Improving the Performance of Batch-Driven Distributed Systems Using Genetic Algorithms, Artificial Neural Networks and Bayesian Networks

Nicolas Grounds NICOLAS.GROUNDS@RISKMETRICS.COM
RiskMetrics Group

Jason Madden JASON.MADDEN@RISKMETRICS.COM
RiskMetrics Group

Abstract

Large-scale distributed systems used to run a long, predictable series of operations on a recurring basis (batch-driven systems) are a prime target for optimization to reduce resource usage or time consumption, since even a small improvement may yield large dividends over time. Machine learning techniques are a good way to approach this optimization problem since there are typically many variables that interact in unknown ways.

This paper discusses a practical approach to optimizing an existing real-world system using an artificial neural network for simulation, a genetic algorithm for exploration, and a Bayesian network for human presentation and generalization.

1. Introduction

Solving increasingly large problems requires increasingly large computational resources. After a point, it is not possible to provide these resources to a single process running within the confines of a single physical machine. Indeed, it is often economically infeasible to reach this point. For this and other reasons such as reliability, it is desirable to divide the problem into smaller sub-problems and distribute them among multiple, cooperating machines. These cooperating machines are known as a distributed system.

Choosing the appropriate division for the problem is a topic in its own right. However, once the division is fixed, there are generally many aspects of the system itself that determine how efficiently the overall problem can be solved. In general, these aspects can be summarized as an attempt to achieve an optimal load balancing which makes full use of available processing, memory, I/O and other resources while simultaneously achieving minimal overhead and thus maximizing efficiency of the system.

Realistic systems tend to have many parameters that can be controlled to affect this balance. These parameters often influence one another; that is, changing one parameter means that other parameters which were formerly optimal now need to change as well. The sheer scale of the problem makes it an ideal area to apply machine learning techniques.

Previous work in this area has often been lim-

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

ited to graph-theory applications for the placement of data as in (Graham and Hinds, 1999) and (Tang et al., 2001) or has assumed a complete knowledge of the "shape" of the problem, like (Chuang and Cheng, 2002). A more realistic and general application is discussed in this paper. Here, we will describe a real-world distributed batch-driven system and discuss the parameters that affect its performance. A genetic algorithm approach to finding optimal parameters for a specific batch job will be developed utilizing a neural network to simulate the behaviour of the system. Finally, a Bayesian network will be constructed on top of the data for presentation to system designers and for possible generalization of the results to other batch jobs. This approach has provided promising performance improvements in synthetic tests.

2. Problem Definition and Algorithm

2.1. The Distributed System

The authors of this paper are primary developers on a large scale distributed system (known internally as BlueBox) used for processing financial data. This system supports both batch and interactive usage and is intended to be highly reliable, fault-tolerant, and to scale horizontally with the addition of new computing resources (a concept sometimes called a *grid*). In addition, there are performance goals, both hard and soft, for interactive and batch usage.

This system is composed of a *container* known as the Framework, and a set of components known as *services*. The Framework is in charge of resource allocation, reliability, fault tolerance, load balancing and communication among services. Individual services perform generally well-specified and orthogonal tasks, and they make varying demands on CPU, memory, storage space, network bandwidth, and database resources. There are currently a few dozen services.

The Framework and all services are written in the Java programming language and run within the Java runtime (JVM). Thus, the system performance is also affected by JVM settings, such as those that influence garbage collection.

Interactive use patterns are somewhat unpredictable, but batch usage is entirely predictable from one run of a batch job to the next. There are multiple batch jobs, and each batch job has a different load profile and utilizes different services (and in turn different computing resources) in differing amounts. Typical batch jobs take several hours to complete.

Parameters that can be tuned include:[1]

- Number of instances of each service
- Number of threads dedicated to each instance of a service, or shared among many services
- Number of concurrent threads processing the batch sequence and generating load
- Average amount of memory allocated to a service instance
- Type of communication used between various services (within the same process, thus being fast but consuming more resources, directly to another instance being somewhat slower but providing better distribution, or to the next available instance, which is slowest but provides the best distribution)

2.2. Task Definition

There are two outputs that can be minimized in the application of a distributed system to

[1]A detailed description of tuning parameters is included in Table 1.

a problem: either overall resource usage can be minimized or overall runtime can be minimized. The former goal might be appropriate if multiple concurrent problems were active at the same time, while the later goal achieves the quickest turnaround time for a single problem.

The BlueBox system supports the collection of the detailed statistics necessary for an analysis and hence minimization of resource usage. Such statistics include:

- CPU usage by service
- Database usage by service
- Number of requests made to a particular service, by a particular service

However, it was judged that minimizing the overall runtime would be a better goal. Minimization of the overall runtime provides more immediate feedback and better matches the way the system is typically used (with only one active batch job and time between different batch jobs). Likewise, the runtime is minimized for a single defined batch job. That is, the particular "shape" of the batch job is implicit; it is held constant and does not change (but see §5, Future Work). This approach is acceptable because the number of different batch jobs is small so it is not difficult to determine optimal parameters for each one using the approach outlined in this paper.

There are a very large number of parameters that can be tuned in the BlueBox system. Many of them are linear in the number of services that are in use. For that reason, the number of services was limited to eight (this is a property of the batch job). Likewise, the number of machines in the system was held fixed at five. The initial formulation of the problem resulted in the number of possible parameter combinations being in the billions. Realizing this was far too large for the limited time period available, some of the more esoteric parameters were eliminated from consideration, resulting in a more workable number of less than 20 million. There are five boolean- or scalar-valued parameters that control trade-offs of time vs. space that are applied globally,[2] and two set-valued parameters that control distribution and load balancing within a particular machine.[3]

2.3. Prerequisite Work

Before learning algorithms could be applied to the problem, data had to be collected, but before that could happen two questions had to be answered. First, what data was going to be collected, and second, how was this data going to be collected.

