2011 18th IEEE International Conference and Workshops on Engineering of Computer-Based Systems

Queral Networks: Toward an Approach for Engineering Large Artificial Neural Networks

Travis A. Hoffman, Jerzy W. Rozenblit, Ali Akoglu, Liana Suantak
Electrical and Computer Engineering Department, The University of Arizona
travish@email.arizona.edu, {jr,akoglu,liana}@ece.arizona.edu

***Abstract*—A generalization of an artificial neuron is introduced in this paper. Called the queron[1], this abstraction is the basic computational node of Queral Networks (QN). QNs are introduced as a parallel architecture expected to be an improvement upon Artificial Neural Networks (ANN). The fundamental properties of QNs are presented here: reusability, complexity management and human-readability. It is expected that this proposed architecture will allow the engineering of large, highly parallel computer systems with the computational benefits of ANNs while overcoming the challenge of developing ANNs. A brief case study is given to illustrate the QN concept.**

***Index Terms*—Artificial Neural Networks, Automatic Programming, Computation Theory, Evolutionary Computation, Parallel Architecture**

I. Introduction

MOORE'S Law suggests improving transistor technology may only allow speed advancements for 16 more years. Silicon transistors have been clocked at 500GHz[1]. Research suggests that carbon-based transistors could operate at 10THz [2]. Recently, processors have been demonstrated at 5GHz, with a distinct emphasis on parallelism [3]. Indeed, the only way to scale overall system computational power beyond the physical limitations of integrated circuits is through greater parallelism. However, Amdahl's Law [4] shows that parallelizing the von Neumann architecture yields diminishing returns except in specific cases [5].

There have been several attempts to create alternative parallel architectures, with varying degrees of success. The Connection Machine [6] is the progenitor of the Network-On-Chip (NOC) architecture [7]. NOCs reduce the communication overhead and are effective for easily parallelizable problems but are still bound by Amdahl's Law because they must synchronize shared states.

One parallel architecture of note is the Kahn Process Network (KPN) [8]. KPNs avoid the need for synchronization by defining non-blocking communication channels between independent computational agents which share state only through message passing. Dataflow languages derive from KPNs, but most recent research focuses on synchronous models [9], which undermines the benefit of parallelization.

Biological Neural Networks (BNNs) offer a very different parallel model of computation (MoC). The neuron's maximum firing rate is only 1kHz, typically around 300Hz, yet BNNs outperform computers in many problems, are more energy efficient, offer unmatched scalable and have been shown to be Turing Computable [10].

Artificial Neural Networks (ANN) help investigate the properties of BNNs, but little progress has been made toward engineering large systems of ANNs. We argue this is because ANNs represent a low-level language analogous to an assembly language for a single-operation CPU. Consider the challenge of developing a complex system with such a language; beyond the simplest of systems, the challenge would quickly become intractable. With a few notable exceptions[11], ANNs are trained *in toto*, limiting the scale of systems which can be investigated.

We argue a high-level language for ANN-like highly connected networks is required to develop large systems. We introduce Queral Networks (QNs) as a high-level language specification, providing human-readability, reusability, and complexity management by separation of concerns [12]. In the remaining sections, we describe the queron, QNs, and the properties of each. We present our system for investigating QNs and present examples of their use, illustrating their advantages over ANNs. We close with a discussion of the work required to fully develop this new approach.

II. The Queron

The term *queron* is derived from the ability to investigate (or query) every computational node's structure and behavior. The goal of the *queron* is to generalize the concept of a neuron so that ANNs can be analyzed, as is common with modern computational systems. The *queron* generalizes the neuron by allowing other operations to be implemented, by adding typing to the data which may be passed, and by providing meta-information about the *queron*.

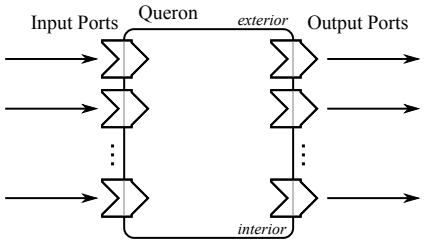

Figure 1. The queron, input and output ports.

The *queron* may informally be defined as an independent computation node with a structure and behavior fulfilling

[1] *Queron* and *queral* are pronounced like *query*.

978-0-7695-4379-6/11 $26.00 © 2011 IEEE
DOI 10.1109/ECBS.2011.27

a *contract*. The *queron* accepts *messages* to its *input ports* and produces *messages* from its *output ports* according to the behavior described by the set of *acceptance tests* in its *contract*. To fulfill a *contract*, a *queron* must have *input ports* accepting all *input types*, *output ports* offering all *output types*, and pass all *acceptance tests* defined in the *contract*. A *queron* may be implemented in one of two ways: by a network of *child querons* connected to the interor of its *input ports* and *output ports*, or by a native implementation. The emphasis is on the *contracts*; details of the implementation are unimportant as long as the *contract* is fulfilled. We now present a formal definition of the *queron*, key sub-components and brief discussions of each.

