An Autonomous Data Language

Tom T.P. Franken[0000−0002−1168−5450], Thomas Neele[0000−0001−6117−9129], and

Jan Friso Groote[0000−0003−2196−6587]

Eindhoven University of Technology, Eindhoven, The Netherlands

Abstract. Nowadays, the main advances in computational power are due to parallelism. However, most parallel languages have been designed with a focus on processors and threads. This makes dealing with data and memory in programs hard, which distances the implementation from its original algorithm. We propose a new paradigm for parallel programming, the *data-autonomous* paradigm, where computation is performed by autonomous data elements. Programs in this paradigm are focused on making the data collaborate in a highly parallel fashion. We furthermore present AuDaLa, the first data autonomous programming language, and include an operational semantics. Programming in AuDaLa is very natural, as illustrated by examples, albeit in a style very different from sequential and contemporary parallel programming.

1 Introduction

As increasing the speed of sequential processing becomes more difficult [28], exploiting parallelism has become one of the main means of obtaining further performance improvements in computing. Thus, languages and frameworks aimed at parallel programming play an increasingly important role in computation. Many existing parallel languages use a *task-parallel* or a *data-parallel* paradigm [14].

Task-parallelism mostly focuses on the computation carried out by individual threads, scheduling tasks to threads depending on which threads are idle. In data-parallelism, threads execute the same function but are distributed over the data, thus performing a parallel computation on the collection of all data.

In a shared memory setting, programs in both paradigms require careful design of memory layout, memory access and movement of data to facilitate the threads used by the program. Examples of this are the use of barriers and data access based on thread id's, as well as access protocols. Not only is extensive data movement costly and hinders some performance optimizations [20, 22], the memory handling necessary throughout the entire program due to the focus on threads only widens the gap between algorithms and implementation as noted by for instance Leiserson *et al.* [28]. Therefore, to promote memory locality and more algorithmic code, a new data-focused paradigm is in order.

In this paper, we propose the new *data-autonomous* paradigm, where *data elements* not only locally store data and references, but also execute their own computations. Computations are always carried out in parallel by all data elements; this is governed by a *schedule*. Data elements can cooperate through

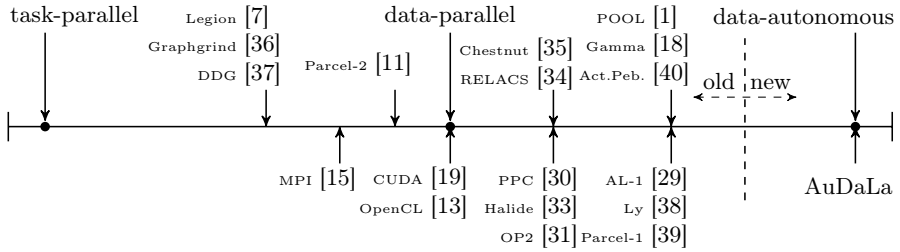

Fig. 1: Approximate placement of related work on an axis from process-focused (left) to data-focused (right) paradigms.

stored references. The paradigm completely abstracts away from processors and memory and is fully focused on data, compared to task- and data-parallelism (see Figure 1).

This provides several benefits. First, it results in a *separation of concerns*: code concerning data structures, algorithms and orchestration is properly separated. Furthermore, parallelism is encouraged by always running computations concurrently on groups of data elements. Finally, the paradigm promotes a bottom-up design process, from data structure to computations to schedule.

Contributions. As a first step towards developing the data-autonomous paradigm, we present *AuDaLa* (*Autonomous Data Language*), the first data-autonomous programming language. In AuDaLa, *structs*, *steps* and a *schedule* are responsible for data, computation and orchestration, respectively. We illustrate our thought process behind AuDaLa by means of a motivating example. We introduce AuDaLa programs for a few standard problems. Compared to programs taken from literature, our AuDaLa programs require less memory management and clearly separate data flow and orchestration.

In this work, we focus on providing a solid theoretical foundation of AuDaLa. Thus, we completely formalise AuDaLa's behaviour in an operational semantics, enabled by its compact syntax. Though we have a prototype compiler of AuDaLa to CUDA, discussing it is out of the scope of this paper.

Overview. We first present a motivating example and show the concepts of AuDaLa in Section 2. We then give the syntax of AuDaLa in Section 3 and a semantics in Section 4. We discuss more examples in Section 5. Lastly, we review related work in Section 6 and conclude in Section 7.

2 Concepts And Motivating Example

In this section, we first discuss the concepts of AuDaLa, and subsequently we design a program for the prefix sum problem in AuDaLa as a motivating example.

AuDaLa has three main components: *structs*, *steps* and a *schedule*. The relation between these components is shown in Figure 2. Structs are data type

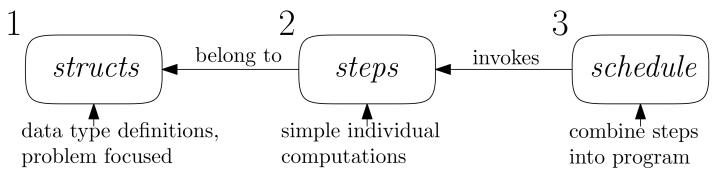

Fig. 2: The three main components of an AuDaLa program.

definitions from which data elements are instantiated during runtime. They contain the name of the data type and the parameters available to data elements of that type. See Listing 1.3 for an example of a struct definition. When starting an AuDaLa program, every struct gets a special *null*-instance, a data element representing the undefined instance of that struct. The parameters of this *null*-instance cannot be changed, but they otherwise function like normal data elements. A *null*-instance can be used for initialisation (since it already exists when launching the program) or as special value, for example to indicate the end of a list.

Each struct contains zero or more *steps*, which represent operations a data element instantiated from that struct can do. A step contains simple, algorithmic code, consisting of conditions and assignments, without loops. This makes steps easy to reason about. Within a step, it is possible to access the parameters of the surrounding struct and also to follow references stored in those parameters. Since these access patterns are known at compile-time, we can increase memory locality by grouping struct instances in a suitable manner.

The *schedule* prescribes an execution order on the steps. It contains step references and fixpoint operators (*Fix*). The occurrences of step references and fixpoint operators are separated by synchronization barriers ('<'). Execution only proceeds past a barrier when all computations that precede the barrier have concluded. Whenever a step occurs in the schedule, it is executed in parallel by all data elements which contain that step, although it is also possible to invoke a step for data elements of a specific type. AuDaLa programs are thus inherently parallel.

We do not make assumptions about a global execution order of statements executed in parallel. In particular, code is not executed by multiple struct instances in *lock-step*. Furthermore, we allow the occurrence of data races within one step, see also Section 5. Thus, barriers (and implicit barriers, see below) are the main method of synchronisation.

Iterative behaviour is achieved through a fixpoint operator, which executes its body repeatedly until an iteration occurs in which no data is changed. At this point, a fixpoint is reached and the schedule continues past the fixpoint operator. Between the iterations of a fixpoint, there is an implicit synchronisation barrier. For an example schedule, see Listing 1.4.

