Executable Modelling for Highly Parallel Accelerators

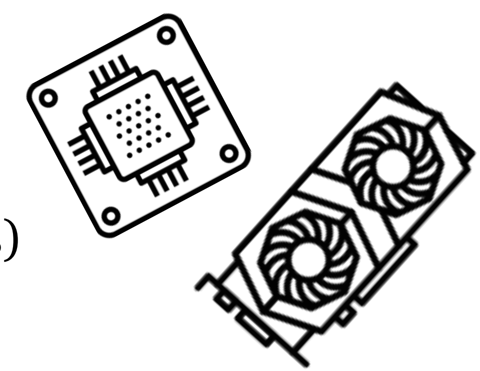
Lorenzo Addazi - Federico Ciccozzi - Björn Lisper
School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden

MÄLARDALEN UNIVERSITY
SWEDEN

High-Performance Embedded Systems

- Large and ever-increasing need for computational power in resource-constrained embedded systems, e.g. autonomous driving applications

- Hardware acceleration to offload heavy tasks from CPUs to dedicated computer hardware:
  - Graphical Processing Units (GPUs)
  - Field-Programmable Gate Arrays (FPGAs)
  - Application-Specific Integrated Circuits (ASICs)



So, what is the problem?

- Numerous architectures with different programming models
  - Complex
  - Low level of abstraction
  - Explicit parallelism

- Lack of appropriate support
  - Languages – high-level programming languages
  - Tools – parallel debuggers, etc.

- Unacceptable risks for safety-critical embedded applications

Our idea

- High-level data-parallel executable modelling language based on fUML/ALF
  - Reusability/Flexibility
    same code, different accelerators
  - Early Analysis
    continuous feedback during development
  - Code Generation
    hardware-specific code generated from models

Data-Parallel Programming Model

Programs expressed as compositions of collective operations on homogeneous data structures, e.g. arrays, lists

- Implicit Parallelism
  - Composition of inherently parallel primitives
- No Race Conditions, No Deadlocks
  - Deterministic single flow of control
- Sequential Cost Analysis Techniques
  - Costs assigned to primitives
- Clear/Succint Code
  - Programs expressed ~Algorithm level

Why fUML/ALF?

- Standard
  - UML is a de-facto standard in software industry and an ISO/IEC (19505) standard
  - fUML provides a precise execution semantics for a subset of UML
  - The Alf action language allows to express complex execution behaviours
- Platform-Independent
  - High-level and platform-independent essence inherited from UML
- Flexible
  - Seamless integration with UML and Profiles
- Executable
  - Different execution semantics → wide support for development activities
    - Interpretative/Compilative – Simulation and debugging
    - Translational – Target-specific deployment and execution
- Analysable

Challenges

- Implicit Parallelism in Alf
  - Introducing implicit data-parallel primitives in Alf
  - Currently, *@parallel* annotation in combination with *block* and *for* statements provide support for explicit parallelism
- fUML Mapping
  - Mapping data-parallel primitives to fUML without disrupting its execution semantics, which is already inherently concurrent for activities
- Object-oriented + Data-parallel = ?

Ongoing Work

- Alf Implementation
  - Xtext + LLVM Front-End
- Integrating Object-oriented and Data-parallel
  - Existing similar approaches in the literature (e.g. Scala)
- Modelling heterogeneous massively parallel architectures and software/hardware allocations using (f)UML and MARTE

Thank you!

Questions?

Data-parallel Primitives

- Element-wise Scalar Operation
- Parallel Read (Get Communication)
- Parallel Write (Send Communication)
- Replication (Flooding)
- Masking (Selection)
- Reduce
- Scan (Parallel Prefix)

Element-wise Scalar Operation

Take one (or several) data structure, and apply a ”scalar” operation to the respective elements in each position. The result is a new data structure.

<u>Example</u>

```
for all k in parallel do C[k] := A[k] + B[k]
```

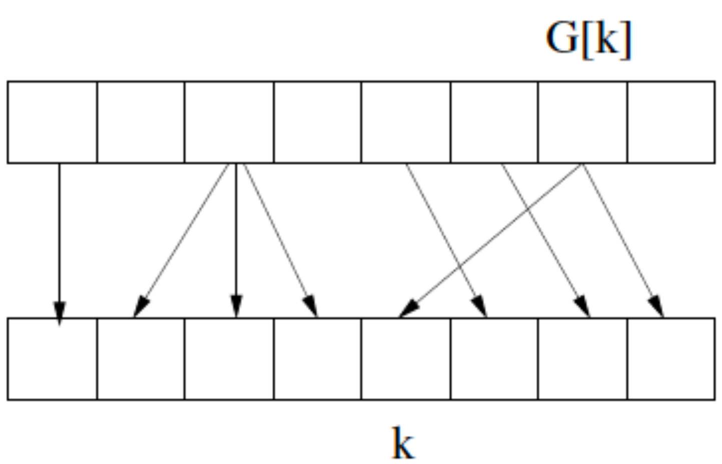
A:	7	9	0	3	22	1	2	–4
	+							
B:	1	2	4	8	16	32	64	128
	=							
C:	8	11	4	11	38	33	66	124

Parallel Read (Get Communication)

A parallel read operation, where each processor *k* reads the element of a data structure from some other processor *G[k]*:

