Flux - A Data-Flow Programming Language

R. Ispas*, L. Negreanu**

**RAI Software, Bucharest*
Romania (Tel: +40 722 475 870; e-mail: rares@raisoftware.ro).
***University Politehnica of Bucharest, Bucharest, Romania (e-mail: lorina.negreanu@cs.pub.ro)*

Abstract: The goal of this paper is to establish the design requirements of a programming language enabled to extract the maximum parallelism from source code. The structure of imperative and functional languages are analyzed in regards to parallelism. Next, the data-flow paradigm is introduced. Its major obstacles are identified and solutions are provided based on compile-time analysis. The resulting language should enable wide-scale parallelism, scaling from instruction to cluster.

Keywords: parallel programming languages, data-flow parallel programming languages, type systems, processes, control systems, neural networks

1. INTRODUCTION

There is a wealth of computing architectures, like GPUs, mesh-processors, Transport Triggered Architectures, FPGAs, which offer much higher performance potential than modern CPUs, but which are underutilized because of the difficulty of their programming.

We propose a programming language (Flux) enabled to extract the maximum parallelism from source code, scaling from instruction to cluster. The imperative and functional paradigms are analyzed in regards to parallelism, and they are found structurally wanting. A thorough analysis of value-types and reference-types reveals both paradigms can and should be unified under a common conceptual framework. The Data-Flow paradigm is presented as an alternative. Our form of Data-Flow allows unification of Imperative and Functional paradigms. Correctness and execution efficiency concerns are addressed using compile-time analysis. A generalized syntax based on hypergraphs is introduced. The resulting language should allow the maximum utilization of existing parallel hardware with the minimum amount of human effort and enables the development of the next generations of parallel architectures.

Our goal is to design and implement a data-flow programming language to be used as the workhorse of parallel programming. That is, develop a simple, efficient, minimal, programming language with automatic parallelism equivalent to what C was for the serial programming age. By automatic parallelism we understand a programming language feature which allows the compiler and runtime to decide what is parallelizable and does not require explicit management of execution resources and locks. We argue that C is inherently serial, its parallelism granularity is too high and had a bad influence on the development of parallel hardware, so it must be replaced. Development of advanced parallel architectures is blocked by the non-portability of existing code across architectures and the lack of automatic parallelization.

Automatic parallelism is essential should the same code be reusable on various parallel architectures, similar to what the optimizing Fortran compiler was for the porting of code amongst different machine instruction sets.

The paper is organized as follows. Section 2 introduces the constraints of a maximal parallel programming language. Section 3 discusses the existing approaches in terms of the imperative and functional paradigms. Section 4 presents Flux, the data-flow parallel programming language. Section 5 defines the syntax of the language. Section 6 discusses the utilization of Flux in the context of control systems. Section 7 concludes the paper.

2. THE MAXIMAL PARALLEL PROGRAMMING LANGUAGE

Assuming an ideal parallel processor with unlimited execution resources, an ideal parallel programming language would execute all the operations in the program simultaneously, yet arrive at the correct result every time. This is of course not possible, because operations depend logically on each other. The Maximal Parallel Programming Language (MPPL) would execute as many of the operations in parallel as possible, yet produce a correct result. The correctness of the result is left undefined on purpose, to allow for a case-by-case optimization.

The necessary constraints of a MPPL running on an ideal parallel processor are surprisingly few: (a) data dependencies, a calculation cannot be performed if all its operands are not available yet; (b) control dependencies, a calculation should count in the final result only if its execution exclusively depends on a control branch (IF, SWITCH, etc) which is chosen. On a finite processor another constraint appears. The execution resource dependency blocks the execution of an

operation until an execution unit capable of executing that operation becomes available.

3. EXISTING APPROACHES

We considered the following programming languages in our search for a superior parallel language: C++, D, Java, LISP, Haskell, OCCAM, Erlang, Rust, LUCID, BitC, Go, VHDL.

3.1 Imperative Programming Languages Are Scrambling Programmer's Intent

Let's consider C, the measuring stick of all imperative programming languages (IPLs). C is considered a serial execution language, but modern super-scalar and SIMD processors cannot attain their maximum level of performance without having the compiler back-end do extremely difficult program analysis beforehand.

In C, the execution order of all instructions within a function is fixed at programming time. Statements are executed sequentially, precisely in the order of statements delimited by semicolons, regardless of data dependencies requirements. More precisely, EVERY instruction is in a implicit control dependency relationship with the previous one in the block. Attempting to remove that implicit control dependencies will trigger hidden data dependencies created by the assignment order of variables, variables which are used both as message channels as well as data storage. Evaluating operations in the wrong order will deliver "messages" to the wrong operation, thus producing incorrect results.