To give an example of these components in action, we consider the *prefix sum* problem: given a sequence of integers $x_1, \ldots, x_n$, we compute for each position $1 \leq k \leq n$ the sum $\Sigma_{i=1}^{k} x_i$. We have included OpenCL and CUDA implemen-

tations of the problem that previously occurred in the literature [13, 24], see Listings 1.1 and 1.2. Here, we omit the initialization to focus on the kernels. Both kernels require synchronization barriers in their algorithmic code, as well as an offset variable to check which data needs to be operated on, against which the thread ids need to be checked multiple times per execution.

```
kernel void koggeStone(const local T *in, local T *out) {
  out[tid] = in[tid];
  barrier();
  for (unsigned offset = 1; offset < n; offset *= 2){
    T temp;
    if (tid ≥ offset) temp = out[tid − offset];
    barrier();
    if (tid ≥ offset) out[tid] = temp ⊕ out[tid];
    barrier();
}}
```

Listing 1.1: OpenCL kernel for Prefix Sum (from [13])

```
__global__ void scan(float *g_odata, float *g_idata, int n){
  extern __shared__ float temp[];
  int thid = threadIdx.x;
  int pout = 0, pin = 1;
  temp[pout*n + thid] = (thid > 0) ? g_idata[thid−1] : 0;
  __syncthreads();
  for (int offset = 1; offset < n; offset *= 2){
    pout = 1 − pout; // swap double buffer indices
    pin = 1 − pout;
    if (thid >= offset)
      temp[pout*n+thid] += temp[pin*n+thid − offset];
    else
      temp[pout*n+thid] = temp[pin*n+thid]
    __syncthreads();
  }
  g_odata[thid] = temp[pout*n+thid];
}
```

Listing 1.2: CUDA kernel for Prefix Sum (or Scan) (from [24])

To design a corresponding AuDaLa program, we follow the design structure suggested in Figure 2. As before, we omit the initialization. In the prefix sum problem, the input is a sequence of integers. We model an element of this sequence with a struct *Position* containing a value *val*. We also give every *Position* a reference to the preceding *Position*, contained in parameter *prev*, as seen in Listing 1.3. This is needed to compute the prefix sum. The value of *prev* for the first position in the list is set to *null*, referencing the *null-Position*. This *null*-instance has the values 0 for *val* and *null* for *prev*.

```
struct Position(val: Int, prev: Position){ [...] }
```

Listing 1.3: Partial AuDaLa code for the structs for Prefix Sum

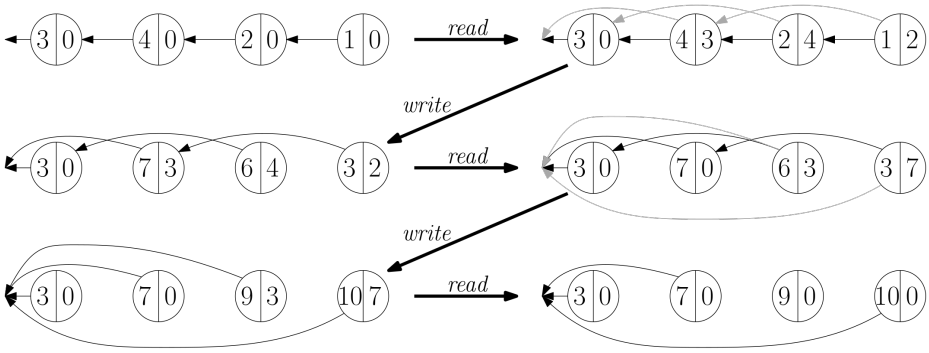

Fig. 3: Execution of Prefix Sum on a small list. The left side of a list element holds the parameter *val*, while the right side holds the parameter *auxval*. The parameter *prev* is shown as unmarked black arrows, while the parameter *auxprev* is shown as unmarked grey arrows.

In Listing 1.4 the steps *read* and *write* of the *Position* struct are shown. These steps are based on the method for computing prefix sum in parallel shown in Figure 3, which was introduced by Hillis and Steele [27]. Every *Position* first reads *prev.prev* and *prev.val* from their predecessor in the step *read*, and after synchronisation, every position updates their *prev* to *prev.prev* and their *val* to *prev.val* in the step *write*. As the scope of local variables in AuDaLa does not exceed a step, the use of additional parameters *auxprev* and *auxval* in the *read* step is required to recover the value in the *write* step. The steps do not need an *offset* variable like the CUDA and OpenCL kernels, as *Positions* which reached the beginning of the list have a *null*-instance as predecessor and can still execute the steps.

```
1  struct Position(val: Int, prev: Position, auxval: Int,  auxprev: Position){
2    read {                                                    /*step definition*/
3      auxval := prev.val;
4      auxprev := prev.prev;
5    }
6    write {                                                   /*step definition*/
7      val := val + auxval;
8      prev := auxprev;
9  }}
10
11 Fix(read < write)                                          /*schedule*/
```

Listing 1.4: AuDaLa code for Prefix Sum with steps and a schedule

For our program schedule, we want to repeat *read* and then *write* until all *Positions* have reached the beginning of the list, which results in the schedule as shown in Listing 1.4. Eventually, all *Positions* will have *null* as their predecessor and no parameters will change further, causing the fixpoint to terminate.

As illustrated by Listing 1.4 and by Figure 2, AuDaLa has a high *separation of concerns*: structs model data and their attributes, steps contain the algorithmic code and the schedule contains the execution. This approach requires no synchronization barriers in the user code for the steps, no variables to find the right indices for memory access and no offset variables to avoid going out of bounds.

3 Syntax

In this section, we highlight the most important parts of the concrete syntax of AuDaLa. In the definitions below, non-terminals are indicated with ⟨–⟩ and symbols with quotes; the empty word is ε. The non-terminal *Id* describes identifiers, and the non-terminal *Type* describes type names, which are either `Int`, `Nat` (natural number), `Bool`, `String` or an identifier (the name of a struct).

An AuDaLa *Program* consists of a list of definitions of *structs* and a schedule:

$$\langle Program \rangle ::= \langle Defs \rangle\ \langle Sched \rangle$$
$$\langle Defs \rangle ::= \langle Struct \rangle \mid \langle Struct \rangle\ \langle Defs \rangle$$

A *struct* definition gives the struct a type name (*Id*), a list of parameters (*Pars*) and a number of steps (*Steps*):

$$\langle Struct \rangle ::= \text{`}\texttt{struct}\text{'}\ \langle Id \rangle\ \text{`}\texttt{(}\text{'}\ \langle Pars \rangle\ \text{`}\texttt{)}\text{'}\ \text{`}\texttt{\{}\text{'}\ \langle Steps \rangle\ \text{`}\texttt{\}}\text{'},$$
$$\langle Pars \rangle ::= \langle Par \rangle \langle ParList \rangle \mid \varepsilon$$
$$\langle ParList \rangle ::= \text{`}\texttt{,}\text{'}\ \langle Par \rangle \langle ParList \rangle \mid \varepsilon$$
$$\langle Par \rangle ::= \langle Id \rangle\ \text{`}\texttt{:}\text{'}\ \langle Type \rangle$$

Steps are defined with a step name (*Id*) and a list of statements:

$$\langle Steps \rangle ::= \langle Id \rangle\ \text{`}\texttt{\{}\text{'}\ \langle Stats \rangle\ \text{`}\texttt{\}}\text{'}\ \langle Steps \rangle \mid \varepsilon$$
$$\langle Stats \rangle ::= \langle Stat \rangle\ \langle Stats \rangle \mid \varepsilon$$

A *statement* adheres to the following syntax:

$\langle Stat \rangle ::= \text{`}\texttt{if}\text{'}\ \langle Exp \rangle\ \text{`}\texttt{then}\text{'}\ \text{`}\texttt{\{}\text{'}\ \langle Stats \rangle\ \text{`}\texttt{\}}\text{'}$	if-then statement
$\mid \langle Type \rangle\ \langle Id \rangle\ \text{`}\texttt{:=}\text{'}\ \langle Exp \rangle\ \text{`}\texttt{;}\text{'}$	variable assignment
$\mid \langle Var \rangle\ \text{`}\texttt{:=}\text{'}\ \langle Exp \rangle\ \text{`}\texttt{;}\text{'}$	variable update
$\mid \langle Id \rangle\ \text{`}\texttt{(}\text{'}\ \langle Exps \rangle\ \text{`}\texttt{)}\text{'}\ \text{`}\texttt{;}\text{'}$	constructor statement

The *Id* in the variable assignment is a variable name. The constructor statement spawns a new data element of the type determined by *Id*, with parameter values determined by the expressions *Exps*. The syntax of *Exps* is similar to that of *Pars*, using *ExpList* and *Exp*. The syntax for a single *expression Exp* is as follows:

⟨*Exp*⟩ ::=	⟨*Exp*⟩ ⟨*BOp*⟩ ⟨*Exp*⟩	binary operator expression
	‘`(`’ ⟨*Exp*⟩ ‘`)`’	brackets
	‘`!`’ ⟨*Exp*⟩	negation
	⟨*Id*⟩ ‘`(`’ ⟨*Exps*⟩ ‘`)`’	constructor expression
	⟨*Var*⟩	variable expression
	⟨*Literal*⟩	literal expression
	‘`null`’	null expression
	‘`this`’	this expression

A *variable* reference follows the syntax:

$$\langle \mathit{Var} \rangle ::= \langle \mathit{Id} \rangle \text{ ‘.’} \langle \mathit{Var} \rangle \mid \langle \mathit{Id} \rangle,$$

where in the first case the *Id* is the name of a struct. Through the first case, one can access the parameters of parameters. For example, *prev.prev.val* would have been valid AuDaLa in Listing 1.4, and would access the value of the *Position* before the previous *Position* of the current *Position*.

Lastly, the schedule consists of the variants as given in the following syntax:

⟨*Sched*⟩ ::=	⟨*Id*⟩	step execution
	⟨*Id*⟩ ‘`.`’ ⟨*Id*⟩	typed step execution
	⟨*Sched*⟩ ‘`<`’ ⟨*Sched*⟩	barrier composition
	‘`Fix`’ ‘`(`’ ⟨*Sched*⟩ ‘`)`’	fixpoint calculation

The *Id* in the step execution is a step name. In the typed step execution, the first *Id* is a type name, while the second is a step name. The typed step execution is used to schedule a step executed by only one specific struct type.

On top of this concrete syntax, we adopt a number of additional requirements for an AuDaLa program to be well-formed. First of all, we have a number of usual sanity requirements, including ‘identifiers may not be keywords’, ‘a step name is declared at most once within each struct definition’, ‘names of local variables do not overlap with parameter names of the surrounding struct definition’, ‘local variables are not accessed from outside their surrounding struct definition’, and ‘local variables are not used before they are declared in a step’. Furthermore, we also assume common rules for well-typedness, so that binary operators are applied to the right types, the types in assignments and variable declarations are equal and constructor calls use the right type of arguments.

4 Semantics

In this section, we present the semantics of AuDaLa. Here, we regularly use lists. List concatenation is denoted with a semicolon, and we identify a singleton list with its only element. The empty list is denoted ε. Schedules are expressed as a list, e.g. the schedule $A < \mathit{Fix}(B)$ is expressed as $A; \mathit{Fix}(B)$.

We define updates for functions as follows. Given a function $f : A \to B$ and $a \in A$ and $b \in B$, then $f[a \mapsto b](a) = b$ and $f[a \mapsto b](x) = f(x)$ for all $x \neq a$. We lift this operation to sets of updates: $f[\{a_1 \mapsto b_1, a_2 \mapsto b_2, \dots\}] = f[a_1 \mapsto$

$b_1][a_2 \mapsto b_2] \ldots$. Since the order of applying updates is relevant, this is only well-defined if the left-hand sides $a_1, a_2, \ldots$ are pairwise distinct. If B contains tuples, that is, $B = B_1 \times \ldots \times B_n$, we can also update a single element of a tuple: if $f(a) = \langle b_1, \ldots, b_n \rangle$, then we define $f[a, i \mapsto b](a) = \langle b_1, \ldots, b_{i-1}, b, b_{i+1}, \ldots, b_n \rangle$ and $f[a, i \mapsto b](x) = f(x)$ for all $x \neq a$.

We assume the existence of a parser and typechecker for the concrete syntax. Henceforth, we work on an *abstract syntax tree* (AST) produced by running the parser and typechecker on a program. We thus do not concern ourselves with operator precedence and parentheses, and we assume that polymorphic elements such as `null` and 42 are labelled with the right type for their context, *viz.*, $null_T$ is the expression *null* of type T.

We have a number of sets containing AST elements: ID is the set of all identifiers, LT is the set of all literals, SC is the set of all schedules, ST is the set of all statements, E contains all expressions and O contains all syntactic binary operators. The set containing all syntactic types is $\mathcal{T} = \{\texttt{Nat}, \texttt{Int}, \texttt{Bool}, \texttt{String}\} \cup ID$.

In our semantics, *labels* reference concrete instances of structs (as opposed to struct definitions). We assume some sufficiently large set $\mathcal{L}$ containing these labels. We also have the semantic types $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{B}$ and *String* corresponding to the natural numbers, the integers, the booleans and the set of all strings, respectively. All semantic values are collected in $\mathcal{V} = \mathcal{L} \cup \mathbb{N} \cup \mathbb{Z} \cup \mathbb{B} \cup String$. The semantic value of a literal $g \in LT$ is $val(g)$.

In addition, we assume for every struct type sL the existence of a *null-label* $\ell^0_{sL} \in \mathcal{L}$, so that we can provide a default value for each syntactical type with the function $defaultVal : \mathcal{T} \to \mathcal{V}$, defined as:

$$defaultVal(T) = \begin{cases} 0 & \text{if } T = \texttt{Nat} \text{ or } T = \texttt{Int} \\ false & \text{if } T = \texttt{Bool} \\ \varepsilon & \text{if } T = \texttt{String} \\ \ell^0_T & \text{if } T \in ID \end{cases} .$$

We define the set of all *null*-labels to be $\mathcal{L}^0$, with $\mathcal{L}^0 \subset \mathcal{L}$.

To facilitate conciseness in our operational semantics, we break down statements and expressions into *commands*: atomic actions in the semantics.

Definition 1 (Commands). *A* command c *is constructed according to the following grammar:*

$$c ::= \mathbf{push}(val) \mid \mathbf{rd}(v) \mid \mathbf{wr}(v) \mid \mathbf{cons}(v) \mid \mathbf{if}(C) \mid \mathbf{not} \mid \mathbf{op}(o)$$

where $val \in \mathcal{V} \cup \{\mathbf{this}\}$ *is a semantic value or* **this**, *a special value,* $v \in ID$ *is an identifier,* C *is a list of commands, and* $o \in O$ *is an operator. The set of all commands is* $\mathcal{C}$.