Definition 1. A VALUE is an ELEMENT of one or more TYPEs.

Definition 2. A TYPE is a SET of zero or more VALUEs.

A *type* is conceptually identical to a *set* in set theory. A *type* may be described in a number of ways. In *querons* and QNs, *types* are used to constrain which connections may be established. A *complex type* is like a *tuple* in set theory; they are aggregates of other *types*. A *composite type* is created from combinations of any number of *types* using standard set operations: union, intersection, etc.

Definition 3. A MESSAGE is a 3-TUPLE (p_s, p_d, v), where

1) p_s is a PORT called the *source port*.
2) p_d is a PORT called the *destination port*.
3) v is a VALUE called the *payload* such that $T(v) \cap T(p_s) \cap T(p_d) \neq \emptyset$. $T(p)$ is a FUNCTION which returns the SET of all TYPEs accepted by the PORT p. $T(v)$ is a FUNCTION which returns the SET of TYPEs containing the VALUE v.

Definition 4. A TEST is a PAIR (I_{IN}, I_{OUT}), where

1) $I_{IN} \in \{\lambda_{IN}, M_{IN}\}$ where λ_{IN} is a FUNCTION called the *test source* which produces M_{IN}. M_{IN} is a finite SEQUENCE of MESSAGEs, called the *source messages*.
2) $I_{OUT} \in \{\lambda_{OUT}, M_{OUT}\}$ where λ_{OUT} is a FUNCTION called the *test expectation* which produces M_{OUT}. M_{OUT} is a finite SEQUENCE of MESSAGEs, called the *expected messages*.

Test implementation is flexible; what matters is conformance to the *inputs* and *outputs* specified in the *contract*.

Definition 5. A CONTRACT is a 3-TUPLE (T_{IN}, T_{OUT}, B) where

1) T_{IN} is a non-empty SET of TYPEs called the *input types*, or *inputs*.
2) T_{OUT} is a non-empty SET of TYPEs called the *output types*, or *outputs*.
3) B is a non-empty finite SET of TESTs called the *behavior*, *tests*, or *acceptance tests.*

Definition 6. A PORT is a 4-TUPLE (Q, T, M, m_0), where

1) Q is a QUERON called the *owner*.
2) T is a SET of TYPEs called the *acceptable types*.
3) M is a SEQUENCE of MESSAGEs called the *input messages*, in the order received.
4) $m_0 \in M$ is a MESSAGE where $\forall m_i \in M - \{m_o\}, m_o \succ m_i$ called the *current message*.

A *port* allows passage between the exterior and interior of its *owner*, creating contextual separation of elements. Separation is enforced by requiring all *messages* to flow through a *port*.

Definition 7. A QUERON is a 4-TUPLE (P_{IN}, P_{OUT}, C, I), where:

1) P_{IN} is a SET of PORTs called the *input ports*.
2) P_{OUT} is a SET of PORTs called the *output ports*.
3) C_f is a SET of CONTRACTs called the *fulfilled contracts.*
4) $I \in \{\lambda, QN\}$ is called the *implementation*, where:
   a) λ is a FUNCTION called the *reaction function*.
   b) QN is a QUERAL NETWORK called the *internal queral network*.

The *queron* is conceptually similar to a *computational agent* in KPNs or a *system* in the Discrete Event Simulation System (DEVS) [13]. As in a KPN, a *queron's* state persists between receipt of *input messages*; state is only shared between *querons* by *messages* via *connections*. Thus, every *queron* may execute asynchronously [14] and a collection of *querons* is infinitely parallelizable [15].

III. The Queral Network

A QN is a *directed graph* with *ports* as vertices, and *connections* as the edges of the graph. A QN only exists as the implementation of a *queron*. The *ports* in the QN may only connect with the *ports* of the *owner* and the other *child querons*, subject to the restrictions in the definition of a *connection*.

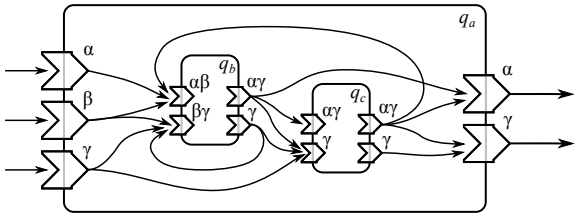

Figure 2. Valid connections between the typed ports of the queron q_a and the ports of q_a's child querons q_b and q_c

Connections are limited by a few simple, yet fundamental restrictions, described in (3). The *source port* and *destination port* must not be the same (1). The *source* and *destination* must have at least one *type* in common (2). The *source* must be a *input port* of q, or an *output* of a *child queron* of q (3). The *destination* must be an *output port* of q, or an *input port* of a *child queron* of q (4). Figure 2 illustrates proper *internal connections* between q_a's *ports* and the *ports* of its *children* q_b and q_c. Each *port* is labeled with a *type* in $\{\alpha, \beta, \gamma\}$, *composite types* are labeled with strings of the *types*, e.g. $\alpha\beta, \beta\gamma, \alpha\gamma$.

