

Explicit Dependencies to Rule Them All

A Framework for Portable Parallel Programming

Eva Burrows *

Supervisor: Magne Haveraaen[†]

Department of Informatics, University of Bergen, Norway

3rd April 2011

1 PROBLEM & MOTIVATION

Computational devices are rapidly evolving into massively parallel systems. Multicore processors are already standard; high performance processors such as the Cell/BE processor [8], graphics processing units (GPUs) featuring hundreds of on-chip processors, and reconfigurable devices such as FPGAs are all developed to deliver high computing power. They make parallelism commonplace, not only the privilege of expensive high-end platforms. However, classical 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 other year or so, entering parallel processing into the mass market, software needs to be parallelized and ported in an efficient way to massively parallel, possibly heterogeneous, architectures.

One way of transforming manycore power into real application performance – an approach adopted by most hardware vendors – is to provide libraries, APIs (e.g. [21]) or some ready-to-use software toolbox that help developers to annotate legacy code with parallel constructs where possible (e.g. [14, 15, 26]). Industry leaders have also joined forces to develop OpenCL, a low-level cross-platform open standard for heterogeneous parallel programming [18]. OpenCL exposes everything of the underlying platforms and abstracts very little, ultimately hoping to become a target itself for higher level frameworks [13].

While these approaches are undoubtedly for the benefit of practicing program developers, they do not constitute a unified platform independent framework. The programming community is still in great need of high-level portable parallel programming models which can easily adapt to the upcoming new generation of commonly available parallel computing devices

and the increasingly more accessible realm of high-performance computing facilities.

2 BACKGROUND & RELATED WORK

One of the major issues in parallelizing applications is to deal with the underlying inherent dependency structure of the program. Automatic dependence analysis provides execution-order constraints between program statements and can establish legitimate ways to carry out program code transformations. The concept of *data dependency* constitutes one class of dependencies obtained throughout dependence analysis. This is usually represented as a directed graph in the compiler and abstracts how parts of a computation depend on data supplied by other parts.

Using data dependencies for program parallelization has a long and speckled history [17, 29]. Automatic parallelization [3], loop transformations [2, 23], systolic arrays [19, 20], dataflow programming [16], etc., all evolved from this basic concept. Many of the underlying ideas and observations related to these approaches served as a basis for parallelizing and optimizing compilers. The idea of embedding a program's communication structure into the hardware topology, for instance, seemed to be a very reasonable approach to deal with parallelism [20]. However, automatic dependence analysis has proved to be too complex [22, 24], and is NP-complete for the general cases [25]. As a result, parallelizing compilers cannot make the most of the underlying dependencies. This and the fact that data dependencies are considered inherent in a program and rather low-level artifacts of the Von Neumann machine led many to the conclusion that data dependencies cannot provide an improved level of abstraction at which to think about parallelization.

In this framework, we go against the flow, and raise the abstraction level by proposing to treat static data dependencies as explicit entities already at the program code level, by means more expressive than just simple annotations. Then a parallelizing compiler can omit data dependence analysis as a whole, and yet harness directly the implicit driving force of depend-

*<http://www.iu.uib.no/~eva/>

[†]<http://www.iu.uib.no/~magne/>

encies. As a result, it can generate parallel code to virtually any parallel system which has a well defined space-time communication structure, be that a shared or distributed memory machine, a GPU, an FPGA, etc.

In addition, the proposed abstraction addresses two main issues of parallel programming: how to map computations to different parallel hardware architectures at a high and easy to manipulate level, and how to do this at a low development cost, i.e., without re-writing the problem solving code.

Besides automatic parallelization techniques and higher level approaches (e.g. [1, 7, 9, 27, 28]), our focus on data dependencies relates us more directly to dataflow programming frameworks [16]. However, while in the latter, the main concept behind any program is the data, in our approach, the dependency is promoted as first class citizen.

3 APPROACH & UNIQUENESS

Miranker and Winkler [20] introduced a general theory about embedding the data dependency graph of a computation into the space-time topology of a systolic array. Haverlaen [10, 11] took this underlying idea further by separating the computation from its dependency in a modular way, so that both become programmable independently from each other. Data dependency graphs are captured by algebraic abstractions – Data Dependency Algebras (DDA) – and turned into explicit entities in the program code. Hardware communication topologies can be also described by DDAs. Then mapping the computation to a target architecture can be dealt with at a high-level, using DDA-embeddings [12].

