Parallelization of Star Identification Neural Networks

Stephan Boettcher
Department of Computer Science
Georgia Technical Institute
sboettcher3@gatech.edu

1 Introduction

Small satellite attitude determination and controls systems are limited by size and power consumption, yet the ability to determine attitude and location is critical. Many of these attitude determination systems use horizon detection systems, single star tracking cameras, or passive magnetic alignment [1]. While the exact requirements and constraints of a satellite system are mission and hardware dependent, most satellites must have some rudimentary form of attitude determination.

Given the nature of our current budget constrained era, the need for smaller, faster, lighter, cheaper systems is constantly growing [2]. Requirements for new systems that are able to replace multiple older systems are becoming more common. New systems must be robust, fault tolerant, low cost, flexible and recoverable in the event of a system failure. Many conventional multi-star identification systems are not near-real time in nature, nor do they have the speed or system overhead of the system described. Current star tracking systems may use *a priori* knowledge of spacecraft position and attitude to provide quicker solutions [3]. The system described does not require any prior knowledge of the orientation of the spacecraft.

1.1 Motivation

Miniaturized satellite systems are becoming increasingly popular as an alternative to large, costly systems. CubeSats are a type of miniaturized satellites for space research. These systems typically have a volume of 1 to 3 liters and weigh approximately 1 to 4 kg. CubeSats also typically use commercial-off-the-shelf parts to keep the system cost down. The size and weight of these satellite systems allow them to piggy back off of other launches and further cut mission cost. In 2010, the University of Michigan launched the first National Science Foundation sponsored mission, the Radio Aurora Explorer (RAX). RAX-1 was a demonstration system capable of making bi-static radar measurements that had previously been performed on larger satellite systems. In order to achieve this success the CubeSat required attitude knowledge within 5 degrees of truth [4]. RAX utilized rate gyros, magnetometers, a course sun sensor, GPS and an extended Kalman filter to achieve this accuracy. However, with a star identification by a neural network system, many of these other systems could have been replaced by a small CMOS camera fixed to the top of the satellite and a GPS receiver.

2 System overview

This project relies on the uniqueness of a ”starburst pattern” or a vector of relative distances between a base star and it neighboring stars within a set field of view. An example of this starburst pattern is seen in Figure 1 below. A star in the belt of the constellation Orion was used as the center point of the pattern. The data set used to calculate this starburst pattern was the Yale Bright Star Catalog [5]; a database of all stars of magnitude 7 or greater. The catalogue was input in to MySQL and the relative distances between stars for a given field of view was calculated. This relative distance database was then used later in the system. The star identification system consists of 3 stages: a star

generation and image processing system, an neural network and the post-processing stage. These three stages work together to generate the final star-id output.

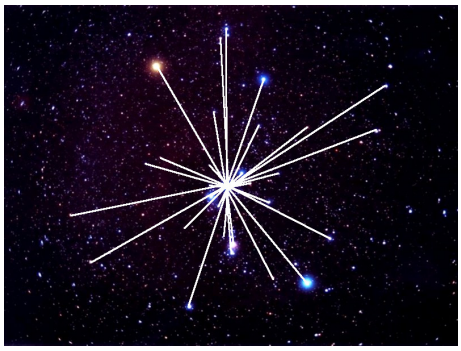

Figure 1: Starburst pattern of the constellation Orion

In this system, the neural network accepts the input starburst from a preprocessor and begins the identification process. During the training stage, the output given by the network is back propagated through the network to optimize the results. As a proof of concept, the neural network was trained on the total star-field to stress the network and make parallelization techniques more apparent. However, for practical applications a number of neural networks covering different features in the star field would be used. Figure 2 gives a graphical representation of this project. The first picture shows the constellation Orion. The middle star on Orion's Belt was chosen in the second frame to be the star-burst center. The different lines on the pattern represent the different distances from the center star to its various related stars. The third frame gives a graphical representation of neural network used to output the final star id.

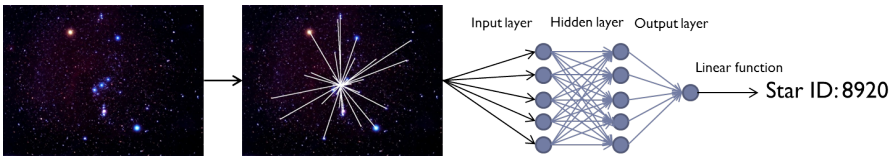

Figure 2: Full process for star identification

2.1 Neural Network Design

The artificial neural network is based on the nervous system design. The neural network used utilized three layers: an input layer, a hidden layer and an output layer. The input given to the neural network propagates through each layer of a predetermined number of nodes. Each node in a given layer are connected to every other node in the next layer. This can be seen in the neural network in Figure 3 below. The nodes in the input layer are connected to all of the nodes in the hidden layer. Each connection is weighted to control the effect of each node.

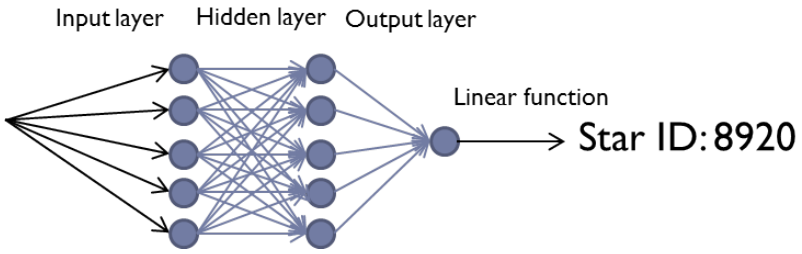

Figure 3: Star Identification Neural Network

This neural network uses back-propagation to minimizing the error. The output error is first calculated. Next, the output error is back-propagated for each hidden unit. The network weight for each connection is updated. This method uses gradient descent over the entire network weight matrix to converge the network. However, gradient descent is often slow and has a tendency to get stuck at local minima. One method of combating local minima is adding a weight momentum term. This weight momentum term helps to keep the method converging and bypass the minima.

2.2 System Hardware

The neural network was implemented on a 2011 era iMac with a 3.4 GHz, 4 core, Intel core i7 processor, 8 GB of RAM, and an AMD Radeon HD 6970M graphics card. The code requires the use of the Eigen matrix and MySQL interface libraries, which are not present on the jinx clusters. The code also has long run times (15 minutes to load the training matrix alone for an approximately 8000 star neural net with 250 inputs), and large memory usage (training matrix $\geq$ 1.1 GB). Attempting to use the jinx nodes may have resulted in the code being a process hog and could have negatively impacted other projects. Unfortunately, this limited the amount of speedup the code could achieve and impacted the parallelization techniques used.

3 The Code

The process to parallelize the neural network and the accompanying training algorithm occurred with 5 distinct steps:
1. Analyze the original code and locate bottle necks
2. Remove all bottle necks possible without parallelization techniques
3. Add Cilk
4. Add Cilk reducers
5. Investigate further parallelization techniques Each step will be covered in the subsections below.