Definition 8. A CONNECTION is a 3-TUPLE (p_s, p_d, q), subject to

$$(p_s \neq p_d) \wedge \tag{1}$$

$$(T(p_s) \cap T(p_d) \neq \emptyset) \wedge \quad (2)$$

$$(p_s \in P_I(\{q\}) \cup P_O(Q_C(q))) \wedge \quad (3)$$

$$(p_d \in P_O(\{q\}) \cup P_I(Q_C(q))) \quad (4)$$

where

1) p_s is a PORT called the *source port*, or *source*.
2) p_d is a PORT called the *destination port*, or *destination*.
3) q is a QUERON called the *parent*. $T(p)$ is a FUNCTION which returns the SET of TYPEs accepted by PORT p. $Q_C(q)$ is a FUNCTION which returns the SET of *child querons* owned by the QUERON q. $P_I(Q)$ is a FUNCTION which returns the union of *input ports* owned by QUERONS in the SET Q. $P_O(Q)$ is a FUNCTION which returns the union of *output ports* owned by QUERONS in the SET Q.

There is no theoretical limit on the number of *connections* in which a *port* may participate but hardware and design considerations will certainly enforce practical limits. Other custom restrictions may be added for specific design criteria, but these fundamental restrictions are always required to ensure consistent separation of implementation from interface and proper flow of *messages*. In Figure 3, examples (d), (h) and (i) violate the separation of interface from implementation; examples (a) through (f) break the proper flow of *messages* in the system. Example (j) is a duplicate *connection*; this is disallowed simply for efficiency.

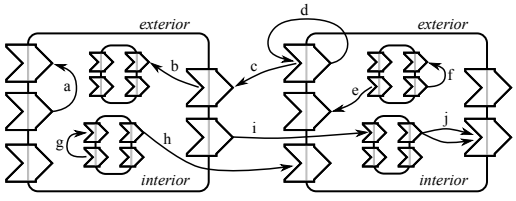

Figure 3. Invalid connections: (a), (f), (h) and (i) invalid end; (b), (c) and (e) wrong direction; (d) same end and start; (g) invalid start; (d), (h) and (i) crossing scope; and (j) duplicate connection

Definition 9. A QUERAL NETWORK is a 3-TUPLE (Q_C, C_N, q), where:

1) Q_C is a *set* of QUERONs called the *children* or *child querons.*
2) C_N is a *set* of CONNECTIONs called the *internal connections.*
3) q is a QUERON called the *owner* such that $\forall q_i \in Q_C$ $q = Q(q_i)$. $Q(q_i)$ is a FUNCTION which returns the *owner* of the ith QUERON in Q_C.

The flexible structure of QNs allows for a very large number of possible Q_C. Consider, as an example, developing a Q_C for a *queron* with two inputs and one output from a *Library* with ten *querons* each with two inputs and one output, there are

$$\left(\frac{10^3}{3!}\right) \cdot 2^{(6+1)\cdot(3+2)} = \left(\frac{500}{3}\right) \cdot 2^{35} \approxeq 1.145 \times 10^{13} \quad (5)$$

possible Q_C to search. Clearly, a brute-force search is impractical for Q_C with more than a few *querons*. It is not necessary to exhaustively search for solutions. We believe we can iteratively build complexity gradually by training increasingly more complicated *querons*. We will manually write some *querons* and evolve others. We believe this hybrid approach will yield positive results.

The strategy by which new QNs are developed is inconsequential to the runtime, as long as the *contract* is properly fulfilled. Evolutionary Programming (EP) has been applied to evolving neural networks[16], machine-language software[17], and has been shown to be an effective search technique for a wide range of problems. QNs are not restricted to any solution for development of new QNs, but initial efforts will investigate EP.

The investigator may construct QNs manually, but the power of QNs comes from this hybrid approach where some pieces of the system are developed by hand, and some by the investigator. It is our goal that an investigator will be able to comprehend an evolved solution, using that knowledge to develop alternative solutions.

Developing complex *querons* will be a process akin to teaching a person to solve a differential equation. It is impossible to start with a lesson on differential calculus; instead, there must be a large amount of training from basic principles: arithmetic, algebra, geometry, calculus. Each level of complexity builds upon the previous. There are many interesting features of QNs we will be able to leverage in their development and application. Thus far, we have identified:

Code Reuse: *Querons* are stored in a *Library* indexed by their *contract*. The *contract* makes it possible for the investigator to identify *querons* to be used in their solution. Each *queron* in a QN is a separate runtime instantiation of the *queron*, allowing many copies of the same *queron* to exist concurrently.

Scale-Free: A *queron* may be implemented with a QN of any number of *querons*, and may itself be included in a QN of arbitrary size. At every layer of such a hierarchy, the topology of the QN is similar. This feature helps manage complexity and allows a *queron* to be used at any layer of a hierarchy.

Runtime Efficiency: *Querons* recalculate their output values only when new input is received. Additionally, *querons* can provide answers with only partial information. *Querons* naturally perform just the right amount of computation required by a new input received. Their inherent modularity makes it possible to tune *querons* to the available hardware.

Scalability: The focus on *contracts* means a particular *queron* may be realized in customized hardware or in software with no effect on the overall operation of the system. Delays in *message* propogation affects only the overall speed of the system, not the behavior of the system. Thus, it is possible to distribute very large QNs over multiple machines. The effects of these delays can be easily mitigated by co-locating heavily-communicating *querons* on the same machine, or by allocating faster communication channels to *querons* with the heaviest traffic.

Automatic Development of QNs: The features of *querons* and QNs support automatic software development by machine learning. *Querons* can solve problems by reusing existing *querons* while incrementally increasing complexity. It is also

possible to continuously develop more efficient solutions to a *contract*, replacing existing *querons* with more efficient versions.

IV. The Queral Environment for Development

The Queral Environment for Development (QED) is a suite of tools for running, developing and investigating the properties of QNs. We have identified four key components for this system: *Nursery*, *Library*, *Runtime* and *Editor*, which we discuss below. Each are relatively standard, but will require tailoring for developing QNs. We have designed the system with an emphasis on parallelism, flexibility and interoperability, illustrated in Figure 4.

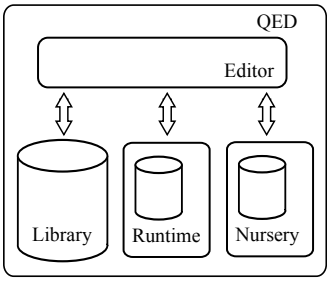

Figure 4. The QED architecture.

Nursery: A subsystem where machine learning techniques are applied to identify new solutions for unfulfilled *contracts*. This subsystem allows for investigating varied strategies for automatic development, including alternative strategies in parallel. A plug-in architecture enables easy integration of new strategies.

Library: A database storing *querons* which may be retrieved by *contract,* or by a portion of *contracts*. For example, it will be possible to search for all *querons* with a Boolean input port. There will be a variety of *Libraries*, tailored to their use case. For example, the *Nursery* will require storage of multiple variations of a *queron* for a given *contract* during the evolutionary process, but the *Runtime* will require only the best known *queron* per *contract*.

Runtime: A virtual machine for running *querons*. The *Runtime* handles loading of *querons* and routing *messages* between them. The initial implementation features a message-passing kernel running on a single machine, routing *messages* between *ports*, and schedules *querons* to recalculate their output when *messages* are received. Future versions will allow for *Runtimes* to be distributed across any number of machines, routing *messages* between them. Each *Runtime* has its own custom *Library*, which allow for versions of *querons* specific to the underlying hardware.

Editor: A graphical user interface for investigating *querons*. QNs may be represented in a textual format, but with potentially high-order graphs, they will be more easily understood visually. We are developing a visual programming environment which enforces the core restrictions on *connections*, allows additional user-defined restrictions, provides contextual help based on *types*, suggests *querons* based on *contracts*, manages complexity by scoping of the view and allows the user to view a QN in a variety of ways. Understanding large QNs will be a very challenging task; an equally advanced tool is necessary for the investigator to succeed.

V. Illustrative Example

A local farmer, Mr. Lausted, has grown frustrated with the unreliability of weather prediction for the area around his farm. He has tried, without success, to come up with some general rules using recent weather conditions to predict whether the next few days will be favorable for making hay. To cure properly, hay needs one to two days of zero rainfall and moderate winds to dry the cut hay before baling. If the hay has not dried properly, it can begin to grow mold, and will be unusable. If the cut hay is rained on, it can lose nutritional value, also making the hay unusable. Having better predictions of the weather will improve the yield of hay by minimizing wasted hay.