<u>Example</u>

```
for all k in parallel do A[k] := B[G[k]]
```

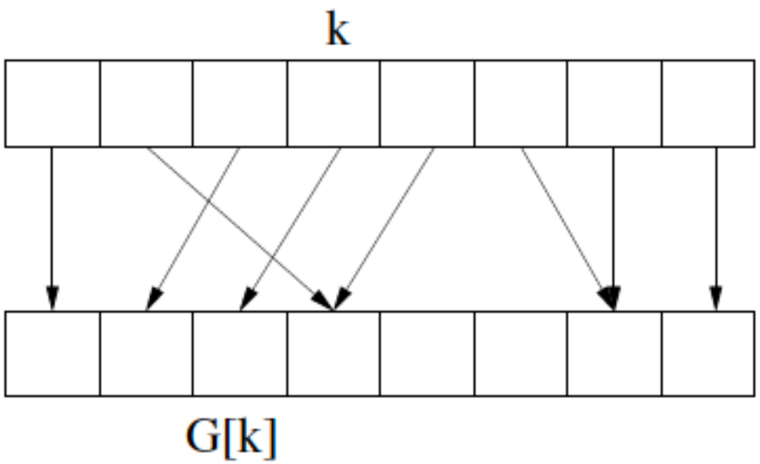


Parallel Write (Send Communication)

A parallel send (or write) operation, where each processor *k* sends the element of a data structure to some target processor *G[k]*:

<u>Example</u>

```
for all k in parallel do A[G[k]] := B[k]
```



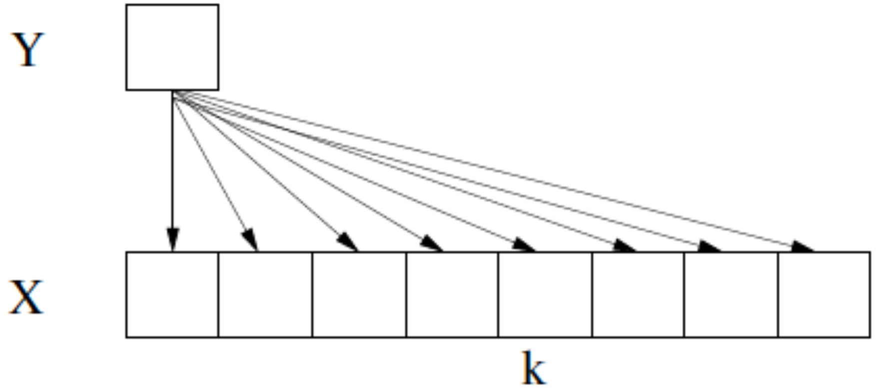
MÄLARDALEN UNIVERSITY SWEDEN

Replication (Flooding)

Replication means to duplicate a single piece of data to many processors, can be seen as special case of get communication.

<u>Example</u>

```
for all k in parallel do X[k] := Y
```



Masking (Selection)

Masking means to select a part of a data structure for some data parallel operation with respect to some boolean *mask* or *guard*.

Example

```
for all k where X[k] < 0 in parallel do X[k] := -X[k]
```

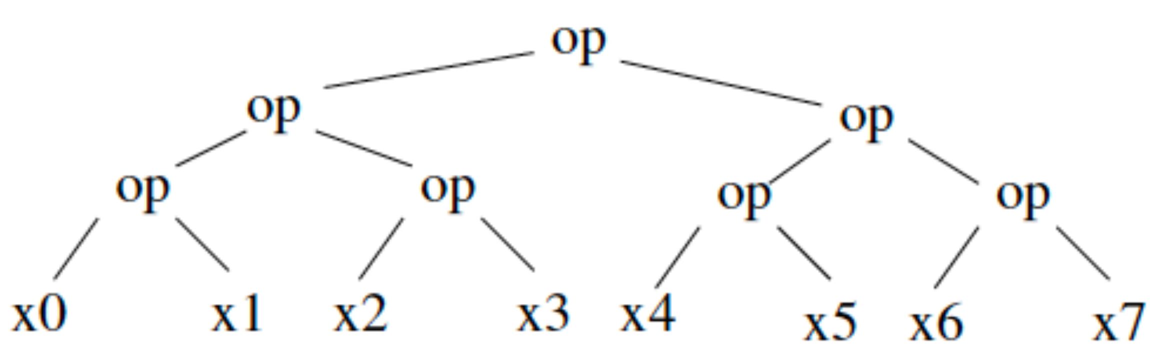
T	T	F	T	T	F	F	T	F	F	F	F	T	F	T	T
x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13	x14	x15	x16

Reduce

Let *op* be a binary, associative operation (e.g. add, min, etc) and *X* a data structure with positions *0,..,n-1* (e.g. array), then

$$reduce(op, X) = X[0]\ op\ \ldots\ op\ X[n-1]$$

Since *op* is associative, the evaluation can be done according to a balanced tree in *O(log n)* time with *O(n)* processors



Scan (Parallel Prefix)

Close relative to *reduce*, computes an array of all *partial sums*

$$scan(op, X) = [X[0], X[0]\ op\ X[1], \ldots, X[0]\ op \ldots op\ X[n-1]]$$

Also *scan* can be evaluated in *O(log n)* time on *O(n)* processors, according to a set of balanced binary trees with shared subtrees: