

# CS545: Assignment 8

## Predicting Parallel I/O Performance

Kate Ericson

December 16, 2009

---

### Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Overview of Data</b>	<b>1</b>
2.1	Splitting the Data . . . . .	2
<b>3</b>	<b>Neural Net Analysis</b>	<b>2</b>
3.1	80% Training Fraction . . . . .	3
3.2	90% Training Fraction . . . . .	3
3.3	95% Training Fraction . . . . .	3
<b>4</b>	<b>Principle Components Analysis</b>	<b>4</b>
4.1	Using PCA . . . . .	4
<b>5</b>	<b>Conclusions and Further Work</b>	<b>7</b>

---

## 1 Introduction

While the Community Climate System Model (CCSM) has been parallelized, the I/O is still serial. We are currently testing the Parallel I/O (PIO) library with the Parallel Ocean Program (POP), to get a general idea of possible performance gains. While PIO has the potential to vastly improve performance, it does require several parameters to be properly set. A “bad” choice of parameters can lead to serial read/write speeds. Unfortunately, we currently have no way of testing a parameter choice without actually running a test. In this project, I explore the feasibility of using a neural network to predict the performance of parameters.

In the second section, I give some background on the data, as well as discuss how I chose what to use as inputs and outputs to my neural network. In Section 3, I describe my neural net experiments, as well as their results. In Section 4, I explore the effects of using Principle Component Analysis (PCA) to reduce the dimensionality of the data, and in Section 5 I conclude and discuss ideas for further work on this topic.

## 2 Overview of Data

For this project, I am using data gathered from NCAR’s Blue Gene/L system Frost. Frost has 6 configurable PIO parameters – there are more than 4,000,000 possible permutations when 64 nodes (128 cores) are used. This number only grows as more cores are added.

At the moment, I am using the results from 576 successful runs of testpio on Frost. Due to access problems, I haven’t been able to get more results – a future paper would benefit from looking at not only

successful, but failed runs as well. These runs were performed with both 64 and 128 node runs (equivalent to 128 and 256 cores).

## 2.1 Splitting the Data

Trying to figure out how to split up this data was a bit of a challenge. After an initial pruning of the data (this involved removing date/time, key, machine, error, and search method columns), I was still left with 11 columns I felt were important. To help give an idea of what these columns are like, I initially ran the R function `summary()` on the data. The results are displayed below:

```
> summary(X)
      nodes      ioFMT      rearr      stride      nprocsIO
Min.   : 64.00    pnc:386    box :296   Min.   : -1.00   Min.   : -1.00
1st Qu.: 64.00    snc:190    none:280  1st Qu.: -1.00  1st Qu.: 24.00
Median : 64.00                                Median :  1.00   Median : 35.00
Mean   : 88.89                                Mean   : 15.36   Mean   : 39.17
3rd Qu.:128.00                                3rd Qu.: 10.00  3rd Qu.: 44.00
Max.   :128.00                                Max.   :128.00  Max.   :126.00

      num_agg      blkdcmp      d_rearr      d_stride
Min.   : -1.00    xye :158    PIO_rearr_ box :296   Min.   :  1.000
1st Qu.: 22.00    xyze:226   PIO_rearr_ none:280  1st Qu.:  1.000
Median : 39.00    xze  : 72                                Median :  1.000
Mean   : 42.98    yze  :120                                Mean   :  2.646
3rd Qu.: 59.25                                3rd Qu.:  4.000
Max.   :126.00                                Max.   : 21.000

      readT      writeT
Min.   : 1156    Min.   : 3720
1st Qu.: 1838    1st Qu.: 4997
Median : 4382    Median : 8878
Mean   :11574    Mean   : 28531
3rd Qu.: 9077    3rd Qu.: 18210
Max.   :60771    Max.   :162218
```

The next problem was how to appropriately split this up between inputs and outputs. For the most part, this is pretty cut and dry – I have 7 different input parameters, and the two main outputs that I’m interested in: the read and write times. The two confusing parameters are chosen by `testpio` if it detects a problem with the user supplied parameters. I eventually decided to have these two parameters be outputs of my neural network – as they are derived by the program, and are not manually input by the user. There is a chance that a neural network may benefit from using these derived inputs as the actual inputs into the neural net – this could be a topic for future work in the area.

## 3 Neural Net Analysis

As the main purpose of this project is to explore the feasibility of using a neural network to predict the performance of parameters, I decided to focus on a broad analysis of performance. As I only have a small amount of data to work with, I decided to use a large percentage of training data: I’m looking at 80%, 90% and 95% training fractions. While I did not perform enough runs to gain true statistical significance, I still believe that I managed to find a good sampling of the area. For all training fractions, I tested the performance of 9, 20, 200, and 1000 hidden units. As a further measure to keep the running time down, I also limited the maximum number of SCG iterations to 5000.

Initially, I was worried about how to accurately report the results from my training sets – I have four outputs, all of varying complexity (one is a simple boolean flag, two are generally in the 4 digit range, and the last is ranges between 1 and 21. With this vast discrepancy in scale, I need to be careful when looking at RMSE.

### 3.1 80% Training Fraction

For the first set of tests, I used an 80% training fraction – there wasn’t any strong reason behind using 80%, simply that I have gotten into the habit of using this percentage as a starting point. Since I have such a small amount of actual data, I’m sure I don’t want to have any training fraction lower than this. A summary my results can be seen in Table 1. Looking at the RMSE in this table there’s a pretty clear, steady increase in performance as hidden units are added.

While the first column (`boxD Rearr`) looks very promising, this is the boolean flag column – while it’s clear that it’s not completely off, it doesn’t mean it’s actually performing that well. The RMSE of `writeT` with 1000 hidden units looks high, but taking a look back at the summary, it doesn’t look like it’s doing particularly bad. I also found it interesting that the 200 hidden unit network was more accurate about `readT` predictions than the 1000 hidden unit network – am I seeing over fitting here, or is it just a result of not having enough tests?

Hidden Units	boxDRearr	d-stride	readT	writeT
9	0.029473153	2.309832	8965.6411	23627.343
20	0.003609077	2.306427	8021.7610	20993.848
200	0.000263402	2.649040	994.9614	4149.712
1000	0.000213997	2.062518	1370.2408	3989.318

Table 1: RMSE with 80% training fraction

### 3.2 90% Training Fraction

Next, I tried using a 90% training fraction. I again ran all tests with 9, 20, 200, and 1000 hidden units. With a larger training fraction, I’m expecting better results overall – I’m also interested in seeing if there’s a trend where 1000 hidden units starts to over fit the problem. The results of these tests are shown in Table 2

Hidden Units	boxDRearr	d-stride	readT	writeT
9	0.008367500	3.033001	8718.773	22685.036
20	0.004386663	3.029828	10988.209	28566.902
200	0.000070252	2.530171	1028.916	2907.007
1000	0.001466849	2.535646	1335.409	3550.497

Table 2: RMSE with 90% training fraction

According to these results, using 200 hidden units worked best across the board – looking at this, I feel much more comfortable claiming that 1000 hidden units starts to over fit. It’s also becoming clear that the network is doing a much better job of predicting `readT` than `writeT`. There must be something going on that allows it to better predict the read times than the write times – is there just too much randomness in the write speed on the system, or is it something else?

### 3.3 95% Training Fraction

For my last set of runs, I used a 95% training fraction. Even with the relatively small dataset I have, I felt that this should have pretty decent results. With this, I’m looking to see if I again see the over fitting trend with 1000 hidden units, as well as comparing RMSE across read and write times – are my networks having an easier time predicting read times than write times? The results of this run are in Table

Using a 95% training fraction, I’m actually seeing that 1000 hidden units is outperforming 200 hidden units. This appears to be completely contrary to my previous theory. The best reason I can come up with for this is that I simply don’t have enough data to adequately train 1000 hidden units with only an 80% or

	boxDRearr	d-stride	readT	writeT
9	0.009975820	1.0806105	10671.731	26784.696
20	0.004661883	0.9974189	10900.197	27940.856
200	0.000763779	2.5490300	1353.403	4381.331
1000	0.000291927	1.2409167	1076.484	3090.679

Table 3: RMSE with 95% training fraction

90% training fraction – this lack of training hurts the overall predictions when there’s less training data, but as I give it more data, it seems to start to put those extra units to work.

Based on the results of all 3 tests, it definitely looks like the neural networks are having a much better time predicting the read time than the write time – this does not look very promising at all. The main thing we’re looking to be able to predict overall from this is the write times. While write times and read times tend to be related, read time is still not generally a good indication of write time. These results also seem to show that this would not be a good approach at all on machines where there is a large variation in write times (such as Kraken).