Mr. Lausted has meticulously recorded the local weather conditions at his farm, and has collected the weather recorded at weather stations in the four closest communities–Wheeler, Colfax, Elk Mound and Menomonie–every day for the last ten years. A map of the farm and the surrounding area is shown in Figure 5, with the weather stations marked. The data collected each day includes the following: the maximum, minimum and average temperature, the relative humidity, the average wind speed and direction, and the amount of precipitation. The database table structure is shown in Table I.

date	location	max °C	min °C	avg °C	hum %	speed m/s	direction °	precip cm
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
7/12/06	WHLR	24	17	20	54	10	345°	0.00
7/12/06	CLFX	23	15	17	67	12	0°	1.20
7/12/06	ELKM	27	21	24	51	7	331°	0.50
7/12/06	MENO	25	21	23	55	9	330°	0.00
7/12/06	FARM	25	18	21	65	8	323°	0.00
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table I
Weather data

A rule-based approach (as attempted by Mr. Lausted) based solely on the current weather conditions has proven ineffective, as has interpolating the predictions of the local weather forecasters. While it is likely that an expert in meteorology would be able to create a detailed model of the micro-climate around the farm, for the sake of this example, we will attempt to create a solution based only on the data available. We have a complete set of observed weather data for the last ten years. Additionally, we have obtained the predictions of weather forecasters in the area, shown in Table II.

We now present three solutions using different strategies: an analytical approach, using a ANN and with a QN. Each example should be considered independently of the others. We will give only an outline solution for the first two strategies; the fine details of implementation are not important. We will go into greater detail for the QN solution, to give a better

sense of the system, which is new. In doing so, we hope to give the reader an understanding of how QNs solve problems, their capabilities and how one uses them to solve complex computational problems.

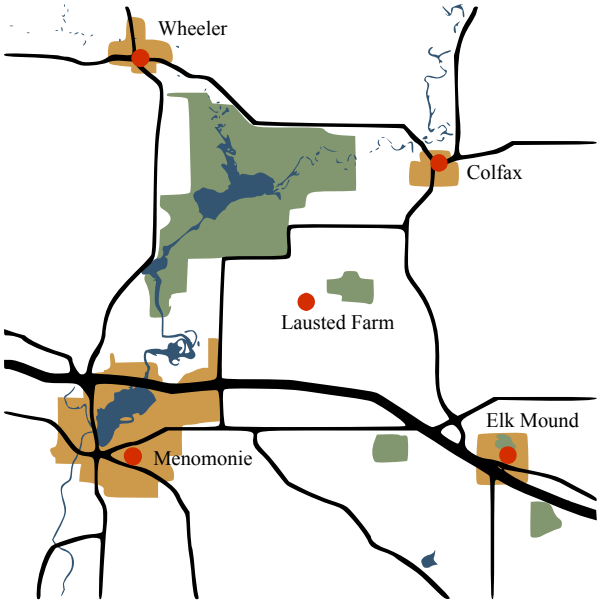

Figure 5. Map of the area around the farm, with locations of weather stations.

A. Solution Using an Analytical Approach

We approach this problem by first performing a meta-analysis of the data. The data may be grouped into classifications: temporal, spatial and meteorological. This suggests that it is prudent to evaluate the data along those three axes. We proceed by next analyzing the data along each of these three axes followed by another meta-analysis of the data within each of the three groupings. Along the meteorological axis, we look for first- and second-order patterns. For example, upon calculating a three-day sliding window average of meteorological data, we find the average temperature and the humidity are relatively good predictors of the day after the window and find the other meteorological data to be less useful. Along the temporal axis, we compare the data at each location year after year, and season by season. This alternating data/meta-data analysis strategy is continued until we have developed sufficiently robust understanding of the data.

We also look for ways to improve the data. For example, the location information is minimal; we enrich this by determining the latitude and longitude of each weather station, then calculating vectors from the farm to each, normalizing the location data to the farm. Other sources of data are added, such as weather forecasts for the towns so that we may evaluate the predictions and integrate them.

Having developed a feel for the data, we create hypotheses about data between axes. As an example, we have created and verified hypotheses about the relative importance of each weather station's data by time of year. Another hypothesis is that upwind weather stations are better predictors of weather

date	days prior	location	max °C	min °C	avg °C	hum %	speed m/s	direction °	rain cm
⋮	⋮	⋮	⋮	⋮		⋮	⋮	⋮	⋮
7/12/06	4	WHLR	26	18	22	45	7	12°	0.00
7/12/06	3	WHLR	25	18	22	49	8	2°	0.50
7/12/06	2	WHLR	24	16	21	53	12	350°	0.00
7/12/06	1	WHLR	24	17	20	57	9	340°	0.00
⋮	⋮	⋮	⋮	⋮		⋮	⋮	⋮	⋮

Table II
WEATHER PREDICTION DATA

to come. Yet another posits that certain weather stations are better predictors at different times of the year. The hypotheses are catalogued and ranked.

