Exploiting Data Parallelism in Artificial Neural Networks with Haskell

by

Gregory Lynn Heartsfield, B.S.

REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2009

Exploiting Data Parallelism in Artificial Neural Networks with Haskell

APPROVED BY

SUPERVISING COMMITTEE:

Joydeep Ghosh, Supervisor

Christine Julien

Dedicated to my family, who have always supported my education.

Exploiting Data Parallelism in Artificial Neural Networks with Haskell

Gregory Lynn Heartsfield, M.S.E.
The University of Texas at Austin, 2009

Supervisor: Joydeep Ghosh

Functional parallel programming techniques for feed-forward artificial neural networks trained using backpropagation learning are analyzed. In particular, the Data Parallel Haskell extension to the Glasgow Haskell Compiler is considered as a tool for achieving data parallelism. We find much potential and elegance in this method, and determine that a sufficiently large workload is critical in achieving real gains. Several additional features are recommended to increase usability and improve results on small datasets.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Efficient classification of data is a common problem found in many fields of study. A wide range of statistical techniques exist for creating automatic classifiers, based upon a set of correctly classified examples. Common examples include handwriting recognition, automated medical diagnosis, and detection of network intrusion. The classification technique we have chosen to focus on is the artificial neural network, specifically the feed-forward variety using backpropagation as a training algorithm. In general, this provides a well-performing classifier in terms of accuracy, space, and time, although initial training can be quite expensive. We are specifically interested in neural networks because they often have a great potential for parallelism, which matches the present trend of computational capacity growing in terms of increasingly many cores instead of additional sequential operations per second. Functional programming languages, such as Haskell, are also well positioned to take advantage of this shift in hardware architecture. It seems natural, therefore, to investigate how the unique parallel programming environment that Haskell offers may be used to aid in the development and use of neural networks.

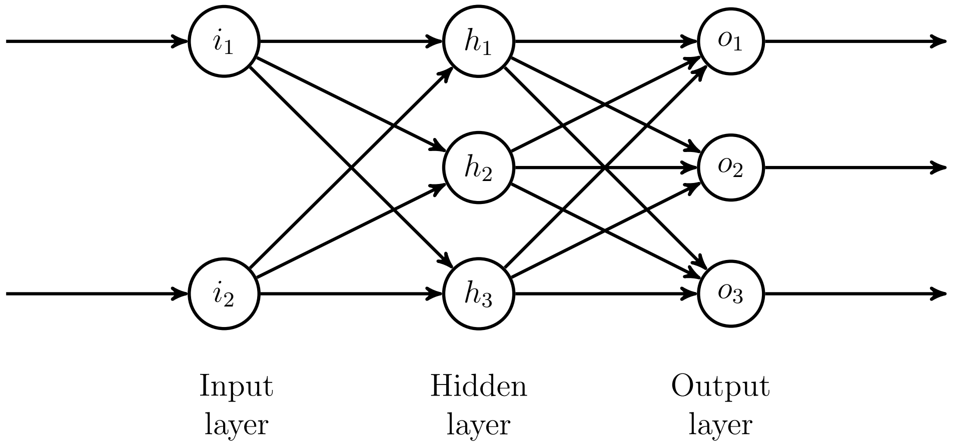
1.1 Neural Networks

Neural networks originated from the study of biological nervous systems, where it was observed that logical systems could be built to model how neurons interacted [27]. In our context, we are interested in artificial networks for their ability to classify data, not mimic biology. Additionally, we want a method for training these networks so that they give acceptably accurate classifications. While there are many possible architectures in which neurons and their connections to each other could be arranged, we restrict ourselves to a well understood and studied subset, called feed-forward networks. As the name implies, connections in these networks always proceed in the same direction, and are always from one grouping of nodes (a layer) to the next. An example of this type of network is shown in Figure 1.1, with three layers that are fully connected in sequence.

1.1.1 Feed-Forward

Performing classification with an existing feed-forward network is a straightforward process of calculating the output of every node, starting with the inputs, and proceeding layer by layer until the output of each node in the final output layer is known. Input nodes are assigned values from the example we would like to classify, which they output unchanged. Every other node's output is determined from a two step process. First, the total input to the node is calculated by taking the sum of all the products of each connection's weight and its source node's output. Then, an activation function is applied,

Figure 1.1: Feed-Forward neural network



which gives the node's actual output. If we say the connection weight to node j from node i is w_{ji}, the output of a node k is z_k, and the activation function is σ then we can give the output for any non-input node as Equation 1.1.

$$z_k = \sigma(\sum_j w_{ji} z_j) \tag{1.1}$$

A common choice of activation function is the binary sigmoid function, shown in Equation 1.2.

$$\sigma(t) = \frac{1}{1 + e^{(-t)}} \tag{1.2}$$

This is an attractive function for several reasons. The range of the sigmoid is $(0, 1)$, which resembles the off-on binary output of the biological neuron. As we will see shortly, one of the most important features of the sigmoid function is that it is smooth at all regions, and therefore differentiable. Finally, it is non-linear, important because many of the problems we wish to solve with these networks will be non-linear as well.

Once all nodes' outputs have been computed, we can take the output node with the largest activation value as the selected class for the given input.

1.1.2 Backpropagation

The challenge, of course, is how do we create a network with the correct configuration of weights, so that running the feed-forward algorithm will yield accurate results? With any non-trivial network, the number of dimensions we can vary in search of a good network is huge, so we need a method of optimizing the search for whatever "good" means. We stated previously that we intend to use a set of data that includes correct input and output values for the classification task. A "good" result then, is when the network is presented with a training input vector, the outputs match those in the training output vector. Taking the sum of the square of the difference for each output results in a convenient and suitable definition for error.

Now we know the error at each output, which is not useful by itself. We need to propagate this error to prior layers, so that we can correct it at its source. This is why the backpropagation algorithm is used, to provide us with a method for assigning error or blame to the nodes. Backpropagation involves two steps. First, we work backwards, assigning error to each node based on the previous layer's output (starting of course, with the output layer). Then, the connection weights are modified based on the error, in a direction that minimizes future error for this instance.

The error for any hidden node is taken to be the product of the derivative of the node's activation function at its current output, and the sum of all downstream (closer to the output layer) weights multiplied by the error of the node to which they connect. Intuitively, a node's responsibility for downstream error is related to how much it has contributed to the error of downstream nodes. The derivative of the sigmoid activation function is defined in terms of the function itself, shown in Equation 1.3. This is particularly nice because if we can keep track of a node's output, computing the derivative is very efficient. The error for a hidden node is then given by Equation 1.4, where o_j is the output of node j, and the summation is over every downstream connection from hidden node j to node i.

$$\frac{d\sigma(t)}{dt} = \sigma(t)(1 - \sigma(t)) \tag{1.3}$$

$$\delta_j = o_j(1 - o_j) \sum_i w_{ji} \delta_i \tag{1.4}$$

Once the error has been assigned to each node, the final step is simple, we update all the connection weights to move in the direction minimizing the error. The amount in which we move is effectively the speed at which the network learns. It is common to use a constant η to scale this amount, which allows us to manipulate the learning rate. By increasing the rate, we can minimize error quickly and increase network performance, but we trade off the ability to differentiate small regions of the solution space, and cause the results to bounce around without actually improving. A common optimization is to dynamically change the learning rate in response to the gradient encountered [19]. For clarity, we assume a fixed learning rate, and update connection weights based on Equations 1.5 and 1.6.