The second question was answered by making use of the extensibility mechanisms built into the BlueBox Framework. A small plugin was written in Java that interfaces with the Framework and stores statistics about batch jobs to an Oracle relational database. This data is then analyzed and reported on using a simple utility. It is worth noting that this plugin collects far more data than is currently used, allowing for future work on resource usage minimization (some 20 million statistics are currently aggregated into approximately 50 values).

The first question was more difficult to answer. Because actual batch jobs require 3–4 hours to run, it was not realistic to collect the necessary data using those jobs. However, the jobs are relatively easy to represent in terms of which services they make use of, and in what percentages. These services, in turn, are (perhaps somewhat arbitrarily) easily classified in terms of their predominant resource usage pattern: CPU, memory, other services, or simply time

[2]The global parameters are: Heap Memory, Concurrent Threads, Message Spool Threshold, Message Compression and Message Format.

[3]The machine-specific parameters are: Service Distribution and In-JVM vs. Out-of-JVM.

(due to usage of relatively unlimited external resources). Armed with this information, four mock services were written to express each such pattern. Together with four existing fundamental services which are heavily used by all batch jobs,[4] this allows the definition of scaled-down batch jobs which represent their larger brethren but can run in reasonable amounts of time (15–20 minutes). Two such jobs were defined with the intent to model existing real-world batch jobs.

Note that the specific services used and the specific combinations in which they are used (the job definitions) do not affect the approach described in this paper, though they do affect the specific results obtained (some combinations and patterns of service usage being obviously more suitable to optimization than others).

2.4. Use of Artificial Neural Networks

Even with the reduced scale batch jobs, 15–20 minutes is too long to wait to find out how well a particular set of parameters will perform, especially since there are many millions of possibilities. To account for this, some mechanism of simulating the behaviour of the system with a particular set of parameters was needed. Artificial neural networks with their ability to approximate arbitrary functions given sufficient training examples seemed ideal for this purpose.[5]

A neural network was implemented in Python. This implementation was designed to use standard incremental back propagation on a fully connected network. The hidden and input layers may use either the Sigmoid or Tanh activation function, and a bias input is applied to each neuron. The output layer may use either of those functions, or simply be linear valued. The network supports momentum, and uses an error function that applies a penalty for large weights. A significant constraint of this implementation is that it allows only one output.

Choosing an encoding of the many parameters of the distributed system to use as input to the neural network proved difficult. In an effort to keep the number of inputs reasonable, early attempts failed to properly account for the fact that certain parameters can differ from one machine to another and from one service to another. This problem was solved by quantizing these values into the fraction of machines that shared each value; for example, each of the eight possible services is assigned a number between 0.0 and 1.0, where 0.0 means it's been completely disabled, and 1.0 means it is available on all five machines. The two boolean values were encoded in binary as 0.0 and 1.0 for true and false, and the scalar values were scaled to fall between 0.0 and 1.0. This resulted in a network with 21 inputs, all between 0.0 and 1.0. The single output of the network was the overall runtime, that which we are seeking to minimize. It was also scaled to be between 0.0 and 1.0.

Many experiments were conducted to determine the optimal network configuration to use. In the end, a configuration of 3 hidden layers of 30 nodes was chosen. Training applied a learning rate of 0.01, with a momentum of 0.001.

2.5. Genetic Algorithm Exploration

Searching across the many millions of possible parameter combinations is implemented using genetic algorithms; that is, it is the job of the genetic algorithm to determine which sets of parameters produce improved performance for the system.

Within a genetic algorithm framework, several

[4]These services implement core functionality like security and storage and can all be classified as simple time consumers. They all make use of external resources.

[5]This approach of using a neural network as a simulator in combination with a genetic algorithm is similar to that described in (Sazonov et al., 2002)

key features must be chosen: a representation of individuals must be found that allows for mutation and crossover, a fitness function is needed to evaluate all individuals, and the genetic operators for mutation and crossover themselves must be defined. In addition, the genetic algorithm has several facets that control its speed of convergence and rate of exploration.

The canonical bit-string representation for individuals, and its corresponding operators, was considered and rejected for this application. While it would have been possible to use an encoding similar to that used for neural network input, this precludes, or at least complicates, the use of expert or prior knowledge to help speed the search. Instead, a more complicated data structure was used, and the operators defined specially to make use of prior knowledge.

The fitness function was defined on top of the neural network simulator. Because the runtime has no definable upper and lower bounds, 0.0 was defined to be the "best" possible runtime. Values output from the simulator were then simply subtracted from this (after being scaled up). For example, if the simulator output a runtime of 200.0 units, the fitness value assigned to it would be −200.0, and an output of 400.0 units would be evaluated as −400.0, or twice as bad.

One place where expert knowledge is applied is the fitness function. Certain individuals are known to be non-viable and these are given a very negative fitness ($-10,000.0$). For instance, a set of parameters that completely disabled one of the required services is obviously bad, as is one that chooses a memory setting that will cause the JVM to fail to start because it is too large or too small. We say these individuals posses "bad genes." In all, 8 parameters are considered in the definition of a bad gene.