3.1 The Original Code

The original code consists of a training algorithm, the neural network algorithm and an overarching control algorithm. The pseudocode below gives a rough analysis of algorithms. The actual code can be found in the accompanying files under Original Code.

```
Procedure: Main
{
// For analysis purposes:
//γ = MySQL query cost, s= iterations, r=number of stars,
//n=input size and κ = MySQL row fetches
Initialize parameters     //Initialize the portion of the sky to train on,
                //the neural net parameters and number of iterations.
                // O(1) work and depth
Create the Neural Network // Initialize NN structs, fill the
                //matrices/vectors with random numbers. O(n^2)
                // node in size=node hidden size=n
                // node out size =1 usually
    {
    for i ← 0 to n do
       for j ← 0 to n do
          weight in matrix[i,j] ← rand(-.5,.5)
          momentum in matrix[i,j] ← rand(-.5,.5)
    for k ← 0 to n size do
        for p ← 0 to node out size do //node out size=1,
            weight out matrix[k,p] ← rand(-.5,.5)
            momentum out matrix[k,p] ← rand(-.5,.5)
        }
Begin Training algorithm
return 0;
}
```

```
Procedure: Training {
    Initialize the mysql connections // O(1)
    Create initial queries //O(1)
    Initialize training variables //O(1)
    Perform initial query // see discussion below for O
    while(x<s) //limit← training limit(s)
        // This whole loop is O(s*r*(n*(n^2)+n*κ+γ)
    {   x+=1;
        while(fetch row from db != NULL) // O(r*(n*(n^2)+n*κ+γ)
        {
            Create second query // O(1)
            perform query //O(γ)
            sum←0
            for(i←0 to n) // O(n*(n^2)+n*κ)
            {
              row2←fetch one row //O(κ)
              numrows←fetch number of rows in query
              if(numrows!=0)
              {
                  dist[i]← row2[0]
                  sum+=dist[i]
              }
              else
              {
                  sum← 0
              }
            }
            if(sum!=0)
            {
            results← UpdateNeuralNet(NN,dist)//O(n^2)
            results2←linear scaling(results)
            error←(results, starid)
            BackPropagate //O discussed below//O(n^2)
             if(result2 within stared ± 1)
                 {correct ← +1}
            }
            free mysql results
        }
        perform book keeping for performance analysis //O(1)
    }
    close mysql connections
}
Procedure: UpdateNeuralNet(NN, distance)     //O(n^2)
{
  if(distance!= n)
  {throw error}
  for(i←0 to n) // bigO(n)
  {
    NN.inVals[i]←distance[i]
  }
  for(j←0 to n){// bigO(n^2)
        temp ←0
        for(k←0 to n){
           temp←temp+NN.inVals[j]*NN.WeightIn[j,k]
        }
        NN.hiddenVals[j]=temp
  }
   for(p←0 to node out size){// bigO(n^2)
        temp ←0
        for(q←0 to n){
           temp←temp+NN.hiddenVals[p]*NN.WeightOut[p,q]
        }
        NN.outVals[p]=temp
```

```
  }
 return NN.outVals
}
Procedure: BackPropagate(NN, starid)
{
retval←0
for(i←0 to node out size){// bigO(1)
  outErr←NN.outVals
  deltaOut← dsig(outErr)*(starid−outErr)
  //dsig is the derivative sigmoid
  retval←retval+.5*(2^(starid−outErr))
  }
 for(j←0 to n){// bigO(n)
    temp,←0
    for(k←0 to node out size){//nodeout =1
        temp←deltaOut[k]*NN.weightOut[j,k]
        }
    deltahid[i]←dsig(NN.hiddenVals[p]*temp)
    }
for(p←0 to n){// bigO(n)
 temp,←0
 for(q←0 to node out size){
   temp←deltaOut*NN.hiddenVals[q]
   NN.weightOut[p,q]←NN.weightOut[p,q]+temp+NN.momentumOut[p,q]
   NN.momentumOut[p,q]←deltaOut*NN.hiddenVals[p]
   }
 }
for(i←0 to n){// bigO(n^2)
 temp,← 0
 for(j←0 to n){
   temp←NN.hiddenVals[j]*NN.inVals[i]
   NN.weightIN[i,j]←NN.weightIN[i,j]+temp+NN.momentumIN[p,q]
   NN.momentumIN[i,j]←NN.*NN.inVals[i]*deltahid[j]
   }
 }
 return retval
}
```

The code above, adapted from a previous neural network project, is not work optimal and suffers from a number of problems that greatly increase the overall runtime. A cursory analysis shows that the algorithm queries the MySQL database for the same set of data each iteration. This is a huge bottle neck and time killer as each query (γ) takes approximately .003 seconds to complete, as seen in Figure 4 below. Fortunately, fetching a row (κ) from a pervious query executes in only 1e-6 seconds. This fact can be used later to parallelize database calls by favoring row fetching over queries. With 8000 stars and thus 8000 queries in a single iteration, this database time adds up quickly. A look at the Update and Back propagation algorithms results in each executing in $\mathcal{O}(n^2)$ time. Thus, these two algorithms would greatly benefit from parallelization. The first step in this code process was to minimize database calls and load the training set into memory.

