GoDEL: A multidirectional dataflow execution model for large-scale computing

Abhishek Kulkarni
Indiana University
Bloomington, IN 47405
adkulkar@cs.indiana.edu

Michael Lang
Los Alamos National Laboratory
Los Alamos, NM 87544
mlang@lanl.gov

Andrew Lumsdaine
Indiana University
Bloomington, IN 47405
lums@cs.indiana.edu

***Abstract*—As the emerging trends in hardware architecture guided by performance, power efficiency and complexity drive us towards massive processor parallelism, there has been a renewed interest in dataflow models for large-scale computing. Dataflow programming models, being declarative in nature, lead to improved programmability at scale by implicitly managing the computation and communication for the application.**

In this paper, we present GoDEL, a multidirectional dataflow execution model based on propagation networks. Propagator networks allow general-purpose parallel computation on partial data. Implemented with efficiency and programmer productivity as its goals, we describe the syntax and semantics of the GoDEL language and discuss its implementation and runtime. We further discuss representative examples from various programming paradigms that are encompassed by and benefit from the flexibility in the multidirectional execution model.

***Keywords*-Parallel programming; Computer languages; Runtime library; Concurrent computing**

I. Introduction

The high performance computing landscape has been a fast moving one with a thousand-fold increase in the peak computing power seen roughly every ten years. With the continuing trend, one hopes to see an exascale (10^{18} FLOPS) machine within the decade. However, recent studies [1], [2] indicate that deployment of machines with such computing capabilities could be expected as early as 2018. Power efficiency, concurrency, resiliency and programmability are among the major challenges encountered at such scales. To address these cross-cutting concerns, emerging architectural trends suggest massive intra-node parallelism possibly aided by accelerators to achieve a sustained exaflops performance. Exemplary architectures [3] are needed to overcome the limits of current supercomputers. To avoid hitting the power-wall, experts predict future many-core processors to have hundreds to thousands of small cores [4]. This would necessitate revisiting some of the design decisions across all layers of the HPC software stack to effectively leverage the abundance in available compute resources.

While alarming to some, the situation brings a sense a déjà vu to others. Dataflow architecture research [5]–[8] in the 1970s and 1980s was born out of coping with massive processor parallelism. With roots in asynchronous digital logic, structured and functional parallel programming and program schemata, dataflow eliminated some of the bottlenecks in Von Neumann architecture by providing an architecture without a global program counter or a global updatable memory store. Early supercomputers, including the Connection Machine, employed dataflow execution models [9], [10] to leverage the parallelism in the architecture. Dataflow architectures like the MIT tagged-token architecture [8] had a special instruction set suited for dataflow where instructions executed based on the availability of the operands. Dataflow execution models are inherently parallel in nature and provide communication-computation overlap by implicitly scheduling the communication. They provide a higher-level abstraction in the form of dataflow graphs which implicitly capture the parallelism in the application. There has been a recent resurgence in data-driven models to improve the programmability for large-scale applications. Recent focus on dataflow graph-based frameworks has been on providing deterministic parallelism by implicitly capturing the control and data dependence in the program. Partitioned Global Address Space (PGAS) languages provide higher productivity through constructs for variable memory addressing schemes and irregular, dynamic task parallelism. Coarse-grained dataflow models also form the basis of several scientific workflow management systems.

We posit that a multidirectional dataflow execution model based on the data propagation model of computation [11] can efficiently exploit extreme-scale computing architectures. It is inherently parallel in nature and liberates computation from the notion of time. Avoiding timing issues and synchronization concerns, it is expressive enough to encompass multiple programming paradigms like dataflow programming, constraint propagation, incremental and reversible computation. GoDEL (Go Dataflow Execution Language) is an embedded domain-specific dataflow language based on the model of propagating and accumulating information using always-on networked computational entities. It provides an adaptive, managed runtime for parallel multidirectional execution of programs. The remainder of this paper is organized as follows. Section II describes the propagator networks computation model. The overview of the GoDEL language, its syntax and semantics is explained in section III. The implementation details for the language and its runtime are described in section III-D. In section IV,

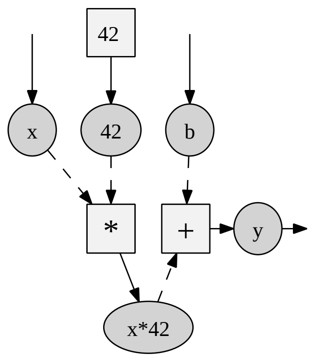


Figure 1. A unidirectional propagator network for the linear equation: y = 42x + b

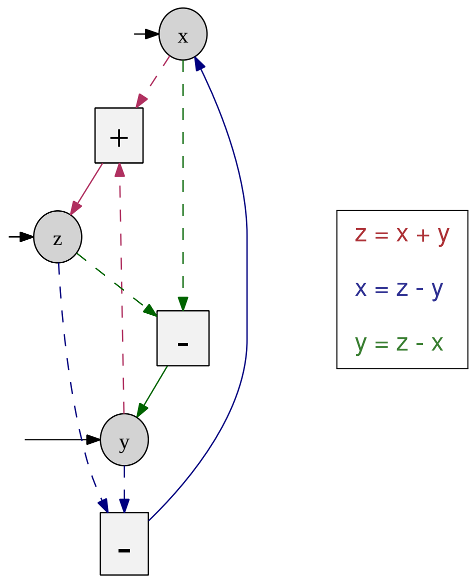


Figure 2. A reversible network for the sum (and differences) of three integers: x, y and z

we discuss an illustrative example that leverages this model of multidirectional execution. Related research is highlighted in section V, and finally section VII concludes discussing future work in section VI.

