence to com-

plete this dissertation. The long, enlightening discussions from the wonders of algebraic abstractions to the beauty of analog photography have always been a source of excitement and refreshment. In the same time, my gratitude goes to Mary Sheeran for taking me onboard, in the first place, and becoming my co-advisor despite her busy schedule, and her constant "How are things?"-check-ups. Her challenging remarks have improved a great deal the breadth and depth of my work. Her workload and attitude in responding my emails within ten minutes, and providing me usually with all the answers I needed, will always remain an example to follow.

The Department of Informatics at the University of Bergen provided me with financial support, a nice office, and countless opportunities for travelling, all of which I am most grateful for. The conferences I have been to were most beneficial for my professional development and networking. I thank my direct colleagues Anya Helene Bagge and Valentin David who helped me in getting through the "PhD system". I hardly ever had a technical or not so technical question that Anya did not have an answer for and has always been a source of information throughout my research. I also thank Ida Holen, Liljan Myhr, Maria Marta Lopez and Tor Bastiansen for doing such a great job in the administration, and getting things done so smoothly.

Many thanks go to people outside the academia, too. Ruth and Magnus Frantzen have become my Norwegian parents, and have shown affection, love and support the way that only parents can do. Their presence made a huge difference to my stay in Norway, for which I am most thankful. Éva Bartha, Ildikó Orbán, Éva Doepp, Ágnes Bálint and Erzsébet Visky have been my best friends for the past 15 years. They are the kind of persons one can always rely on. Their friendship and support have always meant a lot.

I am most grateful to my parents for just simply ALWAYS being there for me. Sadly, my mother passed away before she could have witnessed this happy day with the rest of my family, to whom many thanks go, too: my brother Sanyika, his wife Lucsi and their daughter Anna. Their love and kindness have always welcomed me. I am also grateful to my new family, especially my mother-in-law Dorothy. Her constant love and support have been a great source of encouragement throughout these years.

Love and countless thanks go to my husband, Andrew, for all his tender care, love and kindness, support, friendship and wisdom that helped me through many of the hickups of this journey and of my life.

Finally, but not least, my foremost gratitude, praise and thanks go to God, for His unfailing love that saved me, and for the blessings of intelligence, ambition and strength He granted me to complete this dissertation. Soli Deo Gloria!

Bergen, March 2011

Contents

List of Figures

CHAPTER 1

Introduction

The potential of parallel computing became evident more than four decades ago. Powerful parallel systems began to appear, and the initial impression that programming such machines was far from obvious has prevailed. The expanding universe of computing architectures was classified in many different ways, e.g., Flynn's famous taxonomy [Flynn, 1972]. These classifications, however, were derived from the execution models of the systems, and said little about how to program them. As Luc Bougé pointed out [Bougé, 1996], massively parallel machines seemed to have developed so quickly that little time was left to design suitable languages. Therefore, the tendency was to reflect the architecture in an architecture specific programming language, typically by starting with a widely used programming language and extending it with a construct for each hardware feature. This resulted in programming models that were mere subsets of the hardware execution models, and could hardly survive or adapt when new systems appeared. This pointed towards the need of higher level parallel programming models which would abstract from low-level hardware details, so that a programmer need not bother about these [Danelutto et al., 1992; Feldman, 1979; Greif, 1977; Skillicorn, 1995].

Over the last decade, the Message Passing Interface (MPI) [Gropp et al., 2000] and OpenMP [Chandra et al., 2000] have become the two most dominant parallel programming models. They have been taken on board by high-performance computing communities and compiler vendors.

MPI is a distributed memory model, which comes as a standard library and is available on almost every parallel platform. It can be used on networks of workstations as well as on parallel supercomputers. The user

writes program code for every participating (parallel) process, and manages the communication between processes by message passing. This is time-consuming and error-prone (e.g. deadlocks, livelocks, etc.). MPI implementations are claimed to be portable, yet they may need adaptations from platform to platform to be more appropriate for the target system.

OpenMP is a shared memory model of parallel hardware, and may achieve better performance than MPI in such an environment. OpenMP is based on a set of compiler directives applied on top of a standard language (e.g. C, Fortran, OCaml). Parallel regions are identified and annotated by the user. In general OpenMP follows a fork-join execution model.

While MPI can easily be adapted to shared memory parallel systems, OpenMP does not scale efficiently to distributed memory parallel systems. However, strategies have been proposed for implementing OpenMP on clusters [Chapman, 2005].

1.1 Parallelism Going Mainstream

Computational devices are now rapidly evolving into massively parallel systems. The number of processors per chip is expected to double every other year or so over the next few years, bringing parallel processing into the mass market. Multi-core processors are standard, and high-performance processors such as the Cell processor [Chen et al., 2007] and graphics processing units (GPUs) featuring hundreds of on-chip processors are all developed to deliver high computing power. They make parallelism commonplace, not only the privilege of expensive high-end platforms. As a consequence, software needs to be parallelized and ported in an efficient way to massively parallel, possibly heterogeneous, architectures. However, current parallel programming paradigms cannot readily exploit these highly parallel systems. In addition, each hardware architecture yet again comes along with a new programming model and/or application programming interface (API). This makes the writing of portable, efficient parallel code even more difficult.

One way of transforming many-core power into real application performance – an approach adopted by most hardware vendors – is to provide libraries, APIs or some ready-to-use software toolbox that help developers to annotate legacy code with parallel constructs where possible (e.g. [Intel, 2009, 2010; Reinders, 2007]). Industry leaders have also joined forces to develop OpenCL, a low-level cross-platform open standard for heterogeneous parallel programming [Khronos, 2010]. OpenCL exposes everything of the underlying platforms and abstracts very little, ultimately hoping to become a target itself for higher level frameworks [HPC Wire, 2010].

The programming community is still in great need of high-level parallel programming models to adapt to the new era of commonly available parallel computing devices as well as to the increasingly more accessible realm of high-performance computing facilities. Parallel computing research blooms like never before. While in the early years of computing parallelism seemed to be desirable for high throughput, by today taking parallelism into account has become inevitable:

> "The «not parallel» era we are now exiting will appear to be a very primitive time in the history of computers when people look back in a hundred years. The world works in parallel, and it is time for computer programs to do the same... In less than a decade, a programmer who does not «Think Parallel» first will not be a programmer." [Reinders, 2010]

1.2 Program Dependence

One of the major issues in parallelizing applications is to deal with the underlying inherent dependency structure of the program. Data dependency graphs can abstract how parts of a computation depend on data supplied by other parts. This served as a basis for parallelizing compilers [Banerjee et al., 1993; Wolfe, 1996], and proved that the idea of embedding a program's communication structure into the hardware topology was a reasonable approach. However, automatic dependence analysis is difficult for the general cases, and as a result parallelizing compilers cannot make the most of the underlying dependencies.

The constructive recursive (CR) approach proposed by Haveraaen [2000] allows the modular separation of computation from its dependency, such that both become programmable independently from each other. Dependencies are captured by algebraic abstractions – Data Dependency Algebras (DDA) – and turned into explicit, programmable entities in the program code. DDA-abstractions can also be used to describe hardware communication topologies. Then mapping the computation to a target architecture can be dealt with at a high-level, using DDA-embeddings.

This dissertation presents a framework for portable parallel programming based on DDA-abstractions. Two main issues of parallel computing are addressed: how to map efficiently computations to different parallel hardware architectures, and how to do this at a low development cost, i.e., without rewriting the problem solving code. Today's most widespread parallel programming paradigms and APIs encourage programmers to disregard the underlying hardware architecture in the name of "user friendliness"

– a line supported by most hardware vendors. Our proposed approach, on the other hand, gives direct access to the hardware communication structure but at a high-level. Direct access to aspects of the hardware model is needed by some architectures, e.g., graphics processors. However, the model is fully portable and not tied to any specific processor or hardware architecture, due to the modularisation of the data dependencies.

1.3 A DDA-based Parallel Programming Model

In the CR approach, the key element in the process of separating the computations from their dependencies is played by DDAs. Here, we abstract from the details of the separation methodology, and refer to Chapters 4, 5 and 6 for more details.

DDAs can also abstract over hardware communication layouts. Then the inherent flexibility of DDAs allows us to deal with embeddings at a high-level. A discussion of embeddings and ways to combine DDAs can be found in [Anderlik and Haveraaen, 2003; Haveraaen, 2009].

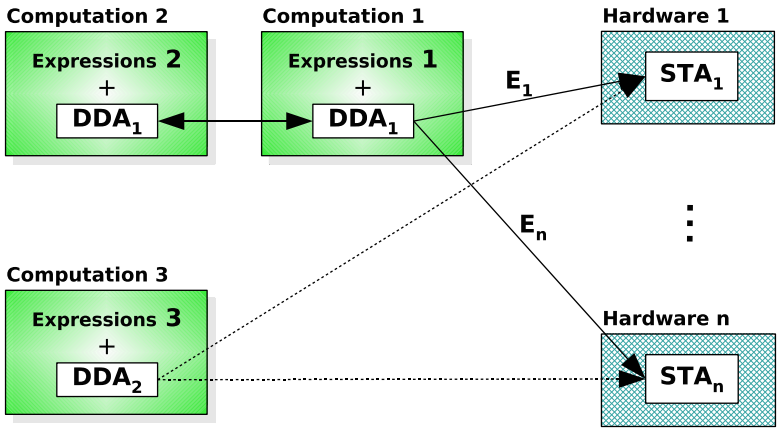

Figure 1.1: *A Hardware Independent Parallel Programming Model*

The conceptual overview of our programming model is presented in Fig. 1.1. The key elements of the model are:

- The data dependency pattern of a computation captured in the form of a *DDA* (e.g. $\texttt{DDA}_1$). In practice, this means that the data dependency graph of the computation is expressed in the formalism of DDAs.
- The computation reformulated as *expressions* on the points of the DDA, such that dependencies between computational steps (DDA points) become explicit entities in the expression.
- A hardware architecture's space-time communication layout or API also captured by a special *space-time DDA* (STA) (e.g. $\texttt{STA}_1$).
- The *embedding* of the computation by the means of a DDA-embedding (e.g. $\texttt{E}_1$, $\texttt{E}_n$). This is a task of finding a mapping of the computation's DDA onto the space-time DDA of the hardware.
- A *DDA-based compiler*, which for a given computation (e.g. computation 1) needs to be fed with:

  1. the computation in terms of $\texttt{DDA}_1$ and the expressions over $\texttt{DDA}_1$
  2. the space-time DDA of the chosen hardware, e.g., $\texttt{STA}_1$
  3. the embedding $\texttt{E}_1$ from $\texttt{DDA}_1$ to this space-time, $\texttt{STA}_1$

  Then the compiler generates code for the chosen hardware architecture. This can for instance be sequential code, CUDA code for GPUs, or vectorized C/C++ for the CELL/BE, and so on.

Characteristics

The model assumes that the space-time DDAs of the hardware architectures are predefined (e.g. $\texttt{STA}_1$,$\texttt{STA}_2$,..., $\texttt{STA}_n$), and are associated with computational mechanisms in the DDA-enabled compiler. The programmer's task is to define the DDA of a computation, re-formalise the computation in term of this DDA, and define an embedding into the space-time DDA of the target architecture. For example, $\texttt{E}_n$ from $\texttt{DDA}_1$ to the space-time DDA $\texttt{STA}_n$. The expressions on $\texttt{DDA}_1$ remain unchanged irrespective of the available hardware resource.

Since there is no need to rewrite the program solving code, the computation is *hardware independent* and *portable*. Also, there is no need for the compiler to do advanced parallelizing program analysis, as the embedding gives all the information needed for efficient parallel code generation. Alternative embeddings can easily be tested in search for optimal solutions,

since each embedding is defined explicitly, yet at a *high, easy to manipulate, level*.

The DDA-based programming model allows other kinds of *software reusability* as well. Different computations may exhibit the same dependency pattern ($\texttt{DDA}_1$ in Computations 1 and 2), in which case all the embeddings defined for $\texttt{DDA}_1$ onto the different architectures can be reused. If a new computation exhibits a new dependency pattern (e.g. $\texttt{DDA}_2$ in Computation 3), all space-time DDAs, and associated execution models, are still available in the compiler, only new embeddings need to be defined into these, illustrated by dashed lines in Fig. 1.1.

DDAs, STAs and DDA-embeddings are discussed in Chapter 4. Language constructs for defining the computational expressions on DDA points are presented in Chapter 5. DDA-based execution models associated with space-time DDAs of different hardware architectures are also presented in Chapter 5. Examples instantiating these ideas are provided in Chapter 6. Mechanisms that promote automatic program refactoring in the compiler are presented in Chapter 7.

1.4 Contribution

The contributions of this dissertation are:

- it illustrates the role that DDAs can play in parallel computing by:
  - giving a fresh perspective of the DDA concept
  - demonstrating how the abstractions available in DDAs can control spatial placements of computations at a high-level
  - showing how modern parallel hardware architectures' communication structure, such as GPUs, can be abstracted in terms of space-time DDAs
  - further expanding the general theory of DDAs
- it presents the foundations of a DDA-based hardware independent parallel programming model
- it defines novel language constructs in accordance with the proposed programming model, and presents associated computational mechanisms for various hardware architectures (sequential, shared memory, GPUs and FPGAs)
- it formalises novel algebraic properties of DDAs that promote program refactoring

- it reports on preliminary practical results which underline that DDA-based embeddings of computations can be dealt with at high-level, and that DDA-based programming is portable

1.5 Published Results

This dissertation incorporates the following published works:

A Hardware Independent Parallel Programming Model, co-authored with Magne Haveraaen, and published in the *Journal of Logic and Algebraic Programming* by Elsevier Inc. [Burrows and Haveraaen, 2009b]. This is a joint work, with the basic ideas originating from Magne. It presents a significant contribution to the work previously published on DDAs, e.g., [Čyras and Haveraaen, 1995; Haveraaen, 2000, 1990a].

Dependency-Driven Parallel Programming, co-authored with Magne Haveraaen, and published in the *Proceeding of Norsk Informatikk Konferanse (NIK)* by Tapir [Burrows and Haveraaen, 2009a]. I am the main author with the work being supervised by Magne. The paper revisits the concept of DDAs in a gentle way, illustrates the results of the visualization tool, and comments on practical experiments.

1.6 Outline

This dissertation is organised as follows:

- **Chapter 1** (this chapter) presents the conceptual overview of the proposed hardware independent parallel programming model.
- **Chapter 2** reflects on the history of data dependency, focusing on how dependence analysis influenced parallel computing research, and discusses some high-level parallel computing research directions of recent years.
- **Chapter 3** gets the reader familiarized with mathematical terminologies used in the formal presentations, and introduces program code style conventions used in the program code examples.
- **Chapter 4** revisits the formal definition of DDAs, giving a fresh perspective and understanding of what DDAs are, significantly expands the study of space-time DDAs and DDA-projections, presents new DDA concepts, related properties and relevant theorems, and defines new DDA examples.

- **Chapter 5** proposes new language constructs, with precise syntax and semantics, designed to encapsulate a DDA-based computational expression, and presents various execution models, targeting different hardware architectures, which compute the semantics of the proposed constructs.
- **Chapter 6** illustrates the essence of DDA-based programming by elaborating DDA-based solutions of well-known computational problems, and shows how these can be mapped to various hardware architectures at a high-level.
- **Chapter 7** gives a formal presentation of novel mechanisms that allow the building of compound DDAs from existing ones. The techniques presented promote program refactoring.
- **Chapter 8** discusses the achieved results, draws some conclusions, and points towards future directions.

CHAPTER 2

Conquering Parallelism

Parallel computing research is about half a century old now. Throughout this period, most research directions aiming at tackling parallelism, in one way or another, always reflected the actual state-of-the-art of (parallel) computing systems. The steady developments of semi-conductor technology have influenced a great deal both the intensity and the enthusiasm at which researchers aimed at conquering parallelism.

Integrated circuit design had gone through a tremendous change. Moore [1965]'s prediction about the promising future of integrated circuits had been criticized, analysied and reformalized several times throughout the 70's, to adapt to the developments of semi-conductor industry. The 80's, however marked a breakthrough. The appearance of Very-large-scale Integration (VLSI) in circuit design allowed thousands of transistors to be combined into a single chip.

The appearance of first parallel systems, the vector processors of the 60-70's, brought forth much energy and launched a very optimistic and intense drive in automatic parallelization research. Much of this work looked back on the theoretical results of the 50's and 60's, which had already addressed some aspects of parallel and concurrent processing. The enthusiasm continued throughout the 80's, since despite the developments of VLSI uni-processor clock frequency did not increase as rapidly as expected. Supercomputers, on the other hand, were available, and many minisupercomputers began also to appear. This highly motivated parallel computing research, as increasing computing power throughput was primarily expected from exploiting the parallelism available on these machines.

The 90's ultimately justified Moore's law, as uni-processor speed began steadily to double every two years. This made uni-processors very appealing and more promising for achieving high computational throughput, for the masses at least. As a result, the driving force of parallel computing research has somewhat decreased, and became more of an interest for scientific computing communities.

By the end of 80's the focus from automatic parallelization research shifted towards programming language-based methods to explore parallelism. At the end of 90's, these efforts led to the standardization of OpenMP and MPI, which became the two dominant and very acceptable programming models for high-performance systems, even up to today.

The beginning of the millenium, however, marked a new era in parallel computing research. In 2005, Moore himself pointed out that his law cannot continue for ever, there are physical limits in the semi-conductor industry which ultimately cannot be pushed further to continuously increase uni-processor speed. As a result, hardware vendors started to build multi-core architectures. The technology was ready, but programming methodologies were lacking. This reinvigorated parallel computing research and made it strive like never before, as parallelism had to be faced now on a daily basis.

In their excellent discourse [Asanovic et al., 2006] on the state-of-the-art of parallel computing research, Berkeley researchers argue that current programming methodologies may be sufficient for systems with up to 8 processors, but they are not likely to scale beyond that. Inspired by the success of parallelism at the extremes of the computing spectrum, i.e., embedded computing and high-performance computing, the report suggests several design targets that a programming model should meet. The target is set to 1000 cores per chip, but programming models should not depend on the number of processors. Models should allow the user to indicate locality, and should support a wide range of data types. They should support well-known forms of parallelism: data-parallelism, task-parallelism, and instruction-level parallelism. Finally, instead of traditional benchmarks, they suggest new methods, borrowed from scientific computing, to design and evaluate programming models and architectures.

In the following two sections, we address only two slices of the vast topic of parallel computing research. The first, reflects on the history of data dependency, showing why dependence analysis became important, and how it influenced parallel computing research as a whole. The second presents some higher level parallel programming models that have emerged in the multi-core revolution era, and relates some of these approaches to our framework.

2.1 A Retrospective of Data Dependency

Early research on the theory of compiling high-level languages for high performance parallel systems was primarily based on program transformations. Dependence analysis provided execution-order constraints between program statements and as such served as a basis for establishing legitimate ways to carry out such transformations. The notion of *data dependency* describes one class of dependencies obtained throughout the process of dependence analysis.

Using data dependencies for program parallelization has a long and speckled history [Bacon et al., 1994; Wolfe and Banerjee, 1987]. Automatic parallelization, loop transformations, systolic arrays, dataflow programming, etc., are all connected to the notion of data dependency, which is fairly wide-spread in compiler design communities. The notion of *data dependency algebra* (DDA) – the key concept of this dissertation, is however less known.

In a general sense, dependencies are considered inherent in a program and low-level artifacts of the Von Neumann machine. As automatic dependence analysis proved to be too difficult for the general cases, the common understanding is that the limits of dependence analysis cannot be pushed much further to provide improved level of abstraction at which to think about parallelization.

The concept of DDA, on the other hand, points into a new direction. When the data dependency is made explicit in the program code (by means more expressive than just simple annotations), a parallelizing compiler can omit data dependence analysis as a whole, and yet harness directly the driving force of the dependency. Hence, DDAs increase the potential that data dependencies can play in program parallelization.

2.1.1 *Automatic Parallelization*

At the dawn of computing when programs were written in the first high-level programming languages, the programmer naturally assumed that the results would be obtained by the machine executing the program statements in the order of their appearance, eventually, obeying any additional control flow constructs, such as if statements, goto-s, or similar branching mechanisms. Programs ran on primitive uni-processors, and there was no need for reordering the statements nor for identifying potential parallelism between the operations. When microprocessor technologies have become more advanced, with more sophisticated memory models, and the appearance of parallel computers, however, made compilers face fundamentally new challenges. For instance, when the micro-architecture was enabled to support

instruction pipelining, compilers had to explore instruction-level parallelism to make the most of the processors' instruction pipelining ability. Exploring data- and task-parallelism, however, placed a significantly heavier burden on both application developers as well as compiler designers.

For the programmer, the most appealing solution was the idea of *automatic parallelization*, when all the hassle of the parallelization process is cast on the compiler. The debate, however, was whether automatically generated parallel code could (ever) achieve the same performance as a handcoded parallel version.

The 1970's mark the beginnings of automatic parallelization research which set off as a joint effort to address portable programming on vector processors [Banerjee, 1976; Lamport, 1974; Loveman, 1976]. Hence this kind of parallelization is often referred to as *automatic vectorization*. A comprehensive overview of automatic program parallelization techniques is presented in [Banerjee et al., 1993]. In the search for program restructuring techniques, the technological tool developed was based on *dependence analysis* [Banerjee, 1988, 1996]:

> "The aim of dependence analysis is to gather useful information about the underlying dependency structure of a program and present it in a suitable form. This analysis can be performed at various levels: we may study dependencies between program statements, iterations of a loop nest, subroutines in the program, etc." [Banerjee, 1996]

The compiler, based on a set of constraints, called *dependencies*, identifies when the reordering of program statements would not change the overall meaning of the program. These constraints are determined by the order in which program statements appear. The early work of Bernstein [1966] had a significant impact on the methods that were developed in the 70-80's.

Two major classes of dependencies have been identified: *control* and *data dependency*. These are usually represented as directed graphs in the compiler. An in-depth coverage of both methods can be found in [Wolfe, 1996] and [Kennedy and Allen, 2002]. When parallelizing or restructuring programs, as a general rule, both data and control dependencies need to be considered.

Control dependency relations provide a rather general method to capture the essential conditions controlling the execution of a program. Intuitively, control dependency occurs in a situation when a statement is executed if a previous statement evaluates in a way that allows its execution. For instance, in the following example:

```
S1: if a<>0
S2:    b = b/a;
S3: c = 2;
```

the statement S2 is executed only if the execution of S1 evaluates to true. Hence S2 is control dependent on S1, whereas S3 is control independent.

Control dependency graphs are usually much larger than the associated control flow graphs. Two principal strategies were introduced to deal with control flow. The first, referred to as *if-conversion* in the literature, eliminates control dependencies by converting them into data dependencies. The second approach counts the control dependency as an extension of the data dependency by including edges of the control dependency graphs into the data dependency graph. These strategies were motivated by the fact that, during program analysis, it was easier to consider and argue about a single dependency graph.

The primary development regarding the use of control dependency in program transformation began with the work of Ferrante et al. [1987], and the framework for building efficient control dependency graphs originates from Cytron et al. [1991]. Various graph algorithms [Aho and Hopcroft, 1974] have been used to argue about the properties of control dependency graphs relevant in the context of program transformations and actual code generation processes, for example, finding strongly connected components of a directed graph. Significant contributions for the code generation process in the presence of control dependency can be found in [Kennedy and McKinley, 1990; McKinley, 1992].

Data dependency relations, on the other hand, ensure that data is provided and used in a correct order. These are less restrictive than control dependency relations, and give more flexibility in the transformation process. However, the order of memory references can be changed only to the extent that wrong values are not used or stored when the rearranged program is executed [Wolfe, 1996].

Three main classes of data dependency constraints are distinguished. Let S1 be a statement and S2 a subsequently executed statement. Then:

- S2 is *flow (or true) dependent* on S1, if S2 uses a variable which is assigned or previously modified in S1.
- S2 is *anti-dependent* on S1, if S2 reassigns a variable which has been used in S1.
- S2 is *output dependent* on S1, if a variable assigned in S1 is reassigned in S2.

The main data dependence problem for the compiler is to determine if two variables have instances that reference the same memory location during program execution [Banerjee, 1996]. The variables considered are usually either scalars or array elements in loop-iterations. In the latter case, the subscript expressions of the arrays are considered in the constraints [Allen, 1983]. The more complex the array subscripts are, the more complicated the dependence analysis becomes. Most accurate results could be obtained when the array subscripts were affine functions of the loop variables and they obeyed some additional properties [Maydan et al., 1995; Wolfe and Banerjee, 1987]. But in some cases, the constraints just simply could not be solved at all at compile-time, the compiler not having enough information.

The presence of procedure calls complicated significantly the process of dependence analysis [Burke and Cytron, 1986; Li and Yew, 1988]. Several early papers addressed the issues arising with interprocedural calls [Callahan et al., 1986; Cooper et al., 1986], but later new methods have been presented [Burke and Cytron, 2004; Müller-Olm, 2004] which solved many of these issues.

Due to these limitations, automatic vectorizers of simple sequential programs did not succeed in providing very significant high-performance. Nonetheless, the techniques based on dependence analysis became standards on most vector machines by the end of 1980's [Allen and Kennedy, 1987; Polychronopoulos et al., 1990; Wolfe, 1990].

With the appearance of shared and distributed memory architectures, automatic parallelization research turned towards exploiting these systems as well [Allen et al., 1988, 1987; Banerjee et al., 1993; Callahan and Kennedy, 1988; Kuck et al., 1980; Zima and Chapman, 1993]. Several additional aspects had to be considered, e.g., how to minimise the overhead of initiating and synchronizing parallel threads on shared memory systems, or how to partition data to the memories of the processors in distributed-memory systems with optimal communication [Amarasinghe and Lam, 1993], etc.

Dependence analysis became increasingly more complex, leading to long compiler running times, and the results did not demonstrate significant success as far as automatic parallelization was considered [Banerjee et al., 1993]. The techniques developed, however, had a great impact on the optimization techniques applied in most modern compilers [Kuck et al., 1980, 1981; Padua and Wolfe, 1986].

At the end of the 80's, the automatic parallelization enterprise came to the conclusion that the long awaited high-performance was more likely to come from a joint effort [Dongarra et al., 2003, p. 361]. While still exploiting methods for automatic parallelization, the programmer should also

provide some useful information at the level of the program code. Hence, parallel programming language design began to explore language-based strategies that provided the compiler with additional information about data-decomposition, task- or data-parallelism. Some of these early research efforts led, for instance, to the standardization of High-Performance Fortran [Loveman, 1993] – a high-level data-parallel language based on Fortran –, which served as a basis for the upcoming generation of high-level parallel languages developed for high-performance systems, e.g., Fortran 95, OpenMP. The focus on language-based approaches, in the same time, naturally raised portability issues. Explicit representation of parallelism in the source code did not guarantee optimal use of parallel hardware. If one way of expressing parallelism worked well on one architecture, this most likely needed to be altered when ported to another system due to performance-related considerations.

2.1.2 *Restructuring Compilers*

As loops tended to be the most time-consuming parts of the programs, program restructuring methods were first highly focused on these. *Loop-based transformations* promised to deliver increased parallelism as well as optimized running time. The first high-level parallel language constructs were also designed to express loop-level parallelism, for the same reason.

Loop-transformations were based on elaborate *dependence tests* of entire loop nests and across subscripts of possibly multiple arrays. Based on these methods, several strategies have been developed for various loop-level transformations, e.g., loop interchange, loop skewing, loop fusion, loop reordering, loop unfolding, loop tiling etc.

In the general cases, these tests became very complex. Low-complexity tests, luckily, produced quite accurate results in many cases, such as the *GCD test* [Psarris, 1996], the *I-Test* [Kong et al., 1991], and *Banerjee's Inequilities* [Banerjee, 1979; Psarris, 1992]. The *Omega test* [Feautrier, 1988; Pugh and Wonnacott, 1992], a constraint-based analysis, was an independent effort to improve array dependence analysis which provided exact results for affine cases. While it was exponential for the general cases, it often provided faster results than its fellow competitors for the common cases. The Omega test is often related to the *polyhedral model* [Pouchet et al., 2008]. This is a mathematical framework based on affine transformations of dependencies, useful in program restructuring, such as tiling [Gupta et al., 2007].

While the idea of purely automatic parallelization did not succeed the way it was hoped for, the vast amount of knowledge gathered about de-

pendence analysis contributed a great deal to building powerful optimizing compilers. [Bacon et al., 1994] is a concise and comprehensive survey of the state-of-the-art of dependence-based transformations of the pre-Fortran 95 era. Wolfe [1996] provides a more technical presentation, and Kennedy and Allen [2002] discusses further developments. Dependence-based approaches have remained crucial in advanced program analysis and transformation techniques in modern compiler design [Srikant and Shankar, 2007].

Research directions for developing more accurate and faster algorithms for dependence tests are still noticeable. A recent comparison of data dependence analysis tests can be found in [Viitanen and Hämäläinen, 2004]. A new dependence analysis tool is described by Kyriakopoulos and Psarris [2005], which empirically is proved to be more efficient in program parallelization than usual dependence tests. Zhou and Zeng [2006] present a more general scheme of dependence test based on integer interval theory to solve difficult dependence test problems.

2.1.3 *Data-driven Paradigms*

Also inspired by the inherent data and flow dependency of computations, the 1970's brought forth other approaches which meant to exploit parallelism in a different way. As hardware were becoming (relatively) less expensive, special purpose micro-architectures began to appear especially designed for signal and image processing. Based on these experiments, new data-driven architecture models evolved as a contrast to the von Neumann instruction-driven paradigm.

The term *systolic array* was coined by Kung and Leiserson [1978]. A systolic system was presented as a regular pipe network arrangement of processors, called *cells*, which computed and passed data through the system rhythmically. Hence the name analogy with the regular pumping of blood by the heart. Since the data was organised as multiple data streams, the underlying idea well-supported data parallelism, and the regularity of the network ensured scalability.

Systolic array research primarily focused on designing algorithms and architectures that laid out well in two dimensions so that they could adapt well to VLSI technology restrictions [Kung, 1982]. The shape of the cell-network varied, e.g., rectangular, triangular or hexagonal, each exploiting higher degree of parallelism in a different way. The actual interconnects between the cells modeled the dataflow of the algorithm that the systolic array was built for. Hence exploiting the flow and data dependency of an algorithm was a key element in the design of systolic arrays, e.g.,

[Jen and Kwai, 1992; McCanny et al., 1990]. Miranker and Winkler [1984] presented a general theory for characterizing and realising algorithms in hardware. The approach was based on data dependency graphs of computations embedded into space-time representations of hardware. The technique developed described the mapping of a particular systolic algorithm into a physical array.

Many systolic algorithms have been proposed, e.g., for matrix arithmetic, data- and graph-algorithms, signal and image processing. They had the potential of taking an exponential algorithm and turn it into a linear-time hardware implementation. However, the building of a systolic array parallelizing compiler that could translate a high-level scientific code for systolic arrays remained challenging.

Despite their great potential for massive parallelism, high throughput, and gradual support for SIMD organizations for vector operations and MIMD for non-homogeneous parallelism, systolic arrays were expensive and hard to build. Due to their inherent characteristics, systolic architecture design, nonetheless, has prevailed, e.g., in reconfigurable computing.

The *dataflow machine* was another data-driven programmable computer which had a specialised hardware optimised for fine-grain data-driven parallel computations. The concept originates back to the 60's when dataflow graphs were first developed [Davis, 1979]. The first implementation of a dataflow machine points back to 1976 [Davis, 1979]. Dataflow architectures disregard from the general notion of program counter, and promote explicit data-driven execution. A survey of early machines can be found in [Veen, 1986]. Dataflow languages have also emerged (e.g., Sisal, Lustre, Lucid, VHDL, etc.).

Binaries compiled for a dataflow machine contain dependency information, which is usually not common to binaries. This information is utilised when the program is executed on the machine, and by a special mechanism it can decide over code segments that can be executed in parallel.

Dataflow machines have been primarily implemented as specialized hardware used in digital signal processing, network routing and graphics processing.

2.1.4 *Parallel Functional Languages in the 90's*

Under the flagship of automatic parallelization research, functional languages also emerged as potential candidates. The basic computational abstractions of a functional language are functions. Since a functional program has no such notion as execution state and has a value-oriented semantics, it usually requires a simpler dependence analysis, at the cost of more complex

data allocation and modification operations. The key question in parallel functional languages was how much parallelism can be extracted automatically and how much should be provided by the programmer. Data partitioning was one of the issues that required additional information. A nice compilation of parallel functional compiler directions of this period can be found in [Szymanski, 1991], including PTRAN, the HPF compiler. Some of these were heavily based on dependence graph transformations in the code generation process.

In particular, EPL and Crystal followed a transformation strategy based on equational annotations, aiming at generating optimized and efficient implementations for a variety of hardware. The optimization of Crystal programs followed a series of systematic transformations of data index-spaces, automated by the compiler, until an optimal data distribution was found which matched the physical characteristics of the given hardware. Both languages hinted that modelling the hardware architectures' physical properties at a higher abstraction level in the compiler would be beneficial in the code generation process.

The book [Szymanski, 1991] also exposes the philosophy of dataflow programming by presenting Sisal, Lucid and Id.

2.1.5 Explicit Data Dependency

As mentioned earlier, Miranker and Winkler [1984] introduced a general theory about embedding the data dependency graph of a computation into the space-time topology of a systolic array. Haveraaen [1990b, 2000, 1990a] took this underlying idea further when developing the *constructive recursive* (CR) approach. This allowed the modular separation of computation from its dependencies, such that both became programmable independently from each other. This also entailed a separation between local dependencies of a function, and the global communication pattern of the whole computation in the hardware. The idea showed many similarities with the *structural blanks approach* of Čyras, and both techniques were presented and compared in [Čyras and Haveraaen, 1995].

Haveraaen's approach was based on algebraic abstractions, referred to as *Data Dependency Algebras* (DDA). DDAs provided a formalism to express data dependencies as explicit, first class, programmable entities in the program code. This makes Haveraaen's approach rather unique, compared to previous frameworks. The extracted data dependencies are primarily true (flow) data dependencies.

As Banerjee [1996] put it, the aim of dependence analysis was to gather useful information about the underlying data dependency structure of a

program and present it in a suitable form. A DDA delivers exactly this information to the compiler, and it comes from the effort of the programmer, and not of the compiler. The programmer analyses the underlying dependency by the means of pencil, scratch paper and human intelligence. If the attempt is successful, the result is a concise, meaningful representation of the data dependency pattern of the computation in the program code, ready to be used by a parallelizing compiler.

The prototype compiler implemented by Søreide [1998] followed an optimised recursion unfolding strategy in the code generation process. This was entirely based on the information provided in the DDA. The compiler adapted a technique inspired by tail recursion optimization, as common to many functional languages, developing it to arbitrary, not just linear, dependency patterns.

Later, Burrows and Haveraaen [2009b] presented a unified programming model based on the basic idea of CR, showing that DDAs provide a high-level, flexible and hardware independent formalism to deal with parallelism. This served as starting point for this dissertation.

2.2 High-level Approaches in the Multi-Core Era

Research directions of earlier times are often revisited. In hindsight it is often easier to understand why a very promising approach did not succeed. High-Performance Fortran (HPF) is one of these [Kennedy et al., 2007]. HPF was accepted with great enthusiasm in the early 90's. However, the initial excitement has faded out. This is attributed to reasons that are worth learning from: immature compiler technology leading to poor performance, lack of flexible distributions, inconsistent implementations, missing tools, and lack of patience by the high-performance computing community. Although HPF itself failed to achieve success, it had a great impact on the development of high-level parallel languages (e.g. Fortran 95, OpenMP, Chapel, Fortress, X10).

In recent years, several new programming models have been proposed, and in some cases implemented. However, the underlying basic ideas point back to approaches from decades ago. In general, they come in either a task parallel or data-parallel fashion, though the latter seems to outweigh the former as to popularity. This should not come as a surprise, since data-parallelism, by its nature, scales better with growing number of processors.

Some of the current programming models targeting multi-core and/or GPU programming in imperative style are:

- CellSs [Perez et al., 2007] has been proposed as an alternative task parallel programming model for multi-core processors, and its current implementation targets IBM's CELL/BE processor. It is based on the automatic exploitation of the functional parallelism of a sequential program through the different processing elements of the Cell/BE architecture. It is based on annotations to a sequential code, similar to that of OpenMP.

- Also on the task parallel front, with Intel's TBB library [Reinders, 2007] one can express parallelism in a C++ program. It is based on a higher-level, task-based parallelism that abstracts platform details and threading mechanism for performance and scalability.

- Intel's Array Building Blocks [Intel, 2010] provides a generalized vector parallel programming solution that abstracts from the dependencies on particular low-level parallelism mechanisms or hardware architectures. It produces scalable, portable, and deterministic parallel implementations from a single high-level source description.

- RapidMind [Monteyne, 2008] is a development and runtime platform to a variety of architectures, including GPUs, the Cell/BE, and multi-core CPUs. It is a data-parallel programming model which takes a high-level abstraction of parallelism and maps it to what is available. A recent article discusses its limitations [Christadler and Weinberg, 2011]. RapidMind is now part of Intel, and the technology is integrated into Intel's Array Building Blocks.

- Microsoft Research's Dryad [Isard et al., 2007] is a general-purpose distributed execution engine for coarse-grain data-parallel applications. It combines computational vertices with communication channels to form a dataflow graph. The user can specify a directed graph to describe the application's communication patterns between the computation vertices.

- A heterogeneous data-parallel computational model for the Cell/BE has been recently presented in [Li et al., 2008]. It proposes to aggregate the computing power of the two different processing elements of the Cell/BE, i.e., of the synergistic processor elements, and of the heavy duty processor element.

- Huckleberry [Collins et al., 2010] is an experimental tool which automatically generates parallel implementations for multi-core platforms from sequential recursive divide-and-conquer programs.

- The Sequoia [Fatahalian et al., 2006] programming language facilitates the development of memory hierarchy aware parallel programs that remain portable across modern architectures and controls data locality. It provides language mechanisms to describe communication vertically through the machine and to localize computation to particular memory locations within it.
- The partitioned global address space (PGAS) [PGAS, 2010] is a relatively new parallel programming model which serves as a basis for several new high-level parallel languages, such as Unified Parallel C, Co-array Fortran, Fortress, Chapel and X10. The model assumes that the global memory space is logically partitioned. Each portion becomes local to each processor, and it may also have an affinity for the particular thread running on the processor, thereby exploiting data locality.
- ParaMeter [Kulkarni et al., 2009] is a tool which can determine how much parallelism is latent in irregular programs wich exhibit amorphous data-parallelism. It produces parallelism profiles for the programs in question, and the methodology used is based on graph-representations.

Declarative approaches also tackle multi-core and GPU programming. Since declarative languages offer a pure view of computation, such settings are suitable to express data-parallelism in high-level descriptions [Lisper, 1996].

- Singh [2008] argues about the importance of the ability to target different computing devices from the same description. He shows how GPUs and FPGAs can be targeted from a single data-parallel description, based on higher order functions and polymorphism.
- Data Parallel Haskell [Chakravarty et al., 2007] aims to implement the programming model of nested data-parallelism into the Glasgow Haskell Compiler, by extending NESL [Blelloch et al., 1994] in terms of expressiveness and efficiency. NESL is a portable nested data-parallel language, appeared in the early 90's, which allows high-level, concise descriptions of nested data-parallel programs.
- Obsidian [Svensson, 2011] is a data-parallel language embedded in Haskell which targets GPU programming. The current version is implemented for NVIDIA's CUDA. The programming style is inspired by Lava combinators [Bjesse et al., 1999], high-level structural hardware design elements.

- Skeletal parallel programming evolved from the idea of higher-order function applications [Cole, 1989]. *Skeletons* ("higher-order functions") are high-level parallel constructs, which describe computational structures without over-specifying the details. They come as library packages with efficient implementations for the different parallel systems. The programmer identifies the patterns that fit his need, and customizes them by filling in the "arguments". These can be other functions/procedures to which the skeleton is applied to produce the problem specific final program [Cole, 2004; Rabhi and Gorlatch, 2003].

- Feldspar [Axelsson et al., 2010] is a domain-specific language, embedded into Haskell, which enables high-level and platform-independent description of digital signal processing algorithms. It offers a high-level dataflow style of programming.

Few, if any, of these programming models represent fundamentally new ideas. Task parallelism goes back to the 60's idea of concurrent processes [Hansen, 1973] with shared memory. Message passing style parallelism goes back to the mid 70's [Hoare, 1978] and its foundation has been investigated in several process algebras from the 80's onwards. Data-parallelism evolved from systolic arrays, and was a firmly established approach by the mid-80's [Hillis and Guy L. Steele, 1986] when it was the main programming model for some of that time's high-end parallel machines. A wide variation of early parallel programming models are surveyed in [Baer, 1973], and a good literature list can be found in [Hibbard, 1980].

Our approach is data-parallel by considering the entire computation at once. We emphasize modular separation of the computational expressions from the data dependency, and separation of the dependency from its embedding onto hardware. The latter allows us to fully control resource usage, including data locality. This separates us from most functional approaches, though our expression-oriented notation would be considered functional.

The focus on dataflow graphs is in common with the ParaMeter and the Dryad approach. In the latter, the graphs, in response to some events that occur during the computation, can change dynamically. Our dependency graphs are fixed throughout the whole computation, and the DDA-based approach targets fine-grain data-parallelism.

The dataflow execution model, originally developed to exploit massive parallelism [Johnston et al., 2004], had a great effect in the development of dataflow visual programming languages. In this model a program is represented as a directed graph. The nodes of the graph are primitive instructions such as arithmetic or comparison operations, and the arcs between

the nodes represent the data dependencies between the instructions. This makes the dataflow programming model related to ours, however, it differs in one fundamental aspect. In the dataflow programming model, the main concept behind any program is the *data*. In the DDA-based approach, it is the *dependency* which is promoted as first class citizen.

Automatic parallelization techniques of regular, loop-based applications, based on compile time dependence analysis [Banerjee et al., 1993], are also related to our approach. It has also been known for some time that compilers can parallelize divide-and-conquer programs by analyzing memory references to detect dependencies [Gupta et al., 2000; Rugina and Rinard, 1999]. The above mentioned Huckleberry tool is also based on this technique, and is related to our approach to the extent that it also utilises data dependency information in the parallelization process. However, in DDA-based parallelization, the compiler is fed with the data dependency, and need not bother about advanced parallelizing program analysis. The information given in the DDA is sufficient for efficient parallel code generation, and offers more flexibility when it comes to mapping the computation onto different parallel hardware architectures. In addition, the DDA-based approach is more general, and not tied to only divide-and-conquer algorithms and loop-based applications.

CHAPTER 3

Preliminaries

Before embarking on the formal presentation of our subject, we recall the mathematical notions relevant in the context of this dissertation, fix the notations, and introduce program code style conventions together with some data type abstractions assumed to be present in the examples. The reader is assumed to be acquainted with elementary set and graph theory as well as general programming language concepts.

3.1 Mathematics

3.1.1 Sets

By *set* we mean a collection of elements. Some particular sets are:

- the set of natural numbers, denoted by the symbol $\mathbf{N} = \{0, 1, 2, \ldots\}$
- non-zero elements of $\mathbf{N}$, denoted by $\mathbf{N}^+ = \{1, 2, \ldots\}$
- set of real numbers, denoted by $\mathbf{R}$
- set of complex numbers, denoted by $\mathbf{C} = \{a + bi \mid a, b \in \mathbf{R}\}$, where i is the *imaginary unit* with the property $i^2 = -1$
- the two-element set of truth values, denoted by $\mathbf{B} = \{0, 1\}$
- the empty set with no elements at all, denoted by $\{\}$

The *cardinality of a set* A, often denoted by $|A|$, is the *number of elements* in A. A *finite set* has a finite number of elements. A set that is not finite is called *infinite*. A set is *countable* if it has at most the same cardinality as the set of natural numbers. A set that is not countable is *uncountable*. E.g. the set of real numbers is uncountable.

The following notations, operations and related properties are common to sets:

- $A \subseteq B$ – A is a *subset* of B, if all elements of A are also in B
- $A \subset B$ – A is a *proper subset* of B, if $A \subseteq B$ but $A \neq B$
- $A \setminus B = \{a \mid a \in A \text{ and } a \notin B\}$ – the *difference* of A and B
- $A \cup B = \{a \mid a \in A \text{ or } a \in B\}$ – the *union* of A and B
- $A \cap B = \{a \mid a \in A \text{ and } a \in B\}$ – the *intersection* of A and B. If $A \cap B = \{\}$ then A and B are called *disjoint*
- $A_{i,c} = \{\langle a, i\rangle \mid a \in A_i\}$ – the *canonical form* of set A_i wrt. some index-set I, where $i \in I$.
- $A_1 \uplus A_2 = A_{1,c} \cup A_{2,c}$ – the *disjoint union* of A_1 and A_2. The elements of the disjoint union are ordered pairs $\langle a, i\rangle$, where the index $i \in \{1, 2\}$ indicates exactly which set A_i the element a comes from in the pair $\langle a, i\rangle$.
- $A_1 \times A_2 \times \ldots A_k = \{\langle a_1, a_2, \ldots, a_k\rangle \mid a_i \in A_i,\ 1 \leq i \leq k\}$ – the *cartesian product* of the sets A_1, A_2,...,A_k with $k \geq 0$. If $k = 0$ or one of the sets is the empty set, then the cartesian product is also the empty set.

Note that one may use the cartesian product to construct the canonical form of a set A_i, $i \in I$:

$$A_{i,c} = A_i \times \{i\} = \{\langle a, i\rangle | \, a \in A_i\}$$

Theorem 3.1.1. The union, intersection and disjoint union of sets obey the following laws:

- Commutativity:

$$A_1 \cup A_2 = A_2 \cup A_1 \tag{3.1}$$
$$A_1 \cap A_2 = A_2 \cap A_1 \tag{3.2}$$
$$A_1 \uplus A_2 = A_2 \uplus A_1 \tag{3.3}$$

- Associativity:

$$(A_1 \cup A_2) \cup A_3 = A_1 \cup (A_2 \cup A_3) \tag{3.4}$$
$$(A_1 \cap A_2) \cap A_3 = A_1 \cap (A_2 \cap A_3) \tag{3.5}$$
$$(A_1 \uplus A_2) \uplus A_3 = A_1 \uplus (A_2 \uplus A_3) \tag{3.6}$$

Proof: Properties 3.1, 3.2, 3.4 and 3.5 are straightforward from the definitions. The commutativity (3.3) and associativity (3.6) of disjoint union follow from the observation that the disjoint union is obtained through the embeddings of the participating sets' canonical forms, i.e.:

$$A_1 \uplus A_2 = A_{1,c} \cup A_{2,c} = A_{2,c} \cup A_{1,c} = A_2 \uplus A_1$$

and

$$A_1 \uplus (A_2 \uplus A_3) = A_{1,c} \cup (A_{2,c} \cup A_{3,c}) = (A_{1,c} \cup A_{2,c}) \cup A_{3,c} = (A_1 \uplus A_2) \uplus A_3$$

□

These properties lead to the following generalized forms extended over a family of sets $(A_i)_{i \in I}$ for some index-set I:

- Union: $\bigcup_{i \in I} A_i = \{a \mid a \in A_i \text{ for some } i \in I\}$ and $\bigcup_{i \in \{\}} A_i = \{\}$
- Disjoint union: $\biguplus_{i \in I} A_i = \bigcup_{i \in I} \{\langle a, i \rangle \mid a \in A_i\}$ and $\biguplus_{i \in \{\}} A_i = \{\}$
- Intersection: $\bigcap_{i \in I} A_i = \{a \mid a \in A_i \text{ for all } i \in I\}$

3.1.2 Arithmetics

Given $n \in \mathbf{N}^+$, then the *n-th root of unity* is a complex number $z \in \mathbf{C}$ satisfying the equation:

$$z^n = 1$$

The n-th root of unity is *primitive*, if it is not a k-th root of unity for some $k \in \mathbf{N}^+$ such that $k < n$, that is:

$$z^k \neq 1$$

The *Euler formula* establishes a relationship between the trigonometric functions and the complex exponential function stating that for any $x \in \mathbf{R}$:

$$e^{ix} = \cos x + i \sin x$$

where e is the base of the natural logarithm and i is the imaginary unit. The Euler formula is used then to transform the formula for the n-th roots of unity into its most familiar form:

$$e^{2\pi i \frac{k}{n}}$$

This will be a primitive root if and only if the fraction k/n is in lowest terms, that is their greatest common divisor is 1.

3.1.3 Relations

If A and B are sets, then a *relation* between A and B is defined as a subset of their cartesian product, i.e., $R \subseteq A \times B$. When $A = B$, one says that R is a relation *on* A.

We will often use the shorthand $R(a, b)$ for $\langle a, b \rangle \in R$.

A relation R on A is called:

- *trichotomous*, if for all $a, b \in A$ exactly one of the following holds: $R(a, b)$, $R(b, a)$ or $a = b$;
- *reflexive*, if for all $a \in A$: $R(a, a)$;
- *symmetric*, if for all $a, b \in A$: $R(a, b)$ implies $R(b, a)$;
- *transitive*, if for all $a, b, c \in A$: $R(a, b)$ and $R(b, c)$ implies $R(a, c)$;
- *equivalence relation*, when it is reflexive, symmetric and transitive;
- *strict total order*, when it is trichotomous and transitive.

For instance, the relation "$<$" on the set of natural numbers is a strict total order, and so is the standard dictionary order with the letters of the latin alphabet, whereas "$=$" on the set of natural numbers is an equivalence relation.

3.1.4 Functions

If A and B are non-empty sets, and f is a relation between A and B, then f is called:

- a *partial function* from A to B, denoted by $f : A \rightsquigarrow B$, if for every element $a \in A$ there exists no or at most one $b \in B$ such that $\langle a, b \rangle \in f$.
- a *total function* from A to B, denoted by $f : A \to B$, if for every element $a \in A$ there exists exactly one $b \in B$ such that $\langle a, b \rangle \in f$.

One then writes $f(a) = b$, where a is called *argument* and b the *value* of f in a. If f is a partial function, and there exists no $b \in B$ such that $f(a) = b$ then one says that $f(a)$ is *undefined*.

A is called the *domain* of f, and B its *codomain*.

All functions are assumed to be total, unless explicitly stated otherwise.

If A and B are the empty sets, then $f : \{\} \to \{\}$ is called the *empty function*.

A *predicate* is a function with the set **B** as its codomain.

If f is a function with some cartesian product as its domain, e.g. $A \times B$, we may omit the use of bracketing for the elements when they appear as arguments, e.g., we write $f(a, b)$ instead of $f(\langle a, b \rangle)$ for $\langle a, b \rangle \in A \times B$.

Given functions $f : A \to B$ and $g : B \to C$, the *composition* of g to f is the function $g \circ f : A \to C$ defined by:

$$(g \circ f)(a) = g(f(a)) \text{ for all } a \in A$$

Theorem 3.1.2. The composition of functions is associative, i.e., if $f : A \to B$, $g : B \to C$ and $h : C \to D$ are functions then:

$$(h \circ g) \circ f = h \circ (g \circ f)$$

Proof: For all $a \in A$ we have:

$$((h \circ g) \circ f)(a) = (h \circ g)(f(a)) = h(g(f(a))) = h((g \circ f)(a)) = (h \circ (g \circ f))(a)$$

□

The *identity function* of a set A is the function $\mathbf{1}_A : A \to A$ defined by $\mathbf{1}_A(a) = a$ for all $a \in A$.

A function $f : A \to B$ is called an *isomorphism*, if there exists a function $f^{-1} : B \to A$, called its *inverse*, such that $f \circ f^{-1} = 1_B$ and $f^{-1} \circ f = 1_A$. One may also call then the sets A and B *isomorphic*, and write $A \simeq B$.

Theorem 3.1.3. The inverse of an isomorphism is unique. □

Proof: Let $f : A \to B$ be an isomorphism with inverse $f^{-1} : B \to A$, i.e., $f \circ f^{-1} = 1_B$ and $f^{-1} \circ f = 1_A$. Assume there exists also inverse $f' : B \to A$ such that $f \circ f' = 1_B$ and $f' \circ f = 1_A$ also. Then we have for all $b \in B$:

$$\begin{aligned}
f'(b) &= \mathbf{1}_A(f'(b)) && \text{(by def. of } \mathbf{1}_\mathrm{A}\text{)} \\
&= (\mathbf{1}_A \circ f')(b) && \text{(by def. of } \circ\text{)} \\
&= ((f^{-1} \circ f) \circ f')(b) && \text{(assumption)} \\
&= (f^{-1} \circ (f \circ f'))(b) && \text{(by Th. 3.1.2)} \\
&= (f^{-1} \circ \mathbf{1}_B)(b) && \text{(assumption)} \\
&= f^{-1}(\mathbf{1}_B(b)) && \text{(by def. of } \circ\text{)} \\
&= f^{-1}(b) && \text{(by def. of } \mathbf{1}_\mathrm{B}\text{)}
\end{aligned}$$

Hence $f' = f^{-1}$. □

A function $f : A \to B$ is called *injective* if for all $a, b \in A$ with $f(a) = f(b)$ implies that $a = b$. f is called *surjective* if for all $b \in B$ there exists $a \in A$ such that $f(a) = b$. An injective and surjective function is called *bijective* or a *bijection*.

Theorem 3.1.4. A function $f : A \to B$ is bijective if and only if it has an inverse. □

Proof: ($\Rightarrow$) Assume f is bijective. Then we define $g : B \to A$ the inverse of f as follows: for each $b \in B$ there exists $a \in A$ such that $b = f(a)$ (since f is surjective). a is also unique since f is injective. Then let $g(b) = a$. Then $(g \circ f)(a) = g(f(a)) = g(b) = a$ for each $a \in A$. And $(f \circ g)(b) = f(g(b)) = f(a) = b$ for each $b \in B$.