A Unique Approach

We showed in [6] how DDA-abstractions can serve as a basis for a hardware independent parallel programming model. The uniqueness of this framework is two-folded. 1. It presents the user with a *programmable interface especially designed to allow the user to express the data dependency of the computation as real code*, hence making data dependencies explicit in the program code. The DDA interface consists of a generic point type, a generic branch index type, and two sets of generic function declarations on these types, requests and supplies, which are duals of each other. 2. The model *gives direct access, within a unified framework, to various hardware architectures' communication layouts, or their APIs, at a high-level*. This allows the embedding of the computation to be fully controlled by the programmer at a high and easy to manipulate level. In turn, this saves the user from the hassle of learning “the dialect” of each targeted hardware architecture in case. Direct access to aspects of the hardware model is needed by some architectures, e.g., GPUs, FPGAs. However, the model is fully portable and not tied to

any specific processor or hardware architecture, due to the modularisation of the data dependencies.

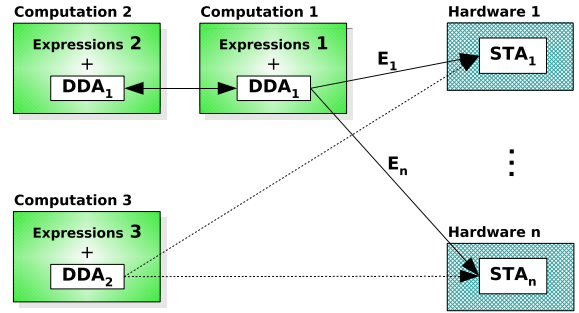


FIGURE 1: A Hardware Independent Parallel Programming Model

A Framework for Portable Parallel Programming

The conceptual overview of our framework is presented in Fig. 1. The key elements of the model are: (i) The data dependency pattern of a computation captured in the form of a DDA (e.g. DDA_1). In practice, this means that the data dependency graph of the computation is expressed in the formalism of the DDA interface. (ii) 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. (iii) A hardware architecture’s space-time communication layout or API also captured by a special *space-time DDA* (STA) (e.g. STA_1). (iv) The *embedding* of the computation by the means of a DDA-embedding (e.g. E_1, E_n). This is a task of finding a mapping of the computation’s DDA onto the STA of the hardware. (v) A *DDA-enabled compiler*, which for a given computation (e.g. computation 1) needs to be fed with: 1. the computation in terms of DDA_1 and the expressions over DDA_1 ; 2. the STA of the chosen hardware, e.g., STA_1 ; 3. the embedding E_1 from DDA_1 to this space-time, STA_1 . Then the compiler can generate code for the chosen hardware architecture. This can for instance be sequential code, MPI code, CUDA code for GPUs, or vectorized C/C++ for the Cell/BE, or FPGA circuit description, and so on.

The model assumes that the space-time DDAs of the hardware architectures are predefined (e.g. $STA_1, STA_2, \dots, STA_n$), 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 STA of the target architecture. For example, E_1 from DDA_1 to STA_1 , or E_n from DDA_1 to STA_n . The DDA-based computational expression defined on DDA_1 remains 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 re-usability* as well. Different computations may exhibit the same dependency pattern (DDA₁ in Computations 1 and 2), in which case all the embeddings defined for DDA₁ onto the different architectures can be reused. If a new computation exhibits a new dependency pattern (e.g. DDA₂ in Computation 3), all STAs, and associated execution models, are still available in the compiler, only new embeddings need to be defined into these, illustrated by dashed lines.

In the following sections, we attempt to make these ideas more concrete through some examples.

(i-ii) The Flavour of DDA-based Programming

The odd-even merge sort is a fast sorting network presented by K.E. Batcher [4]. The network is based on the repeated merging of ascendingly-ordered sequences of increasing size into one ascendingly-ordered sequence. We consider the data dependency graph of the odd-even merge sorting (see Fig. 2), when the number of inputs $n = 2^h$. The merging steps will define distinguishable sub-patterns of increasing size. In all DDA illustrations, the arrows point in the request directions. Hence, data flows opposite the arrows (bottom-up), along the supply directions.