$$w_{ji} = \Delta w_{ji} + w_{ji} \tag{1.5}$$

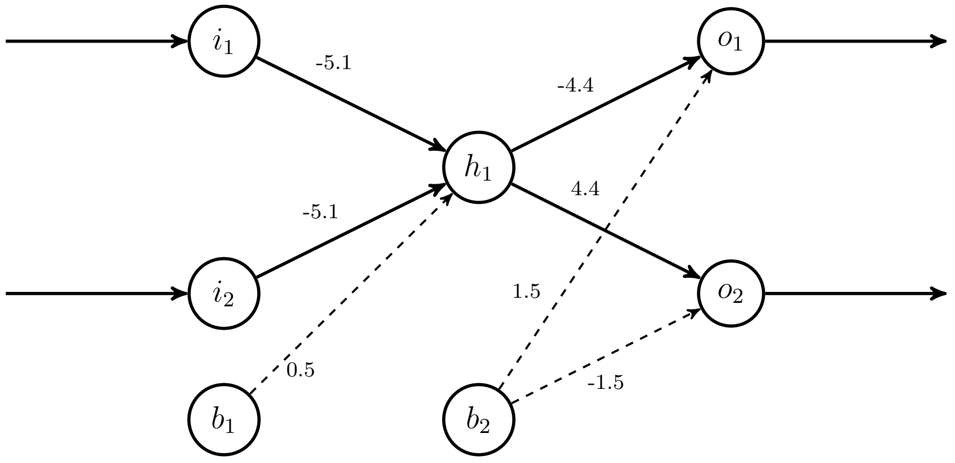
$$\Delta w_{ji} = \eta \delta_i w_{ji} \tag{1.6}$$

Figure 1.2 shows a network that was trained using backpropagation to correctly classify logical disjunction. This network includes the use of bias nodes, which have fixed activation values, shown here as 0.95. These inputs allow the decision boundaries to not be restricted to crossing through the origin, but instead move depending on the connection weight from the bias to the node [12]. Table 1.1 gives the activation value for the nodes in the network, and shows how this leads to a correct classification for all inputs.

1.2 Haskell and Functional Programming

The Haskell language originated from a desire to unify the functional programming research community around a common syntax, from the multi-

Figure 1.2: Neural network approximating logical disjunction



tude of functionally inspired languages that had been invented in the 1980's [17]. The distinguishing and notable features of the language are that it is pure, functional, lazy and is statically typed. We discuss each of these in turn, making notes when relevant to how this affects parallel programming.

1.2.1 Purity

In contrast with the vast majority of languages, Haskell disallows side-effecting statements. There is no built-in notion of destructive update, instead, changes to data structures are defined in terms of the creation of a new data structure. This imposes a certain rigour on programs, since it is no longer possible for any part of the program to manipulate program state. Programming is done by building up functions which consume only what they are given, and

Table 1.1: Node activation values for logical disjunction network

i_1	i_2	b_1	h_1	b_2	o_1	o_2	**result**
0	0	0.95	0.617	0.95	0.216	0.784	false
0	1	0.95	0.010	0.95	0.799	0.201	true
1	0	0.95	0.010	0.95	0.799	0.201	true
1	1	0.95	0.000	0.95	0.806	0.194	true

produce only what their type signatures declare.

1.2.2 Higher-Order Functions

Haskell functions are no different than other datatypes such as integers or strings. They can be named, passed as arguments, and returned as a result. Therefore, we say that Haskell supports higher-order functional programming. This turns out to be a very convenient property for parallel programming, since it provides an abstraction that often eliminates unnecessary imperative programming. Consider two program fragments that are intended to do the same type of work, incrementing a list of integers, shown written in Java first:

```
for (Integer i : collection) {
    i++;
}
```

and now in Haskell:

```
map (+1) collection
```

While the programmer's intent in both of these fragments appears the same, to perform an action over every element of a collection, the Haskell version is amenable to parallelization, while the Java version is not. This is because there is no implied order by the use of Haskell's `map` function, the compiler is free to increment the elements forwards, backwards, or all at once. And in fact, by using the syntactically equivalent `parMap` function, work on different parts of the list may happen concurrently. By emphasizing functions instead of iterative procedures, opportunities for parallelization are greatly increased.

1.2.3 Laziness

Just as it sounds, laziness is all about doing as little work as is required. In the context of a programming language, it means that Haskell does not perform computations unless there is some assurance that the result will be used. While the obvious benefit would seem to be in performance, since no computation is faster than doing nothing at all, a more important advantage may be that it enables more modular programming [18]. In practice, laziness is not free, since work that is not performed immediately must be remembered in case it is needed, and is stored as an unused computations (known as a "thunk"). In certain applications, it can be more space and time efficient to simply do the work immediately (strict evaluation). Fortunately, Haskell is only lazy by default, and there are several methods of annotating datatypes and functions so that evaluation is performed immediately.

1.2.4 Static Typing

Finally, Haskell has a static type system with inferencing performed by the Hindley-Milner algorithm, so that explicit type annotations are rarely required. In addition to controlling side-effects, this allows for good performance. The language can be compiled down to machine code, and performs quite competitively, even with C in some circumstances. With respect to parallelism, static typing is advantageous here as well. It is because of the static guarantees that pure functions provide that we are able to automatically parallelize many loop-like structures in Haskell to run concurrently, and do so in a safe manner.

Chapter 2

Parallelizing Neural Networks

After describing the backpropagation algorithm, Rumelhart et al. [31] made the following remark which characterizes why we are interested in finding methods to efficiently parallelize neural networks.

> Parallel computers are notoriously difficult to program. Here we have a mechanism whereby we do not actually have to know how to write the program in order to get the system to do it.

In other words, we can do the hard work of writing a parallel artificial neural network once, and then obtain parallel versions of the particular networks that run on our implementation for free.

As to why having a parallel implementation is desirable, it has been widely recognized that future increases in microprocessor performance will come in the form of increased parallelism instead of faster sequential processing [4]. As an anecdote, the author notes that he recently upgraded from a 2.3 Ghz desktop computer, to one that has a clock rate of 2.26 Ghz. The difference, taken for granted now, is that the new computer is capable of concurrently running 16 hardware threads, while the previous machine could only manage two.

2.1 Parallel Algorithms

A common way of looking at parallelism is to divide it into two models: task parallelism, and data parallelism [3]. We focus later in this paper on data parallelism, but for completeness, both will be discussed in this chapter. Task parallelism is a low-level method for explicitly controlling concurrent activities in a system, often used both for more advanced control structures, and greater performance in parallel environments. It is characterized by threads, mutexes, semaphores: the things that make concurrency “notoriously difficult”. Data parallelism on the other hand, hardly looks like concurrency at all from the programmers point of view. Programs are written in a sequential style, where one operation may be performed repeatedly over a large set of data. Behind the scenes, task parallelism may be used to support this illusion of sequencing. One significant advantage of the data parallel style is that, given the underlying technology is correct, parallelism can be exploited without risk of the program going wrong due to deadlock, livelock, or other concurrency related problems.

2.2 Multiple Network Parallelism

There are several methods which can easily make use of multiple processors, even when a whole network has been programmed to run sequentially. These typically fall on the side of task parallelism, since the operations run in parallel are distinct and complex, namely the entire feed-forward and back-propagation algorithms.

2.2.1 Competing Network Architectures

Choosing a neural network architecture in itself may be quite challenging, and so experimenting with many different networks simultaneously may be desirable. Some of the variables that may be adjusted include how many hidden layers, the number of nodes per layer, fully or partially connected layers, coding for output nodes, activation functions, and many others.

Training multiple varying architectures can be done using entirely isolated processes, with the only communication required being notification that the completion criteria (accuracy, training iterations, etc.) has been satisfied. For example, a problem may require a minimum threshold of accuracy, using the simplest network architecture. Training and evaluating several increasingly complex networks, each individual network running sequentially, but with a sufficient number of them running concurrently, would be a simple way to take advantage of the multiple processing resources on a single machine.

2.2.2 Varying Initial Conditions

The backpropagation algorithm is a gradient descent method that always attempts to move in the direction of decreasing error from its starting location, which is determined by the random connection weights we assign during network creation. The usual concern with local optimization techniques is that they may get stuck in local minima, since there is no inherent mechanism for a network to determine the difference between a local minima, and the global minima.