There is a huge amount of research related to the untangling of real data dependencies from sequential code. While this 50 years old research domain has produced amazing results (SSA, PDG, pointer aliasing analysis, superscalar processors, etc), it is clear that its progresses have slowed to a stop. All the easy fruits have already been picked, and yet there is still significant parallelism hidden in the C code. The tremendous complexity of modern code analysis technologies (such as context-sensitive pointer aliasing) begets the question: Why flatten all the logic mesh on a line of statements, then attempt to reconstruct the original logical structure using sophisticated techniques, rather than preserve the ORIGINAL logic and dependencies, at programming time?

3.2 Functional Programming Considered Harmful

The next largest programming paradigm and the very best hope of the academic world is represented by the (pure) functional languages. They are called pure because calculations contain no side effects. Another perspective is that all operands are semantically Pass-By-Value, without references. This is commonly implemented by having each write operation create a new copy of the object.

The intended goal for this constraint is that the name of a value always represents the same value (same content) within the same scope, effectively describing totally defined data dependencies. Obtaining data-dependencies would enable automatic parallelism at compiler level. It turns out it is not so, and the reasons are subtle.

Firstly, totally defined data-dependencies are too restrictive when applied to complex data structures (like arrays and structures). In their case, pure functional languages will required defining a new name for a container for each modification of an internal member. As such, parallel assignment of the items in an array is impossible in a pure functional programming language. Yes, exactly the most parallelizable construct.

Secondly, inefficiencies generated by copy-semantics ruin irreparably the performance and are described in the next section.

To understand more of the structural limitations of pure functional languages, let us compare the properties of Value Types (VT) versus Reference Types (RT) (Table 1):

Table 1. Value Types vs Reference types

Characteristic	Value Types	Reference Types	Advantage
Deterministic Scheduling(result the same regardless of schedule ordering)	Total	Partial (parallel write activity may insert extra changes between explicit deps)	VT
Pointer Aliasing	No	Yes	VT
Parallel Writes	No	Yes	VT
Requires Locks	No	Yes	VT
Allocation	Automatic - Static or Stack	Explicit Dynamic on Heap	VT
Free	Automatic/Container Lifetime	Explicit or Garbage Collection	VT
Memory Requirements	Write_Branches_Count * sizeof(struct)	sizeof(struct)	RT
Extraneous Update Distribution Memory	Recursively Multiplied (1)	None	RT
Extraneous Update Distribution Processing	Explicitly Coded Reconstruction (2)	None	RT
Basic Efficient Implementation	Requires Special Structures or Flow-Analysis (to minimize effects above, but infeasible in the general case)	Trivial	RT

In pure functional programming the update of a member *V* contained within a structure *C*, contained in another object *B*, within *A* has the following effects:

- duplicates the memory allocated to *V*, *C*, *B* and *A*. Differential storage is used in some cases for optimization, but that still requires extra memory and processing and highly complicates the data structures. The intuitive optimization of updating in-place when there is only one copy is not known to be implemented in current compilers, because it can be safely performed only on corner cases which are non-trivial to produce automatically.
- distributing the updated objects to interested code locations requires explicitly coded logic aware of:
    - the code locations which used the precedent structure version
    - the in-order reconstruction of the *V*, *C*, *B*, and *A* structures, because the parent structures are using the original *V* value. That implies a cascade of reconstructions operations, which are simply not necessary in the case of references. More generally, all the nodes in the code graph which logically refer to that value, must be TRANSITIVELY reconstructed.
- beyond the implied expense in processing and memory, these update distribution operations and explicit state management break the code encapsulation.

3.2.1 Erlang

Erlang (Erlang, 2003) is arguably the canonical parallel language. It exhibits a parallelism level at least orders of magnitude higher than a multi-threaded C program. For example, an Erlang program can easily have tens of thousands to hundreds of thousands of parallel processes, while C has difficulty coping with thousand of threads. The copy-on-write semantics is at the foundation of its parallelism concept and implementation. It is also its greatest weakness, because of the highly inefficient data structures it forces upon Erlang programs. Copy-on-write is the most common implementation of Value-Types (VT) semantics, also used in other pure functional languages, such as Haskell. It is the very reason why functional programming languages are considered slow and are not used in performance critical software. For example, it is common for serial Erlang code to execute 20 times slower than the equivalent C++ code. We argue that while Value-Types semantics is critical, we can enforce it using compile-time analysis without resorting to Copy-on-write implementations.