II. Propagation Networks

Propagation networks are a continuous-time, general-purpose model for concurrent and distributed computation developed by Radul and Sussman [11], [12]. The model liberates program execution from the temporal tyranny providing multidirectional dataflow between autonomous networked machines. These networks consist of "cells" which store *information* about the data and "propagators" which are stateless computational entities that read and write *information* from and to the cells. The set of axioms imposed on the cells and propagators make it a simple yet powerful model suitable for general computation. In particular, cells do not store values but instead accumulate values over time. Once a value has been written to a cell, it cannot be deleted or modified. Cells are a form of memory that remember values over time. Although this looks limiting for general-purpose computation, cells can add *information* about values. For instance, cells can merge values using a user-defined *merge* operation. No propagator is, thus, delegated to exclusively control the data or value in a cell. Propagators can merge a "partial" value to a cell and hence incrementally refine the *information* that a cell knows. This adaptive behavior associated with cells give the propagation network its emergent properties and allow it to encompass multiple programming paradigms. The restrictions impose a regular structure on the cell and its corresponding *merge* operation. The *merge* operation is enforced to be idempotent, commutative, associative and monotonic with respect to the cell.

Propagators are asynchronous, autonomic units waiting to perform computation. Propagators process data from multiple input cells and write the results to an output cell. Propagators can register interest in cells they wish to read data from. On arrival of new information, cells notify those propagators which are interested in receiving data from them. Propagators are also monotonic since they can operate on a partial input to produce a partial output. They allow checking for emptiness of a cell and multiple reads of the value in a cell. Propagators could be as coarse-grained as being a heavyweight process or an ensemble of processes; or as fine-grained and lightweight as a function or a single instruction.

Propagators in the dataflow network are *fired* on the arrival of new *information* on its *firing set* (input cells). The execution is thus asynchronous, but strictly data-driven. Propagators do not execute when its output cell is "read". It resembles a push-based dataflow instead of a pull-based dataflow model. Propagators can be executed in parallel as long as the data they depend on is available. This model, thus, lends itself naturally to concurrent execution. Fig. 1 shows a propagator network for the linear equation $y = 42x + b$. The whole expression is represented as a directed acyclic graph (DAG) with the boxes representing propagators, and ovals representing cells. The dashed lines indicate input arcs to a propagator, whereas the output arcs are represented by a solid line. The constants in the expressions are emitted by a *constant propagator* which writes 42 to its output cell. When an input value is written to the cell x, it triggers the execution of the network until quiescence, after which the value from the output cell y can be safely read.

In [11], Radul shows how, in addition to primitive datatypes like numbers, compound datatypes like intervals and pairs can be propagated. When propagating intervals, the *merge* function intersects the intervals from multiple propagators, thus providing a powerful dataflow network for interval arithmetic. The distinct separation between the basic computational units (propagators), the memory (cells) and its corresponding generic *merge* functions allows the core propagation network substrate to deal cleanly with different datatypes flowing through the network.

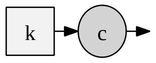
Since an input cell could be connected to multiple propagators, many interesting networks with multidirectional dataflow can be constructed in this model. Maintaining invariants on the expressions, such networks allow *partial* concurrent evaluation of expressions in any order and direction. A reversible network for a simple add operation between integers is shown in Fig. 2. Depending on how the data dependencies are satisfied, this network can be executed in any order. When the values of x and y are known, the value of z is computed using the equation $x + y = z$. However, when z and y are known, x is computed using $x = z - y$. This way of declarative programming is reminiscent of logic programming, but propagation networks are powerful enough to subsume multiple programming paradigms including imperative programming, logic programming, constraint programming and reversible computation among others.

As mentioned earlier, propagation networks define a continuous-time dataflow model where the outputs are observed only after the network reaches a quiescent state, that is a fixed-point of the dataflow graph function is obtained. Further, since the cell values cannot be overwritten, the network avoids divergence in common cases like the example in Fig. 2 when conflicting constraints are specified as the input.