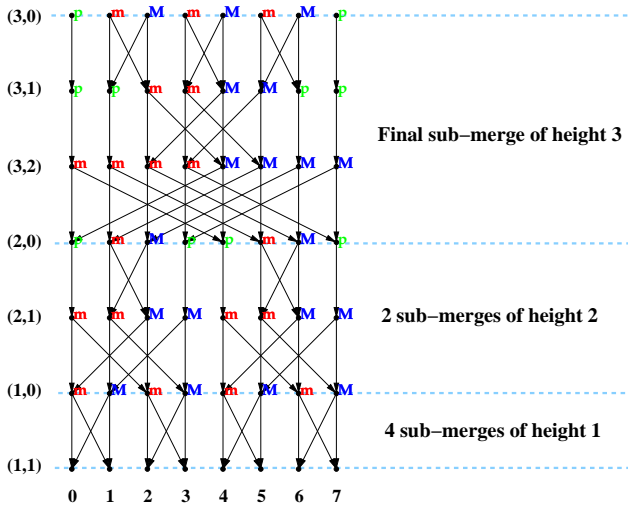


FIGURE 2: The odd-even merge sort DDA for 2^3 inputs. Nodes are annotated with computational expression labels: *m* stands for minimum, *M* for maximum, and *p* for pass-on.

We define the points of the DDA to be the tuple type $P = \text{sm Nat} * \text{row Nat} * \text{col Nat}$ where $\text{sm}:P \rightarrow \text{Nat}$, $\text{row}:P \rightarrow \text{Nat}$ and $\text{col}:P \rightarrow \text{Nat}$ are implicit projections of the tuple type such that *sm* identifies the

sub-merge step, *row* the local row number within a sub-merge step, and *col* the global column projection. P is also associated with a data invariant defined by:

$$\text{DI}_h(p) = (\emptyset < \text{sm}(p) \leq h) \ \&\& \ (\text{col}(p) < 2^h) \ \&\& \ ((\text{row}(p) < \text{sm}(p)) \ || \ (\text{row}(p) \leq \text{sm}(p) \ \&\& \ \text{sm}(p) = 1))$$

In addition, we will use the type name P as a constructor as well, i.e., $P:\text{Nat}, \text{Nat}, \text{Nat} \rightarrow P$.

We choose the type $B = \{0, 1\}$ for branch indices such that all vertical arcs are labelled with 0 and all arcs crossing columns are labelled with 1.

The DDA interface requires the definition of so called request and supply components (needed in the code generation), however, here we will only provide one element of the request component, which is also needed in the formalization of the computational expression on DDA points, and refer to [5] for more details. The function $\text{rp}:P, B \rightarrow P$ is defined such that $\text{rp}(p, b)$ returns the point that p requests data from along branch index $b:B$:

$$\begin{aligned} \text{rp}(p, b) = & \\ & \text{if} \ (\text{row}(p) \leq \text{sm}(p) - 2 \ || \ (\text{sm}(p) = 1) \) \\ & \quad \text{if} \ (b=0) \ P(\text{sm}(p), \text{row}(p)+1, \text{col}(p)) \\ & \quad \text{else if} \ (\text{max}(p)) \ P(\text{sm}(p), \text{row}(p)+1, \text{col}(p) - 2^{\text{row}(p)}) \\ & \quad \quad \text{else} \ P(\text{sm}(p), \text{row}(p)+1, \text{col}(p) + 2^{\text{row}(p)}) \\ & \text{else} \\ & \quad \text{if} \ (b=0) \ P(\text{sm}(p)-1, 0, \text{col}(p)) \\ & \quad \text{else if} \ (\text{max}(p)) \ P(\text{sm}(p)-1, 0, \text{col}(p) - 2^{\text{row}(p)}) \\ & \quad \quad \text{else} \ P(\text{sm}(p)-1, 0, \text{col}(p) + 2^{\text{row}(p)}) \end{aligned}$$

where $\text{max}:P \rightarrow \text{bool}$ is a boolean function identifying exactly those points which receive data along a vertical arc and one which is descending to the left from that point. This definition, cumbersome as it may seem, can be easily decoded by following the meaning of the DDA point projections and constructor in Fig. 2.