Lastly, we create software based on this set of hypotheses. Integrating and weighting the various components is very complicated and requires a lot of hand-tuning, but we have a robust set of data to confirm correct behavior. With the experience we have gained, we have a good idea about how to proceed on similar projects, several sources of data identified, and hypotheses to leverage in future endeavors. Even with these tools, it is clear that analysis of a different set of data will require significant effort.

B. Solution Using an Artificial Neural Network Approach

We review the sources of data with a focus on quality for training a neural network. It is helpful to normalize some data to values in the range [0,1). Other data, such as the dates of measurements are normalized in the training data, in terms of relative dates.

There are many ANN designs which would successfully solve this problem. Because the data has three major axes for comparison we choose a two-layer non-linear perceptron, as illustrated in Figure 6, a feed-forward ANN with two hidden layers. There are a large number of inputs: the last four days of weather data for each weather station plus the farm, and the last four days of predictions for each weather station. The weather records have 10 values, predictions have 11. Thus, we have 376 raw inputs.

Selecting the appropriate number of hidden nodes is not an exact science. With too many hidden nodes, the ANN will memorize the training data, not find a general solution; with too few, the ANN will generalize the solution, but with too high an error rate. Selecting an appropriate training data set is similarly inexact and requires care. One common heuristic is to use 10% of the full set of data.

We randomly sample 365 days (10%) of data for our training data. To start, we construct an ANN with a relatively small number of hidden nodes following a 3/2/1 heuristic. This first ANN has 376 input neurons, 230 neurons in the first hidden layer, 115 in the second hidden layer, with one output neuron.

Next, we iteratively try ANN solutions; training the ANN and evaluating its accuracy during each step. If the error rate

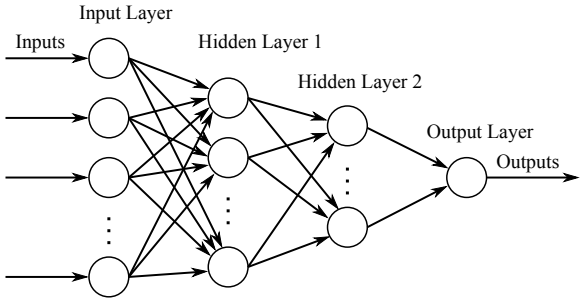

Figure 6. ANN with two hidden layers

is low enough, the process is complete. If the error rate is too high, we modify the number of hidden nodes in the ANN, train it and evaluate its accuracy. We repeat this process until a suitable solution is found. With the experience we have gained, we have a good idea about how to proceed on similar projects, several sources of data identified for use in future endeavors. Even with these tools, it is clear that we will have to repeat the entire process of training any subsequent ANN.

C. Solution Using a Queral Network Approach

We identify the *types* used along these axes. The basic *types* include: Location, Date, Temperature, Humidity, Windspeed, Direction, Name, Precipitation. From these basic *types*, we can compose two other *types*: WeatherData and Prediction, representing the high-level data structures shown in Table I and Table II.

Continuing, we develop a *contract* for the goal *queron*. Recall that a *contract* defines the structure and behavior of a *queron*; we first need to define the structure, which provides the interface to the *queron*. Restating in terms of a *contract*, the structure of the goal *queron* is: a *queron* which accepts weather data and predictions from nearby weather stations, then produces a prediction for the farm and answers the question, "Are conditions right to make hay?" The most concise representation has just two inputs and two outputs, illustrated in Figure 7. For this design, we leverage the ability of a *queron*'s output value to reflect more than just the most recently received *message*.

Next, we define the behavior of the *contract* as a set of *acceptance tests* derived from the gathered data. Each *test* must adhere to the structure of the *contract* and specifies the correct *output messages* for a set of *input messages*. The goal *queron* will accept new WeatherData and Predictions each day while retaining the previous day's data, then sending new *output messages*. Our *tests* are sets of *input messages* and the resulting *output messages* expected to be received. By carefully altering the order and randomizing the *input messages* within a *test* we create a much larger number of *tests* from the data at hand. Doing so helps ensure the *queron* does not become dependent on the ordering of *input messages*, only on which *messages* have been received. By strategically removing, modifying and adding some *messages*, we can introduce noise into the inputs, creating additional *tests* which adds robustness and helps generalize the solution.

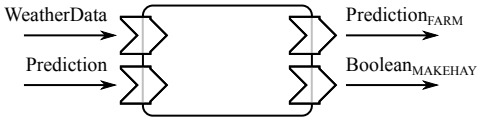

Figure 7. Structure of queron to predict weather at the farm

With a well-specified structure and behavior of the goal *queron* we can now pass the *contract* to the *Nursery* to begin the search for solutions. The *Nursery* is a task which eventually returns an appropriate QN implementation for the specified *contract*. In practice, the search space is incredibly large, and the search is very expensive. We can help by reducing the search space by a process, which should sound familiar to any experienced problem solver.