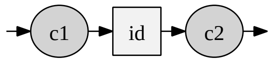
III. GoDEL

GoDEL is a concurrent, scalable implementation of propagation networks designed with efficiency and expressivity as its primary goals. Propagators are lightweight, schedulable coroutines whereas cells are modeled by a pair comprising of a shared memory location and an accompanying coroutine monitoring this location. GoDEL is implemented as an embedded domain-specific language (EDSL) embedded within a more general-purpose, feature-rich host language. This increases the productivity by allowing all host language constructs and library functions to be used within GoDEL's propagator functions. GoDEL can be used as a coarse-grained dataflow system where the abstractions provided by GoDEL are used for intrinsically capturing the data and control dependence and offering implicit synchronization for the parallel execution of the host language program.

GoDEL treats propagator computations as opaque functions of the host language. It offers primitive multidirectional propagators (arithmetic operators and their inverses like $+$ and $-$, $*$ and $\div$, ^{x} and $\sqrt[x]{}$). If a propagator function is ill-defined or non-existent (for instance, it could be a partial function), the dataflow in that direction is prohibited. With its primitive propagators, GoDEL constructs reversible



(a) A constant propagator $c \leftarrow k$



(b) An identity propagator $c_1 \leftarrow c_2$

Figure 4. Trivial propagator expressions

multidirectional networks by default, but functions defined from the host language can be used to construct networks in which data flows in a particular direction. Following the design principles of propagation networks, GoDEL prohibits reassigning or modifying values of "cell" variables. It, however, offers a convenient construct to reuse a network instead of discarding it after a single use. We found that this is very handy, in practice, to execute the GoDEL graph with different input arguments or use it in a stream programming context.

Similarly, cells in GoDEL are polymorphically typed – they support any type from the host language to be used for the cell's content as long as an appropriate *merge* function with the correct function type is provided. Cells are internally implemented as objects which allows any function from the host language to override the default merge operation associated with a cell. Since we initially designed GoDEL to primarily target scientific applications, it provides two more primitive types: `interval` for imprecise interval computations, and `matrix` for matrix computations.

A. Syntax

A GoDEL program is defined as a set of dataflow equations that form a directed graph of propagators mutually connected through cells. An *id* represents an identifier that can uniquely reference a cell. Since a GoDEL program is embedded in a host language H_L, it can contain any host language expressions H_E or a series of cell declarations *cell*, propagator expressions *prop* and finally, cell assignments *assign* to trigger the execution of the network. A cell declaration consists of a unique identifier not defined a priori as a variable in the preceeding host language expressions H_E. The merge method supporting primitive GoDEL types exported by a cell can be overridden as shown. Propagator expressions define the dataflow equations modeled by the network. They involve either primitive propagator operators (like $+, -, *, \div$) or any function H_f from the host language. Constants k in the propagator expressions are implicitly hoisted to propagator functions emitting these constants as shown in Fig. 4(a). Aliases to cells can be defined as in Fig. 4(b) which implicitly constructs an *identity* propagator.

Once a value is assigned to a cell variable using the *assign* expression, it flows through the directed graph, triggering propagators which depend on that value. A convenience construct := is provided that resets a propagation network and assigns a new value to the given cell location. This is

P_g	$::=$	$H_E \mid cells\ \overline{prop}\ assign$	*GoDEL program*
cells	$::=$	var $\overline{c}$ Cell	*Cell declarations*
	$\mid$	c.merge $= H_f$	*Merge functions*
prop	$::=$	$c \leftarrow H_f(\ \overline{v}\ )$	*Propagator expressions*
	$\mid$	$c \leftarrow v$	
	$\mid$	$c \leftarrow v + v \mid c \leftarrow v - v$	
	$\mid$	$c \leftarrow v \times v \mid c \leftarrow v/v$	
	$\mid$	$c \leftarrow v^n$	
assign	$::=$	$c = k$	*Cell assignment*
	$\mid$	$c := k$	*Reset network and assign*
v	$::=$	$c \mid k$	*Values*
k	$::=$	$H_n \mid i \mid m$	*Constants*
c	$::=$	id	*Cell identifier*
i	$::=$	Interval$[n : m]$	*Intervals*
m	$::=$	Matrix$[x, y]$	*Matrices*
H_E			*Host language expressions*
H_f			*Host language functions*
H_n			*Host language constants*

Figure 3. GoDEL syntax

handy when reusing a previously constructed propagation network to execute it with different input values. Although this is powerful enough to implement iteration and recursion in the network, its use is discouraged. Iteration in the network can either be implemented by using a *list* datatype and *cons*ing values to the list or using a multi-valued *switch* propagator. GoDEL treats intervals and matrices as first-class values and supports conveniently defining them statically by specifying the lower and upper limits for `interval` and the dimensions for `matrix` respectively.