Intuitively, **this** is the semantic equivalent to the syntactic `this`-expression. The precise effect of each command is discussed later in this section when the inference rules are given. Statements and expressions are compiled into a list of commands according to the following recursive interpretation function:

Definition 2 (Interpretation function). *Let $v, v_1, \ldots, v_n \in ID$ be variables, $a, a_1, \ldots, a_m \in E$ expressions, $g \in LT$ a literal, $sL \in ID$ a struct type, $s \in ST$ a statement, $S \in ST^*$ a list of statements, $T \in \mathcal{T}$ a type and $\texttt{op} \in O$ an operator from the syntax. Let the list $v_1; ...; v_n$ be the list of n variables from v_1 to v_n. We define the* interpretation function *$[\![\cdot]\!] : ST^* \cup E \to \mathcal{C}^*$ transforming a list of statements into a list of commands:*

$$
\begin{aligned}
[\![g]\!] &= \mathbf{push}(val(g)) \\
[\![\texttt{this}]\!] &= \mathbf{push}(\mathbf{this}) \\
[\![\texttt{null}_T]\!] &= \mathbf{push}(defaultVal(T)) \\
[\![v_1; \ldots; v_n]\!] &= \mathbf{push}(\mathbf{this}); \mathbf{rd}(v_1); \ldots; \mathbf{rd}(v_n) \\
[\![\texttt{!a}]\!] &= [\![a]\!]; \mathbf{not} \\
[\![\texttt{a}_1\ \texttt{op}\ \texttt{a}_2]\!] &= [\![a_1]\!]; [\![a_2]\!]; \mathbf{op}(\texttt{op}) \\
[\![\texttt{if}\ a\ \texttt{then}\{S\}]\!] &= [\![a]\!]; \mathbf{if}([\![S]\!]) \\
[\![T\ \texttt{v} := \texttt{a}]\!] &= [\![\texttt{v} := \texttt{a}]\!] \\
[\![\texttt{v}_1; \ldots; \texttt{v}_n; \texttt{v} := \texttt{a}]\!] &= [\![a]\!]; [\![v_1; \ldots; v_n]\!]; \mathbf{wr}(v) \\
[\![sL(\texttt{a}_1; \ldots; \texttt{a}_m)]\!] &= [\![a_1]\!]; \ldots; [\![a_m]\!]; \mathbf{cons}(sL) \\
[\![\varepsilon]\!] &= \varepsilon \\
[\![s; S]\!] &= [\![s]\!]; [\![S]\!]
\end{aligned}
$$

During the runtime of a program, multiple instances of a struct definition may exist simultaneously. We refer to these as *struct instances*.

Definition 3 (Struct instance). *A* struct instance *is a tuple $\langle sL, \gamma, \chi, \xi \rangle$ where:*
- *$sL \in ID$ is the type of the struct,*
- *$\gamma \in \mathcal{C}^*$ is a list of commands that are to be executed,*
- *$\chi \in \mathcal{V}^*$ is a stack that stores values during the evaluation of an expression,*
- *$\xi : ID \to \mathcal{V}$ is an environment that stores the values of local variables as well as parameters.*

We define $\mathcal{S}$ as the set of all possible struct instances.

A state of a program is the combination of a schedule that remains to be executed, a collection with all the struct instances that currently exist and a stack of Boolean values that are required to determine whether a fixpoint has been reached. Note that every label can refer to at most one distinct struct instance.

Definition 4 (State). *A* state *is a tuple $\langle Sc, \sigma, s\chi \rangle$, where:*
- *$Sc \in SC$ is a schedule expressed as a list,*
- *$\sigma : \mathcal{L} \to \mathcal{S} \cup \{\bot\}$ is a* struct environment*,*
- *$s\chi \in \mathbb{B}^*$ is a* stability stack*.*

The set of all states is defined as $S_\mathcal{G} = SC \times (\mathcal{L} \to \mathcal{S} \cup \{\bot\}) \times \mathbb{B}^*$.

With a notion of states and struct instances, we define *null-instances*:

Definition 5 (Null-instances). *Let $St = \langle Sc, \sigma, s\chi \rangle \in S_\mathcal{G}$ be a state. Then the set of* null-instances *in state St is defined as $\{\sigma(\ell) \mid \sigma(\ell) \neq \bot \wedge \ell \in \mathcal{L}^0\}$.*

Thus, each struct instance that is labelled with a *null*-label is a *null*-instance.

Henceforth, we fix an AuDaLa program $\mathcal{P}$ and define $SL_{\mathcal{P}} \subseteq ID$ to be the set of all struct types defined in $\mathcal{P}$. The initial variable environment for a struct instance of type sL is ξ^0_{sL}, defined as $\xi^0_{sL}(p) = \mathit{defaultVal}(T)$ for all $p \in Par_{sL}$ where T is the type of p and Par_{sL} refers to the parameters of sL. For other variables $v \in ID$, $\xi^0_{sL}(v)$ is left arbitrary. Recall that $Sc_{\mathcal{P}}$ is the schedule defined in $\mathcal{P}$.

The initial state of a graph machine program depends on which program is going to be executed (the state space does depend on the program, *cf.* Def. 4):

Definition 6 (Initial state). *The* initial state *of* $\mathcal{P}$ *is* $P^0_{\mathcal{P}} = \langle Sc_{\mathcal{P}}, \sigma^0_{\mathcal{P}}, \varepsilon\rangle$, *where* $\sigma^0_{\mathcal{P}}(\ell^0_{sL}) = \langle sL, \varepsilon, \varepsilon, \xi^0_{sL}\rangle$ *for all* $sL \in SL_{\mathcal{P}}$ *and* $\sigma^0_{\mathcal{P}}(\ell) = \bot$ *for all other labels.*

Intuitively, this definition states that the initial state of a program $\mathcal{P}$ consists of the schedule as found in the program, a struct environment filled with *null*-instances for every struct type declared in $\mathcal{P}$ and an empty stack.

We proceed by defining the transition relation $\Rightarrow$ by means of inference rules. There are rules that define the execution of commands and rules for the execution of a schedule. We start with the former. Command $\mathbf{push}(v)$ pushes value v on the stack χ, and $\mathbf{push}(\mathbf{this})$ pushes the label of the structure instance on χ:

$$(\mathbf{ComPush})\frac{\sigma(\ell) = \langle sL, \mathbf{push}(val); \gamma, \chi, \xi\rangle}{\langle Sc, \sigma, s\chi\rangle \Rightarrow \langle Sc, \sigma[\ell \mapsto \langle sL, \gamma, \chi; val, \xi\rangle], s\chi\rangle}$$

$$(\mathbf{ComPushThis})\frac{\sigma(\ell) = \langle sL, \mathbf{push}(\mathbf{this}); \gamma, \chi, \xi\rangle}{\langle Sc, \sigma, s\chi\rangle \Rightarrow \langle Sc, \sigma[\ell \mapsto \langle sL, \gamma, \chi; \ell, \xi\rangle], s\chi\rangle}$$

The command $\mathbf{rd}(v)$ reads the value of variable v from environment ξ' of ℓ' and places it onto the stack:

$$(\mathbf{ComRd})\frac{\begin{array}{c}\sigma(\ell) = \langle sL, \mathbf{rd}(v); \gamma, \chi; \ell', \xi\rangle \\ \sigma(\ell') = \langle sL', \gamma', \chi', \xi'\rangle\end{array}}{\langle Sc, \sigma, s\chi\rangle \Rightarrow \langle Sc, \sigma[\ell \mapsto \langle sL, \gamma, \chi; \xi'(v), \xi\rangle], s\chi\rangle}$$