Both of the genetic operators for mutation and crossover also apply expert knowledge to help reduce the generation of these non-viable individuals. When an individual is picked for mutation, one of the parameters is randomly changed using a process like that outlined below.

```
choose random parameter
if parameter is global:
  if parameter is boolean:
      parameter <= not(parameter)
  else:
      parameter
        <= random_good_value(parameter)
else:
  choose random machine
  add or remove (random) service
    to list of running services
    or list of distributed services
```

The method used for crossover was selected to give a close approximation of the traditional method of uniform crossover. Each of the global scalar parameters is simply copied randomly from one or the other parent individual. A slightly more complicated but conceptually similar method is applied to the machine-level settings; each machine in the child runs and distributes some number of services selected from the union of the services present in the corresponding parents.

The genetic algorithm framework used we developed in Python. It uses tournament selection with a probability of 0.7 of selecting the more fit individual; this number has been gradually raised to 0.9. This selection mechanism was chosen to support a diverse population in the early phases when our fitness function was less trustworthy (since it had trained on less data). After initial experimentation, the probability of crossover was fixed at 0.3 and the mutation rate at 0.2. The target population size was 2,000.

2.6. Understanding with Bayesian Networks

Although the collection of an accurate neural network simulator and a genetic algorithm-based search has proven in experiments to find better sets of parameter values for running batch jobs, this approach in of itself is limited to the amount of time and resources that can be devoted to such experiments. In hopes of using the data from experiments to find a way to generalize findings and discover underlying properties of the nature of these parameters a Bayesian network learning algorithm was implemented.[6] Bayesian networks are an excellent way to present human-readable findings about the interactions of parameters and their effect on system performance. Using experiment results and EM (the Expectation-Maximization algorithm) a Bayesian network is trained to predict performance based on the system parameters. The real key to using Bayesian networks, however, comes from a search on the different possible structures for such Bayesian networks and a scoring function to rate structures. In doing so we hope to find an optimal Bayesian network structure which not only predicts performance but can graphically describe how parameters combine (or conflict) to affect performance.

The Bayesian network, EM, and structure search were all developed in Python. EM is evaluated for either 100 iterations or until the maximum change made to any Bayesian Network probability is less than 0.0001. The structural search began with 2 hidden network nodes and iterated up to 10 hidden nodes. Within each iteration the structure search constructed a Bayesian network by connecting the runtime input node to all hidden nodes. Then, random edges were added from hidden nodes to parameter nodes for as long as such edges increased the score of the Bayesian network (or until the network was fully connected.) This is restarted (to generate more random connections between hidden nodes and parameter nodes) a total of 50 times. The function for scoring the Bayesian network was Schwartz Information Criteria (SIC) which is also called Bayesian Information Criteria (BIC). The Bayesian network with the highest score is considered the best overall.

[6](Heckerman, 1999) presents a complete tutorial on using Bayesian networks in learning.

3. Experiments

The overarching goal of the work presented in this paper was an attempt to determine if machine learning techniques could be successfully applied to the domain of distributed system optimization.

3.1. Experimental Procedure

The basic procedure to collect data and refine the parameters works as follows. Initially, a random set of parameters are generated. The batch job is run on the physical system using these parameters. This is repeated several times to generate enough data to begin training the neural network simulator. When the network has reached acceptable accuracy, the genetic algorithm is started; after 1,000 generations, the output of the genetic algorithm is a set of three individuals (sets of parameters): the best individual in the final generation, a randomly selected individual from the final generation, and a new, randomly generated individual. These three sets of parameters are then run in the physical system. This new data, plus the existing old data, is used to retrain the neural network for increased accuracy and generalization abilities. The process then repeats with the genetic algorithm creating a new set of individuals. Each iteration of this takes about an hour.

The overall stopping criteria are in need of further definition pending a more detailed analysis of the data. It is expected that after a certain point no further improvements will be possible. The runtime of the batch job will oscillate within a small range. This oscillation must be detected and iteration ceased; but doing so is fairly involved.

3.2. Neural Network

The performance of the neural network is critical the the overall viability of this approach. Fortunately, the overall error of a neural network is easy to visualize and analyze given a set of training and test data. Also useful is the availability of other neural network implementations for comparison purposes.

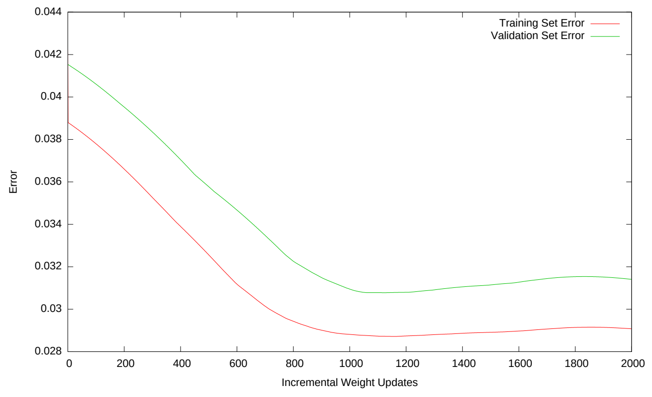

Figure 2. Error Versus Weight Updates

First, Figure 1 shows how the performance of the simulator is becoming more accurate as it is fed more data. The line labeled “Actual Runtime” shows the actual runtimes of each hypothesis. The line labeled “Net-10” represents a network trained on only ten hypothesis when it is asked to generalize across more than 100 hypothesis; it does quite poorly. However, the line “Net-50,” which shows this same network after being trained on at least five times as many examples shows the network to be matching the curve acceptably well.

Both nets were allowed to train for at most 2,500 iterations to avoid overfitting. This simple method was used early on due to the paucity of data for validation purposes and was continued for consistency. However, once enough data was obtained, it was possible to produce Figure 2, which compares the error for separate training and validation sets over many weight update iterations and tends to support this stopping criteria.