There are many techniques for avoiding or reducing the impact of local minima, such as incorporating momentum (past gradients combined with the current gradient) into weight correction [12], choosing the initial weights more carefully [32], and the use of the exponential function as an energy function (in place of the sum-squared error function) [1].

A standard solution to this local optimization challenge is to simply run the network multiple times, from different random initial configurations. This may yield several different solutions, which can be evaluated against each other in order to determine the best result. Methods exist for determining how many random networks should be trained in this fashion in order to get acceptably close to the optimum result [23].

2.3 Single Network Parallelism

In contrast to running many different networks simultaneously, we may want to focus on executing or training a single network as quickly as possible. If we have used the multiple network techniques for selecting an architecture, and have chosen a more refined method for avoiding local minima than rerunning, optimizing a single network could be quite worthwhile. The parallelism involved is much finer grained however, and therefore more difficult and performance sensitive. At this level, data parallelism starts to become important, although this distinction can become blurry as some algorithms may incorporate both task and data parallelism.

2.3.1 Epoch Training

Updates to each connection weight do not have to be applied immediately after error backpropagation. Instead, weight updates can be calculated for each training instance in batch, summed, and then applied at the end of a training "epoch." Since the result of running the feed-forward algorithm remain the same throughout a single epoch, multiple threads may concurrently and independently calculate error across training instances. At the end of the epoch, threads sum up the combined contribution to each weight, and the next training epoch begins. Summing the weight contributions across a small number of processors is fast, so this allows both the feed-forward and backpropagation steps to be parallelized. The disadvantage to this method is that the cumulative affect of all the backpropagation steps results in much smoother corrections being made, which increases the risk of becoming stuck in local minima [12].

2.3.2 Node Parallelism

We can also view a neural network as a collection of independent processing units, that is, each node is a simple processor. This extracts the maximum possible parallelism from the model, but shifts the burden to the communication links in a way that standard processors, even multi-core, are not designed for. Hardware has been created that takes advantage of this model, employing thousands of simple processors in a highly connected mesh or hypercube arrangement [14] [13]. This hardware is not common however, and it is likely that for quite some time we will be simulating networks with far more nodes than we have processors available on a single computer.

2.3.3 Layer Parallelism

In the standard feed-forward and backpropagation algorithms, we update nodes one layer at a time, either in the forward or back directions. If epoch training is not used, then we are repeatedly shifting between sequential evaluation of layers in each direction. In a non-trivial network, each layer may have a significant amount of work to perform. Feed-forward networks require we sum the product of each node/connection combination, which corresponds to multiplication of a vector (the node outputs) by a matrix (connection weights between two layers). The result of this is the output values for the next layer.

The vector-matrix product operation is very widely used, and highly-optimized versions exist in numerical linear algebra software packages, examples of which are LAPACK [2] and BLAS [24]. Additionally, it can be readily parallelized [6]. Matrix operations are also a prototypical example of data parallel algorithms, since we specify an operation to be performed over a large set of data.

Chapter 3

Data Parallel Haskell

The Glasgow Haskell Compiler (GHC) contains a rich set of parallel/concurrent programming models, supporting explicit threads, semi-explicit hints for concurrency, data parallelism [26], and transactional memory [15]. To some extent, all of these components utilize the underlying Concurrent Haskell system [30]. This is important, so that as these higher-level abstractions are built, developers do not need to reinvent the wheel with respect to thread management and scheduling, mapping to underlying hardware, and the interactions that concurrency can have with other well-established parts of the system, such as input/output, exceptions and the foreign-function C interface [20].

The Data Parallel Haskell (DPH) extension to GHC is one of these powerful abstractions, allowing a developer to write purely functional code that is executed by multiple processors, with most of the hard work carried out automatically by the system [29]. This includes vectorization of both data structures and functions, division of workload (even with nested, irregularly sized data structures), and assignment and execution on whatever hardware is available at runtime. The ability to work on more than just flat data structures, called nested data parallelism, is discussed first. We then describe the additional parallel array syntax, the critical vectorization/flattening operation, and the mapping to underlying hardware.

3.1 Nested Data Parallelism

Data parallelism is usually performed on flat data, which makes the division of labor between processing units straightforward, since the operation to be performed is identical across the data. Many applications could benefit from parallel operations over more complicated types, where data may be stored in sparse vectors or matrices, or may be represented by a recursive data type, such as a tree. In these situations, the problem is more easily described by nesting calls to parallel collection-oriented functions. Without that capability, program modularity suffers, and the programmer may have to resort to task-parallelism to express an otherwise fundamentally data-parallel algorithm [8]. One of the key features in DPH, and its inspiration, NESL [7], is the flattening operation which converts these irregular nested collections into traditional flat arrays for processing.

3.2 Parallel Array Comprehension Syntax

Haskell natively supports a form of syntactic sugar for building lists, called list comprehensions [21], which resembles "set-builder" notation from set theory. For example, we may specify all of the multiples of three in set-builder notation by equation 3.1.

$$\{3x : x \in \mathbf{N}\} \tag{3.1}$$

The equivalent notation in the Haskell language is shown below, along with an example of its output. This is, of course, an infinite list, so we use the `take` function to restrict output to the first ten examples. List comprehensions have the notion of order, unlike pure set-builder notation, and are widely used in Haskell.

```
> let k = [ 3*x | x <- [1..] ]
> take 10 k
[3,6,9,12,15,18,21,24,27,30]
```

List comprehensions are also used to define computations. The dot product of two lists is shown below, which utilizes an additional "parallel list" syntax. That is, elements are pulled from lists `a` and `b` at the same time, multiplied, and used to produce a list of elements which are then summed to give the final result.

```
> let as = [1,4,3]
> let bs = [2,1,3]
> sum [ a * b | a <- as | b <- bs ]
15
```

This style of expressing computations is the standard interface into the DPH system, through the very similar parallel array comprehension syntax.

The primary difference from the previously shown parallel list comprehensions is that DPH acts upon parallel arrays, which are denoted with the syntax `[:a:]`, where `a` is the type of element stored in the array. The function definition for dot product over arrays, defined with parallel array comprehensions is shown below. Notice that the only difference between the array and list syntax is the use of the parallel array brackets, and an explicit parallel summation function.

```
dotp as bs = sumP [: a * b | a <- as | b <- bs :]
```

This syntax illustrates the minimal interface required to perform data parallel computation. The DPH system desugars this syntax into standard function application, which then receives standard treatment, including optimizations and profiling support from the Haskell compiler. Much of the rest of DPH is implemented simply as library functions.

3.3 Flattening Transformation

DPH may be used with very complicated data. Nested arrays, user-defined types, product types, and recursive data types are all allowed and can be acted on in the same way as a flat list of floating point numbers. This is extremely powerful in allowing programmers to continue operating at a high level of abstraction, without sacrificing opportunities for parallelism. The key to this happening is the vectorization, or flattening operation that is performed. The goal of vectorization is to take potentially nested programs and data, and convert them to run on flat vectors of primitive types. Vectorization is a very complex operation, so we will only describe three important aspects, conversion from nested to flat arrays, the representation of user-defined types as unboxed arrays, and the construction of vectorized functions.

3.3.1 Flattening Nested Arrays

As an example of how nested arrays can be transformed into something flat again, we take a ragged matrix, with varying numbers of elements per row, and turn it into an array that can be processed with standard data parallel algorithms.