Throughout the odd-even merge sort, at every DDA point either the minimum or the maximum of the inputs is computed, or if there is only one input, the value is just passed on. Let us denote by $\text{pass}:P \rightarrow \text{bool}$ the boolean function that identifies exactly these pass-on points. Then the following expression defined on DDA points P defines the core computation of odd-even merge sort, where V is an array indexed by P :

$$\begin{aligned} V[p] = & \text{if} \ (\text{pass}(p)) \ V[\text{rp}(p, 0)] \\ & \text{else if} \ (\text{max}(p)) \ \text{max}(V[\text{rp}(p, 0)], V[\text{rp}(p, 1)]) \\ & \quad \text{else} \ \text{min}(V[\text{rp}(p, 0)], V[\text{rp}(p, 1)]) \end{aligned}$$

assuming that $V[P(1, 1, i)]$ for $i \in \{0, 1, \dots, 2^h - 1\}$ are the input values. Note how the dependency pattern given by rp (as part of the DDA) becomes an explicit entity in the problem solving code, separated now from the computation in a modular way.

Also note that the expression does not imply any specific execution model. This will be taken care of by the compiler, utilising the DDA and its embedding.

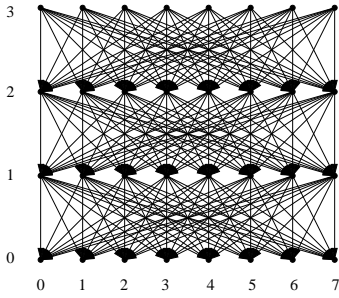


FIGURE 3: Shared memory STA for 8 processors in 4 time-steps.

(iii) Some Space-time DDAs

DDAs, by their very nature, can abstract 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.

Consider a shared or a distributed memory model architecture. In both models, every processor/process is assumed to be able to communicate with any other processor/process at any time-step. The only difference between the two models is the associated communication method: in shared memory there is direct access, in distributed memory the access is indirect, via message passing. The communication topology itself, however, is the same in both models. This can be abstracted by an STA, and the compiler provides different code in the shared versus distributed memory case. Then every STA point corresponds to a processor node at a given time-step with all its branches pointing to all other processors in the previous and upcoming time-steps, including itself, see Fig. 3. The *shared memory STA* for S processors, e.g., can be defined over $\text{Space} = \{0, 1, \dots, S-1\}$ by STA point type $\text{SMST}_S = \text{node Space} * \text{time Nat}$, branch indices $B = \text{Space}$, with the request function $\text{rp}(p, b) = \text{SMST}_S(b, \text{time}(p) - 1)$.

Likewise, the CUDA API of Nvidia GPUs [21] can also be abstracted as an STA. In CUDA, the GPU device operates as a Co-processor to the main CPU. The GPU

is 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 a *grid of blocks* upon the kernel invocation. 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.

Let us denote by T and B the number of threads per block, and the number of blocks that execute the kernel, respectively. Then the *CUDA kernel STA* launched with $B * T$ threads, can be defined by STA points: $\text{CUST}_{B,T} = \text{space CUB}_{B,T} * \text{time Nat}$ where space is comprised by $\text{CUB}_{B,T} = \text{block Nat} * \text{thread Nat}$ with data invariant: $\text{DIB}_{B,T}(s) = (\text{block}(s) < B) \ \&\& \ (\text{thread}(s) < T)$. Branch indices are defined again by the space component: $\text{CUB}_{B,T}$, and the request function then becomes $\text{rp}(p, b) = \text{CUST}_{B,T}(b, \text{time}(p) - 1)$.

We have fast, intra-block communication whenever $\text{block}(\text{space}(p)) = \text{block}(\text{space}(\text{rp}(p, b)))$, 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.