B. Semantics

A GoDEL program is essentially a dataflow graph consisting of *cells* and *propagators* which we treat as opaque deterministic functions acting on the values in the cells. The propagators themselves can be non-deterministic as long as they do not communicate externally using mechanisms other than the cells. Since the first-class values in the syntax are converted to constant or identity propagators depending on the context, the propagator expressions reduce down to the following core form:

$$e \quad ::= \quad c \leftarrow f(\overline{c})$$

The dataflow graph defined by a GoDEL program can be represented by a sequence of such expressions. The $\leftarrow$ operator abstracts the communication between a propagator and cell. Similar to [13], we define semantics for GoDEL by making the state and non-determinism in the calculus as core concepts. The notation used in the semantics is as follows. c denotes a cell, a location that holds a value. $\overline{c}$ represents a finite sequence $c_1, c_2, c_3, \ldots, c_n$ of cells. $\langle x, y \rangle$ stands for a tuple of elements x and y. $\emptyset$ denotes an empty environment, whereas $[c \mapsto v]E$ denotes an environment that maps a cell c to its value v.

At each computation step, the state of a GoDEL program can be modeled by the values of all its cells $\overline{c}$ represented by the environment C. The store C maps the cell names with their values in domain $\mathcal{D}$. $C(c_i)$ returns the value v indexed by the cell c_i. Symbol ρ represents the propagator environment which associates propagator names with their corresponding implementations. For instance, $\rho(+)$ can be used to look up the *sum* propagator from the environment. The transition $\rho \vdash \langle C, C' \rangle \rightarrow C''$ represents a computation step caused by the firing of a propagator.

Rule E-INITPROP in Fig. I represents the initial transition $\rho \vdash \langle \emptyset, C \rangle \rightarrow C'$. A propagator p with no input cells is selected and fired to get a value v. Using rule E-UPDATEC, the cell environment C is updated with the value v to obtain a new environment C'. After the initial propagator firings, the network is in a quiescent state as long as no values are explicitly written to any cell c. Rule E-SELECTC non-deterministically selects a cell c_i such that the following three conditions are met: 1) c_i occurs as an input cell to a propagator. 2) c_i maps to a value v in the environment C', denoted by $[c_i \mapsto v]C'$ and 3) Given a preceeding environment C, the value of c_i in the existing environment C' is not equal to the value of c_i in the old environment C.

Once the rule E-SELECTC selects a suitable cell which has a new fresh value from the preceeding transition, rule E-FIREPROP fires the propagator that depends on this input cell c_i, merges the value with the existing value of the cell c_i and stores the new merged value into the new cell environment C'. A suitable condition for network quiescence in this model is when all the cell environments are equivalent

Table I
GoDEL SEMANTICS

$$\frac{\begin{array}{c} v = selectC(C, C', c_i) \\ p = c_o \leftarrow f(\overline{c}) \\ v' = \rho(p)(v, C'(\overline{c})) \\ v'' = merge(v, v') \\ C'' = updateC(p, C', v'') \end{array}}{\rho \vdash \langle C, C' \rangle \rightarrow \; C''} \qquad \text{E-FIREPROP}$$

$$\frac{p = c_o \leftarrow f(\overline{c})}{updateC(p, C', v'') = [c_o \mapsto v'']C'} \qquad \text{E-UPDATEC}$$

$$\frac{\begin{array}{c} \exists c_i \in \overline{c} : c_o \leftarrow f(\overline{c}) \\ \exists c_i \in \; C : [c_i \mapsto v]C(c_i) \; \neq [c_i \mapsto v']C'(c_i) \end{array}}{selectC(C, C', c_i) = v} \qquad \text{E-SELECTC}$$

$$\frac{\begin{array}{c} p = c_o \leftarrow f() \\ v = \rho(p)() \\ C' = updateC(p, C, v) \end{array}}{\rho \vdash \langle \emptyset, C \rangle \rightarrow \; C'} \qquad \text{E-INITPROP}$$

to each other.

C. GoDEL Example: Temperature Converter

An example of a reversible GoDEL program is shown in Fig. 5. This program implements a dataflow equation $c = \frac{5}{9} \times (f - 32)$ to convert temperatures between Celsius and Fahrenheit and vice versa. When a value is written to c (Line 8), the corresponding temperature value in f is computed. On the other hand, if a value in f is known (Line 10), the resultant value in c is written out.

D. Implementation

The GoDEL runtime is implemented as a library in the Google Go [14] programming language. Go is a statically typed, garbage-collected, compiled language that offers performance comparable to low-level languages like C and C++. It has its roots in Hoare's Communicating Sequential Processes (CSP) model and is a successor to the Limbo programming language [15]. Go offers lightweight coroutines called *goroutines* and shared memory communication between them using *channels*. The Go runtime scheduler can schedule *goroutines* to run on a single kernel thread, or multiplex them to run on multiple kernel threads running on separate cores in a multicore system. The scheduler follows a simple round-robin scheduling strategy which turns out to be inefficient for unbalanced parallelism exhibited by irregular applications. *goroutines* have segmented stacks and are almost as lightweight as functions. They serve as appropriate lightweight user-level threads for fine-grained propagators, or as more dedicated kernel threads for coarse-grained propagators.