In code prior to vectorization, ragged arrays are simply parallel arrays of parallel arrays, with the type `[:[:a:]:]`. A sensible flat representation would be to simply stack the individual rows into an array, one after another, and keep track of their individual starting and ending positions. Indeed, this is precisely how the vectorization operation proceeds [29]. Once the desugaring of parallel array notation has occurred, we have standard Haskell types, and our ragged matrix has the type `PA (PA a)`, or a parallel array containing parallel arrays of elements of type `a`. The concrete representation of the matrix is a single flat array, along with what DPH calls a *segment descriptor*, which is a structure recording the lengths of the nested arrays, their starting indices, and how many data elements the segment descriptor contains. This type, which contains no more nesting, is referred to as being "unlifted". The final type is therefore a product of the flat array, and its segment descriptor.

3.3.2 Unboxing User-Defined Types

In most instances, arrays of user-defined types in Haskell are represented as arrays of thunks, which when evaluated, yield the requested data. This is part of how Haskell implements laziness, but it has serious negative effects for data parallel computations. One problem is that it destroys locality of data. A significant amount of work is done in the flattening transformation to arrange data in a flat vector. Having to then dereference and evaluate each array element would be extremely expensive. While main memory is often thought of as "random access", there are significant performance gains possible from fetching items in order from memory, so that cache lines and spatial locality are used effectively [16].

The alternative to using thunks, is to store data "unboxed", without the overhead required to support lazy evaluation. This is analogous to the difference between storing structures as array elements directly in C, and having an array of *pointers* to structs. In order to achieve good performance, the representation of types, including user-defined types, must be controlled. This is accomplished through a feature of Haskell's type system called associated types [9], which allow varying data representations. A representation can be specified for parallel arrays specifically, enabling primitive types like `Integer` and `Double` to be raw byte arrays. Pair types like `(a,b)` are represented as pairs of parallel arrays, showing the flexibility of associated types, since this is essentially the inverse of how they are stored by default (arrays of thunks to pairs).

3.3.3 Function Lifting

Now that we have some idea how data representation works for parallel arrays, we need to understand how work is performed on these structures. As an example, we consider applying the sigmoid function across a parallel array of node outputs. We have defined sigmoid in standard Haskell below.

```
sigmoid :: Double -> Double
sigmoid t = 1 / (1 + (exp (negate t)))
```

Function lifting refers to converting standard "unlifted" functions to work in the context of parallel arrays. It is analogous to the lifting operation that occurs with monads [25], which is used to transform an existing function into one that operates in a different ("lifted") context. Say we had a list, which is a monad, and wanted to apply the sigmoid operation to every element. We have a version of sigmoid, but it only works on a single number. One method for achieving this is to produce a new version of sigmoid, which works in the List monad (as well as any other monad type), by use of the `liftM` function, which is demonstrated below.

```
> sigmoid 0.2
0.549833997312478
> let a = [0.2, 0.3, -0.4]
> let sigmoidM = liftM sigmoid
> sigmoidM a
[0.549833997312478, 0.574442516811659, 0.401312339887548]
```

For vectorization, we actually require three functions, the standard scalar version, a lifted version, and the combination of these is used to form the vectorized version [29]. The lifted version of a function is then used in any other vectorization operations. Applying the lifting rules, we can construct a lifted sigmoid$_L$. Constants are replaced by collections of the same (large enough to match the size of the input), internal function calls are replaced by lifted variants, and parameters are left alone.

Our lifted sigmoid might look like the following, where subscripted $_L$'s show the replacement of a function with its lifted version.

```
sigmoid ::  [:Double:]  -> [:Double:]
sigmoid t = n /_L (n +_L (exp_L (negate_L t)))
  where
    n = (replicateP (lengthP t) 1}
```

At the core of DPH are implementations for primitive functions, like lifted addition ($+_L$) and the lifted exponential function (`exp`$_L$). There are currently two implementations, `seq` for purely sequential operation, and `par` for multi-core environments. Either of these modules can be included when the program is linked, giving us the ability to choose a sequential or parallel backend.

3.4 Mapping to Hardware

Finally, we look at how the flattened data representation, and vectorized functions work together along with the Haskell run-time concurrency system, to provide speedups for data parallel operations. Prior to DPH, the GHC compiler contained a mature parallel runtime [26], with support for explicit and semi-explicit parallelism. DPH puts this infrastructure to work, so it is only concerned with mapping to the relatively high-level runtime offered through GHC.

For a programmer, parallelism can be exploited by simply using the `par` and `pseq` functions to mark computations as candidates for running concurrently. These computations are called “sparks”, and are placed in pools that idle physical processors are able to work on. Meanwhile, idle threads execute these sparks as they become free. Keeping a group of system threads alive is more efficient than continuously creating threads for new work. When a user runs a program, they are able to specify by a run-time flag how many hardware threads (referred to as “capabilities”) should be allocated to the program (including the option of having this determined automatically).

The actual mapping from parallel structures to individual sequential cores is done through a mechanism called *Distributed Types* [11] [22]. These represent at the type level the distinction between parallel and sequential operations, as well as the necessary synchronization that is required to distribute work. Synchronization is achieved in the creation and destruction of distributed types, primarily through the `splitD` and `joinD` functions. Splitting takes an array and divides it into chunks, evenly distributed across the available members of a group of threads called a “gang”. This can then be operated on sequentially by each gang member. Joining happens after the parallel array has been split, and work done on each chunk, and serves as a coordination mechanism for each of the gang threads. Finally, joining returns a parallel array, that can be again split and worked on for the next operation. One of the important optimizations that occurs in this process is automatic removal, or “fusion”, of unnecessary split/join pairs [29].

Chapter 4

Implementation

Data Parallel Haskell has the potential to be used almost transparently by the user. Parallel arrays are just another type of data, with constructors for creating new instances, and functions that operate upon the data. In practice however, the system is still quite experimental. While it has been available as part of stable feature releases, going back to version 6.8 of GHC in late 2007, it is currently still a "technology preview", with the main goals of encouraging experimentation and early feedback. In order to use a current implementation, with significant performance and stability fixes, the latest development version of the GHC compiler must be acquired and built from source.

4.1 Sequential Implementation

In the course of evaluating both neural networks and Haskell, a small sequential neural network library was built, capable of reading input, using backpropagation to learn from examples, and performing classification on a test set, as well as evaluating its own accuracy. This implementation was tested using several datasets from the UCI Machine Learning Repository [5]. Results from training several networks of varying sizes against the UCI letter recognition data set are shown in Figure 4.1. Performing these experiments were important in order for us to gain insight into how many nodes were required for good results. Clearly, a problem that requires more nodes will have increased potential for parallel speedup, as any implementation overhead becomes dwarfed by the actual computation.

4.2 Interface Limitations

Parallel array comprehension syntax, automatic vectorization of functions and data structures, and tight integration with the existing Haskell thread system are the high points of what DPH provides the programmer in the current implementation. There are some pitfalls however, which make using this style of parallelism difficult. The first, is the requirement that all vectorized code reside in dedicated modules. The next limitation we discuss is the cost of marshalling data across the boundary of vectorized and unvectorized modules, which we find to be significant. Finally, we examine the treatment of matrices, which must be managed as nested data structures.

4.2.1 Module Vectorization

Any code that performs operations on parallel arrays needs to be vectorized, and vectorization is currently a whole-module transformation. That is, GHC cannot selectively choose to vectorize only those functions which are used in parallel array context, it must act on the entire module in which those functions reside. This is an improvement over languages like NESL [7], which require whole-program transformation, but is not ideal. Improvements in this area appear to be on their way, Chakravarty et al. have a strategy for performing "partial vectorization", even down to the sub-expression level [10]. This is what currently allows vectorized and non-vectorized modules to co-exist. Increasing the granularity and automation of vectorization will be a significant usability improvement in the library.

4.2.2 Marshalling Overhead

Any data that will be worked on in parallel first needs to be in a supported container. The cost involved in this operation, if done frequently, can dominate execution time. A minor change to our sequential program to invoke the parallel dot product operation during the feed-forward stage yielded significantly worse performance than simply running sequentially. Profiling results showed that most of the program execution time was spent creating `Gang` objects (groups of threads). Moreover, adding additional processing cores to the system actually made performance worse, meaning the cost of gang creation increases with additional hardware threads to be utilized. Investigation showed that gang creation spawns new OS threads every time work is requested from vectorized code, one thread per capability (physical core requested at runtime). Since thread creation costs are significantly greater than the relatively small matrix multiplication operation cost, we are not able to take advantage of the division of workload between cores.