Another weakness of Erlang is the lack of symmetry between internal function parallelism and process-level parallelism. This bad smell points to an insufficient generalization. We propose a hypergraph program structure which is auto-similar at all magnification levels.

3.2.2 Rust

Rust (Rust, 2013) is an experimental programming language developed by the Mozilla Foundation which is conceptually similar to our approach, more than any other language. Same as in Flux, the parallelism properties of the data structures are verified at compile-time, and have no runtime performance hit (Dobrescu-Balaur et al., 2015).

The Rust language is very complex, even more complex than C++, but there are two main issues relevant to parallelism: pointer ownership and local storage versus heap (boxed) storage. Both are present as attributes in the type system and have their own operators. We argue that these are the wrong concepts to use, as they derive from the Value-type/Reference-Type paradigm, and not the other way around.

The complicated type-system of Rust must be manually controlled and involves a lot of work with very ugly code. In Flux explicit typing is optional; similar type checks exist, but the attributes involved are automatically inferred. The result is code which is much more readable than Rust's.

In Rust parallelism is manually coded by the programmer as explicitly defined heavy-weight threads. In Flux parallelism is implicit in the logical description of the computations to be made; logical threads are very lightweight and are batched together on the working queue of a physical processor; the allocation of threads to physical resources is optional and is done using optimization hints.

The nearest comparison of what we are trying to achieve would be a combination of Rust, but with most of the dreadful type specifiers automatized and made invisible and Flow Based Programming (StreamIt).

3.3 Call Stack Considered Harmful

A programming language construct common to both programming paradigms is the call stack. This behind-the-scenes primitive construct has a pervasive influence on the structure of source code, as it forces synchronous message passing with strict Request-Reply semantics. That is, the return value of a function (Reply) is passed to the same location the parameters originated from (Request). The stack itself is designed to thread the call tree and cannot function with message passing structures resembling direct acyclic or cyclic graphs (eg coroutines). The limitations this rigid structure forces upon the minds of millions of programmers cannot be overestimated.

From a parallelization perspective, the synchronous nature of call stacks blocks pipelined execution. This essential parallel construct is unused in modern programming languages.

Finally the stack requires permanent management, at every call and every return the stack must be pushed or pop, even when the value on the stack will be required at a subsequent function call. With the generous amounts of memory and cache available today, the management of the call stack can be replaced with allocation of static buffers for input

parameters and return values, with the lifetime synchronous with that of the containing process.

3.3.1 Garbage Collection

Garbage collection (GC) is an expensive operation, whose cost is commonly underestimated: poor performance scalability of GCs on massive parallel systems; memory inefficiency of GC; complexity of GC logic and the non-locality of GC forces a high-load on the caching and VM subsystems.

The lifetime of all VT objects within a process can be decided statically, and thus reduce the number of instances which must be tracked by GC. Furthermore, using escape-analysis all the references which do not escape the scope of a process can be destroyed when the process terminates. Heap can be partitioned in separate heaps for each thread context.

3.3.2 PI-Calculus

Flux's control of naming aliasing is directly related to Pi-Calculus (Milner et al., 1992) and its results can and should be used to verify the correctness properties of Flux programs. However Pi-Calculus says nothing of the efficiency of the resulting computations, nor about the human effort required to describe code structures. Our primary concerns are to deliver a language which is easy for programmers to write with and efficient by default for idiomatic code, and then add as much correctness verifications as possible.

The flaws identified by our analysis of the paradigms of imperative and functional languages also apply to OpenCL and Cuda (which are slightly adapted implementations of C), PQL (an extension of Java), Parallel Haskel and others. In particular, OpenCL exhibits bad traits of compiler design, such as minimal type safety checks and manual coding of basic operations which should have been in the scope of the compiler such as scheduling of command queues and SIMD, synchronization and data movement.

3.4 Hardware Evolution Blocked By Lack of Parallel Languages

The instruction sets of modern CPUs are using a von Neumann model. In this model the CPU is equivalent to processing a single node at a time in the Program Dependency Graph (PDG). Modern pipelined superscalar processors, are attempting at run-time to find and execute neighboring nodes in the PDG. This approach is severly limited in scope, generic superscalar CPUs are unable to fill more than 4-8 ALUs.