It is instructive to compare with a third party neural network. The FANN implementation(Nissen, 2003) is a freely available artificial neural network library implementation. FANN supports many training algorithms, but defaults to iRPROP, the improved resilient backpropagation algorithm, as described in (Igel and Hüsken, 2000). This training algorithm adaptively adjusts the weights based solely on the change of direction of the derivative, allowing it to eschew the need for many of the finicky parameters required by standard backpropagation, and to typically converge much faster. Indeed, in our experiments iRPROP did converge much faster and avoided local minima better than our own backpropagation implementation, as can be seen in Figure 3. Work on an iRPROP implementation of our own is underway.

3.3. Genetic Algorithms

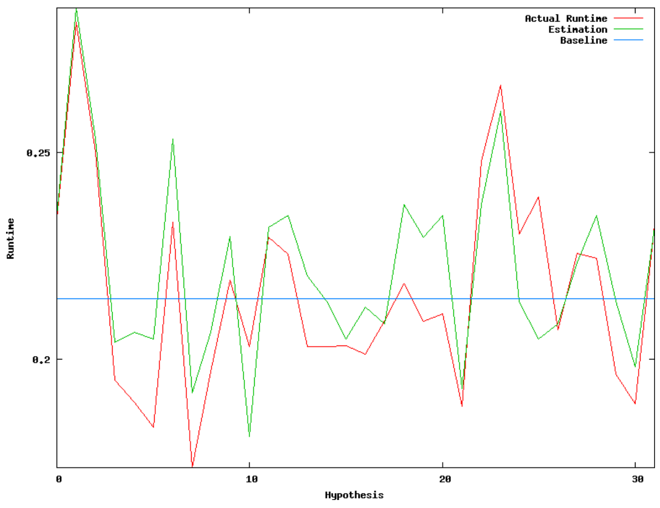

Figure 4. Best Individual’s Runtime

The genetic algorithm exploration is well represented in Figure 4. This graph is a smoothed

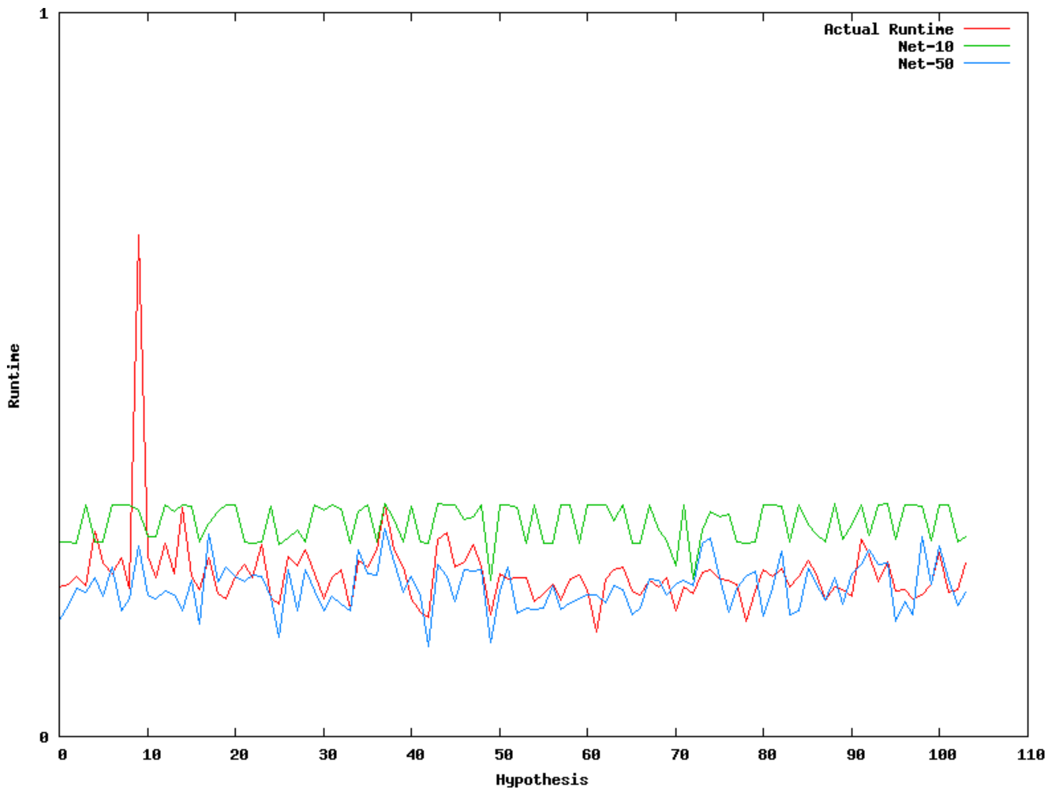

Figure 1. Comparison of Neural Nets Trained on Different Data

representation of how well the best hypothesis of a given population (after a fixed number (1,000) of generations) compares to the baseline runtime. This graph can be contrasted with Figure 5, which simply compares a random hypothesis to the baseline (one significant outlier has been left off this graph to make it easier to compare to the previous graph). The baseline runtime comes from a set of parameters chosen by the human operators of the system to perform well.