Embedding GoDEL in a more feature-rich, general-purpose host language like Go allows more expressivity in the propagators that could be defined. GoDEL is implemented as an embedded DSL following the multi-language paradigm. Although this results in impure semantics and consequently much-less guarantees about the generate program, it meets our goals of efficiency and expressivity. Since GoDEL is a valid subset of the Go language [14], the GoDEL compiler is implemented as a source-to-source compiler. It parses GoDEL programs into Go's Abstract Syntax Tree (AST), applies appropriate transformations on them and finally generates Go source. The Go compiler is invoked as a second-stage compiler to generate a platform-specific, architecture-dependent executable.

Similar to propagators, cells are also implemented as *goroutines*. On arrival of new information, a cell has to alert all its neighboring propagators interested in that data. In absence of asynchronous userspace memory notifications, it is requisite to monitor the shared memory location by polling for changes. Instead we treat the cells as processes that send and receive information to the propagators. Cells and propagators communicate with each other using channels. In addition to the primitive datatypes supported, cells are allowed to store any valid datatype that can be defined in Go as long as the corresponding *merge* function to handle that datatype is defined. Lacking generics [14], the generic layer in GoDEL is implemented through ad-hoc polymorphism. Go supports runtime reflection which is leveraged by GoDEL to call the appropriate methods using a method dispatch table.

```
 1  func main() {
 2   var f_32, c_9, f, c Cell

 4   f   <- 32.0 + f_32
 5   c_9 <- 5.0 * f_32
 6   c_9 <- c * 9.0

 8   c   = 25.0
 9   fmt.Printf("Celsius %v C = %v F\n", c, f)
10   f   := 77.0
11   fmt.Printf("Celsius %v C = %v F\n", c, f)
12  }
```

Figure 5. Temperature converter in GoDEL

IV. Representative Examples

In this section, we review some of the interesting ways in which propagation networks can be used for execution of large-scale applications. We discuss idiomatic examples in GoDEL to implement incremental computation, interval computations and standard dataflow computations. Instead of benchmarking against a real scientific application, we choose a representative method of finding the root of a real-valued function used in a large number of numerical kernels. In the specific case listed below, the multiparadigm

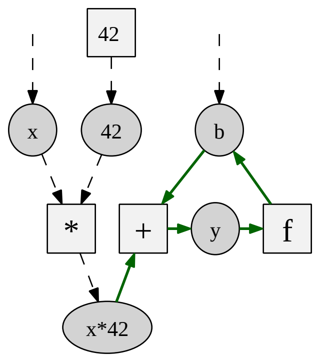


Figure 6. A unidirectional propagator network for the linear equation: y = 42x + f(y)

execution facilities provided by GoDEL result in terser and simpler code, relieving application programmers of common mundane tasks.

A. Incremental Computation

The GoDEL data-driven execution model forces evaluation of expressions as data flows through the network. Thus, dataflow graphs constructed in the model can incrementally compute avoiding redundant computations involving the same data. As discussed in [16], many programs contain a *compute-mutate* loop in which a shared state is updated after every iteration. In such cases, it is beneficial to identify the redundancy in computations and only evaluate the expressions that involve changed inputs. We consider a simpler example of *self-adjusting* computation by revisiting the network discussed in Fig. 1.

In the network shown in Fig. 6, the value of cell b depends on the output value of propagator f. With an appropriate merge function that appends values in a cell representing them as a tuple consisting of an index and the corresponding cell content, only a part of the propagator network (shown with bold lines) is activated each time the value of cell b is iteratively refined by f. Choosing a suitable index, the continuous-time semantics of GoDEL programs can be discretized to implement arbitrary functional-reactive programs.

B. Parallel Dataflow Computation

Nested parallel algorithms are implemented in GoDEL by leveraging the hybrid parallelism in the model – implicit task parallelism between propagators and data parallelism implicit in the cells. Fig. 7 shows a representative example of parallel matrix multiplication using row-decompositions. Input cells a and b hold matrices in $\mathbb{M}(4,4)$ whose product is to be computed. rN are projection propagators which project the Nth row from the matrix in its input cell. aNb are propagators that perform matrix-vector product on their input cells and merge the output to cell ab. The merge

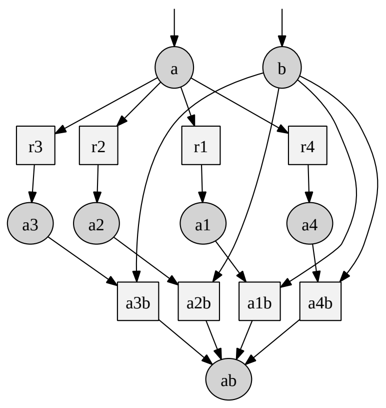


Figure 7. Matrix multiplication example: A × B = AB

operator associated with ab has to write the resultant vector at the appropriate row index to construct the final product matrix in ab. In the above example, the propagators aNb output a tuple of an index and the resulting vector.

It may look cumbersome to structure parallel computations as shown above. In practice, however, the propagators are coarsened such that the dataflow network is well-balanced. We further notice that merging values in a cell often results in several intermediate copies resulting in high space complexity. For shared-memory applications, the cells, however, hold pointers to the data as opposed to the actual data thus sharing data between subsequent cell refinements.