Average MySQL execution times	
Time to execute single mysql query	0.002944 sec
Time to execute mysql row fetch:	0.000001 sec

Figure 4: Average MySQL execution times for query/fetch operations

3.2 Modified Serial Code

The modified serial code consists of a training algorithm, the neural network algorithm and an overarching control algorithm. For brevity, the pseudocode below gives a rough analysis of algorithms

that changed from the section above. The actual code can be found in the accompanying files under Modified Serial.

```
Procedure: Training {
    Initialize the mysql connections // O(1)
    Create initial queries //O(1)
    Initialize training variables //O(1)
    Perform initial query // O(s*γ)
    //NEW CODE ————————————
    Create star flag array, training matrix, iteration counter=0
    while(fetch row from db != NULL) // O(r*γ*n*κ)
    {
        Create second query // O(1)
        perform query //O(γ*r)
        flag(iteration)=1
        for(i←0 to n) // O((n)κ)
            {
              row2←fetch one row //O(κ)
              numrows←fetch number of rows in query
              if(numrows!=0)
              {
                 load data into training matrix
              }
              else
              {
                 flag(iteration)=0
              }
            }
            iteration++
            free MySQL results
    }
 while(x<s) //limit← training limit (s)
 {
        // This whole loop is O(s*r*n^2)
            counter=0
            while(counter<r) // O(n^2*r)
            {
              if(flag(counter))
              {
            dist=training matrix row
            results← UpdateNeuralNet(NN, dist) //O(n^2)
            results2←linear scaling(results)
            error←(results, starid)
            BackPropagate //O(n^2)
             if(result2 within stared ± 1)
                 {correct ← +1}
            }
            free mysql results
        }
        perform book keeping for performance analysis //O(1)
    }
    close mysql connections
}
```

As can be seen in the section above, the final cost of loading the training matrix was $\mathcal{O}(r*\gamma*n*\kappa)$ and the total cost of the training algorithm is $\mathcal{O}(r*\gamma*n*\kappa)$ + $\mathcal{O}(s*r*n^2)$. The original code, the overall cost of the algorithm was $\mathcal{O}(s*r*(n*(n^2)+n*\kappa+\gamma)$. This reordering of the code and one time loading of the training matrix reduces the cost of the algorithm by a factor of s. This is extremely important give the high cost of the MySQL calls(γ) and row fetches (κ). In the next step, the database effects will be further hidden by implementing Cilk with the training set load.

3.3 Cilk

The Cilk code consists of a training algorithm, the neural network algorithm and an overarching control algorithm. For brevity, the pseudocode below gives a rough analysis of algorithms that changed from the section above.The actual code can be found in the accompanying files under Cilk.

```
//given p= number of processors
Create the Neural Network // O(1) for large P
// still have work=n^2, but for p=n, O(1).

        { cilk_for(i←0 to n)
                {cilk_for(j←0 to n)
                  { NN.weightIN , NN.momentumIN ← rand(-.5,.5)}
                  NN.inVals=NN.hiddenVals=1
                  }
        cilk_for(k←0 to n)
                {cilk_for(m←0 to n)
                  { NN.weightOUT , NN.momentumOUT← rand(-.5,.5)}
                  }
                 NN.outVals=1
        }

Procedure: Training {
//NEW TRAINING CODE
   cilk_for(count←0 to r) // O(γ*n*κ*(r/p))
    {
        Create second query // O(1)
        perform query //O(γ*r)
        flag(iteration)=1
        for(i←0 to n) // O((n)κ)
           {
```

```
              .
              .
              .
              }
    }
              // END NEW CODE IN TRAINING

Procedure: UpdateNeuralNet(NN, distance)     //O(n)
{
  if(distance!= n)
  {throw error}

  cilk_for(i←0 to n) // bigT(1)
  {
    NN.inVals[i]←distance[i]
  }
  cilk_for(j←0 to n){// bigO(n)
        temp ←0
        for(k←0 to n){
           temp←temp+NN.inVals[j]*NN.WeightIn[j,k]
        }
        NN.hiddenVals[j]=temp
  }
   cilk_for(p←0 to node out size){// bigT(n)
        temp ←0
        for(q←0 to n){
           temp←temp+NN.hiddenVals[p]*NN.WeightOut[p,q]
        }
        NN.outVals[p]=temp
  }
 return NN.outVals

}
Procedure: BackPropagate(NN, starid)
```

```
{
⋮
 clik_for(j←0 to n){// bigT(n)
    temp,←0
    for(k←0 to node out size){//nodeout =1
        temp←deltaOut[k]*NN.weightOut[j,k]
        }
    deltahid[i]←dsig(NN.hiddenVals[p]*temp)
    }

cilk_for(p←0 to n){// bigT(1)
        //because for this neural net, node out size−1
 temp,←0
 for(q←0 to node out size){
   temp←deltaOut*NN.hiddenVals[q]
   NN.weightOut[p,q]←NN.weightOut[p,q]+temp+NN.momentumOut[p,q]
   NN.momentumOut[p,q]←deltaOut*NN.hiddenVals[p]
   }

 }
cilk_for(i←0 to n){// bigT(n)
 temp,← 0
 for(j←0 to n){
   temp←NN.hiddenVals[j]*NN.inVals[i]
   NN.weightIN[i,j]←NN.weightIN[i,j]+temp+NN.momentumIN[p,q]
   NN.momentumIN[i,j]←NN.*NN.inVals[i]*deltahid[j]
   }
 }
 return retval
}
```