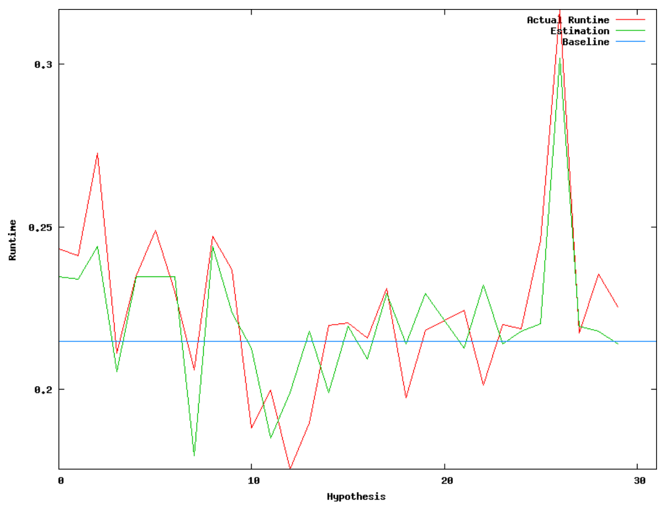

Figure 5. Random Hypothesis's Runtime

Although difficult to distinguish from the graphs, 53% of the best hypothesis runs improve on the baseline time, while only 33% of the random hypothesis runs are better than the baseline. This would be encouraging, except that a right-tailed normal test rejects the hypothesis that the mean of the best runs differs from the mean of the random runs (the acceptance value is 2.33, while the test statistic is only 1.55). However, given the relatively small sample size, the fact that the actual distribution is not clear, and the likelihood of noisy data, this evaluation should not be considered rigorous.

Several general observations can be made re-

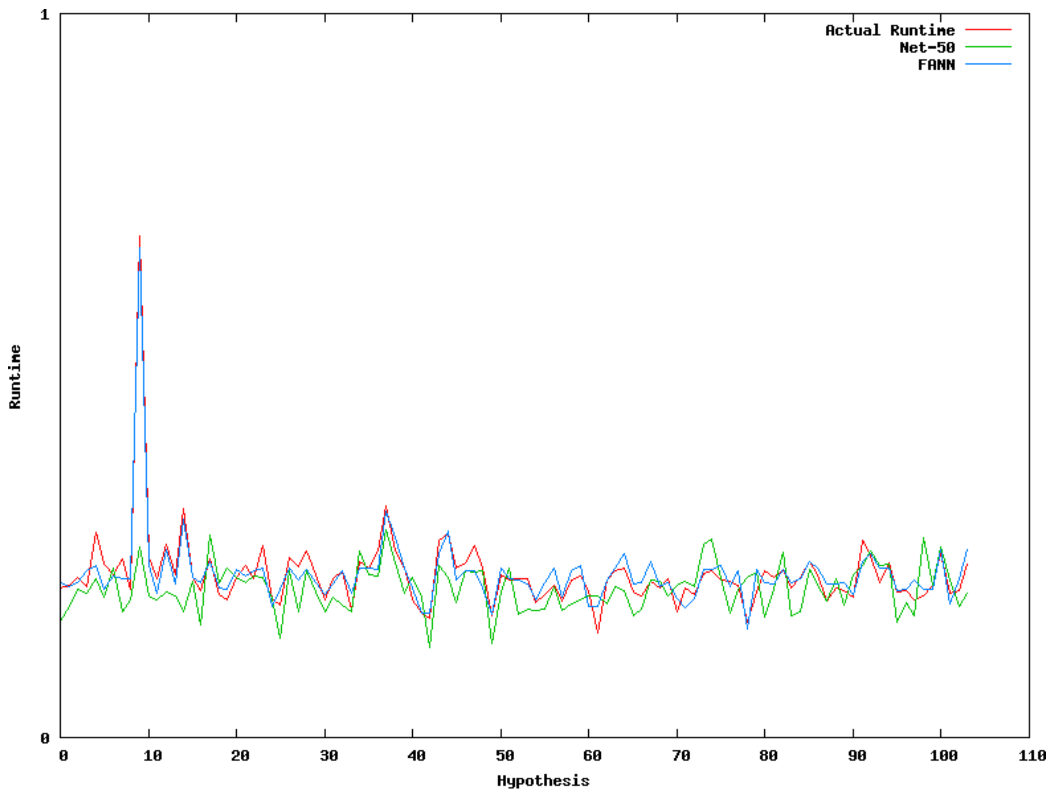

Figure 3. iRPROP with FANN versus Incremental Backpropagation

garding the performance of the genetic algorithm. First, the degree of variance from the baseline in Figure 4 shows that exploration is in fact taking place. There is also a general trend to reduce the amplitude of the exploration over time, which can be interpreted as the algorithm "zeroing-in" on a good set of parameters (possibly a local minima).

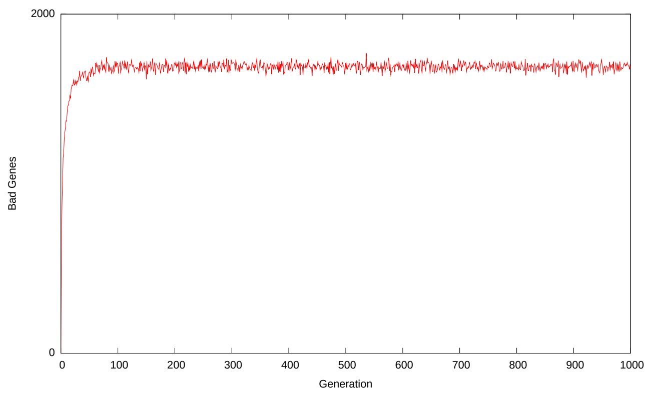

Figure 6. Prevalence of Bad Genes

The prevalence and increase of bad genes in the population, as shown for one trial in Figure 6, unfortunately indicates that there is room for improvement in the genetic operators defined. It takes very few generations for a population that is composed completely of good genes to evolve mostly bad genes. Even using "expert" knowledge for mutation and crossover in an attempt to reduce the generation of bad genes, the bad genes come to dominate the population in relatively short order, making it difficult to maintain population variance.