## 4 Principle Components Analysis

While the results from directly using a neural network weren’t altogether promising, there is some hope that I can achieve better results by cutting down on the dimensionality of the inputs. To do this, I decided to use Principle Components Analysis, as discussed in [1]. To do this I used the function provided in class [2]:

```
pca <- function(X) {
  if (missing(X)) {
    cat("Usage: r <- pca(X)
      X is nSamples x nComponents
      r$V is nComponents x nComponents eigenvector matrix (X %*% V)
      r$values are the nComponents eigenvalues \n")
    return(invisible(NULL))
  }
  eigenResult <- eigen(cov(X))
  list(V = eigenResult$vectors, values=eigenResult$values)
}
```

To see exactly what was going on, I decided to create graphs to show the results of running PCA on my data. These graphs are shown in Figure 1. The actual values for the image on the right side of Figure 1 can be seen in Table 4. Looking at the eigenvalues, I’m not entirely certain that there’s much to gain from cutting dimensionality. The eigenvalues only range from 2 to 0, and I only have 9 dimensions to begin with. It looks like my best bet will be to try looking at only 6 of the dimensions (from Figure 1, it looks like this is around 0.8).

### 4.1 Using PCA

While the PCA results weren’t overly promising, I decided to see if I could get better results by reducing the dimensionality to 6. Instead of rerunning the full suite of tests I went through in the Section 3, I decided to only look at 200 and 1000 hidden units, for training fractions of 90% and 95%. I have gathered the results from this in Table 5 and Table 6.

Quite frankly, the performance was abysmal. Going back to Figure 1, there is no real flattening of the eigenvalue curve. While I was hoping that the drop below 1 meant I would be able to prune the dimensionality,