In order to show real gains, it is imperative that we minimize the number of times we create parallel arrays from scratch, since this leads to thread creation in order to handle distributed computations. Otherwise, we spend more time setting up the system than doing real work. Comments in the source code acknowledge this is a “hack”, and we feel that decreasing gang creation costs through use of a more permanent thread pool could make the entire system far more usable, especially for small tasks. This is the same optimization currently used for assigning “spark” computations to threads.

4.2.3 Matrices

Support for matrices is critical for many algorithms that would typically be used in a data parallel context. Both the feed-forward and backpropagation algorithms are defined in terms of matrix operations. Unfortunately, there is currently very poor support for creating matrices from non-vectorized code. Converting from flat structures (both arrays and lists) is supported natively, but since matrices are treated as “nested” data, there are more restrictions. Creating matrices requires understanding how DPH represents nested structures internally, through the use of segment descriptors, which provide indexes into flattened arrays. We were able to create matrices for representing connection weights, doing so required being familiar with how nested data is represented as flat structures, and constructing the flat version ourselves.

4.3 Strategy

There are two main components we discuss for a DPH implementation of a neural network. The representation of the neural network structure, and how we create vectorized implementations of data parallel algorithms to operate on the network structure.

4.3.1 Network Representation

The choice of neural network representation is crucial for achieving good performance, but we also want something that is easily extensible. We are able to use standard Haskell primitive types, records, and tuples as elements of parallel arrays, and maintain sensible representations in memory. For example, we know that for any given node, we will be maintaining its output value, and it may be useful to store its error as well, for use in updating connection weights during backpropagation. Using a record type is ideal, since it allows us to extend the information contained in a node, without affecting any existing code. However, we do not want to jeopardize the efficient layout of node outputs by using anything other than a byte array. Due to the vectorization procedure, we can have it both ways. The following definition is high level and extensible, but will be implemented as two separate arrays of doubles, making indexing and data parallel operations efficient. In standard non-vectorized code, an array of `Node` types would instead be represented by an array of thunks/pointers to the actual storage site, requiring an additional level of indirection, and greatly increasing the costs to access and manipulate data in a uniform way.

```
data Node = Node {output :: Double, error :: Double}
```

With nodes defined, we can now think about layers of the network as parallel arrays of nodes.

```
type Layer = [:Node:]
```

The connection weights from one layer to another are straightforward to define as a matrix.

```
type FullConnection = [:[:Double:]:]
```

Layers that are not fully connected could be represented with a sparse matrix instead, which takes greater advantage of nested data parallelism, since some rows will have more data (and require more computation) than others. Sparse matrix multiplication is one of the common examples used with DPH, and we reuse the definition given in its status report [11]. Each row is now a vector of index/value pairs. The status report gives a succinct definition of dot product on this sparse structure as well.

```
type SparseVector = [:(Int,Double):]
type PartialConnection = [:SparseVector:]
```

4.3.2 Vectorized Function

We require vectorized versions of the feed-forward and backpropagation algorithm. These are very similar, so we only show the feed-forward version here. As mentioned previously, vectorization is enabled by module, so we must keep all vectorized components together. Marshalling of parallel arrays is required so that external functions can utilize data parallel operations. This is accomplished by converting flat `PArray a` structures into the parallel form `[:a:]` using the low-level marshalling functions `toPArrayP`, `fromPArrayP`, and `fromNestedPArrayP`. As shown here, the `feedForward` function is only a wrapper that allows data to pass into the vectorized module, which we explicitly disable from any inlining optimizations.

```
feedForward :: PArray Double
            -> PArray (PArray Double)
            -> PArray Double
{-# NOINLINE feedForward #-}
feedForward i w =
  toPArrayP (feedForward'
                (fromPArrayP i)
                (fromNestedPArrayP w))
```

Working from the bottom up, we know we will need an activation function, so we define the binary sigmoid as a scalar function. Vectorization will lift this into something that works on arrays, automatically.

```
sigmoid :: Double -> Double
sigmoid t = 1 / (1 + (exp (negate t)))
```

For clarity, we separate the function that produces an actual node's activation value from the full layer's feed-forward step. This function's type indicates that it takes two vectors, and produces a single number, the activation value of a node. The first argument is the vector of input node values, and the second is a vector of the weights connecting each of those inputs to the node we are currently considering. The `zipWithP` function multiplies each input node value with the corresponding weight, in parallel. The result of that is passed to `sumP`, which performs a summation on all the products, to form the input to our node. Finally, we use the activation function we just defined to squash the node's input into the output range $(0, 1)$.

```
activation :: [:Double:] -> [:Double:] -> Double
activation i c = sigmoid (sumP (zipWithP (*) i c))
```

We are able to compute the entire layer at the same time, so the `feedForward'` function uses a parallel array comprehension to perform the `activation` function using every row of the connection weight matrix. The final result is a vector of output values for the layer, calculated in parallel.

```
feedForward' :: [:Double:] -> [:[:Double:]:] -> [:Double:]
feedForward' i w = [: activation i c | c <- w :]
```

4.4 Test Environment

Testing was conducted on an Apple Mac Pro, with two quad-core Xeon 2.26GHz Xeon 5520 processors, and 6GB of memory. Memory was arranged as a single 1GB module for each of the six memory channels (three per processor). Each physical processor core is exposed as two processors to the operating system due to hyperthreading, the total number of concurrent threads that can execute is 16. One complexity of testing on this platform is the "turbo mode" feature, which dynamically varies the clock speed of the processing cores based on load. A die not using all of its cores is able to disable some, and increase the power to those remaining. This can result in a small but measurable increase to performance for applications utilizing less than the full number of physical cores. In practice, this means that system clock speed is likely to be 2.53Ghz with one to two cores active, 2.4Ghz with three to four, and 2.26Ghz with five or more active. It may also be part of the reason that performance at low thread counts had a relatively high variance, as shown in our benchmarks. We also speculate that it may contribute to the localized dip in performance that was consistently seen after surpassing 5 threads. This was not corrected for in our measurements, since it is simply the reality of this platform, and is likely to be more commonplace in the future.

Final testing and benchmarked code was compiled using the Glasgow Haskell Compiler, built from the latest development source obtained on July 14, 2009. The `dph-par` array library was used in all cases to produce multi-threaded code. Parallel garbage collection was enabled by default. The compiler flags used were those recommended to give the best performance for vectorized DPH code: `-threaded`, `-fcpr-off`, `-funfolding-use-threshold30`, `-funbox-strict-fields`, and `-Odph`.

4.5 Results

We examined our data parallel feed-forward implementation by running 500 iterations between two fully-connected layers with the same number of nodes. We varied the number of processing elements from one to 16, the full range that was possible in our test environment. We also varied the number of nodes in each layer, to determine at what point initialization costs are overcome by the extra processing capacity.

Previous results in the DPH status report showed significant speedups on a similar problem, sparse matrix/vector multiplication, where the smallest matrix used had one million non-zero elements [11]. This roughly corresponds to two layers of one thousand nodes each in our application. Absolute run time is shown for a network this size in Figure 4.5. Performance improves significantly with each additional processor up to the fourth, with all subsequent additions having mixed results.

We validated that with large enough data sets, we can get very encouraging results. With 7,000 nodes in each layer, we have 49 million weights that must be calculated, and here we see an advantage even up to 16 threads. Figure 4.5 illustrates the run time being reduced from 1422ms with a single thread, to 222ms with 16 threads, a 6.4x speedup. As we consistently observed, there is a great deal more variation when using between 3 and 7 processors.