($\Leftarrow$) Assume now that $g : B \to A$ is the inverse of f. Then f is injective since $f(a) = f(b)$ implies $g(f(a)) = g(f(b))$, that is $(g \circ f)(a) = (g \circ f)(b)$, that is, $a = b$. And f is also surjective: Let $b \in B$ arbitrary, and set $a = g(b)$. Then $f(a) = f(g(b)) = (f \circ g)(b) = b$. □

If $f : A \to B$ is a function and $C \subseteq A$, then the *restriction* of f to C is the function $f\big|_C : C \to B$ defined by:

$$f\big|_C(a) = f(a) \text{ for all } a \in C$$

A *binary operator over a set* A is given by a function $f : A \times A \to A$.

3.1.5 Graphs

A *directed multigraph* or *directed graph* is an abstract representation of a set of objects with links between the objects. The objects are referred to as *nodes*, and the links as *edges*. In the discussion of data dependency graphs objects will often be referred to as *points*, and the links as *branches*, as well. Formally, in *classical graph representation*, a directed multigraph is given by a 4-tuple $\mathcal{G} = \langle \mathcal{N},\ \mathcal{E},\ s : \mathcal{E} \to \mathcal{N},\ t : \mathcal{E} \to \mathcal{N} \rangle$ where $\mathcal{N}$ is the set of nodes, $\mathcal{E}$ is the set of edges, and functions $s : \mathcal{E} \to \mathcal{N}$ and $t : \mathcal{E} \to \mathcal{N}$ identify the *source* and *target* nodes of an edge, respectively.

If $\mathcal{G} = \langle \mathcal{N},\ \mathcal{E},\ s : \mathcal{E} \to \mathcal{N},\ t : \mathcal{E} \to \mathcal{N} \rangle$ and $\mathcal{G}' = \langle \mathcal{N}',\ \mathcal{E}',\ s' : \mathcal{E}' \to \mathcal{N}',\ t' : \mathcal{E}' \to \mathcal{N}' \rangle$ are graphs then:

- $\mathcal{G}$ and $\mathcal{G}'$ are called *isomorphic*, denoted by $\mathcal{G} \simeq \mathcal{G}'$, if there exist isomorphisms $\phi : \mathcal{N} \to \mathcal{N}'$ and $\psi : \mathcal{E} \to \mathcal{E}'$ that respect the source and target components, i.e., for all $e \in \mathcal{E}$ the following hold:

$$\phi(s(e)) = s'(\psi(e))$$

$$\phi(t(e)) = t'(\psi(e))$$

- $\mathcal{G}$ is a *sub-graph* of $\mathcal{G}'$, if $\mathcal{N} \subseteq \mathcal{N}'$, $\mathcal{E} \subseteq \mathcal{E}'$, $s = s'\big|_{\mathcal{E}}$ and $t = t'\big|_{\mathcal{E}}$.

3.2 Program Code Style Conventions

The underlying leading mathematical structures and properties are presented using mathematical notations, e.g., see Chapters 4 and 7. However, the examples instantiating these structures for programming purposes will be presented in program code style, using `typewriter face`. Hence, the mathematical notations of the definitions will be adapted, e.g., a set B will be presented as a data type `B`, a mathematical function declaration $r_p : r_g \to P$ of an earlier definition will become a function declaration `rp:rg→P` in program code style, and so on.

3.2.1 Language Constructs

The program codes and language constructs used in the examples are inspired by functional programming notations, and in general should be easily interpreted by anyone who has some familiarity with elementary programming language concepts. For better readability, the reserved words of our coding language will be set in **`bold typewriter face`**.

3.2.2 Guards

Whenever we declare a partial function we will associate a *guard* with it. For instance, `rg` can be seen as the guard of `rp` in any DDA example (see Definition 4.1.1 then some examples for it in Section 4.2). A guard is a predicate that identifies for which arguments the corresponding function returns a well-defined value [Haveraaen and G. Wagner, 2000].

A judicious use of guards – assumed to be managed by the compiler – will allow, for every expression, the automatic establishment of a (syntactic) condition that ensures the well-definedness of the entire expression. By this, we avoid syntactic clutter when writing code, as these implicit conditions are assumed to be checked by the compiler for every expression, as long as the guards have been defined. In the examples arguments for which a partial function returns a well-defined value, in the presence of these guards, will often be referred to as *guarded* or *relevant* arguments in the explanatory comments.

3.2.3 Data Types

The data types of the program code examples will either be simple, tuple or array data types:

1. *simple data types* such as:
   - `Bool` – represents the type corresponding to the two-element set of truth values, consisting of values `true` and `false`;
   - `Nat` – represents the type corresponding to the set of natural numbers;
2. a *tuple data type* is a structured data type declared together with:
   - its *type name*
   - its *components' types*
   - implicit *projection functions*
   - implicit *constructor* for the tuple type
   - and its *data invariant*.

   An example declaration:

   ```
   T = p1 Nat * p2 Nat | DI
   ```

   where the data invariant `DI:T→Bool` is given by:

```
DI(t) = (p1(t)<1000) && (p2(t)<1000)
```

would mean that we introduce a type T with a data structure consisting of two natural numbers as components whose values are limited to natural numbers less than 1000. Note that the data invariant, in this sense, restricts the range of values that are meaningful for the type in a given context. When the data invariant is omitted in a type declaration it is assumed to implicitly hold for all values of the type.

The two projection functions are p1:T→Nat and p2:T→Nat, and the constructor is named T with profile T:Nat,Nat→T. (Giving a constructor the same name as the type it constructs is common in many programming languages.)

The constructor and the implicit projections of any tuple type are related by the following *consistency requirements*, where n1 and n2 are of type Nat and t is of type T:

```
T(p1(t),p2(t)) = t
p1(T(n1,n2))   = n1
p2(T(n1,n2))   = n2
```

This pattern applies for any set of names and any number of components in the data structure. Hence, we will call any set of functions that satisfy such requirements with respect to a type T its projection functions and its constructor.

The data invariant will implicitly induce a guard on the constructor T, i.e., CG:Nat,Nat→Bool with CG(n1,n2) = (n1<1000) && (n2<1000), given that p1(T(n1,n2))=n1 and p2(T(n1,n2))=n2. Hence, T(n1,n2) will only be considered to construct a value of type T when CG(n1,n2) holds. The constructor guard is assumed to be managed by the compiler in the manner discussed in Section 3.2.2.

3. an *array data type*, e.g., A, is a parameterised data type with some designated index type P and element type E. For computability reasons, we will limit the index type to be a countable data type, i.e., either finite or representing a set isomorphic to **N**. With every array type we consider a partial indexing operation _[_]:A,P→E together with its implicit guard ig:A,P→Bool. Hence, if V is declared to be an array of type A, i.e. V:A, and p is a variable of type P, i.e., p:P, then V[p] denotes a value of type E whenever ig(V,p) holds. Then V is also said to be *defined* for p. If ig(V,p) does not hold, then V is said to be *undefined* for p.

Type:	Nat	Bool	Data Type Name
&&		and	
||		or	
=		equals	
!=		not equal	
+	addition		disjoint union
-	subtraction		
/	integer division		
*	multiplication		
%	modulus		
>>	bit-wise right shift		

Table 3.1: *Coding symbols used for binary operations on the specified argument types.*

Equality

Every data type is associated with an *equality predicate*, denoted by `EQ`. The equality predicate is an equivalence relation defined on the data type, identifying elements that represent the same value in the data type. All comparisons of the form `e1 = e2`, for any expressions `e1` and `e2` of identical types, are checked using the equality predicate of that type.

In case of simple data types, unless explicitly stated otherwise, the equality predicate will be the common equality on the set of natural numbers, and on the set of two-element truth values.

In case of array types: if `V1:A` and `V2:A` are arrays of the same type, then `EQ(V1,V2)` holds whenever for all `p:P` it is the case that `ig(V1,p)=ig(V2,p)`, and `EQ(V1[p],V2[p])` holds for all relevant `p:P`, where the latter is the equality predicate associated with the element type of the array type `A`.

In case of tuple types, unless explicitly stated otherwise, the equality predicate boils down to the conjunction of the equality predicates associated with each component type in case.

Disjoint Union Data Types

The disjoint union of two types will be defined as a tuple type with special components.

Definition 3.2.1 (Disjoint Union of Two Types)**.** Let `A` and `B` be two data types with `d1:A` and `d2:B` some default values. Then *the disjoint union of* `A` *and* `B` is the data type `A+B = v1 A * v2 B * tag {1,2} | DI` with:

- data invariant for all `p:A+B`:

=	equals
!=	not equal
<	less
<=	less or equal
>	greater
>=	greater or equal

TABLE 3.2: *Coding symbols used for common comparison operators on* `Nat` *with return type* `Bool`.

```
DI(p) = ((tag(p) = 1) && (v2(p) = d2)) ||
        ((tag(p) = 2) && (v1(p) = d1))
```

- equality predicate for arbitrary `p,q:A+B`:

```
EQ(p,q) = ((tag(p) = 1) && (tag(q) = 1) && (v1(p) = v1(q)))
          ||
          ((tag(p) = 2) && (tag(q) = 2) && (v2(p) = v2(q)))
```

- and injections `i1:A→A+B`, `i2:B→A+B` defined by:

```
i1(a) = A+B(a,d2,1) for all a:A
i2(b) = A+B(d1,b,2) for all b:B
```

□

The data invariant limits the possible values of the components to only those that are meaningful for the disjoint union type by making the irrelevant component void (assigning a default value to it). This will induce the implicit constructor guard, e.g., `A+B(a,b,t)` will only be considered, if `b=d2` and `t=1` or `a=d1` and `t=2`.

The equality predicate ensures that the irrelevant component of the triples does not count when testing whether two disjoint union type elements are equal. Note that in practice we always refer to disjoint union type elements in an atomic way or via its injections. We may however access the components via its projections.

We will use the same function names for the projections and injections that come with the disjoint type definition for any given disjoint union type. If several disjoint union types are defined, it will be the type of the variable or expression to which the particular function is applied that will determine the right function call. We may also introduce a shorthand for a disjoint

union type, e.g., `DU=A+B`. If ambiguity arises, e.g., when applying the functions to terminals or constants, or invoking an injection call, we may use the `.` selector operator: `DU.v1(a)` or `DU.i1(0)`.

The definition can easily be expanded for the disjoint union of `n` types. In this case the new data type will have a structure of `n+1` components with all associated projections, injections, constructor, data invariant, and equality predicate. The `tag` component's type then will have `n` elements. Then `vi(p)`, e.g., will refer to the projection associated with the $\texttt{i}^{\text{th}}$ participating data type, and so on.

3.2.4 *Operations*

All operations used in the program code examples are typically binary operations and comparisons. These are collected and presented in Tables 3.1 and 3.2. Note that the equality sign "=" is used as a comparison symbol as well as in function definitions. E.g., a function definition `pred(p) = (p=0)`, where `pred:P→Bool` is a boolean function, means that `pred(p)` is defined to be `true` whenever `p=0`.

CHAPTER 4

Data Dependency Algebras

The concept of data dependency algebra (DDA) was established and first introduced by [Haveraaen, 1990a]. The initial formalism has somewhat changed and evolved in subsequent publications [Čyras and Haveraaen, 1995; Haveraaen, 2000], finally maturing into the notation style best described in [Haveraaen, 2009]. This formalism was taken aboard in our joint works [Burrows and Haveraaen, 2009a,b], and all upcoming works are likely to adhere to this. This chapter revisits the formal definition of DDAs, giving a fresh perspective and understanding of what DDAs are. It improves the DDA presentation given in [Burrows and Haveraaen, 2009a,b], significantly expands the study of space-time DDAs and DDA-projections, and in addition, presents new DDA concepts, related properties and relevant theorems, and defines new DDA examples.

4.1 A Gentle Introduction to DDAs

The concept of data dependency algebra is equivalent to that of a directed multigraph. Miranker and Winkler [1984] suggested that program data dependency graphs can abstract how parts of a computation depend on data supplied by other parts. This is a basis for parallelizing compilers, see e.g. [Wolfe, 1996]. A dependency graph, in classical graph representation, is seen as a set of nodes and a set of edges with specified source and target nodes. In the formalism of DDAs, we have a set of *points* P, a set of *branch indices* B, and two special components: *requests* and *supplies*. As a general

rule, a branch between two points of a data dependency graph stands for a request–supply connection. Intuitively, the flow of the data along the branch can be seen as a data request direction as well as a data supply direction where these directions are opposite to each other. DDAs differentiate between these opposite directions along the same branch by using branch indices from B. These are local at each end of the branch: one identifies the request and one the supply direction.

At each point (nodes of the graph), request directions (arcs, identified now by branch indices) lead to points from which data is required in order to perform a computation at this point. Supply directions (the opposite arcs, identified again by branch indices) lead to points that are supplied with the data computed at the point. This duality of the request–supply connection along the same branch leads to the isomorphic sets of request arcs $r_g \subseteq P \times B$ and supply arcs $s_g \subseteq P \times B$, i.e., the isomorphism $\tilde{r} : r_g \to s_g$ with inverse $\tilde{s} : s_g \to r_g$.

Obviously, we want to keep B as small as possible, but different points may have a varying number of request and supply arcs, motivating the need for *request guard* r_g and *supply guard* s_g. Input points typically do not have any request arcs, since the computation is supposed to start up from these. Likewise, output points typically do not have associated supply arcs, since the computation is supposed to cease on these.

When doing computations on DDAs we need access to component-wise information about these connections. Therefore we split the request isomorphism into a pair of functions, $\tilde{r}(p,b) = \langle r_p(p,b), r_b(p,b)\rangle$ where $r_p : r_g \to P$ and $r_b : r_g \to B$, and likewise the supply isomorphism $\tilde{s}(q,d) = \langle s_p(q,d), s_b(q,d)\rangle$ where $s_p : s_g \to P$ and $s_b : s_g \to B$.

For a pair of point and branch index guarded by r_g, the first component of requests, the *request function* r_p, identifies the point the request direction leads to (i.e. where data is requested from). Likewise, for a pair of point and branch index guarded by s_g, the first component of supplies, the *supply function* s_p, identifies the point where the supply direction leads to (i.e. where data is supplied to).

The second component of both requests and supplies, the *branch-back* functions (r_b and s_b), identify the branch index of the opposite direction along the same branch. The request branch-back r_b gives the branch index at the supplying end of the branch. Similarly, the supply branch-back s_b gives the branch index at the requesting end of the branch.

Following [Burrows and Haveraaen, 2009b], these notations lead to the following formal definition of DDA.

Definition 4.1.1 (DDA)**.** A *data dependency algebra* is given by a 4-tuple $\mathcal{D} = \langle P, B, req, sup\rangle$, where:

1. P is a set of points,
2. B is a set of branch indices,
3. *req* is the data request consisting of $r_g \subseteq P \times B, r_p : r_g \to P, r_b : r_g \to B$
4. sup is the data supply consisting of $s_g \subseteq P \times B, s_p : s_g \to P, s_b : s_g \to B$

such that for all $p : P$ and for all $b : B$ the following axioms hold:

$$s_g(p,b) \Rightarrow r_g(s_p(p,b), s_b(p,b)) \tag{4.1}$$
$$s_g(p,b) \Rightarrow r_p(s_p(p,b), s_b(p,b)) = p \tag{4.2}$$
$$s_g(p,b) \Rightarrow r_b(s_p(p,b), s_b(p,b)) = b \tag{4.3}$$

$$r_g(p,b) \Rightarrow s_g(r_p(p,b), r_b(p,b)) \tag{4.4}$$
$$r_g(p,b) \Rightarrow s_p(r_p(p,b), r_b(p,b)) = p \tag{4.5}$$
$$r_g(p,b) \Rightarrow s_b(r_p(p,b), r_b(p,b)) = b \tag{4.6}$$

□

The axioms here express the duality of requests and supplies as previously discussed. Expanding now the details of [Burrows and Haveraaen, 2009b], the following theorem establishes a direct correlation between the axiomatic definition of DDAs and the presence of isomorphisms $\tilde{r}$ and $\tilde{s}$.

Theorem 4.1.2. Let $\mathcal{D} = \langle P, B, req, sup \rangle$ be a 4-tuple where P and B are sets, *req* consists of three components $r_g \subseteq P \times B$, $r_p : r_g \to P$ and $r_b : r_g \to B$, and sup also consists of three components $s_g \subseteq P \times B$, $s_p : s_g \to P$ and $s_b : s_g \to B$. Let further be $\tilde{r} : r_g \to s_g$ and $\tilde{s} : s_g \to r_g$ two functions defined by: $\tilde{r}(p,b) = \langle r_p(p,b), r_b(p,b) \rangle$ and $\tilde{s}(q,d) = \langle s_p(q,d), s_b(q,d) \rangle$, respectively.
Then $\mathcal{D}$ is a DDA if and only if $\tilde{r}$ is an isomorphism with inverse $\tilde{s}$. □

Proof: We show that the isomorphism $\tilde{r} : r_g \to s_g$ with inverse $\tilde{s} : s_g \to r_g$ is a necessary and sufficient condition for $\mathcal{D}$ to qualify as a DDA.

From $\tilde{r}$ and $\tilde{s}$, we can define the functions $\tilde{r} \circ \tilde{s} : s_g \to s_g$ and $\tilde{s} \circ \tilde{r} : r_g \to r_g$. Then for arbitrary $\langle p, b \rangle \in s_g$, applying the definition of $\tilde{r}$ and $\tilde{s}$, we obtain:

$$\begin{aligned} (\tilde{r} \circ \tilde{s})(p,b) &= \tilde{r}(\tilde{s}(p,b)) \\ &= \tilde{r}(s_p(p,b), s_b(p,b)) \\ &= \langle r_p(s_p(p,b), s_b(p,b)), r_b(s_p(p,b), s_b(p,b)) \rangle \end{aligned} \tag{4.7}$$

Likewise, for an arbitrary $\langle p, b\rangle \in r_g$ we have:

$$\begin{aligned}(\tilde{s} \circ \tilde{r})(p,b) &= \tilde{s}(\tilde{r}(p,b)) \\ &= \tilde{s}(r_p(p,b), r_b(p,b)) \\ &= \langle s_p(r_p(p,b), r_b(p,b)), s_b(r_p(p,b), r_b(p,b))\rangle\end{aligned} \tag{4.8}$$

The function $\tilde{r}$ is an isomorphism with inverse $\tilde{s}$ if and only if $\tilde{r} \circ \tilde{s} = 1_{s_g}$ and $\tilde{s} \circ \tilde{r} = 1_{r_g}$. From the above equations we see that $(\tilde{r} \circ \tilde{s})(p,b) = \langle p, b\rangle$ and $(\tilde{s} \circ \tilde{r})(p,b) = \langle p, b\rangle$ if and only if the DDA axioms 4.1–4.3 and 4.4–4.6 hold, respectively. □

Note that in the axiomatic DDA definition the content of the sets P and B are not fixed, nor are the actual definitions of the request and supply components. Hence, the definition can serve as a programmable interface, where P and B would represent some generic types, and the components of requests and supplies are generic function declarations on these types. All concrete DDA implementations will fix the meaning of P and B and will define the request and supply components in detail. However these should satisfy all DDA-axioms, and – by abuse of language – we will call all such implementations data dependency algebras, or DDAs. This obviously gives rise to infinitely many implementations of the same interface.

We have seen that the DDA-axioms express the duality of requests and supplies, which follows directly from the isomorphisms between r_g and s_g. This and the symmetric requirements of the definition lead us to the following theorem, expanding further the details of [Burrows and Haveraaen, 2009b]:

Theorem 4.1.3. If $\mathcal{D} = \langle P, B, req, sup\rangle$ is a DDA, then $\mathcal{D} = \langle P, B, sup, req\rangle$ is also a DDA. □

Proof: Since $\mathcal{D} = \langle P, B, req, sup\rangle$ is a DDA, by Def. 4.1.1 we have all DDA-axioms satisfied. Reordering the axioms by swapping axiom 4.1 with 4.4, axiom 4.2 with 4.5 and axiom 4.3 with 4.6, we attain the right order for the axioms which make the 4-tuple $\mathcal{D} = \langle P, B, sup, req\rangle$ a DDA. □

Intuitively, the theorem states that we can obtain a new DDA with a reversed dataflow by simply swapping *req* and *sup*.

4.1.1 DDAs vs. Classical Graph Representations

Before presenting some of the basic properties of DDAs, we first motivate their use for describing data dependency graphs. For the sake of the

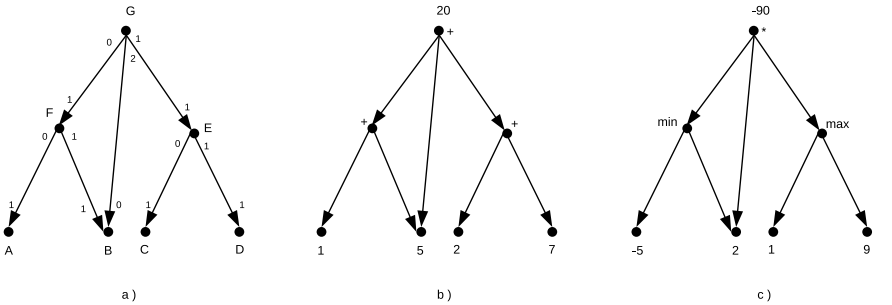

FIGURE 4.1: *a) A data dependency graph with DDA-specific labelling. b) Identical computations are performed at each point of the DDA, for a given input. c) Different computations are performed on the points, for a different input set.*

example, a simple expression tree is defined as a DDA and the related computations in terms of this dependency. The example sketches the essence of DDA-based programming and hints at its potential for parallel computing.

Example 4.1.4 (Expression-tree DDA)**.** Consider the data dependency graph given in Fig. 4.1.a. The pattern can be described in the DDA formalism as follows:

- The set of points $P = \{A, B, C, D, E, F, G\}$
- The set of branch indices $B = \{0, 1, 2\}$
- The requests are defined by:
  - $r_g = \{(F,0), (F,1), (E,0), (E,1), (G,0), (G,1), (G,2)\}$
  - the request-function is given by:

    $r_p(F,0) = A \quad r_p(F,1) = B \quad r_p(E,0) = C \quad r_p(E,1) = D$
    $r_p(G,0) = F \quad r_p(G,1) = E \quad r_p(G,2) = B$
  - and the request branch-back by:

    $r_b(F,0) = 1 \quad r_b(F,1) = 1 \quad r_b(E,0) = 1 \quad r_b(E,1) = 1$
    $r_b(G,0) = 1 \quad r_b(G,1) = 1 \quad r_b(G,2) = 0$
- And the supplies are defined by:
  - $s_g = \{(A,1), (B,1), (B,0), (C,1), (D,1), (F,1), (E,1)\}$

– the supply-function is given by:

$$s_p(A,1) = F \quad s_p(B,1) = F \quad s_p(B,0) = G \quad s_p(C,1) = E$$
$$s_p(D,1) = E \quad s_p(F,1) = G \quad s_p(E,1) = G$$

– and the supply branch-back by:

$$s_b(A,1) = 0 \quad s_b(B,1) = 1 \quad s_b(B,0) = 2 \quad s_b(C,1) = 0$$
$$s_b(D,1) = 1 \quad s_b(F,1) = 0 \quad s_b(E,1) = 1$$

□

The dependency graph of Fig. 4.1.a can be also defined as a simple directed graph, $\mathcal{G} = \langle \mathcal{N},\ \mathcal{E},\ s : \mathcal{E} \to \mathcal{N},\ t : \mathcal{E} \to \mathcal{N} \rangle$ with:

- the set of nodes $\mathcal{N} = \{A, B, C, D, E, F, G\}$
- the set of edges defined as $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$, i.e.,

$$\mathcal{E} = \{\langle G,F\rangle, \langle G,B\rangle, \langle G,E\rangle, \langle F,A\rangle, \langle F,B\rangle, \langle E,C\rangle, \langle E,D\rangle\}$$

- $s(a,b) = a$ for all $\langle a,b\rangle \in \mathcal{E}$ and
- $t(a,b) = b$ for all $\langle a,b\rangle \in \mathcal{E}$

Though both the DDA- and graph-based representations faithfully describe the dependency itself, the latter proves to be inadequate to express certain computations. For instance, if we want to specify a computation to be performed at a given node in terms of its dependencies, the classical graph-representation lacks expressive power, whereas the DDA-based representation can fully be exploited for this purpose, as shown next.

4.1.2 The Expressive Power of DDAs

A DDA may serve as a leading pattern for different computations. In the default view of DDAs, the computations are *point-valued*, i.e., only one value is computed at a point and this is passed on along all its supply directions. But DDAs can be also endowed with *branch-valued* computations. Then, instead of DDA points, the computations are defined on supply directions, i.e., each pair $\langle p,b\rangle$ guarded by s_g is assigned a computation.

Consider now the expression tree DDA, with point-valued computations, to evaluate different expressions. E.g., let's consider A, B, C, D as input points and F the output, so that the data flows in the opposite direction of the arrows. In Fig. 4.1.b each point of the DDA is associated with a summing operation, which adds together the data values coming in along

the request branches. Consider the array of integers w indexed by the elements of P. Provided that initial values have been assigned to $w[A]$, $w[B]$, $w[C]$ and $w[D]$, the computation to be performed on the rest of the points can be defined by:

$$w(p) = \begin{cases} w[r_p(p,0)] + w[r_p(p,1)] & \text{if} \quad p = F \quad \text{or} \quad p = E \\ w[r_p(p,0)] + w[r_p(p,1)] + w[r_p(p,2)] & \text{if} \quad p = G \end{cases}$$

This ultimately yields the desired final result at the (output) point G in $w[G]$.

Fig. 4.1.c illustrates yet another computation being performed on the points of the DDA, resulting in the evaluation of another expression, for a different input set. We can define this for another array of integers v indexed also by P:

$$v(p) \quad = \quad \begin{cases} min(v[r_p(p,0)], v[r_p(p,1)]) & \text{if} \quad p = F \\ max(v[r_p(p,0)], v[r_p(p,1)]) & \text{if} \quad p = E \\ v[r_p(p,0)] * v[r_p(p,1)] * v[r_p(p,2)] & \text{if} \quad p = G \end{cases} \tag{4.9}$$

Similarly, the desired final result will reside at point G in $v[G]$.

Note how the dependency pattern described by r_p becomes an explicit entity in both computations. Hence, computation and dependency are now separated in a modular way [Čyras and Haveraaen, 1995], so that both are programmable independently from each other. E.g., for the same set of points, branch indices and request guard, one could define a new expression tree (i.e. new DDA) by giving a different interpretation of the supply guard, the request and supply functions, and ultimately of the branch-back functions. The above computations still make sense, but yield different results typical for the new expression tree DDA. (E.g. Fig. 4.2 shows a slightly modified DDA with the associated point-valued computations defined exactly as in (4.9).)

We see that the modular separation of the computation from its dependency is facilitated explicitly by the request component of the DDA. To understand the role of the supply component, note that data dependency graphs defined as DDAs together with the associated computations and input values, have a double reading. On the one hand, utilizing the request directions, they specify in a concise way the computations that are to be performed at the points in order to achieve the final result. This corresponds to a top-down reading. On the other hand, the supply directions facilitate a bottom-up reading. Starting from the input values, it drives the computation along the dependencies. The latter reading yields various dependency-driven computational mechanisms, which depending on

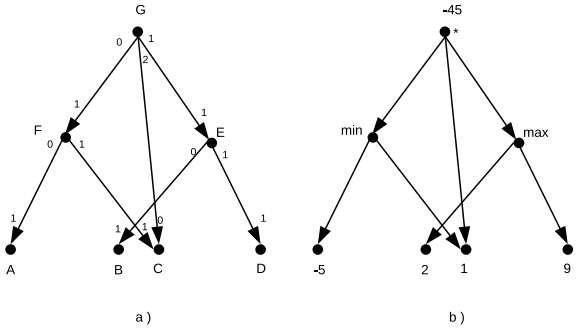

Figure 4.2: *a) A new DDA, defined by slightly altering the request and supply components of the original DDA. b) Same DDA with associated point-wise computations identical to those given in Fig. 4.1.c and formally defined in 4.9.*

the target machine and the inherent properties of the DDA in question, can be turned into sequential, shared or distributed memory parallel code, or be adjusted to some specialized multi-core or GPU code. A DDA-enabled parallelizing compiler therefore can harness directly the implicit driving force of dependencies and generate parallel code to virtually any parallel system which has a well defined space-time communication structure. While Chapter 5 presents these DDA-based computational mechanisms in more detail, Sections 4.3 and 4.4 of this chapter will focus on specific DDA-abstractions that make all these possible.

4.1.3 Structuring DDAs

Haveraaen [2009] showed that from every directed multigraph a DDA can be derived, and vice versa, from every DDA a directed multigraph can be defined. Moreover, if a graph $\mathcal{G}$ is derived from a DDA which itself is derived from another graph $\mathcal{G}'$, then $\mathcal{G} \simeq \mathcal{G}'$. These properties establish a very strong connection between DDAs and directed multigraphs. The class of all DDAs, in this sense, happens to be too *rich* for programming purposes. The following definitions allow us to delimit ourselves to different subclasses of DDAs relevant in an appropriate computational context.

Definition 4.1.5 (Countable DDA)**.** A *countable data dependency algebra* is a DDA $\mathcal{D} = \langle P, B, req, sup \rangle$ where P is countable and B is finite. □

Definition 4.1.6 (Cyclic and Acyclic DDAs)**.** A DDA has *cycles* or is *cyclic*, if repeated application of requests from some starting point $p : P$ takes us back to p. If a DDA has no cycles it is *acyclic*. □

Definition 4.1.7 (Well-founded DDA)**.** A DDA is *well-founded* if for any point $p : P$ all requests paths from p, obtained via repeated applications of requests, are finite.

For instance, DDAs that describe the static connectivity of a parallel hardware are typically cyclic, whereas the space-time unfoldings and computation related DDAs are as a general rule acyclic.

Two new DDA-concepts are introduced next. Both are defined wrt. the class of all DDAs. The *sub-DDA* concept will give rise to the definition of one of the DDA *combinators* presented in Chapter 7. The second establishes the notion of *isomorphic DDAs* which identifies DDAs that define isomorphic graphs.

Intuitively, two DDAs are in the sub-DDA relation, when the two derived directed multigraphs are in the sub-graph relation. The following definition captures this notion solely in terms of DDA components.

Definition 4.1.8 (Sub-DDA)**.** Given DDAs $\mathcal{D} = \langle P, B, req, sup \rangle$ and $\mathcal{D}' = \langle P', B', req', sup' \rangle$, D' is a *sub-DDA* of D, i.e., $D' \subseteq D$, if and only if the following conditions hold:

1. $P' \subseteq P$
2. $B' \subseteq B$
3. $r'_g \subseteq r_g$
4. $\tilde{r}' = \tilde{r}\big|_{r'_g}$ □

Theorem 4.1.9. Given DDAs $\mathcal{D} = \langle P, B, req, sup \rangle$ and $\mathcal{D}' = \langle P', B', req', sup' \rangle$ such that $D' \subseteq D$, then the following also hold:

1. $s'_g \subseteq s_g$
2. $\tilde{s}' = \tilde{s}\big|_{s'_g}$

Proof: 1. By $D' \subseteq D$ we have $\tilde{r}' = \tilde{r}\big|_{r'_g}$ and $r'_g \subseteq r_g$. Further, note that $\tilde{r}' : r'_g \to s'_g$ and $\tilde{r} : r_g \to s_g$ are isomorphisms. Then $s'_g = \tilde{r}'(r'_g) = \tilde{r}\big|_{r'_g}(r'_g) \subseteq \tilde{r}(r_g) = s_g$.

2. In the above equality we see that $\tilde{r}\big|_{r'_g}(r'_g) = s'_g$, and since $\tilde{r}$ is an isomorphism with inverse $\tilde{s}$, then the inverse of $\tilde{r}\big|_{r'_g}$ is $\tilde{s}\big|_{s'_g}$. On the other hand the inverse of $\tilde{r}'$ is $\tilde{s}'$. By the uniqueness of the inverse (see Theorem 3.1.3) and the equality $\tilde{r}' = \tilde{r}\big|_{r'_g}$, we conclude that $\tilde{s}' = \tilde{s}\big|_{s'_g}$ □

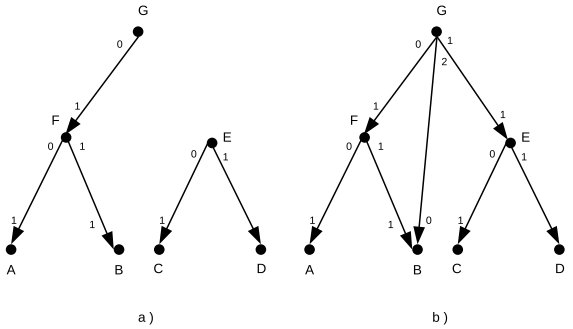

Figure 4.3: *The DDA shown in a) is a sub-DDA of the DDA shown in b).*

From a computational point of view, the sub-DDA is most likely to serve as the dependency pattern of a different computation, i.e., other than the one allowed by the main DDA. However, the sub-DDA concept, when applied as a DDA-combinator, serves as a refactoring tool to support code reusability (see Chapter 7).

A very common scenario, when defining a DDA for a given dependency pattern, is that there are several options out there for picking the set of points (i.e. their type) and the set of branch indices. These may ultimately lead to two different DDA-representations of the same dependency pattern. E.g., in order to define the expression-tree dependency as a DDA (Example 4.1.4), one could have chosen the set $\{-2,-1,0,1,2,3,4\}$ to be the set of points P, instead of $P = \{A,B,C,D,E,F,G\}$, and the set $\{a,b,c,d,e,f,g\}$ as the set of branch indices, one for each branch, instead of $B = \{0,1,2\}$ (see Fig. 4.4.a and b). These choices then lead to different request and supply definitions, resulting in a *different* expression-tree *DDA*, but not a different expression-tree dependency. The difference is only as per *implementation*. The semantic behaviour of both DDAs are the same: both describe the same dependency pattern, or in other words, the graphs derived from the two DDAs will be isomorphic.

Note that while isomorphic graphs require isomorphic sets of nodes and isomorphic sets of edges, isomorphic DDAs may have non-isomorphic sets of branch indices. E.g., the branch indices set $\{0,1,2\}$ of the DDA in Fig. 4.4.a is not isomorphic with the branch indices set $\{a,b,c,d,e,f,g\}$ of the DDA in Fig. 4.4.b.

Definition 4.1.10 (Isomorphic DDAs)**.** Given DDAs $\mathcal{D} = \langle P, B, req, sup\rangle$ and $\mathcal{D}' = \langle P', B', req', sup'\rangle$. Then D is *isomorphic* to D', denoted by $D \simeq D'$, if and only if $P \simeq P'$ and the following holds:

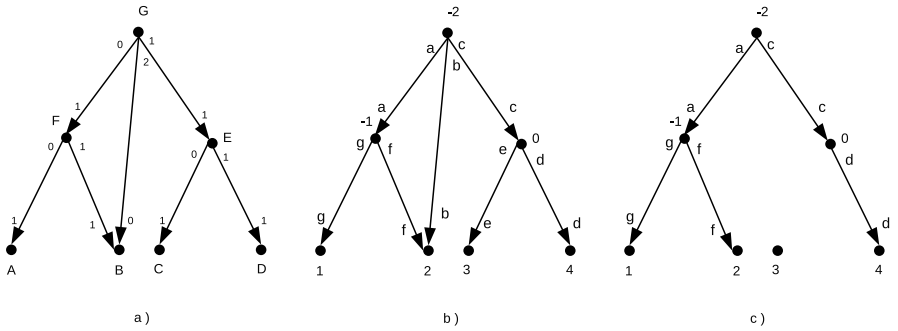

FIGURE 4.4: *The semantic behaviour of the DDA shown in a) is identical to the one shown in b): they both describe the same dependency pattern, i.e., the graphs derived from them are isomorphic. The DDA shown in c) fails to be isomorphic with the rest.*

There exist isomorphisms, each respecting the isomorphism between the set of points P and P', $\iota_{r_g} : r_g \rightarrow r'_g$ with inverse $\iota^{-1}_{r_g} : r'_g \rightarrow r_g$ and $\iota_{s_g} : s_g \rightarrow s'_g$ with inverse $\iota^{-1}_{s_g} : s'_g \rightarrow s_g$ such that:

$$\tilde{r} \quad = \quad \iota^{-1}_{s_g} \circ \tilde{r}' \circ \iota_{r_g} \tag{4.10}$$

the latter constituting the *DDA isomorphism condition*. □

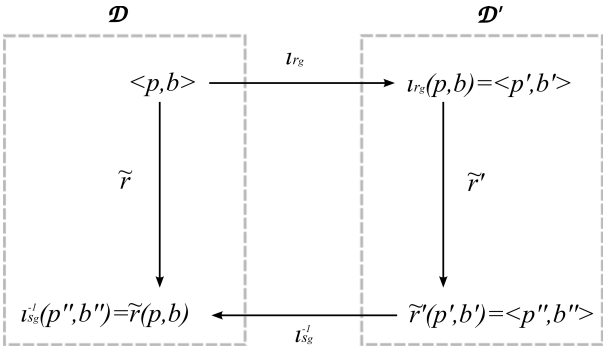

FIGURE 4.5: *Illustrating the DDA isomorphism condition for DDAs $\mathcal{D}$ and $\mathcal{D}'$.*

We have seen that the essence of DDAs are captured in the isomorphism $\tilde{r} : r_g \rightarrow s_g$ with inverse $\tilde{s} : s_g \rightarrow r_g$. It is not a surprise therefore that isomorphic DDAs should have isomorphic request and consequently isomorphic supply guards, such that their own request component (see Fig. 4.5), e.g., $\tilde{r}$, will become expressible in terms of the other DDAs request component, e.g. $\tilde{r}'$. This in turn does not require isomorphism between the sets of branch indices. On the other hand, the isomorphism between the point sets should be respected, e.g., if we denote this by $\iota_p : P \rightarrow P'$, then for all $\langle p, b\rangle \in r_g$

$$\iota_{r_g}(p, b) = \langle q, d\rangle \Leftrightarrow \iota_p(p) = q$$

and for all $\langle p, b\rangle \in s_g$

$$\iota_{s_g}(p, b) = \langle q, d\rangle \Leftrightarrow \iota_p(p) = q$$

The property then automatically holds for the inverses $\iota_{r_g}^{-1}$ and $\iota_{s_g}^{-1}$. The requirement of isomorphism between P and P' spawns from the fact that the graphs derived from the DDAs otherwise would not have the chance at all to be isomorphic, e.g., see Fig. 4.6.a. If this isomorphism is not respected by the mapping of request/supply directions of $\mathcal{D}$ to the request/supply directions of $\mathcal{D}'$, then it may be the case that the isomorphism condition is satisfied, yet the two DDAs fail to define the same dependency, e.g., see Fig. 4.6.b. Here the isomorphisms $\iota_{r_g}(F, 0) = \langle A, 1\rangle$, $\iota_{r_g}(F, 1) = \langle B, 1\rangle$ and $\iota_{s_g}^{-1}(F, 0) = \langle A, 1\rangle$, $\iota_{s_g}^{-1}(F, 1) = \langle B, 1\rangle$ will make the isomorphism condition hold, but they do not respect the isomorphism between the point sets, since F cannot be mapped to both A and B.

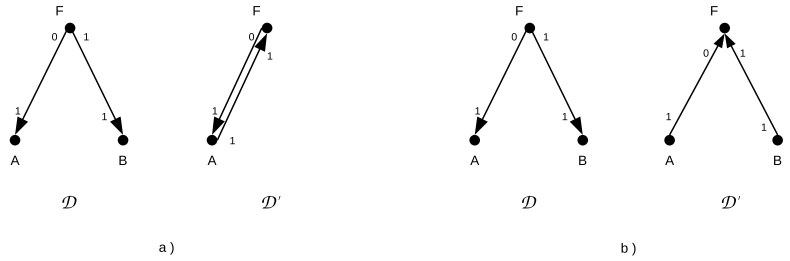

Figure 4.6: *Example of non-isomorphic DDAs: a) point sets are not isomorphic; b) there exist isomorphisms ι_{r_g} and ι_{s_g} that would make the DDA isomorphism condition hold, but they do not respect the isomorphism between the DDAs' point sets.*

We show now that the isomorphism condition may take up different forms, as a direct result of the duality of requests and supplies as well as the isomorphisms between the guards of two isomorphic DDAs.

Theorem 4.1.11. The DDA isomorphism condition is equivalent to any of the following equations:

$$\tilde{r}' = \iota_{s_g} \circ \tilde{r} \circ \iota_{r_g}^{-1} \tag{4.11}$$

$$\tilde{s} = \iota_{r_g}^{-1} \circ \tilde{s}' \circ \iota_{s_g} \tag{4.12}$$

$$\tilde{s}' = \iota_{r_g} \circ \tilde{s} \circ \iota_{s_g}^{-1} \tag{4.13}$$

□

Proof: Equation (4.11):

$$\begin{aligned} & \tilde{r} = \iota_{s_g}^{-1} \circ \tilde{r}' \circ \iota_{r_g} && (\text{by } (4.10)) \\ \Leftrightarrow \quad & \iota_{s_g} \circ \tilde{r} = \tilde{r}' \circ \iota_{r_g} && (\text{by } \iota_{s_g} \circ \,|) \\ \Leftrightarrow \quad & \iota_{s_g} \circ \tilde{r} \circ \iota_{r_g}^{-1} = \tilde{r}' && (\text{by } | \circ \iota_{r_g}^{-1}) \end{aligned}$$

Equation (4.12):

$$\begin{aligned} & \tilde{r} \circ \tilde{s} = \iota_{s_g}^{-1} \circ \tilde{r}' \circ \iota_{r_g} \circ \tilde{s} && (\text{by } (4.10) \text{ and } | \circ \tilde{s}) \\ \Leftrightarrow \quad & \mathbf{1}_{s_g} = \iota_{s_g}^{-1} \circ \tilde{r}' \circ \iota_{r_g} \circ \tilde{s} \\ \Leftrightarrow \quad & \iota_{s_g} = \tilde{r}' \circ \iota_{r_g} \circ \tilde{s} && (\text{by } \iota_{s_g} \circ \,|) \\ \Leftrightarrow \quad & \tilde{s}' \circ \iota_{s_g} = \iota_{r_g} \circ \tilde{s} && (\text{by } \tilde{s}' \circ \,|) \qquad (4.14) \\ \Leftrightarrow \quad & \iota_{r_g}^{-1} \circ \tilde{s}' \circ \iota_{s_g} = \tilde{s} && (\text{by } \iota_{r_g}^{-1} \circ \,|) \end{aligned}$$

Equation (4.13):

$$\begin{aligned} & \tilde{s}' \circ \iota_{s_g} = \iota_{r_g} \circ \tilde{s} && (\text{by } 4.14) && (4.15) \\ \Leftrightarrow \quad & \tilde{s}' = \iota_{r_g} \circ \tilde{s} \circ \iota_{s_g}^{-1} && (\text{by } | \circ \iota_{s_g}^{-1}) && (4.16) \end{aligned}$$

□

Proposition 4.1.12. The isomorphism condition of Definition 4.1.10 is interchangeable with any of the equations listed in Theorem 4.1.11. □

Proof: Straightforward. □

The relevance of isomorphic DDAs shows up in cases when one is faced with two different DDA-implementations of possibly the same data dependency. The conditions of isomorphic DDAs discussed here provide the means to argue, or perhaps to even prove that the two implementations have the same semantic behaviour, i.e., they indeed define the same dependency.

Definition 4.1.13 (Sub-isomorphic DDA)**.** Given DDAs $\mathcal{D} = \langle P, B, req, sup \rangle$ and $\mathcal{D}' = \langle P', B', req', sup' \rangle$, D' is *sub-isomorphic* to $\mathcal{D}$, i.e., $\mathcal{D}' \subsetneqq \mathcal{D}$, if and only if there exists DDA $\mathcal{D}'' = \langle P'', B'', req'', sup'' \rangle$ such that:

1. $\mathcal{D}'' \subseteq \mathcal{D}$
2. $\mathcal{D}'' \simeq \mathcal{D}'$ □

With the above definition one can identify sub-DDAs up to isomorphism. While the sub-DDA relation requires that the graphs derived from the two DDAs should be in the sub-graph relation, a graph derived from a sub-isomorphic DDA is required only to be isomorphic to a sub-graph.

4.2 DDAs for Computations

As a general rule, defining the data dependency pattern of a computation as a DDA works best when the pattern shows a certain amount of regularity that can be captured in an algorithmic or functional way. The expression-tree DDA example of Section 4.1.1 was a small dependency graph, so the lack of regularity did not really matter, since it can be described very neatly, in just a few lines, by listing every guarded combination of the arguments. However, in practice data dependency graphs are much larger. In this section, we present more elaborate data dependency patterns defined as DDAs, and show some techniques that promote DDA code reusability.

Remark 4.1. All concrete DDA examples presented here and in consequent chapters:

1. are countable DDAs, defined in conformity with the coding style introduced in Section 3.2;
2. in all DDA illustrations, the arrows go in the direction of the requests. Data flows in the opposite direction of the arrows, i.e., along the supply directions.
3. DDA points and branch indices will often be referred to as *DDA point sort* or *type* and *branch index sort* or *type*, respectively.

□

We first illustrate the above-mentioned coding style on a simple DDA which defines a *forking* pattern, see Fig.4.7, a dependency pattern common to vector computations.

Example 4.2.1. A *forking DDA* of size $\mathtt{n} \in \mathbf{N}$, $\mathtt{DFK_n}$, is defined by:

1. DDA points: $\mathtt{FK_n}$ `= row Nat * col Nat |` $\mathtt{DI_n}$ where:

```
DIn(p) = ((row(p)=1) && (col(p)<n)) ||
         ((row(p)=0) && (col(p)<2n))
```

2. branch indices: $\mathtt{B} = \{\mathtt{0}, \mathtt{1}\}$

3. request components `(rg,rp,rb)` where:

```
rg(p,b) = (row(p)=1)
rp(p,b) = if (b=0) FKn(0,col(p))
          else FKn(0,col(p)+n)
rb(p,b) = 0
```

4. supply components `(sg,sp,sb)` where:

```
sg(p,b) = (row(p)=0) && (b=0)
sp(p,b) = if (col(p)<n) FKn(1,col(p))
          else FKn(1,col(p)-n)
sb(p,b) = if (col(p)<n) 0
          else 1
```

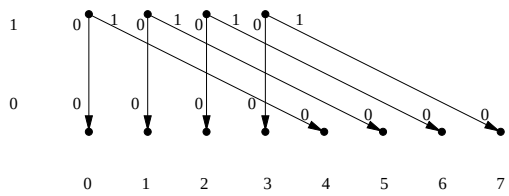

FIGURE 4.7: *Forking DDA of size* 4. *The* `col` *and* `row` *projections of* $\mathtt{FK_4}$ *points are illustrated as coordinates.*

The data invariant selects a sub-sort or sub-type of the DDA point sort which is meaningful for the DDA. For all branches across, branch index `1` identifies the request direction and branch index `0` the supply direction. The vertical branches are labelled for both directions with branch index `0`. Request directions exist only for points of the top row, whereas supply directions exist only for the points of the bottom row. These properties are captured by the corresponding guards. The rest of both components are defined as simple functional expressions which return the desired values for guarded pair of inputs of a point and a branch index.

Next, the butterfly dependency is defined as a DDA. The butterfly, see Fig. 4.8, appears in many divide-and-conquer algorithms, one of the most common being the Fast Fourier Transform (FFT) [Bergland, 1969].

Example 4.2.2. A *butterfly DDA of height* $\mathtt{h} \in \mathbf{N}$, $\mathtt{DBF_h}$, is defined by:

1. DDA points: $\mathtt{BF_h}$ `= row Nat * col Nat |` $\mathtt{DI_h}$ where:

   $\mathtt{DI_h}$`(p) = (row(p)<=h) && (col(p)<`$2^{\mathtt{h}}$`)`

2. branch indices: $\mathtt{B} = \{\mathtt{0}, \mathtt{1}\}$

3. request components `(rg,rp,rb)` where:

   ```
   rg(p,b) = (row(p)<h)
   rp(p,b) = if (b=0) BFh(row(p)+1,col(p))
   ```

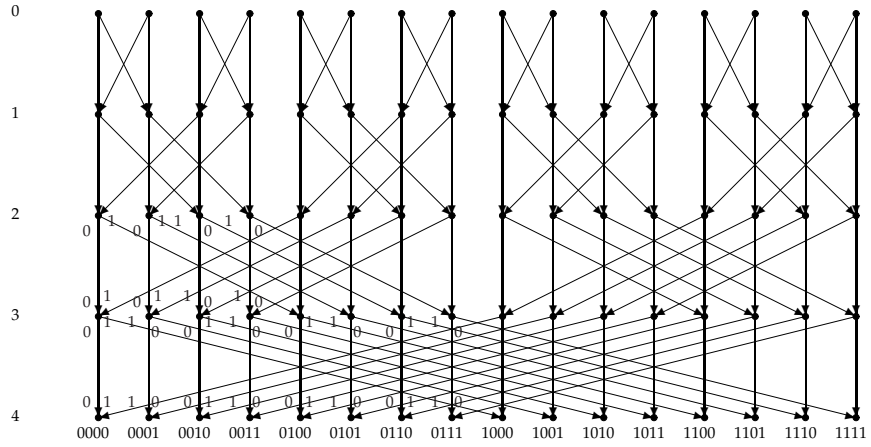

Figure 4.8: *Butterfly DDA of height* 4. *The* `col` *projections of* $\mathtt{BF_4}$ *points are presented as binary numbers.*

```
                else BF_h(row(p)+1,flip(row(p),col(p)))
        rb(p,b) = b
```

4. supply components `(sg,sp,sb)` where:

```
        sg(p,b) = (0<row(p))
        sp(p,b) = if (b=0) BF_h(row(p)-1,col(p))
                  else BF_h(row(p)-1,flip(row(p)-1,col(p)))
        sb(p,b) = b
```

where `flip(i,n)` flips the $\mathtt{i}^{\mathrm{th}}$ bit (where bit 0 is the rightmost, least significant bit of `n`) in the binary representation of the integer `n`. □

In Fig. 4.8 the layout of the butterfly DDA is given as it is laid out in a grid by the use of `row` and `col` projections of the DDA points. Inputs are assumed to reside on the points of the bottom row, and an eventual computation proceeds upward. The flow of the data throughout the whole computation is defined explicitly by the supply functions. Consecutive points have their `row` projections decreased by 1 at every step. When data is passed upright, along branch index 0, the `col` projections of the consecutive points are preserved. And when data is passed across, along branch index 1, the `col` projections of consecutive points differ in their binary representation: the bit pointed to by the `row` projection flips. The definition of the supply function is based on this observation. The definition of the request function is dual.

When we have to deal with a butterfly pattern where the dataflow is reversed, as it is often the case in FFT-related computations, a corresponding reversed butterfly can be defined again from the old, see Fig. 4.9. We preserve both the branch indices and the DDA point sort, without changing the data invariant, and only swap the request and supply components. By Theorem 4.1.3 we know that the result will be a DDA, leading to a direct example of code reusability.

Example 4.2.3. A *reversed butterfly DDA of height* $\mathtt{h} \in \mathbf{N}$, $\mathtt{DRBF_h}$ is defined by:

1. DDA points: $\mathtt{BF_h}$
2. branch indices: $\mathtt{B} = \{\mathtt{0}, \mathtt{1}\}$
3. request components `(rg',rp',rb')` where:

```
        rg'(p,b) = sg(p,b)
        rp'(p,b) = sp(p,b)
        rb'(p,b) = sb(p,b)
```

4. supply component `(sg',sp',sb')` where:

```
sg'(p,b) = rg(p,b)
sp'(p,b) = rp(p,b)
sb'(p,b) = rb(p,b)
```

where `sg`, `sp`, `sb` and `rg`, `rp`, `rb` denote, respectively, the supply and request components of the butterfly DDA $\texttt{DBF}_\texttt{h}$=<$\texttt{BF}_\texttt{h}$,B,(rg,rp,rb),(sg,sp,sb)>. □

By manipulating the data invariant of the DDA point sort, we may restrict further the range of points we are interested in, resulting in a new DDA, corresponding to a new dependency. E.g., a binary tree DDA can be defined from the reversed butterfly DDA, see Fig. 4.10.

Example 4.2.4. The *binary tree DDA of height* $\texttt{h} \in \mathbf{N}$, $\texttt{DBT}_\texttt{h}$, is defined by:

1. DDA points: $\texttt{BT}_\texttt{h}$ = $\texttt{BF}_\texttt{h}$ | DIT where:

   $$\texttt{DIT(p)} = \texttt{(col(p)\%2}^{\texttt{row(p)}}\texttt{=0)}$$

2. branch indices: $\texttt{B} = \{\texttt{0}, \texttt{1}\}$

3. request components `(rgt,rpt,rbt)` where:

```
rgt(p,b) = rg'(p,b)
rpt(p,b) = rp'(p,b)
rbt(p,b) = rb'(p,b)
```

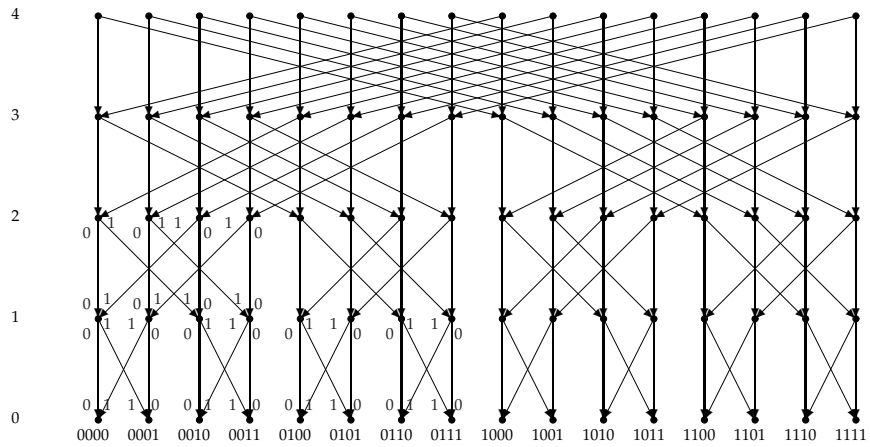

Figure 4.9: *Reversed butterfly DDA of height* 4.