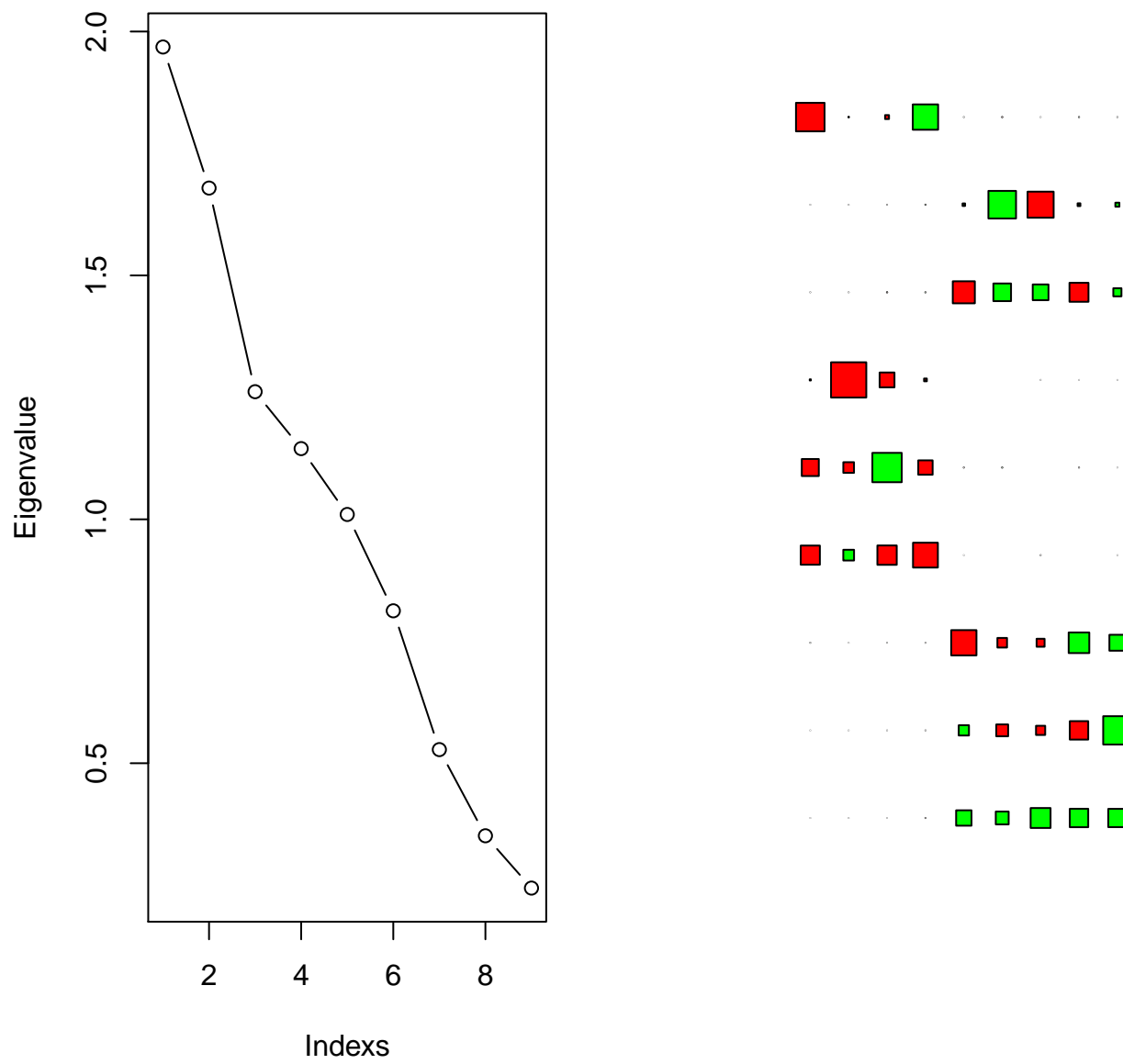


Figure 1: Results of PCA analysis

pcaResults	V1	V2	V3	V4	V5	V6	V7	V8	V9
nodes	-0.744	0.016	-0.103	0.660	0.003	-0.007	0.002	0.006	0.003
pncFMT	-0.001	0.002	-0.003	0.007	0.067	0.720	-0.679	0.073	0.105
boxRearr	-0.002	0.002	-0.008	0.007	-0.575	0.460	0.410	-0.495	0.212
stride	-0.030	-0.919	-0.387	-0.073	0.000	0.000	-0.001	0.000	-0.001
nprocsIO	-0.444	-0.278	0.765	-0.375	-0.005	0.007	0.000	-0.004	-0.001
num-agg	-0.499	0.280	-0.504	-0.647	-0.001	0.000	-0.003	0.000	-0.001
xye	0.003	-0.001	0.002	-0.002	-0.658	-0.250	-0.208	0.537	0.415
xze	0.000	-0.001	0.002	-0.002	0.267	-0.310	-0.239	-0.480	0.738
yze	0.000	-0.001	0.000	-0.005	0.401	0.335	0.520	0.480	0.476

Table 4: Results of PCA

	boxDRearr	d-stride	readT	writeT
200	830.3799	3567.673	32495665	88388332
1000	512.4829	1176.750	29096473	76025746

Table 5: RMSE after dropping to 6 dimensions with 90% training Accuracy

	boxDRearr	d-stride	readT	writeT
200	514.3987	2892.744	38737573	108108318
1000	564.3742	1794.745	17977164	47099491

Table 6: RMSE after dropping to 6 dimensions with 95% training Accuracy

it appears that this is not the case. On the other hand, it is now obvious that there is a definite need to include those last few dimensions, as they have a large impact on the performance of the network.

## 5 Conclusions and Further Work

While I think I managed to get an idea of what's going on with these experiments, I really didn't have enough data to reach any definitive conclusion. I also ran all these experiments on only "good" data – none of my neural networks saw a bad configuration, I want to see how this will affect the performance. Will it be beneficial, or will it throw the network completely off? I'd also be interested in seeing how well a network can perform on data from Kraken, which has much more variation in write speeds. I would also be interested in having a neural net that searched through the parameter space itself – this could theoretically be a great speedup in exploring the parameter area.

Eventually, I would like to be able to use a neural network to at least approximate the performance of given parameters in a search function. This would reduce the cost of any exploration of parameter space – not only of time, but in actual dollars as well. It would be a definite boon when moving up to using more cores in experiments, when actual dollar cost starts to become an issue.

## References

- [1] Anderson, C., *CS545: Dimensionality Reduction*, <http://www.cs.colostate.edu/~anderson/cs545/Lectures/week9day2/week9day2.pdf>, 2009.
- [2] Anderson, C., *pca.R*, <http://www.cs.colostate.edu/~anderson/cs545/Lectures/week9day2/pca.R>, 2009.