Taking a top-down perspective, we break the problem down into sub-problems, specifying *contracts* for *querons* which can be used by the *Nursery*. In doing so, we can begin to solve the problem in layers. The first simplification is a *queron* with an *input port* accepting Prediction *messages* and an *output port* offering Boolean *messages* answering the question, "Are conditions right to make hay?" This effectively splits the problem into two sub-problems. Another useful *queron* accepts each component of the Prediction *type* individually and returns a Prediction. Yet another converts a Prediction *input message* into a WeatherData *output message*.

Turning to a bottom-up perspective, we create a suite of basic *querons* from which complex solutions can be built. We initialize the suite with all *querons* accepting or sending *messages* of any of the *types* in use. We find a full suite of *querons* in the *Library* that perform basic arithmetic (addition, subtraction, etc.), higher-level mathematical (sine, cosine, integration, etc.) and comparison (less than, greater than, etc.) operations on real numbers, integers and booleans. The suite also includes *querons* for working with Dates and Locations. Next, we add *querons* to convert a Precipitation, Temperature and Humidity value to a RealNumber. We also create *querons* for converting Windspeed, Direction and a Location into a Vector; we think these may prove useful in normalizing and evaluating the relative impact of a prediction, by its location and wind conditions.

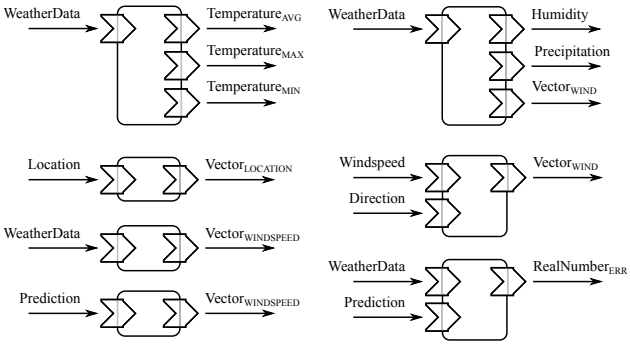

Figure 8. A sampling of sub-components for the solution QN

We continue to develop more complex components from both the "bottom" and problem sub-components from the

"top" of the problem, until we have a manageable gap between *querons* in the "middle" of the problem. Ideally, this progresses until the *Nursery* only has to find a solution for the "mysterious" part of the solution. We then pass the bootstrap suite of *querons* for the *Nursery* to begin its search.

D. Discussion

The example given is simple enough that an effective solution can be developed by any of the three techniques. Let us assume we have successfully developed a reliable system. The farmer is so pleased with the results that he has told all his friends and now we have several potential customers with similar requests.

With the traditional software solution, we have easily reusable software and a good understanding of the area's weather patterns. With a little effort, we should be able to reapply our knowledge to adapt the system to any new customers. If a customer is skeptical, we can explain how our software works at a conceptual level.

With the ANN solution, we have not developed an understanding of the weather patterns. If we are lucky, we will be able to reuse our ANN for the other farmers. It seems most likely, however, that we will have to retrain an ANN for each farmer's conditions, duplicating effort and providing a unique solution for each. It would be challenging to estimate the reliability of each solution since there is no code reuse between them. One alternative approach is to train a new ANN which works with the data for all farmers. A solution for this will certainly be more complex and, therefore, more difficult to train successfully. In either case, if a customer is skeptical, we most likely cannot explain how our software works at a conceptual level.

QNs provide a hybrid approach to development, allowing the investigator to focus on the architecture of the solution, letting the details of the implementation be evolved by the system. In the process of incrementally developing the *queron* to predict the weather, we have gained insight into the structure of the QN and required sub-components. We have a suite of components that can be reused to create similar solutions for new customers. With careful analysis of the QN, we can gain additional insight into the way these components are connected to each other. With a solution in hand, using the *Editor,* we can explore modifications as a means toward improvement and to gain insight into the workings of the QN. With careful analysis, we can develop a conceptual understanding of how the QN works at a conceptual level.

VI. Perceived Benefits

We believe QNs have several features which provide distinct advantages over ANNs and the von Neumann architecture. QNs represent a generalization of the computational model of ANNs. We can create a *queron* which behaves just like a neuron, and connect these in the same ways that neurons are connected in ANNs. With this, we can seamlessly integrate ANNs into QNs, by implementing the desired ANN as a *queron*. The benefit of this is the ability to leverage existing research into ANNs.

The *queron* architecture also has a distinct advantage over the von Neumann architecture in terms of parallelism. *Querons*, as with KPNs are naturally parallelizable. They do not require synchronization because there is no shared memory resources. Therefore, we claim *querons* are not subject to the limitations of Amdahl's Law. Experimentation will demonstrate the validity of this claim.