The destructiveness of mutation and crossover is a well-known problem in evolutionary computing. Typical approaches include increasing the size of individuals, changing their encoding, re-defining the genetic operators, or simply further reducing the mutation and crossover rates while producing more generations. With additional work, any of those approaches could be applied here.

3.4. Bayesian Networks

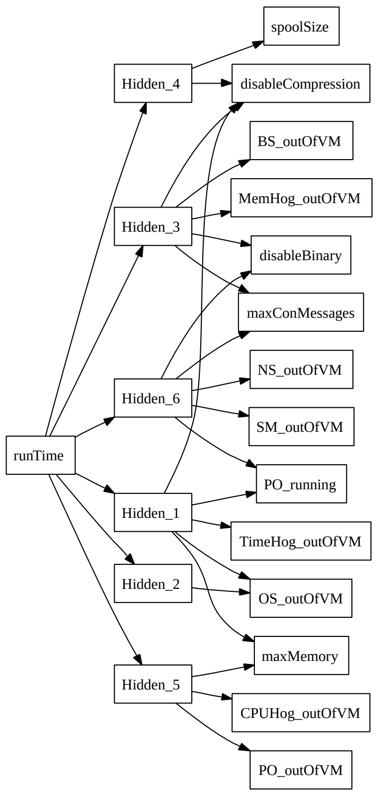

Figure 7. Bayesian Network for Job 1

Since the true relationship of parameters was unknown before beginning this project it is difficult to analyze the results of the Bayesian network structure search. However, it is encouraging the the structure search did produce results that are believable. For instance, in the Bayesian network for Batch Job 1, Figure 7, the message spool threshold (spoolSize) and message compression (disableCompression) parameters were analyzed as being related (through their mutual parent, Hidden_4). This is an expected result, since compression determines the size of messages, which in turn influences the maximum size of messages that will be kept in memory vs. spooled to disk.

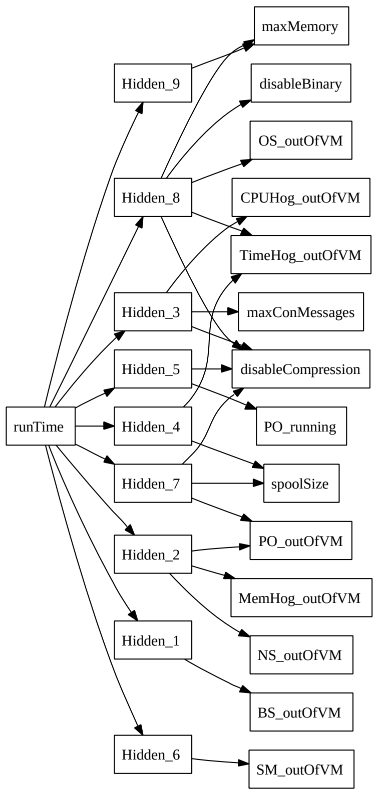

Figure 8. Bayesian Network for Job 2

Also in the Bayesian network for Batch Job 1, the compression and heap memory (maxMemory) settings are related (through the node, Hidden_1) which is also logical since the enabling/disabling of compression would have an affect on the amount of memory needed to keep the batch job running efficiently. Interestingly, the Bayesian network for Batch Job 2, Figure 8, found the same two relations.

In addition to the relationships found to give the highest BIC score it is also useful to analyze what relationships created during the structure search process resulted in raising the BIC score. Figures 9 through 12 describe how many edges added from a particular parameter node to a hidden node already connected to other parameter nodes (siblings) resulted in an increase in BIC score: a productive edge. These numbers depend heavily on how many