There is a wealth of hardware architectures which exhibit high levels of parallelism, but are currently only used in specialized applications, because there is no generic programming language able to scale from the very fine granularity of logic gates to the very coarse granularity of server clusters:

- FPGAs
- Transport Triggered Architectures
- GPU programming
- ultra-wide superscalar processors
- computer clusters

A more worrying trend is the huge decrease in transistor utilization efficiency:

Table 2. Transistor utilization

CPU	Transistors	MHz	Instructions /Hz	MIPS	Transist./ Instr. / Clock
8086	29K	10	0.075	0.75	386K
i7-3930	2270M	3200	24	76800	94.5M

We can notice that the transistor budget has increased 245 times for a unit of work. This overhead can be lowered by simplified hardware architectures, which move more of the work to compilation.

4. FLUX - A DATA-FLOW PARALLEL PROGRAMMING LANGUAGE

4.1 The Data Flow Paradigm

The benefits of the dataflow paradigm have been know for the last 40 years, but a combination of factors delayed its rise. Only in the last 5 years the growth from the hardware side made the current approach non-economical. It is just too hard to use a GPU at its full potential using current compiler technologies.

As (Kosinski, 1973) and (Johnston et al., 2004) showed, a totally asynchronous model with value-type message semantics has remarkable properties, such simplicity, automatic, maximal parallelism with implicit-locking, excellent modularization properties, safety, force VTS on a case by case basis. All visual programming environments use a dataflow model - visual programming can open up computing to a larger audience.

However, the implementation of such a system presents significant problems, such as inefficiency of transporting large value type messages using copying, inefficiency of asynchronous message-passing using small-granularity threading, unavailability of hardware implementations with enough execution resources to run large programs.

Our proposal addresses each such problem, resulting in a programming language which unifies the current best solutions from the imperative and functional paradigms into a whole, with a focus on performance and safety.

4.2 Message Passing

In the deterministic Data Flow model, each message be a VT because a message sent to multiple destinations must be received verbatim by all the receivers.

That is, it must not be modified on-the-flight by instructions executing in another branch. This requirement insures that graph nodes can be connected in a defined manner, by using the names of their dependencies, even if the connection list is not itself ordered.

That does not mean messages objects should be immutable (= cannot receive messages on non-const ports), just that a VT message should have a single version existing at all times. If there are multiple branches and at least one of them modifies the message, explicit Copy (%) operations must be inserted to restore single object version per name property (and thus split aliasing entanglement).

As opposed to Erlang, a node (process) can have multiple named inbox queues.

4.3 Reference Passing with Value-Type Semantics

Naive copying of message value is prohibitively expensive, which is the main reason functional languages are so much slower than imperative languages. Java implements VT semantics for object references by disabling set() operators on common object types such a String. A purely functional language, Haskell implements VT semantics by forcing every update operation to return a new copy of the full object. Flux will use in-place update of VTs, while still enforcing VT semantics, by verifying at compile-time that when an object reference receives a write message, that is the only branch with read or write access to that that object reference. In other words, the name of a VT object would expire after the first message sent to it. Since no other node following in the program graph will be able to connect to it unless a new name is assigned total ordering will be preserved. It becomes the responsibility of the programmer to insert explicit copy operations (%) on necessary branches, instead of automatic copying the object on every write operation as all pure functional languages do.

This compile-time analysis neatly solves the problem of passing copies of large value objects, because they are always passed by reference, and VT or RT semantics is determined based on code context, on a case by case basis instead of object interfaces.

4.4 Locks Removal

In a Maximally Parallel Programming Language we start with the assumption that every statement and every object is executed in a parallel context. In this case RT objects may receive simultaneously messages from different threads of execution, for which the receiving queue must be locked. Locking is an expensive operation and it is unrealistic to lock a large majority of the objects in the application. Hence, every effort must be made to remove locks while preserving the meaning of the program.

When total ordering can be guaranteed for an object, locks can be removed because the scheduler guarantees ordering when the receiving object is a VT or the messages are delivered within the same execution thread.

4.5 Data Structures as Processes

In Flux every data structure is a process itself, which communicates with other processes by receiving and sending messages to other processes ports. Each process has its own internal memory, exclusively controlled, which can be accessed only through the defined input and output ports, that is, there is no shared state. The execution of a process is controlled by its execution context, i.e. processes are run sequentially within a thread mapped to a hardware resource.

4.6 Ports

For a process, ports are the only access points to and from the outside environment. Ports can be either for input or output, but not both at the same time. A compound port with multiple 'wires' must have all its components activated (available) before the port itself will become activated.