4. supply components `(sgt,spt,sbt)` where:

```
sgt(p,b) = sg'(p,b)
spt(p,b) = sp'(p,b)
sbt(p,b) = sb'(p,b)
```

where `rg'`, `rp'`, `rb'` and `sg'`, `sp'`, `sb'` denote, respectively, the request and supply components of the reversed butterfly DDA. □

Note that the new data invariant `DIT` is preserved by `rpt` and `spt`, even though these are defined by the reversed butterly's corresponding components. It is easy to see also that the binary tree DDA is a sub-DDA of the reversed butterfly DDA, i.e., $\mathtt{DBT_h} \subseteq \mathtt{DRBF_h}$. This construction also underlines the fact that a high-level manipulation of the data invariant also endows code reusability. Similarly, restriction of guards and consequent restrictions of the other DDA-components support code reusability. This can be achieved by the application of the sub-DDA-combinator, which is presented in details in Chapter 7.

4.3 Space-Time DDAs

Following Miranker and Winkler [1984] we may control the parallel execution of a computation by embedding the computation into the space-time connectivity of a parallel machine. DDAs, by their very nature, can abstract

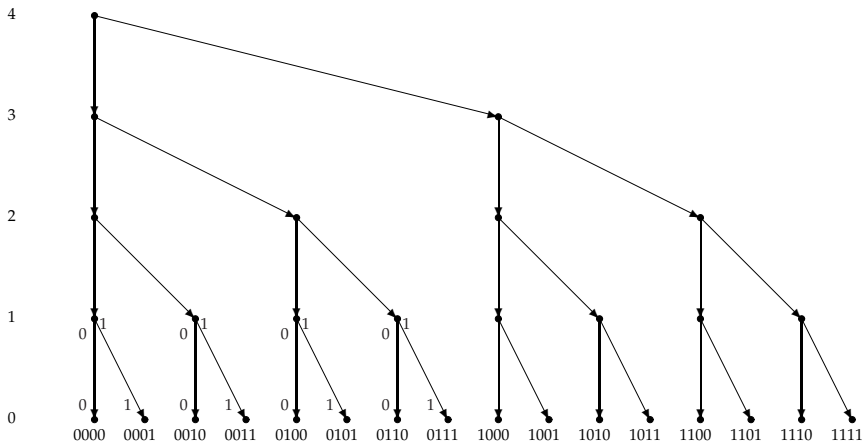

Figure 4.10: *Binary tree DDA of height* 4.

over both the *static* and the *dynamic connectivity* of a (parallel) machine architecture. The former can be defined by ordinary DDAs, referred to as *hardware DDAs*, whereas the latter can be defined by special DDAs, referred to as *space-time DDAs* (STA). In a hardware DDA, the points identify the processors and the branches the available communication channels between the processors. The granularity of a *processor* can range from a single logical gate (e.g. on FPGAs) to an arbitrarily complex function, or from a (parallel) thread or MPI process to a general CPU. The dynamic connectivity of the architecture, i.e., its space-time, is usually obtained by projecting its hardware DDA over time. Then a computation on a processor takes place at the points of the so obtained space-time DDA, while communications between processors along the communication channels take a time increment, basically the pairing of the hardware DDA with time-stepping. This attributes a special property to space-time DDAs, which is made concrete next.

In order to be able to formalise this property, we first introduce some sorts. Let `Space` be a sort, representing the processors of a parallel machine. And let `Time` be a sort representing the natural numbers, with a strict total order <, a starting point `zero`, and an increment operator `next:Time→Time`, such that `t<next(t)` for all `t:Time`.

Definition 4.3.1 (Space-time DDA)**.** A *space-time DDA* (STA) is given by a DDA `D=<P,B,req,sup>` with implicit projections `space:P→Space` and `time:P→Time`, and constructor `P:Space,Time→P`, such that, on top of the general consistency requirements imposed on implicit projections and constructor (see Section 3.2.3), the following property also holds for all pair `(p,b)` guarded by `rg`:

$$\texttt{time(rp(p,b))<time(p)}$$

constituting the *additional consistency requirement of space-time projections*. □

Example 4.3.2 (Expression tree STA)**.** Consider the expression tree DDA of Fig. 4.1.a. For instance, we could define the following space and time projections, with `next` being the successor function on natural numbers, and `Space={0, 1, 2, 3}`:

```
time(A) = time(B) = time(C) = time(D) = 0
time(F) = time(E) = 1
time(G) = 2


space(A) = space(F) = space(G) = 0
space(B) = 1
```

```
space(E) = space(C) = 2
space(D) = 3
```

It is easy to verify that these projections comply with all consistency requirements of space-time projections. □

Definition 4.3.3 (Finite span STA)**.** A *finite span space-time DDA* is a space-time DDA where the type `Space` is finite, and there is a natural number `tspan` such that at most `tspan` applications of `next` takes the value `time(rp(p,b))` to the value `time(p)`, for all pair `(p,b)` guarded by `rg`, i.e.,

$$\texttt{tspan} = \max_{\texttt{p,b}}(\texttt{time(p)-time(rp(p,b))}$$

□

Definition 4.3.4 (Single span STA)**.** A *single span STA* is a finite span STA with `tspan=1`. □

In case of the expression tree STA with the projections defined as above, `tspan=2`, since the maximum number of time-steps needed to get the value at `B` to `G` is `2`, i.e., `time(G)-time(rp(G,2)=2`.

In certain cases STAs with `tspan>1` may be difficult or impossible to embed to a given parallel machine's space-time if, e.g., the machine happens to have a restricted communication pattern. We can however transform any computation with a large tspan STA to an equivalent computation with a single span STA. This can be achieved by adding extra points along these arcs, one corresponding to each "missing" time-step, and in the context of the whole computation, appointing identity-like functions to each of the new points.

Example 4.3.5 (Eliminating Large Tspans)**.** Let `D=<P,B,req,sup>` be an STA with `tspan>1` where `req=(rg,rp,rb)` and `sup=(sg,sp,sb)`. We transform `D` into a single span STA `D'=<P',B,req',sup'>` where `req'=(rg',rp',rb')` and `sup'=(sg',sp',sb')` as follows.

Initially, `D'` is identical to `D`, then the following steps are repeated until there is no more `p:P'` with `time'(p)-time'(rp'(p,b))>1`:

- Pick a point `p:P'` such that `time'(p)-time'(rp'(p,b))=k`, for some relevant `b:B`, where `k>1`.
- Define the DDA `D"=<P",B,req",sup">` as follows:
  1. DDA points: `P"=P'+{1,2,...,k-1}` is a disjoint union data type
  2. branch indices: `B` (does not change)

3. request component `req"=(rg",rp",rb")` where:

```
rg"(q,d) = ((tag(q)=1) && rg'(v1(q),d)) ||
           ((tag(q)=2) && (d=b))
rp"(q,d) =
    if (tag(q)=2)
       if (v2(q)<k-1) P"(i2(v2(q)+1))
       else P"(i1(rp'(p,b)))
    else if (v1(q)=p) && (d=b) P"(i2(1))
         else P"(i1(rp'(v1(q),d)))

rb"(q,d) = if (tag(q)=2) rb'(p,b) else rb'(v1(q),d)
```

4. supply component `sup"=(sg",sp",sb")` where:

```
sg"(q,d) = ((tag(q)=1) && sg'(v1(q),d)) ||
           ((tag(q)=2) && (d=rb'(p,b)))
sp"(q,d) =
    if (tag(q)=2)
       if (v2(q)>1) P"(i2(v2(q)-1))
       else P"(i1(p))
    else if (v1(q)=rp'(p,b)) && (d=rb'(p,b)) P"(i2(k-1))
         else P"(i1(sp'(v1(q),d)))

sb"(q,d) = if (tag(q)=2) b else sb'(v1(q),d)
```

- Define new time projections `time":P"`$\rightarrow$`Nat` as follows:

```
time"(q) = if (tag(q)=1) time'(v1(q))
           else time'(p)-v2(q)
```

- Define new space projections `space":P"`$\rightarrow$`Space` by applying some heuristics that utilizes unused space coordinates over `Space` at the new time-steps, or by extending `Space`, if necessary, to appoint new space coordinates.

- Let `D'=D"` and rename `time"` to `time'` and `space"` to `space'`.

□

It is easy to see, that repeated applications of the above steps will turn `D'` into a DDA with `tspan=1`.

Then any computation defined on `D` in terms of its request component can be defined on `D'`, by defining identity-like functions on the new points.

4.3.1 DDA-Embeddings

As we have seen, the DDA of a computation can be turned into a space-time DDA, if we can define a pair of space-time projections that comply with the additional consistency requirement of Definition 4.3.1. This then yields a perfect overview over the parallel execution of any computation defined on the DDA. E.g., a corresponding space-time unfolding of the butterfly DDA can be defined with the projections $\texttt{space:BF}_\texttt{h} \rightarrow \texttt{Nat}$ and $\texttt{time:BF}_\texttt{h} \rightarrow \texttt{Nat}$ as follows:

```
space(p) = col(p)
time(p) = h-row(p)
```

We can see that all computations across space with the same time projection can be executed in parallel, so that the butterfly DDA can be embedded instantly, e.g., to a shared memory model architecture. The "dynamic connectivity", or space-time, of a shared memory model architecture is such that every processor can communicate with every processor at any time, orchestrated by relevant synchronizations (see Section 4.3.2). Hence the space projection can be interpreted directly as processors, and the time projections step through the computation utilizing `next` and requiring synchronization at every time-step.

In other cases, the parallel machine's space-time connectivity is more restricted, i.e., just certain processors can communicate with each other within a time-step. The task is then to find a suitable "match" between the space-time DDA of the computation and the hardware space-time DDA, so that request/supply directions are mapped to existing communication channels. E.g., the butterfly DDA maps nicely to the hypercube STA (see Section 4.3.3) as well. As a general rule, this kind of embedding should always be possible, if the computation's DDA is sub-isomorphic to the hardware space-time DDA. If this is not possible, e.g., the number of processors is significantly less than a one-to-one mapping would require, then the space-time points of the computation can be partitioned, so that a group of points is mapped onto one processor.

In more complicated cases, the embedding will only be possible if it may involve multiple-step communication paths on the hardware side.

Since most examples presented in this dissertation only need to utilize single-step communications on the hardware devices, we somewhat simplify our definition of DDA-embeddings over the formal presentation given in [Haveraaen, 2009, 1990a]. A typical multiple-step hardware communication embedding occurs, e.g., in the shared memory model architecture when

the STA to be embedded is of tspan>1. This is however a specific case and is therefore discussed separately in Section 4.3.2.

Accordingly, given a DDA with an associated computation and a hardware space-time DDA, *an embedding of the computation* into the hardware will be defined by three projections: *EP* which defines how DDA points map into hardware space and time coordinates, *ER* which defines how a request direction at a DDA point is translated into an incoming communication channel, and *ES* which defines how a supply direction at a DDA point is translated into an outgoing communication channel. Hence, DDA-embeddings can utilize explicitly the available hardware resource.

When no ambiguity arises, one may omit the explicit definition of request and supply directions' mappings. In such cases, the embedding can be usually specified only via the space-time projections defined from the DDA point sort.

Since DDA-embeddings are primarily defined via projections, the study of DDA-projections presented in Section 4.4 will be especially relevant in our further investigations.

In the following sections, hardware space-time DDAs of important parallel architectures are presented. They become relevant in later examples when concrete DDA-embeddings are defined for them, e.g., in Section 4.4 and Chapter 6.

4.3.2 *Shared Memory Model Architecture*

In a shared memory model architecture every processor has access to a (typically) large block of random access memory that is shared among all processors with the intent to provide direct communication among them. The processors run in parallel such that their communication is synchronised in some way. DDAs can abstract the processors' communication-topology over time in the form of an STA. Every point of the STA corresponds to a processor at a given time-step with all its branches pointing to all other processors in the previous and upcoming time-steps, including itself.

Definition 4.3.6 (Shared Memory STA)**.** Let `Space = Nat` be a type. A *shared memory space-time DDA for* $\mathtt{S} \in \mathbf{N}^+$ processors, $\mathtt{DSMST_S}$, is defined by:

1. DDA points: $\mathtt{SMST_S}$ `= node Space * time Nat |` $\mathtt{DI_S}$ where:

   $\mathtt{DI_S(p)}$ `= node(p)<S`

2. branch indices: $\mathtt{B_S}$`={0,1,...,S-1}`

3. request components `(rg,rp,rb)` where:

```
rg(p,b) = (0<time(p))
rp(p,b) = SMST_S(b,time(p)-1)
rb(p,b) = node(p)
```

4. supply components `(sg,sp,sb)` where:

```
sg(p,b) = true
sp(p,b) = SMST_S(b,time(p)+1)
sb(p,b) = node(p)
```

□

The choice of branch indices is motivated by the fact that every processor can communicate with any other at any time-step. This has a direct effect on any embedding defined into this space-time. When the embedded DDA is of `tspan=1`, the way the DDA points are mapped into SMST_S space and time coordinates, $\text{EP:P}\rightarrow\text{SMST}_S$, will automatically determine the request and supply direction embedding projections, i.e., `ER(p,b)=node(EP(rp(p,b)))` and `ES(p,b)=node(EP(sp(p,b)))`.

If the embedded DDA happens to be of `tspan>1`, then request directions implicitly get projected into relevant *incoming communication paths* on the SMST_S side, and supply directions into relevant *outgoing communication paths* as demanded by the actual computational strategy utilising the point-projection `EP`. (A theory of multiple-step hardware communication embeddings can be found in [Haveraaen, 2009, 1990a].) To see this, note that

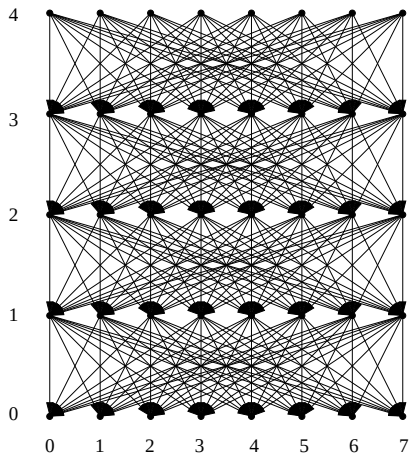

FIGURE 4.11: *Shared memory space-time DDA for* 8 *processors, illustrated in* 5 *time-steps.*

for any processors n:Space and m:Space, in our model, there exists no direct outgoing communication channel ($\mathtt{SMST_S}$ branch) from $\mathtt{SMST_S(n,t)}$ to $\mathtt{SMST_S(m,t+k)}$ whenever k>1. So if for some point p of the computational DDA $\mathtt{EP(p)=SMST_S(n,t)}$ and $\mathtt{EP(sp(p,b))=SMST_S(m,t+k)}$, then there always exists an implicit embedding projection for the supply direction sg(p,b) that will make the value at processor n available at processor m even across multiple tspans, e.g., via the following outgoing communication path in $\mathtt{DSMST_S}$, presented as a list of $\mathtt{SMST_S}$ branches:

$$\mathtt{ES(p,b)=[}\underbrace{\mathtt{node(EP(sp(p,b))),\ldots,node(EP(sp(p,b)))}}_{\mathtt{k\ times}}\mathtt{]=[}\underbrace{\mathtt{m,\ldots,m}}_{\mathtt{k\ times}}\mathtt{]}$$

and the corresponding incoming communication path will be the opposite path.

$$\mathtt{ER(sp(p,b),sb(p,b))=[}\underbrace{\mathtt{m,\ldots,m}}_{\mathtt{k-1\ times}}\mathtt{,n]}$$

The branch-back function of $\mathtt{DSMST_S}$ gives us: $\mathtt{sb(SMST_S(m,t+k-i),m)=m}$ for 1<=i<=k-1, and $\mathtt{sb(SMST_S(n,t),m)=n}$. This information is used in the above embeddings.

Intuitively, the construction shows that the value computed at processor node(EP(p))=n at time-step time(EP(p))=t can be made available for processor node(EP(sp(p,b)))=m already at time-step t+1 and it will remain available for all following k-1 time-steps. The importance of this constructions is purely theoretical, emphasising the fact that even if the embedded DDA is with tspan>1, information can be exchanged over large tspans via implicit hardware communication paths.

Thus whenever we define a DDA-embedding into a shared memory model STA, we will often do so via designated space and time coordinates only defined from the DDA point sort, assuming the presence of request and supply direction embedding projections to be implicit and further not specified.

4.3.3 Hypercube

The *hypercube interconnection network of dimension* $\mathtt{d} \in \mathbf{N}$ is given by $2^{\mathtt{d}}$ processors, each labelled with exactly one of the numbers $\{0,\ldots,2^{\mathtt{d}}-1\}$, such that there exists exactly one communication channel between any two processors whose labels' differ by exactly one bit in their binary representation. This interconnection comprises the static connectivity of the hypercube. Its dynamic connectivity can be defined as follows:

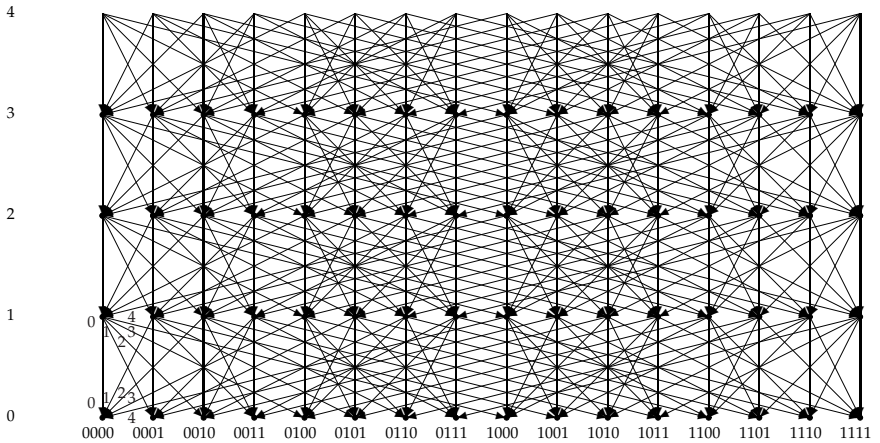

FIGURE 4.12: *Hypercube space-time DDA of dimension* 4 *illustrated in* 5 *time-steps. Channel* 0 *is the communication within a node, the other channels between nodes. The* `node` *projections of* $\mathtt{HST}_4$ *points are presented in binary.*

Definition 4.3.7 (Hypercube STA)**.** A *hypercube space-time DDA of dimension* $\mathtt{d} \in \mathbf{N}$, $\mathtt{DHST}_\mathtt{d}$, is defined by (see Fig. 4.12):

1. DDA points: $\mathtt{HST}_\mathtt{d}$ `= node Nat * time Nat |` $\mathtt{DI}_\mathtt{d}$ where:

   $\mathtt{DI}_\mathtt{d}$`(p) = node(p)<`$2^\mathtt{d}$

2. branch indices: $\mathtt{BHST}_\mathtt{d}$ `= {0, 1, ..., d}`

3. request components `(rg,rp,rb)` where:

   ```
   rg(p,b) = 0<time(p)
   rp(p,b) = if (b = 0) HSTd(node(p),time(p)-1)
             else HSTd(flip(b-1,node(p)),time(p)-1)
   rb(p,b) = b
   ```

4. supply components `(sg,sp,sb)` where:

   ```
   sg(p,b) = 0<=time(p)
   sp(p,b) = if (b = 0) HSTd(node(p),time(p)+1)
             else HSTd(flip(b-1,node(p)),time(p)+1)
   sb(p,b) = b
   ```

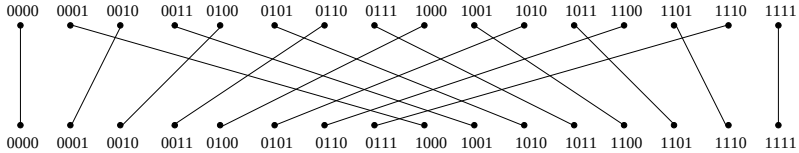

Figure 4.13: *The perfect shuffle permutation of* 2^4 *elements. The new positions are obtained by cyclic shifts of the binary representations of the old positions. Hence* $4 = \log 2^4$*-step interconnection of this pattern with itself brings each element back to its original position.*

□

Branch indices are picked in such a way that they can easily determine the differing bit's position in the binary representation of the processors' labels, now defined by the projection `node(p)`.

Note, that in this dynamic connectivity, each processor gains a new communication channel, i.e., the communication channel within the processor itself, labelled by branch index `0`.

The hypercube space-time DDA is an example of an infinite countable DDA. Here the computations have a starting point (time `0`), but may continue forever.

4.3.4 Omega Network

The *omega network* is an important repetitive network topology for parallel processing based on the perfect shuffle [Stone, 1971]. Given an array of elements, the *perfect shuffle permutation* of the array elements (see Fig. 4.13) is achieved by interlacing the first half of the array with the elements of the second half. This can be viewed as shuffling a deck of cards: first dividing them in half, and then shuffling the two halves perfectly. The omega network is obtained by interconnecting the perfect shuffle permutation pattern with itself such that adjacent elements are coupled together by a network switch performing the right shuffle of the two inputs.

Since a 2^h-element omega network has the property that `h`-step interconnection of this pattern with itself brings each element back to its original position, it is usually defined as a `h`-step network. However, without any loss of generality, we can define it as an infinite countable space-time DDA as follows:

Definition 4.3.8 (Omega Network STA)**.** An *omega network space-time DDA of dimension* $\mathtt{h} \in \mathbf{N}$, $\mathtt{DONST}_\mathtt{h}$ is defined by:

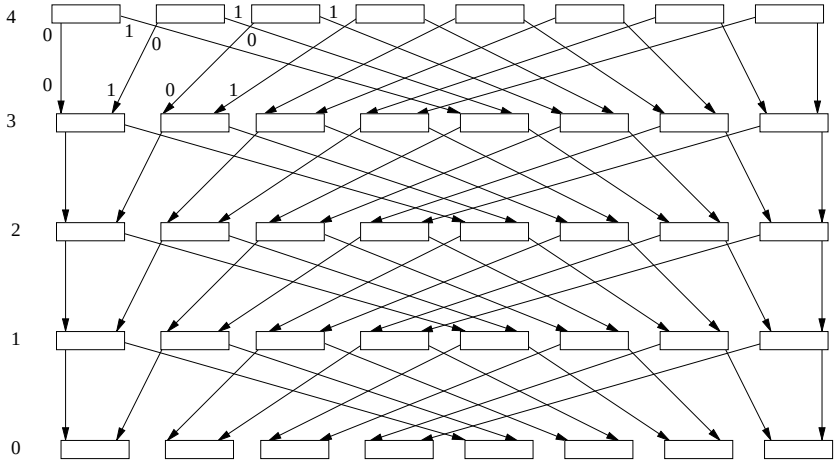

FIGURE 4.14: *Omega network space-time DDA of dimension* 4 *for* 2^4 *elements, where adjacent elements of the perfect shuffle are coupled by an omega network switch, illustrated in* 5 *time-steps.*

1. DDA points: $\mathtt{ONST_h}$ `= switch Nat * time Nat |` $\mathtt{DI_h}$ where:

   $\mathtt{DI_h(p)}$ `= switch(p)<`2^{h-1}

2. branch indices: $\mathtt{B} = \{\mathtt{0},\mathtt{1}\}$

3. request components `(rg,rp,rb)` where:

   ```
   rg(p,b) = (0<time(p))
   rp(p,b) = ONST_h(ShR_h(switch(p)*2+b,1)/2,time(p)-1)
   rb(p,b) = ShR_h(switch(p)*2+b,1)%2
   ```

4. supply components `(sg,sp,sb)` where:

   ```
   sg(p,b) = (0<=time(p))
   sp(p,b) = ONST_h(ShL_h(switch(p)*2+b,1)/2,time(p)+1,)
   sb(p,b) = ShL_h(switch(p)*2+b,1)%2
   ```

   where the shift-right function $\mathtt{ShR_h}$`:Nat,Nat→Nat` is defined such that $\mathtt{ShR_h}$`(n,i)` returns the value of a cyclic shift to the right on the `h`-bit binary representation of `n` by `i` positions. $\mathtt{ShL_h}$`:Nat,Nat→Nat` performs the opposite of $\mathtt{ShR_h}$. □

Each point $\texttt{p:ONST}_\texttt{h}$ stands for a coupled pair of elements, i.e., an omega network switch node. Branch index `0` identifies all left incoming and outgoing communication channels, and branch index `1` all right incoming and outgoing communication channels. The branch indices have the auxiliary role to identify the elements in the switch node as well. Branch index `0` identifies the left, and branch index `1` the right element of each node. This additional information is being explicitly used in the definition of requests and supplies in order to obtain the corresponding element's position in the original index-set of the $\texttt{2}^\texttt{h}$ elements. Then, in the case of the request function, e.g., to this a cyclic shift to the right is applied first, and then the result is projected into the switch node (integer division by `2`) in the row below (`time(p)-1`). Supplies are defined similarly, but there the cyclic shift-left function plays the central role.

4.3.5 CUDA Programming Model

In the past decade, Graphics Processing Units (GPUs), primarily driven by computer game industry, have matured into very powerful computing devices [Owens et al., 2007], featuring hundreds of on-chip processors, and have yet remained "budget" choices. Non-graphics oriented communities soon realised the huge potential of GPU computing power. This prompted graphics vendors to provide higher-level programming models (e.g., ATI's CTM [AMD, 2006], Nvidia's CUDA [NVIDIA, 2010]), especially designed for general-purpose, compute-intensive data-parallel applications. Hence, GPU computing is now widely used in demanding consumer applications as well as high-performance computing [Nickolls and Dally, 2010].

In this section, we focus on NVIDIA's relatively new application programming interface, the *CUDA Programming Model* [Kirk and Hwu, 2010], which provides an intermediate layer for users familiar with the C programming language to easily write programs for NVIDIA GPU devices. CUDA programming is an instance of the single-program multiple-data (SPMD) parallel programming style [Darema et al., 1988] – a popular choice for programming massively parallel processors –, with certain restrictions.

In CUDA, the GPU device operates as a Co-processor to the main CPU. The GPU is capable of handling a huge number of parallel threads each executing the same program, called a *kernel*. The total number of CUDA threads that execute a kernel is specified by the host when the kernel is called and downloaded onto the device. The threads are organised in *grid of blocks* upon the kernel invocation. A *block* contains a limited number of threads, but a grid of blocks may contain any (reasonable) number of blocks. Each thread executing the kernel is automatically assigned a unique thread

and block ID that is accessible within the kernel through built-in variables, and through which the thread can access memory locations on the GPU device. Threads within one block can synchronize and share data through fast on-chip shared memory. Threads in different blocks can only communicate asynchronously via the main GPU memory. There is no guarantee as to which blocks run in parallel, or in which order blocks are sequenced, when the grid has more blocks than can be physically executed in parallel on the device. So threads in different blocks are in practice unable to exchange information within the same kernel. However, since the GPU memory is persistent across kernels, inter-block communications can be attained by ending and re-invoking the kernel. The GPU memory is also a means for initialising computations and outputting results.

Definition 4.3.9. Let ℓ be the maximum number of threads per block as allowed by the hardware, and let `B` $\in \mathbf{N}$ and `T` $\in \mathbf{N}$, `T<=`ℓ represent the number of blocks and threads per block, respectively, of a kernel invocation. Then the *CUDA kernel space-time DDA with* `B*T` *threads*, $\texttt{DCUST}_{\texttt{B,T}}$, is defined by:

1. DDA points:

   $\texttt{CUST}_{\texttt{B,T}}$ `= space` $\texttt{CUB}_{\texttt{B,T}}$ `* time Nat`

   where space is comprised by

   $\texttt{CUB}_{\texttt{B,T}}$ `= block Nat * thread Nat |` $\texttt{DIB}_{\texttt{B,T}}$

   with data invariant:

   $\texttt{DIB}_{\texttt{B,T}}$`(s) = (block(s)<B) && (thread(s)<T)` for all `s:`$\texttt{CUB}_{\texttt{B,T}}$

2. branch indices: $\texttt{CUB}_{\texttt{B,T}}$

3. requests components `(rg,rp,rb)` where:

   ```
   rg(p,b) = (0<time(p))
   rp(p,b) = CUST_B,T(b,time(p)-1)
   rb(p,b) = space(p)
   ```

4. supply components `(sg,sp,sb)` where:

   ```
   sg(p,b) = (0<=time(p))
   sp(p,b) = CUST_B,T(b,time(p)+1)
   sb(p,b) = space(p)
   ```

□

The constant ℓ is device-dependent and on the newest GPUs is 1024. All threads of a kernel are interpreted as the space component, such that $\texttt{CUB}_{\texttt{B,T}}$ identifies the local thread- and block-index a thread belongs to. Here threads have a local thread index `0` through `T-1` and local block index `0` through `B-1`. Branch indices are also of type $\texttt{CUB}_{\texttt{B,T}}$, representing both inter-block and intra-block communication channels between threads. Hence the model assumes that every thread can virtually communicate with every thread at any time-step. Request and supplies are then defined along $\texttt{CUB}_{\texttt{B,T}}$ branches, describing communication between threads in successive time-steps.

Given an embedding into $\texttt{DCUST}_{\texttt{B,T}}$, that is when we want to utilise this space-time for our computation, only certain threads need to exchange data at any one time. Note that since branches here are again identified by the space component (similarly to the shared memory STA), any embedding defined into $\texttt{DCUST}_{\texttt{B,T}}$, can be done so via designated space (threads) and time projections only defined from the point sort of the DDA to be embedded. Thus whenever

```
block(space(EP(p))) = block(space(EP(rp(p,b))))
```

we have fast, intra-block communication, otherwise inter-block communication through the GPU memory. The latter also implies that the kernel has to be ended, control has to be handed over to the host, which then will invoke a new kernel again in order to continue the computation.

Inside the kernel, there is no control over which of the threads – comprised here by the space coordinates of an embedding – at a given time-step would trigger the completion of the kernel. Hence the inter-block communication test cannot be placed inside the kernel, as it would otherwise lead to destructive race-conditions, e.g., if one of the threads requires inter-block communication, yet another does not. However, a *kernel scheduler* can be built outside the kernel. This will be specific to the embedding and will be independent of the actual computation. Based on this, the host loops through all kernel invocations needed to complete the computation. In turn, each kernel terminates at a time appointed by the host, exactly when an inter-block communication is about to take place, to give space to the next kernel. This technique will be shown in Section 5.3.3 where the space-time controlled CUDA execution model is presented.

Note that the abstractions we use for identifying threads and their communications implicitly provide information about data locality as well. This can be extracted by referring to the projections of $\texttt{CUB}_{\texttt{B,T}}$ and $\texttt{CUST}_{\texttt{B,T}}$.

The CUDA kernel space-time DDA abstracts over the connectivity of a system with one GPU device available for the host. However, there exist CUDA-enabled multi-GPU systems, such as the *Tesla Personal Supercomputer*, in which several GPUs are made available for the host. These systems, e.g., support the execution of programs with large data sets that would otherwise overscale the physical limits of one GPU.

The abstraction technique used over one GPU can be extended to obtain the space-time connectivity of a system with an arbitrary number of (possibly heterogeneous) GPUs. If `n:Nat`, `n>1` represents the number of GPUs, then let `B` and `T` be now two arrays indexed by `{0,1,...,n-1}` and element type `Nat` representing the number of blocks and threads per blocks, respectively, of each kernel run on the individual GPUs. Then the space of a Tesla system with `n` GPUs can be comprised by:

$$\texttt{TESLA}_{\texttt{B,T}} = \texttt{gpu \{0,1,...,n-1\} * block Nat * thread Nat | DI}_{\texttt{B,T}}$$

with data invariant for all $\texttt{s:TESLA}_{\texttt{B,T}}$:

$$\texttt{DI}_{\texttt{B,T}}\texttt{(s) = (block(s)<B[gpu(s)]) \&\& (thread(s)<T[gpu(s)])}$$

Branch indices across time can be defined again by the space component, here $\texttt{TESLA}_{\texttt{B,T}}$. The parameter-arrays `B` and `T` indicate that the model distinguishes between the block-grid and thread-block sizes of the kernels to be executed on the GPUs in parallel. Hence the abstraction allows the use of heterogeneous GPUs as well. Then for a given embedding, whenever

```
gpu(space(EP(p))) = gpu(space(EP(rp(p,b)))
```

the thread identified by `space(EP(p))` at time-step `time(EP(p))` requires communication with a thread running on the same device, otherwise the thread is on a different device. Again, all time-steps requiring inter-gpu communication (and within this inter-block communication) can be precomputed, entailing the building of a more elaborate kernel scheduler that instructs the host when to issue kernels for each device, and when to carry out appropriate memory management across the GPUs' memory content.

4.4 DDA-Projections

In all previous examples, we used the implicit projections of the DDA point sort to define how points of the particular DDA is mapped into a

two dimensional spatial grid for layout purposes. For instance, projections $\texttt{row:BF}_h\rightarrow\texttt{Nat}$ and $\texttt{col:BF}_h\rightarrow\texttt{Nat}$ determined the layout of the butterfly DDA $\texttt{DBF}_h$, see Fig. 4.8. In general, for any given DDA, the points can be placed in many different ways in a possibly multi-dimensional grid. We can simply provide new projections from the DDA point sort, one for each dimension, to give a new layout. This is obtained by placing the DDA points in the chosen grid as defined by the new projections, and drawing the corresponding branches between them utilizing the original DDA definition. The DDA point constructor plays an important role here, as it serves as a *gate* between the two sets of projections. Imagine the branches as elastic bands pinpointed by DDA points. The relation between the pins (points) never changes: they are tied together via the elastic bands (branches) – illustrating the dependency itself. So if we reposition the pinpoints, the elastic bands will either shrink or extend obeying the new positions of the pins, resulting in a new layout.

4.4.1 Variations on the Butterly Theme

Let us illustrate this by showing various layouts of the butterfly DDA of height 4, all obtained exclusively from various DDA-projections.

First consider projections: $\texttt{altRow,altCol:BF}_h\rightarrow\texttt{Nat}$ with (see Fig. 4.15):

```
altRow(p) = row(p)
altCol(p) = ShR_h(col(p),1)
```

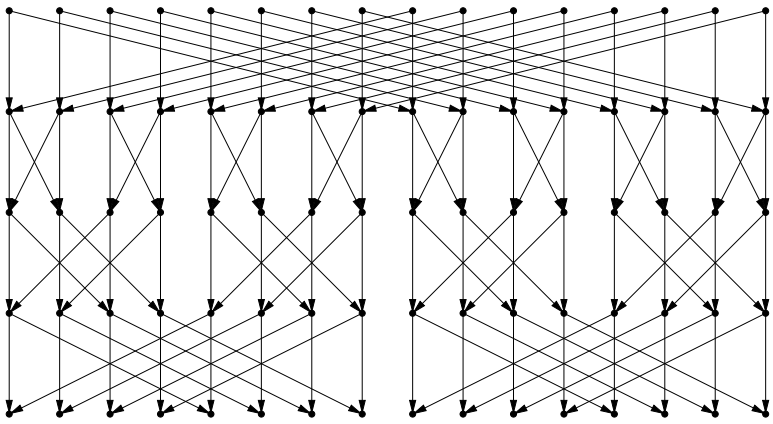

Figure 4.15: *Butterfly DDA of height* 4 *with spatial grid representation defined by* `altRow` *and* `altCol`.

Note that the way butterfly branches cross each other in this new layout is different from the one in Fig. 4.9, yet the dependency is the same.

Each new set of projections requires a constructor such that they together satisfy the consistency requirements discussed in Section 3.2. We denote the new constructor for the point sort $\texttt{BF}_\texttt{h}$ by $\texttt{altBF}_\texttt{h} : \texttt{Nat},\texttt{Nat} \to \texttt{BF}_\texttt{h}$, which is defined in terms of the original constructor $\texttt{BF}_\texttt{h}$, where a and b are arbitrary parameters of type Nat:

$$\texttt{altBF}_\texttt{h}\texttt{(a,b)} = \texttt{BF}_\texttt{h}\texttt{(a,ShL}_\texttt{h}\texttt{(b,1))}$$

It is easy to verify that all consistency requirements (see Section 3.2) of the new projections wrt. this constructor are satisfied, since they are satisfied for the implicit set of projections and constructor by default. The presence of constructors ensures a smooth swap from one set of projections to the other.

We can define projections which may completely ruin the butterfly's symmetric layout, without ruining the dependency itself. E.g., consider the projections $\texttt{assRow,assCol:BF}_\texttt{h}\texttt{→Nat}$, in which we apply the cyclic shifts to only some of the points in the left half of the butterfly (see Fig. 4.16):

$$\begin{aligned}&\texttt{assRow(p)} = \texttt{row(p)}\\ &\texttt{assCol(p)} = \textbf{if}\ \texttt{((row(p)<h-1) \&\& (col(p)<2}^{\texttt{h}-1}\texttt{))}\ \texttt{ShR}_{\texttt{h}-1}\texttt{(col(p),1)}\\ &\qquad\qquad \textbf{else}\ \texttt{col(p)}\end{aligned}$$

with constructor $\texttt{assBF}_\texttt{h} : \texttt{Nat},\texttt{Nat} \to \texttt{BF}_\texttt{h}$:

$$\begin{aligned}&\texttt{assBF}_\texttt{h}\texttt{(a,b)} = \textbf{if}\ \texttt{((a<h-1) \&\& (b<2}^{\texttt{h}-1}\texttt{))}\ \texttt{BF}_\texttt{h}\texttt{(a,ShL}_{\texttt{h}-1}\texttt{(b,1))}\\ &\qquad\qquad\qquad \textbf{else}\ \texttt{BF}_\texttt{h}\texttt{(a,b)}\end{aligned}$$

In projections $\texttt{altAssRow,altAssCol:BF}_\texttt{h}\texttt{→Nat}$ we apply a new projection to some of the points in the right half of the butterfly (see Fig. 4.17) as well:

$$\begin{aligned}&\texttt{altAssRow(p)} = \texttt{row(p)}\\ &\texttt{altAssCol(p)} = \textbf{if}\ \texttt{(row(p)<h-1)}\\ &\qquad\qquad\qquad \textbf{if}\ \texttt{(col(p)<2}^{\texttt{h}-1}\texttt{)}\ \texttt{ShR}_{\texttt{h}-1}\texttt{(col(p),1)}\\ &\qquad\qquad\qquad \textbf{else}\ \texttt{ShR}_{\texttt{h}-1}\texttt{(col(p)-2}^{\texttt{h}-1}\texttt{,row(p))} + 2^{\texttt{h}-1}\\ &\qquad\qquad \textbf{else}\ \texttt{col(p)}\end{aligned}$$

with constructor $\texttt{altAssBF}_\texttt{h} : \texttt{Nat},\texttt{Nat} \to \texttt{BF}_\texttt{h}$:

```
altAssBF_h(a,b) = if (a<h-1)
                     if (b<2^(h-1)) BF_h(a,ShL_(h-1)(b,1))
                     else BF_h(a, ShL_(h-1)(b-2^(h-1),a)+2^(h-1))
                  else BF_h(a,b)
```

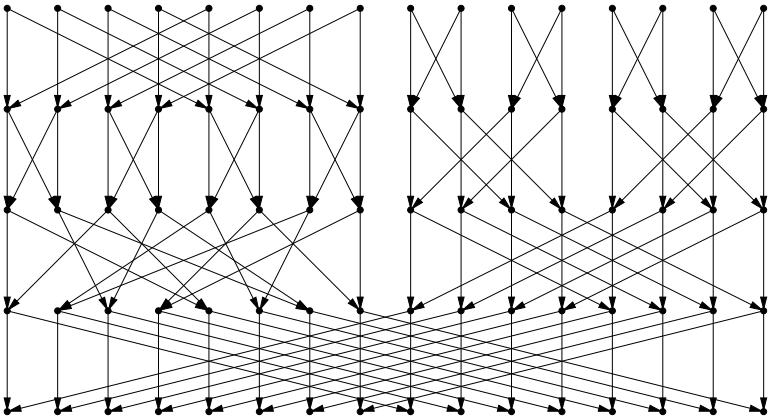

Figure 4.16: *Butterfly DDA of height* 4 *with spatial grid representation defined by* `assRow` *and* `assCol`.

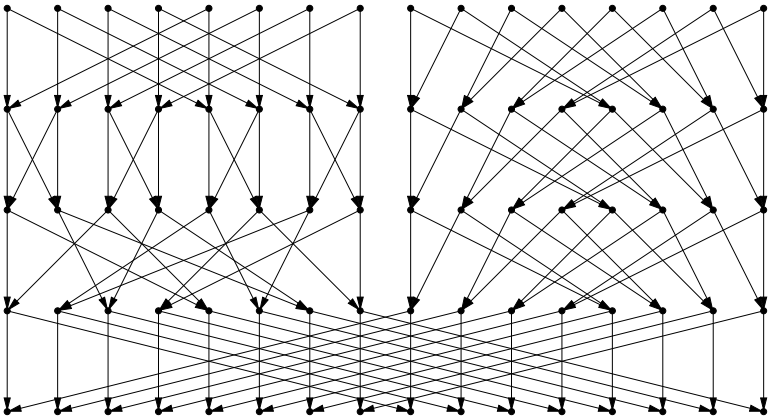

Figure 4.17: *Butterfly DDA of height* 4 *with spatial grid representation defined by* `altAssRow` *and* `altAssCol`.

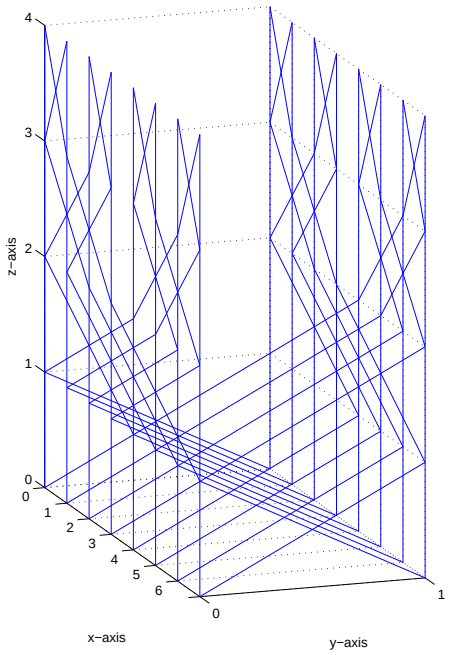

FIGURE 4.18: *Butterfly DDA of height* 4 *with three dimensional grid representation defined by projections* $\texttt{xcor}_3$, $\texttt{ycor}_3$, zcor.

The projections $\texttt{xcor}_\texttt{d}$,$\texttt{ycor}_\texttt{d}$,zcor:$\texttt{BF}_\texttt{h}\to$Nat defined next will lead to three dimensional grid representations of different shapes depending on the value of the parameter $\texttt{d} \in \mathbf{N}$, 0<d<h, see Fig. 4.18 and 4.19:

$$\texttt{xcor}_\texttt{d}\texttt{(p)} = \texttt{col(p)\%2}^\texttt{d}$$
$$\texttt{ycor}_\texttt{d}\texttt{(p)} = \texttt{col(p)/2}^\texttt{d}$$
$$\texttt{zcor(p)} = \texttt{h-row(p)}$$

and constructor $\texttt{dim3BF}_\texttt{h} : \texttt{Nat,Nat,Nat} \to \texttt{BF}_\texttt{h}$:

$$\texttt{dim3BF}_\texttt{h}\texttt{(x,y,z)} = \texttt{BF}_\texttt{h}\texttt{(h-z,y*2}^\texttt{d}\texttt{+x)}$$

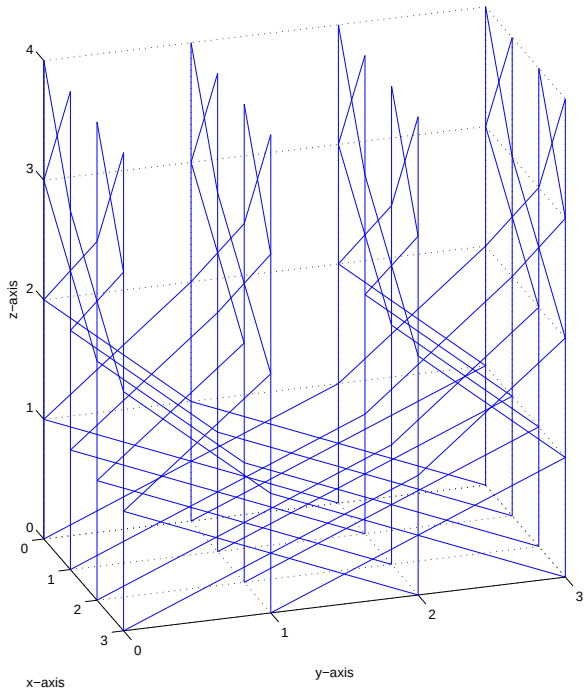

Figure 4.19: *Butterfly DDA of height* 4 *with three dimensional grid representation defined by projections* $\texttt{xcor}_2$, $\texttt{ycor}_2$, `zcor`.

We may use such projections not only when laying out a DDA on paper or in space for visualization purposes, but also as a means of placing computations (at each point) on processors in a network, or in a multi-core, as discussed in Section 4.3. For instance, in the 2D illustrations the various `col` projections or in the 3D illustrations the `xcor` and `ycor` projections may comprise positions in 1D or 2D network topologies, respectively, of multi-core processors, or GPUs, etc. The asymmetric projections further illustrate that a computation, from some time-step onward, could even be split and mapped onto different hardware topologies. Note that `row` and the corresponding alternative row-projections did not change in the examples. Though they can also be altered, within the limits of consistency

requirements, here they play the role of time-projections which determine the space-time execution of any computation defined on the butterfly.

If the network topology happens to be of higher dimension, e.g. a torus, a new set of projections, carved in a similar fashion as before, maps the DDA points directly to torus points. This is achieved by further specifying each DDA point, e.g., in the XY plane as a 3D shape, and so on.

We can also define projections that will result in repetitive network topologies. Due to their structural properties, repetitive networks can be realised at reduced costs. Only the repeating pattern need to be implemented in terms of real hardware. The interconnection is achieved by leading outputs back as inputs. This makes repetitive networks very popular, especially in hardware design.

Consider the *shuffle projections* $\texttt{shuffleRow,shuffleCol:BF}_h \rightarrow \texttt{Nat}$:

$$\texttt{shuffleRow(p)} = \texttt{row(p)}$$
$$\texttt{shuffleCol(p)} = \texttt{ShR}_h\texttt{(col(p),row(p))}$$

with constructor $\texttt{shuffleBF}_h : \texttt{Nat,Nat} \rightarrow \texttt{BF}_h$:

$$\texttt{shuffleBF}_h\texttt{(a,b)} = \texttt{BF}_h\texttt{(a,ShL}_h\texttt{(b,a))}$$

We see that the *shuffle layout* of the butterfly DDA of height 4, achieved in Fig. 4.20, shows similarities with the omega network STA of Fig. 4.14,

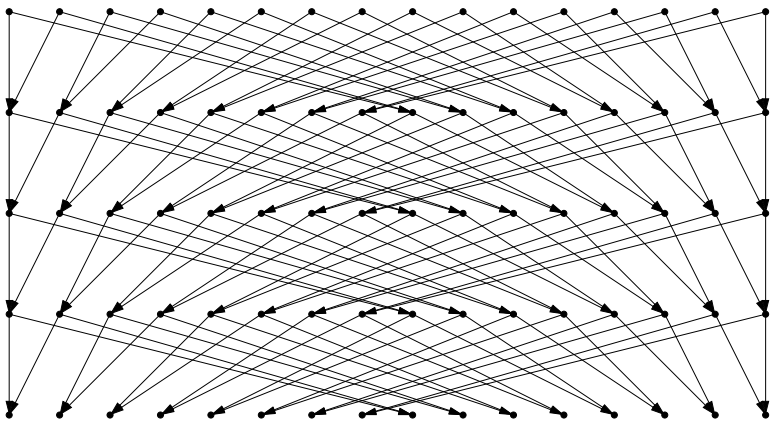

FIGURE 4.20: *Shuffle layout of the butterfly DDA of height* 4 *obtained by projections* `shuffleRow` *and* `shuffleCol`.

the only difference being the differing number of DDA points. However, a 5 time-step omega network STA of dimension 5 yields exactly the shuffle layout of Fig. 4.20. Hence, the above shuffle projections can be used to define a one-to-one embedding of the butterfly DDA $\texttt{DBF}_\texttt{h}$ into the omega network STA $\texttt{DONST}_{\texttt{h}+1}$. However, with DDA-embeddings, we can define a more efficient mapping, which only requires an omega network STA of size `h`, instead of `h+1`.

4.4.2 *An Example of Non-injective DDA-Embedding*

Note that in Fig. 4.20, as by the default view of point-valued computations, each point passes on its result to two different points along its supply directions, in a manner resembling the perfect shuffle. In the omega network, these two receiving points happen to be coupled in a network switch node. So if we couple the adjacent elements in the shuffle layout as well, then all branches carrying the same result will appear as duplicated branches between the coupled nodes. The superfluous presence of wires especially in hardware design is undesired. More wires mean more heat, hence more consumption. So it is desirable to reduce the number of wires as well, if possible.

We can achieve this by defining a non-injective embedding: adjacent points of the shuffle layout will be projected into an omega network space and time coordinate, comprised by $\texttt{ONST}_\texttt{h}$; all request directions of $\texttt{DBF}_\texttt{h}$ will be projected into incoming communication channels of the omega network; and all supply directions into outgoing ones. As a result, branches carrying the same value in Fig. 4.20 will be mapped onto one omega network channel.

We define the *omega embedding projections* $\texttt{EP:BF}_\texttt{h}\rightarrow\texttt{ONST}_\texttt{h}$, $\texttt{ER:rg}\rightarrow\texttt{\{0,1\}}$ and $\texttt{ES:sg}\rightarrow\texttt{\{0,1\}}$ as follows:

$$\begin{aligned}
&\texttt{EP(p)} = \texttt{ONST}_\texttt{h}\texttt{(shuffleCol(p)/2,h-row(p))}\\
&\texttt{ER(p,b)} = \textbf{if } \texttt{(shuffleCol(rp(p,b))} < 2^{\texttt{h}-1}\texttt{)} \textbf{ 0 else } 1\\
&\texttt{ES(p,b)} = \texttt{shuffleCol(p)\%2}
\end{aligned}$$

The result of these projections is seen in Fig. 4.21. This layout is now isomorphic to the one in Fig. 4.14.

It may not be obvious from the definition of the omega projections that they are consistent wrt. the connectivity of the omega network STA, i.e., the projected points, request and supply directions comply with the dependency properties of the omega network STA, expressed in its request and supply components. For instance, if `(p,b)` is a request direction in the

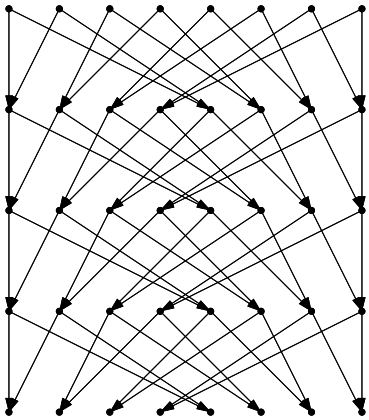

FIGURE 4.21: *Omega network space-time embedding of the butterfly DDA of height* 4 *obtained by embedding projections* `EP`, `ER` *and* `ES`. *All supply directions at a point in the butterfly are mapped onto one outgoing communication channel of the omega network STA.*

butterfly DDA, then `(EP(p),ER(p,b))` should be a request direction in the omega network STA as well, otherwise the omega projections are invalid. We can see, that this property holds, since both request guards are defined in terms of the `row` projection only. Likewise, `(EP(p),ES(p,b))` will be a supply direction in the omega network STA, whenever `(p,b)` is a supply direction in the butterfly DDA.

The reader may easily check that the following properties also hold, which ensure that request points and branch back values also get projected consistently onto correct omega network STA request points and branch back values, where `rp'`, `rb'`, `sp'` and `sb'` denote the omega network STA request and supply components:

```
EP(rp(p,b)) = rp'(EP(p),ER(p,b))
ES(rp(p,b),rb(p,b)) = rb'(EP(p),ER(p,b))
```

Likewise, the supplies will also be consistent:

```
EP(sp(p,b)) = sp'(EP(p),ES(p,b))
ER(sp(p,b),sb(p,b)) = sb'(EP(p),ES(p,b))
```

These properties allow us also to reformulate any computation defined on the butterfly DDA to a computation defined on the omega network STA of reduced dimension. This involves the mapping of the point-valued computations of $\mathtt{DBF_h}$ onto branch-valued computations of $\mathtt{DONST_h}$.

Assume that we have the following point-valued computations associated with $\mathtt{DBF_h}$, expressed in the array `V` indexed by $\mathtt{BF_h}$ and some element type `E`, as follows:

```
V[p] = foo(V[rp(p,0)],V[rp(p,1)])
```

Since each supply direction of a point carries the same result, we can write the above expression to an equivalent one by the means of an array `V'` indexed now by `sg` and some element type `E`, as follows:

```
V'[p,b] = foo(V'[rp(p,0),rb(p,0)],V'[rp(p,1),rb(p,1)])
```

Note that the appearance of `b` on the left side is not read on the right side of the equality.

Now we are ready to make use of the omega projections. First note that a value `V'[p,b]` at the `(p,b)` supply direction in $\mathtt{DBF_h}$ will be computed at the `(EP(p),ES(p,b))` supply direction in $\mathtt{DONST_h}$ (note that `ES(p,b)` does not depend on `b`). So we create an array for $\mathtt{DONST_h}$, and transfer the computation into that. Let `W` be an array of index type `sg'` – denoting the supply guard of $\mathtt{DONST_h}$, and same element type `E`. Then the expression above can be turned into:

```
W[EP(p),ES(p,b)] =
  foo(W[EP(rp(p,0)),ES(rp(p,0),rb(p,0))],
      W[EP(rp(p,1)),ES(rp(p,1),rb(p,1))])
```

Utilising the consistency properties of the omega projections discussed above, we finally obtain an equivalent branch-valued expression in terms of the omega network STA dependency as follows:

```
W[EP(p),ES(p,b)] =
  foo(W[rp'(EP(p),ER(p,0)),rb'(EP(p),ER(p,0))],
      W[rp'(EP(p),ER(p,1)),rb'(EP(p),ER(p,1))])
```

The omega projections illustrates how a DDA can be mapped onto a smaller network topology with well defined connectivity, expressed here in terms of the omega network STA, with the aim of porting computations onto the network. The technique of mapping the result of a computation at

a DDA point onto an outgoing communication channel of a network node hints at the idea that computations abstracted over a DDA can be easily realised as circuits, e.g., on an FPGA. In Section 5.3.4, a DDA-based high-level work-flow of FPGA programming is presented.