However, as was discussed previously, smaller workloads do not see the same types of gains. In fact, performance can decrease with every additional processor. Figure 4.5 shows the results of using two 100-node layers, which demonstrates how performance can be strictly decreasing as more processors are used. In between that example and the positive results for large workloads, we find that there is a range of workloads where adding a small number of processors is beneficial, but performance begins to decrease again after too many are involved. On this test platform, for this problem, the cutoff was on the order of 10 milliseconds. That is, when an individual feed-forward operation (determining the output of the entire next layer) took greater than 10 milliseconds, we started to see some improvement when using all available processors. At 300 nodes (6ms), we saw the first improvement when moving from one to two threads, while at 500 nodes (11ms), we began to see a slight improvement when using all 16 threads.

We can examine the effect of changing the workload size by plotting increasing sizes along with their speedup per additional processor. This is shown in Figure 4.5, and makes it clear how smaller workloads are hurt by additional processors, medium-sized workloads only achieve speedups for small thread counts, and more predictable speedups are achievable for large workloads.

One consequence of note is that there may be a need for a more intelligent system for determining how many processors to use for a distributed task. Currently, only one setting can be given at program initialization time, which is not ideal for all machines or problem sizes. For example, we would only want to use a single thread for calculating layer outputs that involved a small number of connections, but might want several threads for a step that had millions of connections. Currently, we must optimize either for the large or small case, at the expense of the other, or complicate our code with multiple implementations of the same function (vectorized, and scalar).

Figure 4.1: Training results for UCI Letter Recognition, varying single-layer network sizes