C. Newton-Raphson Root-Finding Method

GoDEL supports interval arithmetic for primitive numeric intervals defined as $Interval[n_L : n_H]$ where n_L represents a lower limit and n_H is the higher limit such that $n_L < n_H$. Interval arithmetic can be extended to vectors and matrices for increasing the reliability of numerical applications. Oftentimes, round-off errors during a computation result in an approximate solution of the problem. Probabilistic algorithms or uncertainty in model parameters also leads to inexact yet rigorous bounds on systems of equations.

Given an initial approximation x_0, the Newton-Raphson method proceeds to find the root for the real-valued function $f(x)$, that is an x for which $f(x) = 0$. Successive approximations are computed using the equation given below:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{1}$$

In some cases, Newton's method exhibits slow initial convergence or no convergence at all. However when extended to interval arithmetic, it is known to have asymptotic quadratic convergence. We consider an example where

$$f(x) = \sqrt{x} + (x+1)\cos x$$

$$f'(x) = \frac{1}{2\sqrt{x}} + \cos x - (x+1)\sin x$$

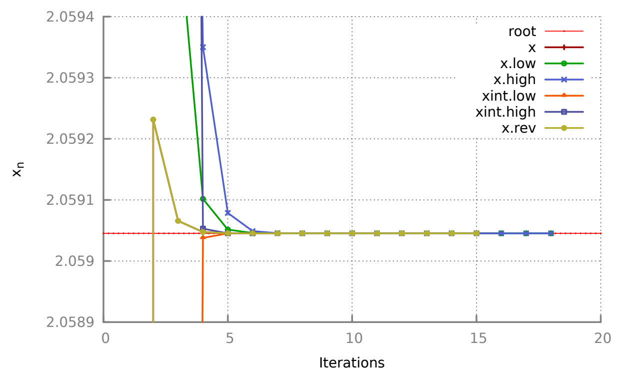


Figure 8. Convergence of Newton's method for $f(x) = \sqrt{x} + (x+1)\cos x$

Fig. 8 compares the convergence of the point and interval versions of Newton's method. The root of the function $f(x)$ is at 2.059045. From Fig. 8 we see that the interval-Newton method converges in about 5 iterations whereas the regular point version and the point version with lower and higher bounds takes about 16 iterations to find the root. The interval-Newton method computes the approximation by the following steps

$$x_{n+1} = x_n \cap \left(mid(x_n) - \frac{f(mid(x_n))}{f'(x_n)} \right)$$

The succinctness of the dataflow code in GoDEL can be seen in Fig. 9. The program uses user-defined operators for division (`div`) and subtraction (`sub`) since the operators ($\div$ and $-$) in GoDEL are reversible by default. This prevents the construction of a reversible network (which is meaningless in this case) and restricts the flow of data in one direction. As described in [17], we compute the inverse of the Newton-Raphson method using the equation

$$g(h) = \frac{f(x_{n+1} - h)}{-h} - f'(x_{n+1} - h)$$

V. Related Work

Static dataflow architectures [5] emerged in the mid-1970's with early work in dataflow languages and compilers focusing on single-assignment languages like LAPSE [18] and SISAL [19]; and on co-ordination languages like Lucid [20] and Id [21]. Most of these early dataflow languages focused on fine-grained dataflow suited to map effectively to the underlying dataflow machines referenced above. In contrast, our work relates more closely to a coarse-grained dataflow model. Since cells can be checked for emptiness and propagators do not block on empty cells (monotonicity), cells differ considerably from the commonly found *write-once*, *read-many* synchronization structures like *I-structures* in Id [21]. As a consequence, a propagator network cannot be expressed as an equivalent Kahn Process Network (KPN).

The advent of the massive multicore parallelism encountered in CPU and GPU architectures is causing a resurgence in data-driven execution models and dataflow has become more relevant in this area.

Other related work includes TFlux [22] which uses data-driven multithreading (DDM) [23] and has been extended to utilize accelerators such as the IBM Cell B.E. The Jade programming language [24] is a coarse-grained parallel programming language that has been ported to many systems and provides an execution model for loosely connected heterogeneous workstations. Capsules [25] is a parallel programming model that composes computations (called *capsules*) supporting multiple granularity of parallelism. The Codelet execution model [26] and Coarse-grained Dataflow [27] are two models that are very closely related to GoDEL – they focus on enhancing the programmability for developing high-performance parallel applications for extreme-scale machines. While most of the above languages allow explicit declaration of task and data dependence graphs, they rely on a slightly restricted form of unidirectional dataflow. GoDEL draws a comparison with multi-paradigm programming languages like Oz [28] since it subsumes multiple programming paradigms. Languages supporting reversible execution include Janus [29], psiLisp [30], and R [31], although most of these languages are not currently under development.