The examples presented here show that projections make DDAs very flexible and easy to map to different network topologies. The illustrations of this section representing various DDA layouts defined from projections are obtained from either an xFig or MATLAB drawing automatically generated from the DDA implementation corresponding to Example 4.2.3, plus the specified set of projections. This underlines the fact that the mapping of computations specified on the top of DDAs can be handled on a high-level, without interfering with the original DDA-implementation or the computation itself. The case studies of Chapter 6 further illustrate these ideas on concrete computations.

CHAPTER 5

DDA-based Execution Models

The previous chapter's primary aim was to get the reader familiarised with the concept of DDAs, their kinds and structural properties, and the notion of DDA-embedding projections as a means to map computations onto various hardware architectures. This chapter shows the other side of the coin by presenting the execution models associated with different hardware space-time DDAs, and consequently with DDA-embeddings.

We proposed a language construct in [Burrows and Haveraaen, 2009b], the *repeat statement*, which was designed to encapsulate a DDA-based computational expression, and presented two execution models aiming at computing the meaning of the repeat statement: one for a single-processor and one for the shared memory model architectures. Starting up with a slightly improved version of that presentation, in this chapter, we discuss issues regarding the initialization of the repeat statement, and motivate the need for an alternative, more elaborate *targeted* repeat statement. Its syntax and semantics are presented, and related execution models are defined for the single-processor and shared memory model architectures. In addition, we expand the arsenal of our execution models and address the message passing programming model, the generalised CUDA programming model and FPGA-programming.

5.1 The Repeat Statement

We motivate the introduction of the repeat statement with an example. Let `V` be an array indexed by a type `P` with element type `E`. Further, let `B` be a

type with constants 0:B, 1:B and a partial function rp:P,B→P guarded by rg. Assume that the array V is such that the data values in V are related to each other by:

```
V[p] = if (cond(p)) min(V[rp(p,0)],V[rp(p,1)])
       else max(V[rp(p,0)],V[rp(p,1)])
```

where min:E,E→E and max:E,E→E are minimum and maximum functions on elements, and cond is some condition on indices p:P. Recall that the guards (here rg and ig – the implicit guard of the indexing operation of the array type) ensure that expressions are only applied when they are well-defined; thus the equation is not considered if ig(V,p) does not hold for some p. However, if the right-hand side expression was well-defined in this case – it could be used to define the value of V at the point p. The relation could then be used to define the value of V at any point p:P for which ig(V,p) does not hold originally but the corresponding right-hand side expression becomes well-defined. This attributes a *repetitive semantics* to this relation, motivating the proposal of a repeat statement that will allow us to do this.

We focus the repeat statement on the array data type. Array types can be considered functions mapping from an index type to an element type. Due to our focus on how to compute the contents of an array step by step, we prefer to use the data oriented array concept rather than the corresponding map.

Arrays can represent any structured data type such as lists and trees. A structured data type can be thought of as implicitly indexed by the traversal pattern used to access a given element. For a list, e.g., we can encode the traversal as the number of tail operations before the head operation that accesses a given element, and use this number as an index.

Before presenting the syntax and semantics of the repeat statement, we need to introduce some concepts.

Definition 5.1.1 (1-reachable points)**.** Let D=<P,B,req,sup> be a DDA and S a selection of points from P. Then a point p:P is 1-*reachable from* S *along* D, if all relevant rp(p,b) belong to S. □

Definition 5.1.2 (e-derived extension)**.** Let D=<P,B,req,sup> be a countable DDA, V an array indexed by P with element type E, and e an expression of type E such that each occurrence of the array V in e has the form V[rp(p,b)] for expressions b of type B.

Let S be the set of points where ig(V,p) holds. Then the e-*derived* 1-*extension of* V *along* D is the array $\texttt{ext}_{\texttt{D,e}}\texttt{(1,V)}$, s.t., $\texttt{ig(ext}_{\texttt{D,e}}\texttt{(1,V),p)}$ holds exactly when p is in S or p is 1-reachable from S along D, and

$\texttt{ext}_{\texttt{D,e}}\texttt{(1,V)[p]} = \textbf{if}\ \texttt{(p}\ \textbf{in}\ \texttt{S) V[p]}\ \textbf{else}\ \texttt{e}$

The e-*derived* k-*extension of* V *along* D is the array $\texttt{ext}_{\texttt{D,e}}\texttt{(k,V)}$, for the remaining $\texttt{k} \in \mathbf{N}$, defined recursively by:

$\texttt{ext}_{\texttt{D,e}}\texttt{(0,V)} = \texttt{V}$
$\texttt{ext}_{\texttt{D,e}}\texttt{(k,V)} = \texttt{ext}_{\texttt{D,e}}\texttt{(1,ext}_{\texttt{D,e}}\texttt{(k-1,V))}$

The e-*derived extension of* V *along* D is the array $\texttt{ext}_{\texttt{D,e}}\texttt{(V)}$ defined as the limit of $\texttt{ext}_{\texttt{D,e}}\texttt{(k,V)}$ as k grows. □

When no ambiguities arise, $\texttt{ext}_{\texttt{D,e}}\texttt{(V)}$ may be referred to as *the derived extension of* V.

Note that the array $\texttt{ext}_{\texttt{D,e}}\texttt{(V)}$ is well-defined for any countable P. Consider any value p:P. For a suitably large k, either:

- $\texttt{ig(ext}_{\texttt{D,e}}\texttt{(k,V),p)}$ holds, in which case $\texttt{ig(ext}_{\texttt{D,e}}\texttt{(V),p)}$ holds, and $\texttt{ext}_{\texttt{D,e}}\texttt{(V)[p]} = \texttt{ext}_{\texttt{D,e}}\texttt{(k,V)[p]}$; or
- $\texttt{ig(ext}_{\texttt{D,e}}\texttt{(k,V),p)}$ does not hold, and there is no larger k that will make it hold, thus $\texttt{ig(ext}_{\texttt{D,e}}\texttt{(V),p)}$ does not hold.

If there are any circularities in the DDA, such points will not be given a value with this extension (unless they are predefined in V). The circularity will prevent all required V[rp(p,d)] to be defined, since some of these (indirectly) depend on V[p] which has not been computed. Also, points which do not require any inputs (the rg does not hold) will not be given a value, unless V already defines them.

The following lemma states that the order in which the 1-reachable points are chosen has no effect on the final result.

Lemma 5.1.3 (e-derived extension is unique)**.** Let D=<P,B,req,sup> be a countable DDA, V be an array indexed by P with element type E, and e an expression of type E such that all occurrences of the array V in e has the form V[rp(p,b)] for b an expression of type B.

Assume p:P is a 1-reachable point from the set containing all points where V is defined, and let V' be the array V extended with the value of e at p. Then the e-derived extension of V along D equals the e-derived extension of V' along D. □

Proof: First note that the e-derived (k+1)-extension of V along D always contains the e-derived k-extension of V' along D. Then note that the e-derived k-extension of V' along D always contains the e-derived k-extension of V along D. Thus at the limit when k grows, both extensions will be equal. □

Following [Burrows and Haveraaen, 2009b], the syntax and semantics of the repeat statement is defined as follows.

Definition 5.1.4 (Syntax of the repeat statement)**.** Consider a countable DDA `D=<P,B,req,sup>`. Let `V` be an array with index type `P` and element type `E`. Then the *the syntax of the repeat statement* is given by:

```
repeat p:P along D from V in e
```

where **`repeat`**, **`along`**, **`from`** and **`in`** are keywords, imposing the following syntactic and type requirements:

- `e` must be an expression of type `E`
- `p` is a freshly declared variable of type `P`, with scope `e`.
- All occurrences of the array `V` in `e` must have the form `V[rp(p,b)]` for expressions `b` of type `B`, and `rp` is the request from the DDA `D`. □

Note that the use of the `rp`-function in the repeat statement indicates that there is a data dependency between the computational steps of the application. The DDA `D` gives exactly what this data dependency is. This provides a clean interface between the application `e` and its possible dependency patterns.

Definition 5.1.5 (Semantics of the repeat statement)**.** The *meaning of a syntactically and type correct repeat statement* with countable data dependency `D=<P,B,req,sup>`,

```
repeat p:P along D from V in e,
```

is the `e`-derived extension of `V` along `D`. □

Example 5.1.6. The instantiation of the repeat statement for our original example, using the butterfly DDA of height `h`, is:

```
repeat p:P along DBF_h from V in
       if (cond(p)) min(V[rp(p,0)],V[rp(p,1)])
       else max(V[rp(p,0)],V[rp(p,1)])
```

□

A repeat statement becomes an imperative statement by placing the computed values into V itself rather than returning a new array.

Note that the statement itself is a closure statement since it computes all missing pieces of V (or of the derived extension) – as much as possible – using e.

From a computational point of view, all points p:P for which ig(V,p) originally holds play the role of *input points*, i.e., all new values of the derived extension are computed starting up from these *initial values*. Hence all points p:P where $\texttt{ig}(\texttt{ext}_{\texttt{D,e}}(\texttt{V}),\texttt{p})$ holds but ig(V,p) did not hold originally are *output points*.

In Example 5.1.6 typical input points are those of the bottom row, i.e., ig(V,p) would normally hold for all points $\texttt{p:BF}_\texttt{h}$ such that row(p)=h. Then all other points of the butterfly DDA for which a value has been computed, given the expression, will be output points. In another setting, initial values could be defined for a different set of input points, which will obviously lead to a different derived extension. Or initial values may not be defined at all.

The fact that the repeat statement does not impose any requirement on the array V (apart from its index and value type) makes such scenarios feasible. Therefore prior to the repeat statement ig(V,p) may hold for an arbitrary set of indices, or may not hold at all for any index. In the latter case, the meaning of the repeat statement will be an array with no computed (nor initial) values at all. Note that this is fully compatible with the definition of the e-derived extension of V along D (see Definition 5.1.2), i.e., S will be the empty set whenever ig(V,p) does not hold at all, hence V cannot be extended, if it has no initial values.

Even though in the expression e some of the points could be initialized via sub-expressions bypassing the dependency, however, these will not have any effect over the meaning of the repeat statement. The reason behind this is that the extension, as per Definition 5.1.2, de facto *is* derived along 1-reachable points by computing a new value *from* the dependencies. This ultimately implies that any initialization of the array V should be dealt with prior to issuing the repeat statement. Hence, in the following we assume, unless otherwise stated, that initial values have been always assigned to V prior to the repeat statement.

In practice, we are often not interested in the whole derived extension of V. For instance, in Example 5.1.6, we would perhaps be interested only in the values of the top row of the butterfly, and not so much in the "intermediate" values. This motivates the need of an additional syntactic element in the repeat statement which, in such cases, can focus the meaning of the repeat statement on a subtype of the indices P deemed directly as *target points*.

Further expanding here the details of [Burrows and Haveraaen, 2009b], a more elaborate repeat statement is introduced, referred to as *the targeted repeat statement*.

Definition 5.1.7 (Target)**.** Let `P` be some index type. Then a *target for* `P` is a predicate `TP:P→Bool`. A point `p:P` is called a *target point* whenever `TP(p)` holds. If `V` is an array with index type `P`, then `V[p]` is called a *target value* if `p` is a target point, i.e., if `TP(p)` holds. □

Note that the target points of `P` are not specific to any DDA having `P` as a point type. Their sole purpose becomes evident only in the context of a specific computation defined on the DDA.

Definition 5.1.8 (Targeted `e`-derived extension)**.** Let `D=<P,B,req,sup>` be a countable DDA, `V` be an array indexed by `P` with element type `E`, and `e` an expression of type `E` such that all occurrences of the array `V` in `e` has the form `V[rp(p,b)]` for expressions `b` of type `B`. Further let `TP:P→Bool` be a target for `P`.

Then the `TP`-*targeted* `e`-*derived extension of* `V` *along* `D`, denoted by $\texttt{ext}^{\texttt{TP}}_{\texttt{D,e}}\texttt{(V)}$, is obtained from $\texttt{ext}_{\texttt{D,e}}\texttt{(V)}$ by disregarding all irrelevant values of the extension, i.e., for all `p:P` where `TP(p)` does not hold `ig(`$\texttt{ext}^{\texttt{TP}}_{\texttt{D,e}}\texttt{(V),p)}$ does not hold either, and $\texttt{ext}^{\texttt{TP}}_{\texttt{D,e}}\texttt{(V)}$ is otherwise identical to $\texttt{ext}_{\texttt{D,e}}\texttt{(V)}$. □

When no ambiguities arise, $\texttt{ext}^{\texttt{TP}}_{\texttt{D,e}}\texttt{(V)}$ may be referred to as *the targeted derived extension of* `V`.

If `D` is a DDA with infinitely, or just way too many points, identifying or computing the collection of all target points (of a computation) may not be possible. These situations can be handled by considering a sub-type of `P` representing finite number of DDA points only, and constructing the corresponding sub-DDA of `D`, and consider the computation on that. A technique showing such a construction is shown in Section 7.1.3. Alternatively, we can achieve similar results by altering the request/supply guards of the DDA. Hence, we assume that the collection of all target points of a computation can always be identified.

Proposition 5.1.9. A `TP`-targeted `e`-derived extension of `V` along `D` is obtainable from the first `e`-derived `k`-extension of `V` along `D` that has all target values defined, for some $\texttt{k} \in \mathbf{N}$. □

Proof: Note that all new values of the `e`-derived extension of `V` along `D` computed from the `e`-derived `k`-extension of `V` along `D` will become irrelevant in the `TP`-targeted `e`-derived extension of `V` along `D`, hence there is no need to compute them. □

Depending on the choice of target points the targeted derived extension of V may or may not have all target values defined. E.g., if the derived extension of V is not defined for some of the target points, the targeted derived extension of V will not be either. This implies that none of the undefined target values will be computable given the information we have, since the derived extension would have otherwise defined them.

Definition 5.1.10 (Syntax of the targeted repeat statement)**.** Consider a countable DDA D=<P,B,req,sup>. Let V be an array with index type P and element type E, and TP:P→Bool a target for P. Then the *the syntax of the targeted repeat statement* is given by:

```
repeat p:P along D from V for TP in e
```

where **repeat**, **along**, **from**, **for** and **in** are keywords, imposing the same syntactic and type requirements as the regular repeat statement of Definition 5.1.4 □

Definition 5.1.11 (Semantics of the targeted repeat statement)**.** The *meaning of a syntactically and type correct targeted repeat statement* with countable data dependency D=<P,B,req,sup>,

```
repeat p:P along D from V for TP in e,
```

is the TP-targeted e-derived extension of V along D. □

The targeted repeat statement is a closure statement here as well: computing now the target values of V (via its derived extension) – as much as possible – using e along the dependency D.

Note that the collection of target points may not be identical with the collection of output points of a targeted repeat statement. If the targeted extension is not defined for some of the target points, then these points cannot be considered output points.

When TP(p) holds for all p:P, i.e., we are interested in anything that can be computed, the targeted repeat statement's semantics becomes the regular repeat statement's semantics.

Example 5.1.12. Let us instantiate the targeted repeat statement for our initial example, and define the target points to be the points of the top row of the butterfly DDA, i.e., $\texttt{TP:BF}_\texttt{h} \rightarrow \texttt{Bool}$ is defined by:

```
TP(p) = (row(p)=0)
```

Then the meaning of the targeted repeat statement:

```
repeat p:P along DBF_h from V for TP in
       if cond(p) min(V[rp(p,0)],V[rp(p,1)])
       else max(V[rp(p,0)],V[rp(p,1)])
```

are the values computed for the top row of the butterfly DDA, as much as possible from the initial values of V. □

Note that none of the repeat statements implies any specific computational algorithm. One possible algorithm would be to take the standard approach to recursion, as we find it in most programming languages. The implementation would start with a point, in case of the targeted repeat statement it would start with a target point, and trace down the requests until we end up at known values, compute what we can, and backtrack the computation one step. This traces the dependency by a tree-directed (top-down) traversal of the points, giving rise to exponential (in the number of branches) computing time per starting point. The use of memoization techniques may reduce such computational work significantly.

Another simple idea is to repeatedly try all points p:P, computing a value every time the conditions for a one-step computation are met. This process terminates when there are no 1-reachable points from the set of (currently) computed values or – in case of the targeted repeat statement – when all target values have been computed. Again this gives an exponential running time.

Instead, we will provide several computing strategies which will explicitly utilise the underlying data dependency, and by doing so they will also provide control over resource usage.

5.2 Dependency-driven Computation

Given either a regular or a targeted repeat statement we may exploit the supply component of the DDA to achieve an efficient computation. We traverse the opposite graph of the requests, using the supply direction, which will lead to a dependency-driven computation.

Definition 5.2.1 (Dependency-driven computation)**.** Given a repeat statement **repeat** p:P **along** D **from** V **in** e with countable data dependency D=<P,B,req,sup> where the element type of V is E.

The *dependency-driven computation of the repeat statement* will gradually fill in a fresh array V' with index type P and element type E. V' contains all the values that have been computed up to a certain step during the computation. Initially V' is a copy of V together with its guard.

The algorithm uses a set F and a support array M which contain intermediate information about the computation, and both will be updated at every step of the computation:

- The set F⊆P is the set of indices where V' contains a value, but where this value has not been propagated in the DDA (along the supply direction). Initially F is the set of all indices where V is defined.
- The array M has index type P and as elements arrays with index type B and element type E. In M[p][b] the value that will be requested by point p via branch index b is stored, until the value at p has been computed. Initially M is an empty array, i.e., its guard never holds.

We need a modified version e' of e, where all occurrences of the pattern V[rp(p,b)] have been replaced by M[p][b]. The computation repeats the following steps until F becomes empty:

- Choose a point q∈F and remove it from F.
- For all branches d:B such that sg(q,d) holds do:
  - Let p=sp(q,d)
  - If ig(V',p) does not hold, do:
    - let M[p][sb(q,d)]=V'[q], creating the sub-array if needed,
    - if p has received values from all branches b where rg(p,b) holds, then add p to F, compute e' and insert the value in V' at point p, and empty the sub-array M[p] (ig(M,p) does not hold).

When the intermediate F becomes empty, M can be freed completely, and the array V' is the result. □

Proposition 5.2.2. Throughout the dependency-driven computation of the repeat statement a point p is 1-reachable from the defined points of V' when M[p][b] has been defined for all b where rg(p,b) holds. □

Proof: Let p be a point such that M[p][b] has been defined for all relevant b. Throughout the computation the only way M[p][b] has been assigned a value is through the assignment M[p][sb(q,d)]=V'[q], where V'[q] is defined, p=sp(q,d) and b=sb(q,d) for some d where sg(q,d) holds. Then, by the DDA axioms of Definition 4.1.1, we get that q=rp(p,b) and d=rb(p,b). Hence V'[rp(p,b)]=V'[q] is also defined. This property can be deduced for all b where rg(p,b) holds, and hence, by Definition 5.1.1, p is 1-reachable from the defined points of V'. □

Proposition 5.2.3. The dependency-driven computation **repeat** p:P **along** D **from** V **in** e with countable DDA D=<P,B,req,sup> computes the e-derived extension of V along D. □

Proof: In its evolving computation of V', the algorithm works by taking a point p 1-reachable from the defined points of V' (in the sense of Proposition 5.2.2), computing a value for it, and adding this value to V'. Since this value is in the e-derived 1-extension of V' along D, we know by Lemma 5.1.3 that the e-derived extension of the extended array V' along D will give the required semantics to the repeat statement.

To see that the right value is computed for a point p note that M[p][b] = V'[rp(p,b)], for all b:B where rg(p,b) holds and thus e' will compute the required value.

The algorithm will stop when F becomes empty, but this can only happen when there is no point p 1-reachable from the defined points of V', hence we are at the limit where V' equals the e-derived extension of V' along D. In other words, the computation stops exactly when it has computed the e-derived extension of V along D. □

Note that in practice the computation will not terminate when an infinitely large part of P can be computed. But at the limit of this computation, the e-derived extension of V' along D will have been computed.

In this way the algorithm traverses P from the defined indices of V, computing V'[p] when all of its dependents have been computed, until we have computed as much of V' as can be defined given the information we have.

The run-time cost of the dependency-driven computation is of the order of the size of the computed elements of V', since every point passes through F at most once, and we need to compute once for every such point. Thus the dependency-driven computation gives us control of the computation time of the repeat construct.

The size of the intermediate data structures is not under control of this algorithm. The order in which the points of F are chosen will influence the number of points to be kept in F and the number of intermediate arrays M must hold at any one time. Thus the maximal space used by the computation is not being controlled, implying that the memory requirements are not known.

If the repeat statement was targeted, at the end of the computation we only retain those values of V' for which TP(p) holds. This would however require to retain all irrelevant values of V' throughout the whole computation, and to compute the whole derived extension of V, even if this is not necessary. Instead, based on the observation of Proposition 5.1.9, we can focus the dependency-driven computation on the target values from the very

beginning, and end the computation as soon as all target values have been computed.

Definition 5.2.4 (Targeted dependency-driven computation)**.** Given a targeted repeat statement **repeat** p:P **along** D **from** V **for** TP **in** e where the DDA D=<P,B,req,sup> is countable and the element type of V is E.

The *targeted dependency-driven computation* will gradually fill in a fresh array V' with index type P and element type E. V' contains all target values that have been computed up to a certain step during the computation and all other values not yet propagated in the DDA. The values of V' that are not target values will be gradually discarded. Initially V' is a copy of V together with its guard.

The algorithm uses two sets F and L, and a support array M which contain intermediate information about the computation, and all will be updated at every step of the computation:

- The set F⊆P, as before, is the set of indices where V' contains a value, but where this value has not been propagated in the DDA. Initially F is the set of all indices where V is defined.
- The set L⊆P is the set of all target points that have not yet been computed. Initially L contains all points p:P such that TP(p) holds.
- The array M has index type P and as elements arrays with index type B and element type E. As before, in M[p][b] the value that will be requested by point p via branch index b is stored, until the value at p has been computed. Initially M is an empty array, i.e., its guard never holds.

We work again with a modified version e' of e, where all occurrences of the pattern V[rp(p,b)] have been replaced by M[p][b]. The computation repeats the following steps until F or L becomes empty:

- Choose a point q∈F and remove it from F.
- For all branches d:B such that sg(q,d) holds do:
  - Let p=sp(q,d)
  - If ig(V',p) does not hold, do:
    - let M[p][sb(q,d)] = V'[q], creating the sub-array if needed,
    - if p has received values from all branches b where rg(p,b) holds, then add p to F, compute e' and insert the value in V' at point p, and empty the sub-array M[p].

- If q∈L then remove it from L, otherwise empty V'[q].

As soon as either F or L becomes empty, M can be freed completely. If the computation terminates at F becoming empty, then the array V' is the result with as much target values computed as possible. If the algorithm terminates at L becoming empty, then for all remaining points q from F, V'[q] is also emptied, leaving as a result the array V' with all target points computed, and only those. □

Proposition 5.2.5. The targeted dependency-driven computation **repeat** p:P **along** D **from** V **for** TP **in** e with countable DDA D=<P,B,req,sup> computes the TP-targeted e-derived extension of V along D. □

Proof: First note that a defined, non-target value of V' is only discarded after this value has been propagated in the DDA. Hence, a point p can be considered 1-reachable from the defined values of V', indirectly, as soon as all M[p][b] have been assigned a value for all relevant b:B, even if V'[rp(p,b)] has been discarded in the meantime (see also Proposition 5.2.2). Then by the DDA axioms (Definition 4.1.1) we know that the right value is computed for any such point. Hence the computation evolves by taking again a 1-reachable point, computing a value for it, and adding it to V'. At the same time it discards any non-target value which has just been propagated in the DDA to all relevant locations, hence these will be "remembered" in the intermediate array M for later computations.

The algorithm stops when either F becomes empty, i.e., no more 1-reachable points can be computed, hence no more target values can be computed either, or when L becomes empty, i.e., we have computed all target values. The systematic disposal of irrelevant values of V' ensures that only target values will be kept in the final V'. Thus the computation stops when the TP-targeted e-derived extension of V along D has been computed. □

The run-time cost of this algorithm is at most the run-time cost of the previous algorithm. It will still be of the order of the the size of all computed elements of V' (including all discarded values as well), but the computation here may end without the need to re-visit every such point in F. The order in which the points of F are picked will influence here the execution time as well, since the sooner target points are picked from F, the sooner the computation may end.

We still do not have control over the maximal space usage. However, compared to the previous algorithm, the size of V' will be reduced.

5.3 Space-time Controlled Repetition

The dependency-driven computations of the previous section are sequential algorithms, dealing with one point at a time. In many circumstances the computation for different points are independent of each other. Therefore, it should be possible to introduce some parallelism into the computation mechanism. We will do this by exploiting the space-time projections of the DDA points.

When the underlying DDA can be seen as a space-time DDA, with the corresponding `space` and `time` projections (see Definition 4.3.1), we have control over the space-time execution of the repeat statement.

Example 5.3.1. To illustrate this, consider the butterfly DDA $\texttt{DBF}_\texttt{h}$ of height `h` (Example 4.2.2) together with some DDA-based repeat statement which defines for each relevant $\texttt{p:BF}_\texttt{h}$ how `V[p]` is to be computed. We consider the `row` and `col` projections as `time` and `space`, as pointed out earlier:

```
space(p) = col(p)
time(p) = h-row(p)
```

One can easily verify that the butterfly dependency with these projections satisfies the consistency requirements of a space-time DDA.

Assume that initial values are defined for the points on the bottom row. Then all new points in the derived 1-extension of `V` can be computed in parallel, independently of each other. In fact, all values at $\texttt{BF}_\texttt{h}$ points within the same `row` are independent of each other, therefore can be computed in parallel. □

We can step through the computations in each repeat statement from time-step `zero` and upwards, using the increment operator `next` to move from one time-step to the next. At each time-step `t:Time`, we choose all the points $\texttt{BF}_\texttt{h}\texttt{(t,s)}$ for all relevant `s:Space`. This allows the parallel computation of all elements at the given time-step `t`, since the requirements assert that the computation for each spatial index is independent of each other.

In the following sections, we present several space-time algorithms as allowed by various parallel hardware architectures.

5.3.1 *Shared Memory Model Execution*

As we have seen in Section 4.3.2, in a shared memory model architecture processors can communicate with each other at any time via the globally available shared memory space. Let `Space` be the type representing

the available processors, and let `Time = Nat`. The computational DDAs of the repeat statements are assumed to be embedded into the corresponding shared memory model STA via dedicated `space` and `time` coordinates defined from the DDA point sort, turning every such DDA into an STA. For computability reasons we assume the STAs to be of finite span.

The space-time controlled repetition in shared memory as presented in [Burrows and Haveraaen, 2009b] computes the meaning of a regular repeat statement primarily utilising the supply components of the DDA. As we shall see, this mechanism can be easily transformed into a targeted repetition that computes the meaning of a targeted repeat statement with optimal resource usage.

Before presenting these, however, we show that the shared memory model architecture entails request based space-time controlled computation strategies as well. The following execution model requires very limited memory space. Nonetheless, it does not transform into an optimal targeted repetition.

Definition 5.3.2 (Request based space-time controlled repetition in shared memory)**.** Consider the repeat statement **`repeat`** `p:P` **`along`** `D` **`from`** `V` **`in`** `e` with finite span space-time DDA `D=<P,B,req,sup>` where the element type of `V` is `E`. Let `tspan:Nat` be the maximum number of time-steps spanned by a request.

The *request based space-time algorithm to compute the repeat statement* will gradually fill in a fresh array `V'` using its previously computed values directly.

- The array `V'` has index type `P` and element type `E`, and it contains all values that have been computed up to a certain time-step. The array is in the shared memory, s.t., `V'[p]` is dealt with by processor `space(p)` for a point `p:P`. Initially `V'` is a copy of `V` together with its guard.

For every time-step `t`, starting at `zero`, the computation repeats the following steps in parallel for all `s:Space`:

- Let `p=P(s,t)`.[1]

[1]Recall that if the space embedding projection does not cover all space range of `Space` at a given time-step, e.g., no DDA point is projected onto `s:Space` at time-step `t`, then the implicit constructor guard `CG` will not allow `P(s,t)` to be considered, and hence the point `p=P(s,t)` will not exist. Thus all computation steps at processor `s` involving `p`, and only those, will not be considered either at time-step `t`. This observation remains valid for all subsequent execution models of Section 5.3.

- If `ig(V',p)` does not hold, and for all `b:B`, `ig(V',rp(p,b))` holds whenever `rg(p,b)`, then compute `e`, and insert its value into `V'[p]`.
- Synchronise with the other processes.

These steps are to be repeated until no processor has computed any new value for `tspan` consecutive time-steps (nothing more can be computed) and for all `p:P` where `ig(V,p)` holds it is the case that `time(p)<t-tspan` (nothing more can be computed from the initial values either).

The result of the computation is the array `V'`. □

Proposition 5.3.3. Given the appropriate conditions, the request based space-time controlled computation of the repeat statement **`repeat`** `p:P` **`along`** `D` **`from`** `V` **`in`** `e` in shared memory computes the `e`-derived extension of `V` along `D`. □

Proof: Note that the computation covers all points in the region `P(s,t)` for all relevant space-time coordinates `(s,t)`. At every time-step, synchronised across all processors, `1`-reachable points are computed in parallel from the defined values of `V'`. The expression `e` at a point `p` is only computed when `V'[p]` should be assigned such a value.

The computation ends when the time-step has advanced enough to make sure that nothing more can be computed from the initial and the newly defined values of `V'`. Thus when the computation ends, `V'` contains the `e`-derived extension of `V` along `D`. □

The request based space-time control repetition gives us full control over the space and time resources via the space and time projections. Utilising the request components directly, there is no need for internal memory locations to store intermediate results, like we needed in the dependency-driven computations. Hence the storage space needed is only of the size of the computed values of `V'` which is the desired output as well.

Definition 5.3.4 (Request based space-time controlled targeted repetition in shared memory)**.** Given a targeted repeat statement **`repeat`** `p:P` **`along`** `D` **`from`** `V` **`for`** `TP` **`in`** `e` where `D=<P,B,req,sup>` is a finite span space-time DDA and the element type of `V` is `E`. Let `tmax:Nat` be the maximum time coordinate defined from the target points:

$$\texttt{tmax} = \max_{\{\texttt{p}|\texttt{TP(p)}=\texttt{true}\}}(\texttt{time(p)})$$

Then the *request based targeted space-time algorithm to compute the targeted repeat statement* will evolve exactly in the same manner as the space-time

controlled repetition in shared memory based on requests, but the computation may end also as soon as the current time-step hits upon `tmax+1`. After this all non-target values of `V'` are discarded, and the array `V'` is the result of the computation. □

Proposition 5.3.5. Given the appropriate conditions, the request based space-time controlled targeted computation of the targeted repeat statement **`repeat`** `p:P` **`along`** `D` **`from`** `V` **`for`** `TP` **`in`** `e` in shared memory computes the `TP`-targeted `e`-derived extension of `V` along `D`.

Proof: The computation evolves aiming at computing the `e`-derived extension of `V` along `D`. It will end either when it has advanced enough to make sure that nothing more can be computed from the initial and the newly defined values of `V'`, or after the very last target points, at time-step `tmax`, have been visited. In the former case, even if there are still undefined target values, the condition assures that they cannot be computed. In the latter case, nothing more relevant can be computed (see Proposition 5.1.9). Hence the computation ends, clearing all irrelevant values of `V'`. □

We are again in full control over space and time resources, however, the irrelevant values of `V'` can only be discarded safely in the last time-step, leading to a non-optimal storage space utilisation.

In the following we present the space-time controlled repetition as proposed in [Burrows and Haveraaen, 2009b] for a regular repeat statement.

Definition 5.3.6 (Supply based space-time controlled repetition in shared memory)**.** Consider the repeat statement **`repeat`** `p:P` **`along`** `D` **`from`** `V` **`in`** `e` with finite span space-time DDA `D=<P,B,req,sup>` where the element type of `V` is `E`. Let `Time = Nat` be used as the time type, and `tspan:Nat` be the maximum number of time-steps spanned by a request, and let `T` be a type defined as: `T = {0,1,...,tspan-1}`.

The *supply based space-time algorithm to compute the repeat statement* will gradually fill in a fresh array `V'`, using a support array `M` which contains intermediate information about the computation. Both will be updated at every step of the computation.

- The array `V'` has index type `P` and element type `E`. It contains all values that have been computed so far. The array is distributed, s.t., `V'[p]` is at processor `space(p)` for a point `p:P`. Initially `V'` is a copy of `V` together with its guard.
- The array `M` has index type `Space*T` and as elements arrays with index type `B` and element type `E`. `M` is used to store, as they become available, all the element values a point `P(s,t)`, for `s:Space` and `t:Time`,

requests from branches `b:B`, until the point has been computed. Then the array is distributed, s.t., `M[s,t%tspan]` is at processor `s`. Initially `M` is an array of empty sub-arrays, i.e., the guard `ig(M[s,t%tspan],b)` does not hold for any `s:Space`, `t:Time` and `b:B`.

We need again a modified version `e'` of `e`, where all occurrences of the pattern `V[rp(p,b)]` have been replaced by `M[space(p),time(p)%tspan][b]`. For every time-step `t`, starting at `zero`, the computation repeats the following steps in parallel for all `s:Space`:

- Let `p=P(s,t)`.
- If `ig(V',p)` does not hold, `M[s,t%tspan]` is not empty, and for all `b:B`, `ig(M[s,t%tspan],b)` holds whenever `rg(p,b)`,
  then compute `e'`, and insert its value into `V'[p]`.
- Empty `M[s,t%tspan]`.
- Synchronise with the other processes.
- If `ig(V',p)` holds,
  then for all relevant branches `d:B` do:
  - Let `q=sp(p,d)`
  - Let `M[space(q),time(q)%tspan][sb(p,d)] = V'[p]`.
- Synchronise with the other processes.

These steps are to be repeated until for all `p:P` where `ig(V,p)` holds it is the case that `time(p)<t` (all initial values have been visited), and all sub-arrays in `M` are empty (nothing more can be computed).

After this the intermediate `M` is cleared, and the array `V'` is the result of the computation. □

With the given assumptions, this algorithm will, time-step by time-step, in parallel compute exactly the `e`-derived extension of `V` along `D`.

Proposition 5.3.7. Given the appropriate conditions, the supply based space-time controlled computation of the repeat statement **`repeat`** `p:P` **`along`** `D` **`from`** `V` **`in`** `e` in shared memory computes the `e`-derived extension of `V` along `D`. □

Proof: The computation covers all points in the region `P(s,t)` for all relevant space-time coordinates `(s,t)`. At every time-step, synchronised across all processors, 1-reachable points are computed in parallel from the defined values of `V'`, indirectly, via the appropiate previously filled-in memory cells (see Proposition 5.2.2). An expression `e'` at a point `p` is only computed when `V'[p]` should be assigned such a value. By the axioms of the DDA, it is clear that the new value `e'` computed using `M[s,t%tspan]` is the value of that point in the `e`-derived extension.

In the meantime, when the related memory cells are emptied, all processors need to be synchronised (within the same time-step), to avoid race conditions, which otherwise may cause newly propagated values in `M` to be deleted, and hence they could not be utilised in a consequent time-step.

Thus when the computation ends, `V'` contains the `e`-derived extension of `V` along `D`. □

Note that the use of memory locations per processor node to store intermediate results is compatible with the implicit supply direction embedding projections into hardware paths sketched in Section 4.3.2 for STAs with `tspan>1`.

We see that the supply based space-time controlled repetition requires more storage space (the intermediate memory locations `M`) than the request based one, but it still gives us full control of the space-time resources. If we denote by `S` the size of the finite sort `Space`, then at most $\texttt{S} * \texttt{tspan}$ nodes will be active at any time during the computation. Thus by controlling the space-time projection we can control both time (number of time-steps needed for the computation) and memory usage. This entails full control over resource usage.

On the other hand, the intermediate memory locations `M` can be fully utilised when the repeat statement is targeted, leading to a targeted space-time controlled repetition with optimal storage space usage.

Definition 5.3.8 (Supply based space-time controlled targeted repetition in shared memory)**.** Consider the targeted repeat statement **`repeat`** `p:P` **`along`** `D` **`from`** `V` **`for`** `TP` **`in`** `e` with `D=<P,B,req,sup>` a finite span space-time DDA where the element type of `V` is `E`. Let `tspan:Nat` be the maximum number of time-steps spanned by a request, and let `T` be a type defined as: `T = {0,1,...,tspan-1}`. Further let `tmax:Nat` be the maximum time coordinate defined from the target points:

$$\texttt{tmax} = \texttt{max}_{\{\texttt{p}|\texttt{TP(p)}=\texttt{true}\}}\texttt{(time(p))}$$

The *supply based targeted space-time algorithm to compute the targeted repeat statement* will gradually fill in a fresh array `V'`, using a support array `M`

which contains intermediate information about the computation. Both will be updated at every step of the computation.

- The array `V'` has index type `P` and element type `E`. It contains all target values computed so far, and all initial and just computed values that have not yet been propageted in the DDA. The non-target values of `V'` will be gradually discarded as the computation proceeds. The array is distributed, s.t., `V'[p]` is dealt with at processor `space(p)` for a point `p:P`. Initially `V'` is a copy of `V` together with its guard.
- The array `M` has index type `Space*T` and as elements arrays with index type `B` and element type `E`. `M` is used to store, as they become available, all the element values a point `P(s,t)`, for `s:Space` and `t:Time`, requests from branches `b:B`, until the point has been computed. Then the array is distributed, s.t., `M[s,t%tspan]` is at processor `s`. Initially `M` is an array of empty sub-arrays, i.e., the guard `ig(M[s,t%tspan],b)` does not hold for any `s:Space`, `t:Time` and `b:B`.

We also need the modified version `e'` of `e`, where all occurrences of the pattern `V[rp(p,b)]` have been replaced by `M[space(p),time(p)%tspan][b]`. For every time-step `t`, starting at `zero`, the computation repeats the following steps in parallel for all `s:Space`:

- Let `p=P(s,t)`.
- If `ig(V',p)` does not hold, `M[s,t%tspan]` is not empty, and for all `b:B`, `ig(M[s,t%tspan],b)` holds whenever `rg(p,b)`,
  then compute `e'`, and insert its value into `V'[p]`.
- Empty `M[s,t%tspan]`.
- Synchronise with the other processes.
- If `ig(V',p)` holds then:
  - For all relevant branches `d:B` do:
    - Let `q=sp(p,d)`.
    - Let `M[space(q),time(q)%tspan][sb(p,d)] = V'[p]`.
  - If `TP(p)` does not hold, empty `V'[p]`.
- Synchronise with the other processes.

These steps are to be repeated until `t=tmax+1` (all target points have been visited) or for all `p:P` where `ig(V,p)` holds it is the case that `time(p)<t` (all initial values have been visited), and all sub-arrays in `M` are empty (nothing more can be computed).

After this the intermediate `M` is cleared, all remaining irrelevant values of `V'` are emptied, leaving `V'` as the result of the computation. □

Proposition 5.3.9. Given the appropriate conditions, the supply based space-time controlled targeted computation of the targeted repeat statement **`repeat`** `p:P` **`along`** `D` **`from`** `V` **`for`** `TP` **`in`** `e` in shared memory computes the `TP`-targeted `e`-derived extension of `V` along `D`. □

Proof: The computation proceeds as the previous algorithm, with all relevant synchronization steps in place to avoid harmful race-conditions, but here computes the `TP`-targeted `e`-derived extension of `V` along `D` by computing `1`-reachable points in parallel at every time-step, and if such a point is not a target point, its value is discarded after the value has been propagated in the DDA. Note that points are considered again `1`-reachable from the defined points of `V'`, indirectly, as soon as all relevant memory locations in `M` at the point have been assigned a value. Likewise, an initial value, unless it is also a target value for some reason, is gradually discarded from `V'` after it has been propagated in the DDA. If the computation ends at hitting the time-step `tmax+1`, then all remaining non-target values of `V'` will be emptied (the leftover of irrelevant initial values). The computation ends with as much target values computed as possible given the information we have. □

We can see that the supply based targeted space-time controlled computation has optimal storage usage compared to the request based targeted space-time controlled computation. There we need to keep all intermediate computed values of `V'`, here we only need to manage the array `M` of size at most `S*tspan` to keep intermediate information.

The space-time controlled computational styles in shared memory can also be adapted to a single processor, in a sequentialised form. In that case, a loop for each spatial index must be run between every synchronisation step. The shared memory versions given here are nonetheless more appropriate for the current many-core architectures.

Note that we can control locality by selecting space-projections that let requests and supplies communicate with neighbouring processors, if possible (see Section 4.4 where the flexibility of DDA-projections was discussed).

5.3.2 Message Passing Execution Model

The MPI (Message Passing Interface) standard is the most popular message passing specification supporting parallel programming. The underlying hardware is assumed to be a collection of processors interconnected via a network such that each processor has its own local memory with the restriction that processors can have access only to their own local memory. The only way processors can communicate with each other is via message passing, in which a processor gives indirect access to its local data values by physically sending it to the other processor which in turn is waiting for the message.

In the message passing programming model, programs are organised around concurrent processes that are assumed to be able to communicate with each other at any time-step, since the underlying interconnection network means that there is an implicit channel between every pair of processes. Note that, in this respect, a corresponding distributed memory model STA would be identical with the shared memory model STA, the only difference being between the associated communication methods: in shared memory there is direct access, in distributed memory the access is indirect, via message passing.

Every concurrent process executes the same program, and each process has a unique ID, hence different processes may carry out different tasks. IDs are also used when specifying messages. The processes can be implicitly synchronised via message passing, e.g., a process cannot receive data unless the data was sent from another process, which in turn provides information about the state of the sending process, and so on.

The abstractions available in the DDAs allows us to explicitly define send and receive message pairs between the relevant processes, by utilising the information available in the isomorphic request and supply components. The computational DDA is embedded into this space-time by distributing, e.g., nearest neighbour points onto a single process, so that each process deals with several DDA points at a given time-step. Each process then can execute one of the space-time controlled computational strategies in shared memory with its own local data, in a sequentialised form, as suggested earlier. Whenever a data value is to be propagated to a DDA point embedded on a different process, a corresponding sending message is instantiated in the sending process. Likewise, the value at a DDA point can only be computed if all relevant dependencies have been defined. If one of these sits on a DDA point embedded onto a different process, then a corresponding receive message is instantiated in the receiving process, thus completing the desired communication between the two processes.

Since a similar computation mechanism was already used when building the Sapphire prototype compiler which generated MPI code from DDA-specifications [Søreide, 1998], we will omit a detailed presentation of the message passing execution model instantiated for the proposed repeat statements' syntax and semantics.

5.3.3 The CUDA Execution Model

Our hardware is now comprised by the CUDA kernel space-time DDA, $\texttt{DCUST}_{\texttt{B,T}}$ as presented in Section 4.3.5 for a single-GPU system, where the kernel is configured for `B` number of blocks and `T` number of threads per block, i.e., `B*T` number of threads will execute the kernel. The DDA of the repeat statement is assumed to be embedded into this space-time via designated space and time projections, where space is comprised by $\texttt{CUB}_{\texttt{B,T}}$ `= block Nat * thread Nat` and $\texttt{CUST}_{\texttt{B,T}}$ `= space` $\texttt{CUB}_{\texttt{B,T}}$ `* time Nat`.

Example 5.3.10 (CUDA Embedding Projection)**.** Consider the CUDA kernel space-time DDA $\texttt{DCUST}_{\texttt{B,T}}$ and let `D=<P,B,req,sup>` be a DDA. Then a *CUDA embedding projection of* `D` *into* $\texttt{DCUST}_{\texttt{B,T}}$ is given by `EP:P`$\rightarrow$$\texttt{CUST}_{\texttt{B,T}}$ together with projections `space:P`$\rightarrow$$\texttt{CUB}_{\texttt{B,T}}$ and `time:P`$\rightarrow$`Nat` and constructor `P:`$\texttt{CUB}_{\texttt{B,T}}$`,Nat`$\rightarrow$`P` such that:

$$\texttt{EP(p)=}\texttt{CUST}_{\texttt{B,T}}\texttt{(space(p),time(p))}$$

A finite span CUDA embedding projection is one in which `D` forms a finite span STA with the projections `space:P`$\rightarrow$$\texttt{CUB}_{\texttt{B,T}}$ and `time:P`$\rightarrow$`Nat`, and constructor `P:`$\texttt{CUB}_{\texttt{B,T}}$`,Nat`$\rightarrow$`P`. □

In general, the CUDA execution of a computation is based on repeated kernel invocations, unless the particular problem we are dealing with is embarrassingly parallel (no dependency exists between the parallel parts), or it is small enough to be executed by a single block of threads. Since the execution order of the thread-blocks, as handled by the CUDA runtime system, is completely random and asynchronous, threads of different blocks can only communicate asynchronously via the GPU memory upon a new kernel invocation. Therefore kernels should only run as long as intra-block communication is guaranteed, otherwise they need to be ended and reinvoked. The exact time-steps for invoking and ending a kernel is an inherent property of the particular embedding in question, and does not depend on the actual computations performed by the threads. This property is made concrete next.

Definition 5.3.11 (CUDA Kernel Scheduler)**.** Let D=<P,B,req,sup> be a DDA with a finite span CUDA embedding projection into $\mathtt{DCUST}_{B,T}$, and let next:Nat→Nat be the increment operator on time. Further, let tmax be the maximum time-coordinate defined by the time projection of the embedding, i.e., time(p)<=tmax for all p:P.

The kernel scheduler will use a support array K, the *kernel schedule*, indexed by Nat and element type {0,1,...,tmax}, containing all time-steps when a kernel has to be ended, built as follows.

Let i be some index-variable of type Nat, such that initially i=0, and let temp be a boolean variable. We start at time-step t=0, and the following steps are to be repeated while t<tmax:

- Let temp=true.
- If for some p:P with time(p)=t and for some relevant b:B

  block(space(p))!=block(space(sp(p,b)))

  then set temp=false.
- If temp=false then set K[i]=t and increase the index i++.
- Increment time-step: t=next(t).

In the end the last time-step tmax is also added to K, i.e., K[i] = tmax, and the length of K thus will be i+1.

The first kernel invocation will be at time-step t=0. If the length of K is tmax+1, the kernel has to be ended at every time-step and reinvoked at the next time-step. When the length of K is 1, i.e., when K[0]=tmax, the kernel will be ended (invoked) a single time.

When the kernel is ended at some time-step K[j], it will be reinvoked at the next time-step, i.e., next(K[j]), unless K[j]=tmax.

Then the *CUDA kernel scheduler* is executed by the host as follows:

- call the kernel the first time: kernel«B,T»(<arg-list>,0,K[0])
- If K[0]<tmax then
  - Let i=1.
  - While K[i]<=tmax do:
    - kernel call: kernel«B,T»(<arg-list>,next(K[i-1]),K[i])

where «B,T» specifies the dimension of the kernel; <arg-list> stands for some actual argument list of the kernel and is specified later; and the last two arguments of the kernel call specify the time-steps for invoking and ending the kernel, respectively. □

Obviously, the length of K will vary depending on the CUDA embedding projection of D into $\mathtt{DCUST}_{B,T}$, and ultimately the parameters B and T as well. Note that for a fixed pair of parameters B and T several CUDA embedding projections could be defined as long as they comply with the consistency requirements of such projections (see Section 4.4).

A more elaborate kernel scheduler instead of precomputing the entire array K would compute every next kernel schedule entry on-the-fly right after a kernel has just been invoked. This could speed up the accumulated total execution time, since CUDA allows concurrent execution between host and GPU device, i.e., control is returned to the host before the device has completed the requested task. On the other hand, building the kernel schedule array K can always be considered fixed and shipped with the CUDA embedding itself, since it is independent of the computation associated with the DDA or any input data.

CUDA syntax is an extension of the C programming language. Hence, guards of partial indexing operations associated with arrays can be implemented as arrays of boolean element type and exactly of the same dimension as the arrays they guard. If an array figures as an argument of the kernel, we assume that its guard is also passed on as argument.

In CUDA the host manages the memory spaces visible to kernels (e.g. its arguments) through calls to the CUDA run-time. This includes device memory allocation and deallocation as well as data transfer between host and device memory. We will, therefore, skip presenting the fine details of this memory management, they are being specific to CUDA and straightforward in the context of a given kernel.

We will present various CUDA execution models, again request and supply based ones, for both regular and targeted repeat statements. They will primarily differ in the way the kernels are built up, their argument lists, and the memory locations they utilise. Each of the execution models is based on the kernel scheduler presented in Definition 5.3.11 with subtle modifications as allowed by the particular model.

Definition 5.3.12 (Request based space-time controlled repetition in CUDA)**.** Given a repeat statement **repeat** p:P **along** D **from** V **in** e with a finite span CUDA embedding projection of D into $\mathtt{DCUST}_{B,T}$, and where the element type of V is E. Let tspan:Nat be the maximum number of time-steps

spanned by a request and let K be the CUDA embedding's related kernel schedule.

In the *request based space-time algorithm to compute the repeat statement in CUDA* the kernel scheduler will systematically invoke CUDA kernels of dimension B*T, one at a time, at the time-steps fetched from the kernel schedule K (see Definition 5.3.11). The threads executing these kernels will gradually fill in a fresh array V' stored in the GPU memory across all kernel invocations together with its guard, updated at every time-step.

- The array V' has index type P and element type E. It contains all values that have been computed so far. The array is distributed in the GPU memory such that V'[p] is dealt with by thread space(p) for a point p:P. Initially V' is a device copy of V together with its guard.

The kernel scheduler will invoke the kernels with 4 arguments: the array V' and its guard – these will be updated by the threads of the kernel and returned to the host; and t_start and t_end (fetched from K) will mark the time-steps for starting and terminating the kernel, respectively.

Then all threads of a kernel across the space $\texttt{s:CUB}_{\texttt{B,T}}$ will execute in parallel[2] the following steps starting at time-step t=t_start, while t<=t_end:

- Let p=P(s,t).
- If ig(V',p) does not hold, and for all b:B, ig(V',rp(p,b)) holds whenever rg(p,b) holds, then compute e and insert its value to V'[p] and set ig(V',p)=true.
- Synchronize threads.
- t=next(t).

Each invoked kernel executes these steps within the given time steps as appointed by the kernel scheduler.

After this, the result of the computation will be the entire array V'. □

The kernel scheduler, suitably modified, may suspend the kernel invocations prior to reaching the kernel schedule entry tmax, by checking whether there has been computed any new value in V' in the last tspan consecutive time-steps and for all p:P where ig(V',p) initially held it is the case that time(p)<K[i]-tspan for some kernel schedule entry K[i] (nothing more can be computed from the initial values either).

[2]Since the actual thread-blocks are executed by the CUDA run-time system in a random order, the execution of all threads across the space $\texttt{s:CUB}_{\texttt{B,T}}$ here is *virtually* parallel, as promoted by the CUDA API.

Proposition 5.3.13. Given the appropriate conditions, the request based space-time controlled algorithm of the repeat statement **repeat** p:P **along** D **from** V **in** e in CUDA computes the e-derived extension of V along D. □

Proof: The computation will cover all DDA points in the region P(s,t) for all relevant space-time coordinates (s,t), by letting each thread with global ID s deal with the DDA point P(s,t) at time-step t. In each kernel, at every time-step, 1-reachable points are computed in parallel from the defined values of V'. The explicit synchronization works across all the threads of a block, and for all blocks individually (as allowed by the CUDA API). This and the way the kernel scheduler is built ensure race-condition free execution between threads of the same and of different blocks. So the test whether all relevant dependencies of V'[p] have been computed is guaranteed to be correct, i.e., if one of the dependencies is across block boundaries and has not been computed by the time-step of the test, then it will not be computed at all. Then the expression e at a point p is only computed when V'[p] should be assigned such a value.

The kernel scheduler will stop invoking new kernels if either it has reached tmax or, if we have the more elaborate kernel scheduler, nothing new is computed by the kernels from the initial and any newly defined values of V'.

Thus, when the computation ends, V' contains the e-derived extension of V along D. □

We have control over execution time and storage space utilisation, however one should bear in mind that this execution model requires continuous GPU memory accesses across all threads which is considered of high-latency, i.e., expensive compared to low-latency, fast shared memory accesses of threads within a block. [NVIDIA, 2010]

Definition 5.3.14 (Request based space-time controlled targeted repetition in CUDA)**.** Given a targeted repeat statement **repeat** p:P **along** D **from** V **for** TP **in** e with a finite span CUDA embedding projection of D into $\mathtt{DCUST}_{B,T}$, and where the element type of V is E. Let tspan:Nat be the maximum number of time-steps spanned by a request and let K be the CUDA embedding's related kernel schedule. Further let tpmax:Nat be the maximum time coordinate defined from the target points:

$$\mathtt{tpmax} = \max_{\{p|\mathtt{TP}(p)=\mathtt{true}\}}(\mathtt{time}(p))$$

Then the *request based targeted space-time algorithm to compute the targeted repeat statement in CUDA* will proceed exactly as the request based regular space-time controlled repetition in CUDA, but the kernel scheduler may

stop invoking the kernels also as soon as `tpmax<K[i]` for some kernel schedule entry `K[i]`, i.e., when all target values have been computed. After this, only target values of `V'` will be considered as the result of the computation. □

Proposition 5.3.15. Given the appropriate conditions, the request based space-time controlled targeted algorithm of the targeted repeat statement **`repeat`** `p:P` **`along`** `D` **`from`** `V` **`for`** `TP` **`in`** `e` in CUDA will compute the `TP`-targeted `e`-derived extension of `V` along `D`. □

Proof: The computation aims at building the `e`-derived extension of `V` along `D` until it has advanced enough to make sure that nothing more can be computed or it has computed all target values of `V'`, in which case only these are kept as a result, hence exactly the `TP`-targeted `e`-derived extension of `V` along `D` is kept. □

Note that the request based space-time controlled computation cannot enhance again the memory management by casting away non-target intermediate values on the go. The following supply based space-time controlled repetition on the other hand will allow this. First it is presented for the regular repeat statement, and then it is formalised for the targeted repeat statement.