4.7 Synchronous versus Asynchronous Channels

Object ports are linked to each other using communication channels. For synchronous channels (i.e. Zero Capacity Channels) a trick that can eliminate the runtime cost of channel synchronization is compile-time scheduling using topological sorting. This way both synchronous and asynchronous channels have the same coding interface and orthogonality of logic and implementation is preserved.

4.8 Pipelining

Even so called "sequential code", that is, a chain of operations each dependent on the previous one can be parallelized in the case of a stream of input messages. While this technique is common in CPU microarchitectures, there is no general purpose language implementation which can do pipelining using static scheduling. The potential of pipelining parallelism is larger than the classic parallelism, as it is NOT limited by the classic Amdahl's law (but a complementary law, limited by the speed of the slowest step and the dynamic program behaviour).

4.9 Mapping of the Logical Data-Flow Structure to Execution Resources

The pure dataflow model assumes dedicated hardware for each operation node. As the hardware architecture is in most cases a given von Neumann machine, which can process only a single operation node at a time (per ALU, per thread, per core), the available operations must be mapped onto available execution resources. This process should be done in a post-

compile but pre-runtime step (like installation), so that the local resources can be fully exploited. A thread scope attribute can be placed around a set of nodes so that all the messaging and operations within the scope are matched to a thread.

4.10 HyperGraph Syntax

A dataflow program is most naturally expressed as a hypergraph, that is, a directed attributed graph in which each node and each edge can be a graph itself. A generic graph notation can thus be used, greatly simplifying the syntactic structure through conceptual unification and elimination of exceptions and special cases. Like LISP, Flux is homoiconic, although this was not a design goal but a result of the generic syntax structure.

It is the authors' opinion that, while general, lists are too low level to preserve the initial structure of the data (similarly to how strings are capable of containing any kind of data, yet their meaning cannot be extracted unless sophisticated parsers are used). By using a more general data structure, the information contained in the data can be extracted with less effort.

A hypergraph has the amazing property that all common compound data structures, such as pairs, records, arrays, lists, maps, sets, etc can be found as parts of some hypergraph. So, the approach is inverse to LISP where all complex structures are constructed out of lists (the strings of symbolic programming), rather the other way around, all the basic structures are incomplete views of a single universal structure.

The language syntax is designed to:

- allow description of complex directed attribute graphs to be expressed in the language, with a minimal number of coding syntactic-sugar.
- be minimalistic, i.e. everything which can be defined as library functions (IF, FOR, WHILE, SWITCH, etc) should not be defined in the compiler BNF
- be concise, input and output ports binding must not repeat the node; anonymous ports and nodes must be possible; use operator associativity instead of markup
- preserve intent;

As opposed to Erlang, where processes are relatively heavyweight and are stacked on-top functions and lists, in Flux the process abstraction is present at the very lowest levels of language design, each object and each function is a process. The proof that this highly abstracted approach can work for VTs is given by the existence of VHDL, a language used in hardware design, as such placed at the lowest levels of abstraction, where processes are the primary organization unit and only VTs exist. The mapping of processes to threads is orthogonal on the code logical structure, unlike Erlang where calling a function is syntactically different from sending a message to a process.

5. SYNTAX DEFINITION

5.1 Context Binding

It is impractical for a human programmer to fully describe the code graph, which is enormously complicated even for relative small instances. That is why the compiler must automate as much connecting as possible. For example, all the input ports of nodes within a block must be automatically connected to the output ports of its ancestors of the same name. This automated binding of "variables" in anonymous code blocks, similar to the mechanism in D, can be used to define custom control structures.

5.2 Automatic Resource Collection

Garbage collection is an expensive operation, whose cost is commonly underestimated. The lifetime of all VT objects within a process can be decided statically, and thus reduce the number of instances which must be tracked by GC. Furthermore, using escape-analysis all the references which do not escape the scope of a process can be destroyed when the process terminates. Heap can be partitioned in separate heaps for each thread context.