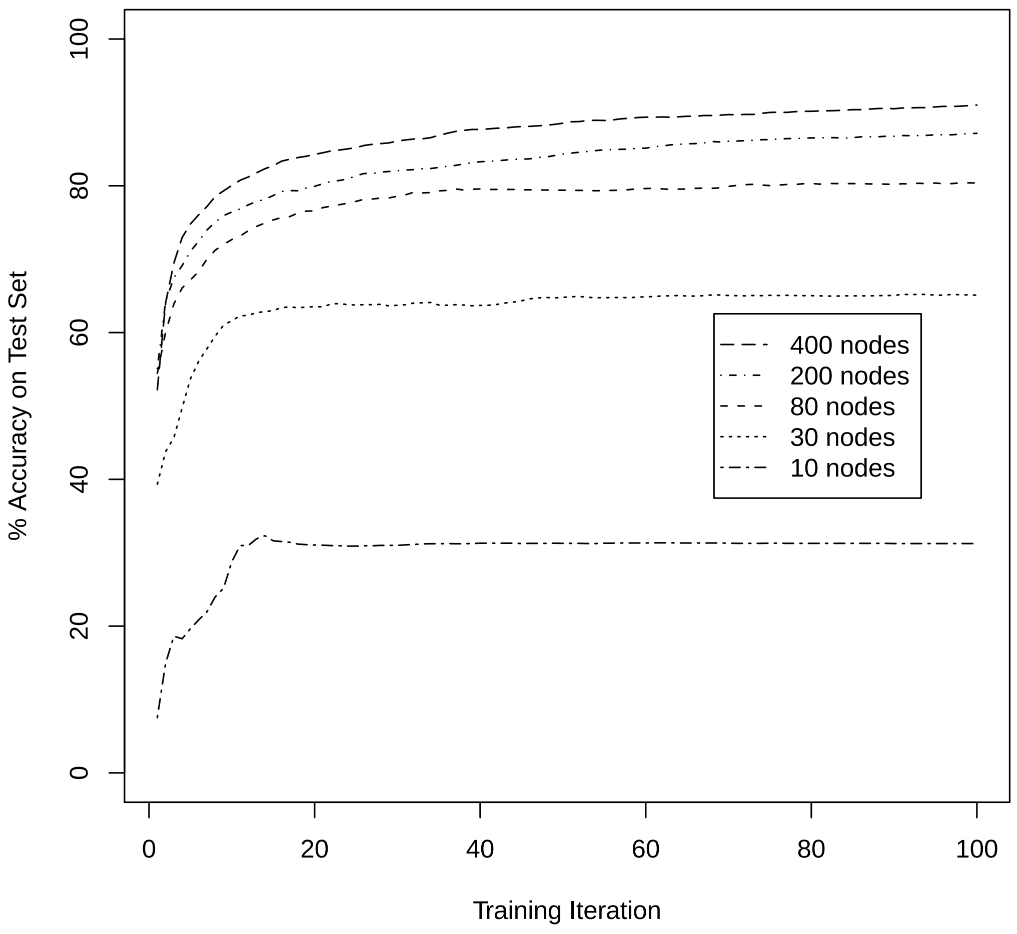


Figure 4.2: Run time of single feed-forward iteration for two 1000-node layers

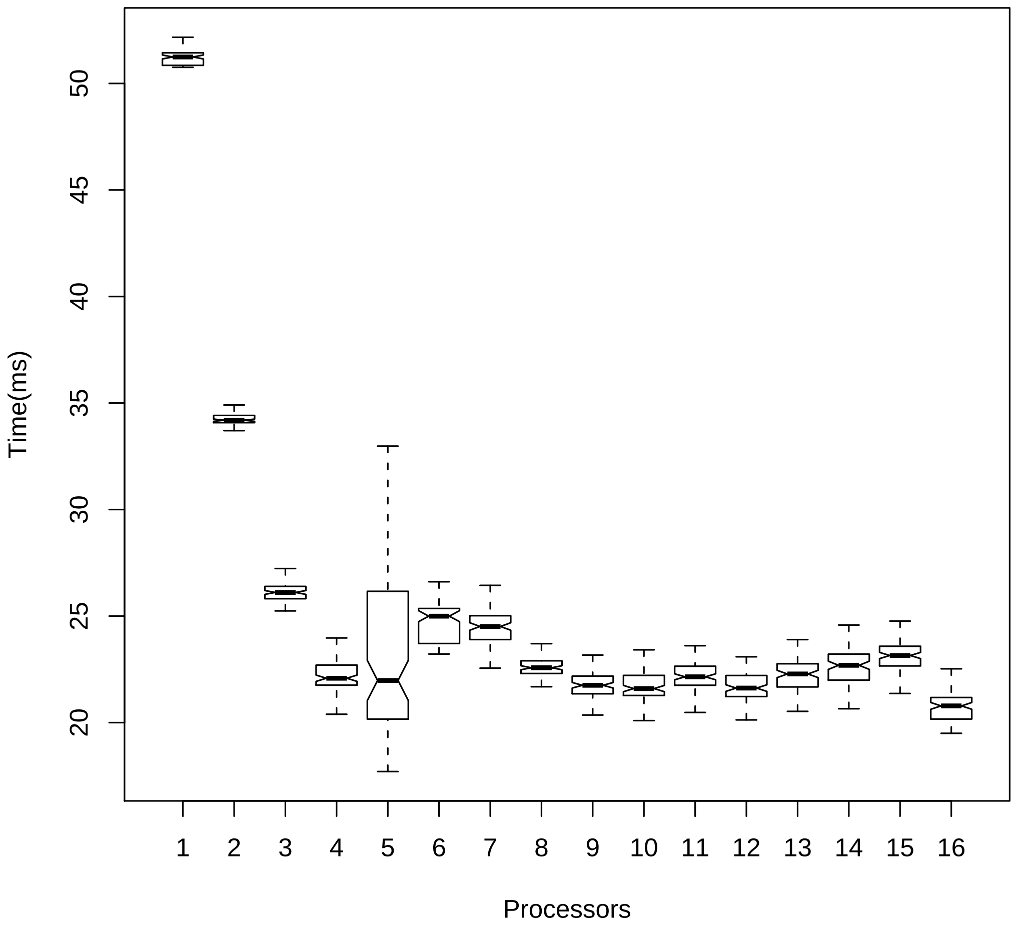


Figure 4.3: Run time of single feed-forward iteration for two 7000-node layers

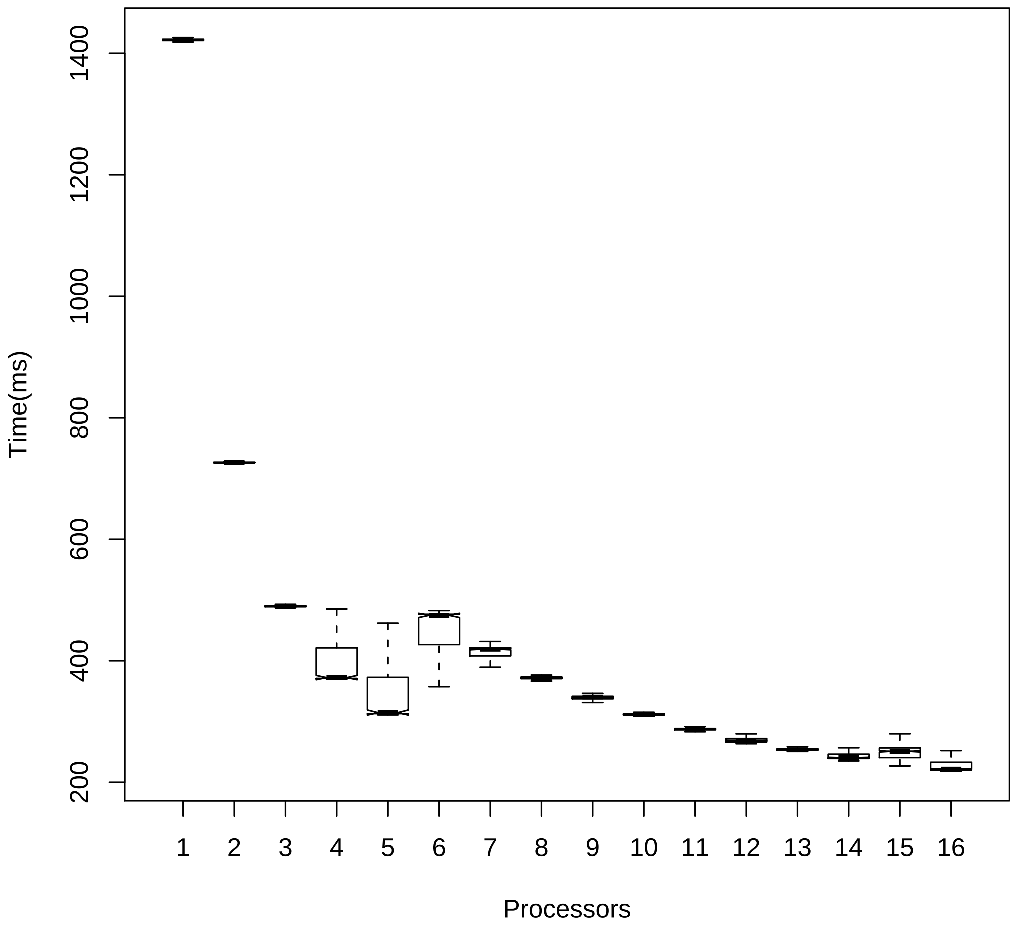


Figure 4.4: Relative speedup for varying workloads

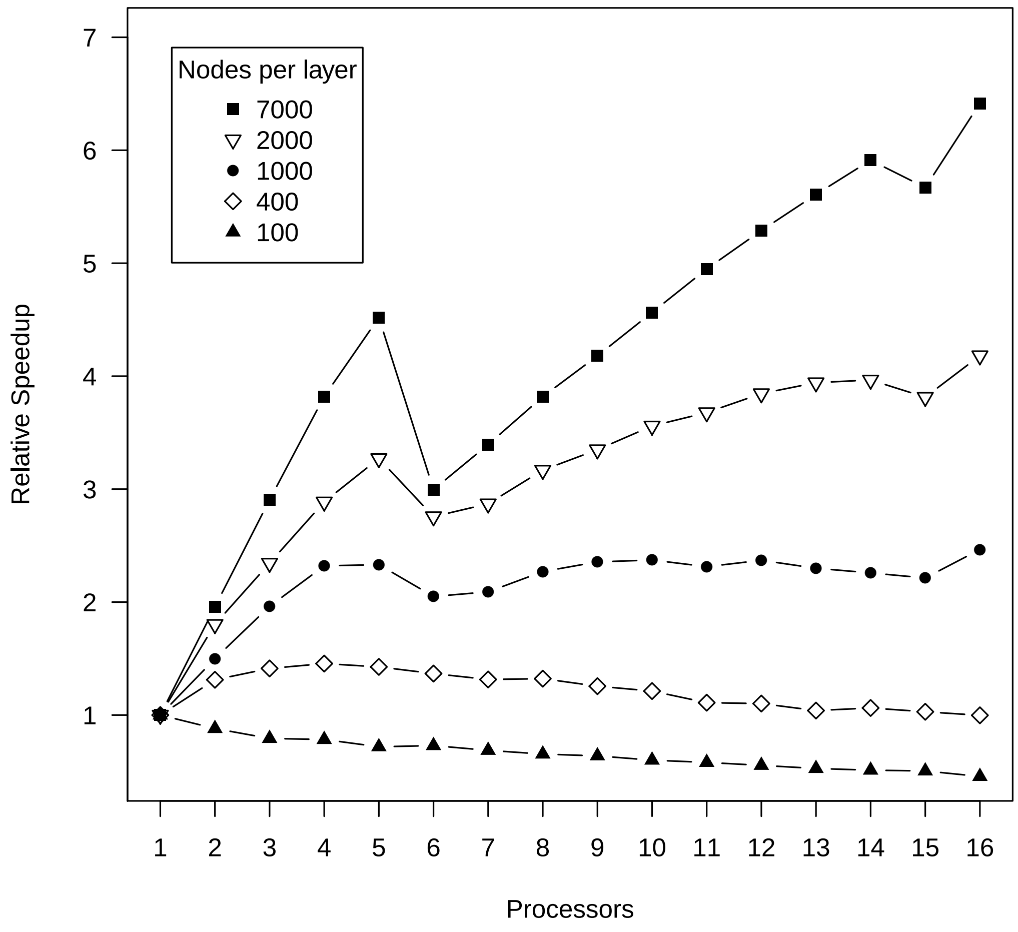
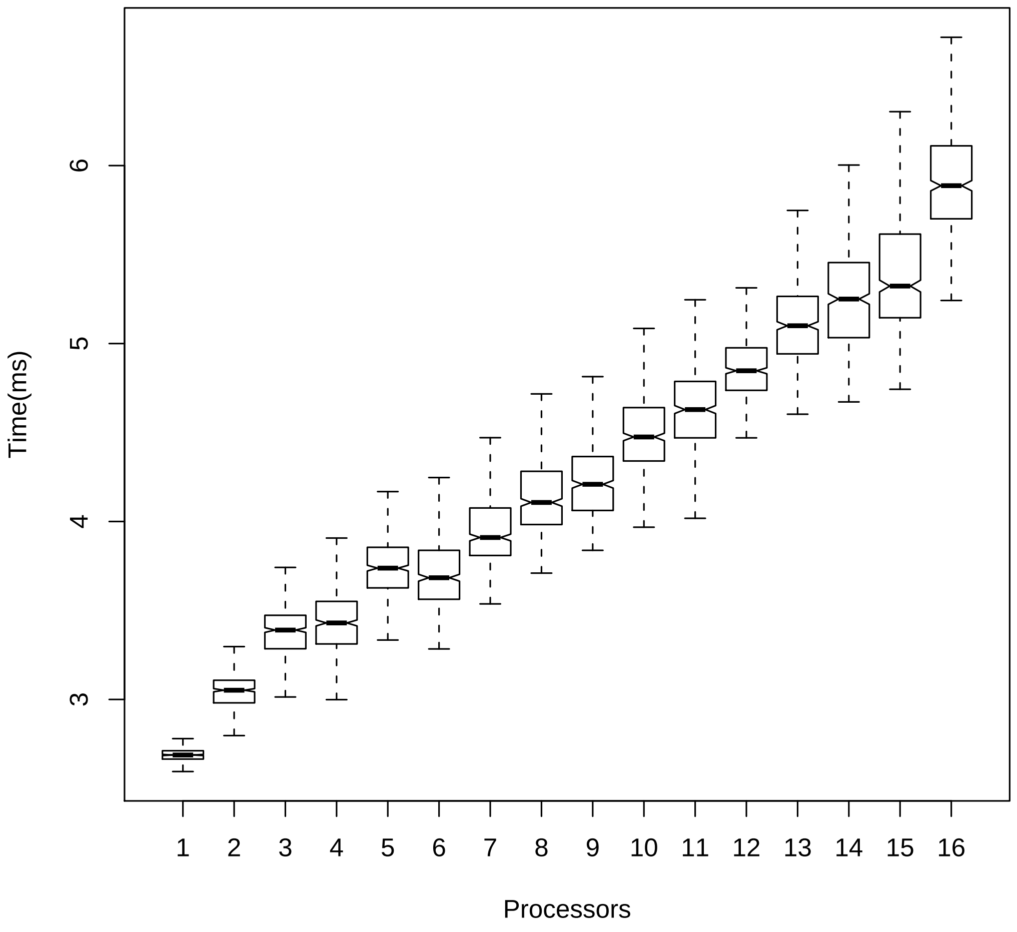