In this step, Cilk was applied to each possible for-loop, without reducers. For this section, with p processors = n values, the Update and Back-propagate code would take approximately $\Theta(n)$ time. In this scenario, the MySQL loading code would dominate the execution time. However, since the hardware was limited to 4 processors and 8 threads, the code execution time will be something closer to $\mathcal{O}(n/p)$ for the Update and Back-propagate codes.

3.4 Cilk Reducers

The Cilk code consists of a training algorithm, the neural network algorithm and an overarching control algorithm. For brevity, the pseudocode below gives a rough analysis of algorithms that changed from the section above.The actual code can be found in the accompanying files under Cilk with reducers.

```
Procedure: UpdateNeuralNet(NN, distance)      //O(n)
{
                                        ⋮
    cilk_for(j←0 to n){// bigO(1)
        cilk::Reducer ←temp(0)
        for(k←0 to n){
            temp←temp+NN.inVals[j]*NN.WeightIn[j,k]
        }
        NN.hiddenVals[j]=temp.get_value()
  }
   cilk_for(p←0 to node out size){// bigT(1)
        cilk::Reducer ←temp(0)
        for(q←0 to n){
            temp←temp+NN.hiddenVals[p]*NN.WeightOut[p,q]
        }
        NN.outVals[p]=temp.get_value()
  }
 return NN.outVals
```

```
}
Procedure: BackPropagate(NN, starid)
{
                                        ⋮
 clik_for(j←0 to n){// bigT(1)
    cilk::Reducer ←temp(0)
    for(k←0 to node out size){//nodeout =1
        temp←deltaOut[k]*NN.weightOut[j,k]
        }
    deltahid[i]←dsig(NN.hiddenVals[p]*temp.get_value())
    }
                                        ⋮

}
```

With the addition of reducers, the code is able to add extra parallelization to the code and achieve extra speedup. The reducers allow this section, with p processors = n values, to execute in approximately $\Omega(1)$ time. In this scenario, the MySQL loading code would dominate the execution time. However, since the hardware was limited to 4 processors and 8 threads, the code execution time will be something closer to $\mathcal{O}(n/p)$ for the Update and Back-propagate codes.

3.5 Other parallelization techniques

An attempt was made to integrate OpenCL into the program. Initial benchmarking tests were performed on a representative dataset to compare sequential, cilk and GPU programming results for the matrix sizes used by the neural network. The results of this initial benchmark were surprising. For the given hardware of the system, the utilizing the GPU was less efficient than using the Cilk. Further investigation into this reviled that due to the constantly changing contents of the matrices, the overhead from reloading the matrix into the GPU memory caused large enough delays that Cilk was able to outperform OpenCL. While this was an unexpected, repeatable outcome, it is most likely a result of the hardware architecture and size of the matrix. On a different architecture with a different data set, it would be expected that OpenCL would outperform Cilk. The benchmark code for this project is located in the "Other Parallelization" folder included with this document.

4 Experimental Results

The results of the parallelization effort proved to be a success. An overall performance increased of roughly 90 x the original code was achieved in this project. The code was modified in discreet steps to ensure that each code modification resulted in increased performance. This was of concern due to the hardware limitations of the main system and the potential for resource contention. Thus, a deliberate and well-reasoned approach was taken to each step.