VI. Future Work

Although GoDEL supports multiple levels of task granularity, it is judicious to use GoDEL as a coarser-grained dataflow programming model than a finer-grained one to maximize performance. The propagator abstractions provided by GoDEL in the form of primitive operators gives an illusion that the operators map down to the corresponding hardware instructions. In reality, each propagator function has to pay a constant cost, that of creating a new coroutine. On a custom chip multiprocessor (CMP), it might be possible to snoop memory requests through a *Memory Mapped*

```
func main() {
  var x0, xi, fx, e Cell
  fx      <- f(x0)
  e       <- div(fx, fprime(x0))
  xi      <- sub(x0,e)

  x0      = Interval[2.0 : 3.0]
  fmt.Printf("Initial approximation %#v\n", x0)
  for i := 0; i < 50; i++ {
    fmt.Printf("%#v %#v %#v\n", x0, xi, fx)
    if xi == x0 {
      break
    }
    x0 := x
  }
}
```

Figure 9. Newton's method in GoDEL

Interface (MMI) and have "propagator instructions" that wait on the availability of "cell operands". Emerging accelerators like the GPU or the Cell B.E. have specialized processors that act on streams of data. Such architectures lend naturally to a dataflow model like the one described in this paper. Efficiently supporting multiple levels of granularity would involve aggressive compiler optimizations like propagator inlining and mapping GoDEL programs into an intermediate dataflow ISA. Other common optimizations include coarsening propagators which have unique dependencies and common sub-expression elimination using graph rewriting techniques.

Barring efficient bootstrapping, it would be trivial to extend GoDEL for distributed execution. Work-stealing models of task parallelism help in balancing the load among the available computational units. Such locality-aware scheduling strategies would favor in efficient distributed execution of applications exhibiting irregular parallelism. We plan to investigate distributed channel implementation using highly-tuned hardware-optimized communication libraries like MPI and GASNet. Our future work also includes extending the GoDEL compiler and runtime stack to implement some of the common distributed high-performance computing applications to compare against alternative execution models.

VII. Conclusion

There has been a significant body of work on dataflow languages. GoDEL builds on a flexible and expressive execution model based on propagation networks. The contribution of this work lies in a modern implementation of a general-purpose computational substrate based on multidirectional dataflow. The execution model supports multi-paradigm programming with first-class concurrency. GoDEL is fundamentally declarative in nature which makes it easier to express parallel programs by explicitly stating the dependencies between parts of the program. It takes a step towards proposing novel methods of execution for extreme-scale computing that support nested, hybrid parallelism, multiple levels of task granularity and explicit declaration of data and task dependencies.

Acknowledgment

The authors would like to extend particular thanks to Latchesar Ionkov for his helpful comments on the design of GoDEL. This work was partly performed at the Ultrascale Systems Research Center (USRC), a collaboration between Los Alamos National Laboratory and the New Mexico Consortium (NMC). NMC provides an environment to foster collaborative research through long-term interactions between LANL, professors, students and industry visitors. This publication has been assigned the LANL identifier LA-UR-11-11324.

References

[1] V. Sarkar *et al.*, "Exascale software study: Software challenges in extreme scale systems," *DARPA Information Processing Techniques Office, Washington DC.*, vol. 14, p. 159, 2009. [Online]. Available: http://users.ece.gatech.edu/~mrichard/ExascaleComputingStudyReports/ECSS%20report

[2] P. M. Kogge and et al, "ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems," *DARPA Information Processing Techniques Office, Washington, DC*, vol. 28, p. 278, 2008.

[3] E. DeBenedictis, "Reversible logic for supercomputing," in *Conf. Computing Frontiers*, N. Bagherzadeh, M. Valero, and A. Ramírez, Eds. ACM, 2005, pp. 391–402.

[4] S. Borkar, "Thousand core chips: a technology perspective," in *DAC '07: Proceedings of the 44th annual Design Automation Conference*. New York, NY, USA: ACM, 2007, pp. 746–749. [Online]. Available: http://dx.doi.org/10.1145/1278480.1278667

[5] J. B. Dennis and D. Misunas, "A preliminary architecture for a basic data flow processor," in *ISCA*, 1974, pp. 126–132.

[6] J. B. Dennis, W. Y.-P. Lim, and W. B. Ackerman, "The mit data flow engineering model," in *IFIP Congress*, 1983, pp. 553–560.

[7] J. R. Gurd, C. C. Kirkham, and I. Watson, "The manchester prototype dataflow computer," *Commun. ACM*, vol. 28, no. 1, pp. 34–52, 1985.

[8] Arvind and R. S. Nikhil, "Executing a program on the mit tagged-token dataflow architecture," in *PARLE (2)*, ser. Lecture Notes in Computer Science, J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, Eds., vol. 259. Springer, 1987, pp. 1–29.

[9] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken, "Empirical Study of a Dataflow Language on the CM-5," in *Proc. of the Dataflow Workshop, 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992, pp. 187–210. [Online]. Available: http://www.cs.cmu.edu/~seth/papers/culler-wdc92.pdf

[10] J. B. Dennis, "Data flow ideas for supercomupters," in *COMPCON*. IEEE Computer Society, 1984, pp. 15–20.