Definition 5.3.16 (Supply based space-time controlled repetition in CUDA)**.** Given a repeat statement **`repeat`** `p:P` **`along`** `D` **`from`** `V` **`in`** `e` with a finite span CUDA embedding projection of `D` into $\texttt{DCUST}_{\texttt{B,T}}$, and where the element type of `V` is `E`. Let `tspan:Nat` be the maximum number of time-steps spanned by a request, and let `TSPAN` be a type defined as: `TSPAN = {0,1,...,tspan-1}`. Further let `K` be the CUDA embedding's related kernel schedule.

In the *supply based space-time algorithm to compute the repeat statement in CUDA* the kernel scheduler will systematically invoke CUDA kernels of dimension `B*T`, one at a time, at the time-steps fetched from the kernel schedule `K` as discussed in Definition 5.3.11. The threads executing these kernels will gradually fill in a fresh array `V'` stored in the GPU memory across all kernel invocations together with its guard, updated at every time-step. The algorithm uses two support arrays `INM` and `OUTM`, stored in the GPU memory together with their guards, which contain intermediate information about the computation. Threads will access global GPU memory content via global thread IDs, i.e., via the type $\texttt{CUB}_{\texttt{B,T}}$.

- The array `V'` has index type `P` and element type `E`. It contains all values that have been computed so far. The array is distributed in the GPU

memory, such that `V'[p]` is dealt with by thread `space(p)` for a point `p:P`. Initially `V'` is a device copy of `V` together with its guard.

- The arrays `INM` and `OUTM` have index type $\texttt{CUB}_{\texttt{B,T}}$`*TSPAN*B` and element type `E`. `INM` will provide additional information for each kernel upon its start; initially all values of `INM` are undefined, i.e., its guard never holds.

  `OUTM` is used to store information about the computation from the moment the kernel terminates until it is reinvoked by the host. At the beginning of each kernel, all values of `OUTM` are undefined, i.e., its guard never holds.

  Both `INM` and `OUTM` are distributed in the GPU memory, such that the sub-arrays `INM[s]` and `OUTM[s]` are dealt with by thread `thread(s)` in the block `block(s)` for $\texttt{s:CUB}_{\texttt{B,T}}$.

In addition, each block of threads, executing the kernel, will use a support array stored in the shared memory of each thread-block. These arrays contain intermediate information about the computation executed by the thread-blocks within one kernel, and will be updated at every time-step. Note that upon the termination of a kernel, all shared memory content is cast away in the CUDA Programming Model. Threads will access local shared memory content via local thread IDs, i.e., via the type `TT={0,...T-1}`:

- `SHM` is an array with index type `TT*TSPAN*B` and element type `E`. All arrays `SHM` across all blocks will be used to store, as they become available, all the element values a point `P(s,t)`, for $\texttt{s:CUB}_{\texttt{B,T}}$ and `t:Nat`, requests from branches `b:B`, until the point has been computed. In total, there will be `B` number of such arrays, one for each block, and they will be distributed such that `SHM[thread(s)][t%tspan]` is at thread `thread(s)` within block `block(s)`. At the beginning of each kernel, the whole sub-array `SHM[thread(s)]` of block `block(s)` is a copy of the sub-array `INM[s]` together with its guard from the global GPU memory. At the end of the kernel, the newly computed content of the sub-array `SHM[thread(s)]` of block `block(s)` is copied to `OUTM[s]`, in order to save it for the next kernel.

And finally, each thread will manage a private variable `TMP` of type `E`, together with a guard `igTMP`, which is used to retrieve information from `V'` in the GPU memory in order to propagate it locally into the fast shared memory. Whenever `TMP` is assigned a value, `igTMP` holds, and whenever `TMP`

is emptied, `igTMP` does not hold. At the start of the kernel `igTMP` does not hold.

We need again a modified version `e'` of `e`, where all occurrences of the pattern `V[rp(p,b)]` have been replaced by the relevant shared memory locations of the relevant blocks: `SHM[thread(space(p)),time(p)%tspan][b]`.

The kernel scheduler will invoke the kernels with 8 arguments: the arrays `V'`, `INM` and `OUTM` together with their guards, of which `V'` and `OUTM` will be updated by the threads of the kernel; and `t_start` and `t_end` (fetched from `K`) will mark the time-steps for starting and terminating the kernel, respectively.

Between two kernel invocations, the host updates the content of `INM` with the received content of `OUTM`, and clears the content of `OUTM`, i.e., its guard will never hold at the beginning of the kernels.

Then all threads of a kernel across the space $\texttt{s:CUB}_{\texttt{B,T}}$ will execute in parallel the following steps starting at time-step `t=t_start`:

- Copy the content of the sub-array `INM[s]` from GPU memory into `SHM[thread(s)]`, together with its guard.
- Synchronize threads.
- The following steps are to be repeated while `t<=t_end`:
  - Let `p=P(s,t)`.
  - If `ig(V',p)` does not hold and for all `b:B` `ig(SHM[thread(s),t%tspan],b)` holds whenever `rg(p,b)` then compute `e'` and insert its value to `V'[p]`
  - Empty `SHM[thread(s),t%tspan][b]` for all `b`.
  - Synchronize threads.
  - If `ig(V',p)` holds then read its value: `TMP=V'[p]`
  - If `igTMP` holds then for all relevant `d:B` do:
    - Let `q=sp(p,d)`.
    - If `t=t_end` then propagate directly to GPU memory, i.e., `OUTM[space(q),time(q)%tspan][sb(p,d)]=TMP`
    - If `t<t_end` then propagate in the shared memory, i.e., `SHM[thread(space(q)),time(q)%tspan][sb(p,d)]=TMP`
    - Empty `TMP`.
  - Synchronize threads.
  - Let `t=next(t)`.

- Let OUTM[s,ts][b]=SHM[thread(s),ts][b] only for those ts:TSPAN and b:B where ig(SHM[thread(s),ts],b) holds.
- Synchronize threads.

Each kernel executes these steps within the given time steps as appointed by the kernel scheduler.

After this, the result of the computation will be the entire array V'. □

The kernel scheduler, suitably modified, may suspend the kernel invocations prior to reaching the kernel schedule entry tmax, by checking if all initial values of V have been visited, e.g., for some kernel schedule entry K[i] we have that time(p)<K[i] for all initial points p:P, and the current content of the returned array OUTM is such that its guard never holds, i.e., nothing more can be computed.

Proposition 5.3.17. Given the appropriate conditions, the supply based space-time controlled algorithm of the repeat statement **repeat** p:P **along** D **from** V **in** e in CUDA will compute the e-derived extension of V along D. □

Proof: The computation covers all points in the region P(s,t) for all relevant space-time coordinates (s,t). In each kernel, at every time-step 1-reachable points are computed in parallel, by all threads, from the defined values of V', indirectly, via the appropriate previously filled-in shared memory cells in SHM (see Proposition 5.2.2). To see that the shared memory cells are appropriately filled in, remember that the kernel scheduler ensures that no inter-block communication is required while t<t_end, and that within each thread-block, all the threads of the block synchronize with each other avoiding harmful race-conditions. Only t=t_end marks the time-step of the need for inter-block communication. Then all computed results of this step are written directly to the corresponding memory cells of OUTM, since threads otherwise cannot access the shared memory of a different block. At the end of the kernel, if there are any intermediate results left in the local shared memories, SHM-s, these will be copied to the relevant locations of OUTM as well. The separate use of INM and OUTM ensures that threads across block boundaries do not overwrite the input (INM) of a thread from a different block. Upon each new kernel invocation, the host takes care of updating the content of INM with that of OUTM, so that intermediate results would not get lost.

The expression e' at a point p is only computed when V'[p] should be assigned such a value. The DDA axioms ensure that the value e' computed

using the relevant locations of the sub-array `SHM[thread(s),t%tspan]`, in block `block(s)`, is the value of that point in the `e`-derived extension.

Thus when the computation ends, `V'` contains the `e`-derived extension of `V` along `D`. □

The execution time and memory space usage of the entire computation is under control via the CUDA embedding space and time projections. However, the storage space allocated on the GPU memory for storing `V'`, `INM` and `OUTM` and their guards will be fixed (preallocated by the host) throughout the entire computation. Even when `OUTM` is "empty", the whole array is allocated on the device for it. Likewise, if a value of `V'` is "not defined", a memory location of size needed to store an element of type `E` will be allocated, and cannot be deallocated in the kernel. Nonetheless, these restrictions originate from the CUDA API.

Comparing the request based space-time computation of the regular repeat statement with the supply based one, we see that here we heavily utilise shared memory accesses instead of continuous GPU memory accesses. This, on the other hand, requires additional memory management and copy of data to the relevant locations. It may well be the case that STAs with small `tspan` and small number of branch indices may perform better under the supply based execution model whereas STAs with large `tspan` and large range of (request) branch indices may perform better under the request based computation strategy. However, this observation is only an intuition, and is not underpinned by experiments.

Definition 5.3.18 (Supply based space-time controlled targeted repetition in CUDA)**.** Given a targeted repeat statement **`repeat`** `p:P` **`along`** `D` **`from`** `V` **`for`** `TP` **`in`** `e` with a finite span CUDA embedding projection of `D` into $\texttt{DCUST}_{\texttt{B,T}}$, and where the element type of `V` is `E`. Let `tspan:Nat` be the maximum number of time-steps spanned by a request, and let `TSPAN` be a type defined as: `TSPAN = {0,1,...,tspan-1}`. Further let `K` be the CUDA embedding's related kernel schedule.

In the *supply based targeted space-time algorithm to compute the targeted repeat statement in CUDA* the kernel scheduler will systematically invoke CUDA kernels of dimension `B*T`, one at a time, at the time-steps fetched from the kernel shedule `K`. The threads executing these kernels will gradually fill in a fresh array `V'` stored in the GPU memory across all kernel invocations together with its guard, updated at every time-step, containing only the target values of `V` and is built as follows:

- Let `N` be the number of target points, and `tind:P`$\rightarrow$`{0,1,...,N-1}` an injective partial function such that `tind(p)` is defined whenever

`TP(p)` holds. The array `V'` has index type `{0,1,...,N-1}` and element type `E`, and it contains all target values computed so far. The array is distributed in the GPU memory such that `V'[tind(p)]` is dealt with by thread `space(p)` for any target point `p:P`, and the target value `V[p]` will be given by `V'[tind(p)]`. Initially the guard of `V'` never holds.

Initial values will be stored in the GPU memory across all kernels in the form of an array of size exactly the number of input points, and is built as follows:

- Let `N'` be the number of input points, i.e., where `ig(V,p)` initially holds. And consider `iind:P`$\rightarrow$`{0,1,...,N'-1}` an injective partial function such that `iind(p)` is defined whenever `ig(V,p)` holds. The array `I` with index type `{0,1,...,N'-1}` and element type `E` contains the initial values of `V` such that `I[iind(p)]=V[p]` for all initial points `p:P`. Note that the guard of `I` will always hold, hence there is no need to send it in the argument list of the kernel.

The algorithm uses two support arrays `INM` and `OUTM`, in the same fashion as in the previous algorithm, stored in the GPU memory together with their guards. Both have index type $\texttt{CUB}_{\texttt{B,T}}$`*TSPAN*B` and element type `E`. Initially all values of `INM` are undefined, i.e., its guard never holds.

In addition, each block of threads, executing the kernel, will use a support array `SHM` with index type `TT*TSPAN*B` and element type `E` stored in the block's shared memory. This will be managed and updated in the same manner as discussed in the previous algorithm.

Each thread will have a private variable `TMP` of type `E`, together with a guard `igTMP`.

We need again a modified version `e'` of `e`, where all occurrences of the pattern `V[rp(p,b)]` have been replaced by the relevant shared memory locations of the relevant blocks: `SHM[thread(space(p)),time(p)%tspan][b]`.

The kernel scheduler will invoke the kernels with 9 arguments: the arrays `V'`, `INM` and `OUTM` and their guards, of which `V'` and `OUTM` will be updated by the threads of the kernel and returned to the host; the initial array `I`; and `t_start` and `t_end` will mark the time-steps for starting and terminating the kernel, respectively.

Between two kernel invocations, again, the host updates the content of `INM` with the received content of `OUTM`, and clears the content of `OUTM`, i.e., its guard never holds at the beginning of the kernels.

Then all threads of a kernel across the space `s:`$\texttt{CUB}_{\texttt{B,T}}$ will execute in parallel the following steps starting at time-step `t=t_start`:

- Copy the content of the sub-array `INM[s]` into `SHM[thread(s)]` together with its guard.
- Synchronize threads.
- The following steps are to be repeated while `t<=t_end`:
  - Let `p=P(s,t)`.
  - If `p` is not an input point and for all `b:B` `ig(SHM[thread(s),t%tspan],b)` holds whenever `rg(p,b)` then compute `e'` and insert its value to `TMP`. If `TP(p)` holds, send this value as a result: `V'[tind(p)]=TMP`.
  - Empty `SHM[thread(s),t%tspan][b]` for all `b`.
  - Synchronize threads.
  - If `p` is input point then read its value: `TMP=I[iind(p)]`.
  - If `igTMP` holds then for all relevant `d:B` do:
    - Let `q=sp(p,d)`.
    - If `t=t_end` then propagate directly to GPU memory, i.e., `OUTM[space(q),time(q)%tspan][sb(p,d)]=TMP`
    - If `t<t_end` then propagate in the shared memory, i.e., `SHM[thread(space(q)),time(q)%tspan][sb(p,d)]=TMP`
    - Empty `TMP`.
  - Synchronize threads.
  - Let `t=next(t)`.
- Let `OUTM[s,ts][b]=SHM[thread(s),ts][b]` only for those `ts:TSPAN` and `b:B` where `ig(SHM[thread(s),ts],b)` holds.
- Synchronize threads.

Each kernel executes these steps within the given time steps as appointed by the kernel scheduler.

After this, the result of the computation will be the entire array `V'`. □

The kernel scheduler, suitably modified, may suspend the kernel invocations prior to reaching the kernel schedule entry `tmax` as soon as all target values have been computed, or when all initial values have been visited and the current content of the returned array `OUTM` is such that its guard never holds, i.e., nothing more can be computed.

Proposition 5.3.19. Given the appropriate conditions, the supply based space-time controlled targeted algorithm of the targeted repeat statement **repeat** p:P **along** D **from** V **for** TP **in** e in CUDA will compute the TP-targeted e-derived extension of V along D such that $\texttt{ext}^{\texttt{TP}}_{\texttt{D,e}}\texttt{(V)[p]=V'[ind(p)]}$ for all p:P where TP(p) holds. □

Proof: The computation proceeds as the regular supply based CUDA algorithm, but here intermediate results of the e-derived extension of V along D are only written in V' if they are target values. Otherwise all intermediate values of the extension, stored across the threads in TMP at a given time-step, are only propagated in the shared memory, or if in the last kernel time-step, in the GPU memory, in order to make them available as defined dependencies when needed in the subsequent computation. The algorithm ends, when as many target values as possible have been computed in the e-derived extension of V along D. □

With the introduction of reduced size target and initial arrays, we require significantly less memory space on the GPU memory, which may also lead to speed ups in the overall execution time. Such a scenario occurs when we have a modified kernel that regularly inspects the guard of V' to see whether all target values have been computed, in order to suspend all subsequent kernel invocations. (The host can only inspect the GPU memory content indirectly, by copying the relevant GPU memory content back into its own memory.)

These execution models can deal with any computational DDA embedded into the CUDA kernel space-time DDA. Depending on the DDA structural properties, however, one computational strategy may lead to better performance than another one. Certain, well-described properties of a DDA and its embedding can lead to simplified execution models. E.g., if the embedding is such that it has a minimal span, i.e., tspan=1, and all input points are embedded to the time-step marking the start of the computation and all target points to the time-step marking the end of the computation, then a much simpler execution model can be derived from the above one. Apparently, most DDAs we are presenting in this discourse are of tspan=1 and the computations built on them share the above mentioned property (e.g. see Chapter 4 and 6). Also remember that we can eliminate large tspans, as described in Example 4.3.5, in which case the execution models would require reduced size intermediate memory locations. Nonetheless, the execution models presented here do not impose any restriction on the DDAs, their CUDA embeddings or their computations; hence they are applicable in general.

5.3.4 FPGA Programming

A *Field Programmable Gate Array* (FPGA) is an integrated circuit designed to be programmed by the user after manufacturing, such that the final result is a concrete circuit on the FPGA chip. The main characteristic of an FPGA is that it is re-programmable, attributing renewed functionalities of the same chip.

Programming an FPGA is primarily about circuit design which is a "hardware" implementation rather than a "software" implementation of a given problem. [Kuon et al., 2008] In this sense, we cannot talk about a specific execution model associated to the semantics of the repeat statement, since, for instance, an FPGA does not have a program counter in the general sense, instead typically clocks all its gates at once. In the following, however, we attempt to describe how the meaning of a targeted repeat statement can be realised as a circuit which being fed with input data via its input wires provides the target values via its designated output wires.

The process of FPGA programming is usually assisted by proprietary software tools that take the specification of the circuit to be realised on the FPGA in some hardware description language, e.g., VHDL, Verilog, etc. Then the functionality of the design is verified by these tools, then synthesized into a gate-level netlist, and finally realised on the FPGA chip. Since the rigorous details of this workflow is out of the scope of this dissertation, we will only focus on the methodology describing how an associated circuit description can be generated for a targeted repeat statement.

We will focus our discussion on point-based computations to comply with the syntax and semantics of the repeat statement as discussed in the rest of this chapter. Note that branch-valued computations (see Sections 4.1.2 and 4.4.2), by their very nature, would induce reduced number of wires in the design process, and thus could lead to "lighter" circuits. However, it may not always be possible to design branch-valued DDA-based computations for a given problem.

Consider the following syntactically and type correct targeted repeat statement, where `D` is a finite span STA, `V` is an array of index type `P` and element type `E`, and where `Space` now is comprised by `Nat`:[3]

```
repeat p:P along D from V for TP in e
```

Then an associated circuit computing the `TP`-targeted `e`-derived extension of `V` along `D` can be built as follows.

[3]Note that FPGA chips have limited resource capacity, hence `Space` and `Time` will be obviously constrained by these physical limits.

First of all, note that DDAs naturally look like circuits, especially in a layout obtained via space-time projections. Hence, the task is to turn all branches into wires, and all computations at each DDA point into a corresponding computational node with I/O ports mapped to the relevant wires.

While the DDA request/supply directions determine the exact mapping of the I/O ports of a computational entity (corresponding to a DDA point) onto wires (or signals), DDA point projections can fully determine the placements of the computational entities on the chip. Carefully chosen placements are essential in FPGA programming as they can optimize FPGA die resource utilisation, leading to speed-ups and less energy consumption [Singh, 2000, 2011].

All branches (NB. not branch indices) of the DDA therefore will become the wires of the circuit represented by some canonical labelling, e.g., defined by `c:rg→Nat`, which assigns every request direction a unique natural number, starting from `0` and upward until some `N`$\in$ **N**, covering `N+1` request directions. The signals will carry values of type `E` presented in the form of a signal-array `V'` indexed by `{0,1,...,N}`. By the DDA axioms the values of `V'` are related to `V` as follows: `V[p]=V'[c(sp(p,b),sb(p,b))]` for all `p` and all its supply directions.

Then from the expression `e` several design elements (circuit entities) will be defined according to the sub-expressions from the sub-branches of the (possibly nested) `if` statements.

For instance, the following expression `e`, where `cond1(p)` and `cond2(p)` are some predicates:

```
if (cond1(p)) foo1(V[rp(p,0)], V[rp(p,1)])
else if (cond2(p)) foo2(V[rp(p,0)], V[rp(p,2)])
     else foo3(V[rp(p,0)])
```

entails three main design entities, one for each sub-expression. Each entity will have as inputs variables associated with the dependencies occurring in the sub-expression, and as outputs variables associated with each supply direction of `p`. E.g., from `foo1(V[rp(p,0)], V[rp(p,1)])` the entity `ENT1(in:a0,a1, out:b0,b1)` could be defined, given that `sg(p,0)` and `sg(p,1)` are the only supply directions from `p`. Then the behaviour of the entity is defined exactly as the sub-expression `foo1` in which every occurrence of `V[rp(p,0)]` is replaced by `a0`, of `V[rp(p,1)]` by `a1`, and so on, thus turning the entity into a generic computational pattern.

Since `p:P` in `e` can have any number of supply directions, unless the number of supply directions from a point is uniform across the whole DDA,

we need to create an entity for every sub-expression and every possible number of supply directions. In the above example, e.g., if the maximum number of supply directions from a point across the whole DDA is 2, then $2+1$ entities will be created for each sub-expression in case. The additional entity is motivated by points that do not have supply directions. Taking the first sub-expression, for instance, the following entities will be created, each with the same architecture, they will only differ in the number of outputs (remember that each output is assigned the same value):

```
ENT1-0(in:a0,a1, out:b0)
ENT1-1(in:a0,a1, out:b0,b1)
ENT1-2(in:a0,a1, out:b0,b1,b2)
```

Note that each entity is declared with an additional output port. The reason is that any of the points could be a target point, in which case we will need an additional output wire to collect the result.

This entails a transformation of the expression `e` into an imperative one which will be used when the structure of `V'` is defined. In `e'` we insert in each sub-expression an additional test on the number of supply directions from `p`. This will determine for each `p` the exact design entity to be used for a given `p`. The I/O ports of the entity will be instantiated in terms of `V'` indexed by the canonical labelling of requests applied for the point.

Consider again our example. Assuming that the number of supply directions from a point in `D` is uniformly 2 across the whole DDA, then the expression `e'` would become (there is no need for additional if statements):

```
if (cond1(p))
  ENT1-1 port map (V'[c(p,rb(p,0))], V'[c(p,rb(p,1))],
   V'[c(sp(p,0),sb(p,0))],V'[c(sp(p,1),sb(p,1))])
else if (cond2(p))
  ENT2-1 port map (V'[c(p,rb(p,0))], V'[c(p,rb(p,2))],
     V'[c(sp(p,0),sb(p,0))],V'[c(sp(p,1),sb(p,1))])
     else ENT3-1 port map (V'[c(p,rb(p,0))],
     V'[c(sp(p,0),sb(p,0))],V'[c(sp(p,1),sb(p,1))])
```

Initial points will need to be assigned special entities which have the mere role to duplicate or triplicate, etc., the input signal according to the supply directions of the initial point. E.g. `INIT0(in:a0, out:b0)` would be used for initial points with one supply direction, `INIT1(in:a0, out:b0,b1)` for initial points with two supply directions, etc.

Finally, we parse our space-time grid with space-time coordinates `(s,t)`. Then the structure of the global signal-array `V'` is obtained by creating for

every point `p=P(s,t)` a circuit-node, e.g., labelled `node-p`. Then each of these will be assigned one of the previously defined (computational) entities, and the entity's I/O ports will be mapped to the relevant signals in `V'`:

- If `p` is an initial point, then depending on the number of its supply directions, `node-p` will be assigned an initial entity `INITx` with input port mapped to a designated global input signal corresponding to the initial value `V[p]`, and each output port will be mapped to `V'[c(sp(p,b),sb(p,b))]` for all relevant `b`.
- If `p` is not an initial point, then we execute `e'` and assign `node-p` the entity determined by `e'`, evaluating all I/O port-expressions for the given `p`. If `p` happens to be a target point, then the next entity in the same range but one additional ouput is assigned to `node-p` instead. The extra output port is mapped to a designated global output signal.

Finally, each `node-p` is attributed with RLOC placements which will determine the node's relative position in the circuit. The space-time coordinates `(s,t)` implicitly provide this information. Naturally, `s` will be interpreted as the X coordinate, and `t` as the Y coordinate, leading to the RLOC labelling "XsYt", for instance. Note that the space-time projections can fully determine the layout of the circuit, and hence a different set of space-time projections will lead to different circuit layout as well.

This concludes the high-level presentation of a targeted repeat statement induced circuit design. The process is mainly concerned with the generation of the circuit description itself (e.g., creating the signal-array, then all possible entities, generating nodes and assigning entities and RLOCs for them). Some of the process can be straightforwardly automatised, especially those related to the structure of the DDA (e.g. the signal-array `V'` or the RLOCs generation). Certain parts of the process are less trivial as they require more elaborate source-code transformation, e.g., when decomposing `e` to transform the sub-expressions `foo` from the host language into the chosen hardware description language describing the behaviour or architecture of the corresponding entity.

CHAPTER 6

Programming with Data Dependencies

DDA-based programming may seem pretty awkward at first encounter and, without doubt, it takes a considerable amount of time to get into the right mindset for it. It forces the programmer *to think* about data dependencies, and to think about them in a completely *new perspective*. When dealing with a given computation the underlying data dependency needs to be analysed in order to capture it, if possible, and turned into real program code, instead of just simply assuming its implicit existence. This involves the making of several sketches, drawings etc. before coming up with a correct DDA definition. DDA-visualization tools may help a good deal in verifying the correctness of the DDA, since the generated graph laid out via the requests should be the same as the one laid out via the supply directions. Then the original computation needs to be *re-formalised* accordingly, so that the data dependency – now as code, but separated in a modular way – will become explicit in the whole computation. The rest is up to the DDA-enabled compiler with execution schemes in place for various hardware. The programmer should only specify which of these hardware is the target, and embed the program DDA into the hardware space-time DDA by means of additional DDA-projections.

DDA-based approaches are best suited when computations exhibit static, scalable, repetitive patterns that do not depend on the value of the actual input data. For instance, Quicksort's dependency pattern depends on the values of the pivots chosen at every step. Even if the pivot is chosen from the same index, each different input array would create a different data

dependency pattern. This makes Quicksort ill-suited for DDA-based approaches. On the other hand, sorting algorithms that can be represented as sorting networks, e.g. the bitonic sorting and odd-even merge sorting are well-suited.[1] Certain partial differential equations-based computational problems, e.g. heat-flow in one dimension, or dynamic programming problems, e.g. the stagecoach problem, the combinatorial calculation of binomial coefficients, optimal polygon triangulation, etc. are also well-suited. These and other DDA-based solutions of DP-problems and of numerical computations are presented in [Haveraaen and Søreide, 1998; Raubotn, 2003].

This chapter illustrates DDA-programming through the elaboration of DDA-based solutions for well-known computational problems and discusses how the computations can be mapped onto various (parallel) hardware from *the same* DDA-code. Computational solutions when targeting specific hardware are ususally characterised by the same property: the more finely-tuned, super-optimized the code is for the specific hardware, the less portable it is. DDA-abstractions, on the other hand, not only give a good intuition into how to parallelize a given computation, but support portability. The emphasis in our solutions is therefore not so much so on the "how to" parallelize these computations – countless solutions can be found in the literature –, but rather on "the high-level" that parallelization can be dealt with from DDAs, under the flag of portability.

Section 6.1 discusses the bitonic sorting which is primarily based on the presentation given in [Burrows and Haveraaen, 2009b], extended with a discussion on how bitonic sorting can be mapped onto a shuffle network based on its underlying DDA. Section 6.2 presents a DDA-based odd-even merge sorting network, Section 6.3 defines the forward and inverse Fast Fourier Transforms in terms of its underlying DDA, and Section 6.4 defines a parallel prefix DDA based on the Sklansky construction.

6.1 Bitonic Sort DDA

This section presents the bitonic sorting as a fully worked example of DDA-based programming. The bitonic sort is a well studied parallel sorting algorithm [Grama et al., 2003], first presented by K.E. Batcher [Batcher, 1968]. It sorts n elements in parallel in $\Theta(\log^2 n)$ time. The basic operation consists of repeatedly merging bitonic sequences of increasing size into ascending

[1] A sorting network is a computational model consisting of horizontal wires (one for each input) and a collection of two-sorters (each of these placed as a vertical segment on two of the wires) such that these can be set in advance for a given sorting algorithm, regardless of the actual inputs.

or descending sequences. A sequence is called bitonic when it is the juxtaposition of an ascending and a descending sequence. In the beginning we have bitonic sequences of length two, which are combined to bitonic sequences of length four, and so forth. In the penultimate step we have a bitonic sequence of the size of the input sequence, and in the final step this last bitonic sequence is merged into a sorted sequence.

The data dependency of a bitonic sorter can be seen as a combination of several butterfly DDAs of different height, see Fig. 6.1. At the top is one butterfly of size $\mathtt{h}$, underneath two butterflies of size $\mathtt{h}-1$, and so forth. At the bottom are $2^{\mathtt{h}-1}$ butterflies of height 1. Each sub-butterfly corresponds to a bitonic merge. Bitonic sorting is defined then as min/max functions on the points of this DDA.

Example 6.1.1. The *bitonic sort DDA for* $2^{\mathtt{h}}$ *inputs*, $\mathtt{h} \in \mathbf{N}$, $\mathtt{DBS_h}$, is defined by:

1. DDA point: $\mathtt{BS_h}$ `= sbf Nat * row Nat * col Nat |` $\mathtt{DI_h}$ where:

```
DIh(p) = (0<sbf(p)<=h) &&
         ((row(p)<sbf(p)) || (row(p)<=sbf(p) && sbf(p)=1)) &&
         (col(p)<2^h)
```

2. branch indices: $\mathtt{B} = \{\mathtt{0}, \mathtt{1}\}$

3. request components `(rg,rp,rb)` where:

```
rg(p,b) = (row(p)!=1) || (sbf(p)!=1)
rp(p,b) =
    if (((sbf(p)>1) && (0<=row(p)<=sbf(p)-2)) ||
       ((sbf(p)=1) && (row(p)=0)))
           if (b=0) BSh(sbf(p),row(p)+1,col(p))
           else BSh(sbf(p),row(p)+1,flip(row(p),col(p)))
    else if ((sbf(p)>1) && (row(p)=sbf(p)-1))
           if (b=0) BSh(sbf(p)-1,0,col(p))
           else BSh(sbf(p)-1,0,flip(row(p),col(p)))
rb(p,b) = b
```

4. supply components `(sg,sp,sb)` where:

```
sg(p,b) = (row(p)!=0) || (sbf(p)!=h)
sp(p,b) =
    if ((sbf(p)!=h) && (row(p)=0))
```

```
            if (b=0) BS_h(sbf(p)+1,sbf(p),col(p))
            else BS_h(sbf(p)+1,sbf(p),flip(sbf(p),col(p)))
      else if (((sbf(p)=1) && (row(p)=1)) ||
         (sbf(p)>1) && (1<=row(p)<=sbf(p)-1))
            if (b=0) BS_h(sbf(p),row(p)-1,col(p))
            else BS_h(sbf(p),row(p)-1,flip(row(p)-1,col(p)))
sb(p,b) = b
```

□

The first projection `sbf` of the point sort $\mathtt{BS}_h$ identifies the sub-butterfly's height, the `row` projection is the local row number in the sub-butterfly, while `col` gives the global column number. Note that for the points which belong to two sub-butterflies, we use the projections corresponding to the lower butterfly.

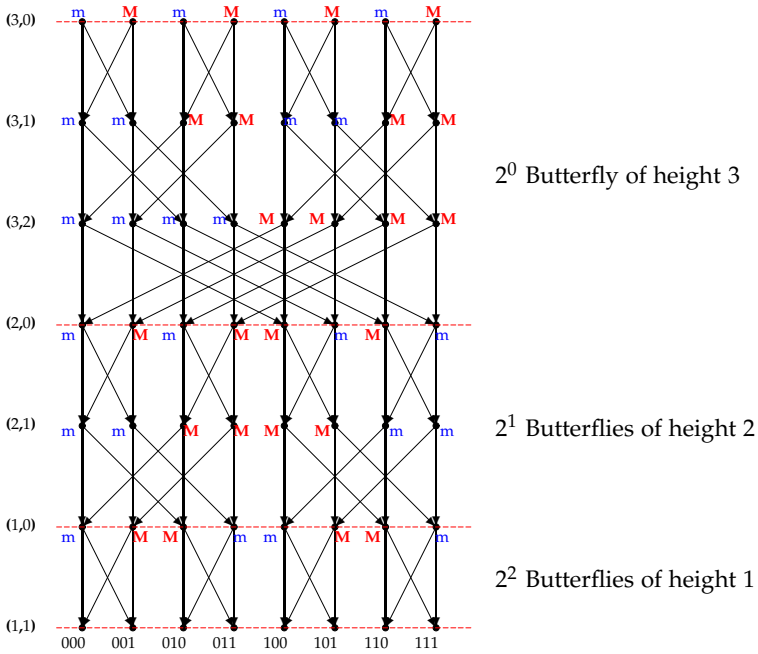

Figure 6.1: *Bitonic sort DDA for* 2^3 *inputs. The* `col` *projections of* $\mathtt{BS}_3$ *points are presented in binary at the bottom. The* `sbf` *and* `row` *projection are presented as pairs on the left. At each node, either a minimum ("m") or a maximum ("M") is computed.*

The role of the outer conditionals is to determine where in the DDA the point is situated (e.g. which sub-butterfly, which row within the sub-buterfly), and the inner conditional simply distinguishes between the two branch indices. Just as in the case of the butterfly DDA, branch index `0` is used to label horizontal arcs, and branch index `1` to label arcs across for both requests and supplies. The definition glues together butterflies of increasing height on top of each other.

The computations to be performed at each node are defined next. The array `V` represents the result of the entire computation, i.e., data for all the steps of the bitonic sort. Let $\texttt{v}_\texttt{i}$, $\texttt{i}\in\{\texttt{0},\texttt{1},\ldots,2^\texttt{h}\texttt{-1}\}$ be the input elements to be sorted. They reside on the bottom points of the DDA graph: $\texttt{V[BS}_\texttt{h}\texttt{(1,1,i)]}=\texttt{v}_\texttt{i}$. We define a condition whether we should do a minimum or a maximum:

```
cond(p) = (bit(sbf(p),col(p)) = bit(row(p),col(p))
```

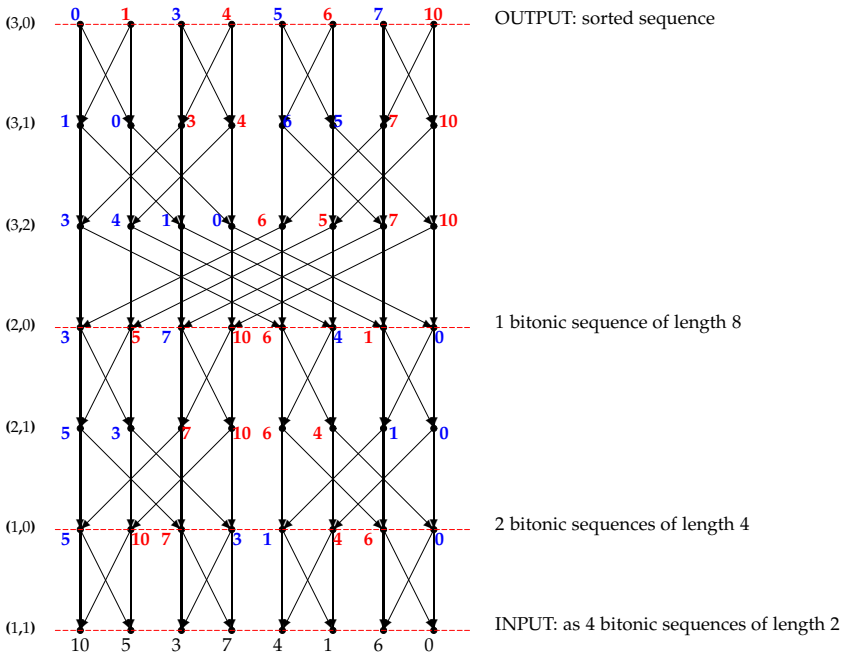

FIGURE 6.2: *Example of sorting* 2^3 *inputs. On the bottom nodes reside the actual initial data values to be sorted. The rest of the nodes show the values* `V[p]` *computed at each time-step. On the top row the initial data values have become sorted.*

Here `bit:Nat,Nat→Nat` is a function, s.t., `bit(i,n)` extracts the $\mathtt{i}^{th}$ bit from the binary representation of the number `n`.

We can now define the expression `V[p]` to be computed for all $\mathtt{p:BS_h}$ by:

```
repeat p:BS_h along DBS_h from V in
       if cond(p) min(V[rp(p,0)],V[rp(p,1)])
       else max(V[rp(p,0)],V[rp(p,1)])
```

To understand the computation defined by the repeat statement recall that the expressions are guarded, so they will stay within the limits of the DDA $\mathtt{DBS_h}$. The explanation below follows the imperative version of the algorithm from Definition 5.2.1 (dependency-driven computation).

Accordingly, each point `p` is associated with a memory cell `M[p][b]` for each $\mathtt{b} \in \mathtt{B}$, and the sub-expressions `V[rp(p,b)]` are replaced with `M[p][b]`. The computation is started by fetching the data $\mathtt{v_{col(p)}}$ and using this as the value of `V[p]` for each point `p` at the bottom row. The value `V[p]` is then supplied to the memory cells `M[sp(p,b)][sb(p,b)]` for every $\mathtt{b}\in\mathtt{B}$ and point `p` on the bottom row.

The computation proceeds, for each point `p` where the required memory cells have been given a value, to compute the expression `V[p]`. This value is then supplied to the memory cells `M[sp(p,b)][sb(p,b)]` for every $\mathtt{b}\in\mathtt{B}$.

The result of the computation are the values of `V[p]` for the points `p` on the top row. Depending on the target machine they may remain in memory for being used in future computations, or offloaded in some way or another.

Fig. 6.2 shows the result of the computations of the values `V[p]` on points $\mathtt{p:BS_3}$ for a concrete initial data set.

How the computation actually proceeds through the DDA (row parallel, a point at a time, or skewed in some strange way), can be controlled by the embedding of the bitonic sort DDA into the space-time of a chosen hardware architecture.

6.1.1 Embedding into a Shared Memory Model Architecture

If the hardware is comprised by a shared memory model architecture then the embedding is given by a simple point projection.

Example 6.1.2. An *embedding of a bitonic sort DDA for* 2^h *inputs into a shared memory STA of size* 2^h is given by $\mathtt{EP:BS_h} \rightarrow \mathtt{SMST_{2^h}}$ as follows:

$$\mathtt{EP(p)} = \mathtt{SMST_{2^h}(col(p),grow(sbf(p),row(p)))}$$

where the function `grow:Nat,Nat→Nat` computes the global row number of a point in the bitonic sort DDA based on the sub-butterfly and local row numbers, defined by `grow(b,r) = b(b-1)/2 + b-r`. □

Then the algorithm follows a row parallel execution model, either the request or supply based space-time controlled repetitions as presented in Definitions 5.3.2 and 5.3.6.

6.1.2 Hypercube Embedding

Example 6.1.3. An *embedding of a bitonic sort DDA for* 2^h *inputs into a hypercube space-time DDA of dimension h* is given by the functions (see Fig. 6.3):

```
EP(p)   = HSTh(col(p),grow(sbf(p),row(p)))
ER(p,b) = if (b = 0) 0
          else row(p)+1
ES(p,b) = if (b = 0) 0
          else if (row(p) = 0) sbf(p)+1
               else row(p)
```

□

We can now execute the bitonic sorter `V` on the hypercube. The steps for the $\mathtt{DBS}_h$ points `p` become iterations at each processor `node(EP(p))=col(p)` for time-steps `time(EP(p))`: receive data from channels `ER(p,b)` for all relevant `b:B`, compute `V`, and then send the result on channels `ES(p,b)` for all relevant `b:B`. In the first step the initial values $v_{\mathtt{col(p)}}$ will reside in the memory of the corresponding node, and in the final step the computed values (sorted data) will also reside in the memory of the nodes.

If there are more data than processors, we introduce multiple memory cells on each node. We will show this principle when discussing the GPU version below.

6.1.3 CUDA Embedding

Example 6.1.4. An *embedding of* $\mathtt{DBS}_h$ *into* $\mathtt{DCUST}_{B,T}$ where `2^h<=B*T` becomes:

```
EP(p)   = CUSTB,T(CUBB,T(col(p)/T,col(p)%T),grow(sbf(p),row(p)))
ER(p,b) = CUBB,T(col(rp(p,b))/T,col(rp(p,b))%T)
ES(p,b) = CUBB,T(col(sp(p,b))/T,col(sp(p,b))%T)
```

where `rp` and `sp` are the request and supply functions of $\mathtt{DBS}_h$. □

Note that the explicit definition of `ER(p,b)` and `ES(p,b)` are in fact superfluous. As discussed earlier in Section 4.3.5 they are deductible from `EP`, for instance, `ER(p,b)=space(EP(rp(p,b)))`.

We see that the bitonic sort will be split across several kernels, as appointed by the kernel scheduler, in order to communicate between blocks

for the wider branches. The process in each thread will receive data by reading from the memory location corresponding to the channel given by $\texttt{CUB}_{\texttt{B,T}}$. If this channel is across block numbers, then the read is from the global GPU memory. Otherwise it is local block memory. Then the appropriate min/max values are computed, and data is stored in the appropriate $\texttt{CUB}_{\texttt{B,T}}$ memory location. Inter-block storage means storing in the global GPU memory. Storage to global GPU memory also takes place at the end of the kernel. With this strategy, the initial data sets are stored in the relevant locations on the global GPU memory, and the result of the bitonic sort algorithm likewise ends in the global GPU memory.

Fig. 6.4 illustrates the embedding for `T=4` and `B=4`. Data shared within one block is through fast shared memory. When the edges of the bitonic sort DDA graph intersects block borders, threads of different blocks need to share data via the global GPU memory. The darker grey frames across

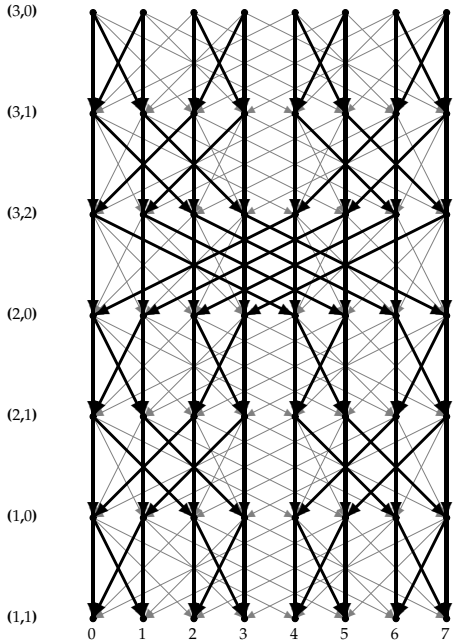

Figure 6.3: *Bitonic sort DDA for* 2^3 *inputs embedded into the hypercube space-time DDA of dimension* 3 *for* 7 *time-steps. Request and supply branches of the bitonic sort DDA (thick lines) are mapped to hypercube communication channels.*

all threads highlight this: threads write to the GPU memory, the kernel terminates, and control is handed over to the host, which invokes a new kernel with threads first fetching data from the GPU memory.

Note that the address splitting technique used here between blocks and threads, also can be used to split large networks between nodes and local memory on, e.g., a hypercube architecture.

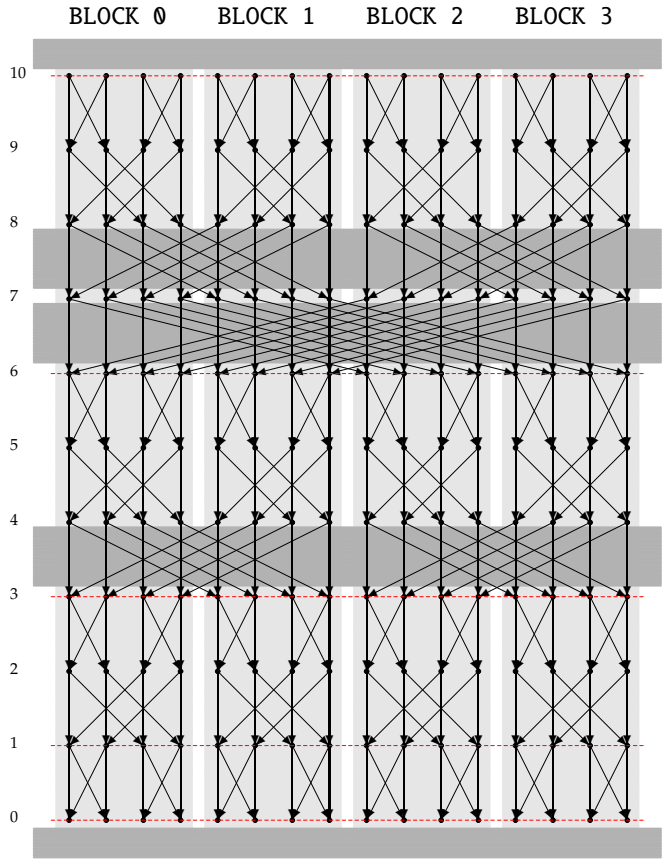

FIGURE 6.4: *Bitonic sort DDA for* 2^4 *inputs, embedded into NVIDIA's CUDA programming model, executed by* 4 *kernel invocations, each consisting of* 4 *blocks of threads.*

6.1.4 *Shuffle Network*

There exists several repetitive network implementations for the bitonic sorting based on the perfect shuffle, e.g. Stone [1971]'s one-stage perfect shuffle or the more efficient recirculating bitonic sorting network of Lee and Batcher [2000]. In both cases additional mechanisms are imposed on the network in order to ensure right dataflow, or to achieve better running time performance.

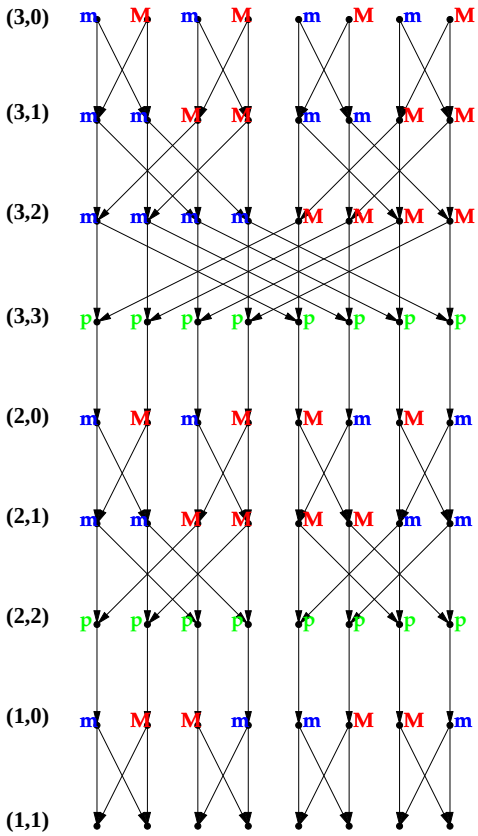

Figure 6.5: *The alternative bitonic sort DDA with extra communication steps between sub-butterfly levels for* 2^3 *inputs. Nodes are annotated with computational expression labels:* m *stands for minimum,* M *stands for maximum and* p *stands for pass-on (equivalent to identity).*

The bitonic sort DDA the way it is defined, is not suitable for mapping it onto an omega network directly. Applying the shuffle projection on its points (see Sec. 4.4), will result in an irregular succession of different shuffles. However, if the DDA is defined such that the sub-butterflies are not glued together, but pasted with an extra one-step communication, then the pattern will become more regular. In the following a DDA-based shuffle network implementation is sketched for the bitonic sorting based on this extended version of the bitonic sort DDA. The implementation-sketch underlines the role of DDA-projections in the design process.

In the new bitonic sort DDA new horizontal arcs are added between each level where sub-butterflies meet, one for each point (Fig. 6.5). This increases

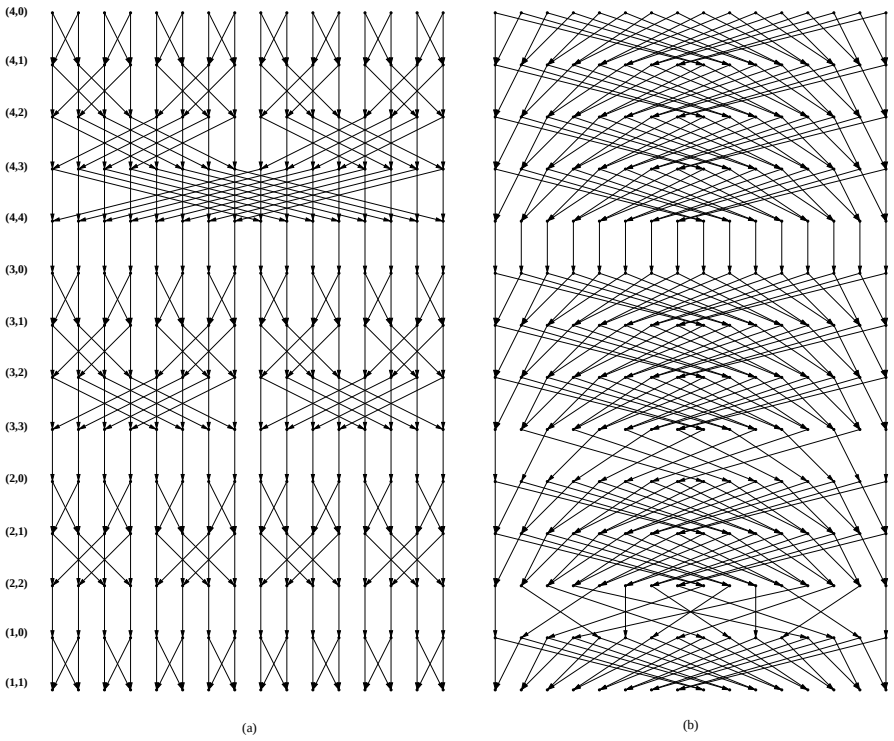

FIGURE 6.6: *The alternative bitonic sort DDA with extra communication steps between sub-butterfly levels for* 2^4 *inputs. (a) Layout with the usual* `row` *and* `col` *projections. (b) Shuffle layout with the alternative column projection* `ShuffleCol`.

the DDA height with `h-1`, and the number of points with $(h-1) * 2^h$. Accordingly, `sbf(p)` at the points of the bottom rows of each sub-butterfly will now return the actual sub-butterfly index, unlike before when they pointed to the sub-butterfly below, and `row(p)=sbf(p)` instead of 0 for these points now. The effect of the computation at any of these new points is the passing-through of data (equivalent to the identity function): it receives the data along the new arc from the point right below and it passes it on.

Then the shuffle projection `ShuffleCol(p) = ShR(col(p),row(p))` will turn every level of the DDA into a perfect shuffle, apart from the new connecting levels (see Fig. 6.6.b). Here the arcs instead of a perfect shuffle will show a different permutation, depending on how many steps are skipped in an imaginary omega network. Hence we will refer to these as the *hopping shuffle permutations*. (Remember that an omega network is a `h`-step interconnection of 2^h-keys perfect shuffles, so that in the end the keys get back to their original positions.)

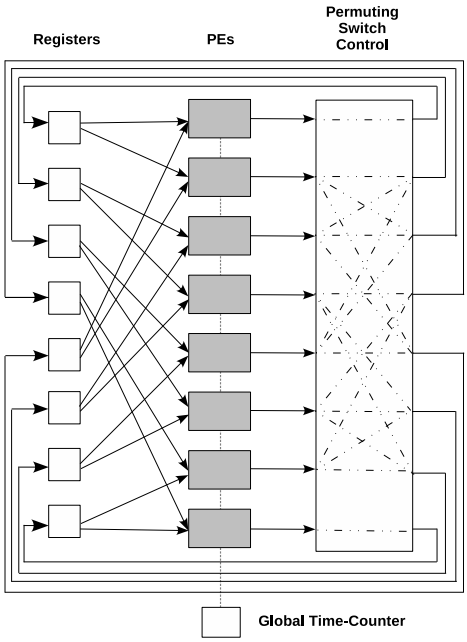

Figure 6.7: *The schematic overview of a DDA-based shuffle network performing bitonic sorting of* 2^3 *inputs. (The dataflow is now identical with the direction of the arrows.)*

In contrast to the shuffle networks proposed by [Lee and Batcher, 2000; Stone, 1971] which are based on 2^{h-1} identical comparators at the processing level, the DDA-based shuffle network is based on 2^h processing elements (PEs), each running the same code. In addition, there are 2^h storage registers, and the network is endowed with a permuting switch control that performs the hopping shuffle permutation. A schematic overview of such a network is presented in Fig. 6.7.

The whole computation is controlled by a global time-counter corresponding to the new bitonic sort DDA global time projection, so that `TMAX = h*(h+1)/2 + h-1` is the final time-step. The code running on the PEs computes either the minimum or the maximum of the corresponding register values depending on the condition defined on the DDA point projected onto the given PE at the global time-step.

The computation starts up by placing the inputs into the registers in a shuffled order, i.e. register $R_{ShR(i,1)}$ receives input data indexed by `i`, and the global time-counter is set to 1. Then each PE deduces from the global time-step and its PE-index which DDA point is assigned to it. PE P_i at time-step `t` is dealing with a computation assigned via a DDA-projection s.t. `i = ShR(col(p),row(p))`. From the global time-step `t`, given the new bitonic sort DDA structure, it can always be deduced the actual value of `row(p)` (and of `sbf(p)`). From this we obtain `col(p) = ShL(i,row(p))`, where `ShL` is the inverse of `ShR`. Which leads us back to the point `p` for which `V[p]` is to be computed by PE P_i. When all PEs have finished computing this, it is checked if a hopping shuffle permutation is needed before the data is passed on to the next network cycle:

- If `row(p)!=0`, the data is passed through the permuting switch control via its original track, and the time-counter is increased by 1.
- If `(row(p)=0) && (t<TMAX)`, then the data is passed through the permuting switch control requesting that the output of P_i should be positioned to the track `ShR(i,sbf(p)+1)`, and the time-counter is increased by 2. This step corresponds to the hopping shuffle permutation.
- If `t=TMAX`, i.e., the elements have been sorted, the data is passed through the switch control via its original track and the result will reside in the registers, in the natural order of register indices.