5.3 Minimal Language

While more complex than LISP, the Flux programming language uses a small number of concepts, even lower than JavaScript (Table 3, Table 4). For example:

- functions, methods, classes and inner classes are all replaced by "process nodes"
- control statements (IF, FOR, WHILE) are defined in the language library from even lower-level constructs (automatic binding, boolean choice and flow merge)
- no explicit thread locks
- no explicit constructor (replaced by copy operator)

Table 3. Flux Keywords

Logic and Arithmetic Operators	Syntax
Greater than	`gt`
Less	`lt`
Modulo	`mod`
Not	`not`
And	`and`
Or	`or`
Xor (bitwise and logical)	`xor`
division	`Div`
multiply	`Mul`

Table 4. Flux Operators & Syntax

Semantic	Syntax	Description
Graph Operators		
Triplet syntax	`SrcNode Relation DestNode`	Default syntax

Directed relation	`>Relation or Relation>`
	`(source)Relation Target`
	
	`<Relation or Relation<`
	`Target Relation(source)`
		Directed relations; all relations (arrows) are marked with special markers

Undirected Relation	`=Relation`	(outPort1 : inPort1 ... outPortN : inPortN otherAttr1, otherAttr2, ...)

Common Source Triplet	`source (>relation target;`
	`>relation2 target)`
	`rt( source ) { >r1 t1; >r2 target2 }`	

Common Relation	`(sources) >relation (targets)`
	
	`st(relation){ s1 > t1; s2 > t2}`	

Common Target	`(source >relation; source >relation) target` `sr(target){ s1 >r1 }`	
List or tuple	`()`	Sequenced list of items or list of attributes specifier
List delimiter	`Space`	Lowest priority delimiter
2nd order list delimiter	`Comma`	Medium priority delimiter
3nd order list delimiter	`;`	Highest priority delimiter; acts as end-of-list marker for lower priority lists
Naming operator	`name <= value // rtl` `name => value // ltr`	
Code Operators		
Code markup	`{ code }`	Node containing a non-ordered (parallel) list of processes. In/out ports do not need to be declared. Eager evaluation proceeds
Type instancing operator	`Type.var // variable cell` `Type.val // const cell`	

Array index	`Array(Type) //type`
	`array.(index)`
	`(index):array`
		Parenthesis is necessary to not discriminate port "i" by array index, and force eval of index

Maps (aka Arrows, Functions)		

Map Declare	`// map of one rule`
	`x -> +(x 1)`
	`// general map`
	`{`
	
	`Symbol1 > Result1;`
	`x -> x + 1; // bounded rule`
	`}`	

Identity function (assignment)	`>`	Copy constructor
Map Eval, arrow notation	`param >f output // ltr` `output f< param // rtl`	
Map Eval, classic	`f( g(x) ) //like ((x) >g ) >f`	
Member/inline ltr	`x:g:f`	Shortcut for f(g(x))
Special Form Relation	`#specialFormName(parameters)` `{content}`	

Basic Control		

If	condition >if{
	then > {stmt1}
	else > {stmt2}
	}
		IF is a node which has two output ports, which are connected to then code block and else code block

While	while(condition {statements});	
For	for(i collection {statements})	
Repeat	({statements} condition)repeat	

Switch	caseVar > {
	case1 > result1;
	case2 > result2;
	}	

Object System		

Class	StringBuilder class
	{
	// fields
	x > var(Int)
	// methods
	append > {}
	remove > {}
	toString > {}
	}
		A class is a map from property name to method node
		var() works like a functional "new", ie is idempotent

Default constructor	Class.var({param1 > value1; param2 > value2 })	Applies a name-to-position map before the function

6. APPLICATION TO NEURAL-NETWORKS

The increasing technological demands of our days have required new solutions to the highly demanding control problems. Neural networks by their massive parallelism and learning capabilities have proved to be an efficient approach to a wide range of applications that involve accuracy of classification (Balti et al., 2013) or control architectures, such as the model reference adaptive control, the model predictive control (Hagan et al., 2002), or the learning controller (Dragoicea et al., 2001).

The field of neural networks covers a broad area. We will concentrate on the most popular network architecture, the multilayer feed-forward neural networks, currently used in control systems. Our focus is the implementation in Flux of the backpropagation algorithm - the principal procedure for training multilayer feed-forward neural networks.

The multilayer feed-forward neural network is built up of simple components. Starting with a single-input neuron, we can extend to multiple inputs, then stack the neurons together to produce layers, and finally cascade the layers to form the network. The strengths of the connections are denoted by parameters called weights, that might be adjusted to improve performance. Each output unit takes, as input, the weighted sum of the outputs from the units in the previous layer, and applies a nonlinear function to the weighted input. Given enough layers, the multilayer networks can closely approximate any function, which recommends them as valuable function approximators for different control architectures.

