Algebraic Combinators for Data Dependencies and Their Applications

Eva Burrows*

Bergen Language Design Laboratory
Department of Informatics
University of Bergen, Norway
http://bldl.ii.uib.no

Abstract

The notion of Data Dependency Algebra (DDA) is an algebraic formalism that turns data dependencies into first class citizens in the program code through a dedicated Application Programming Interface (API). This forms the basis of a platform independent parallel programming model [BH09]. In this paper, we further expand the theory of DDAs by proposing algebraic combinators operating on top of DDAs as a means to declare compound DDAs of custom complexity. The purpose is to allow the programmer to combine existing DDA implementations via high-level language constructs using simple declarations. The implementation of the compound DDA is generated at compile-time yet through the API its components are readily available for the programmer after declaration. We instantiate these ideas through the case-study of a DDA-based polynomial multiplication.

Introduction

Dependence analysis is a complex process through which a compiler collects relevant information about the execution-order of program statements [Ban96]. The aim is to identify situations when statements can be reordered for optimisation purposes without changing the semantics of the program. This may lead to improved instruction scheduling with decreased number of stalls, better exploitation of instruction-level parallelism when the hardware supports it, or improved memory locality, etc. With the appearance of early parallel computing systems, compilers also met the challenge of how to generate parallel executable on demand. This formed the basis of the concept known as automatic parallelization. While most modern compilers have successfully adopted optimisation techniques based on dependence analysis, its applicability for automatic parallelization has remained limited. Dependence analysis is NP-complete in the presence of recursion, indirect addressing, pointers, or when the behaviour of the program is dynamically determined, for instance, loops with non-fixed iteration spaces, or algorithms with input-dependent dynamic dependencies. In addition, a major problem with automatic parallelization is that sequential and parallel versions of an algorithm are fundamentally different. They are based on solution paradigms that do not necessarily relate to each other. Compiler transformations, on the other hand, generally preserve the solution paradigm. Hence, dependence analysis of sequential code cannot supply sufficient knowledge to aid the complex task of parallelization: parallel task decomposition, load balancing, data distribution, synchronisation, etc.

Over the last decades, research in parallel programming models and compilers has shown that a different coding technique is required when the aim is to execute certain parts of a computation in parallel. Be that through the means of language constructs with a dedicated parallel (or concurrent)

*This research has been supported by The Research Council of Norway through the project Design of a Mouldable Programming Language.

execution model, using for instance threads, parallel loops, data-parallel constructs, skeletons, message passing, or based on directives which instruct the compiler that annotated parts of the code can be executed in parallel, or based on other abstractions that help the compiler in the parallelization process.

With all that, parallel programming has proved to be very difficult and error-prone. Portability across multiple platforms and flexibility is also a major issue. Today, this is even more accentuated by the fact that applications need to be parallelized to adapt to the rapidly growing and versatile realm of parallel hardware systems. This applies to all range of computing systems, from commodity computers via embedded systems up to supercomputers. Multi-cores, many-cores, and accelerators like Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs) are becoming standard yet the search for parallel programming models that meet the requirements of portability, flexibility, efficiency, scalability and programming productivity across these platforms is still ongoing.

Parallel Programming with Data Dependencies

We pointed out that automatic dependence analysis cannot provide fine-grained details about the data dependencies occurring in a computation. Nonetheless, it is the flow of data and the presence or lack of dependencies between computational steps which determine any parallel execution. Therefore, we set our focus on fine-grained data dependencies.

The notion of *Data Dependency Algebra* (DDA), introduced in [Hav00], is an algebraic formalism that allows the programmer to present the data dependency graph of a computation as program code to a compiler. The abstraction is powerful enough to serve as the basis of a platform independent parallel programming model ensuring flexibility, productivity and portability across the platforms [BH09]. The approach also provides a high and easy to manipulate level for the programmer to deal with data distribution and placement [BH12]. In general, DDA-based parallel code generation is doable for any parallel systems with a well defined space-time communication structure. This has been shown for shared- and distributed-memory model computers, GPUs and FPGAs [Sør98, BH09, Bur14].