For normal struct instances, $\mathbf{wr}(v)$ takes a label ℓ' and a value *val* from the stack and writes this to $\xi'(v)$, the environment of the struct instance corresponding to ℓ'. If v is a parameter and writing *val* changes its value, then any surrounding fixpoint in the schedule becomes unstable. In that case, we set the auxiliary value *su* (for *stability update*) to false and clear the stability stack by setting it to $s\chi_1 \wedge su; \ldots; s\chi_{|s\chi|} \wedge su$. Note that this leaves the stack unchanged if *su* is true. Below, in the update "$[\ell', 4 \mapsto \xi'[v \mapsto val]]$", recall that $f[a, i \mapsto b]$ denotes the update of a function that returns a tuple.

$$(\mathbf{ComWr})\frac{\begin{array}{c}\sigma(\ell) = \langle sL, \mathbf{wr}(v); \gamma, \chi; val; \ell', \xi\rangle \\ \sigma(\ell') = \langle sL', \gamma', \chi', \xi'\rangle \\ \ell' \notin \mathcal{L}^0 \vee v \notin Par_{sL'} \\ su = (v \notin Par_{sL'} \vee \xi'(v) = val)\end{array}}{\begin{array}{c}\langle Sc, \sigma, s\chi\rangle \Rightarrow \langle Sc, \sigma[\ell \mapsto \langle sL, \gamma, \chi, \xi\rangle][\ell', 4 \mapsto \xi'[v \mapsto val]], \\ s\chi_1 \wedge su; \ldots; s\chi_{|s\chi|} \wedge su\rangle\end{array}}$$

The next rule skips the write if the target is a parameter of a *null*-instance, which ensures that the parameters of a *null*-instance cannot be changed:

$$(\textbf{ComWrNSkip})\frac{\begin{array}{c}\sigma(\ell) = \langle sL, \textbf{wr}(v); \gamma, \chi; val; \ell', \xi\rangle \\ \sigma(\ell') = \langle sL', \gamma', \chi', \xi'\rangle \\ \ell' \in \mathcal{L}^0 \wedge v \in Par_{sL'}\end{array}}{\langle Sc, \sigma, s\chi\rangle \Rightarrow \langle Sc, \sigma[\ell \mapsto \langle sL, \gamma, \chi, \xi\rangle], s\chi\rangle}$$

A **not** command negates the top value of the stack χ:

$$(\textbf{ComNot})\frac{\sigma(\ell) = \langle sL, \textbf{not}; \gamma, \chi; b, \xi\rangle}{\langle Sc, \sigma, s\chi\rangle \Rightarrow \langle Sc, \sigma[\ell \mapsto \langle sL, \gamma, \chi; \neg b, \xi\rangle], s\chi\rangle}$$

An $\textbf{op}(o)$ command applies the semantic equivalent $\circ \in \{=, \neq, \leq, \geq, <, >, *, /, \%, +, -, \hat{}, \wedge, \vee\}$ of the syntactic operator $o \in O$ to the two values at the top of χ, of which the result is put on top of the stack:

$$(\textbf{ComOp})\frac{\sigma(\ell) = \langle sL, \textbf{op}(o); \gamma, \chi; a; b, \xi\rangle}{\langle Sc, \sigma, s\chi\rangle \Rightarrow \langle Sc, \sigma[\ell \mapsto \langle sL, \gamma, \chi; (a \circ b), \xi\rangle], s\chi\rangle}$$

Let sL' be the type of a struct with n parameters. The command $\textbf{cons}(sL')$ creates a new struct instance of type sL' in the struct environment σ with a fresh label ℓ', and initializes the parameters to the top n values of the stack:

$$(\textbf{ComCons})\frac{\begin{array}{c}\sigma(\ell) = \langle sL, \textbf{cons}(sL'); \gamma, \chi; p_1; \ldots; p_n, \xi\rangle \\ Par_{sL'} = par_1; ...; par_n \\ \sigma(\ell') = \bot\end{array}}{\begin{array}{l}\langle Sc, \sigma, s\chi\rangle \Rightarrow \langle Sc, \sigma[\{\ell \mapsto \langle sL, \gamma, \chi; \ell', \xi\rangle, \\ \quad \ell' \mapsto \langle sL', \varepsilon, \varepsilon, \xi^0_{sL'}[\{par_1 \mapsto p_1, \ldots, par_n \mapsto p_n\}]\rangle\}], false^{|s\chi|}\rangle\end{array}}$$

The command $\textbf{if}(C)$ with $C \in \mathcal{C}^*$ adds commands C to the start of γ if the top value of the stack is *true*. If the top value is *false*, the command does nothing:

$$(\textbf{ComIfT})\frac{\sigma(\ell) = \langle sL, \textbf{if}(C); \gamma, \chi; true, \xi\rangle}{\langle Sc, \sigma, s\chi\rangle \Rightarrow \langle Sc, \sigma[\ell \mapsto \langle sL, C; \gamma, \chi, \xi\rangle], s\chi\rangle}$$

$$(\textbf{ComIfF})\frac{\sigma(\ell) = \langle sL, \textbf{if}(C); \gamma, \chi; false, \xi\rangle}{\langle Sc, \sigma, s\chi\rangle \Rightarrow \langle Sc, \sigma[\ell \mapsto \langle sL, \gamma, \chi, \xi\rangle], s\chi\rangle}$$

In the remaining rules, let $Done(\sigma) = \forall \ell.(\sigma(\ell) = \bot \vee \exists sL, \chi, \xi.\sigma(\ell) = \langle sL, \varepsilon, \chi, \xi\rangle)$, and let F_1 be a (possibly empty) schedule. The predicate $Done(\sigma)$ holds when all commands have been executed in all struct instances in σ.

We can initiate steps globally and locally. The global step initiation converts all statements in a step to commands for any structure instance that has that step and adds the commands to γ. Let S^F_{sL} be the statements in a step with name F in a struct instance with type sL. Note that the schedule is expressed as

a list in the operational semantics, and is therefore separated by ';' as opposed to '<'.

$$(\textbf{InitG})\frac{Done(\sigma)}{\langle F; F_1, \sigma, s\chi\rangle \Rightarrow \langle F_1, \sigma[\{\ell \mapsto \langle sL_\ell, [\![S^F_{sL_\ell}]\!], \varepsilon, \xi_\ell\rangle \mid \sigma(\ell) = \langle sL_\ell, \gamma_\ell, \chi_\ell, \xi_\ell\rangle\}], s\chi\rangle}$$

The local step initiation converts the step to commands and adds those commands to γ only for struct instances of a specified struct x:

$$(\textbf{InitL})\frac{Done(\sigma)}{\langle x.F; F_1, \sigma, s\chi\rangle \Rightarrow \langle F_1, \sigma[\{\ell \mapsto \langle x, [\![S^F_x]\!], \varepsilon, \xi_\ell\rangle \mid \sigma(\ell) = \langle x, \gamma_\ell, \chi_\ell, \xi_\ell\rangle\}], s\chi\rangle}$$

Fixpoints are initiated when first encountered:

$$(\textbf{FixInit})\frac{Done(\sigma)}{\langle Fix(F); F_1, \sigma, s\chi\rangle \Rightarrow \langle F; aFix(F); F_1, \sigma, s\chi; true\rangle}$$