(iv) Various DDA-Embeddings for the Same Code

We focus our discussion now only on the STAs presented above. Given some DDA with an associated computational expression, and one of the STAs presented above, an *embedding of the computation* into the hardware is given by a point projection, EP , which defines how DDA points map into hardware space and time coordinates. For instance, the odd-even merge sort DDA embedding into a shared memory model architecture, $EP : P \rightarrow \text{SMST}_{2h}$, becomes:

$$EP(p) = \text{SMST}_{2h}(\text{col}(p), \text{grow}(\text{sm}(p), \text{row}(p)))$$

where the function $\text{grow} : \text{Nat}, \text{Nat} \rightarrow \text{Nat}$ computes the global row number of a point in the odd-even merge sort DDA based on the sub-merge and local row numbers, i.e., $\text{grow}(b, r) = b(b-1)/2 + b-r$.

Likewise, the embedding into the CUDA STA, $EP' : P \rightarrow \text{CUST}_{B,T}$, can be defined by (see Fig. 4.A):

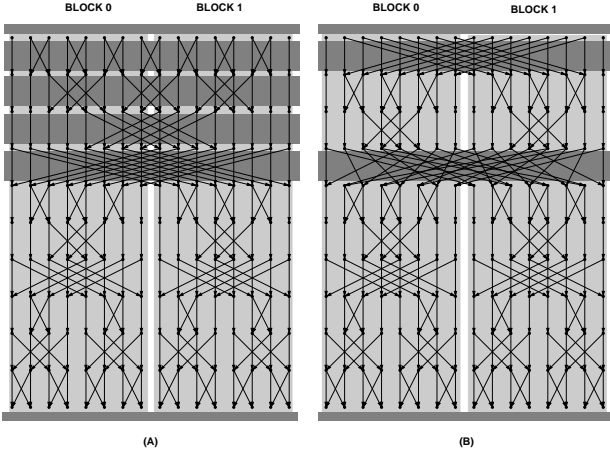


FIGURE 4: Odd-even merge sort DDA with CUDA embeddings for 2^4 inputs for $T=8$ and $B=2$ as given by EP' in (A) and by EP'' in (B). EP' induces 5, EP'' induces only 3 kernel invocations.

$$EP'(p) = CUST_{B,T}(CUB_{B,T}(\text{col}(p)/T, \text{col}(p)\%T), \text{grow}(\text{sm}(p), \text{row}(p)))$$

These embeddings, in fact, define a one-to-one correspondence between the points of the computational DDA and the hardware STA, so that the computational dependencies overlap hardware communication channels. However, the inherent flexibility of DDAs allows us to define embeddings in many different ways for the same hardware. E.g. the following projection (see Fig. 4.B) results in reduced number of kernel invocations in the last sub-merge compared to the previous embedding, since less branches cross over block boundaries in the top sub-merge.

$$EP''(p) = \begin{cases} \text{if } (\text{sm}(p)=h) \\ CUST_{B,T}(CUB_{B,T}(\text{ShR}_h(\text{col}(p), 1)/T, \text{ShR}_h(\text{col}(p), 1)\%T), \text{grow}(\text{sm}(p), \text{row}(p))) \\ \text{else} \\ CUST_{B,T}(CUB_{B,T}(\text{col}(p)/T, \text{col}(p)\%T), \text{grow}(\text{sm}(p), \text{row}(p))) \end{cases}$$

where $\text{ShR}_h: \text{Nat}, \text{Nat} \rightarrow \text{Nat}$ is defined s.t. $\text{ShR}_h(n, i)$ returns the value of a cyclic shift to the right on the h -bit binary representation of n by i positions.

(v) DDA-based Code Generation

The user is to describe a DDA-based computational expression and its embedding onto a given hardware following a methodology similar to that of (i),(ii) and (iv). The STAs of point (iii) are provided by the DDA-enabled compiler developer. In the compiler, each STA is associated with a computational mechanism specific to the hardware architecture that the STA describes. Throughout the code-generation process, when targeting a hardware, the computational mechanism is instantiated for the DDA-based computational expression and its embedding. Each execu-

tion model follows a bottom-up recursion unfolding strategy entirely based on the information provided in the DDA, hence, making the most of the underlying original dependency. Based on these strategies, we defined several execution models for uni-processors, shared-memory model architectures, the CUDA API, and formalised a methodology how to generate FPGA-circuit descriptions from DDA-based computational expressions [5, 6].