[11] A. Radul, "Propagation networks: A flexible and expressive substrate for computation," Ph.D. dissertation, MIT, Sep. 2009. [Online]. Available: http://dspace.mit.edu/handle/1721.1/49525

[12] G. J. Sussman and A. Radul, "The art of the propagator," MIT Computer Science and Artificial Intelligence Lab, Tech. Rep., 2009.

[13] R. Soulé, M. Hirzel, R. Grimm, B. Gedik, H. Andrade, V. Kumar, and K.-L. Wu, "A universal calculus for stream processing languages," in *ESOP*, ser. Lecture Notes in Computer Science, A. D. Gordon, Ed., vol. 6012. Springer, 2010, pp. 507–528.

[14] The Go Authors, "The Go Programming Language Specification," June 2011. [Online]. Available: http://golang.org/doc/go_spec.html

[15] P. Stanley-Marbell, *Inferno Programming with Limbo*. John Wiley and Sons, 2003, ch. Inferno Programming with Limbo.

[16] S. Burckhardt, D. Leijen, C. Sadowski, J. Yi, and T. Ball, "Two for the price of one: A model for parallel and incremental computation," *researchmicrosoftcom*, vol. 1, 2011. [Online]. Available: http://research.microsoft.com/pubs/150180/submitted.pdf

[17] K. S. Perumalla, J. P. Wright, and P. T. Kuruganti, "On the reversibility of newton-raphson root-finding method," 2008.

[18] J. R. Gurd, J. R. W. Glauert, and C. C. Kirkham, "Generation of dataflow graphical object code for the lapse programming language," in *Proceedings of the Conference on Analysing Problem Classes and Programming for Parallel Computing*, ser. CONPAR '81. London, UK: Springer-Verlag, 1981, pp. 155–168. [Online]. Available: http://portal.acm.org/citation.cfm?id=646739.702114

[19] R. R. Oldehoeft and D. C. Cann, "Applicative parallelism on a shared-memory multiprocessor," *IEEE Softw.*, vol. 5, pp. 62–70, January 1988. [Online]. Available: http://dl.acm.org/citation.cfm?id=624567.624690

[20] W. Wadge and E. Ashcroft, *Lucid, the dataflow programming language*, ser. A.P.I.C. studies in data processing. Academic Press, 1985. [Online]. Available: http://books.google.com/books?id=wLImAAAAMAAJ

[21] Arvind, Culler, and Maa, "Assessing the benefits of fine-grain parallelism in dataflow programs," *SC Conference*, vol. 1, pp. 60–69, 1988.

[22] K. Stavrou, M. Nikolaides, D. Pavlou, S. Arandi, P. Evripidou, and P. Trancoso, "Tflux: A portable platform for data-driven multithreading on commodity multicore systems," in *ICPP*. IEEE Computer Society, 2008, pp. 25–34.

[23] P. Evripidou and J.-L. Gaudiot, "A decoupled graph/computation data-driven architecture with variable-resolution actors." in *ICPP (1)'90*, 1990, pp. 405–414.

[24] M. S. Lam and M. C. Rinard, "Coarse-grain parallel programming in jade," in *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPOPP '91. New York, NY, USA: ACM, 1991, pp. 94–105. [Online]. Available: http://doi.acm.org/10.1145/109625.109636

[25] H. A. Mandviwala, U. Ramachandran, and K. Knobe, "Capsules: Expressing composable computations in a parallel programming model." in *LCPC*, ser. Lecture Notes in Computer Science, V. S. Adve, M. J. Garzarn, and P. Petersen, Eds., vol. 5234. Springer, 2007, pp. 276–291. [Online]. Available: http://dblp.uni-trier.de/db/conf/lcpc/lcpc2007.html#MandviwalaRK07

[26] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, "Using a "codelet" program execution model for exascale machines: position paper," in *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, ser. EXADAPT '11. New York, NY, USA: ACM, 2011, pp. 64–69. [Online]. Available: http://doi.acm.org/10.1145/2000417.2000424

[27] A. Soviani and J. P. Singh, "Optimizing communication scheduling using dataflow semantics," in *Proceedings of the 2009 International Conference on Parallel Processing*, ser. ICPP '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 301–308. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2009.66

[28] P. Van Roy, Ed., *Multiparadigm Programming in Mozart/Oz, Second International Conference, MOZ 2004, Charleroi, Belgium, October 7-8, 2004, Revised Selected and Invited Papers*, ser. Lecture Notes in Computer Science, vol. 3389. Springer, 2005.

[29] C. Lutz and H. Derby, "Janus: A time-reversible language," 1982. [Online]. Available: http://www.cise.ufl.edu/~mpf/rc/janus.html

[30] H. G. Baker, "Thermodynamics and garbage collection," in *In ACM Sigplan Notices*, 1994, pp. 55–59.

[31] C. R. Clark, D. Michael, and P. Frank, "Improving the reversible programming language r and its supporting tools," 2001.