Dependency-driven Parallel Programming

Eva Burrows * Magne Haveraaen †

Department of Informatics, University of Bergen, Norway
May 15, 2009

Abstract

The appearance of low-cost highly parallel hardware architectures has raised the alarm that a radically new way of thinking is required in programming to face the continually increasing parallelism of hardware. In our data dependency based framework, we treat data dependencies as first class entities in programs. Programming a highly parallel machine or chip is formulated as finding an efficient embedding of the computation's data dependency into the underlying hardware's communication layout. With the data dependency pattern of a computation extracted as an explicit entity in a program, one has a powerful tool to deal with parallelism.

1 Parallelism today

Computational devices are rapidly evolving into massively parallel systems. Multi-core processors are standard, and high performance processors such as the Cell processor [3] and graphics processing units (GPUs) featuring hundreds of on-chip processors (e.g. [5]) are all developed to accumulate processing power. They make parallelism commonplace, not only the privilege of expensive high-end platforms. However, current parallel programming paradigms cannot readily exploit these highly parallel systems. In addition, each hardware architecture comes along with a new programming model and/or application programming interface (API). This makes the writing of portable, efficient parallel code difficult. As the number of processors per chip is expected to double every year over the next few years, entering parallel processing into the mass market, software needs to be parallelized and ported in an efficient way to massively parallel, possibly heterogeneous, architectures. The programming community is in great need of high-level parallel programming models to adapt to the new era of commonly available parallel computing devices.

*http://www.ii.uib.no/~eva/
†http://www.ii.uib.no/~magne/

2 Dependency-driven thinking

Miranker and Winkler [4] suggested that program data dependency graphs can abstract how parts of a computation depends on data supplied by other parts. In our formalism these graphs are captured by algebraic abstractions – Data Dependency Algebras (DDAs) – and turned into first-class citizens in program code. This allows us to formulate the computation as expressions over consecutive computational points of the dependency pattern, such that dependencies between computational steps (DDA points) become explicit entities in the expression itself.

A parallel hardware architecture's space-time communication layout, its API, can also be captured by special space-time dependency patterns – Space-Time Algebras (STAs). This is obtained by projecting the static spatial connectivity pattern of the hardware over time. Mapping a computation to an available hardware resource then becomes a task of finding an embedding of the computation's DDA into the STA of the hardware [2]. This can be defined and easily modified at a high-level using DDA-embeddings, and a DDA-based compiler then can generate the executable for the required target mashine.

3 Little abstraction can do big things

Due to limitations on paper size, we do not formally define DDAs and STAs. Instead, we illustrate by the means of some figures what DDA abstractions enable us to do. Consider first the butterfly dependency, which appears in many divide-and-conquer algorithms, such as the Fast Fourier Transform. Fig. 1 shows the most common way it is layed out in a two dimensional spatial grid.

In general, the points can be placed in many different ways in a grid. The abstractions available in the DDA concept allow us to easily define different mappings of the same dependency by placing the points in different positions in the grid but preserving the de-

pendency relation between the points. This endows us to control, at a high-level, the embedding of the computation into the available hardware.

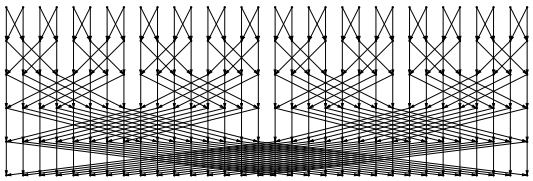

Figure 1: Butterfly dependency.

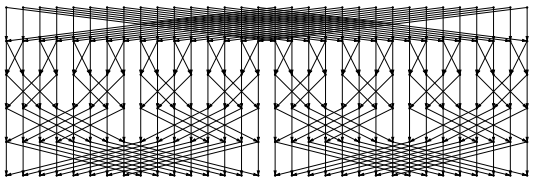

Figure 2: Butterfly dependency layed out using an alternativ DDA-projection setting.

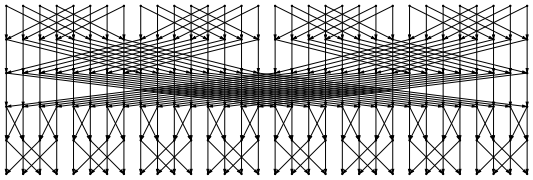

Figure 3: Butterfly dependency with yet another pair of projections.

Figures 2-4 illustrates the same butterfly dependency layed out using different DDA-projections. Out of these, in particular the shuffle network plays an important role in parallel processing.

Fig. 5 shows the dependency pattern of the Bitonic sorter. This can be seen as a combination of several butterfly dependencies of different height, each sub-butterfly corresponding to a bitonic merge. This also illustrates how data dependency abstractions entail code-reusability and modularity.

4 Implementation

To be able to compile the embeddings onto a target mashine, a DDA-enabled compiler is needed, and a corresponding base language with constructs to express our proposed formalism. Currently, these efforst are being carried out in the framework of the Magnolia programming language [1], which itself is under development. Magnolia allows the definition of concepts to specify the interface and behaviour of abstract data types which are useful to express the DDA concepts.

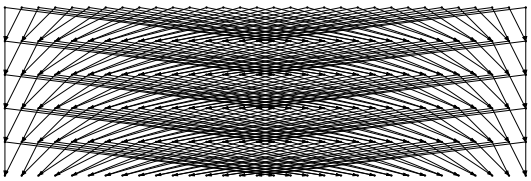

Figure 4: Butterfly dependency layed out as a shuffle network, as controlled by DDA-projections.

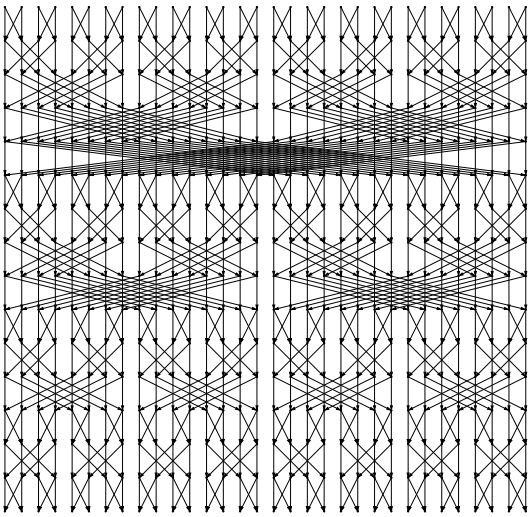

Figure 5: Bitonic sort DDA for sorting 32 inputs.

References

[1] Anya Bagge and Magne Haveraaen. Interfacing concepts. In T. Ekman and J. Vinju, editors, *Proceedings of the ninth Workshop on Language Decsriptions Tools and Applications LDTA 2009*, pages 238–252, 2009.

[2] Eva Burrows and Magne Haveraaen. A hardware independent parallel programming model. *Journal of Logic and Algebraic Programming*, (to appear), 2009.

[3] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell Broadband Engine Architecture and its first implementation – A performance view. *IBM Journal of Research and Development*, 51(5):559–572, 2007.

[4] W L. Miranker and A Winkler. Spacetime representations of computational structures. *Computing*, 32(2):93–114, 1984.

[5] NVIDIA. *CUDA Programming Guide*, 2008.