Querons have tight integration between specification and implementation; every *queron* must fulfill a *contract*. Most programming languages rely on a carefully-worded natural-language specification of behavior, and perhaps a suite of tests for key libraries; in QNs a more rigorous, less ambiguous definition is the focus. From the beginning, the investigator focuses on specification of the *contract* for a *queron*, with solutions to fulfill the *contract* developed later by the system, freeing the investigator to focus on the goals for the system, less on the details.

Lastly, we argue that QNs may make it possible to develop systems in more challenging problem spaces, potentially by subject-matter experts who are not trained as programmers. Typically, non-programmers do not have the skills nor training to define an algorithm to solve a problem, yet most can identify when a program is operating correctly. Subject matter experts can develop useful software as long as they can create appropriate tests, training, or are able to verify correctness. Moreover, there is a whole class of problems which are challenging to define or are somehow subjective. ANNs have provided one technique for solving such problems. QNs, we claim, provide a much better approach for solving and investigating these problems.

VII. Future Work

Many of the concepts used in defining *querons* are not unique. *Querons* are similar to artificial neurons, use features of KPNs and use EP, all of which have been researched. Yet, much research into the proper application and tailoring of these features for QNs remains.

We need to develop a thorough understanding of the complexity of QNs. This will be crucial for developing heuristics for estimating the minimum size of a solution. These heuristics will be especially useful while developing new solutions by EP. By carefully choosing the starting size we hope to avoid unnecessary effort and evolutionary dead ends. It is also desirable to develop a more rigorous proof demonstrating that QNs represent a generalization of ANNs, and a proof that QNs are Turing computable.

Development of the QED is in the early stages. We will soon present an initial implementation for testing purposes, but much research into the implementation remains. The *Nursery*, by its pluggable architecture, allows investigation of state-of-the-art machine learning algorithms. The *Runtime* requires investigation into scalability and in how best to allocate resources. The *Editor* will require research into how best to manage and present the complexity to the investigator.

VIII. Concluding Remarks

We have presented a computation architecture providing a higher-level perspective which we feel is novel, highly paral-

lelizable, scalable and powerful. While many of the concepts are well known, we believe we have combined these technologies in an unprecedented way. Much work remains, but we feel QNs will make it possible to develop more complex parallel systems and improve research into biologically-inspired and high-performance computation.

REFERENCES

[1] M. Cooke, "Silicon transistor hits 500ghz performance," *III-Vs Review*, vol. 19, no. 5, pp. 30–31, 2006.

[2] S. J. Tans, A. R. M. Verschueren, and C. Dekker, "Room temperature transistor based on a single carbon nanotube," *Nature*, vol. 393, pp. 49 – 52, May 1998.

[3] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar, "A 5-ghz mesh interconnected for a teraflops processor," *IEEE Micro*, vol. 27, pp. 51–61, Sept.-Oct. 2007.

[4] J. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, May 1988.

[5] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, July 2008.

[6] W. D. Hillis, *The Connection Machine*. PhD thesis, MIT, 1985.

[7] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Öberg, K. Tiensyrjä, and A. Hemani, "A network on chip architecture and design methodology," in *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, pp. 105–112, 2002.

[8] G. Kahn, "The semantics of a simple language for parallel processing," *Information Processing*, 1974.

[9] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys*, vol. 36, pp. 1–34, March 2004.

[10] H. T. Seigelmann and E. D. Sontag, "Turing computability with neural nets," *Applied Mathematics Letters*, vol. 4, no. 6, pp. 77 – 80, 1991.

[11] J. Reisinger, K. O. Stanley, and R. Miikkulainen, "Evolving reusable neural modules," in *Genetic and Eolutionary Computation - GECCO 2004*, vol. 3103, pp. 69–81, Springer Berlin / Heidelberg, 2004.

[12] E. W. Dijkstra, "On the role of scientific thought," in *Selected Writings on Computing: A Personal Perspective*, pp. 60 – 66, Springer-Verlag, 1982.

[13] B. P. Zeigler, *Multifacetted modelling and discrete event simulation*. San Diego, CA, USA: Academic Press Professional, Inc., 1984.

[14] N. A. Lynch and E. W. Stark, "A proof of the kahn principle for input/output automata," *Information and Computation*, vol. 82, no. 1, pp. 81–92, 1989.

[15] T. M. Parks, *Bounded Scheduling of Process Networks*. PhD thesis, University of California at Berkeley, 1995.

[16] X. Yao, "Evolving artificial neural networks," *Proceedings of the IEEE*, vol. 87, pp. 1423–1447, Sep 1999.

[17] R. L. Crepeau, "Genetic evolution of machine language software," in *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications* (J. P. Rosca, ed.), pp. 121–134, July 1995.