6.2 Odd-Even Merge Sort

The odd-even merge sort is another fast sorting network presented by K.E. Batcher alongside the bitonic sort [Batcher, 1968]. It sorts n elements in

$\Theta(\log^2 n)$ time in parallel. The network is based on the repeated merging of ascendingly-ordered sequences of increasing size into one ascendingly-ordered sequence. The merging of two sequences of size one consists of just a simple comparison. The merging of larger ascendingly-ordered sequences is defined recursively by presenting the odd-indexed elements of both sequences to a smaller merge (the odd-merge) and the even-indexed elements of both sequences to another smaller merge (the even merge), and then comparing the results of these smaller merges with a row of pairwise comparisons. In the beginning we merge sequences of size one (simple comparison) into ascendingly-ordered sequences of size two. These are then merged to ascendingly-ordered sequences of length four, and so forth.

The data dependency of the odd-even merge sorting is presented next when n is a power of two (see Fig. 6.8). The merging steps will define distinguishable sub-patterns of increasing size. The first projection of the DDA point sort identifies the height of a sub-merge step, the `row` projection is the local row number in a sub-merge, while `col` gives the global column number. Similarly to the bitonic sort DDA (as first presented), for the points which belong to two sub-merges, the `row` and `sm` projections refer to the lower sub-merge. All vertical arcs are labelled with `0` and all arcs across are labelled with 1.

Example 6.2.1. The *odd-even merge sort DDA for* 2^h *inputs*, $h \in \mathbf{N}$, $\mathtt{DOEMS}_h$ is defined by:

1. DDA point: $\mathtt{OEMS}_h$ `= sm Nat * row Nat * col Nat |` $\mathtt{DI}_h$ where:

```
DIh(p) = (0<sm(p)<=h) &&
         ((row(p)<sm(p)) || (row(p)<=sm(p) && sm(p)=1)) &&
         (col(p)<2^h)
```

2. branch indices: $\mathtt{B} = \{0, 1\}$

3. request components `(rg,rp,rb)` where:

```
rg(p,b) = !((row(p)=1) && (sm(p)=1)) && !(pass(p) && (b=1))
rp(p,b) =
    if (row(p) <= sm(p)-2 || sm(p)=1 )
        if (b=0) OEMSh(sm(p),row(p)+1,col(p))
        else if max(p) OEMSh(sm(p),row(p)+1,col(p)-2^row(p))
             else OEMSh(sm(p),row(p)+1,col(p)+2^row(p))
    else
```

```
        if (b=0) OEMS_h(sm(p)-1,0,col(p))
        else if max(p) OEMS_h(sm(p)-1,0,col(p)-2^row(p))
             else OEMS_h(sm(p)-1,0,col(p)+2^row(p))
rb(p,b) = b
```

4. supply components `(sg,sp,sb)` where:

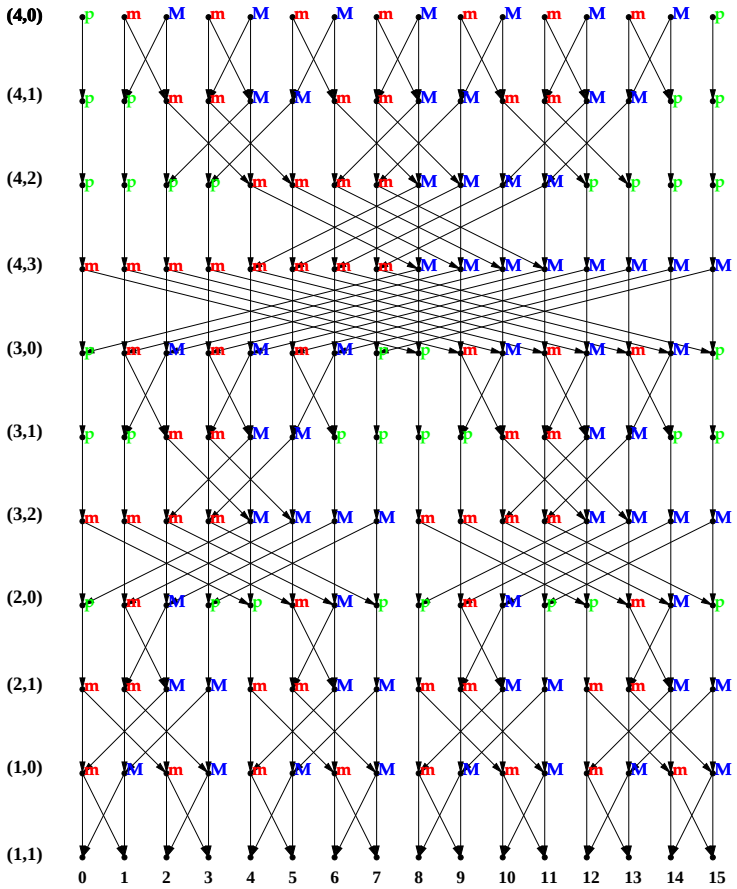

FIGURE 6.8: *The odd-even merge sort DDA for* 2^4 *inputs. Nodes are annotated with computational expression labels:* m *stands for minimum,* M *stands for maximum and* p *stands for pass-on (equivalent to identity).*

```
sg(p,b) = !((row(p)=0) && (sm(p)=h)) &&
          !((row(p)>=1) && (b=1) &&
            pass(OEMS_h(sm(p),row(p)-1,col(p))) )
sp(p,b) =
    if (row(p) = 0)
        if (b=0) OEMS_h(sm(p)+1,sm(p),col(p))
        else if max(OEMS_h(sm(p)+1,sm(p),col(p)))
                 OEMS_h(sm(p)+1,sm(p),col(p)-2^sm(p))
             else OEMS_h(sm(p)+1,sm(p),col(p)+2^sm(p))
    else if (b=0) OEMS_h(sm(p),row(p)-1,col(p))
         else if max(OEMS_h(sm(p),row(p)-1,col(p)))
             OEMS_h(sm(p),row(p)-1,col(p)-2^row(p)-1)
              else OEMS_h(sm(p),row(p)-1,col(p)+2^row(p)-1)
sb(p,b) = b
```

where `pass:`$\mathtt{OEMS}_h \to$`bool` and `max:`$\mathtt{OEMS}_h \to$`bool` are two functions defined on points `p:`$\mathtt{OEMS}_h$ as follows:

```
pass(p) = (row(p) < sm(p)-1) &&
          ((col(p)%2^sm(p) < 2^row(p))||(2^sm(p)-2^row(p) <= col(p)%2^sm(p))

max(p) =
    ((row(p)<sm(p)-1) && !(pass(p)) && (col(p)%2^row(p)+1<2^row(p)))
    ||
    ((row(p)=sm(p)-1) && (2^row(p)<=col(p)%2^row(p)+1))
```

□

The above definition, cumbersome as it is, can be easily decoded by following the meaning of the DDA-projections and constructor in Fig. 6.8. The auxiliary boolean functions `pass` and `max` capture common properties of the DDA points. E.g. each point for which `pass(p)` holds receives data along only one vertical arc. All points for which `max(p)` holds receive data along a vertical arc and one which is descending to the left from that point. All the rest of the points, for which none of the above properties hold, receive data along a vertical arc and one which is descending to the right. Alternatively, one could have chosen a larger set of branch indices to differentiate between left-descending and right-descending arcs, and so on. That could have resulted in simpler DDA code, but in turn it would have complicated the expression inside the repeat statement. As we shall see, the auxiliary functions will make the latter fairly straightforward. In the definition of requests, the auxiliary functions are applied directly to the point in case. In

the definition of supplies, they are applied to the point right above the point in case, hence the use of the constructor $\texttt{OEMS}_\texttt{h}$ in the argument of `pass` and `max` in the definitions of `sg` and `sp`, respectively.

The basic operations of the odd-even merge sort are the usual minimum and maximum, and in addition there is the identity or pass-on computation, in which data is received and passed-on along each supply arc without any change. The predicates we used in the definition of the DDA now come in very handy. They also help to differentiate between the points as far as the computation at that point is concerned.

Given an array `V` with element type `E` and index type $\texttt{OEMS}_\texttt{h}$, the expression `V[p]` to be computed for all $\texttt{p:OEMS}_\texttt{h}$ will be given by the repeat statement as follows:

```
repeat p:OEMSh along DOEMSh from V in
    if (pass(p)) V[rp(p,0)]
    else if (max(p)) max(V[rp(p,0)],V[rp(p,1)])
         else min(V[rp(p,0)],V[rp(p,1)])
```

The embedding of the computation is again controlled by defining embedding projections for $\texttt{DOEMS}_\texttt{h}$ into the space-time of an available hardware architecture. If this is a shared memory model architecture the embedding projection is similar to the one defined for the bitonic sort DDA:

$$\texttt{EP(p) = SMST}_{2^\texttt{h}}\texttt{(col(p),grow(sm(p),row(p)))}$$

Regarding the hypercube, we see that $\texttt{DOEMS}_\texttt{h}$ is not sub-isomorphic to $\texttt{DHST}_\texttt{h}$, hence one could only attempt to define embeddings which require multiple-step communication channels on the hypercube. A theory for multiple-step communication embeddings can be found in [Haveraaen, 2009].

A Cuda embedding projection, on the other hand, is easily defined:

$$\texttt{EP(p) = CUST}_{\texttt{B,T}}\texttt{(CUB}_{\texttt{B,T}}\texttt{(col(p)/T,col(p)\%T),grow(sm(p),row(p)))}$$

This will partition the threads in a similar manner as presented for the bitonic sort DDA in Fig. 6.4. However, remember that embeddings for the same hardware architecture can be defined in many different ways by modifying the point projections. Consider the following alternative Cuda embedding projection:

$$\begin{aligned}&\texttt{EP'(p)=}\\&\quad\texttt{if (sm(p)=h) CUST}_{\texttt{B,T}}\texttt{(CUB}_{\texttt{B,T}}\texttt{(ShR}_\texttt{h}\texttt{(col(p),1)/T,ShR}_\texttt{h}\texttt{(col(p),1)\%T),}\\&\qquad\qquad\qquad\texttt{grow(sm(p),row(p)))}\\&\quad\texttt{else CUST}_{\texttt{B,T}}\texttt{(CUB}_{\texttt{B,T}}\texttt{(col(p)/T,col(p)\%T),grow(sm(p),row(p)))}\end{aligned}$$

Fig. 6.9 illustrates the result of this projection. We can see that it is only the layout of the top sub-merge which differs from the one in Fig. 6.8, and for the better. When the number of threads `T` is chosen to be half of the number of inputs, $\texttt{T}=2^{h-1}$, then the number of kernel invocations is significantly reduced in the last sub-merge. Here less branches cross over block boundaries, hence there is no need for a new kernel invocation at every time-step (of the last sub-merge).

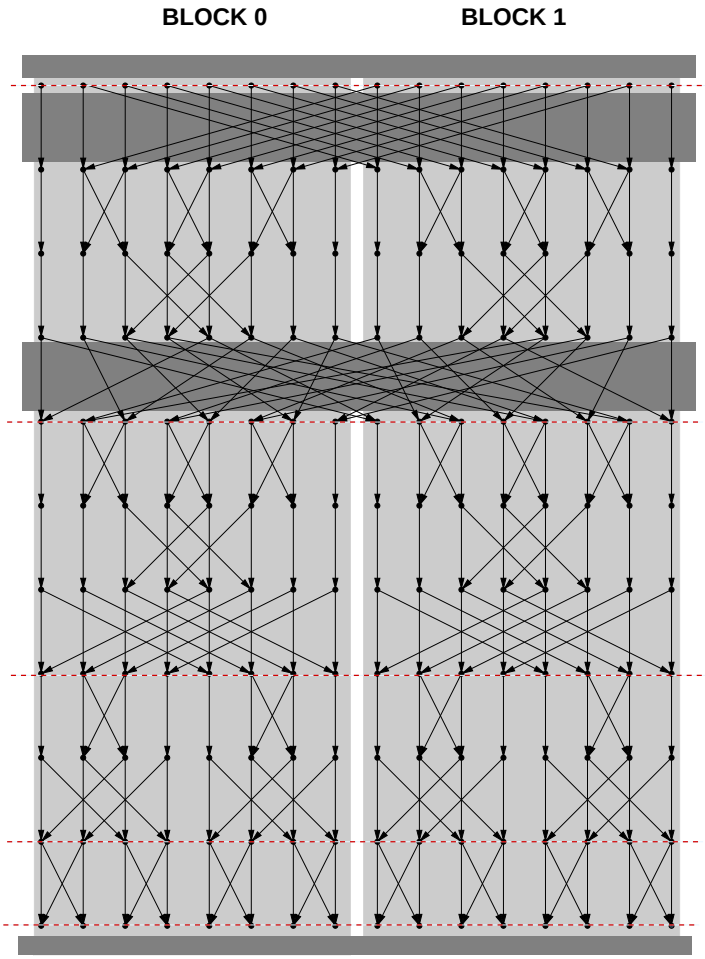

Figure 6.9: *The odd-even merge sort DDA for* 2^4 *inputs with Cuda embedding projections given by* `EP'` *for* `T=8` *and* `B=2`*. The embedding induces only* 3 *kernel invocations.*

6.3 Fast Fourier Transform

The *Discrete Fourier Transform* (DFT) is a linear transformation applied to a function over a discrete domain, e.g. time, in order to turn it into a function over another domain, e.g. frequency. DFT plays an important role in countless scientific and technical applications, for instance time series and waveform analysis, solving partial differential equations in computational fluid dynamics, digital signal processing, multiplying large polynomials, etc. [Bergland, 1969; Emiris and Pan, 2010]

The input of DFT can be seen as a sequence of $n \in \mathbf{N}$ complex numbers $X = \{x_0, x_1, \ldots, x_{n-1}\}$ which is transformed by DFT into the sequence of complex numbers $Y = \{y_0, y_1, \ldots, y_{n-1}\}$ by the formula:

$$y_i = \sum_{k=0}^{n-1} x_k \omega^{ik} \qquad i = 0, \ldots, n-1 \tag{6.1}$$

where $\omega = e^{\frac{2\pi i}{n}}$ is a primitive n-th root of unity (see Section 3.1.2). The powers of ω in the formula are usually referred to as *twiddle factors*.

Then the *inverse DFT* is given by the formula:

$$x_k = \frac{1}{n} \sum_{i=0}^{n-1} y_i \omega^{-ik} \qquad k = 0, \ldots, n-1$$

The time complexity of computing Y by the formula (6.1) is characterized by $\Theta(n^2)$ operations (multiplication and addition of complex numbers). Since DFT has been used in many scientific applications, the machine calculation of these complex Fourier series needed to be optimised. As a result many efficient algorithms have been proposed along the years, which commonly are referred to as *Fast Fourier Transform* (FFT) algorithms. [Duhamel and Vetterli, 1990]

By far the most common FFT is the *Cooley–Tukey algorithm*, based on a divide and conquer technique that recursively breaks down a DFT into smaller DFTs. [Cooley and Tukey, 1965] In general the Cooley–Tukey FFT algorithms reduce the complexity to $\Theta(n \log n)$. When n is a power of two, and applying the property $\omega^k = -\omega^{k+\frac{n}{2}}$ on the twiddle factors, the complexity can be further reduced to $\Theta(\frac{n}{2} \log n)$.

We will present now a DDA-based version of the *radix-2 FFT*, one of the most well-known Cooley–Tukey algorithm, in which the split is of factor 2 at every step, and n is assumed to be a power of two.

Following [Grama et al., 2003, p. 538], equation (6.1) can be transformed into:

$$y_i = \sum_{k=0}^{(n/2)-1} x_{2k}\tilde{\omega}^{ik} + \sum_{k=0}^{(n/2)-1} x_{2k+1}\tilde{\omega}^{ik} \tag{6.2}$$

where $\tilde{\omega} = e^{\frac{2\pi i}{\frac{n}{2}}} = \omega^2$ is a primitive $\frac{n}{2}$-th root of unity.

Each summation of the formula (6.2) is now a DFT of size $\frac{n}{2}$. These can be further divided in a recursive manner, such that the maximum number of levels of recursion is $\log n$ for any initial sequence of length n. At the deepest level of recursion (row 1 in Fig. 6.10), the elements of the sequence whose indices differ by $\frac{n}{2}$ are used in the computation. In each subsequent level, the difference between the indices of the elements used together in a computation decreases by a factor of two. This leads to the usual butterfly dependency which here is topped with an extra dataflow level responsible for shuffling the output (of row 3) into a bit-reverse order, so that the indices of the initial sequence and the final sequence become aligned.

This dependency is defined next as a DDA. The branch index used in the shuffling will be 0. The branch indices used on the butterfly will be as usual: all horizontal branches will have branch index 0 and all branches across will have branch index 1.

Example 6.3.1. The *Radix-2 Fast Fourier Transform DDA for* $2^{\mathtt{h}}$ *inputs*, $\mathtt{h} \in \mathbf{N}$, $\mathtt{DFFT_h}$, is defined by:

1. DDA point: $\mathtt{FFT_h}$ `= row Nat * col Nat |` $\mathtt{DI_h}$ where:

   $\mathtt{DI_h}$`(p) = (row(p)<=h+1) && (col(p)<`$2^{\mathtt{h}}$`)`

2. set of branch indices: $\mathtt{B} = \{0, 1\}$

3. request components `(rg,rp,rb)` where:

   ```
   rg(p,b) = (row(p)>0) && ((row(p)<h+1)||(b=0))
   rp(p,b) = if (row(p)=h+1) FFT_h(h,rev_h(col(p)))
             else if (b=0) FFT_h(row(p)-1,col(p))
                  else FFT_h(row(p)-1,flip(row(p),col(p)))
   rb(p,b) = b
   ```

4. supply components `(sg,sp,sb)` where:

   ```
   sg(p,b) = (row(p)<=h) && ((row(p)<h)||(b=0))
   sp(p,b) = if (row(p)=h) FFT_h(h+1,rev_h(col(p)))
             else if (b=0) FFT_h(row(p)+1,col(p))
                  else FFT_h(row(p)+1,flip(row(p)+1,col(p)))
   sb(p,b) = b
   ```

where `flip(i,m)` flips the i^{th} bit in the binary representation of `m`, and $\texttt{rev}_h(\texttt{m})$ reverses the bits-order in the `h`-bit binary representation of `m`. □

Let `V` be an array with index type $\texttt{FFT}_h$ and some element type `C` corresponding to complex numbers. `V` is initialised with the input sequence of type `C`, `x[0],x[1],...,x[n-1]`, such that $\texttt{V[FFT}_h\texttt{(0,i)]=x[i]}$ for all `i:{0,1,...,n-1}`, where $\texttt{n=2}^h$. That is, all input values reside on the bottom row of the DDA.

Then the forward FFT computation is given by the following repeat statement, where `w` denotes ω, the primitive `n`-th root of unity, and >> stands for the bit-wise shift operation:

```
repeat p:FFT_h along DFFT_h from V in
    if (row(p)<h+1)
        if (col(p)<col(rp(p,1)))
           V[rp(p,0)] + V[rp(p,1)]*w^rev_h(col(p)>>h−row(p))
        else V[rp(p,1)] + V[rp(p,0)]*w^rev_h(col(p)>>h−row(p))
    else V[rp(p,0)]
```

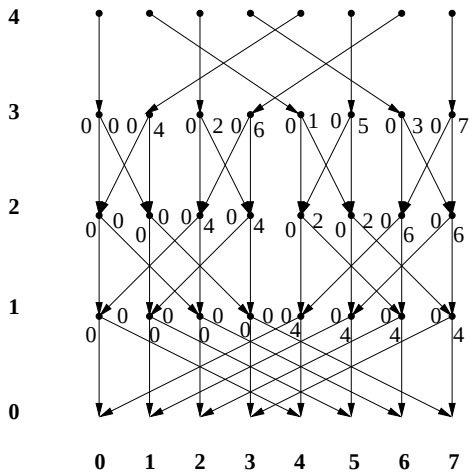

FIGURE 6.10: *The Radix-2 Fast Fourier Transform DDA for* 2^3 *inputs. Branches are weighed here with the twiddle factor exponents to illustrate how these fold out in the recursion, though note that these are not part of the DDA itself. It will be in the computation that the twiddle factor is multiplied with the value received along the corresponding request direction before the summation takes place at each point of the rows* 1-3.

Then the result of the Fourier transform will reside on the top row of the DDA. If {y[0],y[1],...,y[n-1]} is the desired output sequence, then y[i]=V[FFT$_h$(h+1,i)].

When it comes to the inverse FFT, the underlying dependency remains unchanged. The expression inside the repeat statement needs changing only: we use w^{-1} as the base of twiddle factors, and in the last step we apply complex number division by 2^h on the result.

Let W be an array with index type FFT$_h$ and element type C. Initial values are assigned again to the bottom row of the DDA, and the result will reside on its top row.

```
repeat p:FFT_h along DFFT_h from W in
    if (row(p)< h+1)
        if (col(p)<col(rp(p,1)))
           V[rp(p,0)] + V[rp(p,1)]*w^(-rev_h(col(p)>>h-row(p)))
        else V[rp(p,1)] + V[rp(p,0)]*w^(-rev_h(col(p)>>h-row(p)))
    else V[rp(p,0)]/2^h
```

Note that making the bit-reverse shuffle part of the DDA, DFFT$_h$ becomes less suitable for a direct embedding onto the hypercube though the butterfly pattern itself matches beautifully with the hypercube. The alternative would be to define the FFT computations on the butterfly dependency DBF$_h$, defined earlier. Then the result can be retrieved by applying the bit-reverse directly, e.g., y[i]=V'[BF$_h$(0,rev$_h$(i)] (in DBF$_h$ the top row is 0), assuming that V' is an array idexed by BF$_h$ and its values are computed by a corresponding repeat statement defined on DBF$_h$.

On the other hand, leaving the bit-reverse shuffle as part of the DDA makes DFFT$_h$ more suitable as a building block DDA when creating compound DDAs (see Chapter 7.2.3).

Regarding the other hardware architectures, DFFT$_h$ can be easily embedded into any shared memory model architecture or Cuda STA by simply defining point projections from the FFT$_h$ into the given architecture's space and time coordinates.

6.4 The Sklansky Parallel Prefix Network

A *parallel prefix network* or *scan* commonly refers to a parallel implementation of the *all-prefix-sums* operation which given a sequence $x_0, x_1, \ldots x_{n-1}$ calculates the sums of the prefixes, i.e., it outputs the sequence $y_0, y_1, \ldots y_{n-1}$ such that $y_i = x_0 \circ x_1 \circ \ldots \circ x_i$ for all $0 \leq i \leq n-1$, where the *sum* operation $\circ$ apparently stands for any associative binary operator [Blelloch, 1990].

Parallel prefix networks are considered fundamental algorithms in computer science since they provide parallel implementations of an otherwise inherently sequential problem. They have many applications, for instance, polynomial evaluation, binary addition, string comparison, lexical analysis just to mention a few [Blelloch, 1990], and they also form important building blocks in modern microprocessors, for instance, in the implementation of fast adders. Therefore the search for efficient parallel prefix networks still plays a central role in circuit design [Sheeran, 2010].

The minimum possible depth of a parallel prefix network of width n is $\lceil \log n \rceil$. This bound is obtained by a usual divide and conquer pattern, and is usually attributed to [Sklansky, 1960] (see Fig. 6.11). The Sklansky construction recursively computes the parallel prefix sum for each half of its inputs, and then combines the last output of the left half with each output of the right half. The underlying dependency of this network is defined next.

Example 6.4.1. We define the *Sklansky Parallel Prefix Network DDA for* $2^{\mathtt{h}}$ *inputs*, $\mathtt{h} \in \mathbf{N}$, $\mathtt{DSK_h}$, as follows:

1. DDA sort: $\mathtt{SK_h}$ `= row Nat * col Nat`, with data invariant:

   `DI(p) = (row(p)<=h) && (col(p)<`$2^{\mathtt{h}}$`)`

2. branch indices: `B = {0,1,2,...,`$2^{\mathtt{h}-1}$`}`

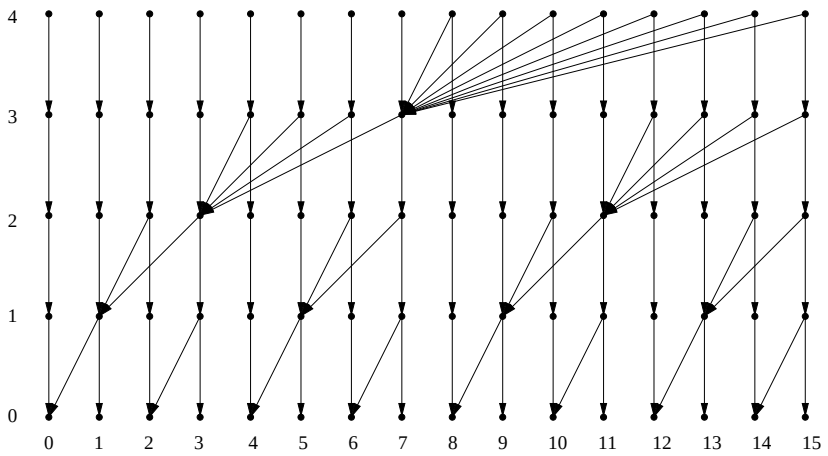

FIGURE 6.11: *Sklansky parallel prefix network DDA for* 2^4 *inputs.*

3. request components (rg,rp,rb) where:

```
rg(p,b) = (row(p)>0) && (b<=1) && ( isSum(p) || (b=0) )

rp(p,b) =
  if (b=0) SK_h(row(p)-1,col(p))
  else SK_h(row(p)-1, (col(p)/2^row(p))*2^row(p) + 2^(row(p)-1) -1)

rb(p,b) = if (b=1) col(p)-col(rp(p,b))
          else 0
```

4. supply components (sg,sp,sb) where:

```
sg(p,b) = (row(p)<h) && (b<=2^row(p)) &&
          (isBcast(p) || (b=0))

sp(p,b) = if (b=0) SK_h(row(p)+1,col(p))
          else SK_h(row(p)+1,col(p)+b)

sb(p,b) = if (b=0) 0 else 1
```

where $\texttt{isSum:SK}_h \rightarrow \texttt{bool}$ and $\texttt{isBcast:SK}_h \rightarrow \texttt{bool}$ are boolean functions defined by:

```
isSum(p) = (2^(row(p)-1) <= col(p)%2^row(p) < 2^row(p))

isBcast(p) = (col(p) = (col(p)/2^(row(p)+1))*2^(row(p)+1))+2^row(p) -1
```

□

All vertical arcs are denoted by 0 at both ends. Branch indices used for request directions are either 0 or 1. A point has two request directions only when isSum(p) is true. This means that a sum operation is to be performed at the point. Otherwise, the point has only one request direction denoted by branch index 0. Then the computation to be performed at the point is equivalent to the identity function. Supply directions are denoted by branch indices: $0, 1, \ldots, 2^{\texttt{row(p)}}$ for a given point $\texttt{p:SK}_h$. However, branch indices $1, \ldots, 2^{\texttt{row(p)}}$ will only exist for those points which have to broadcast their computed values to exactly $2^{\texttt{row(p)}}+1$ points, i.e., whenever isBcast(p) is true. Otherwise p only broadcasts along branch index 0 upward.

Let V be an array indexed by SK_h and some element type E. Assume that all input values have been assigned, that is, $V[SK_h(0,i)]=x_i$ for all $i \in \{0,1,\ldots,2^h-1\}$. Further, let $TP:SK_h \rightarrow Bool$ be a target for SK_h defined by TP(p) = (row(p)=h). Then the parallel prefix sums will be computed by the following targeted repeat statement such that $y_i=V[SK_h(h,i)]$ for all $i \in \{0,1,\ldots,2^h-1\}$:

```
repeat p:SK_h along DSK_h from V for TP in
    if isSum(p) V[rp(p,0)]+V[rp(p,1)]
    else V[rp(p,0)]
```

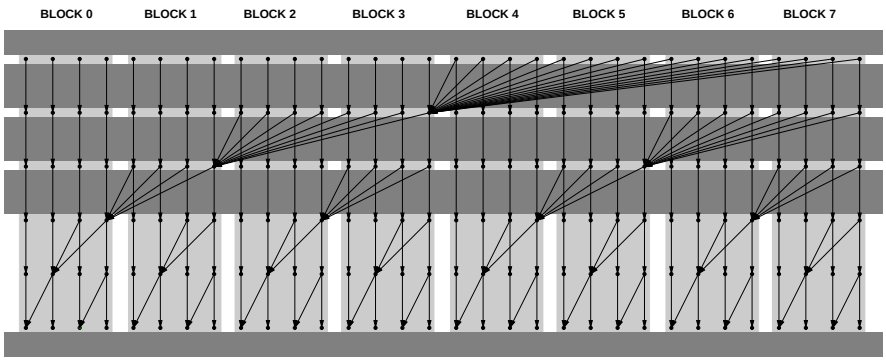

FIGURE 6.12: *Sklansky parallel prefix network DDA for* 2^5 *inputs embedded into CUDA by* EP *for* T=4*, executed by* 4 *kernel invocations.*

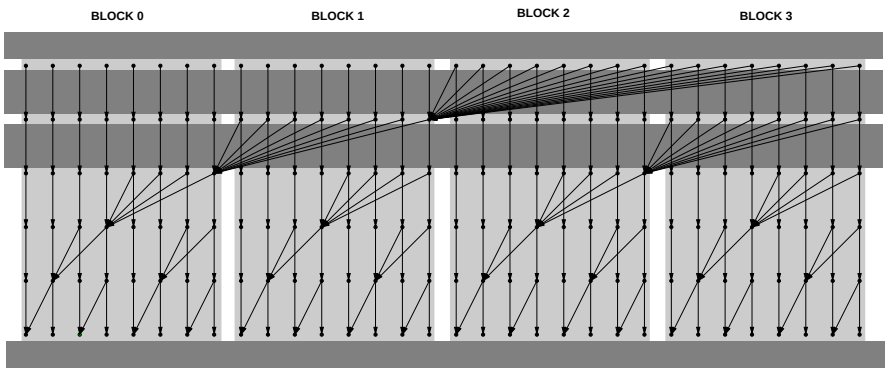

FIGURE 6.13: *Sklansky parallel prefix network DDA for* 2^5 *inputs embedded into CUDA by* EP *for* T=8*, executed by* 3 *kernel invocations.*

The Sklansky DDA can be easily embedded into any shared memory model architecture or Cuda STA by defining point projections from the $\mathtt{SK}_\mathtt{h}$ into the given architecture's space and time coordinates. For Cuda this will become for all $\mathtt{p}:\mathtt{SK}_\mathtt{h}$:

$$\mathtt{EP(p)} = \mathtt{CUST}_{\mathtt{B,T}}(\mathtt{CUB}_{\mathtt{B,T}}(\mathtt{col(p)/T,col(p)\%T}),\mathtt{row(p)})$$

Depending on the choice of T, the number of kernel invocations determined by the kernel scheduler will vary. Fig. 6.12 and 6.13 illustrate the embeddings defined by EP for different choices of T.

6.5 Tools

Graphical tools that make data dependencies evident are essential in parallel processing. We built a small visualization tool that taking a DDA-implementation, generates either xFig or Matlab descriptions, targeting 2D or 3D visualizations, respectively.

All DDA-examples presented here have been coded in C and tested by the visualization tool, which has been also used to generate the underlying DDA-drawings. The benefits of the visualization tool are three-folded:

1. *it tests the correctness of the DDA-implementation*: as a straightforward consequence of the symmetric properties of the DDA-axioms, a DDA-drawing generated via the supply directions should be identical with the layout generated via the request directions.

2. *it helps to define embeddings into hardware space-time coordinates*: for instance, in case of CUDA, it helps in choosing the right size parameters for the GPU kernel so that the number of kernel invocations may be optimal. Likewise, for the same hardware, different embedding projections can instantly be visualized and compared with each other.

3. *it gives an intuition about what is happening at a large scale*: since the number of inputs can be arbitrarily modified in the tool, one can inspect the dependency for larger problem sizes, and more over, with various embedding projections. This may provide insight, for instance, into how to achieve some efficiency goals in trying to avoid certain kinds of network congestion, e.g., see Fig. 6.14.

6.6 Experiments

The DDA-based implementations of the computational mechanisms instantiated for the examples of this chapter have been primarily handcoded. They followed request based execution models optimised for shared memory usage and targeting sequential and CUDA architectures. These experiments provided us with the insight to formalise the execution models devised for the general cases presented in Chapter 5.

It is beyond the purpose of this dissertation to evaluate the efficiency of the dependency-driven implementations against handcoded parallelised versions known from the literature. This surely will become a task when a fully working DDA-based compiler will be in place. Our focus, for the moment, has been primarily portability, and efficiency issues were addressed within the "constraints" of portability.

We present now some running time comparisons for the bitonic sort. The results were collected by comparing two dependency-driven bitonic sorters based on the bitonic sort dependency and the corresponding expression described in Section 6.1. The first implementation was a sequential C program, and the second a parallel CUDA program targeting Nvidia GPUs. The bitonic sort dependency was implemented in a separate module (C header file), so that both implementations could use the same module[2].

The sequential DDA-based bitonic sorter was tested on an AMD Athlon 64 X2 Dual Core processor against the classic divide-and-conquer implementation of bitonic sort, see Fig. 6.15.a. Experiments show that the dependency-driven computation is 1.4 times slower than the classic one. This should not come as a surprise, if we think about the machinery that

[2]In case of the CUDA-implementation CUDA-specific keywords needed to be inserted in front of the function declarations, but the rest of the module remained intact.

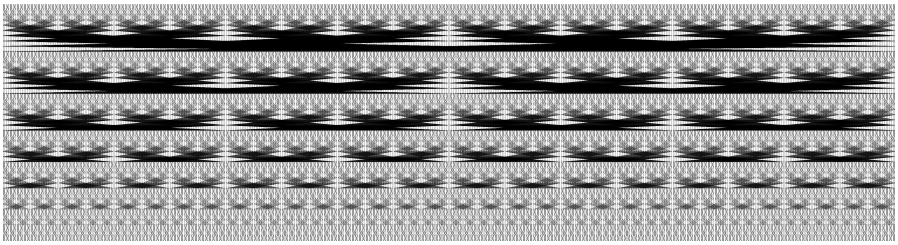

Figure 6.14: *Generated bitonic sort dependency for* 512 *inputs.*

drives the computation (various DDA function calls, projection-, constructor-calculations, etc. at every step). A rigorous profiling could probably help to optimise the DDA-module, but we did not look into this closely.

On the other hand, the DDA-abstraction made it pretty easy to port the dependency-driven bitonic sorter to CUDA. We compared the sequential DDA-based bitonic sorter against the CUDA implementation. The latter was tested on the above mentioned CPU together with Nvidia GeForce 9600 GT and Nvidia GeForce GTS 250, respectively. Fig. 6.15.b summarises the results over these platforms. The speedup when going parallel is significant: the parallel version runs $7 - 11$ times faster than the sequential version.

While our dependency-driven CUDA-implementation compared to fine-tuned, sophisticated, and in most cases hybrid GPU sorting algorithms [Govindaraju et al., 2006; Satish et al., 2009; Sintorn and Assarsson, 2008] is probably less efficient, the experiments underpin that DDA-based programming is highly portable and flexible, making the approach promising for parallel computing.

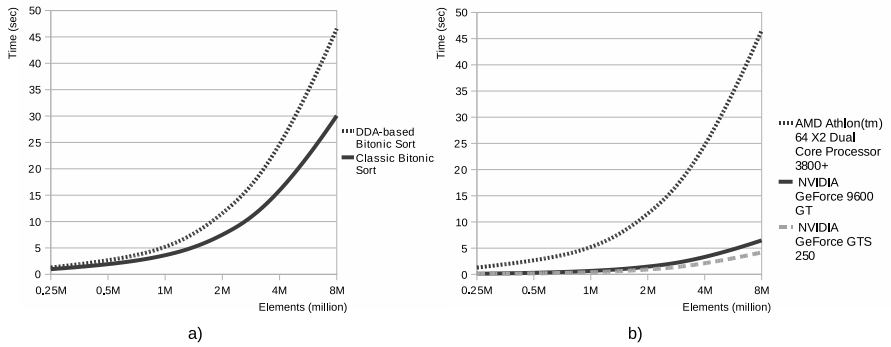

Figure 6.15: *a) DDA-based bitonic sort vs. classical bitonic sort running times on the same CPU. b) DDA-based bitonic sort running times on various platforms.*

CHAPTER 7

Algebraic Properties of DDAs

Haveraaen [2009] showed how complex DDAs can be built from simpler ones using various DDA-product-like constructors and investigated their properties. One of these products is the underlying construct for defining the space-time communication structure of a hardware as a space-time DDA (see Section 4.3). This can be seen as a construct of two DDAs: the hardware DDA (describing the hardware's static connectivity) projected over time (a linear-time DDA).

In this chapter we look into other ways of building DDAs, possibly more complex ones. It starts up with a formal presentation of various mechanisms that allow the building of compound DDAs from existing ones. Then it introduces some language constructs to denote their pragmatic counterparts for programming. The purpose is to allow the user to combine existing DDA implementations via high-level language constructs, and let the compiler generate the corresponding new compound DDA's implementation according to the formal definitions. We also discuss some aspects of this related to the repeat statement. Finally, we start on the investigation of a theory that allows the building of user-controlled tools which can identify situations when the combining of DDAs could result in optimized code.

7.1 DDA-Combinators

Some of the DDA-constructs discussed in this section are primarily inspired by the Lava serial and parallel composition combinators [Bjesse et al., 1999;

Claessen et al., 2003]. Lava is a domain specific languages embedded in the functional programming language Haskell. It is designed to help specify the layout of circuits in order to improve performance and reduce area utilization.

In the fashion of Lava combinators, DDAs can be either combined relative to each other with no communication, i.e. in parallel – along *space*, or connected serially with explicit communication – along *time*.

In addition to these, the sub-DDA-combinator and the nesting-DDA-combinator are presented. The first entails the selection of a sub-DDA of an existing DDA, the latter provide means for more elaborate constructions.

7.1.1 The Parallel DDA-Combinator

Intuitively, the parallel DDA-combinator allows two (possibly different) DDAs to be placed next to each other in space, thus resulting in a new, larger DDA. All points and all dependency arcs from both DDAs are preserved, no new points nor new dependency branches are being created.

Definition 7.1.1 (Parallel DDA-Combinator)**.** Let $\mathcal{D}^1 = \langle P^1, B^1, req^1, sup^1 \rangle$ and $\mathcal{D}^2 = \langle P^2, B^2, req^2, sup^2 \rangle$ be two DDAs. Then the *parallel combination of* $\mathcal{D}^1$ *and* $\mathcal{D}^2$ is a 4-tuple

$$\mathcal{D}^1 \parallel \mathcal{D}^2 \stackrel{Def}{=} \mathcal{P} = \langle P^{\mathcal{P}}, B^{\mathcal{P}}, req^{\mathcal{P}}, sup^{\mathcal{P}} \rangle$$

where:

- $P^{\mathcal{P}} = P^1 \uplus P^2$,
- $B^{\mathcal{P}} = B^1 \cup B^2$
- $req^{\mathcal{P}}$ consists of $(r_g^{\mathcal{P}}, r_p^{\mathcal{P}}, r_b^{\mathcal{P}})$ where:

$$\begin{aligned} r_g^{\mathcal{P}}(\langle n, i \rangle, d) &= (d \in B^i) \wedge r_g^i(n, d) \\ r_p^{\mathcal{P}}(\langle n, i \rangle, d) &= \langle r_p^i(n, d), i \rangle \\ r_b^{\mathcal{P}}(\langle n, i \rangle, d) &= r_b^i(n, d) \end{aligned}$$

- $sup^{\mathcal{P}}$ consists of $(s_g^{\mathcal{P}}, s_p^{\mathcal{P}}, s_b^{\mathcal{P}})$ where:

$$\begin{aligned} s_g^{\mathcal{P}}(\langle n, i \rangle, d) &= (d \in B^i) \wedge s_g^i(n, d) \\ s_p^{\mathcal{P}}(\langle n, i \rangle, d) &= \langle s_p^i(n, d), i \rangle \\ s_b^{\mathcal{P}}(\langle n, i \rangle, d) &= s_b^i(n, d) \end{aligned}$$

□

Theorem 7.1.2. The parallel combination of two DDAs as given in Def. 7.1.1 is a DDA.

Proof: The 4-tuple $\langle P^{\mathcal{P}}, B^{\mathcal{P}}, req^{\mathcal{P}}, sup^{\mathcal{P}} \rangle$ is a DDA, if it satisfies all DDA axioms (see Def. 4.1.1). Note that the new point set $P^{\mathcal{P}}$ is a disjoint union formed as the union of the canonical forms of sets P^1 and P^2. The second component of each point of $P^{\mathcal{P}}$ hence will distinguish which component DDA the point originally belonged to. This information is being explicitly used in the definition of the new request and supply components, so that the DDA axioms will be satisfied on these, as they are satisfied for the underlying building block DDAs. □

Proposition 7.1.3. The parallel DDA-combinator is associative.

Proof: Follows directly from the fact that the union and disjoint union of sets is associative (see Theorem 3.1.1), and that the latter is formed by the union of the participating sets' canonical forms. □

This property entails the generalization of this combinator which will allow the parallel combination of several DDAs in one step.

Definition 7.1.4 (k-ary Parallel DDA-Combinator)**.** Given k DDAs, $k \geq 0$, $\mathcal{D}^1 = \langle P^1, B^1, req^1, sup^1 \rangle, \ldots, \mathcal{D}^k = \langle P^k, B^k, req^k, sup^k \rangle$. Then the *parallel combination of* $\mathcal{D}^1, \ldots, \mathcal{D}^k$ is defined by

$$\mathcal{D}^1 \parallel \mathcal{D}^2 \ldots \parallel \mathcal{D}^k \stackrel{Def}{=} \mathcal{P}_k = \langle P^{\mathcal{P}_k}, B^{\mathcal{P}_k}, req^{\mathcal{P}_k}, sup^{\mathcal{P}_k} \rangle$$

where:

- $P^{\mathcal{P}_k} = \biguplus_{1 \leq i \leq k} P^i$
- $B^{\mathcal{P}_k} = \bigcup_{1 \leq i \leq k} B^i$
- $req^{\mathcal{P}_k}$ consists of $(r_g^{\mathcal{P}_k}, r_p^{\mathcal{P}_k}, r_b^{\mathcal{P}_k})$ where:

$$\begin{aligned} r_g^{\mathcal{P}_k}(\langle n, i \rangle, d) &= (d \in B^i) \wedge r_g^i(n, d) \\ r_p^{\mathcal{P}_k}(\langle n, i \rangle, d) &= \langle r_p^i(n, d), i \rangle \\ r_b^{\mathcal{P}_k}(\langle n, i \rangle, d) &= r_b^i(n, d) \end{aligned}$$

- $sup^{\mathcal{P}_k}$ consists of $(s_g^{\mathcal{P}_k}, s_p^{\mathcal{P}_k}, s_b^{\mathcal{P}_k})$ where:

$$\begin{aligned} s_g^{\mathcal{P}_k}(\langle n,i\rangle,d) &= (d \in B^i) \wedge s_g^i(n,d) \\ s_p^{\mathcal{P}_k}(\langle n,i\rangle,d) &= \langle s_p^i(n,d), i\rangle \\ s_b^{\mathcal{P}_k}(\langle n,i\rangle,d) &= s_b^i(n,d) \end{aligned}$$

□

Theorem 7.1.5. The parallel combination of k DDAs, $k \geq 0$ as given in Def. 7.1.4 is a DDA. □

Proof: Note that if $k = 0$ the data structure obtained in the above construction will be the empty DDA, which has the empty set as its set of points, set of branch indices, request and supply guards, and the empty function for the rest of the components of both request and supply.

If $k \geq 1$, then the DDA axioms will be satisfied for the components of $req^{\mathcal{P}_k}$ and $sup^{\mathcal{P}_k}$ as they are satisfied by assumption for the components of each building block DDA's req^i and sup^i in case. □

7.1.2 The Serial DDA-Combinator

While the parallel DDA-combinator does not establish new connections between the participating DDAs, the serial DDA-combinator is applied for two DDAs in order to make them *connected*. The process of connecting DDAs consists of the creation of new branches that will connect one DDA's points to the other DDA's points. This can be done in many different ways, however, one should avoid creating loops. The presence of loops in the compound DDA would result in a cyclic DDA, which is not suitable for defining space-time unfoldings and computation-related DDAs, and therefore would fall out from our application domain.

In this setting, intuitively, one DDA is placed on the top of an other DDA and some designated points of the lower DDA will supply data to some designated points of the upper DDA along new branches. These are determined by a given transfer function. Depending on the properties of the transfer function two kinds of serial combinators will be introduced. First we look at the simpler but less general one.

Definition 7.1.6 (Bijective Serial DDA-Combinator)**.** Given two DDAs $\mathcal{D}^1 = \langle P^1, B^1, req^1, sup^1\rangle$ and $\mathcal{D}^2 = \langle P^2, B^2, req^2, sup^2\rangle$, and $t : I \to O$ a bijection, where $I \subseteq P^2$ and $O \subseteq P^1$. Then the *serial combination of D^1 and D^2 along bijection t*,

$$\mathcal{D}^1 \overset{t}{\hookleftarrow} \mathcal{D}^2 \overset{Def}{=} \mathcal{S}_b = \langle P^{\mathcal{S}_b}, B^{\mathcal{S}_b}, req^{\mathcal{S}_b}, sup^{\mathcal{S}_b}\rangle$$

is defined by:

- $P^{\mathcal{S}_b} = P^1 \uplus P^2$
- $B^{\mathcal{S}_b} = (B^1 \cup B^2) \uplus \{c\}$ for some c
- $req^{\mathcal{S}_b}$ consists of $(r_g^{\mathcal{S}_b}, r_p^{\mathcal{S}_b}, r_b^{\mathcal{S}_b})$ where:

$$\begin{aligned}
r_g^{\mathcal{S}_b}(\langle n,i\rangle,\langle d,j\rangle) &= \big((j=1)\wedge(d\in B^i)\wedge r_g^i(n,d)\big)\vee \\
&\quad \big((j=2)\wedge(i=2)\wedge(n\in I)\big) \\
r_p^{\mathcal{S}_b}(\langle n,i\rangle,\langle d,j\rangle) &= \begin{cases} \langle r_p^i(n,d),i\rangle & \text{if} \quad j=1 \\ \langle t(n),1\rangle & \text{if} \quad j=2 \end{cases} \\
r_b^{\mathcal{S}_b}(\langle n,i\rangle,\langle d,j\rangle) &= \begin{cases} \langle r_b^i(n,d),1\rangle & \text{if} \quad j=1 \\ \langle c,2\rangle & \text{if} \quad j=2 \end{cases}
\end{aligned}$$

- $sup^{\mathcal{S}_b}$ consists of $(s_g^{\mathcal{S}_b}, s_p^{\mathcal{S}_b}, s_b^{\mathcal{S}_b})$ where:

$$\begin{aligned}
s_g^{\mathcal{S}_b}(\langle n,i\rangle,\langle d,j\rangle) &= \big((j=1)\wedge(d\in B^i)\wedge s_g^i(n,d)\big)\vee \\
&\quad \big((j=2)\wedge(i=1)\wedge(n\in O)\big) \\
s_p^{\mathcal{S}_b}(\langle n,i\rangle,\langle d,j\rangle) &= \begin{cases} \langle s_p^i(n,d),i\rangle & \text{if} \quad j=1 \\ \langle t^{-1}(n),2\rangle & \text{if} \quad j=2 \end{cases} \\
s_b^{\mathcal{S}_b}(\langle n,i\rangle,\langle d,j\rangle) &= \begin{cases} \langle s_b^i(n,d),1\rangle & \text{if} \quad j=1 \\ \langle c,2\rangle & \text{if} \quad j=2 \end{cases}
\end{aligned}$$

  where $t^{-1} : O \to I$ is the inverse of t.

□

Theorem 7.1.7. The serial combination of D^1 and D^2 along the bijection t as given in Definition 7.1.6 is a DDA. □

Proof: A formal proof showing that $\mathcal{S}_b$ satisfies all DDA axioms follows a similar pattern provided for Theorem 7.1.9 when showing that the general serial DDA-combinator defines a DDA. □

Fig. 7.1 illustrates the use of new branch indices: $\langle c, 2\rangle$ identifies a new supply direction in DDA $\mathcal{D}^1$ for all points in O, and a new request direction in DDA $\mathcal{D}^2$ for all points in I.

The requirement imposed on t, i.e. to be a bijection, comes in very handy, as the existence of t^{-1} makes the definition of supply well-defined.

However, this requirement might prove be too strong for the general cases. The DDA sizes may differ, therefore it would be natural to be able

to combine DDAs even when $|O| \neq |I|$, or when we want to create several new supply directions from some point of O. (See Fig. 7.2.) In such cases, it is sufficient to require t to only be a total function instead of a bijection. For each point $n \in I$ a new branch index is introduced to identify the corresponding supply direction that leads back to the point n from $t(n)$.

The bijective serial DDA-combinator (Definition 7.1.6) becomes a simplified version of the general serial DDA-combinator which is defined next.

Definition 7.1.8 (Serial DDA-Combinator)**.** Let $\mathcal{D}^1 = \langle P^1, B^1, req^1, sup^1 \rangle$ and $\mathcal{D}^2 = \langle P^2, B^2, req^2, sup^2 \rangle$ be two DDAs, and $t : I \to O$ a total function where $I \subseteq P^2$ and $O \subseteq P^1$. Then the *serial combination of D^1 and D^2 along the (total) function t,*

$$\mathcal{D}^1 \overset{t}{\mapsto} \mathcal{D}^2 \overset{Def}{=} \mathcal{S}_t = \langle P^{\mathcal{S}_t}, B^{\mathcal{S}_t}, req^{\mathcal{S}_t}, sup^{\mathcal{S}_t} \rangle$$

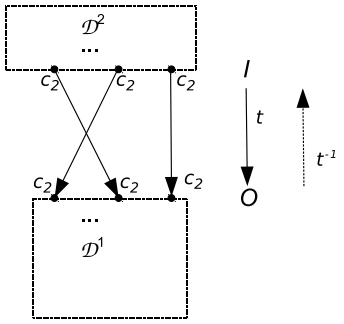

Figure 7.1: *Serial combination of DDAs $\mathcal{D}^1$ and $\mathcal{D}^2$ along the bijection t.*

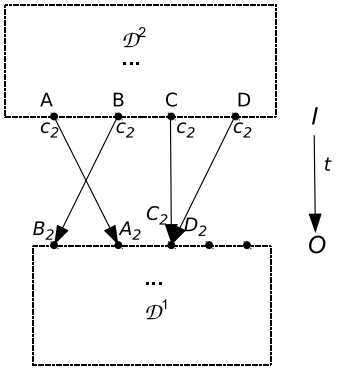

Figure 7.2: *Serial combination of DDAs $\mathcal{D}^1$ and $\mathcal{D}^2$ along a total function t.*

is defined by:

- $P^{\mathcal{S}_t} = P^1 \uplus P^2$
- $B^{\mathcal{S}_t} = (B^1 \cup B^2) \uplus \big(\{c\} \cup \{n \mid n \in I\}\big)$ for some c
- $req^{\mathcal{S}_t}$ consists of $(r_g^{\mathcal{S}_t}, r_p^{\mathcal{S}_t}, r_b^{\mathcal{S}_t})$ where:

$$\begin{array}{rcl} r_g^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle) & = & \big((j=1)\wedge(d\in B^i)\wedge r_g^i(n,d)\big)\vee \\ & & \big((j=2)\wedge(i=2)\wedge(d=c)\wedge(n\in I)\big) \\ r_p^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle) & = & \begin{cases} \langle r_p^i(n,d),i\rangle & \text{if} \quad j=1 \\ \langle t(n),1\rangle & \text{if} \quad j=2 \end{cases} \\ r_b^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle) & = & \begin{cases} \langle r_b^i(n,d),1\rangle & \text{if} \quad j=1 \\ \langle n,2\rangle & \text{if} \quad j=2 \end{cases} \end{array}$$

- $sup^{\mathcal{S}_t}$ consists of $(s_g^{\mathcal{S}_t}, s_p^{\mathcal{S}_t}, s_b^{\mathcal{S}_t})$ where:

$$\begin{array}{rcl} s_g^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle) & = & \big((j=1)\wedge(d\in B^i)\wedge s_g^i(n,d)\big)\vee \\ & & \big((j=2)\wedge(i=1)\wedge(d\in I)\wedge(t(d)=n)\big) \\ s_p^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle) & = & \begin{cases} \langle s_p^i(n,d),i\rangle & \text{if} \quad j=1 \\ \langle d,2\rangle & \text{if} \quad j=2 \end{cases} \\ s_b^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle) & = & \begin{cases} \langle s_b^i(n,d),1\rangle & \text{if} \quad j=1 \\ \langle c,2\rangle & \text{if} \quad j=2 \end{cases} \end{array}$$

□

Theorem 7.1.9. The serial combination of D^1 and D^2 along a total function t as given in Definition 7.1.8 is a DDA. □

Proof: To ensure that $\mathcal{S}_t$ is a DDA we need to verify whether its $req^{\mathcal{S}_t}$ and $sup^{\mathcal{S}_t}$ components satisfy all DDA axioms. We show this for the supplies, i.e., $sup^{\mathcal{S}_t}$. The proof for the requests, $req^{\mathcal{S}_t}$, is similar.

Assuming that $r_g^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle)$ holds, it is either the case that 1) $(j = 1) \wedge (d \in B^i) \wedge r_g^i(n,d)$ or 2) $(j = 2) \wedge (i = 2) \wedge (d = c) \wedge (n \in I)$ holds. We distinguish therefore these two cases:

1. Under the assumption that $(j = 1) \wedge (d \in B^i) \wedge r_g^i(n,d)$ holds we get:

$$r_p^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle) = \langle r_p^i(n,d),i\rangle \tag{7.1}$$

$$r_b^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle) = \langle r_b^i(n,d),1\rangle \tag{7.2}$$

This leads to:

$$
\begin{aligned}
& s_g^{\mathcal{S}_t}\big(r_p^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle), r_b^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle)\big) = && (7.3)\\
& \quad = s_g^{\mathcal{S}_t}(\langle r_p^i(n,d),i\rangle,\langle r_b^i(n,d),1\rangle) && \text{(by (7.1) and (7.2))}\\
& \quad = s_g^i(r_p^i(n,d), r_b^i(n,d)) && \text{(by def. of } s_g^{\mathcal{S}_t} \text{ and ass.)}
\end{aligned}
$$

which holds given that $r_g^i(n,d)$ holds and that $\mathcal{D}^i$ is a DDA.