The first change to the code was responsible for roughly half of the performance gain and responsible for a good portion of the code runtime. MySQL database queries can slow the code and cause longer runtimes and thus must be minimized. In the second parallelization step of the code, the queries were parallelized and some of the query costs were able to be hidden. Thus, a modest 4x speedup in the MySQL portion of the code was achieved. The Unfortunately, further parallelization of the database calls could not be achieved. While the code is insensitive to the order of the training sets within the overall training iteration, it is extremely sensitive to the position of the data within each training set. Although Cilk has code in place to allow for the training loops to access a certain position in an array, the MySQL row fetches cannot be guaranteed to be executed in order. Thus, if the inner most loop of the data loading code were to be executed in parallel, the data almost certainly would be out of order. Incorrect ordering of the training set would result in the incorrect training of the neural network. Thus, it was paramount to maintain the proper ordering of stars within the training set. Fortunately, even parallelizing the outer loop of the MySQL calls has a large effect.

Parallelizing steps	Time to execute MySQL load (sec):	MySQL speedup
More efficient memory loading	5492.7	
Algorithm with Cilk	1412.07	3.889821326
Algorithm with reducers	1392	3.945905172

Figure 5: Execution time of the MySQL code

Given that: γ = MySQL query cost, s= iterations, r=number of stars, and κ = MySQL row fetches, the addition of cilk results in a reduction from $\mathcal{O}(s*r*(n*(n^2)+n*\kappa+\gamma)$ to $\mathcal{O}(r(n+n*\kappa+\gamma))$. In order to achieve this reduction of s, the hardware would require s processors and s connections to the database. However, because the hardware used only had 4 processors, only a 4x reduction in speed was achieved, as can be seen in Figure 5 above. Throughout the data gathering process, the MySQL load times with Cilk hovered within 20-30 seconds of the 1400 second mark. The stability of the execution time was impressive given that the background load of the system was not held constant. While the first two parallelization efforts focused on the training algorithm's interaction with the MySQL database, the following steps focused on speeding up the main neural network code. The