The backpropagation algorithm (Han et al., 2012) learns by iteratively processing a dataset of training tuples, comparing the network's predictions for each tuple with the actual known target value. For each training tuple, the weights are modified so as to minimize the mean squared error between the network prediction and the actual target value. Although it is not guaranteed, in general the weights will eventually converge, and the learning process stops. The Flux implementation of the algorithm follows (Source 1):

Source 1.

```
// neuron implementation
Neuron class<
{

  inWeights > Array(Real).var
  outWeights > Array(Real).var
  bias > Real.var
  output > Real.var
  error > Real.var

  init >
  {
    inputs ->
    (
      inputs.0 > output; //output of an unit in the
input layer is its actual input value
    })
  }

  // compute the output for an unit in the hidden
and output layers
```

```
  signal >
  {
    inputs >
    (
      sum( i inputs.indexes
      {
        i > mul( inWeights.(i) inputs.(i) )
      }) bias >+ >sigmoid > output
    )
  }

  // compute the error for an unit in the output
layer
  initialError >
  {
    target ->
    (
      error < mul( output -(1 output) -(target
output) )
    )
  }

// compute the error for an unit in the hidden
layer
  hiddenError >
  {
    previousError outWeights ->
    (
      error < mul( output -(1 output) sum( k
outWeights.indexes mul( previousError.(k)
outWeights.(k) ) )
    )
  }
  sigmoid > { x > div( 1 +(1 exp(-(x)) ) ) }
}

// layer implementation
NeuronLayer class<
{
  neurons > Array(Neuron).var()

  mapInputs >
  {
    (what inputs) ->
      for( neuron neurons
        neuron.(what) < inputs )
  }

  mapErrors >
  {
    (what errors) ->
       for( neuron neurons
         neuron.(what) < errors )
  }

}

// network, a multilayer feed-forward network
Network class<
{
  layers > Array( NeuronLayer ).var
  learnRate > 0.2

  trainTuple >
  {
    (initials targets) ->
      (
        // compute the net output of current layer
with respect to the previous layer
        initials > previousLayer

        for( layer layers
        {
          layer.mapInputs( signal previousLayer )
        })

        // backpropagate the errors
        previousLayer < layers.last

        for( layer layers.reverse
        {
          deltaWeight > Array(Real).var

          for( neuronIdx layer.neurons.indexes
            neuron < layer.neurons.(neuronIdx)
            eq( layer layers.last ) >if
            {
              // only for last layer
              then > (neuron.initialError <
targets.(neuronIdx) );
              else >
              (
                // compute the error with respect
to the higher layer
                neuron.hiddenError <
previousLayer.errors.(neuronIdx)

                // recalculate weights
                for( outNeuronIdx
previousLayers.neurons.indexes
                {

                  // weight increment
                  deltaWeight.(outNeuronIdx) < mul(
learnRate neuron.output outNeuron.error )

                  // weight update
                  neuron.outWeight( outNeuronIdx )
=> ow + deltaWeight > ow;
                  outNeuron.inWeight( neuronIdx )
=> iw + deltaWeight > iw ;
                })
              );
            }

            neuron.bias => b + mul( learnRate
neuron.error ) > b //bias increment + update
          )

          previousLayer < layer
        })

      )
  }
}

//Input: TrainingSet, a data set consisting of the
training tuples and their associated values
// learnRate, the learning rate
// Initialize all weights and biases in network;

trainAll >

  for( epoch range(1 maxEpochs)
  {
    for( trainingTuple TrainingSet
    {
      trainTuple< trainingTuple.initials
trainingTuple.targets
    }
  })
} // end Network class
```

The program is structured in three classes describing the structure and functionalities of neurons, layers and network. The implementation reflects quite accurately the algorithm presented in (Han et al., 2012). The idea of the algorithm is to repeatedly process training tuples until some terminating condition is satisfied. Each training tuple is processed in two steps. Firstly, the inputs are propagated forward: the input layer passes the inputs unchanged; the output values for the

hidden and output layers are computed, which gives the network's prediction. Secondly, the errors are propagated backward by updating the weights and biases to reflect the error of the network's prediction. We considered that the whole process is repeated until a specified number of epochs is reached. The comments in the Flux program detail each step.

The Flux implementation of the algorithm benefits from the parallel capabilities of the language. Moreover, Flux has a natural formal representation for all kinds of Petri Nets. For example the network in (Vasiliu et al., 2009) figure 1, can be represented as simple as:

```
closedLoopPetriNet >
(
 p1 >t1 p2; p2 p7 >t2 p3 >t3 (p1 p8); p4 >
 t4 p5; p5 p7 >t5 p6 >t6 (p4 p8); p8 >
 t7 p7;
)
```

7. CONCLUSIONS

It looks possible to create a programming language faster than idiomatic C, the current champion. This paper proposes an approach intended to provide a new workhorse for the parallel age. Research on programming languages disregarded the efficiency of idiomatic programs in favour of other attributes important to the academic world, such as conceptual elegance, formal analysis-ability and ideological. Flux is designed for practical engineering reasons, like run-time and programmer efficiency, for example even the whole infrastructure supporting the VT/RT dichotomy appears as a result of the need to tackle parallel computation.