Then (7.3) also holds and this makes the remaning axioms for the supplies hold:

$$
\begin{aligned}
& s_p^{\mathcal{S}_t}\big(r_p^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle), r_b^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle)\big) = \\
& \quad = s_p^{\mathcal{S}_t}(\langle r_p^i(n,d),i\rangle,\langle r_b^i(n,d),1\rangle) && \text{(by (7.1) and (7.2))}\\
& \quad = \langle s_p^i(r_p^i(n,d), r_b^i(n,d)),i\rangle && \text{(by def. of } s_p^{\mathcal{S}_t} \text{ and ass.)}\\
& \quad = \langle n,i\rangle && \text{(as } \mathcal{D}^{\mathrm{i}} \text{ is a DDA)}
\end{aligned}
$$

$$
\begin{aligned}
& s_b^{\mathcal{S}_t}\big(r_p^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle), r_b^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle)\big) = \\
& \quad = s_b^{\mathcal{S}_t}(\langle r_p^i(n,d),i\rangle,\langle r_b^i(n,d),1\rangle) && \text{(by (7.1) and (7.2))}\\
& \quad = \langle s_b^i(r_p^i(n,d), r_b^i(n,d)),1\rangle && \text{(by def. of } s_b^{\mathcal{S}_t} \text{ and ass.)}\\
& \quad = \langle d,1\rangle && \text{(as } \mathcal{D}^{\mathrm{i}} \text{ is a DDA)}\\
& \quad = \langle d,j\rangle && \text{(by ass.)}
\end{aligned}
$$

2. Under the assumption that $(j=2)\wedge(i=2)\wedge(d=c)\wedge(n\in I)$ holds we get:

$$
\begin{aligned}
r_p^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle) &= r_p^{\mathcal{S}_t}(\langle n,2\rangle,\langle c,2\rangle) && \text{(by ass.)}\\
&= \langle t(n),1\rangle && \text{(by def. of } r_p^{\mathcal{S}_t}) \qquad (7.4)\\
r_b^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle) &= r_b^{\mathcal{S}_t}(\langle n,2\rangle,\langle c,2\rangle) && \text{(by ass.)}\\
&= \langle n,2\rangle && \text{(by def. of } r_b^{\mathcal{S}_t}) \qquad (7.5)
\end{aligned}
$$

This leads to:

$$
\begin{aligned}
& s_g^{\mathcal{S}_t}\big(r_p^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle), r_b^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle)\big) = && (7.6)\\
& \quad = s_g^{\mathcal{S}_t}(\langle t(n),1\rangle,\langle n,2\rangle) && \text{(by (7.4) and (7.5))}
\end{aligned}
$$

which holds by the definition of $s_g^{\mathcal{S}_t}$ and our assumption. Hence (7.6) also holds, and leads to the rest of the supply axioms to hold as follows:

$$\begin{aligned} & s_p^{\mathcal{S}_t}\left(r_p^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle), r_b^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle)\right) = \\ & \quad = s_p^{\mathcal{S}_t}(\langle t(n),1\rangle,\langle n,2\rangle) && \text{(by (7.4) and (7.5))} \\ & \quad = \langle n,2\rangle && \text{(by def. of } s_p^{\mathcal{S}_t}\text{)} \\ & \quad = \langle n,i\rangle && \text{(by ass.)} \end{aligned}$$

$$\begin{aligned} & s_b^{\mathcal{S}_t}\left(r_p^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle), r_b^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle)\right) = \\ & \quad = s_b^{\mathcal{S}_t}(\langle t(n),1\rangle,\langle n,2\rangle) && \text{(by (7.4) and (7.5))} \\ & \quad = \langle c,2\rangle && \text{(by def. of } s_b^{\mathcal{S}_t}\text{)} \\ & \quad = \langle d,j\rangle && \text{(by ass.)} \end{aligned}$$

□

We show next that the DDA obtained by the serial combination of two DDAs along a bijection is isomorphic to the DDA obtained by the general serial combinator when the total function has the additional property of being a bijection.

Theorem 7.1.10. Given $\mathcal{D}^1 = \langle P^1, B^1, req^1, sup^1\rangle$ and $\mathcal{D}^2 = \langle P^2, B^2, req^2, sup^2\rangle$ two DDAs, and $t : I \to O$ a bijection where $I \subseteq P^2$ and $O \subseteq P^1$. Then:

$$\mathcal{D}^1 \overset{t}{\leftrightarrow} \mathcal{D}^2 \simeq \mathcal{D}^1 \overset{t}{\mapsto} \mathcal{D}^2$$

□

Proof: By the previously introduced notations, we need to show that $\mathcal{S}_b \simeq \mathcal{S}_t$. First note that $P^{\mathcal{S}_b} = P^{\mathcal{S}_t}$, hence they are isomorphic. Following the definition of isomorphic DDAs (Definition 4.1.10), we need to define then isomorphisms $\iota_{r_g} : r_g^{\mathcal{S}_b} \to r_g^{\mathcal{S}_t}$ with inverse $\iota_{r_g}^{-1} : r_g^{\mathcal{S}_t} \to r_g^{\mathcal{S}_b}$ and $\iota_{s_g} : s_g^{\mathcal{S}_b} \to s_g^{\mathcal{S}_t}$ with inverse $\iota_{s_g}^{-1} : s_g^{\mathcal{S}_t} \to s_g^{\mathcal{S}_b}$ that preserve the points and which satisfy $\tilde{r}^{\mathcal{S}_b} = \iota_{s_g}^{-1} \circ \tilde{r}^{\mathcal{S}_t} \circ \iota_{r_g}$.

Note that $r_g^{\mathcal{S}_b} = r_g^{\mathcal{S}_t}$, hence we define

$$\iota_{r_g} = \iota_{r_g}^{-1} = \mathbf{1}_{r_g^{\mathcal{S}_b}} \tag{7.7}$$

We define ι_{s_g} as follows:

$$\iota_{s_g}(\langle n, i\rangle, \langle d, j\rangle) = \begin{cases} \langle\langle n, i\rangle, \langle d, j\rangle\rangle & \text{if } j = 1 \\ \langle\langle n, i\rangle, \langle t^{-1}(n), j\rangle\rangle & \text{if } j = 2 \end{cases}$$

and $\iota_{s_g}^{-1}$ as follows:

$$\iota_{s_g}^{-1}(\langle n, i\rangle, \langle d, j\rangle) = \begin{cases} \langle\langle n, i\rangle, \langle d, j\rangle\rangle & \text{if } j = 1 \\ \langle\langle n, i\rangle, \langle c, j\rangle\rangle & \text{if } j = 2 \end{cases}$$

Note that when $j = 1$ it is straightforward that $\iota_{s_g} \circ \iota_{s_g}^{-1} = \mathbf{1}_{s_g^{\mathcal{S}_t}}$ and $\iota_{s_g}^{-1} \circ \iota_{s_g} = \mathbf{1}_{s_g^{\mathcal{S}_b}}$. We show that these hold also when $j = 2$.

Consider $(\iota_{s_g} \circ \iota_{s_g}^{-1})(\langle n, i\rangle, \langle d, j\rangle)$ and note that $t(d) = n$, $i = 1$ and $j = 2$ by $s_g^{\mathcal{S}_t}$. Hence we have:

$$\begin{aligned} \iota_{s_g}(\iota_{s_g}^{-1}(\langle n, 1\rangle, \langle d, 2\rangle)) &= \iota_{s_g}(\langle n, 1\rangle, \langle c, 2\rangle) && \text{(def. of } \iota_{s_g}^{-1}) \\ &= \langle\langle n, 1\rangle, \langle t^{-1}(n), 2\rangle\rangle && \text{(def. of } \iota_{s_g}) \\ &= \langle\langle n, 1\rangle, \langle d, 2\rangle\rangle && (t \text{ is bij. and } t(d) = n) \\ &= \mathbf{1}_{s_g^{\mathcal{S}_t}}(\langle n, 1\rangle, \langle d, 2\rangle) \end{aligned}$$

Likewise consider $(\iota_{s_g}^{-1} \circ \iota_{s_g})(\langle n, i\rangle, \langle d, j\rangle)$ and note that $i = 1$ and $j = 2$ by $s_g^{\mathcal{S}_b}$, and then $d = c$ by the definition of $B^{\mathcal{S}_b}$. Hence we have:

$$\begin{aligned} \iota_{s_g}^{-1}(\iota_{s_g}(\langle n, 1\rangle, \langle c, 2\rangle)) &= \iota_{s_g}(\langle n, 1\rangle, \langle t^{-1}(n), 2\rangle) && \text{(def. of } \iota_{s_g}) \\ &= \langle\langle n, 1\rangle, \langle c, 2\rangle\rangle && \text{(def. of } \iota_{s_g}^{-1}) \\ &= \mathbf{1}_{s_g^{\mathcal{S}_b}}(\langle n, 1\rangle, \langle c, 2\rangle) \end{aligned}$$

Hence the isomorphisms and their inverses exist. Finally, we need to check whether the isomorphism condition holds. By (7.7), this boils down to: $\tilde{r}^{\mathcal{S}_b} = \iota_{s_g}^{-1} \circ \tilde{r}^{\mathcal{S}_t}$, that is:

$$\begin{aligned}
&\langle r_p^{\mathcal{S}_b}(\langle n,i\rangle,\langle d,j\rangle), r_b^{\mathcal{S}_b}(\langle n,i\rangle,\langle d,j\rangle)\rangle = \\
&= \iota_{s_g}^{-1}(r_p^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle), r_b^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle) \\
&= \langle \iota_{s_g,p}^{-1}(r_p^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle), r_b^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle), \iota_{s_g,b}^{-1}(r_p^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle), r_b^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle)\rangle
\end{aligned}$$

where by $\iota_{s_g,p}^{-1}$ and $\iota_{s_g,b}^{-1}$ we denote the first and the second component of $\iota_{s_g}^{-1}$, respectively. But this holds, as the equality holds component-wise:

1. If $j = 1$ we have $r_p^{\mathcal{S}_b}(\langle n,i\rangle,\langle d,j\rangle) = \langle r_p^i(n,d), i\rangle$ and

$$\begin{aligned}
&\iota_{s_g,p}^{-1}(r_p^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle), r_b^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle) = \\
&\quad = \iota_{s_g,p}^{-1}(\langle r_p^i(n,d), i\rangle, \langle r_b^i(n,d), 1\rangle) && \text{(by def. of } r_p^{\mathcal{S}_t}, r_b^{\mathcal{S}_t}) \\
&\quad = \langle r_p^i(n,d), i\rangle && \text{(first comp. of } \iota_{s_g}^{-1})
\end{aligned}$$

   If $j = 2$ we have $r_p^{\mathcal{S}_b}(\langle n,i\rangle,\langle d,j\rangle) = \langle t(n), 1\rangle$ and

$$\begin{aligned}
&\iota_{s_g,p}^{-1}(r_p^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle), r_b^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle) = \\
&\quad = \iota_{s_g,p}^{-1}(\langle t(n), 1\rangle, \langle n, 2\rangle) && \text{(by def. of } r_p^{\mathcal{S}_t}, r_b^{\mathcal{S}_t}) \\
&\quad = \langle t(n), 1\rangle && \text{(first comp. of } \iota_{s_g}^{-1})
\end{aligned}$$

   Hence $r_p^{\mathcal{S}_b}(\langle n,i\rangle,\langle d,j\rangle) = \iota_{s_g,p}^{-1}(r_p^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle), r_b^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle)$.

2. If $j = 1$ we have $r_b^{\mathcal{S}_b}(\langle n,i\rangle,\langle d,j\rangle) = \langle r_b^i(n,d), 1\rangle$ and

$$\begin{aligned}
&\iota_{s_g,b}^{-1}(r_p^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle), r_b^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle) = \\
&\quad = \iota_{s_g,p}^{-1}(\langle r_p^i(n,d), i\rangle, \langle r_b^i(n,d), 1\rangle) && \text{(by def. of } r_p^{\mathcal{S}_t}, r_b^{\mathcal{S}_t}) \\
&\quad = \langle r_b^i(n,d), 1\rangle && \text{(sec. comp. of } \iota_{s_g}^{-1})
\end{aligned}$$

   If $j = 2$ we have $r_b^{\mathcal{S}_b}(\langle n,i\rangle,\langle d,j\rangle) = \langle c, 2\rangle$ and

$$\begin{aligned}
&\iota^{-1}_{s_g,b}(r_p^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle)), r_b^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle)) = \\
&\quad = \iota^{-1}_{s_g,b}(\langle t(n),1\rangle,\langle n,2\rangle) && \text{(by def. of } r_p^{\mathcal{S}_t}, r_b^{\mathcal{S}_t}) \\
&\quad = \langle c,2\rangle && \text{(sec. comp. of } \iota^{-1}_{s_g})
\end{aligned}$$

Hence $r_b^{\mathcal{S}_b}(\langle n,i\rangle,\langle d,j\rangle) = \iota^{-1}_{s_g,b}(r_p^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle)), r_b^{\mathcal{S}_t}(\langle n,i\rangle,\langle d,j\rangle))$ as well.

Then the isomorphism condition is satisfied, and hence $\mathcal{S}_b \simeq \mathcal{S}_t$ when t is bijective. □

7.1.3 *The Sub-DDA-Combinator*

In Section 4.1.3 we have introduced the notion of sub-DDA, and have seen an example of this in Example 4.2.4 through the manipulation of data invariant.

We define now the sub-DDA-combinator based on a subset of the point set and a subset of the branch indices of an existing DDA. The construction leads to a sub-DDA.

Definition 7.1.11 (Sub-DDA-Combinator)**.** Given a DDA $\mathcal{D} = \langle P, B, req, sup\rangle$ with $P' \subseteq P$, $B' \subseteq B$ two sets. Then the *sub-DDA-combinator along P' and B' applied for $\mathcal{D}$* is given by:

$$\mathcal{D}\big|_{P',B'} \stackrel{Def}{=} \mathcal{D}' = \langle P', B', req', sup'\rangle$$

where:

- req' consists of:

$$\begin{aligned}
r'_g(n,d) &= r_g(n,d) \wedge (n \in P') \wedge (d \in B') \wedge \\
&\quad (r_p(n,d) \in P') \wedge (r_b(n,d) \in B') \\
r'_p &= r_p\big|_{r'_g} \\
r'_b &= r_b\big|_{r'_g}
\end{aligned}$$

- sup' consists of:

$$\begin{aligned} s'_g(n,d) &= s_g(n,d) \wedge (n \in P') \wedge (d \in B') \wedge \\ &\quad (s_p(n,d) \in P') \wedge (s_b(n,d) \in B') \\ s'_p &= s_p\big|_{s'_g} \\ s'_b &= s_b\big|_{s'_g} \end{aligned}$$

□

Theorem 7.1.12. If $\mathcal{D} = \langle P, B, req, sup \rangle$ is a DDA and $P' \subseteq P$, $B' \subseteq B$ are two sets, then:

$$\mathcal{D}\big|_{P',B'} \subseteq \mathcal{D}$$

□

Proof: First note that r'_p is a function with P' as codomain, i.e., $r'_p : r'_g \to P'$. This is guaranteed by the way r'_g is defined and that $r'_p(n,d) = r_p(n,d)$ which, with $(n,d) \in r'_g$, can only lead to points in P'. Anything that would lead to $P \setminus P'$ is thrown away in r'_g. The same arguments apply for the following: $r'_b : r'_g \to B'$, $s'_p : s'_g \to P'$ and $s'_b : s'_g \to B'$. Second, all DDA axioms will be satisfied by the components of req' and sup', as r'_p, r'_b, s'_p and s'_b are defined exactly as the corresponding components of $\mathcal{D}$. Hence $\mathcal{D}\big|_{P',B'}$ is a DDA.

To see that $\mathcal{D}\big|_{P',B'}$ is in effect a sub-DDA of $\mathcal{D}$ (see Definition 4.1.8), note that $r'_g \subseteq r_g$ and that:

$$\tilde{r}' = \langle r'_p, r'_b \rangle = \langle r_p\big|_{r'_g}, r_b\big|_{r'_g} \rangle = \tilde{r}\big|_{r'_g}$$

□

What the sub-DDA-combinator does is in fact to omit all points of P and all branch indices of B that are not in P' and B', respectively, and omit all dependency arcs which do not lead to or start from P'. Everything else of the original DDA is otherwise kept, hence the sub-DDA-combinator supports code reusability.

7.1.4 The Nesting-DDA-Combinator

In Section 6.1 we have defined the bitonic sort DDA. As pointed out, this dependency pattern consists of the repetitive combination of several butterfly dependency patterns of increasing size. The repetitive combination itself,

however, follows a dependency pattern, i.e., the binary tree dependency. This is due to the divide and conquer nature of the algorithm. While the *divide* part is responsible for the global binary tree dependency, the *conquer* part yields each sub-butterfly dependency. A similar decomposition can be identified in the other divide and conquer based odd-even merge sort dependency, discussed in Section 6.2. There the global dependency is again the binary tree.

This naturally raises the idea of *nesting* DDAs: given a global DDA pattern and a bunch of local DDAs, such that each of these is associated with one or more points of the global DDA, how can we obtain a corresponding standalone large DDA. The following definition proposes a formalism for this.

Definition 7.1.13 (Nested DDAs)**.** Let $\mathcal{G} = \langle P^{\mathcal{G}}, B^{\mathcal{G}}, req^{\mathcal{G}}, sup^{\mathcal{G}} \rangle$ be an acyclic, well-founded (global) DDA such that $I^{\mathcal{G}} \subset P^{\mathcal{G}}$, $I^{\mathcal{G}} \neq \{\}$ is the subset of all

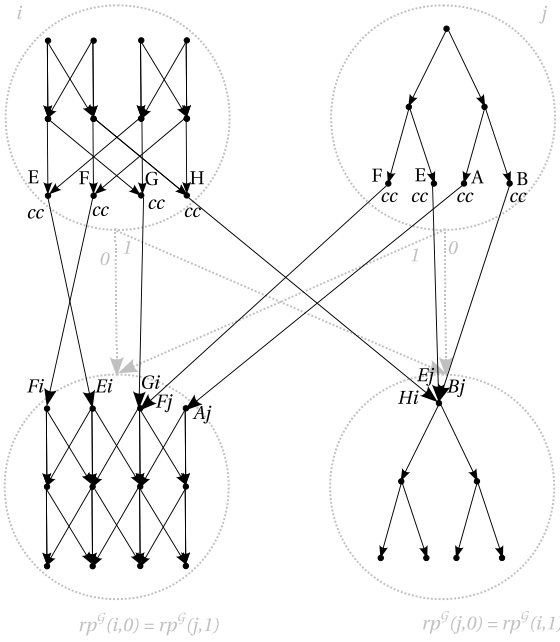

Figure 7.3: *Detail of a nested DDA (black coloured), obtained from* 4 *points of a possibly larger global DDA (grey coloured in the background), and their associated local DDAs. Several new dependency arcs have been created as allowed by the global dependencies, labeled with the corresponding new branch indices.*

points from $P^{\mathcal{G}}$ which do not have request directions. Further let $\mathcal{D} = (\mathcal{D}_i)_{i \in P^{\mathcal{G}}}$ be a family of (local) DDAs such that each point $i \in P^{\mathcal{G}}$ in the global DDA is associated with the DDA $\mathcal{D}_i = \langle P^i, B^i, req^i, sup^i \rangle$. Further let $t = (t^i)_{i \in P^{\mathcal{G}} \setminus I^{\mathcal{G}}}$ be a family of total functions such that $t^i : I^i \to \uplus_{j \in r_p^{\mathcal{G}}(i, B^{\mathcal{G}})} P^j$ where $I^i \subseteq P^i$.

Then *the nesting of $\mathcal{G}$ with $\mathcal{D}$ along t* is given by

$$\mathcal{N}^{\mathcal{G}}_{\mathcal{D},t} \stackrel{Def}{=} \mathcal{N} = \langle P^{\mathcal{N}}, B^{\mathcal{N}}, req^{\mathcal{N}}, sup^{\mathcal{N}} \rangle$$

where:

- $P^{\mathcal{N}} = \uplus_{i \in P^{\mathcal{G}}} P^i$
- $B^{\mathcal{N}} = \bigcup_{i \in P^{\mathcal{G}}} B^i \times \{\ell\} \quad \cup \quad \uplus_{i \in P^{\mathcal{G}} \setminus I^{\mathcal{G}}} I^i \quad \cup \quad \{\langle c, c \rangle\}$
  where $c \neq \ell$ and $c, \ell \notin P^{\mathcal{G}}$
- $req^{\mathcal{N}}$ consists of $(r_g^{\mathcal{N}}, r_p^{\mathcal{N}}, r_b^{\mathcal{N}})$ where:

$$\begin{aligned}
r_g^{\mathcal{N}}(\langle n, i \rangle, \langle d, j \rangle) &= \big((j = \ell) \wedge (d \in B^i) \wedge r_g^i(n, d)\big) \vee \\
&\quad \big((j = c) \wedge (d = c) \wedge (n \in I^i) \wedge (i \notin I^{\mathcal{G}})\big) \\
r_p^{\mathcal{N}}(\langle n, i \rangle, \langle d, j \rangle) &= \begin{cases} \langle r_p^i(n, d), i \rangle & \text{if} \quad j = \ell \\ t^i(n) & \text{if} \quad j = c \end{cases} \\
r_b^{\mathcal{N}}(\langle n, i \rangle, \langle d, j \rangle) &= \begin{cases} \langle r_b^i(n, d), \ell \rangle & \text{if} \quad j = \ell \\ \langle n, i \rangle & \text{if} \quad j = c \end{cases}
\end{aligned}$$

- $sup^{\mathcal{N}}$ consists of $(s_g^{\mathcal{N}}, s_p^{\mathcal{N}}, s_b^{\mathcal{N}})$ where:

$$\begin{aligned}
s_g^{\mathcal{N}}(\langle n, i \rangle, \langle d, j \rangle) &= \big((j = \ell) \wedge (d \in B^i) \wedge s_g^i(n, d)\big) \vee \\
&\quad \big((j \in P^{\mathcal{G}} \setminus I^{\mathcal{G}}) \wedge (d \in I^j) \wedge (t^j(d) = \langle n, i \rangle)\big) \\
s_p^{\mathcal{N}}(\langle n, i \rangle, \langle d, j \rangle) &= \begin{cases} \langle s_p^i(n, d), i \rangle & \text{if} \quad j = \ell \\ \langle d, j \rangle & \text{if} \quad j \in P^{\mathcal{G}} \setminus I^{\mathcal{G}} \end{cases} \\
s_b^{\mathcal{N}}(\langle n, i \rangle, \langle d, j \rangle) &= \begin{cases} \langle s_b^i(n, d), \ell \rangle & \text{if} \quad j = \ell \\ \langle c, c \rangle & \text{if} \quad j \in P^{\mathcal{G}} \setminus I^{\mathcal{G}} \end{cases}
\end{aligned}$$

□

The result of this nesting will in fact be a new DDA (see Fig. 7.3) as this is stated and formally proven in the next theorem. We will often refer to the functions t_i as *transfer* functions. Each point of the global DDA will

be replaced by its associated local DDA, and new dependency arcs will be created between the points of these local DDAs as appointed by the transfer functions. However, note that the transfer functions are defined in such a way, that new arcs will be added only along existing global dependencies. All new points of the nested DDA will be defined as tuples in which the second component will refer back to the original global DDA point. Branch indices again will be tuples: those with the second component being l will identify the original local branch indices of the local DDAs. Branch index $\langle c, c\rangle$ will identify the request end of all new dependency arcs, whereas for their supply ends the remaining branch indices defined in $B^{\mathcal{N}}$ will be used, in a manner similar presented for the serial DDA-combinator.

Theorem 7.1.14. The nesting of $\mathcal{G}$ with $\mathcal{D}$ along t as given in Definition 7.1.13 defines a DDA. □

Proof: The proof technique follows the same pattern first showed for the serial DDA-combinator in Theorem 7.1.9.

$\mathcal{N}$ will be a DDA if its $req^{\mathcal{N}}$ and $sup^{\mathcal{N}}$ components satisfy all DDA axioms. We show first for the supplies, i.e., $sup^{\mathcal{N}}$.

If $r_g^{\mathcal{N}}(\langle n, i\rangle, \langle d, j\rangle)$ holds, it is either the case that 1) $(j = l) \wedge (d \in B^i) \wedge r_g^i(n, d)$ or 2) $(j = c) \wedge (d = c) \wedge (n \in I^i) \wedge (i \notin I^{\mathcal{G}})$ holds. We distinguish therefore these two cases:

1. Under the assumption that $(j = l) \wedge (d \in B^i) \wedge r_g^i(n, d)$ holds we get:

$$r_p^{\mathcal{N}}(\langle n, i\rangle, \langle d, j\rangle) = \langle r_p^i(n, d), i\rangle \tag{7.8}$$

$$r_b^{\mathcal{N}}(\langle n, i\rangle, \langle d, j\rangle) = \langle r_b^i(n, d), l\rangle \tag{7.9}$$

This leads to:

$$\begin{aligned} & s_g^{\mathcal{N}}\big(r_p^{\mathcal{N}}(\langle n, i\rangle, \langle d, j\rangle), r_b^{\mathcal{N}}(\langle n, i\rangle, \langle d, j\rangle)\big) = && (7.10)\\ & \quad = s_g^{\mathcal{N}}(\langle r_p^i(n, d), i\rangle, \langle r_b^i(n, d), l\rangle) && \text{(by (7.8) and (7.9))}\\ & \quad = s_g^i(r_p^i(n, d), r_b^i(n, d)) && \text{(by def. of } s_g^{\mathcal{N}} \text{ and ass.)} \end{aligned}$$

which holds given that $r_g^i(n, d)$ holds and that $\mathcal{D}^i$ is a DDA. Hence (7.10) also holds.

The rest of the axioms for the supplies also hold:

$$\begin{aligned}
&s_p^{\mathcal{N}}\left(r_p^{\mathcal{N}}(\langle n,i\rangle,\langle d,j\rangle), r_b^{\mathcal{N}}(\langle n,i\rangle,\langle d,j\rangle)\right) = \\
&\quad = s_p^{\mathcal{N}}(\langle r_p^i(n,d),i\rangle,\langle r_b^i(n,d),l\rangle) && \text{(by (7.8) and (7.9))}\\
&\quad = \langle s_p^i(r_p^i(n,d),r_b^i(n,d)),i\rangle && \text{(by def. of } s_p^{\mathcal{N}} \text{ and ass.)}\\
&\quad = \langle n,i\rangle && \text{(as } \mathcal{D}^{\mathrm{i}} \text{ is a DDA)}
\end{aligned}$$

$$\begin{aligned}
&s_b^{\mathcal{N}}\left(r_p^{\mathcal{N}}(\langle n,i\rangle,\langle d,j\rangle), r_b^{\mathcal{N}}(\langle n,i\rangle,\langle d,j\rangle)\right) = \\
&\quad = s_b^{\mathcal{N}}(\langle r_p^i(n,d),i\rangle,\langle r_b^i(n,d),l\rangle) && \text{(by (7.8) and (7.9))}\\
&\quad = \langle s_b^i(r_p^i(n,d),r_b^i(n,d)),l\rangle && \text{(by def. of } s_b^{\mathcal{N}} \text{ and ass.)}\\
&\quad = \langle d,l\rangle && \text{(as } \mathcal{D}^{\mathrm{i}} \text{ is a DDA)}\\
&\quad = \langle d,j\rangle && \text{(by ass.)}
\end{aligned}$$

2. Under the assumption that $(j = c) \wedge (d = c) \wedge (n \in I^i) \wedge (i \notin I^{\mathcal{G}})$ holds we get:

$$\begin{aligned}
r_p^{\mathcal{N}}(\langle n,i\rangle,\langle d,j\rangle) &= r_p^{\mathcal{N}}(\langle n,i\rangle,\langle c,c\rangle) && \text{(by ass.)}\\
&= t^i(n) && \text{(by def. of } r_p^{\mathcal{N}}) \qquad (7.11)\\
r_b^{\mathcal{N}}(\langle n,i\rangle,\langle d,j\rangle) &= r_b^{\mathcal{N}}(\langle n,i\rangle,\langle c,c\rangle) && \text{(by ass.)}\\
&= \langle n,i\rangle && \text{(by def. of } r_b^{\mathcal{N}}) \qquad (7.12)
\end{aligned}$$

This leads to:

$$\begin{aligned}
&s_g^{\mathcal{N}}\left(r_p^{\mathcal{N}}(\langle n,i\rangle,\langle d,j\rangle), r_b^{\mathcal{N}}(\langle n,i\rangle,\langle d,j\rangle)\right) = && (7.13)\\
&\quad = s_g^{\mathcal{N}}(t^i(n),\langle n,i\rangle) && \text{(by (7.11) and (7.12))}
\end{aligned}$$

which holds by the definition of $s_g^{\mathcal{N}}$ and our assumption, hence (7.13) also holds.

The rest of the supply axioms hold as follows:

$$\begin{aligned}
&s_p^{\mathcal{N}}\left(r_p^{\mathcal{N}}(\langle n,i\rangle,\langle d,j\rangle), r_b^{\mathcal{N}}(\langle n,i\rangle,\langle d,j\rangle)\right) = \\
&\quad = s_p^{\mathcal{N}}(t^i(n),\langle n,i\rangle) && \text{(by (7.11) and (7.12))}\\
&\quad = \langle n,i\rangle && \text{(by def. of } s_p^{\mathcal{N}})
\end{aligned}$$

$$
\begin{aligned}
& s_b^{\mathcal{N}}\left(r_p^{\mathcal{N}}(\langle n,i\rangle,\langle d,j\rangle), r_b^{\mathcal{N}}(\langle n,i\rangle,\langle d,j\rangle)\right) = \\
& \quad = s_b^{\mathcal{N}}(t^i(n),\langle n,i\rangle) && \text{(by (7.11) and (7.12))} \\
& \quad = \langle c,c\rangle && \text{(by def. of } s_b^{\mathcal{N}}\text{)} \\
& \quad = \langle d,j\rangle && \text{(by ass.)}
\end{aligned}
$$

The axioms will hold for the requests, $req^{\mathcal{N}}$, as well:

If $s_g^{\mathcal{N}}(\langle n,i\rangle,\langle d,j\rangle)$ holds, it is either the case that 1) $(j=l)\wedge(d\in B^i)\wedge s_g^i(n,d)$ or 2) $(j\in P^{\mathcal{G}}\setminus I^{\mathcal{G}})\wedge(d\in I^j)\wedge(t^j(d)=\langle n,i\rangle)$ holds. We distinguish again two cases:

1. Under the assumption that $(j=l)\wedge(d\in B^i)\wedge s_g^i(n,d)$ holds we get:

$$s_p^{\mathcal{N}}(\langle n,i\rangle,\langle d,j\rangle) = \langle s_p^i(n,d),i\rangle \tag{7.14}$$

$$s_b^{\mathcal{N}}(\langle n,i\rangle,\langle d,j\rangle) = \langle s_b^i(n,d),l\rangle \tag{7.15}$$

This leads to:

$$
\begin{aligned}
& r_g^{\mathcal{N}}\left(s_p^{\mathcal{N}}(\langle n,i\rangle,\langle d,j\rangle), s_b^{\mathcal{N}}(\langle n,i\rangle,\langle d,j\rangle)\right) = && (7.16) \\
& \quad = r_g^{\mathcal{N}}(\langle s_p^i(n,d),i\rangle,\langle s_b^i(n,d),l\rangle) && \text{(by (7.14) and (7.15))} \\
& \quad = r_g^i(s_p^i(n,d),s_b^i(n,d)) && \text{(by def. of } r_g^{\mathcal{N}} \text{ and ass.)}
\end{aligned}
$$

which holds given that $s_g^i(n,d)$ holds and that $\mathcal{D}^i$ is a DDA. Hence (7.16) also holds.

The rest of the axioms for the requests come as follows:

$$
\begin{aligned}
& r_p^{\mathcal{N}}\left(s_p^{\mathcal{N}}(\langle n,i\rangle,\langle d,j\rangle), s_b^{\mathcal{N}}(\langle n,i\rangle,\langle d,j\rangle)\right) = \\
& \quad = r_p^{\mathcal{N}}(\langle s_p^i(n,d),i\rangle,\langle s_b^i(n,d),l\rangle) && \text{(by (7.14) and (7.15))} \\
& \quad = \langle r_p^i(s_p^i(n,d),s_b^i(n,d)),i\rangle && \text{(by def. of } r_p^{\mathcal{N}} \text{ and ass.)} \\
& \quad = \langle n,i\rangle && \text{(as } \mathcal{D}^{\mathrm{i}} \text{ is a DDA)}
\end{aligned}
$$

$$
\begin{aligned}
& r_b^{\mathcal{N}}\left(s_p^{\mathcal{N}}(\langle n,i\rangle,\langle d,j\rangle), s_b^{\mathcal{N}}(\langle n,i\rangle,\langle d,j\rangle)\right) = \\
& \quad = r_b^{\mathcal{N}}(\langle s_p^i(n,d),i\rangle,\langle s_b^i(n,d),l\rangle) && \text{(by (7.14) and (7.15))} \\
& \quad = \langle r_b^i(s_p^i(n,d),s_b^i(n,d)),l\rangle && \text{(by def. of } r_b^{\mathcal{N}} \text{ and ass.)} \\
& \quad = \langle d,l\rangle && \text{(as } \mathcal{D}^{\mathrm{i}} \text{ is a DDA)} \\
& \quad = \langle d,j\rangle && \text{(by ass.)}
\end{aligned}
$$

2. Under the assumption that $(j \in P^{\mathcal{G}} \setminus I^{\mathcal{G}}) \wedge (d \in I^j) \wedge (t^j(d) = \langle n, i \rangle)$ holds we get:

$$\begin{aligned} s_p^{\mathcal{N}}(\langle n,i\rangle, \langle d,j\rangle) &= s_p^{\mathcal{N}}(t^j(d), \langle d,j\rangle) && \text{(by ass.)} \\ &= \langle d,j\rangle && \text{(by def. of } s_p^{\mathcal{N}}) \qquad (7.17) \\ s_b^{\mathcal{N}}(\langle n,i\rangle, \langle d,j\rangle) &= s_b^{\mathcal{N}}(t^j(d), \langle d,j\rangle) && \text{(by ass.)} \\ &= \langle c,c\rangle && \text{(by def. of } s_b^{\mathcal{N}}) \qquad (7.18) \end{aligned}$$

This leads to:

$$\begin{aligned} & r_g^{\mathcal{N}}\left(s_p^{\mathcal{N}}(\langle n,i\rangle, \langle d,j\rangle), s_b^{\mathcal{N}}(\langle n,i\rangle, \langle d,j\rangle)\right) = \qquad (7.19) \\ & \quad = r_g^{\mathcal{N}}(\langle d,j\rangle, \langle c,c\rangle) \qquad \text{(by (7.17) and (7.18))} \end{aligned}$$

which holds by the definition of $r_g^{\mathcal{N}}$ and our assumption, hence (7.19) also holds.

The rest of the request axioms hold as follows:

$$\begin{aligned} & r_p^{\mathcal{N}}\left(s_p^{\mathcal{N}}(\langle n,i\rangle, \langle d,j\rangle), s_b^{\mathcal{N}}(\langle n,i\rangle, \langle d,j\rangle)\right) = \\ & \quad = r_p^{\mathcal{N}}(\langle d,j\rangle, \langle c,c\rangle) && \text{(by (7.17) and (7.18))} \\ & \quad = t^j(d) && \text{(by def. of } r_p^{\mathcal{N}}) \\ & \quad = \langle n,i\rangle && \text{(by ass.)} \end{aligned}$$

$$\begin{aligned} & r_b^{\mathcal{N}}\left(s_p^{\mathcal{N}}(\langle n,i\rangle, \langle d,j\rangle), s_b^{\mathcal{N}}(\langle n,i\rangle, \langle d,j\rangle)\right) = \\ & \quad = r_b^{\mathcal{N}}(\langle d,j\rangle, \langle c,c\rangle) && \text{(by (7.17) and (7.18))} \\ & \quad = \langle d,j\rangle && \text{(by def. of } r_b^{\mathcal{N}}) \end{aligned}$$

□

Note that the parallel DDA-combinator can be seen as a special case of nesting DDAs. When the global DDA consists of two distinct points with no arcs between them, then the family of transfer functions is the empty set, as $I^{\mathcal{G}} = P^{\mathcal{G}}$. Then as a result of "the nesting" the two points will be replaced with the corresponding two local DDAs and no new arcs will be added. Likewise, the serial DDA-combinator is also a special case of

nesting DDAs. Here the global DDA consists of two points with one single dependency arc between the points. Then the family of transfer functions consists of only one function, the one that is required in the serial DDA-combinator. However, note that formally the DDAs obtained via the parallel DDA-combinator and the serial DDA-combinator will be isomorphic, and not identical, with the corresponding nested versions.

7.2 Programming with Compound DDAs

The previous sections have presented various ways to combine DDAs. The compound DDAs have been proved to be genuine DDAs, hence a DDA-enabled compiler can directly generate new DDAs based on these formal descriptions. In this section we propose some programming language constructs that allow us to declare and program with compound DDAs, and motivates why, from a programming perspective, these constructs are useful.

7.2.1 *The Transfer Function*

Both the serial DDA-combinator and the nested DDA require the existence of a transfer function through which one can explicitly specify new connections between the points of the participating DDAs. The transfer function's domain and codomain as a general rule were subsets of DDA point sets. In practice we can control this with data invariants. If `P` is a point type of a DDA, then a subtype `I` of `P` can be defined as `I = P | DI` where `DI` is the new data invariant restricting the range of values of `P` to the ones we are interested in.

Let `I` and `O` be some types. If `exp` is an expression of type `O`, possibly containing a variable `p` of type `I`, then a transfer function `t:I→O` can be defined by `t(p)=exp`. When `t` is supposed to be bijective and serves as a transfer function for a bijective serial DDA-combinator, depending on the complexity of `exp` and the sub-types `I` and `O`, the compiler may not be able to automatically derive the inverse of `t` or it may just be too costly (e.g. administrating a huge look-up table). In such cases, its inverse should also be defined as another expression.

7.2.2 *Language Constructs for DDA-Combinators*

Imagine a scenario, e.g., in which we have some signal processing computation at hand, and for the sake of this example, it is based on two independent applications of the FFT (see Section 6.3). Assume that the two FFT-based sub-computations are expressed as two repeat statements in terms of

the butterfly DDA only. Since the butterfly DDA has a well-defined parallel execution scheme, each repeat statement can be compiled into some parallel code. Since they are independent, it would be natural to run them in parallel, but in this setting they will run one after the other, depending on the order they appear. Assume that the accumulated problem size of both sub-computations is at most the number of the available parallel processors. Then in practice each individual repeat statement will lead to a non-optimal resource utilization – a significant number of processors will still remain idle. Instead, we can use the parallel DDA-combinator to declare a compound DDA as the underlying dependency and to express a combined computation on it. This will allow a simultaneous execution of the two sub-computations of the signal processing: one is computed on the left butterfly, the other on the right, which can lead to both better resource utilization as well as running-time speed-ups.

When declaring a compound DDA `D` to be the parallel combination of DDAs `D1` and `D2` we use the **`par`** keyword:

```
D = D1 par D2
```

For better readability, we may place the whole expression between a set of parantheses, if needed: `(D1 par D2)`.

Example 7.2.1 (Cloned DBFs)**.** Let $\texttt{DBF}_\texttt{h}$ be the butterfly DDA of height `h` as defined in Example 4.2.2. Then the *cloned butterfly DDA of height* `h` will be the DDA $\texttt{DBF}_\texttt{h}$ **`par`** $\texttt{DBF}_\texttt{h}$ defined according to Definition 7.1.1. □

Accordingly, the point type of $\texttt{DBF}_\texttt{h}$ **`par`** $\texttt{DBF}_\texttt{h}$ will be $\texttt{BF}_\texttt{h}$+$\texttt{BF}_\texttt{h}$ and branch index type `{0,1}`, and let us denote by `rp'` its request function.

Let `V:A` be an array type, where `A` has index type $\texttt{BF}_\texttt{h}$+$\texttt{BF}_\texttt{h}$, some element type `E` and partial indexing operation $\texttt{\_[\_]:A,BF}_\texttt{h}\texttt{+BF}_\texttt{h}\texttt{→E}$.

We can now define a repeat statement on points $\texttt{p:BF}_\texttt{h}$+$\texttt{BF}_\texttt{h}$ along the compound DDA to do the required computations for our signal processing problem. For the sake of type-correct expressions we utilize the implicit projections of the disjoint union type (see Section 3.2.3).

```
repeat p:BFh+BFh along (DBFh par DBFh) from V in
       if (tag(p)=1) exprSP1(V[rp'(p,0)],V[rp'(p,1)])
       else exprSP2(V[rp'(p,0)],V[rp'(p,1)])
```

where $\texttt{expr}_\texttt{SP1}$ and $\texttt{expr}_\texttt{SP2}$ are the two sub-computations involved in the overall computation defined in terms of the compound DDA's request component.

Note that if the two sub-computations are based on butterfly DDAs of different heights, e.g., h_1 and h_2, one can accordingly define a repeat statement on the compound DDA DBF_{h_1} **par** DBF_{h_2} instead. While good resource utilization is still probable, speed-ups are likely to be more significant when $|h_1 - h_2|$ is small.

The **par** combinator can also be used in the specification of more complex compound DDAs, as this is illustrated in the next section. But first we need to introduce some language constructs for the serial DDA-combinator.

Let D1 be a DDA with point type P1 and D2 a DDA with point type P2. Further let t:I→O be a transfer function, where I is a sub-type of P2 and O is a sub-type of P1. Then the serial combination of D1 with D2 along t will be the DDA D denoted by **seq** and **via** keywords as follows:

```
D = D1 seq D2 via t
```

If t is a bijection we replace **via** with **bij**. Note that from a pragmatical point of view, if $|I|$ is large, the size of the compound DDA's branch index type obtained by the bijective serial DDA-combinator can be significantly smaller than the one obtained by the application of the general serial DDA-combinator via a total function (see Definition 7.1.8). On the other hand, in the latter case, the definition of the supply function omits completely the use of some corresponding inverse expression of t (though in certain cases it may be feasible to craft such an "inverse expression" automatically, even when t is only total). So instead, the new supply function uses directly these extra branch indices on the new connecting arcs to obtain the point the supply leads to. This could be beneficial for the running time as the execution won't be slowed down by the inverse expression's evaluation. Ultimately, the right choice between the two serial DDA-combinator is influenced by the actual building block DDA's properties and the way these combinators are finally implemented.

A typical example for the use of the serial combinator is when the result of a DDA-based computation, e.g., the target values (or some of the output values) of a repeat statement, are the initial values of a DDA-based computation right away, e.g., the next repeat statement. Hence it seems reasonable that, instead of two repeat statements, we issue only one with the compound DDA as the underlying dependency. This especially can be useful when the repeat statements are executed on an external (highly-parallel) accelerator (e.g. GPU, FPGA). Then there is no need to initialise twice the external hardware, and the desired result can be achieved in one go instead of additional data and device code transfers between host and device.

We illustrate the use of **seq** in a situation where, for the sake of the example, we need both the result of an FFT and its sorted form (e.g. for some statistical analysis). The underlying DDA of the FFT is again the butterfly DDA of height h, i.e., $\mathtt{DBF}_\mathtt{h}$ (Example 4.2.2), and the bitonic sort DDA for $2^\mathtt{h}$ inputs, i.e., $\mathtt{DBS}_\mathtt{h}$ is used for sorting the results (Example 6.1.1).

Further let $\mathtt{t}:\mathtt{I}_\mathtt{h}\rightarrow\mathtt{O}_\mathtt{h}$ be a transfer function, where:

- $\mathtt{I}_\mathtt{h}$ = $\mathtt{BS}_\mathtt{h}$ | DII
  with DII(p) = ((row(p)=1) && (sbf(p)=1))
- $\mathtt{O}_\mathtt{h}$ = $\mathtt{BF}_\mathtt{h}$ | DIO
  with DIO(p) = (row(p)=0)
- defined by t(p) = $\mathtt{BF}_\mathtt{h}$(0,col(p)) for all $\mathtt{p}:\mathtt{I}_\mathtt{h}$

Note that t is a bijection, $\mathtt{t}^{-1}$(p) = $\mathtt{BS}_\mathtt{h}$(1,1,col(p)) for all $\mathtt{p}:\mathtt{O}_\mathtt{h}$, hence we will consider the compound DDA obtained by the serial combination of $\mathtt{DBF}_\mathtt{h}$ and $\mathtt{DBS}_\mathtt{h}$ via t:

```
D = DBF_h seq DBS_h bij t
```

D will have $\mathtt{BF}_\mathtt{h}$+$\mathtt{BS}_\mathtt{h}$ as its point type, and {0,1}+{c} its branch index type, and we will denote its request function by rp'.

We can now define a targeted repeat statement on D, in which the target points of the computation will be the points of the top row of the butterfly DDA (end result of FFT) and of the top row of the bitonic sort DDA (end result of sorting). We define the target points for all $\mathtt{p}:\mathtt{BF}_\mathtt{h}$+$\mathtt{BS}_\mathtt{h}$ as follows:

```
TP(p) = ((tag(p)=1) && (row(v1(p))=0)) ||
        ((tag(p)=2) && (row(v2(p))=0) && (sbf(v2(p))=h))
```

where v1 and v2 are the usual projections of the disjoint union type.

Let W be an array of index type $\mathtt{BF}_\mathtt{h}$+$\mathtt{BS}_\mathtt{h}$ and some element type E, initialised with the inputs of the FFT, such that each W[p] with tag(p)=1 and row(v1(p))=h gets an initial value (the points of the bottom row of the butterfly DDA). Then the targeted repeat statement will be as follows:

```
repeat p:BF_h+BS_h along D from W for TP in
       if (tag(p)=1) expr_FFT(V[rp'(p,i1(0))],V[rp'(p,i1(1))])
       else if ((row(v2(p))=1) && (sbf(v2(p))=1)) V[rp'(p,i2(c))]
            else expr_BS(V[rp'(p,i1(0))],V[rp'(p,i1(1))])
```

where `i1` and `i2` are the usual injections of the disjoint union type; $\texttt{expr}_{\texttt{FFT}}$ is the computational expression on the bottom DDA performing the FFT; $\texttt{expr}_{\texttt{BS}}$ is the expression doing the sorting on the upper bitonic sort DDA.

Those points of the bitonic sort DDA for which `(row(v2(p))=1` and `sbf(v2(p))=1)` (i.e. exactly those in `I`) receive the value coming along on the new connecting arcs, i.e., with branch index `i2(c)`.

The sub-DDA-combinator applied to a DDA `D=<P,B,req,sup>` along `P1` and `B1`, which are sub-types of `P` and `B`, respectively, is declared with the keywords **`sub`** and **`along`** :

```
D1 = sub D along P1,B1
```

We can instantiate this construct on the reversed butterfly DDA, $\texttt{DRBF}_\texttt{h}$ (Example 4.2.3) to define the binary tree DDA as its sub-DDA. In Example 4.2.4 we defined the binary tree DDA explicitly. The same result can be obtained in one line by applying the sub-DDA-combinator:

$\texttt{DBT}_\texttt{h}$ = **`sub`** $\texttt{DRBF}_\texttt{h}$ **`along`** $\texttt{BT}_\texttt{h}$`,B`

In the explicit definition of $\texttt{DBT}_\texttt{h}$, in Example 4.2.4, we do not redefine the request and supply guards, and do not use restrictions on the arguments of the original DDA's components when defining the binary tree DDA's requests and supplies. This is due to the assumption we made in the preliminaries about guards and partial functions to avoid syntactic clutter. Hence the new $\texttt{BT}_\texttt{h}$ point type will be propagated in the binary tree DDA's newly defined components by the compiler. Likewise, in the formal definition of the sub-DDA-combinator the mechanism keeping things well-defined is spelt out in details, and this will also be handled by the compiler.

Projections of Compound DDAs

The parallel execution schemes of repeat statements are based on the space and time projections of the underlying DDA's point type (Section 4.3). Compound DDAs either inherit the building block DDAs own projections – adjusted to the compound DDA's point type, or if the programmer is aware of some additional properties of the compound DDA, perhaps more suitable projections can be defined.

7.2.3 An Example of "Combining" the Combinators

Ultimately each combinator promotes code reusability. Looking at a more complex data dependency graph of a given computation, sometimes it may be easier to express it as a compound DDA in terms of some smaller, easier definable DDAs. Then these "smaller" implementations can be reused by letting the compiler create the desired new DDA implementation. We need not to worry about what the new requests/supplies are, however we can refer to them as known when writing a repeat statement to solve our initial problem on the compund DDA.

We illustrate this via an example. There exists an efficient algorithm for multiplying large polynomials (e.g. [Quinn, 1986, p.98-99]) based on the repetitive application of the forward and the inverse FFT (Section 6.3). Any polynomial of degree $n-1$ is uniquely determined by its values at the n-th roots of unity. Applying the FFT on the coefficients of a polynomial of degree $n-1$ generates the value of the polynomial at the n roots of unity. Then the value of the product polynomial at any point is simply the product of the values of the two polynomials at the point. Then performing the inverse FFT on the values of the product polynomial at the n roots of unity generates its coefficients.

The underlying dependency of this computation can be defined as a compound DDA (see Fig. 7.4). Consider two polynomials of degrees `n1` and `n2`. The product polynomial will have degree `n1+n2`. For each direct and inverse FFT we need a copy of the radix-2 FFT DDA, $\texttt{DFFT}_\texttt{h}$. In order to make the product polynomial fit our FFT we choose the size of $\texttt{DFFT}_\texttt{h}$ such that $2^\texttt{h}$>`n1+n2`.

The evaluation of the two initial polynomials can be done in parallel, therefore we first combine two copies of $\texttt{DFFT}_\texttt{h}$ with **par**. Then the result of these direct FFTs need to be multiplied pair-wise, hence we connect $\texttt{DFFT}_\texttt{h}$ **par** $\texttt{DFFT}_\texttt{h}$ with a forking DDA of size $2^\texttt{h}$, $\texttt{DFK}_{2^\texttt{h}}$, serially via a bijection `t`. Finally, this is further combined serially with a new $\texttt{DFFT}_\texttt{h}$ via another bijection `t'` along which the product polynomial's coefficients will be obtained in the inverse FFT.

Example 7.2.2. The *polynomial multiplication DDA of degree* $2^\texttt{h}$, $\texttt{h} \in \mathbf{N}$ is a DDA $\texttt{DPM}_\texttt{h}$=<P,B,req,sup> defined as follows:

$$\texttt{DPM}_\texttt{h} = ((\texttt{DFFT}_\texttt{h}\ \textbf{par}\ \texttt{DFFT}_\texttt{h})\ \textbf{seq}\ \texttt{DFK}_{2^\texttt{h}}\ \textbf{bij}\ \texttt{t})\ \textbf{seq}\ \texttt{DFFT}_\texttt{h}\ \textbf{bij}\ \texttt{t'}$$

where:

- $\texttt{DFFT}_\texttt{h}$ is the radix-2 FFT DDA of size `h`

- $\mathtt{DFK}_{2^h}$ is the forking DDA of size 2^h
- $\mathtt{t{:}I_h} \rightarrow \mathtt{O_h}$ is a bijection with
  - $\mathtt{I_h} = \mathtt{FK}_{2^h}$ | DII
    DII(p) = (row(p)=0)
  - $\mathtt{O_h} = \mathtt{FFT_h{+}FFT_h}$ | DIO
    DIO(p) = (((tag(p)=1) && (row(v1(p))=h+1)) ||
    ((tag(p)=2) && (row(v2(p))=h+1)))
  - t(p) = **if** (col(p) < 2^h) i1($\mathtt{FFT_h}$(h+1,col(p)))
    else i2($\mathtt{FFT_h}$(h+1,col(p)-2^h))

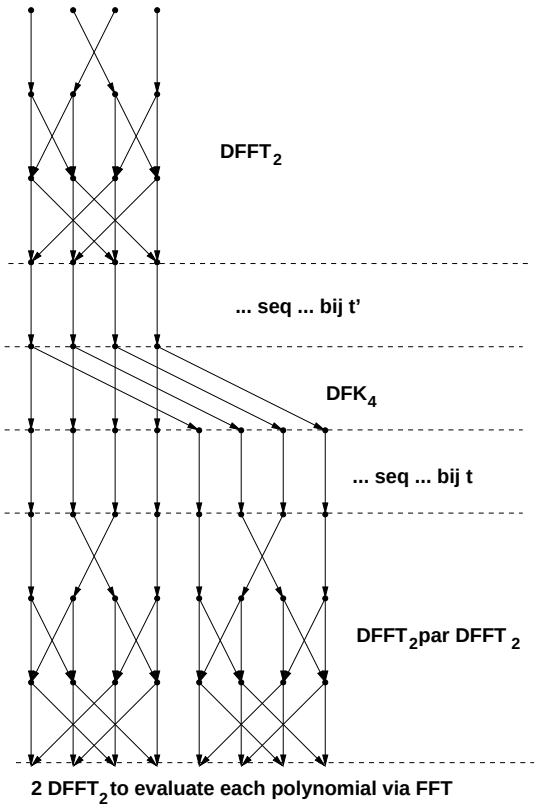

Figure 7.4: *The structural composition of the DDA underlying the multiplication of polynomials with degree* n1 *and* n2 *such that* n1+n2<4.