The symbol *aFix* is a semantic symbol used to denote a fixpoint which has been initiated. When an initiated fixpoint is encountered again, the stability stack is used to determine whether the body should be executed again:

$$(\textbf{FixIter})\frac{Done(\sigma)}{\langle aFix(F); F_1, \sigma, s\chi; false\rangle \Rightarrow \langle F; aFix(F); F_1, \sigma, s\chi; true\rangle}$$

$$(\textbf{FixTerm})\frac{Done(\sigma)}{\langle aFix(F); F_1, \sigma, s\chi; true\rangle \Rightarrow \langle F_1, \sigma, s\chi\rangle}$$

With these rules, we give an operational semantics for AuDaLa:

Definition 7 (Operational semantics). *The semantics of $\mathcal{P}$ is the graph $[\![\mathcal{P}]\!] = \langle S_\mathcal{G}, \Rightarrow, P^0_\mathcal{P}\rangle$, where $S_\mathcal{G}$ is the set of all states (Def. 4), $\Rightarrow$ is the transition relation as given above and $P^0_\mathcal{P}$ is the initial state of $\mathcal{P}$ (Def. 6).*

5 Standard Algorithms

In this section, we provide more intuition on how AuDaLa works in practice by means of two example AuDaLa programs. The first creates a spanning tree and the second is a sorting program.

5.1 Creating a spanning tree

Given a connected directed graph $G = (V, E)$ and a root node $u \in V$, we can create a spanning tree of G rooted in u using *breadth-first search*. In this tree, for every node v, the path from u to v is a shortest path in G. We do this by incrementally adding nodes from G with a higher distance to u to the spanning tree.

```
1  struct Node(dist: Int, in: Edge){}
2
3  struct Edge(s: Node, t: Node){
4    linkEdge{
5      if s.dist != −1 && t.dist == −1 then {
6        t.in := this;
7    }}
8    handleEdge {
9      if t.in != null then {
10       if t.in == this then {
11         t.dist := s.dist + 1;
12       }
13       if t.in != this then {
14         s := null;
15         t := null;
16   }}}}
17
18 Fix(linkEdge < handleEdge)
```

Listing 1.5: AuDaLa code for creating a spanning tree

We first sketch our approach. In the ith BFS iteration, the algorithm adds all edges (s, t) to the tree such that the distance from u to s is $i - 1$ and the distance from u to t is still unknown. If multiple such edges lead to the same t, the algorithm uses a *data race* to determine which edge is chosen. As any edge will suffice, this data race is benign. The distance from u to t is then set to i and we continue with the next iteration. The program runs with $O(|V| + |E|)$ data elements in $O(d)$ time, where d is the diameter of the graph.

Contained in Listing 1.5 is an AuDaLa program that implements this approach. The program defines the struct *Node* (line 1) with parameters *dist*, to store the distance from root node u, and *in*, a reference to its incoming spanning tree edge. The struct *Edge* (line 3) has a source s and a target t.

During initialization, the input should be a directed graph, with a root *Node* u with *dist* 0 and with the *dist* parameter of the other *Nodes* set to -1. For every *Node*, the parameter *in* should be *null*.

Both steps in the program belong to *Edge*. The first step, *linkEdge*, first determines whether an *Edge* e from *Node* s to *Node* t is at the frontier of the tree in line 5. This is the case when s is in the tree, but t is not. If so, e nominates itself as the *Edge* connecting t to the tree, $t.in$ (line 6). This is a data race won by only one edge for t, the edge which applies the semantic rule **ComWr** last. In the second step, *handleEdge*, if the nomination for t has finished (line 9) and e has won the nomination (line 10), e will update t's distance to the root. If e has lost, it will remove itself from the graph, here coded as setting the source and target parameters to *null* in lines 13 to 16.

To create a full spanning tree, this must be executed until all *Nodes* have a positive *dist* and all *Edges* are either $t.in$ for their target *Node* t or have *null* as their source and target. To this end, the schedule (line 18) contains a fixpoint,

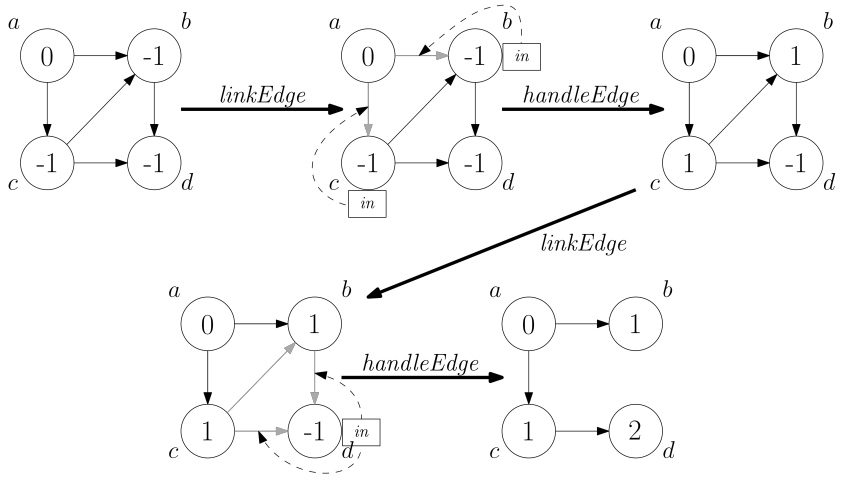

Fig. 4: Execution of Listing 1.5 on a small graph. Every *Edge* newly considered in the current step is grey. Considered *Edges* stay considered, but stable. The dotted arrows denote the possible new values for $t.in$ of a target node t. Note that the *Edge* from c to d wins the data race to the reference $d.in$.

in which *Edges* first nominate themselves and then update the distances of new *Nodes*. This fixpoint terminates, as *Edges* in the spanning tree will continuously update their targets with the same information and *Edges* which lost their nomination will not get past the first conditions of the two steps, causing the data elements to stabilize after all *Nodes* have received a distance from u. Initialization steps should be placed at the start of the shown schedule. An execution of the program on a small graph is shown in Figure 4, where the edges and nodes of the graph are modelled by their respective structs.

5.2 Sorting

```
1  struct ListElem(val: Int, next: ListElem, newNext: ListElem, comp: ListElem){
2    compareElement {
3      if comp != null then {
4        if (comp.val > val && (comp.val < newNext.val || newNext == null)) then
5        {newNext := comp;}
6        comp := comp.next;
7    }}
8    reorder {
9      next := newNext;
10 }}
11
12 Fix(compareElement) < reorder
```

Listing 1.6: AuDaLa code for sorting

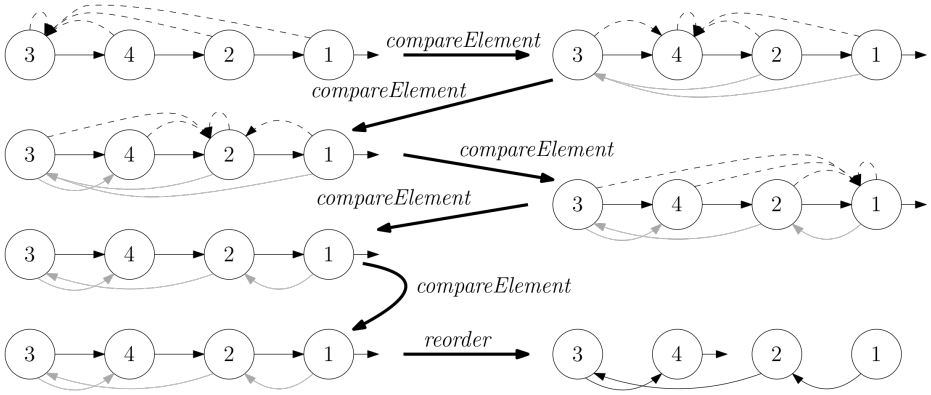

Fig. 5: Execution of Listing 1.6 on a small list. The parameters *next*, *newNext* and *comp* are shown as black, grey and dashed unmarked arrows respectively, and the *null*-references for *newNext* and *comp* are not shown. The nodes corresponding to *ListElems* contain the value of parameter *val*.