It is worth mentioning some contributions beyond state-of-the-art of the approach: (a) implement Value-Type Semantics by using compile-time verification of references instead of Copy-on-write; (b) achieve automatic parallelization of computations using Value-Type Semantics; (c) the first general-purpose programming language enabling static scheduling of Pipeline Parallelism; even so called "sequential code", that is, a chain of operations each dependent on the previous one can be parallelized in the case of a stream of input messages; while this technique is common in CPU microarchitectures, there is no general purpose language implementation which can do pipelining using static scheduling; the potential of pipelining parallelism is larger than the classic parallelism, as it is NOT limited by the classic Amdahl's law (but a complementary law, limited by the latency of processing steps and the dynamic program behaviour); (d) remove the need for explicit threading locks: the compiler will insert them for Reference-Types, where messages go over thread scopes; (e) preserve the appearance of asynchronous parallel programming for very small-granularity processes like variables and primitive data structures by using Zero Capacity Channels (static scheduling), which preserves the execution efficiency; (f) introduce a unified syntax based on hyper-graphs (similarly to how LISP is based on lists); (g) replace the Call Stack with an appropriate parallel data structure; (h) replace the global heap with a heap separate to each process groups and minimization of the needs for Garbage Collection; (i) massive simplification of program structure by replacement of variables, methods, members, objects and classes with generic nodes; (j) achieve similarity of program structure at block, function, object and module level; (k) implement the first compiler using high-level inference rules instead of hard-coded Abstract Syntax Tree traversals.

Data-Flow technology research has stalled today. The last significant book about Data-Flow (Sharp, 1992) has been released in 1992. We argue that a software focused research able to work on current CPUs and GPUs can avoid the pitfalls associated with low-volume esoteric hardware.

REFERENCES

Balti, A., Sayadi, M., Fnaiech, F. (2013). Fingerprint Verification Based on Back Propagation Neural Network. *Journal of Control Engineering and Applied Informatics*, Vol.15, No.3, pp. 53-60.

Dobrescu-Balaur, M., and Negreanu, L. (2015). Enhancing RUSTDOC to allow search by types. *Studies in Informatics and Control*, Vol. 24, Issue 2, pp. 221-228.

Dragoicea, M., Dumitrache, I., and Constantin N. (2001). On Some Aspects Of Neural Networks Control For Autonomous Mobile Robots. *Journal of Control Engineering and Applied Informatics*, Vol 3, No 3.

Erlang (2003). http://www.erlang.org/doc/reference_manual/users_guide.html

Hagan, M.T., Demuth, H.B., and De Jesus, O. (2002). An Introduction to the Use of Neural Networks in Control Systems. *International Journal of Robust and Nonlinear Control* 12(11), pp. 959 - 985. DOI: 10.1002/rnc.727

Han, J., Kamber, M., and Pei, J. (2012). *Data Mining Concepts and Techniques*, chapter 9, pp. 398-408. Morgan Kaufmann Publishers, San Francisco.

Johnston, W.M., Paul Hanna, J.R., and Millar, R.J. (2004). Advances in dataflow programming languages. *ACM Comput. Surv.* 36, 1 1–34. DOI: http://dx.doi.org/10.1145/1013208.1013209

Kosinski, P.R. (1973). A data flow language for operating systems programming. *Proceeding of ACM SIGPLAN - SIGOPS interface meeting on Programming languages - operating systems*. ACM, New York, NY, USA, 89–94. DOI: http://dx.doi.org/10.1145/800021.808289

Milner, R., Parrow, J., and Walker, D. (1992). A Calculus of Mobile Processes, *Information and Computation* 100(1) pp. 1-40.

Rust (2013). http://www.rust-lang.org/

Sharp, J. (1992). *Data Flow Computing: Theory and Practice*, Ablex Publishing Corporation, ISBN-10: 0893916544.

Vasiliu, A.I., Dideban, A., and Alla, H. (2009). Control Synthesis for Manufacturing Systems. *Journal of Control Engineering and Applied Informatics*, Vol.11, No.2, pp. 43-50.