  - and inverse:

    t^{-1}(p) = **if** (tag(p)=1) FK_{2^h}(0,col(v1(p)))
    else FK_{2^h}(0,col(v2(p)))

- t':I_h'→O_h' is a bijection with

  - I_h' = FFT_h | DII'
    DII'(p) = (row(p)=0)
  - O_h' = (FFT_h+FFT_h)+FK_{2^h} | DIO'
    DIO'(p) = ((tag(p)=3) && (row(v3(p))=1))
  - t'(p) = i3(FK_{2^h}(1,col(p)))
  - and inverse:

    t'^{-1}(p) = FFT_h(0,col(v3(p)))

□

Let us denote by PM_h the point type of DPM_h, and by B its branch indices. Then by the definitions of the applied combinators we have:

PM_h = ((((FFT_h+FFT_h)+FK_{2^h})+FFT_h)

B = ({0,1}+{c})+{c}

The use of parantheses in the disjoint union type definitions will determine the tag component. For instance for a point p:PM_h of the compound DDA we have:

- if tag(p)=1, then p is a point of the left bottom $DFFT_h$
- if tag(p)=2, then p is a point of the right bottom $DFFT_h$
- if tag(p)=3, then p is a point of the forking DDA DFK_{2^h}
- if tag(p)=4, then p is a point of the top $DFFT_h$

We can now define the polynomial multiplication computations on DPM_h in form of a repeat statement. Let V be an array of index type p:PM_h and some complex number element type C. Our target values are the product polynomial's coefficients computed at the top row points of the compound DDA, that is our target points are those p:PM_h such that:

TP(p) = ((tag(p)=4) && (row(v4(p))=h+1))

Assume that `ig(V,p)` holds whenever `(tag(p)=1) && row(v1(p))=0` or `(tag(p)=2) && row(v2(p))=0`, that is `V` is properly initialised with the initial polynomials' coefficients. Let us denote by `rp` the request function of $\texttt{DPM}_\texttt{h}$. Then the following targeted repeat statement defines the multiplication of two polynomials of degree $\texttt{n1} \in \mathbf{N}$ and $\texttt{n2} \in \mathbf{N}$, respectively, where $\texttt{n1+n2<2}^\texttt{h}$:

```
  repeat p:PMh along DPMh from V for TP in
1   if ((tag(p)=4) && (row(v4(p))=0)) V[rp(p,i3(c))]
2   else if ((tag(p)=3) && (row(v3(p))=0)) V[rp(p,i2(c))]
3        else if (tag(p)=1) expr1FFT
4             else if (tag(p)=2) expr2FFT
5                  else if (tag(p)=3) V[rp(p,i1(0))]*V[rp(p,i1(1))]
6                       else expr3INV−FFT
```

where $\texttt{expr1}_\texttt{FFT}$, $\texttt{expr2}_\texttt{FFT}$ are the forward FFT expressions adopted from Section 6.3 and applied on the related parts of the compound DDA, i.e., the bottom left and bottom right $\texttt{DFFT}_\texttt{h}$-s, respectively. Likewise $\texttt{expr3}_{\texttt{INV}-\texttt{FFT}}$ is the inverse FFT applied on the top $\texttt{DFFT}_\texttt{h}$. For instance, $\texttt{expr1}_\texttt{FFT}$ stands for:

```
if (row(v1(p))<h+1)
    if (col(v1(p))<col(v1(rp(p,i1(1))))
       V[rp(p,i1(0))] + V[rp(p,i1(1))]*w^revh(col(v1(p))>>h−row(v1(p)))
    else V[rp(p,i1(1))] + V[rp(p,i1(0))]*w^revh(col(v1(p))>>h−row(v1(p)))
else V[rp(p,i1(0))]
```

Lines 1-2 pass the values on along the new connecting arcs of the compound DDA. Lines 3-4 do the forward FFTs. Line 5 multiplies the results pairwise (NB. multiplication here is supposed to be of complex numbers). And line 6 deals with the inverse FFT.

7.3 Compile Time Optimizations

We see that programming with compound DDAs is feasible, though it may result in hard-to-read repeat statements, if the compound DDA is too complex. Writing up all nested if-branches inside the repeat statement, with all the correct expressions within each if-branch, may not be very appealing for the programmer, and it is error-prone. The computation above defined for polynomial multiplication on the compound DDA illustrates exactly this problem.

Another scenario of a DDA-based polynomial multiplication would be to write two repeat statements on the simple FFT-DDA, then do the pairwise multiplication of the results on the forking DDA, and finally interpolate the result in the final repeat statement on the FFT-DDA again.

The theory of compound DDAs naturally raises the question whether a DDA-enabled compiler could identify situations like this, when repeat statements, defined on simple DDAs, can be safely merged together and build the underlying compound DDA? Intuitively, this should be possible when, for instance, the repeat statements' semantics are independent, or when the output values of one repeat statement serve as input values for the next one, or when the repeat statements are skewed or nested in a specific way satisfying certain properties.

This could be presented then in the form of an additional toolset that analysing the source code it points out possible ways of optimisations, allowing the programmer to consult the compound DDA and define embedding projections for it for the target hardware, or modify compiler-generated intermediate code and analyse its effect on the overall execution-time.

In the following section, we present some theoretical results that pinpoint these ideas. However note that these are just preliminary results, only a foretaste of the theory backing up the building of such optimisation tools.

Merging Consecutive Repeat Statements

The program codes and language constructs presented in the examples have the feeling of functional programming style. This applies to both the DDA examples as well as the repeat statements. Since the mathematical formulation of a DDA consists of a collection of functions, it is straightforward to implement them as functions. On the other hand, the repeat statement's functional recursive style is more deliberate. When the programmer issues a repeat statement, it will be at compile time that the compiler, based on the embedding space-time projections defined for the DDA, picks the corresponding execution scheme (sequential, MPI, CUDA, etc.) and generates code for it. However, prior to issuing the repeat statement the array involved in the repeat statement is supposed to be initialised otherwise no new values will be computed. Likewise, in subsequent code, the semantics of the repeat statement is assumed, i.e., the output or target values of the array to be computed by the repeat statement can be used, modified or thrown. Therefore we will assume an order of execution of the repeat statements following the order they appear in a program.

As suggested in Section 7.2.2 the use of **par** can lead to better resource utilization of a parallel machine and possibly improved running time. Programmers may not always be aware of this when writing repeat statements. The following theorem characterizes a situation when consecutive repeat statements can be merged automatically. In a sequential execution scheme this may not have much effect. In a parallel execution scheme, however, it may result in better resource utilization depending on how the size of the new compound DDA compares with the number of available processors. If the number of processors is smaller, then new space-time projections can further tune the setting. A straightforward benefit of the DDA-enabled compiler and the above-mentioned toolset would be that one is able to deal with such details at a high-level, resulting in efficient prototyping and implementation.

Theorem 7.3.1 (**par**-Based Optimization)**.** Let `D1=<P1,B1,req1,sup1>` and `D2=<P2,B2,req2,sup2>` be two countable DDAs, `M1` an array indexed by `P1` with element type `E1`, and `M2` an array indexed by `P2` with element type `E2`. Further let `exp1` and `exp2` be expressions of type `E1` and `E2`, respectively, such that the following repeat statements are both syntactically and type correct:

```
repeat n:P1 along D1 from M1 in exp1;
repeat m:P2 along D2 from M2 in exp2;
```

If `exp2` has no occurrences of `M1`, i.e., the values of array `M1` can be computed independently from values of `M2` and vice versa, then the overall semantics of the above repeat statements is the same as the semantics of the merged repeat statements from below:

```
repeat p:P1+P2 along (D1 par D2) from M in
       if (tag(p)=1) (E1+E2).i1(exp1')
       else (E1+E2).i2(exp2')
```

where

- `p` is fresh wrt. `exp1` and `exp2`
- `M` is an array indexed by `P1+P2` and element type `E1+E2` related to `M1` and `M2` such that:
  - for all `n:P1` where `ig(M1,n)` initially holds `M[(P1+P2).i1(n)] = (E1+E2).i1(M1[n])`, otherwise `ig(M,(P1+P2).i1(n))` does not hold .

  - for all `m:P2` where `ig(M2,m)` initially holds `M[(P1+P2).i2(m)] = (E1+E2).i2(M2[m])`, otherwise `ig(M,(P1+P2).i2(m))` does not hold.

- `exp1'` is an expression of type `E1` obtained from `exp1` as follows:
  - each occurrence of `M1[rp1(n,b)]` for any expression `b` of type `B1` is substituted with `v1(M[rp(p,b)])`
  - any other occurrence of `rg1(n,b)`, `rp1(n,b)`, `rb1(n,b)`, if any, for any expression `b` of type `B1` is replaced by `rg(p,b)`, `v1(rp(p,b))`, `rb(p,b)`, respectively
  - any occurrence of `sg1(n,b)`, `sp1(n,b)`, `sb1(n,b)`, if any, for any expression `b` of type `B1` is replaced by `sg(p,b)`, `v1(sp(p,b))`, `sb(p,b)`, respectively
  - all other occurrences of `n` is substituted with `v1(p)`
- `exp2'` is an expression of type `E2` obtained from `exp2` as follows:
  - each occurrence of `M2[rp2(m,b)]` for any expression `b` of type `B2` is substituted with `v2(M[rp(p,b)])`
  - any other occurrence of `rg2(m,b)`, `rp2(m,b)`, `rb2(m,b)`, if any, for any expression `b` of type `B2` is replaced by `rg(p,b)`, `v2(rp(p,b))`, `rb(p,b)`, respectively
  - any occurrence of `sg2(m,b)`, `sp2(m,b)`, `sb2(m,b)`, if any, for any expression `b` of type `B2` is replaced by `sg(p,b)`, `v2(sp(p,b))`, `sb(p,b)`, respectively
  - all other occurrences of `m` is substituted with `v2(p)`

  (`rg`, `rp`, `rb`) denoting the request and (`sg`, `sp`, `sb`) the supply components of `(D1 par D2)`.

□

Proof: First note that the merged repeat statement is both syntactically and type correct due to the carefully chosen substitutions applied to `exp1` and `exp2` and the fact that `p` was chosen to be fresh wrt. to these. Its semantics will be the array `M` filled with as many computed values as possible from its initial points – injected from the initial points of `M1` and `M2` – along the compound dependency obtained from the individual underlying dependencies of the original repeat statements. This also means that no less and no more is going to be computed than what is allowed by the original statements.

Secondly, all `p:P1+P2` s.t. `ig(M,p)` holds after the execution of the merged repeat statement it is the case that `tag(p) = tag(M[p])` which will ensure a direct correlation between the elements of the array `M` and of the arrays `M1` and `M2`. In other words, the values of `M1` and `M2` computable in the original repeat statements can be obtained from the computed values of `M` and vice versa as follows: `M1[v1(p)]=v1(M[p])` whenever `tag(p)=1`, and `M2[v2(p)]=v2(M[p])` otherwise. □

In practice, the initialisation of the arrays `M1` and `M2` takes place before issuing the repeat statements. It is also likely that some other statements may appear between the two repeat statements. The following proposition deals with such a scenario:

Proposition 7.3.2. Consider the following statements:

```
initialiseM1();
repeat n:P1 along D1 from M1 in exp1;
otherstatements();
initialiseM2();
repeat m:P2 along D2 from M2 in exp2;
```

If there is no occurrence of `M1` in `initialiseM2()`, and `initialiseM2()` is not dependent on `otherstatements()` then the result of the above program code is also a result of the one below:

```
initialiseM1(); initialiseM2();
createinitialiseM();
repeat p:P1+P2 along (D1 par D2) from M in
       if (tag(p)=1) then (E1+E2).i1(exp1')
                     else (E1+E2).i2(exp2')
updateM1(); updateM2();
otherstatements();
```

where:

- `expr1'` and `expr1'` are obtained in the manner discussed in Theorem 7.3.1
- `createinitialiseM()` creates an array `M` indexed by `P1+P2` and element type `E1+E2` and initialises it as discussed in Theorem 7.3.1

- `updateM1()` and `updateM2()` updates the original arrays with the computed values of `M`, i.e., for all `p:P1+P2` where `ig(M,p)` holds `M1[v1(p)] = v1(M[p])` whenever `tag(p)=1`, and `M2[v2(p)] = v2(M[p])`

□

Proof: The result of the first set of statements is basically the computed values of `M1`, `M2` and whatever else it has been computed in `otherstatements()`. (The latter may even use values of `M1`.) The result of the second set of statements contains the computed values of `M`, and through the updates, the computed values of `M1` and `M2`, and whatever else is being computed in `otherstatements()`. That the computed values of `M1` and `M2` are identical after both statements follows directly from Theorem 7.3.1 and the condition imposed on the intermediate statements. □

If `initialiseM2()` is dependent on `M1` then a corresponding transfer function and the automatic application of **`seq ...via ...`** combinator would lead to a merged repeat statement based on a compound DDA obtained via the serial combinator. The automatic application of the serial combinator and nested DDAs however is not addressed here, a theory for these is to be investigated in future work.

CHAPTER 8

Discussion

This chapter reflects on the characteristics of our DDA-based compiler to be, draws some conclusions, and points towards future directions.

8.1 Magnolia: a DDA-enabled Compiler

All previous chapters have played the role to demonstrate or emphasize, in one way or another, various aspects of our proposed programming model. The compiler, however, is one aspect which has received little attention. The primary reason behind this is the absence of a fully working DDA-based compiler.

In the late 90's, following the constructive recursive (CR) approach, Søreide [1998] built a prototype compiler based on Sapphire, a functional language designed for specifying CR problems in a rather simple way for rapid prototyping. This later was revisioned by Raubotn [2003]. The compiler generated sequential and MPI-code from simple DDA-descriptions, expressed in Sapphire.

Later, as we investigated the upcoming new generation of hardware architectures, the framework of our programming model has crystalised. This led us to the conclusion that merging the idea of a DDA-based compiler with the Magnolia framework [Bagge, 2009] would be a more promising enterprise. The tools used for the building of the Sapphire compiler, by this time, became obsolete, therefore, dusting and adjusting Sapphire was a far less appealing alternative.

Magnolia is a new programming language[1], currently under development, which aims at experimenting with program transformation and optimization. Though it tries to remain general-purpose, it is highly motivated by the numerical software domain where high-performance and support for parallel architectures are critical. This is the crucial point where our DDA-based approach fits beautifully in the Magnolia framework. The new directions pinpointed in Section 7.3 about DDA-combinators become very feasible given the apparatus through which the Magnolia compiler is being developed. Magnolia also allows the definition of axiom-based concepts by which one can specify the interface and behaviour of abstract data types [Bagge and Haveraaen, 2010]. This is central for our axiom-based DDA-formalism, since the compiler can verify the correctness of any DDA-implementation by checking whether they satisfy all DDA-axioms. This is essential for all supply-based execution models. For instance, if the DDA-implementation fails the test, the compiler should not generate code, as there is no guarantee whatsoever that the code would do the right thing.

The implementation of a DDA-enabled compiler, nonetheless, is not tied to Magnolia. All essential elements needed for the building of the compiler have been covered. From here, it is only a matter of taste how to choose the means.

8.2 Conclusion

We have presented the fundamental elements of a high-level portable parallel programming model, based on the formal theory of DDAs and their embeddings, and provided computational mechanisms that can serve as a basis for a DDA-enabled parallelizing compiler.

The evaluation of the model has been limited as we could only report on handcoded experiments. These, on the other hand, provided us with the necessary insight to be able to formalise and to argue about the correctness of the proposed computational mechanisms.

The practical experiments underlined that DDA-based programming is hardware independent, portable and flexible. Once the data dependency pattern of the computation is defined in a separate module, this can be re-used over various platforms. This results in high portability, but at the cost of slightly less optimal running times compared to fine-tuned platform specific program codes. A DDA-enabled compiler, however, with a proper optimizing mechanism in place is likely to be able to alleviate this problem.

[1] `http://magnolia-lang.org/`

The efficiency of the approach, in this sense, has been only addressed within the constrain of portability.

It has been shown that DDAs can capture the space-time communication topology of parallel systems, including modern architectures as well, such as GPUs. DDA-embeddings then allow us to modify at a high-level the way computations are mapped onto these hardware topologies. They give us full control over data locality, and spatial placements of computations. The latter led us to the idea that DDAs are suitable abstractions for defining FPGA placements of computations, and a circuit design process has been presented based on DDA-descriptions.

The visualization tool, which illustrated the effect of various embeddings, demonstrated the inherent flexibility of DDAs. The tool has been a great help in generating the majority of DDA-illustrations.

DDAs provide full control over the computation time of the execution models. Since spatial placements of computations are controlled from DDAs, this gives full control over space usage, whether sequential or parallel execution is desired. In the parallel cases, DDAs give full control over processor and memory allocation, and communication channel usage, while still at the abstraction level of the source program.

The framework primarily suits algorithms that exhibit static data dependencies, extractable as program code, for instance, certain recursive, loop-based or numerical computations. Due to their program refactoring properties, the DDA-combinators promise to expand this domain.

8.3 Future Work

Building the compiler with all back-end features presented is of utmost priority. Evaluation then can continue at various levels. Expanding the front-end with DDA-combinators is also considered of high priority. We plan building a more advanced visualization tool based on OpenGL, and expanding the back-end features with new execution models as soon as they have been formalised, amongst others, for Multi-GPU systems, Cell/BE and architecture specific instances of OpenCL.

Computations expressed on branch-valued DDAs will be explored as these suit better circuit descriptions, for instance, when targeting FPGAs. The study of DDA-combinators is going to be further developed as they point into directions with applicability in numerical software and high-performance computing. One essential aspect, for instance, is identifying some heuristics to automatically define optimal space-time projections of

compound DDAs. Since other hardware aspects, such as network/bus issues, traffic congestion, cache issues, etc., greatly affect system performance, it will be investigated how space-time DDAs can model these aspects as well. Then all execution models can take this additional information into account, which ultimately should lead to improved performance. Computations with dynamic data dependencies will also be explored in future work.

Summary

The foundations of a high-level hardware independent parallel programming model are presented. The theory of the underlying technical tools are expanded, and related mechanisms formalised for program refactoring.

The model addresses two main issues of parallel programming: how to map computations to different parallel hardware architectures, and how to do this at a low development cost, i.e., without rewriting the problem solving code. The user is presented with a programmable interface which allows to express the data dependency of the computation as real code. This in turn provides the means for a parallelizing compiler to harness directly the implicit driving force of dependencies and generate parallel code to virtually any parallel system which has a well defined space-time communication structure.

The underlying technical tool is based on the theory of Data Dependency Algebras (DDAs) and their embeddings. The computations and hardware communication layouts are defined in terms of DDAs. This entails the embedding of the computation onto various hardware architectures to be expressed in the form of DDA-embeddings. A DDA-based parallelizing compiler then can produce parallel code specific for each target architecture. The embedding is fully controlled by the programmer, at a high- and easy to modify level, without the need of rewriting the problem solving code.

The precise syntax and semantics of new language constructs are defined which provide the means to express DDA-based computations. This leads to a viable programming approach based on explicitly dependency-driven computational mechanisms. Several DDA-based computational mechanisms are presented, each targeting different hardware architectures, such as uni-processors, shared-memory model architectures, NVIDIA GPUs and FPGAs.

The ideas are instantiated through several well-known computational problems including sorting algorithms, parallel prefix sum and the Fast Fourier Transform. Preliminary experiments show that the approach is high-level, flexible and portable.

The theory of DDAs is further developed, expanding the study of space-time DDAs and DDA-projections, and presenting new DDA concepts.

Finally, new mechanisms are introduced which allow the building of more complex DDAs, in full suport of program refactoring.

Acronyms

- Cell/BE – Cell Broadband Engine: a heterogeneous microprocessor architecture jointly developed by Sony Computer Entertainment, IBM, and Toshiba
- CPU – Central Processing Unit: the central part of a computer that executes the instructions of a program.
  - Core: part of a CPU that de facto performs instruction execution.
  - Multi-core processor – a processor endowed with two or more cores interconnected on a single chip for enhanced performance, reduced power consumption, and more efficient simultaneous processing of multiple tasks.
  - Many-core processor – a common term referring to a multi-core processor possessing hundreds or thousands of on-chip cores
- CR – Constructive Recursion: is a programming methodology developed by Haveraaen [2000] to deliver a space and time optimal compilation of recursive programs.
- CUDA – Compute Unified Device Architecture: a parallel computing architecture developed by NVIDIA that serves as a high-level interface to program NVIDIA GPUs.
- DDA – Data Dependency Algebra: an algebraic abstraction developed to define data dependency graphs of computations as well as hardware connectivities. It constitutes the key concept of this dissertation.
- FPGA – Field Programmable Gate Array: an integrated circuit designed to be programmed by the user after manufacturing. The program is to be specified in some hardware description language, e.g. VHDL, which is then realised as a concrete circuit on the FPGA chip. It is re-programmable, attributing renewed functionalities of the same chip. A major supplier of FPGAs is Xilinx, Inc.

- GPU – Graphics Processing Unit: a processor designed to work as a co-processor to the main CPU in order to accelerate 2D and 3D graphics rendering.
- GPGPU – General Purpose Computations on GPUs: it refers to the non-graphics related use of GPUs to execute general purpose computations traditionally handled by the main CPU.
- MPI – Message Passing Interface: a standard communications protocol in parallel programming, especially used to program parallel computers connected in a network.
- NVIDIA – a multinational hardware vendor specialised in chipset technologies and GPUs.
- RLOC – Relative Location: a basic relational mapping and placement macro used to describe the layout of the circuit to be realised on FPGAs using Xilinx software. It is primarily used to increase speed and to use FPGA die resources efficiently.
- STA – Space-Time Algebra: a special DDA used to abstract over the dynamic connectivity of a (parallel) hardware or to define the space-time unfolding of a computation.

References

Aho, A. V. and Hopcroft, J. E. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition. ISBN 0201000296.

Allen, F.; Burke, M.; Charles, P.; Cytron, R.; and Ferrante, J. (1988). "An overview of the PTRAN analysis system for multiprocessing". In *Proceedings of the 1st International Conference on Supercomputing*, pages 194–211, New York, NY, USA. Springer-Verlag New York, Inc. ISBN 0-387-18991-2. URL http://portal.acm.org/citation.cfm?id=73950.73962.

Allen, J. R. (1983). *Dependence analysis for subscripted variables and its application to program transformations*. PhD thesis, Rice University, Houston, TX, USA. AAI8314916.

Allen, r.; Callahan, D.; and Kennedy, K. (1987). "Automatic decomposition of scientific programs for parallel execution". In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 63–76, New York, NY, USA. ACM. ISBN 0-89791-215-2. DOI http://doi.acm.org/10.1145/41625.41631.

Allen, R. and Kennedy, K. (1987). "Automatic translation of FORTRAN programs to vector form". *ACM Trans. Program. Lang. Syst.*, 9, pp. 491–542. ISSN 0164-0925. DOI http://doi.acm.org/10.1145/29873.29875.

Amarasinghe, S. P. and Lam, M. S. (1993). "Communication optimization and code generation for distributed memory machines". *SIGPLAN Not.*, 28, pp. 126–138. ISSN 0362-1340. DOI http://doi.acm.org/10.1145/173262.155102.

AMD (2006). "ATI CTM Guide". Technical reference manual, ATI. URL http://ati.amd.com.

Anderlik, A. and Haveraaen, M. (2003). "On the Category of Data Dependency Algebras and Embeddings". *Proceedings of the Estonian Academy of Sciences, Physics, Mathematics*, 52(4), pp. 337–355.

Asanovic, K.; Bodik, R.; Catanzaro, B. C.; Gebis, J. J.; Husbands, P.; Keutzer, K.; Patterson, D. A.; Plishker, W. L.; Shalf, J.; Williams, S. W.; and Yelick, K. A. (2006). "The Landscape of Parallel Computing Research: A View from Berkeley". Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

Axelsson, E.; Claessen, K.; Dévai, G.; Horváth, Z.; Keijzer, K.; Lyckegård, B.; Persson, A.; Sheeran, M.; Svenningsson, J.; and Vajda, A. (2010). "Feldspar: A domain specific language for digital signal processing algorithms". In *Formal Methods and Models for Codesign (MEMOCODE) 2010, 8th IEEE/ACM International Conference on*, pages 169 –178. DOI http://dx.doi.org/10.1109/MEMCOD.2010.5558637.

Bacon, D. F.; Graham, S. L.; and Sharp, O. J. (1994). "Compiler transformations for high-performance computing". *ACM Comput. Surv.*, 26, pp. 345–420. ISSN 0360-0300. DOI http://doi.acm.org/10.1145/197405.197406.

Baer, J. L. (1973). "A Survey of Some Theoretical Aspects of Multiprocessing". *ACM Comput. Surv.*, 5(1), pp. 31–80. ISSN 0360-0300. DOI http://doi.acm.org/10.1145/356612.356615.

Bagge, A. H. (2009). *Constructs & Concepts, Language Design for Flexibility and Reliability*. PhD thesis, Department of Informatics, University of Bergen, Norway, P.O. Box 7800, N-5020 Bergen, Norway.

Bagge, A. H. and Haveraaen, M. (2010). "Interfacing Concepts: Why Declaration Style Shouldn't Matter". In Ekman, T. and Vinju, J. J., editors, *Proceedings of the Ninth Workshop on Language Descriptions, Tools and Applications (LDTA '09)*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 37–50, York, UK. Elsevier. DOI http://dx.doi.org/10.1016/j.entcs.2010.08.030.

Banerjee, U. (1976). "Data dependence in ordinary programs". Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign.

Banerjee, U. (1979). *Speedup of ordinary programs*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA. AAI8008967.

Banerjee, U.; Eigenmann, R.; Nicolau, A.; and Padua, D. (1993). "Automatic program parallelization". *Proceedings of the IEEE*, 81(2), pp. 211 –243. ISSN 0018-9219. DOI http://dx.doi.org/10.1109/5.214548.

Banerjee, U. K. (1988). *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA. ISBN 0898382890.

Banerjee, U. K. (1996). *Dependence Analysis*. Kluwer Academic Publishers, Norwell, MA, USA. ISBN 0792398092.

Batcher, K. E. (1968). "Sorting Networks and Their Applications". In *AFIPS Spring Joint Computing Conference*, pages 307–314.

Bergland, G. D. (1969). "A Guided Tour of the Fast Fourier Transform". *IEEE Spectrum*, 6, pp. 41–52.
DOI http://dx.doi.org/10.1109/MSPEC.1969.5213896.

Bernstein, A. J. (1966). "Analysis of Programs for Parallel Processing". *Electronic Computers, IEEE Transactions on*, EC-15(5), pp. 757 –763. ISSN 0367-7508.
DOI http://dx.doi.org/10.1109/PGEC.1966.264565.

Bjesse, P.; Claessen, K.; Sheeran, M.; and Singh, S. (1999). "Lava: hardware design in Haskell". *SIGPLAN Not.*, 34(1), pp. 174–184. ISSN 0362-1340. DOI http://doi.acm.org/10.1145/291251.289440.

Blelloch, G. E. (1990). "Prefix Sums and Their Applications". Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University.

Blelloch, G. E.; Hardwick, J. C.; Sipelstein, J.; Zagha, M.; and Chatterjee, S. (1994). "Implementation of a portable nested data-parallel language". *J. Parallel Distrib. Comput.*, 21(1), pp. 4–14. ISSN 0743-7315. DOI http://dx.doi.org/10.1006/jpdc.1994.1038.

Bougé, L. (1996). "The Data Parallel Programming Model: A Semantic Perspective". In *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, volume 1132 of *Lecture Notes in Computer Science*, pages 4–26. Springer. ISBN 3-540-61736-1.

Burke, M. and Cytron, R. (1986). "Interprocedural dependence analysis and parallelization". In *Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, SIGPLAN '86, pages 162–175, New York, NY, USA. ACM. ISBN 0-89791-197-0.
DOI http://doi.acm.org/10.1145/12276.13328.

Burke, M. G. and Cytron, R. K. (2004). "Interprocedural dependence analysis and parallelization". *SIGPLAN Not.*, 39, pp. 139–154. ISSN 0362-1340. DOI http://doi.acm.org/10.1145/989393.989411.

Burrows, E. and Haveraaen, M. (2009a). "Dependency-driven Parallel Programming". In *Proceedings of the Norsk Informatikk Konferanse (NIK 2009)*, Trondheim, Norway. URL http://www.ii.uib.no/~eva/Publications/BuHa-nik09.pdf.

Burrows, E. and Haveraaen, M. (2009b). "A Hardware Independent Parallel Programming Model". *Journal of Logic and Algebraic Programming*, 78, pp. 519–538. DOI http://dx.doi.org/10.1016/j.jlap.2009.06.002.

Callahan, D.; Cooper, K. D.; Kennedy, K.; and Torczon, L. (1986). "Interprocedural constant propagation". In *Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, SIGPLAN '86, pages 152–161, New York, NY, USA. ACM. ISBN 0-89791-197-0. DOI http://doi.acm.org/10.1145/12276.13327.

Callahan, D. and Kennedy, K. (1988). "Compiling programs for distributed-memory multiprocessors". *The Journal of Supercomputing*, 2, pp. 151–169. ISSN 0920-8542. URL http://dx.doi.org/10.1007/BF00128175.

Chakravarty, M. M. T.; Leshchinskiy, R.; Jones, S. P.; Keller, G.; and Marlow, S. (2007). "Data parallel Haskell: a status report". In *DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18, New York, NY, USA. ACM. ISBN 978-1-59593-690-5. DOI http://doi.acm.org/10.1145/1248648.1248652.

Chandra, R.; Menon, R.; Dagum, L.; Kohr, D.; Maydan, D.; and McDonald, J. (2000). *Parallel programming in OpenMP*. Morgan Kauffman, San Franciso. URL http://www.OpenMP.org.

Chapman, B. (2005). "The Challenge of Providing A High-Level Programming Model for High-Performance Computing". In *High-Performance Computing: Paradigm and Infrastructure*, pages 21–49. Wiley. DOI http://dx.doi.org/10.1002/0471732710.ch2.

Chen, T.; Raghavan, R.; Dale, J. N.; and Iwata, E. (2007). "Cell Broadband Engine Architecture and its first implementation – A performance view". *IBM Journal of Research and Development*, 51(5), pp. 559–572. ISSN 0018-8646.

Christadler, I. and Weinberg, V. (2011). "RapidMind: Portability across Architectures and Its Limitations". In Keller, R.; Kramer, D.; and Weiss, J.-P., editors, *Facing the Multicore-Challenge*, volume 6310 of *Lecture Notes in Computer Science*, pages 4–15. Springer Berlin / Heidelberg. URL http://dx.doi.org/10.1007/978-3-642-16233-6_4.

Claessen, K.; Sheeran, M.; and Singh, S. (2003). "Using Lava to design and verify recursive and periodic sorters". *International Journal on Software Tools for Technology Transfer (STTT)*, 4, pp. 349–358. ISSN 1433-2779. URL http://dx.doi.org/10.1007/s10009-002-0089-y.

Cole, M. (1989). *Algorithmic skeletons: Structured management of parallel computation*. MIT press, Mass.

Cole, M. (2004). "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming". *Parallel Comput.*, 30(3), pp. 389–406. ISSN 0167-8191. DOI http://dx.doi.org/10.1016/j.parco.2003.12.002.

Collins, R. L.; Vellore, B.; and Carloni, L. P. (2010). "Recursion-driven parallel code generation for multi-core platforms". In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 190–195, 3001 Leuven, Belgium, Belgium. European Design and Automation Association. ISBN 978-3-9810801-6-2. URL http://portal.acm.org/citation.cfm?id=1870926.1870972.

Cooley, J. and Tukey, J. (1965). "An Algorithm for the Machine Calculation of Complex Fourier Series". *Mathematics of Computation*, 19(90), pp. 297–301. DOI http://dx.doi.org/10.2307/2003354.

Cooper, K. D.; Kennedy, K.; and Torczon, L. (1986). "Interprocedural optimization: eliminating unnecessary recompilation". In *Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, SIGPLAN '86, pages 58–67, New York, NY, USA. ACM. ISBN 0-89791-197-0. DOI http://doi.acm.org/10.1145/12276.13317.

Čyras, V. and Haveraaen, M. (1995). "Modular Programming of Recurrencies: a Comparison of Two Approaches". *Informatica*, 6(4), pp. 397–444.

Cytron, R.; Ferrante, J.; Rosen, B. K.; Wegman, M. N.; and Zadeck, F. K. (1991). "Efficiently computing static single assignment form and the control dependence graph". *ACM Trans. Program. Lang. Syst.*, 13, pp.

451–490. ISSN 0164-0925.
DOI http://doi.acm.org/10.1145/115372.115320.

Danelutto, M.; Di Meglio, R.; Orlando, S.; Pelagatti, S.; and Vanneschi, M. (1992). "A methodology for the development and the support of massively parallel programs". *Future Gener. Comput. Syst.*, 8, pp. 205–220. ISSN 0167-739X.
DOI http://dx.doi.org/10.1016/0167-739X(92)90040-I.

Darema, F.; George, D. A.; Norton, V. A.; and Pfister, G. F. (1988). "A single-program-multiple-data computational model for EPEX/FORTRAN". *Parallel Computing*, 7(1), pp. 11 – 24. ISSN 0167-8191.
DOI http://dx.doi.org/10.1016/0167-8191(88)90094-4.

Davis, A. L. (1979). "A data flow evaluation system based on the concept of recursive locality". *Managing Requirements Knowledge, International Workshop on*, 0, pp. 1079.
DOI http://doi.ieeecomputersociety.org/10.1109/AFIPS.1979.2.

Dongarra, J.; Foster, I.; Fox, G.; Gropp, W.; Kennedy, K.; Torczon, L.; and White, A., editors (2003). *Sourcebook of parallel computing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 1-55860-871-0.

Duhamel, P. and Vetterli, M. (1990). "Fast fourier-transforms – A tutorial review and a state-of the art". *Signal Processing*, 4(19), pp. 259–299.
DOI http://dx.doi.org/10.1016/0165-1684(90)90158-U.

Emiris, I. Z. and Pan, V. Y. (2010). "Applications of FFT and structured matrices". In Atallah, M. J. and Blanton, M., editors, *Algorithms and theory of computation handbook*, pages 18–18. Chapman & Hall/CRC. ISBN 978-1-58488-822-2. URL
http://portal.acm.org/citation.cfm?id=1882757.1882775.

Fatahalian, K.; Horn, D. R.; Knight, T. J.; Leem, L.; Houston, M.; Park, J. Y.; Erez, M.; Ren, M.; Aiken, A.; Dally, W. J.; and Hanrahan, P. (2006). "Sequoia: programming the memory hierarchy". In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA. ACM. ISBN 0-7695-2700-0.
DOI http://doi.acm.org/10.1145/1188455.1188543.

Feautrier, P. (1988). "Array expansion". In *Proceedings of the 2nd international conference on Supercomputing*, ICS '88, pages 429–441, New York, NY, USA. ACM. ISBN 0-89791-272-1. DOI http://doi.acm.org/10.1145/55364.55406.

Feldman, J. A. (1979). "High level programming for distributed computing". *Commun. ACM*, 22, pp. 353–368. ISSN 0001-0782. DOI http://doi.acm.org/10.1145/359114.359127.

Ferrante, J.; Ottenstein, K. J.; and Warren, J. D. (1987). "The program dependence graph and its use in optimization". *ACM Transactions on Programming Languages and Systems*, 9, pp. 319–349.

Flynn, M. J. (1972). "Some Computer Organizations and Their Effectiveness". *IEEE Trans. on Computers*, C-21(9), pp. 948–960.

Govindaraju, N.; Gray, J.; Kumar, R.; and Manocha, D. (2006). "GPUTeraSort: high performance graphics co-processor sorting for large database management". In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 325–336, New York, NY, USA. ACM. ISBN 1-59593-434-0. DOI http://doi.acm.org/10.1145/1142473.1142511.

Grama, A.; Gupta, A.; Karypis, G.; and Kumar, V. (2003). *Introduction to Parallel Computing. Second Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN 0201648652.

Greif, I. (1977). "A language for formal problem specification". *Commun. ACM*, 20, pp. 931–935. ISSN 0001-0782. DOI http://doi.acm.org/10.1145/359897.359904.

Gropp, W.; Lusk, E.; and Skjellum, A. (2000). *Using MPI: Portable Parallel Programming with the Message-passing Interface. 2nd edition*. MIT Press.

Gupta, G.; Renganarayanan, L.; Rajopadhye, S.; and Strout, M. (2007). "Computations on Iteration Spaces". In Srikant, Y. N. and Shankar, P., editors, *The Compiler Design Handbook: Optimization and Machine Code Generation, 2nd edition*. CRC Press. ISBN 9781420043822.

Gupta, M.; Mukhopadhyay, S.; and Sinha, N. (2000). "Automatic Parallelization of Recursive Procedures". *Int. J. Parallel Program.*, 28, pp. 537–562. ISSN 0885-7458. DOI http://dx.doi.org/10.1023/A:1007560600904.

Hansen, P. B. (1973). "Concurrent Programming Concepts". *ACM Comput. Surv.*, 5(4), pp. 223–245. ISSN 0360-0300. DOI http://doi.acm.org/10.1145/356622.356624.

Haveraaen, M. (1990b). "Distributing Programs on Different Parallel Architectures". In Padua, D. A., editor, *Proceedings of the 1990 International Conference on Parallel Processing (ICPP)*, volume II Software, pages 288–289. The Pennsylvania State University Press, University Park and London. ISBN 0-271-00728-1.

Haveraaen, M. (2000). "Efficient Parallelisation of Recursive Problems Using Constructive Recursion". In *Euro-Par 2000: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, number 1900 in Lecture Notes in Computer Science, pages 758–761, London, UK. Springer-Verlag. ISBN 3-540-67956-1. DOI http://dx.doi.org/10.1007/3-540-44520-X_104.

Haveraaen, M. (2009). "An algebra of data dependencies and embeddings for parallel programming". *Formal Aspects of Computing*. To appear.

Haveraaen, M. (June 1990a). "Data Dependencies and Space Time Algebras in Parallel Programming". Technical Report 45, Department of Informatics, University of Bergen, Norway.

Haveraaen, M. and G. Wagner, E. (2000). "Guarded Algebras: Disguising Partiality so You Won't Know Whether Its There". In Bert, D.; Choppy, C.; and Mosses, P., editors, *Recent Trends in Algebraic Development Techniques*, volume 1827 of *Lecture Notes in Computer Science*, pages 3–11. Springer Berlin / Heidelberg. ISBN 978-3-540-67898-4. DOI http://dx.doi.org/10.1007/978-3-540-44616-3_11.

Haveraaen, M. and Søreide, S. (1998). "Solving Recursive Problems in Linear Time Using Constructive Recursion". In *Norsk Informatikkonferance NIK'98*, pages 310–321, Trondheim, Norway. Tapir. URL http://www.ii.uib.no/saga/papers/tech2-5c.ps.

Hibbard, P. (1980). "Multiprocessor software design". In *ACM '80: Proceedings of the ACM 1980 annual conference*, pages 527–536, New York, NY, USA. ACM. ISBN 0-89791-028-1. DOI http://doi.acm.org/10.1145/800176.810011.

Hillis, W. D. and Guy L. Steele, J. (1986). "Data parallel algorithms". *Commun. ACM*, 29(12), pp. 1170–1183. ISSN 0001-0782. DOI http://doi.acm.org/10.1145/7902.7903.

Hoare, C. A. R. (1978). "Communicating sequential processes". *Commun. ACM*, 21(8), pp. 666–677. ISSN 0001-0782. DOI http://doi.acm.org/10.1145/359576.359585.

HPC Wire (2010). "A Call to Arms for Parallel Programming Standards. An interview with Intel's Tim Mattson". URL http://www.hpcwire.com.

Intel (2009). "Intel Parallel Studio". URL http://www.intel.com/go/parallel.

Intel (2010). "Intel's Array Building Blocks". URL http://software.intel.com/en-us/data-parallel/.

Isard, M.; Budiu, M.; Yu, Y.; Birrell, A.; and Fetterly, D. (2007). "Dryad: distributed data-parallel programs from sequential building blocks". *SIGOPS Oper. Syst. Rev.*, 41(3), pp. 59–72. ISSN 0163-5980. DOI http://doi.acm.org/10.1145/1272998.1273005.

Jen, C.-W. and Kwai, D.-M. (1992). "Data flow representation of iterative algorithms for systolic arrays". *Computers, IEEE Transactions on*, 41(3), pp. 351 –355. ISSN 0018-9340. DOI http://dx/.doi.org/10.1109/12.127448.

Johnston, W. M.; Hanna, J. R. P.; and Millar, R. J. (2004). "Advances in dataflow programming languages". *ACM Comput. Surv.*, 36, pp. 1–34. ISSN 0360-0300. DOI http://doi.acm.org/10.1145/1013208.1013209.

Kennedy, K. and Allen, J. R. (2002). *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 1-55860-286-0.

Kennedy, K.; Koelbel, C.; and Zima, H. (2007). "The rise and fall of High Performance Fortran: an historical object lesson". In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 7–1–7–22, New York, NY, USA. ACM. ISBN 978-1-59593-766-X. DOI http://doi.acm.org/10.1145/1238844.1238851.

Kennedy, K. and McKinley, K. S. (1990). "Loop distribution with arbitrary control flow". In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, pages 407–416, Los Alamitos, CA, USA. IEEE Computer Society Press. ISBN 0-89791-412-0. URL http://portal.acm.org/citation.cfm?id=110382.110458.

Khronos (2010). "The OpenCL Specification". Manual, Khronos OpenCL Working Group. URL http://www.khronos.org/registry/cl/.

Kirk, D. B. and Hwu, W.-m. W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann. ISBN 978-0-12-381472-2.

Kong, X.; Klappholz, D.; and Psarris, K. (1991). "The I Test: An Improved Dependence Test for Automatic Parallelization and Vectorization". *IEEE Trans. Parallel Distrib. Syst.*, 2, pp. 342–349. ISSN 1045-9219. DOI http://dx.doi.org/10.1109/71.86109.

Kuck, D.; Kuhn, R.; Leasure, B.; and Wolfe, M. (1980). "Analysis and transformation of programs for parallel computation". In *Proceedings of the 4th International Computer Software and Applications Conference*, pages 709–715, New York, NY, USA. IEEE.

Kuck, D.; Kuhn, R.; Padua, D.; Leasure, B.; and Wolfe, M. (1981). "Dependence graphs and compiler optimizations". In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, page 207–218, New York, NY, USA. ACM Press.

Kulkarni, M.; Burtscher, M.; Inkulu, R.; Pingali, K.; and Casçaval, C. (2009). "How much parallelism is there in irregular applications?". *SIGPLAN Not.*, 44, pp. 3–14. ISSN 0362-1340. URL http://doi.acm.org/10.1145/1594835.1504181.

Kung, H. T. (1982). "Why Systolic Architectures?". *Computer*, 15, pp. 37–46. ISSN 0018-9162. DOI http://dx.doi.org/10.1109/MC.1982.1653825.

Kung, H. T. and Leiserson, C. E. (1978). "Systolic Arrays (for VLSI)". In *Spare Matrix Proc.* Academic Press.

Kuon, I.; Tessier, R.; and Rose, J. (2008). "FPGA Architecture: Survey and Challenges". *Found. Trends Electron. Des. Autom.*, 2, pp. 135–253. ISSN 1551-3076. DOI http://dx.doi.org/10.1561/1000000005.

Kyriakopoulos, K. and Psarris, K. (2005). "Efficient Techniques for Advanced Data Dependence Analysis". In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 143–156, Washington, DC, USA. IEEE Computer Society. ISBN 0-7695-2429-X. DOI http://dx.doi.org/10.1109/PACT.2005.19.

Lamport, L. (1974). "The parallel execution of DO loops". *Commun. ACM*, 17, pp. 83–93. ISSN 0001-0782. DOI http://doi.acm.org/10.1145/360827.360844.

Lee, J.-D. and Batcher, K. E. (2000). "Minimizing Communication in the Bitonic Sort". *IEEE Trans. Parallel Distrib. Syst.*, 11(5), pp. 459–474. ISSN 1045-9219. DOI http://dx.doi.org/10.1109/71.852399.

Li, B.; Jin, H.; Zheng, R.; and Zhang, Q. (2008). "A Heterogeneous Data Parallel Computational Model for Cell Broadband Engine". *ChinaGrid, Annual Conference*, pages 325–330. DOI http://dx.doi.org/10.1109/ChinaGrid.2008.56.

Li, Z. and Yew, P.-C. (1988). "Efficient interprocedural analysis for program parallelization and restructuring". *SIGPLAN Not.*, 23, pp. 85–99. ISSN 0362-1340. DOI http://doi.acm.org/10.1145/62116.62125.

Lisper, B. (1996). "Data Parallelism and Functional Programming". In *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, volume 1132 of *Lecture Notes in Computer Science*, pages 220–250. Springer. ISBN 3-540-61736-1.

Loveman, D. (1993). "High performance Fortran". *IEEE Parallel & Distributed Technology: Systems & Applications*, 1(1), pp. 25–42. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=219857.

Loveman, D. B. (1976). "Program improvement by source to source transformation". In *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, POPL '76, pages 140–152, New York, NY, USA. ACM. DOI http://doi.acm.org/10.1145/800168.811548.

Maydan, D. E.; Hennessy, J. L.; and Lam, M. S. (1995). "Effectiveness of data dependence analysis". *International Journal of Parallel Programming*, 23, pp. 63–81. ISSN 0885-7458. DOI http://dx.doi.org/10.1007/BF02577784.

McCanny, J.; McWhirter, J.; and Kung, S.-Y. (1990). "The use of data dependence graphs in the design of bit-level systolic arrays". *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 38(5), pp. 787 –793. ISSN 0096-3518. DOI http://dx.doi.org/10.1109/29.56023.

McKinley, K. S. (1992). *Automatic and interactive parallelization*. Phd thesis, Rice University Houston, TX, USA, Houston, TX, USA. URL www.cs.utexas.edu/users/mckinley/papers/thesis.pdf.

Miranker, W. L. and Winkler, A. (1984). "Spacetime representations of computational structures". *Computing*, 32(2), pp. 93–114. ISSN 0010-485X. DOI http://dx.doi.org/10.1007/BF02253685.

Monteyne, M. (2008). "RapidMind Platform". Technical report, RapidMind.

Moore, G. E. (1965). "Cramming more components onto integrated circuits". *Electronics*, 38(8).

Müller-Olm, M. (2004). "Precise interprocedural dependence analysis of parallel programs". *Theoretical Computer Science*, 311, pp. 325–388. ISSN 0304-3975. DOI http://dx.doi.org/10.1016/j.tcs.2003.09.002.

Nickolls, J. and Dally, W. J. (2010). "The GPU Computing Era". *IEEE Micro*, 30, pp. 56–69. ISSN 0272-1732. DOI http://dx.doi.org/10.1109/MM.2010.41.

NVIDIA (2010). "CUDA Programming Guide". Manual, Nvidia. URL http://www.nvidia.com.

Owens, J. D.; Luebke, D.; Govindaraju, N.; Harris, M.; Krüger, J.; Lefohn, A. E.; and Purcell, T. J. (2007). "A Survey of General-Purpose Computation on Graphics Hardware". *Computer Graphics Forum*, 26(1), pp. 80–113. DOI http://dx.doi.org/10.1111/j.1467-8659.2007.01012.x.

Padua, D. A. and Wolfe, M. J. (1986). "Advanced compiler optimizations for supercomputers". *Commun. ACM*, 29, pp. 1184–1201. ISSN 0001-0782. DOI http://doi.acm.org/10.1145/7902.7904.

Perez, J.; Bellens, P.; Badia, R. M.; and Labarta, J. (2007). "CellSs: Making it easier to program the Cell Broadband Engine processor". *IBM Journal of Research and Development*, 51(5), pp. 593–604. ISSN 0018-8646. DOI http://dx.doi.org/10.1147/rd.515.0593.

PGAS (2010). "Partioned Global Address Space". URL http://pgas.org.

Polychronopoulos, C. D.; Gikar, M. B.; Haghighat, M. R.; Lee, C. L.; Leung, B. P.; and Schouten, D. A. (1990). "The structure of parafrase-2: an advanced parallelizing compiler for C and FORTRAN". In *Selected papers of the second workshop on Languages and compilers for parallel computing*, pages 423–453, London, UK, UK. Pitman Publishing. URL `http://portal.acm.org/citation.cfm?id=92402.92562`.

Pouchet, L.-N.; Bastoul, C.; Cohen, A.; and Cavazos, J. (2008). "Iterative optimization in the polyhedral model: part ii, multidimensional time". In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 90–100, New York, NY, USA. ACM.

Psarris, K. (1992). "On exact data dependence analysis". In *Proceedings of the 6th international conference on Supercomputing*, ICS '92, pages 303–312, New York, NY, USA. ACM. ISBN 0-89791-485-6. DOI `http://doi.acm.org/10.1145/143369.143424`.

Psarris, K. (1996). "The Banerjee-Wolfe and GCD tests on exact data dependence information". *J. Parallel Distrib. Comput.*, 32, pp. 119–138. ISSN 0743-7315. DOI `http://dx.doi.org/10.1006/jpdc.1996.0009`.

Pugh, W. and Wonnacott, D. (1992). "Eliminating false data dependences using the Omega test". *SIGPLAN Not.*, 27, pp. 140–151. ISSN 0362-1340. DOI `http://doi.acm.org/10.1145/143103.143129`.

Quinn, M. J. (1986). *Designing efficient algorithms for parallel computers*. McGraw-Hill, Inc., New York, NY, USA. ISBN 0-070-51071-7.

Rabhi, F. A. and Gorlatch, S., editors (2003). *Patterns and skeletons for parallel and distributed computing*. Springer-Verlag, London, UK. ISBN 1-85233-506-8.

Raubotn, S. (October 2003). *Hvordan implementere DP-problemer i Safir v.h.a. Konstruktiv Rekursjon. Master Thesis*. Master thesis, University of Bergen, P.O. Box 7800, N-5020 Bergen, Norway.

Reinders, J. (2007). *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly.

Reinders, J. (2010). "Think parallel or perish". *The Parallel Universe (Intel)*, 1(1).

Rugina, R. and Rinard, M. (1999). "Automatic parallelization of divide and conquer algorithms". In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '99, pages 72–83, New York, NY, USA. ACM. ISBN 1-58113-100-3. DOI http://doi.acm.org/10.1145/301104.301111.

Satish, N.; Harris, M.; and Garland, M. (2009). "Designing efficient sorting algorithms for manycore GPUs". In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. URL http://dx.doi.org/10.1109/IPDPS.2009.5161005.

Sheeran, M. (2010). "Functional and dynamic programming in the design of parallel prefix networks". *Journal of Functional Programming*, FirstView, pp. 1–56. DOI http://dx.doi.org/10.1017/S0956796810000304.

Singh, S. (2000). "Death of the RLOC?". *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, page 145. ISSN 1082-3409. DOI http://dx.doi.org/10.1109/FPGA.2000.903401.

Singh, S. (2008). "Declarative Programming Techniques for Many-Core Architecures". URL http://research.microsoft.com/~satnams/dec_manycore.pdf.

Singh, S. (2011). "The RLOC is Dead – Long Live the RLOC". In *Field Programmable Gate Arrays (FPGA), 2011 ACM/SIGA International Symposium on*. ACM.

Sintorn, E. and Assarsson, U. (2008). "Fast parallel GPU-sorting using a hybrid algorithm". *J. Parallel Distrib. Comput.*, 68(10), pp. 1381–1388. ISSN 0743-7315. DOI http://dx.doi.org/10.1016/j.jpdc.2008.05.012.

Skillicorn, D. B. (1995). "Towards a higher level of abstraction in parallel programming". In *Proceedings of the conference on Programming Models for Massively Parallel Computers*, PMMP '95, pages 78–, Washington, DC, USA. IEEE Computer Society. ISBN 0-8186-7177-7. URL http://portal.acm.org/citation.cfm?id=525697.826754.

Sklansky, J. (1960). "Conditional-Sum Addition Logic". *Electronic Computers, IRE Transactions on*, EC-9(2), pp. 226–231. ISSN 0367-9950. DOI http://dx.doi.org/10.1109/TEC.1960.5219822.

Søreide, S. (1998). *Compiling Sapphire into Sequential and Parallel Code Using Assertions*. Master thesis, Department of Informatics, University of Bergen, Norway, P.O. Box 7800, N-5020 Bergen, Norway.

Srikant, Y. N. and Shankar, P., editors (2007). *The Compiler Design Handbook: Optimizations and Machine Code Generation, 2nd edition*. CRC Press. ISBN 9781420043822.

Stone, H. (Feb. 1971). "Parallel Processing with the Perfect Shuffle". *Computers, IEEE Transactions on*, C-20(2), pp. 153–161. ISSN 0018-9340. DOI http://dx.doi.org/10.1109/T-C.1971.223205.

Svensson, J. (2011). *Obsidian: GPU Kernel Programming in Haskell*. Licentiate thesis 77l, Computer Science and Enginering, Chalmers University of Technology, Gothenburg. URL http://www.cse.chalmers.se/~joels/writing/lic.pdf.

Szymanski, B. K., editor (1991). *Parallel functional languages and compilers*. ACM, New York, NY, USA. ISBN 0-201-52243-8.

Veen, A. H. (1986). "Dataflow machine architecture". *ACM Comput. Surv.*, 18, pp. 365–396. ISSN 0360-0300. DOI http://doi.acm.org/10.1145/27633.28055.

Viitanen, M. and Hämäläinen, T. D. (2004). "Comparison of Data Dependence Analysis Tests". In Pimentel, A. and Vassiliadis, S., editors, *Computer Systems: Architectures, Modeling, and Simulation*, volume 3133 of *Lecture Notes in Computer Science*, pages 183–192. Springer. DOI http://dx.doi.org/10.1007/978-3-540-27776-7_16.

Wolfe, M. and Banerjee, U. (1987). "Data dependence and its application to parallel processing". *Int. J. Parallel Program.*, 16, pp. 137–178. ISSN 0885-7458. DOI http://dx.doi.org/10.1007/BF01379099.

Wolfe, M. J. (1990). *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, MA, USA. ISBN 0262730820.

Wolfe, M. J. (1996). *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN 0805327304.

Zhou, J. and Zeng, G. (2006). "A general data dependence analysis to nested loop using integer interval theory". In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 386–386, Washington, DC, USA. IEEE Computer Society. ISBN 1-4244-0054-6. URL http://portal.acm.org/citation.cfm?id=1898699.1898952.

Zima, H. and Chapman, B. (1993). "Compiling for distributed-memory systems". *Proceedings of the IEEE*, 81(2), pp. 264 –287. ISSN 0018-9219. DOI http://dx.doi.org/10.1109/5.214550.