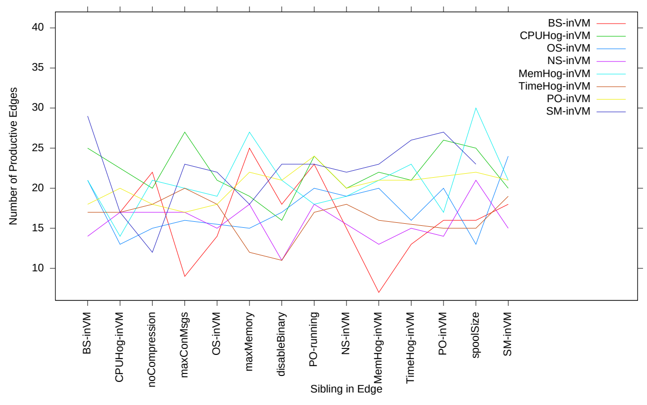

Figure 9. Number of Productive Edges for Job 1 for inVM/outOfVM nodes

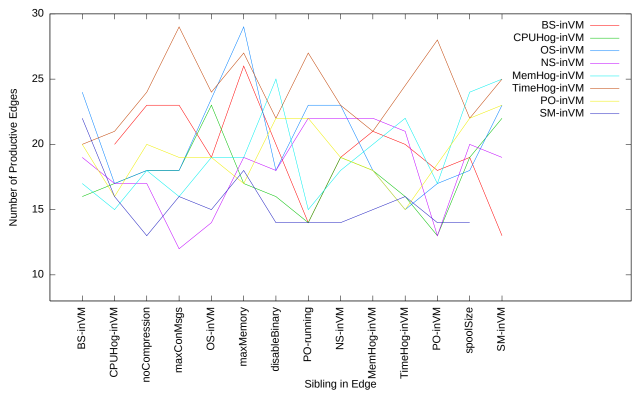

Figure 11. Number of Productive Edges for Job 2 for inVM/outOfVM nodes

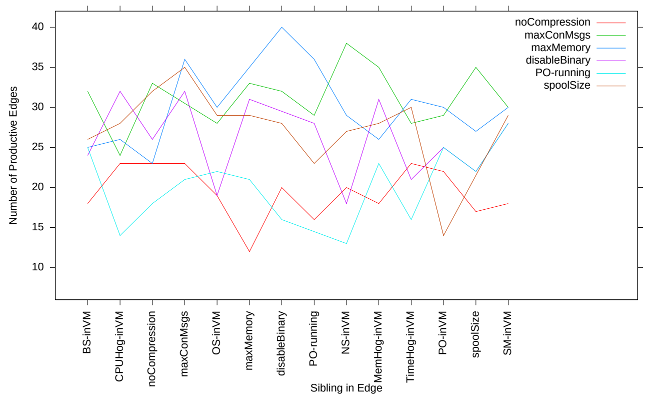

Figure 10. Number of Productive Edges for Job 1 for other nodes

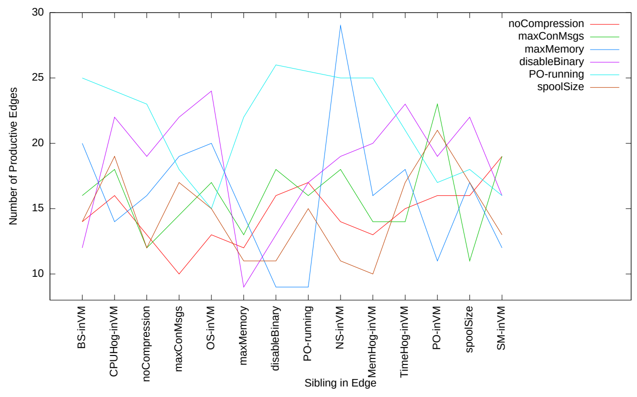

Figure 12. Number of Productive Edges for Job 2 for other nodes

times such edges are even considered which is a random probability in the structure search algorithm. To alleviate this a higher number of iterations (random restarts) could be chosen to ensure that all edges are randomly chosen roughly an equal number of times, although we feel that the value chosen was a fair. The interesting results of these sets of figures is the differences in which relationships were found to be productive most often.

For Job 1, the most often productive relationship was between the maximum amount of memory and message format (disableBinary). This means that adding an edge in the network from the maxMemory parameter node to a hidden node already connected to disableBinary parameter node resulted in a higher network score more than any other edge combination. For Job 2, however, this relationship was actually found to rarely result in a higher network score. The relationships for Job 2 that most often resulted in increasing the network score were whether the TimeHog service was in VM and the maximum number of concurrent messages (maxConMsgs) allowed per machine, whether the OS service was in VM and the maximum amount of memory, and maximum amount of memory and whether the NS service was in VM.

Although a complete analysis of these relationships would require a great deal of knowledge about the system and services and interactions between the parameters which would take many pages, some simple and intuitive (and in some cases, completely hypothetical) explanations are proposed in the following paragraph.

The relationship of maxMemory to disableBinary possibly signifies that the trade-off of how much memory it takes to process binary messages vs. non-binary messages is important. The relationship between TimeHog-inVM and maxConMsgs possibly signifies that since the TimeHog service requires no actual resources (it mearly consumes one of the allowed message counts for a period of time) the maximum allowed number of concurrently processing messages is important. The relationship between OS-inVM and maxMemory possibly signifies that the OS service typically consumes a great deal of memory and so allowing it to receive service calls within the same VM requires that VM to have more memory available to it. The relationship between maxMemory and NS-inVM possibly signifies the same as maxMemory and OS-inVM above or even possibly signifies that since the NS service receives a inordinate amount of the number of service calls during the batch job process (and those calls are often asynchronous) making all those calls within the VM simply requires a lot of memory.

Regardless of the actual truthfulness of some of the explanations above one clear result of the Bayes network learning is that the maximum amount of memory interacts with many other parameters in ways that greatly affect the performance of the batch job.

4. Related Work

Genetic algorithms are a popular approach to solving large problems and exploring large spaces in a directed fashion. There are several applications that make use of this method to solve database-like problems (graph-theory problems) of making resources available where they are used the most with the least cost, such as those documented in (Chuang and Cheng, 2002), (Tang et al., 2001), and (Graham and Hinds, 1999). Genetic algorithms have also been applied specifically to scheduling problems (a type of resource utilization problem), as in (Meijer, 2004). Both the scope and combinations of the approach presented in this paper would seem to make it relatively unique.

Much work has also been done in the area of Bayesian network structure learning. Although this experiment used a simple hill-climbing greedy search with simulated annealing (restart with a new random network) for searching the structure space others have designed and applied useful algorithms such as those described in (Koivisto and Sood, 2002) or the Bayesian Structural EM Algorithm defined in (Friedman, 1998). In their work they propose a new algorithm for discovering the best Bayesian network structure, i.e. the one that maximized the posterior likelihood: $p(\text{data}|\text{structure})$. Their algorithm was designed to find best structures for subnetworks but is also modifiable to work globally. In (Getoor et al., 2002), Getoor et. al. explored other representations in addition to Bayesian Networks to represent relationships in data.

5. Future Work

In the interests of time, the scope of this experiment has been deliberately limited. This means that there are unanswered questions of both a practical and theoretical nature that remain to be explored.

Perhaps the most interesting and the most useful next step would be to attempt to generalize the neural network simulator and the Bayesian network classifier. As it stands, both networks are trained on the implicit behaviour of one batch job (saving time because this reduces the size of the input and hence the number of training examples needed). For a small number of production batch jobs this can still be useful; since the batch jobs are run every day it will be possible to perform a limited amount of exploration on a daily basis, and within a month or so the networks will be useful enough to allow the genetic algorithm to output parameters that are likely to be improvements. However, by providing these networks with input statistics that describe the shape of a batch job to execute, if their simulation is accurate enough it could be possible to have the genetic algorithm immediately produce a targeted set of parameters for an entirely new batch job, and for the Bayesian network to reveal interesting inter-service relationships. The major problem with this remains the probable requirement for a large number of training examples; however, it's possible that the number really wouldn't be that large.