4 RESULTS & CONTRIBUTIONS

This work contributes to a novel understanding that data dependencies can play in parallel computing by: 1. showing that DDAs raise the level of abstraction at which one can think about parallelization; 2. providing a unified programming model to port computations to various hardware architectures at a high and easy to manipulate level; 3. formalising DDA-based execution models for various hardware.

DDAs provide full control over the execution models' computation time. Since spatial placements of computations are controlled from DDAs, this gives full control over space usage as well, 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. These benefits should outweigh the initial challenge which forces the programmer to first *think* about data dependencies, and extract them as program code, instead of just simply assuming their implicit existence.

The framework primarily suits algorithms that exhibit static data dependencies, extractable as program code, for instance, certain recursive, loop-based or numerical computations.

The evaluation of the model has been limited to handcoded experiments. These provided us with the insight to formalise and to argue about the correctness of our proposed execution models [5]. The building of a DDA-enabled compiler is an ongoing project. Practical experiments, however, showed 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 in the compiler.

We built a DDA visualization tool that makes evident the inherent flexibility of DDA-embeddings. The tool has also the additional benefits to test the correctness of DDA-implementations, and to generate very large DDA-drawings to get an intuition about the underlying dependencies at a large scale.

REFERENCES

- [1] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A domain spe-

- cific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMO-CODE) 2010, 8th IEEE/ACM International Conference on*, pages 169–178, 2010.
- [2] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26:345–420, December 1994.
- [3] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, Feb. 1993.
- [4] K. E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, pages 307–314, 1968.
- [5] E. Burrows. *Programming with Explicit Dependencies. A Framework for Portable Parallel Programming*. PhD thesis, Department of Informatics, University of Bergen, Norway, P.O. Box 7800, N-5020 Bergen, Norway, 2011.
- [6] E. Burrows and M. Haverdaen. A hardware independent parallel programming model. *Journal of Logic and Algebraic Programming*, 78:519–538, 2009.
- [7] M. M. T. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow. 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, 2007. ACM.
- [8] 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.
- [9] M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, 2004.
- [10] M. Haverdaen. Distributing programs on different parallel architectures. In D. A. Padua, 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, 1990.
- [11] M. Haverdaen. 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.
- [12] M. Haverdaen. An algebra of data dependencies and embeddings for parallel programming. *Formal Aspects of Computing*, 2009. To appear.
- [13] HPC Wire. A call to arms for parallel programming standards. An interview with Intel’s Tim Mattson, november 2010.
- [14] Intel. Intel Parallel Studio, 2009.
- [15] Intel. Intel’s array building blocks, 2010.
- [16] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36:1–34, March 2004.
- [17] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
- [18] Khronos. The OpenCL Specification. Manual, Khronos OpenCL Working Group, 2010.
- [19] J. McCanny, J. McWhirter, and S.-Y. Kung. The use of data dependence graphs in the design of bit-level systolic arrays. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 38(5):787–793, May 1990.
- [20] W. L. Miranker and A. Winkler. Spacetime representations of computational structures. *Computing*, 32(2):93–114, 1984.
- [21] NVIDIA. CUDA Programming Guide. Manual, Nvidia, 2010.
- [22] P. Petersen and D. Padua. Static and dynamic evaluation of data dependence analysis techniques. *Parallel and Distributed Systems, IEEE Transactions on*, 7(11):1121–1132, Nov. 1996.
- [23] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. 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, 2008. ACM.
- [24] K. Psarris and K. Kyriakopoulos. An experimental evaluation of data dependence analysis techniques. *Parallel and Distributed Systems, IEEE Transactions on*, 15(3):196–213, 2004.
- [25] W. Pugh and D. Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Trans. Program. Lang. Syst.*, 16:1248–1278, July 1994.
- [26] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O’Reilly, 2007.
- [27] S. Singh. Declarative programming techniques for many-core architectures, 2008.
- [28] J. Svensson. *Obsidian: GPU Kernel Programming in Haskell*. Licentiate thesis 771, Computer Science and Engineering, Chalmers University of Technology, Gothenburg, 2011.
- [29] M. Wolfe and U. Banerjee. Data dependence and its application to parallel processing. *Int. J. Parallel Program.*, 16:137–178, April 1987.