Figure 4.5: Run time of single feed-forward iteration for two 100-node layers



Chapter 5

Conclusions

We looked at a standard tool for performing automatic classification, feed-forward neural networks using backpropagation for learning, and examined several different methods for taking advantage of parallelism. These included both task and data parallel techniques. The functional programming language Haskell, and its Data Parallel Haskell library were analyzed for application to the problem of parallelizing neural network learning and classification.

We found many desirable features from the DPH library that would allow us to program simultaneously at a high level of abstraction, but without sacrificing some critical performance-oriented features such as efficient data representation. The vectorization transformation provided a powerful abstraction for describing parallel array operations in a very readable format.

A strategy for implementing a data parallel neural network was described, demonstrating how different aspects of DPH are used. Unfortunately, we found that small tasks are currently unsuitable for use with DPH, due to the static notion of capabilities and how thread gangs are initialized. Since operating system thread creation time may be greater than our task, any increase in the number of threads results in lower overall performance. Since system capabilities are static, mixing tasks that can take advantage of only a small number of threads, and those that can use many, may optimize one task at the expense of the other.

For larger network sizes, with millions of connections, we demonstrated significant speedups, repeating the positive results seen in other applications with this tool. Our best result was speedup of 6.4 over a single-threaded execution, on a machine with 8 physical processor cores, capable of running 16 concurrent threads.

Finally, we note that going forward, the startup costs of the DPH system should be more carefully considered to broaden the applicability of this tool to smaller problems. Operating the neural network described here has significant opportunities for parallelism, but this takes place in a large number of small steps, making startup overhead critical.

The Data Parallel Haskell compiler extensions and library are still incomplete, rapidly changing, and experimental. However, the abstractions are powerful, and performance gains are real. We look forward to the necessary usability improvements that will enable developers to use this more seamlessly in their applications.

Bibliography

[1] M. Ahmad and F.M.A. Salam. Dynamic learning using exponential energy functions. In *Neural Networks, 1992. IJCNN., International Joint Conference on*, volume 2, pages 121–126 vol.2, Jun 1992.

[2] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. Lapack: a portable linear algebra library for high-performance computers. In *Supercomputing '90: Proceedings of the 1990 conference on Supercomputing*, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[3] D. Andrade, B.B. Fraguela, J. Brodman, and D. Padua. Task-parallel versus data-parallel library-based programming in multicore systems. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 101–110, Feb. 2009.

[4] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[5] A. Asuncion and D.J. Newman. UCI machine learning repository, 2007.

[6] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[7] Guy E. Blelloch. NESL: A nested data-parallel language. Technical report, Pittsburgh, PA, USA, 1992.

[8] Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *J. Parallel Distrib. Comput.*, 8(2):119–134, 1990.

[9] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. *SIGPLAN Not.*, 40(1):1–13, 2005.

[10] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Gabriele Keller. Partial vectorisation of haskell programs. In *DAMP '08: Proceedings of the 2008 workshop on Declarative aspects of multicore programming*, New York, NY, USA, 2008. ACM.

[11] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *DAMP '07: Proceedings of the 2007 workshop on Declarative*

aspects of multicore programming, pages 10–18, New York, NY, USA, 2007. ACM.

[12] Laurene Fausett. *Fundamentals of neural networks: architectures, algorithms, and applications.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[13] Thomas Fontaine. Grad-cm2: A data-parallel connectionist network simulator. Technical report, 1992.

[14] Paolo Frasconi, Marco Gori, and Giovanni Soda. Daphne: Data parallelism neural network simulator, 1992.

[15] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[16] John L. Hennessy and David A. Patterson. *Computer organization and design (2nd ed.): the hardware/software interface.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[17] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[18] J. Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, 1989.

[19] Robert A. Jacobs. Increased rates of convergence through learning rate adaptation. Technical report, Amherst, MA, USA, 1987.

[20] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In *Engineering theories of software construction*, pages 47–96. Press, 2002.

[21] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[22] Gabriele Keller and Manuel M. T. Chakravarty. On the distributed implementation of aggregate data structures by program transformation. In *In Proceedings of the 4th IPPS/SDP International Workshop on High-Level Parallel Programming Models and Supportive Environments,IPPS/SDP99*, pages 108–122. Springer-Verlag, 1999.

[23] B. Kryzhanovsky, B. Magomedov, and A. Fonarev. On the probability of finding local minima in optimization problems. In *Neural Networks, 2006. IJCNN '06. International Joint Conference on*, pages 3243–3248, 0-0 2006.

[24] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.

[25] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *In Proceedings of the 22nd ACM Symposium on Principles of Programming Languages. ACMPress*, 1995.

[26] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore haskell. In *ICFP '09: Proceeding of the 14th ACM SIGPLAN international conference on Functional programming*, August 2009.

[27] Warren Mcculloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 5(4):115–133, December 1943.

[28] Manavendra Misra. Parallel environments for implementing neural networks. *Neural Computing Survey*, 1:48–60, 1997.

[29] Simon Peyton Jones. Harnessing the multicores: Nested data parallelism in haskell. In *APLAS '08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, pages 138–138, Berlin, Heidelberg, 2008. Springer-Verlag.

[30] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–308, New York, NY, USA, 1996. ACM.

[31] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. pages 318–362, 1986.

[32] L.F.A. Wessels and E. Barnard. Avoiding false local minima by proper initialization of connections. *Neural Networks, IEEE Transactions on*, 3(6):899–905, Nov 1992.

Vita

Gregory Lynn Heartsfield was born in Arlington, Texas on 3 March 1983, the son of Gary Heartsfield and Teresa Heartsfield. He received the Bachelor of Science degree in Computer Science and Software Engineering from the University of Texas at Dallas in 2004. He was hired as a computer programmer at Bell Helicopter Textron upon graduation, where he worked on the design and implementation of product data management, identity management, and access control systems. He was accepted and started graduate studies at the University of Texas in their Software Engineering program in 2007. Concurrent with those studies, he continued his career at Bell Helicopter Textron.

Permanent address: 1309 Cummings Drive
Bedford, Texas 76021

This report was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.