Another interesting avenue for continued exploration is the dual of the goal discussed in this paper, minimizing runtime for a single batch job, which is minimizing overall resource usage for a batch job. Minimizing resource usage is more challenging because it requires an accurate picture of the resource utilization. This means extremely detailed statistics need to be captured (done) but then these statistics have to be translated into input for the learning algorithms. As with the previous paragraph, this means that more real training data must be obtained, a time-consuming proposition. The benefit to being able to do this is to allow for greater scalability and more concurrency in the distributed system.

The problem at hand was made more tractable by reducing the combinations of parameters under consideration. However, it is quite likely that many of the parameters left out, particularly those related to the behaviour of the JVM, could have significant impacts on the performance of the system. Therefore, reconsidering the problem with those parameters available for exploration could yield dividends. Again, the practical problem with this is the exploding number of inputs requiring increasingly more training data.

There is a good deal of work that could be done to improve or alter the use of Bayesian Networks in this project. In addition to using the algorithms or possible other relational data structures referenced in the Related Work section above, one excellent area for future work would be to use the genetic algorithm implementation to perform the structure search. The only sustantial work that would need to be done is to find a suitable encoding for the Bayesian Network structure such that appropriate crossover and mutation functions could be applied.

The genetic algorithm itself could potentially be improved by changing its starting set of hypothesis. Currently, each iteration generates a new set of random individuals and evolves them from there. Saving the previous iteration's final generation and starting from there could produce improved results, or at least be aesthetically satisfying. Of course, the dominance of bad genes would need to be overcome first.

6. Conclusion

The chief hypothesis set to be explored in this project was that machine learning techniques could be applied to find better, if not an optimal, set of tuning parameter values to configure a distributed system with that would outperform current standard, human-defined values. It is clear from results that several sets of parameters values were discovered that could improve performance. This also shows that the genetic algorithm is properly searching the possible parameter space and finding those sets of parameter values that will increase performance. This, together with the results above, also shows the neural network is able to, within a certain degree of accuracy, predict performance. With more examples and training time, it is expected that the neural network would improve even more.

References

Chuang, P.-J. and Cheng, C.-W. (2002). On file and task placements and dynamic load balancing in distributed systems. *Tamkang Journal of Science and Engineering*, 5(4):241–252.

Friedman, N. (1998). The bayesian structural em algorithm.

Getoor, L., Friedman, N., Koller, D., and Taskar, B. (2002). Learning probabilistic models of link structure. *Journal of Machine Learning Research*, 3:679–707.

Graham, J. M. and Hinds, C. V. (1999). Optimal placement of distributed interrelated data components using genetic algorithms. In *ACM Southeast Regional Conference*.

Heckerman, D. (1999). A tutorial on learning with bayesian networks. *Learning in Graphical Models*.

Igel, C. and Hüsken, M. (2000). Improving the Rprop learning algorithm. In Bothe, H. and Rojas, R., editors, *Proceedings of the Second International ICSC Symposium on Neural Computation (NC 2000)*, pages 115–121. ICSC Academic Press.

Koivisto, M. and Sood, K. (2002). Exact bayesian structure discovery in bayesian networks. *Journal of Machine Learning Research*, 5:549–573.

Meijer, M. (2004). Scheduling parallel processes using genetic algorithms. Master's thesis, Department of Computer Science, University of Amsterdam.

Nissen, S. (2003). Implementation of a fast artificial neural network library (fann). Master's thesis, Department of Computer Science, University of Copenhagen.

Sazonov, E. S., Del Gobbo, D., Klinkhachorn, P., and Klein, R. L. (2002). Failure-free genetic algorithm optimization of a system controller using safe/learning controllers in tandem.

Tang, K.-S., Ko, K.-T., Chan, S., and Wong, E. W. M. (2001). Optimal file placement in vod system using genetic algorithm. *IEEE Transactions on Industrial Electronics*, 48(5).

Parameter	Comments
Heap Memory	Amount of memory dedicated to Java heap. Setting this too high can limit the number of threads available or limit the amount of space for internal JVM data, and ultimately cause the JVM or Framework to crash. Setting this too low will result in reduced scalability and can ultimately cause the Framework to crash.
Concurrent Threads	Total number of operations allowed to be handled by all services in a Framework instance. Setting this too low creates unnecessary delays; setting this too high can trigger memory or threading problems.
Service Distribution	Controls which services are running on which Framework instances. Running more services means higher availability and potentially better throughput. However, the characteristics of each service need to be considered; it may not be wise to put two services that each need large amounts of memory on the same Framework.
In-JVM vs. Out-of-JVM	Controls whether requests from one service to another service running in the same Framework are allowed to go directly to that service in that same JVM, or must be load balanced across any available machines. In-JVM requests have lower overhead but reduce load balancing and can trigger memory or threading problems. Set for each service on each machine.
Message Spool Threshold	This property is a trade off between memory usage and speed. It determines when messages are held in memory for processing and when they must be spooled to disk.
Message Compression	Disabling compression reduces CPU usage and so might reduce latency at cost of increased memory usage and transport times (particularly when going out-of-VM).
Message Format	There are two formats in which messages can be transmitted. The XML format compresses extremely well, but can be slower to read and write. The binary format can be faster to read and write, particularly when the message is small. Its performance tends to degrade, however, when compression is enabled.

Table 1. Tuning Parameters