A concise example of a AuDaLa program for sorting a linked list of n elements can be found in Listing 1.6. In it, the elements of the list traverse the list together, during which each element e is looking for its successor in the sorted list. After the traversal, the successor element is saved and the link is updated to the saved element. This reorders the list to the sorted list. The program runs in $O(n)$ time with n data elements. We can achieve a time complexity of $O(\log n)$ by implementing Cole's algorithm [16] in AuDaLa, but that is outside the scope for this paper.

The program defines the struct *ListElem*, modeling the nodes of the list, with parameters *val*, *next*, a reference to the next *ListElem*, *newNext*, a reference to the *ListElem* that should come next in the sorted list, and *comp*, a reference to the current *ListElem newNext* is compared to. The initialization needs to make sure that every element has a distinct value, and that in every element, *comp* is set to the first element of the list and *newNext* is set to *null*.

To facilitate our strategy we give our *ListElem* two steps, one to check an element in the list called *compareElement* and one to reorder the list at the end called *reorder*. With the step *compareElement*, an element checks whether the element to which the *comp* reference leads is a better next element than the current element saved in *newNext* (line 4) and updates *newNext* if that is the case. Afterwards, the *comp* reference is updated to the next element in the list (line 6). With the *reorder* step, an element replaces their old *next* reference with *newNext* (line 9).

To have the program execute our strategy, we call a fixpoint on *compareElement*, such that every element checks all elements in the list. After that is done, the schedule tells the elements to *reorder* (line 12).

6 Related Work

Conceptually, our work is related to the Parallel Pointer Machine (PPM) [23], which models memory as a graph that is traversed by processors. In AuDaLa, on the other hand, processors are implicit and data is the main focus.

The concept of cooperating data elements is present in the Chemical Abstract Machine [8], based on the Γ-language [5, 6]. In the data-autonomous paradigm these components are coordinated by a schedule as opposed to the Chemical Abstract Machine, where the data elements float around freely. By extension, AuDaLa is related to the Γ-Calculus Parallel Programming Framework [18].

The data-autonomous paradigm shares the same focus on data as *message passing* languages like Active Pebbles [40], ParCel-2 [11] and AL-1 [29], but differs in using shared variables instead of synchronisation and messages. It also does not allow the use of data as passive elements, like in the messages of MPI [15].

The *specialist-parallel* approach [12] models a problem as a network of relatively autonomous nodes which perform one specified task. In comparison, the data-autonomous paradigm defines their specialists around data instead of tasks and data elements perform multiple or no tasks depending on their steps.

In AuDaLa, the relations between data elements can be viewed as a graph, which is also the case for *graph based* languages, such as DDG [37], a scheduling language, and GraphGrind [36], a graph partitioning language. The Connection Machine [26] uses a graph-based hardware architecture for parallel computation. Similarly, the way data is expressed in Legion [7] and OP2 [31] is similar to AuDaLa. However, these two languages work top down from a main process that calls functions on data, which is unlike the data-autonomous paradigm.

Since the data-autonomous paradigm extends data-parallelism (see Figure 1), AuDaLa shares concepts with other data-parallel languages like CUDA [19, 24] and OpenCL [13]. It has the most in common with object-oriented approaches to data-parallelism, like the POOL family of languages [1], languages in which small elements do parallel computations based on their neighbours, like ReLaCS [34], PPC [30], Chestnut [35] and the ParCel languages [11, 39], and *actor languages*.

Actor languages, like Ly [38], ParCel-1 [39], POb$C{+}{+}$ [32] and A-NETL [4], treat objects as independent, collaborating actors, in a similar way as how the data-autonomous paradigm treats data. Often, these languages use the *message passing* model to cooperate, which AuDaLa does not. Of those who do not, OpenABL [17] uses agents similar to data elements, but gives the agents to functions instead of functions to agents. *Active Object* languages [9, 10] do give their objects functions, which is very closely related to data elements. The execution of functions in these objects however, is fully asynchronous: objects can activate other objects by calling methods in them for them to execute. This is less structured than in AuDaLa, in which the functions to be executed are defined in the schedule. As a result, AuDaLa does not use futures, unlike most active object languages.

The use of a schedule in the data-autonomous paradigm relates AuDaLa to some more functional data-parallel languages as well, like Halide [33], which uses a schedule as well, and even Futhark [25], in which the manipulation of

an array has some similarity to calling a step in AuDaLa. The schedule can be considered as a coordination language [2] for the paradigm and AuDaLa, but is fully integrated and required for both to function. It also does not need to create channels between components, like for example Reo [3].

Similar to our motivation, ICE [21], which is a framework for implementing PRAM algorithms, sets the goal of bridging the gap between algorithms and implementation. However, as ICE is based on a PRAM, it is not data autonomous.

7 Conclusion

In this paper, we presented the data-autonomous paradigm and introduced it by means of the Autonomous Data Language, by giving examples of standard algorithms and discussing the syntax and semantics.

In the future, we will extend these foundations in multiple directions. First, we plan to perform an extensive practical evaluation of AuDaLa. Currently, we have prototypes of a sequential interpreter and a compiler to CUDA (for parallel execution on GPUs). Using these, we will investigate methods for efficient parallel execution of AuDaLa programs. Based on these experiences we may further extend the language and semantics, for example by introducing variants of the fixpoint operator.

On the theoretical side, one immediate avenue of research is to determine the expressivity of the language, which we have started to investigate. We also plan on creating formal analysis methods for AuDaLa programs, including methods for finding data races in AuDaLa programs and methods for proving functional correctness. For finding the data-races, we have already laid the groundwork in the operational semantics. We may also investigate how extensions to the current semantics impact the design of the envisioned formal analyses.

Acknowledgments We would like to thank the AVVA project members for their insights and comments and Gijs Leemrijse, Clemens Dubslaff, Erik de Vink and the reviewers for their feedback.