Central to this approach is the ability to extract manually the data dependency graph of an algorithm and code the computation in terms of the DDA API. This approach primarily suits computations with static and scalable data dependencies where the patterns are regular. Some data dependencies are more complex and probably less regular than for instance the butterfly pattern of the Fast Fourier Transform, sorting networks, or stencil computations of PDE solvers. Coding complex dependencies may easily become cumbersome.

In this presentation, we propose algebraic combinators operating on top of DDAs as a means to declare compound DDAs of custom complexity. The purpose is to allow the programmer to combine existing, easy to code DDA implementations via high-level language constructs. The implementation of the compound DDA is generated at compile-time yet through the API its components are readily available for the programmer after declaration. The following combinators are presented:

1. The *parallel DDA combinator* allows DDAs to be placed next to each other resulting in a larger DDA, to be referenced as a standalone DDA, without defining any additional connection between them.

2. On the contrary, the *serial DDA combinator* creates a larger DDA by connecting two DDAs in a "sequential" fashion. New branches are added in the compound DDA that will connect a designated set of points from the first DDA to a designated set of points of the second DDA determined by a given transfer function.

3. The *sub-DDA combinator* is a unary operator resulting in a smaller DDA "forgetting" parts of the original DDA as specified in the construct. It resembles the sub-graph relation from graph-theory.

4. The *nesting DDA combinator* requires a *global DDA*, a collection of *local DDAs*, one for each global DDA point, and a family of transfer functions, see Fig. 1. Each point of the global DDA is replaced by its associated local DDA, and new dependency branches are added along the global depen-

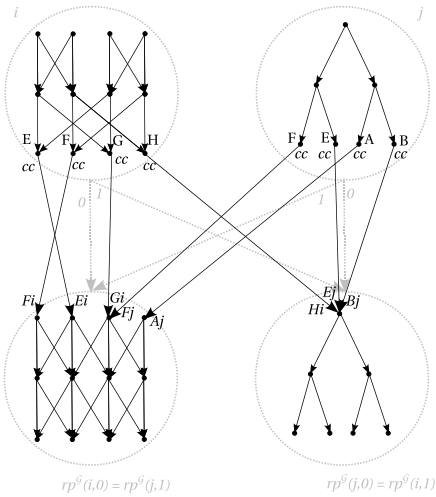

Figure 1: Detail of a nested DDA (black coloured), obtained using 4 points of a global DDA (grey coloured in the background), and the associated local DDAs.

dencies between the local DDAs as specified by the transfer functions.

Combinators can be applied in arbitrary order when declaring compound DDAs. Implementing the combinators in the compiler according to the proposed formalisms ensures that the compound DDA is in effect a DDA, i.e., its components satisfy the axiomatic requirements of the DDA API which is quintessential in the framework.

We discuss the benefits of the combinators in the programing model context and instantiate their use by presenting a compound DDA-based polynomial multiplication.

References

[Ban96] Utpal K. Banerjee. *Dependence Analysis*. Kluwer Academic Publishers, Norwell, MA, USA, 1996. ISBN:0792398092.

[BH09] Eva Burrows and Magne Haveraaen. A hardware independent parallel programming model. *The Journal of Logic and Algebraic Programming*, 78(7):519 – 538, 2009. The 19th Nordic Workshop on Programming Theory (NWPT 2007). Available from: `http://dx.doi.org/10.1016/j.jlap.2009.06.002`.

[BH12] Eva Burrows and Magne Haveraaen. Programmable data dependencies and placements. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, DAMP '12, pages 31–40, New York, NY, USA, 2012. ACM. Available from: `http://doi.acm.org/10.1145/2103736.2103741`.

[Bur14] Eva Burrows. Compiling a dataflow-based language abstraction onto an FPGA. In *Parallel Computing: Accelerating Computational Science and Engineering (CSE), Proceedings of the International Conference on Parallel Computing, ParCo 2013, 10-13 September 2013, Garching (near Munich), Germany*, pages 507–514, 2014. Available from: `http://dx.doi.org/10.3233/978-1-61499-381-0-507`.

[Hav00] Magne Haveraaen. 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, 2000. Springer-Verlag. Available from: `http://dx.doi.org/10.1007/3-540-44520-X_104`.

[Sør98] Steinar Søreide. *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, 1998.