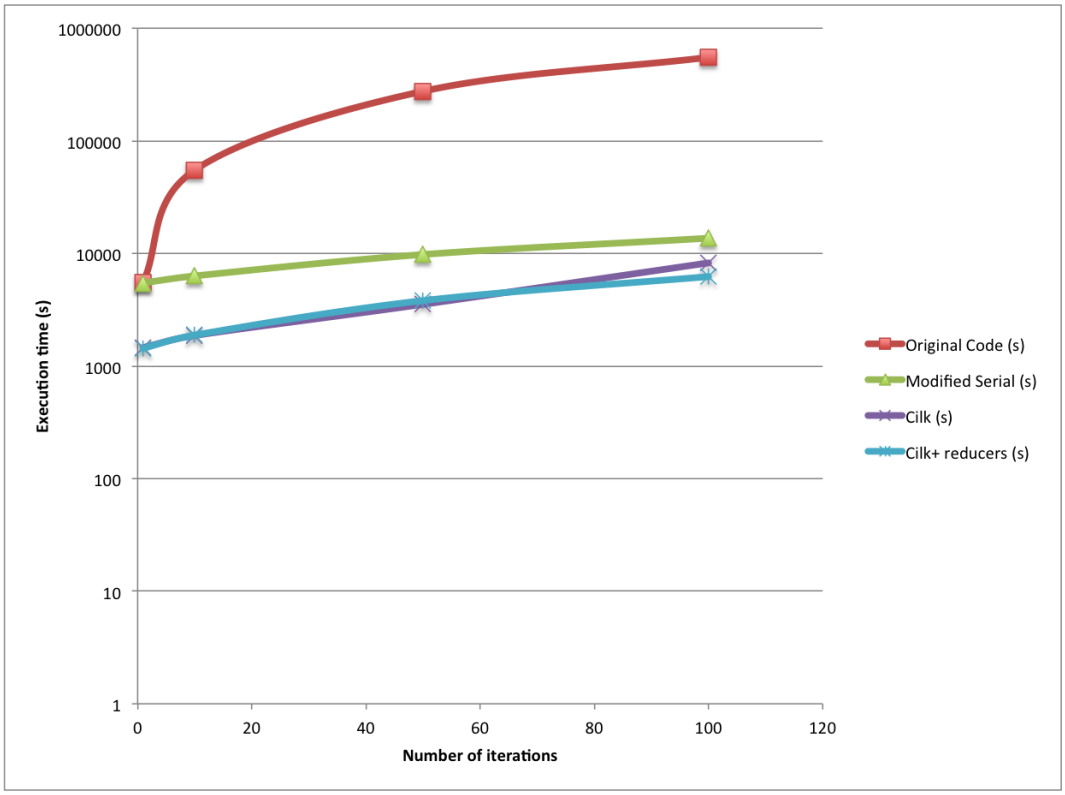

Figure 6: Execution time of the four steps

third step taken to speed up the code was to insert cilk_for statements wherever possible. The use of the cilk_for code would allow the program to utilize all the computer cores and threads available. While the overall work done by the algorithm remained the same, the depth changed because of the parallelization done by cilk. The theoretical depth of the algorithm on a system with n processors would be $\mathcal{O}(n)$ for the cilk for loops with an embedded serial for loop. However, on the hardware used, this value was lower. In this first use of Cilk on the neural network algorithm, reducers were not used to see the effects, if any, of resource contention on the hardware. Therefore, this third step of parallelization was effectively a baseline for parallelization effects. With just the application of cilk_for to the outer loops of the matrix multiplication code within the neural network, a significant speedup was realized. All 8 threads and 4 processors were fully utilized and the cumulative speedup

from the original serial code was approximately 70 x faster. The final step was to add cilk reducers and parallelize both the inner and outer loops of the matrix multiplication code.

Cilk reducers allow for the inner loop of a matrix-vector multiplication code to execute without race conditions developing. They allow for values to be summed up and collect the results of all the executing threads. By doing this, Cilk allows further parallelization to occur. When the reducers were applied to the code, a substantial increase of approximately 90 x speedup relative to the original code was seen. There were not any problems involving resource contention on a resource constrained system. Figure 6 below shows the overall results of the the parallelization efforts. While the figure seems to show most of the speedup coming from the changes in the modified serial code, in reality the changes here accounted for roughly half of the total speedup. Overall, the Cilk parallelization efforts were able to have a significant impact on the code execution time.

Number of iterations	Original Code (s)	Modified Serial	Cilk (s)	Cilk+ reducers (s)
1	5535.18	5455.53	1450.27	1426.61
10	54984.8	6313.63	1867.48	1885.06
50	275841.5	9789.84	3531.25	3812.73
100	551683	13654.6	8242.04	6230.55
Speedup:		40.40272143	66.935249	88.54483152

Figure 7: Execution times of the four steps and final speed up values

Testing at larger iteration numbers than 100 was not performed mainly due to the long runtimes of the serial code. At s=100, the serial code executes in approximately 6.3 days, as seen in Figure 7 above. Given the log-linear growth of the execution time for all the codes, larger iteration permutations were untenable.

5 Conclusion

This project was able to demonstrate the effects of optimization and parallelization of a standard neural network. The effects of database queries and fetches were minimized and parallelized in as much as is possible, given the overall constraints of the training order required. Along with minimizing the impact of the MySQL calls, this project parallelized the neural network training algorithm as well as the update and back-propagation algorithms. This led to a total execution speedup of approximately 90x on the hardware tested. Future tests on a cluster network will yield further improvements in database access time and overall training execution time. Further improvement on a GPU cluster may be achieved through the use of OpenCL, even though it was slower on the local hardware. Overall, this project was able to show that parallelization of a neural network and its accompanying training code is a good way to minimize the training time of a neural network.

6 References

[1] Bardwell G. (1995) On-Board Artificial Neural Network Multi-Star Identification System For 3-Axis Attitude Determination *Acta Astronautica Vol. 35, Suppl.,* pp. 753-761, Elsevier Science Ltd Great Britain

[2] Bezooijen W. H. (1996) Autonomous star trackers for geostationary satellites *Lockheed Martin Advanced Technology Center*

[3]Liebe C. (1992) Pattern Recognition of Star Constellations for Spacecraft Applications *Aerospace and Electronic Systems Magazine*, IEEE, Volume: 7 , Issue: 6 pp.34-41

[4]Springmann, J. C., Sloboda, A. J., Klesh, A. T., Bennett, M. W., Cutler, J. W., The Attitude Determination System of the RAX Satellite, *Acta Astronautica, Volume 75, June-July 2012*, pages 120-135

[5] Hoffleit D. & Warren W.H. (1991) The Bright Star Catalogue, 5th Revised Ed. (Preliminary Version) *Astronomical Data Center, NSSDC/ADC* Yale University