References

1. America, P., van der Linden, F.: A parallel object-oriented language with inheritance and subtyping. SIGPLAN Not. **25**(10), 161–168 (1990). https://doi.org/10.1145/97946.97966
2. Arbab, F., Ciancarini, P., Hankin, C.: Coordination languages for parallel programming. Parallel Computing **24**(7), 989–1004 (1998). https://doi.org/10.1016/S0167-8191(98)00039-8
3. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science **14**(3), 329–366 (2004). https://doi.org/10.1017/S0960129504004153
4. Baba, T., Yoshinaga, T.: A-NETL: a language for massively parallel object-oriented computing. In: PMMPC Proc. pp. 98–105. IEEE (1995). https://doi.org/10.1109/PMMPC.1995.504346

5. Banâtre, J.P., Coutant, A., Le Metayer, D.: A parallel machine for multiset transformation and its programming style. Future Generation Computer Systems **4**(2), 133–144 (1988). https://doi.org/10.1016/0167-739X(88)90012-X
6. Banâtre, J.P., Le Métayer, D.: The gamma model and its discipline of programming. Science of Computer Programming **15**(1), 55–77 (1990). https://doi.org/10.1016/0167-6423(90)90044-E
7. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing locality and independence with logical regions. In: SC '12. pp. 1–11 (2012). https://doi.org/10.1109/SC.2012.71
8. Berry, G., Boudol, G.: The chemical abstract machine. Theor. Comput. Sci. **96**(1), 217–248 (Apr 1992). https://doi.org/10.1016/0304-3975(92)90185-I
9. de Boer, F., et al.: A Survey of Active Object Languages. ACM Comput. Surv. **50**(5), 76:1–76:39 (2017). https://doi.org/10.1145/3122848
10. de Boer, F.S., Clarke, D., Johnsen, E.B.: A Complete Guide to the Future. In: Programming Languages and Systems. pp. 316–330. Springer (2007). https://doi.org/10.1007/978-3-540-71316-6_22
11. Cagnard, P.J.: The ParCeL-2 Programming Language. In: Euro-Par 2000. pp. 767–770. LNCS, Springer (2000). https://doi.org/10.1007/3-540-44520-X_106
12. Carriero, N., Gelernter, D.: How to write parallel programs: a guide to the perplexed. ACM Comput. Surv. **21**(3), 323–357 (1989). https://doi.org/10.1145/72551.72553
13. Chong, N., Donaldson, A.F., Ketema, J.: A sound and complete abstraction for reasoning about parallel prefix sums. SIGPLAN Not. **49**(1), 397–409 (2014). https://doi.org/10.1145/2578855.2535882
14. Ciccozzi, F., et al.: A Comprehensive Exploration of Languages for Parallel Computing. ACM Comput. Surv. **55**(2), 24:1–24:39 (2022). https://doi.org/10.1145/3485008
15. Clarke, L., Glendinning, I., Hempel, R.: The MPI Message Passing Interface Standard. In: Programming Environments for Massively Parallel Distributed Systems. pp. 213–218. Monte Verità, Birkhäuser (1994). https://doi.org/10.1007/978-3-0348-8534-8_21
16. Cole, R.: Parallel Merge Sort. SIAM J. Comput. **17**, 770–785 (1988). https://doi.org/10.1137/0217049
17. Cosenza, B., et al.: OpenABL: A Domain-Specific Language for Parallel and Distributed Agent-Based Simulations. In: Euro-Par 2018. pp. 505–518. LNCS, Springer (2018). https://doi.org/10.1007/978-3-319-96983-1_36
18. Gannouni, S.: A Gamma-calculus GPU-based parallel programming framework. In: WSWAN Proc. pp. 1–4. IEEE (2015). https://doi.org/10.1109/WSWAN.2015.7210299
19. Garland, M., et al.: Parallel Computing Experiences with CUDA. IEEE Micro **28**(4), 13–27 (2008). https://doi.org/10.1109/MM.2008.57
20. Geist, A., Reed, D.A.: A survey of high-performance computing scaling challenges. Int. J. High Perform. Comput. Appl. **31**(1), 104–113 (2017). https://doi.org/10.1177/1094342015597083
21. Ghanim, F., Vishkin, U., Barua, R.: Easy PRAM-Based High-Performance Parallel Programming with ICE. IEEE Trans. Parallel Distrib. Syst. **29**(2), 377–390 (2018). https://doi.org/10.1109/TPDS.2017.2754376
22. Giles, M.B., Reguly, I.: Trends in high-performance computing for engineering calculations. Phil. Trans. R. Soc. A. **372**(2022) (2014). https://doi.org/10.1098/rsta.2013.0319

23. Goodrich, M.T., Kosaraju, S.R.: Sorting on a parallel pointer machine with applications to set expression evaluation. J. ACM **43**(2), 331–361 (1996). https://doi.org/10.1145/226643.226670
24. Harris, M., Sengupta, S., Owens, J.D.: Parallel Prefix Sum (Scan) with CUDA. GPU gems **3**(39), 851–876 (2007)
25. Henriksen, T., et al.: Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In: PLDI 2017. pp. 556–571. PLDI 2017, ACM (2017). https://doi.org/10.1145/3062341.3062354
26. Hillis, W.D.: The connection machine. MIT Press, Cambridge, Mass (1989)
27. Hillis, W.D., Steele, G.L.: Data parallel algorithms. Commun. ACM **29**(12), 1170–1183 (1986). https://doi.org/10.1145/7902.7903
28. Leiserson, C.E., et al.: There's plenty of room at the Top: What will drive computer performance after Moore's law? Science **368**(6495), eaam9744 (2020). https://doi.org/10.1126/science.aam9744
29. Marcoux, A., Maurel, C., Salle, P.: AL 1: a language for distributed applications. In: FTDCS1990 Workshop Proc. pp. 270–276. IEEE (1988). https://doi.org/10.1109/FTDCS.1988.26707
30. Maresca, M., Baglietto, P.: A programming model for reconfigurable mesh based parallel computers. In: PMMPC Workshop Proc. pp. 124–133. IEEE (1993). https://doi.org/10.1109/PMMP.1993.315547
31. Mudalige, G., Giles, M., Reguly, I., Bertolli, C., Kelly, P.: OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In: InPar 2012. pp. 1–12 (2012). https://doi.org/10.1109/InPar.2012.6339594
32. Pinho, E.G., de Carvalho, F.H.: An object-oriented parallel programming language for distributed-memory parallel computing platforms. Science of Computer Programming **80**, 65–90 (2014). https://doi.org/10.1016/j.scico.2013.03.014
33. Ragan-Kelley, J., et al.: Halide: decoupling algorithms from schedules for high-performance image processing. Commun. ACM **61**, 106–115 (2017). https://doi.org/10.1145/3150211
34. Raimbault, F., Lavenier, D.: RELACS for systolic programming. In: ASAP Proc. pp. 132–135. IEEE (1993). https://doi.org/10.1109/ASAP.1993.397128
35. Stromme, A., Carlson, R., Newhall, T.: Chestnut: a GPU programming language for non-experts. In: PMAM Proc. pp. 156–167. ACM (2012). https://doi.org/10.1145/2141702.2141720
36. Sun, J., Vandierendonck, H., Nikolopoulos, D.S.: GraphGrind: addressing load imbalance of graph partitioning. In: ICS Proc. pp. 1–10. ACM (2017). https://doi.org/10.1145/3079079.3079097
37. Tran, V., Hluchy, L., Nguyen, G.: Parallel programming with data driven model. In: EMPDP Proc. pp. 205–211. IEEE (2000). https://doi.org/10.1109/EMPDP.2000.823413
38. Ungar, D., Adams, S.S.: Harnessing emergence for manycore programming: early experience integrating ensembles, adverbs, and object-based inheritance. In: OOPSLA Proc. pp. 19–26. ACM (2010). https://doi.org/10.1145/1869542.1869546
39. Vialle, S., Cornu, T., Lallement, Y.: ParCeL-1: a parallel programming language based on autonomous and synchronous actors. SIGPLAN Not. **31**(8), 43–51 (1996). https://doi.org/10.1145/242903.242945
40. Willcock, J.J., Hoefler, T., Edmonds, N.G., Lumsdaine, A.: Active pebbles: parallel programming for data-driven applications. In: ICS Proc. p. 235. ACM (2011). https://doi.org/10.1145/1995896.1995934